Context-Aware Prefetching at the Storage Server

Gokul Soundararajan, Madalin Mihailescu[†], and Cristiana Amza
Department of Electrical and Computer Engineering
Department of Computer Science[†]
University of Toronto

Abstract

In many of today's applications, access to storage constitutes the major cost of processing a user request. Data prefetching has been used to alleviate the storage access latency. Under current prefetching techniques, the storage system prefetches a batch of blocks upon detecting an access pattern. However, the high level of concurrency in today's applications typically leads to interleaved block accesses, which makes detecting an access pattern a very challenging problem. Towards this, we propose and evaluate *QuickMine*, a novel, lightweight and minimally intrusive method for context-aware prefetching. Under *QuickMine*, we capture application contexts, such as a transaction or query, and leverage them for context-aware prediction and improved prefetching effectiveness in the storage cache.

We implement a prototype of our context-aware prefetching algorithm in a storage-area network (SAN) built using Network Block Device (NBD). Our prototype shows that context-aware prefetching clearly outperforms existing context-oblivious prefetching algorithms, resulting in factors of up to 2 improvements in application latency for two e-commerce workloads with repeatable access patterns, TPC-W and RUBiS.

1 Introduction

In many of today's applications, such as, e-commerce, on-line stores, file utilities, photo galleries, etc., access to storage constitutes *the* major cost of processing a user request. Therefore, recent research has focused on techniques for alleviating the storage access latency through storage caching [14, 23, 29] and prefetching techniques [24, 25, 35, 36, 37]. Many traditional storage prefetching algorithms implement sequential prefetching, where the storage server prefetches a batch of sequential blocks upon detecting a sequential access pattern. Recent algorithms, like *C-Miner** [24, 25], capture repeatable non-sequential access patterns as well. However, the storage system receives interleaved requests originating from many concurrent application streams. Thus, even if the logical I/O sequence of a particular application translates into physically sequential accesses, and/or the application pattern is highly repeatable, this pattern may be hard to recognize at the storage system. This is the case for concurrent execution of several applications sharing a network attached storage, e.g., as shown in Figure 1, and also for a single application with multiple threads exhibiting different access patterns, e.g., a database application running multiple queries, as also shown in the figure.

We investigate prefetching in storage systems and present a novel caching and prefetching technique that exploits logical application contexts to improve prefetching effectiveness. Our technique employs a context tracking mechanism, as well as a lightweight frequent sequence mining [38] technique. The context tracking mechanism captures application contexts in an *application independent manner*, with minimal instrumentation. These contexts are leveraged by the sequence mining technique for detecting block access patterns.

In our context tracking mechanism, we simply tag each application I/O block request with a context identifier corresponding to the higher level application context, e.g., a web interaction, database transaction, application thread, or database query, where the I/O request to the storage manager occurs. Such contexts are readily available in any application and can be easily captured. We then pass this context identifier along with each read block request, through the operating system, to the storage server. This allows the storage server to correlate the block accesses that it sees into frequent block sequences according to their higher level context. Based on the derived block correlations, the storage cache manager then issues block prefetches per context rather than globally.

At the storage server, correlating block accesses is performed by the frequent sequence mining component of

our approach. In particular, we design and implement a lightweight and dynamic frequent sequence mining technique, called *QuickMine*.

Just like state-of-the-art prefetching algorithms [24, 25], *QuickMine* detects sequential as well as non-sequential correlations using a history-based mechanism. *QuickMine's* key novelty lies in detecting and leveraging block correlations within logical application contexts. In addition, *QuickMine* generates and adapts block correlations *incrementally, on-the-fly*, through a lightweight mining algorithm. As we will show in our experimental evaluation, these novel features make *QuickMine* uniquely suitable for on-line pattern mining and prefetching by i) substantially reducing the footprint of the block correlations it generates, ii) improving the likelihood that the block correlations maintained will lead to accurate prefetches and iii) providing flexibility to dynamic changes in the application pattern, and concurrency degree.

We implement *QuickMine* in a lightweight storage cache prototype embedded into the Network Block Device (NBD) code. We also implement several alternative approaches for comparison with our scheme, including a baseline LRU cache replacement algorithm with no prefetching, and the following state-of-the-art context-oblivious prefetching schemes: two adaptive sequential prefetching schemes [10, 16] and the recently proposed *C-Miner** storage prefetching algorithm [24, 25].

In our experimental evaluation, we use three standard database applications: the TPC-W e-commerce benchmark [1], the RUBiS auctions benchmark and DBT-2 [40], a TPC-C-like benchmark [30]. The applications have a wide range of access patterns. TPC-W and RUBiS are read-intensive workloads with highly repeatable access patterns; they contain 80% and 85% read-only transactions, respectively, in their workload mix. In contrast, DBT-2 is a write-intensive application with rapidly changing access patterns; it contains only 4% read-only transactions in its workload mix. We instrument the MySQL/InnoDB database engine to track the contexts of interest. We found that changing the DBMS to incorporate the context into an I/O request was trivial; the DBMS already tracks various contexts, such as database thread, transaction or query, and these contexts are easily obtained for each I/O operation. We perform experiments using our storage cache deployed within NBD, running in a storage area network (SAN) environment.

Our experiments show that the context-aware *QuickMine* brings substantial latency reductions of up to factors of 2.0. The latency reductions correspond to reductions of miss rates in the storage cache of up to 60%. In contrast, the context oblivious schemes perform poorly for all benchmarks, with latencies comparable to, or worse than the baseline. This is due to either i) inaccurate

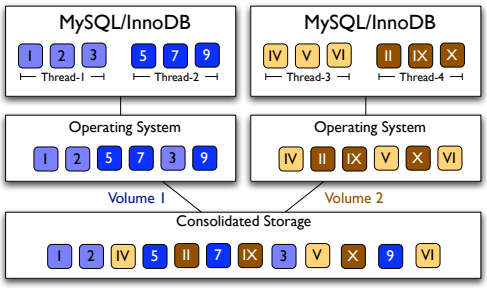


Figure 1: **Interleaved Accesses.** Accesses from concurrent processes/threads are interleaved at the storage server.

prefetches or ii) non-repeatable (false) block correlations at context boundaries, hence useless prefetch rules in the context-oblivious approaches. Our evaluation shows that *QuickMine* generates substantially more effective block correlation rules overall, in terms of both the number of prefetches triggered and the prefetching accuracy. We also show that *QuickMine* is capable of adjusting its correlation rules dynamically, without incurring undue overhead for rapidly changing patterns.

The rest of this paper is organized as follows. Sections 2 provides the necessary background and motivates our dynamic, context-aware algorithm. Section 3 introduces our *QuickMine* context-aware prefetching solution. Section 4 provides details of our implementation. Section 5 describes our experimental platform, methodology, other approaches in storage cache management that we evaluate in our experiments. Section 6 presents our experimental results. Section 7 discusses related work and Section 8 concludes the paper.

2 Background and Motivation

We focus on improving the cache hit rate at the storage cache in a SAN environment through prefetching. Our techniques are applicable to situations where the working set of storage clients, like a database system or file system, does not fit into the storage cache hierarchy i.e., into the combined caching capabilities of storage client and server. This situation is, and will remain common in the foreseeable future due to the following reasons.

First, while both client and storage server cache sizes are increasing, so do the memory demands of storage applications e.g., very large databases. Second, previous research has shown that access patterns at the storage server cache typically exhibit long reuse distances [7, 26], hence poor cache utilization. Third, due to server consolidation trends towards reducing the costs of management in large data centers, several applica-

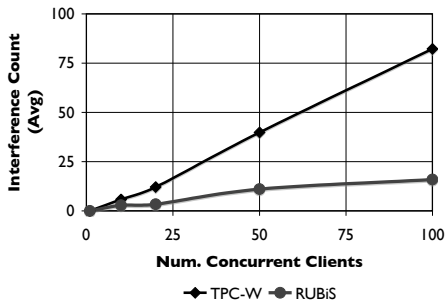


Figure 2: **Interference count.** Access interleavings of different threads with increasing number of concurrent clients.

tions typically run on a cluster with consolidated storage, or even on the same physical server. This creates application interference, hence potential capacity misses, and reduced prefetching effectiveness [19] in the shared storage-level cache.

In the following, we motivate our context-aware prefetching approach through a workload characterization for two e-commerce applications.

2.1 Workload Characterization

We analyze the access patterns of two popular e-commerce benchmarks: TPC-W, and RUBiS. We conduct experiments using a network storage server based on the NBD (network block device) protocol built into Linux. The NBD server manages access to physical storage and provides virtual block devices to applications. We experiment with each benchmark separately, varying the number of clients from 1 to 100. Each application runs on MySQL/InnoDB, which uses the NBD client to mount the virtual block device. We provide more details on our experimental setup in Section 5.

We characterize the access patterns of the two applications using the following metrics. The average/maximum *sequential run length* [39] is the average/maximum length of physically sequential block sequences for the duration of the application's execution. The average *context access length* is the average number of I/O requests issued by a logical unit of work in the application, i.e., by a transaction. Finally, the *interference count* [19] measures the interference in the storage cache, defined as the number of requests from other transactions that occur between consecutive requests from a given transaction stream.

In our experiments, we first compute the sequential run lengths when each thread is run in isolation i.e., on the *de-tangled* trace [39] for each of the two benchmarks. The lengths are: 1.05 (average) and 277 (maximum) for TPC-W and 1.14 (average) and 64 (maximum) for RUBiS. We then measure the sequential run lengths on the interleaved access traces, while progressively increasing the number of clients. We find that the sequential run length decreases significantly as we increase the concurrency degree. For example, with 10 concurrently running clients, the sequential run length is already affected: 1.04 (average) and 65 (maximum) for TPC-W, and 1.05 (average) and 64 (maximum) for RUBiS. With the common e-commerce workload of 100 clients, the average sequential run length asymptotically approaches 1.0 for both benchmarks. To further understand the drop in sequential run length, we plot the interference count for each benchmark when increasing the number of clients in Figure 2. The figure shows that the interference count increases steadily with the number of concurrent clients, from 5.87 for TPC-W and 2.91 for RUBiS at 10 clients, to 82.22 for TPC-W and 15.95 for RUBiS with 100 concurrently running clients.

To study the lower interference count in RUBiS compared to TPC-W, we compute the average *context access length* per transaction, in the two benchmarks. We find that the average context access length for RUBiS is 71 blocks, compared to 1223 blocks for TPC-W, 87% of the RUBiS transactions are short, reading only 1 to 10 blocks of data, compared to 79% in TPC-W, and several RUBiS transactions access a single block. Hence in TPC-W, longer logical access sequences result in higher interference opportunities, and for both benchmarks only a few logical sequences translate into physically sequential accesses.

The preliminary results presented in this section show that: i) opportunities for sequential prefetching in e-commerce workloads are low, and ii) random (non-repeatable) access interleavings can occur for the high levels of application concurrency common in e-commerce workloads. Accordingly, these results motivate the need for a prefetching scheme that i) exploits generic (non-sequential) access patterns in the application, and ii) is aware of application concurrency and capable of detecting access patterns per application context. For this purpose, in the following, we introduce the *QuickMine* algorithm that employs data mining principles to discover access correlations at runtime in a context-aware manner.

3 Context-aware Mining and Prefetching

In this section, we describe our approach to context-aware prefetching at the storage server. We first present an overview of our approach, and introduce the terminology we use. We then describe in more detail our technique for tracking application contexts, the *QuickMine* algorithm for discovering block correlations and discuss how we leverage them in our prefetching algorithm.

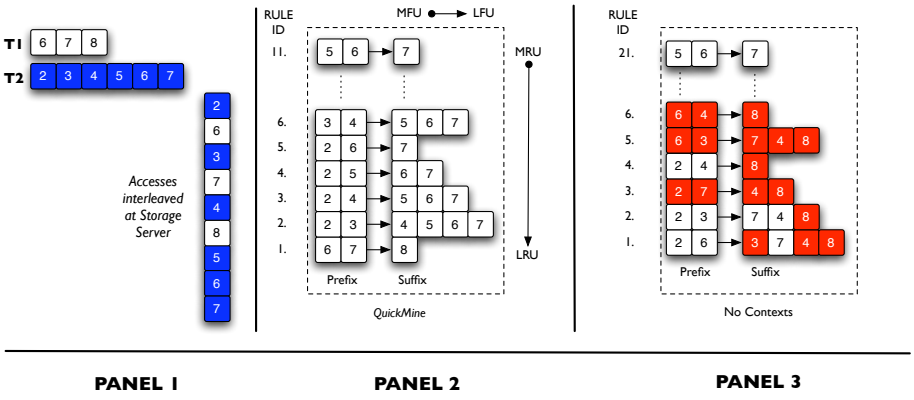


Figure 3: **Walk-through.** We compare *QuickMine* with a context-oblivious mining algorithm

3.1 Overview

We use application-level contexts to guide I/O block prefetching at the storage server. An application-level *context* is a logical unit of work corresponding to a specific level in the application's structural hierarchy e.g., a thread, a web interaction, a transaction, a query template, or a query instance.

We tag each I/O access with a context identifier provided by the application and pass these identifiers through the operating system to the storage server. This allows the storage server to group block accesses per application-level context. In the example in Figure 1, assuming that the context identifier of an I/O access is the thread identifier, the storage server is able to differentiate that blocks $\{1, 2, 3\}$ are accessed by *Thread-1* and blocks $\{5, 7, 9\}$ are accessed by *Thread-2*, from the interleaved access pattern.

Within each sequence of block accesses thus grouped by application context, the storage server applies our frequent sequence mining algorithm, called *QuickMine*. The *QuickMine* algorithm predicts a set of blocks to be accessed with high probability in the near future. The predictions are made based on mining past access patterns. Specifically, *QuickMine* derives *per-context* correlation rules for blocks that appear together frequently for a given context. Creation of new correlation rules and pruning useless old rules for an application and its various contexts occurs incrementally, on-the-fly, while the system performs its regular activities, including running other applications. The *QuickMine* algorithm is embedded in the storage server cache. The storage cache uses the derived rules to issue prefetches for blocks that are expected to occur within short order after a sequence of already seen blocks.

Terminology: For the purposes of our data mining algorithm, a *sequence* is a list of I/O reads issued by an application context ordered by the timestamp of their disk requests. A sequence database $D = \{S_1, S_2, \ldots, S_n\}$ is a set of sequences. The *support* of a sequence R in the database D is the number of sequences for which R is a *subsequence*. A subsequence is considered *frequent* if it occurs with a frequency higher than a predefined *min-support* threshold. Blocks in frequent subsequences are said to be *correlated*. Correlated blocks do not have to be consecutive. They should occur within a small distance, called a *gap* or *lookahead distance*, denoted G. The larger the *lookahead* distance, the more aggressive the algorithm is in determining correlations. For the purposes of our storage caching and prefetching algorithm, we distinguish between three types of cache accesses. A cache *hit* is an application demand access to a block currently in the storage cache. A *promote* is a block demand access for which a prefetch has been previously issued; the respective block may or may not have arrived at the cache at the time of the demand access. All other accesses are cache *misses*.

3.2 Context Tracking

Contexts are delineated with begin and end delimiters and can be nested. We use our context tracking for database applications and track information about three types of contexts: *application thread*, *database transaction* and *database query*. For database queries, we can track the context of each query instance, or of each query template i.e., the same query with different argument values. We reuse pre-existing begin and end markers, such as, connection establishment/connection teardown, `BEGIN` and `COMMIT`/`ROLLBACK` statements, and thread creation and destruction to identify the start and end of a context. For *application thread* contexts, we tag block accesses with the thread identifier of the database system thread running the interaction. We differentiate

database transaction contexts by tagging all block accesses between the `BEGIN` and `COMMIT`/`ABORT` with the transaction identifier. A *database query* context simply associates each block access with the query or query template identifier. We studied the feasibility of our tagging approach in three open-source database engines: MySQL, PostgreSQL, and Apache Derby, and we found the necessary changes to be trivial in all these existing code bases. The implementation and results presented in this paper are based on minimal instrumentation of the MySQL/InnoDB database server to track transaction and query template contexts.

While defining meaningful contexts is intuitive, defining the right context granularity for optimizing the prefetching algorithm may be non-trivial. There is a trade-off between using coarse-grained contexts and fine-grained contexts. Fine-grained contexts provide greater prediction accuracy, but may limit prefetching aggressiveness because they contain fewer accesses. Coarse-grained contexts, on the other hand, provide more prefetching opportunities, but lower accuracy due to more variability in access patterns, e.g., due to control flow within a transaction or thread.

3.3 QuickMine

QuickMine derives block correlation rules for each application context as follows. Given a sequence of already accessed blocks $\{a_1, a_2, \ldots, a_k\}$, and a lookahead parameter G, *QuickMine* derives block correlation rules of the form $\{a_i \& a_j \rightarrow a_k\}$ for all i, j and k, where $\{a_i, a_j, a_k\}$ is a subsequence and $i < j < k$, $(j-i) < G$ and $(k-i) < G$.

For each rule of the form $\{a \& b \rightarrow c\}$, $\{a \& b\}$ is called a sequence prefix, and represents two blocks already seen on a *hot path* through the data i.e., a path taken repeatedly during the execution of the corresponding application context. For the same rule, $\{c\}$ is one of the block accesses about to follow the *prefix* with high probability and is called a sequence *suffix*. Typically, the same *prefix* has several different suffixes depending on the lookahead distance G and on the variability of the access patterns within the given context, e.g., due to control flow. For each prefix, we maintain a list of possible suffixes, up to a cut-off *max-suffix* number. In addition, with each suffix, we maintain a *frequency* counter to track the number of times the suffix was accessed i.e., the *support* for that block. The list of suffixes is maintained in order of their respective frequencies to help predict the most probable block(s) to be accessed next. For example, assume that *QuickMine* has seen access patterns $\{(a_1, a_2, a_3, a_4), (a_2, a_3, a_4), (a_2, a_3, a_5)\}$ in the past for a given context. *QuickMine* creates rules $\{a_2 \& a_3 \rightarrow a_4\}$ and $\{a_2 \& a_3 \rightarrow a_5\}$ for this context. Further assume that the current access sequence matches the rule prefix $\{a_2 \& a_3\}$. *QuickMine* predicts that the next block to be accessed will be suffix $\{a_4\}$ or $\{a_5\}$ in this order of probability because $\{a_2, a_3, a_4\}$ occurred twice while $\{a_2, a_3, a_5\}$ occurred only once.

We track the blocks accessed within each context and create/update block correlations for that context whenever a context ends. We maintain all block correlation rules in a *rule cache*, which allows pruning old rules through simple cache replacement policies. The cache replacement policies act in two dimensions: i) for rule prefixes and ii) within the suffixes of each prefix. We keep the most recent *max-prefix* prefixes in the cache. For each prefix, we keep *max-suffix* most probable suffixes. Hence, the cache replacement policies are LRU (Least-Recently-Used) for rule prefixes and LFU (Least-Frequently-Used) for suffixes. Intuitively, these policies match the goals of our mining algorithm well. Since access paths change over time as the underlying data changes, we need to remember recent hot paths and forget past paths. Furthermore, as mentioned before, we need to remember only the most probable suffixes for each prefix. To prevent quick evictions, newly added suffixes are given a grace period.

Example: We show an example of *QuickMine* in Figure 3. We contrast context-aware mining with context-oblivious mining. On the left hand side, we show an example of an interleaved access pattern that is seen at the storage server when two transactions (denoted T_1 and T_2) are running concurrently. Panel 2 shows the result of *QuickMine*. Panel 3 contrasts *QuickMine* with a context-oblivious mining algorithm. To aid the reader, we do not prune/evict items from the rule cache. In Panel 2, we obtain the list of block accesses after T_1 and T_2 finish execution. T_1 accesses $\{6, 7, 8\}$ so we obtain the correlation $\{6\&7 \rightarrow 8\}$ (rule 1 in Panel 2). T_2 accesses $\{2, 3, 4, 5, 6, 7\}$ leading to more correlations. We set the lookahead parameter, $G = 5$, so we look ahead by at most 5 entries when discovering correlations. We discover correlations $\{(6\&7 \rightarrow 8), (2\&3 \rightarrow 4), (2\&3 \rightarrow 5), \ldots, (5\&6 \rightarrow 7)\}$. Finally, we show a snapshot of context-oblivious mining, with false correlations highlighted, in Panel 3. At the end of transaction T_2, the access pattern $\{2, 6, 3, 7, 4, 8, 5, 6, 7\}$ is extracted and the mined correlations are $\{(2\&6 \rightarrow 3), (2\&6 \rightarrow 7), (2\&6 \rightarrow 4), (2\&6 \rightarrow 8), \ldots, (5\&6 \rightarrow 7)\}$. With context-oblivious mining, false correlations are generated, e.g., $\{(2\&6 \rightarrow 3), (2\&6 \rightarrow 4), (2\&6 \rightarrow 8)\}$ are incorrect. False correlations will be eventually pruned since they occur infrequently, hence have low support, but the pruning process may evict some true correlations as well.

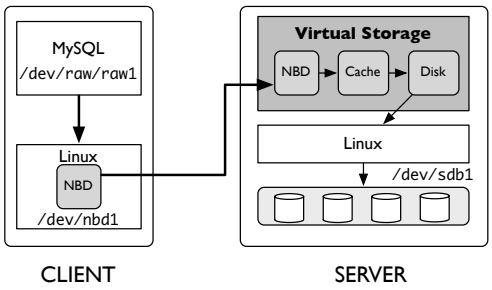


Figure 4: **Storage Architecture**: We show one client connected to a storage server using NBD.

3.4 Prefetching Algorithm

The storage cache uses the block correlation rules to issue block prefetches for predicted future *read* accesses. Block prefetches are issued upon a *read* block miss. We use the last two accesses of the corresponding context to search the rule cache for the prefix just seen in the I/O block sequence. We determine the set of possible blocks to be accessed next as the set of suffixes stored for the corresponding prefix. We prefetch blocks that are not currently in the cache starting with the highest support block up to either the maximum suffixes stored or the maximum prefetching degree.

The number of block prefetches issued upon a block miss called *prefetch aggressiveness*, is a configurable parameter. We set the prefetching aggressiveness to the same value as *max-suffix* for all contexts. However, in heavy load situations, we limit the prefetching aggressiveness to prevent saturating the storage bandwidth. Specifically, we leverage our context information to selectively throttle or disable prefetching for contexts where prefetching is not beneficial. Prefetching benefit is determined per context, as the ratio of prefetches issued versus prefetched blocks used by the application.

The prefetched blocks brought in by one application context may be, however, consumed by a different application context due to data affinity. We call such contexts *symbiotic contexts*. We determine *symbiotic contexts* sets by assigning context identifiers tracking the issuing and using contexts for each prefetched cache block. We then monitor the prefetching benefit at the level of *symbiotic context* sets rather than per individual context. We disable prefetching for contexts (or symbiotic context sets) performing poorly.

4 Implementation Details

We implement our algorithms in our Linux-based virtual storage prototype, which can be deployed over commodity storage firmware. The architecture of our prototype is shown in Figure 4. MySQL communicates to the virtual storage device through standard Linux system calls and drivers, either iSCSI or NBD (network block device), as shown in the figure. Our storage cache is located on the same physical node as the storage controller, which in our case does not have a cache of its own. The storage cache communicates to a *disk module* which maps virtual disk accesses to physical disk accesses. We modified existing *client* and *server* NBD protocol processing modules for the storage client and server, respectively, in order to incorporate context awareness on the I/O communication path.

In the following, we first describe the interfaces and communication between the core modules, then describe the role of each module in more detail.

4.1 Interfaces and Communication

Storage clients, such as MySQL, use NBD for reading and writing logical blocks. For example, as shown in Figure 4, MySQL mounts the NBD device (`/dev/nbd1`) on `/dev/raw/raw1`. The Linux virtual disk driver uses the NBD protocol to communicate with the storage server. In NBD, an I/O request from the client takes the form `<type,offset,length>` where `type` is a *read* or *write*. The I/O request is passed by the OS to the NBD kernel driver on the client, which transfers the request over the network to the NBD protocol module running on the storage server.

4.2 Modules

Each module consists of several threads processing requests. The modules are interconnected through in-memory bounded buffers. The modular design allows us to build many storage configurations by simply connecting different modules together.

Disk module: The disk module sits at the lowest level of the module hierarchy. It provides the interface with the underlying physical disk by translating application I/O requests to the virtual disk into `pread()`/`pwrite()` system calls, reading/writing the underlying physical data. We disable the operating system buffer cache by using direct I/O i.e., the I/O `O_DIRECT` flag in Linux.

Cache module: The cache module supports context-aware caching and prefetching. We developed a portable caching library providing a simple hashtable-like interface modelled after *Berkeley DB*. If the requested block is found in the cache, the access is a cache *hit* and we return the data. Otherwise, the access is a cache *miss*, we fetch the block from the next level in the storage hierarchy, store it in the cache, then return the data. When prefetching is enabled, the cache is partitioned into two

areas: a main cache ($\mathbb{MC}$) and a prefetch cache ($\mathbb{PFC}$). The $\mathbb{PFC}$ contains blocks that were fetched from disk by the prefetching algorithm. The $\mathbb{MC}$ contains blocks that were requested by application threads. If an application thread requests a block for which a prefetch has been issued, we classify the access as a *promote*. A block *promote* may imply either waiting for the block to arrive from disk, or simply moving the block from $\mathbb{PFC}$ to $\mathbb{MC}$ if the block is already in the cache.

We use *Berkeley DB* to implement the rule cache and store the mined correlations. The caching keys are rule *prefixes* and the cached data are the rule *suffixes*. The suffixes are maintained using the LFU replacement algorithm and the prefixes are maintained using LRU. The LRU policy is implemented using *timeouts*, where we periodically purge old entries. We configure *Berkeley DB*'s environment to use a memory pool of 4MB.

NBD Protocol module: We modify the original NBD protocol module on the server side, used in Linux for virtual disk access, to convert the NBD packets into our own internal protocol packets, i.e., into calls to our server cache module.

4.3 Code Changes

To allow context awareness, we make minor changes to MySQL, the Linux kernel, and the NBD protocol.

Linux: The changes required in the kernel are straightforward and minimal. In the simplest case, we need to pass a context identifier on I/O calls as a separate argument into the kernel. In order to allow more flexibility in our implementation, and enhancements such as per-context tracking of prefetch effectiveness, we pass a handle to a context structure, which contains the transaction identifier, and query template identifier.

We add three new system calls, `ctx_pread()`, `ctx_pwrite()` and `ctx_action()`. `ctx_action()` allows us to inform the storage server of context begin/end delimiters. Each system call takes a `struct context *` as a parameter representing the context of the I/O call. This context handle is passed along the kernel until it reaches the lowest level where the kernel contacts the block storage device. Specifically, we add a field `context` to `struct request`, which allows us to pass the context information through the I/O subsystem with no additional code changes. Once the I/O request reaches the NBD driver code, we copy the context information into the NBD request packet and pass the information to the storage server.

NBD Protocol: We simply piggyback the context information on the NBD packet. In addition, we add two new messages to the NBD protocol, for the corresponding system call `ctx_action()`, to signify the beginning of a context (`CTX_BEG`) and the end of a context (`CTX_END`).

MySQL: A `THD` object is allocated for each connection made to MySQL. The `THD` object contains all contextual information that we communicate to the storage server. For example, `THD.query` contains the query currently being executed by the thread. We generate the query template identifier using the query string. In addition, we call our `ctx_action()` as appropriate, e.g., at transaction *begin/end* and at connection setup/tear-down to inform the storage server of the start/end of a context.

5 Evaluation

In this section, we describe several prefetching algorithms we use for comparison with *QuickMine* and evaluate the performance using three industry-standard benchmarks: TPC-W, RUBiS, and DBT-2.

5.1 Prefetching Algorithms used for Comparison

In this section, we describe several prefetching algorithms that we use for comparison with *QuickMine*. These algorithms are categorized into sequential prefetching schemes (RUN and SEQ) and history based prefetching schemes (*C-Miner**). The RUN and *C-Miner** algorithms share some of the features of *QuickMine*, specifically, some form of concurrency awareness (RUN) and history-based access pattern detection and prefetching (*C-Miner**).

Adaptive Sequential Prefetching (SEQ): We implement an adaptive prefetching scheme following the sequence-based read-ahead algorithm implemented in Linux [10]. Sequence based prefetching is activated when the algorithm detects a sequence (S) of accesses to K contiguous blocks. The algorithm uses two windows: a *current window* and a *read-ahead window*. Prefetches are issued for blocks contained in both windows. In addition, the number of prefetched blocks used within each window is monitored i.e., the block hits in the current window and the read ahead window, $S.curHit$ and $S.reaHit$, respectively. When a block is accessed in the *read-ahead window*, the *current window* is set to the *read-ahead window* and a new read-ahead window of size is $2 * S.curHit$ is created. To limit the prefetching aggressiveness, the size of the read-ahead window is limited by 128KB as suggested by the authors [10].

Run-Based Prefetching (RUN): Hsu et al. [16] show that many workloads, particularly database workloads, do not exhibit strict sequential behavior, mainly due to high application concurrency. To capture sequentiality in a multi-threaded environment, Hsu et al. introduce *run-based prefetching* (RUN) [17]. In *run-based prefetch-*

ing, prefetching is initiated when a sequential run is detected. A reference r to block b is considered to be part of a sequential run R if b lies within $-extent_{backward}$ and $+extent_{forward}$ of the largest block accessed in R, denoted by $R.maxBlock$. This modified definition of sequentiality thus accommodates small jumps and small reverses within an access trace. Once the size of the run R exceeds a sequentiality threshold (32KB), prefetching is initiated for 16 blocks from $R.maxBlock + 1$ to $R.maxBlock + 16$.

History-based Prefetching: There are several history based prefetching algorithms proposed in recent work [17, 13, 20, 26]. *C-Miner** is a static mining algorithm that extracts frequent block subsequences by mining the entire sequence database [25].

C-Miner* builds on a frequent subsequence mining algorithm, *CloSpan* [42]. It differs from *QuickMine* by mining block correlations off-line, on a long sequence of I/O block accesses. First, *C-Miner** breaks the long sequence trace into smaller sequences and creates a sequence database. From these sequences, as in *QuickMine*, the algorithm considers frequent sequences of blocks that occur within a *gap* window. Given the sequence databases and using the *gap* and *min_support* parameters, the algorithm extracts frequent *closed* sequences, i.e., subsequences whose support value is different from that of its super-sequences. For example, if $\{a_1, a_2, a_3, a_4\}$ is a frequent subsequence with support value of 5 and $\{a_1, a_2, a_3\}$ is a subsequence with support value of 5 then, only $\{a_1, a_2, a_3, a_4\}$ will be used in the final result. On the other hand, if $\{a_1, a_2, a_3\}$ has a support of 6 then, both sequences are recorded. For each *closed* frequent sequence e.g., $\{a_1, a_2, a_3, a_4\}$, *C-Miner** generates association rules of the form $\{(a_1 \rightarrow a_2), (a_1 \& a_2 \rightarrow a_3), \ldots, (a_3 \rightarrow a_4)\}$.

As an optimization, *C-Miner** uses the frequency of a rule suffix in the rule set to prune predictions of low probability through a parameter called *min_confidence*. For example, if the mined trace contains 80 sequences with $\{a_2 \& a_3 \rightarrow a_4\}$ and 20 sequences with $\{a_2 \& a_3 \rightarrow a_5\}$, then $\{a_2 \& a_3 \rightarrow a_5\}$ has a (relatively low) confidence of 20% and might be pruned depending on the *min_confidence* threshold. In our experiments, we use $max\_gap = 10$, $min\_support = 1$, and $min\_confidence = 10\%$ for *C-Miner**.

5.2 Benchmarks

TPC-W^{10}: The TPC-W benchmark from the Transaction Processing Council [1] is a transactional web benchmark designed for evaluating e-commerce systems. Several web interactions are used to simulate the activity of a retail store. The database size is determined by the number of items in the inventory and the size of the customer population. We use 100K items and 2.8 million customers which results in a database of about 4 GB. We use the *shopping* workload that consists of 20% writes. To fully stress our architecture, we create TPC-W^{10} by running 10 TPC-W instances in parallel creating a database of 40 GB.

RUBiS^{10}: We use the RUBiS Auction Benchmark to simulate a bidding workload similar to e-Bay. The benchmark implements the core functionality of an auction site: selling, browsing, and bidding. We do not implement complementary services like instant messaging, or newsgroups. We distinguish between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are only allowed to browse. Buyer and seller sessions require registration. In addition to the functionality provided during the visitor sessions, during a buyer session, users can bid on items and consult a summary of their current bid, rating, and comments left by other users. We are using the default RUBiS bidding workload containing 15% writes, considered the most representative of an auction site workload according to an earlier study of e-Bay workloads [34]. We create a scaled workload, RUBiS^{10} by running 10 RUBiS instances in parallel.

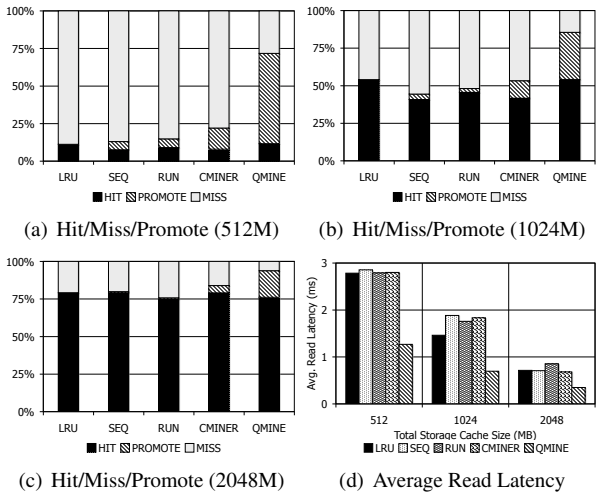
DBT-2: DBT-2 is an OLTP workload derived from the TPC-C benchmark [30, 40]. It simulates a wholesale parts supplier that operates using a number of warehouse and sales districts. Each warehouse has 10 sales districts and each district serves 3000 customers. The workload involves transactions from a number of terminal operators centered around an order entry environment. There are 5 main transactions for: (1) entering orders (*New Order*), (2) delivering orders (*Delivery*), (3) recording payments (*Payment*), (4) checking the status of the orders (*Order Status*), and (5) monitoring the level of stock at the warehouses (*Stock Level*). Of the 5 transactions, only *Stock Level* is read only, but constitutes only 4% of the workload mix. We scale DBT-2 by using 256 warehouses, which gives a database footprint of 60GB.

5.3 Evaluation Methodology

We run our Web based applications on a dynamic content infrastructure consisting of the Apache web server, the PHP application server and the MySQL/InnoDB (version 5.0.24) database storage engine. For the database applications, we use the test harness provided by each benchmark while hosting the database on MySQL. We run the Apache Web server and MySQL on Dell PowerEdge SC1450 with dual Intel Xeon processors running at 3.0 Ghz with 2GB of memory. MySQL connects to the raw device hosted by the NBD server. We run the NBD server on a Dell PowerEdge PE1950 with 8 Intel Xeon processors running at 2.8 Ghz with 3GB of memory. To maximize IO bandwidth, we use RAID 0 on 15

10K RPM 250GB hard disks. We install Ubuntu 6.06 on both the client and server machines with Linux kernel version 2.6.18-smp.

We configure our caching library to use 16KB block size to match the MySQL/InnoDB block size. We use 100 clients for TPC-W[10] and RUBiS[10]. For DBT-2, we use 256 warehouses. We run each experiment for two hours. We train *C-Miner** on a trace collected from the first hour of the experiment. We measure statistics for both *C-Miner** and *QuickMine* during the second hour of the experiment. We use a lookahead value of 10 for *C-Miner**, which is the best value found experimentally for the given applications and number of clients used in the experiments. *QuickMine* is less sensitive to the lookahead value, and any lookahead value between 5 and 10 gives similar results. We use a lookahead value of 5 for *QuickMine*.



(a) Hit/Miss/Promote (512M) (b) Hit/Miss/Promote (1024M)

(c) Hit/Miss/Promote (2048M) (d) Average Read Latency

Figure 5: **TPC-W.** Prefetching benefit in terms of miss rate and average read latency.

6 Results

We evaluate the performance of the following schemes: a baseline caching scheme with no prefetching (denoted as LRU), adaptive sequential prefetching (SEQ), run-based prefetching (RUN), *C-Miner** (CMINER), and *QuickMine* (QMINE). In Section 6.1, we present the overall performance of those schemes, whereas in Section 6.2, we provide detailed analysis to further understand the achieved overall performance of each scheme.

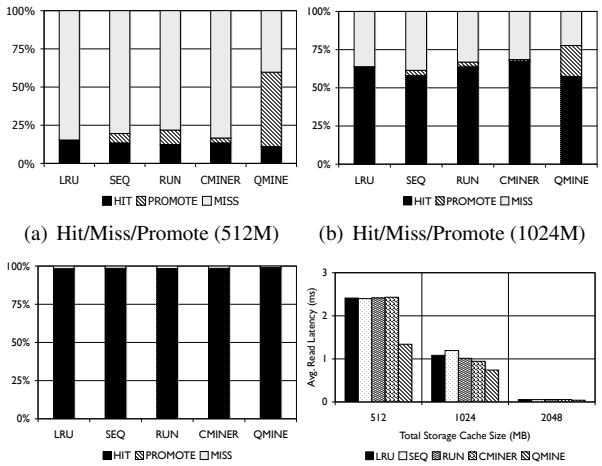
6.1 Overall Performance

In this section, we measure the storage cache hit rates, miss rates, promote rates and the average read latency for each of the prefetching schemes by running our three benchmarks in several cache configurations. For all experiments, the MySQL/InnoDB buffer pool is set to 1024MB and we partition the storage cache such that the prefetching area is a fixed 4% of the total storage cache size. For TPC-W[10] and RUBiS[10], we use 100 clients and we vary the storage cache size, showing results for 512MB, 1024MB, and 2048MB storage cache for each benchmark. For DBT-2, we show results only for the 1024MB storage cache, since results for other cache sizes are similar. In *QuickMine*, we use query-template contexts for TPC-W and RUBiS and transaction contexts for DBT-2. However, the results vary only slightly with the context granularity for our three benchmarks.

TPC-W: Figures 5(a)-5(c) show the hit rates (black), miss rates (white), and promote rates (shaded) for all prefetching schemes with TPC-W[10]. For a 512MB storage cache, as shown in Fig. 5(a), the baseline (LRU) miss rate is 89%. The sequential prefetching schemes reduce the miss rate by 5% on average. *C-Miner** reduces the miss rate by 15%, while *QuickMine* reduces the miss rate by 60%. The benefit of sequential prefetching schemes is low due to lack of sequentiality in the workload. With larger cache sizes, the baseline miss rates are reduced to 45% for the 1024MB cache (Fig. 5(b)) and to 20% for the 2048MB cache (Fig. 5(c)). With lower miss rates, there are lower opportunities for prefetching. In spite of this, *QuickMine* still provides a 30% and 17% reduction in miss rates for the 1024MB and 2048MB cache sizes, respectively.

For *QuickMine*, the cache miss reductions translate into substantial read latency reductions as well, as shown in Figure 5(d). In their turn, these translate into decreases in overall storage access latency by factors ranging from 2.0 for the 512MB to 1.22 for the 2048MB caches, respectively. This is in contrast to the other prefetching algorithms, where the average read latency is comparable to the baseline average latency for all storage cache sizes.

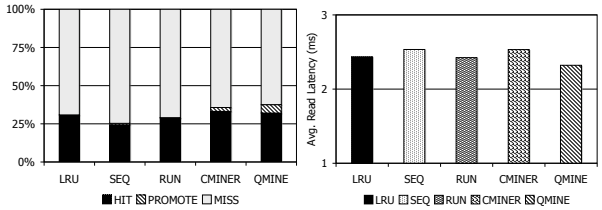
RUBiS: Context-aware prefetching benefits RUBiS as well, as shown in Figure 6. The baseline (LRU) cache miss rates are 85%, 36% and 2% for the 512MB, 1024MB, and 2048MB cache sizes, respectively. For a 512MB cache, as shown in Figure 6(a), the SEQ and RUN schemes reduce the miss rate by 6% and 9%, respectively. *C-Miner** reduces the miss rate by only 3% while *QuickMine* reduces the miss rate by 48%. In RUBiS, the queries access a spatially-local cluster of blocks. Thus, triggering prefetching for a weakly sequential access pattern, as in RUN, results in more promotes than for SEQ. *C-Miner** performs poorly for RUBiS because many RUBiS contexts are short. This results in many false correlations across context boundaries in *C-Miner**.



(a) Hit/Miss/Promote (512M) (b) Hit/Miss/Promote (1024M)

(c) Hit/Miss/Promote (2048M) (d) Average Read Latency

Figure 6: **RUBiS**. Prefetching benefit in terms of miss rate and average read latency.



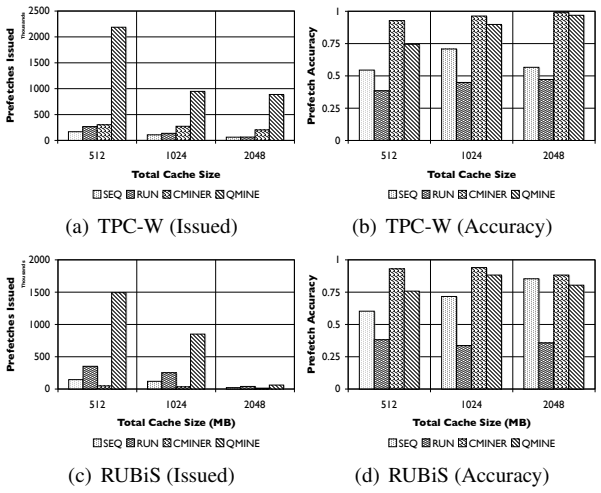
(a) Hit/Miss/Promote (b) Average Read Latency

Figure 7: **DBT-2.** Prefetching benefit in terms of miss rate and average read latency for 1024MB storage cache.

Hence, only a few prefetch rule prefixes derived during training can be later matched on-line, while many rule suffixes are pruned due to low confidence. *QuickMine* overcomes the limitations of *C-Miner** by tracking correlations per context.

The performance of the prefetching algorithms is reflected in the average read latency as well. As shown in Figure 6(d), the sequential prefetching schemes (SEQ and RUN) reduce the average read latency by up to 10% compared to LRU. The reductions in miss rate using *QuickMine* translate to reductions in read latencies of 45% (512MB) and 22% (1024MB) compared to LRU, corresponding to an overall storage access latency reduction by a factor of 1.63 for the 512MB cache and 1.3 for the 1024MB cache.

DBT-2: Prefetching is difficult for DBT-2, since the workload mix for this benchmark contains a high fraction of writes; furthermore, some of its transactions issue very few I/Os. The I/O accesses are not sequential. As Figure 7 shows, the lack of sequentiality causes the sequential prefetching algorithms to perform poorly. Sequential prefetching schemes decrease the miss rate by less than 1%. *C-Miner** and *QuickMine* perform slightly better. *C-Miner** lowers the miss rate by 2% and *QuickMine* reduces the miss rate by 6%. However, the high I/O footprint of this benchmark causes disk congestion, hence increases the promote latency. Overall, the average read latency increases by 2% for the sequential prefetching schemes and *C-Miner**, while the read latency is reduced by 3% for *QuickMine*.



(a) TPC-W (Issued) (b) TPC-W (Accuracy)

(c) RUBiS (Issued) (d) RUBiS (Accuracy)

Figure 8: **RUBiS**^{10}**/TPC-W**^{10}. We show the number of prefetches issued (in thousands) and the accuracy of the prefetches.

6.2 Detailed Analysis

In this section, we evaluate the prefetching *effectiveness* of the different schemes by measuring the number of prefetches issued and their *accuracy*, i.e., the percentage of prefetched blocks consumed by the application. Both metrics are important since if only a few prefetches are issued, their overall impact is low, even at high accuracy for these prefetches. We also compare the two history-based schemes, *QuickMine* and *C-Miner**, in more detail. Specifically, we show the benefits of context awareness and the benefit of incremental mining, versus static mining.

Detailed Comparison of Prefetching Effectiveness: In Figure 8, we show the number of prefetches issued, and their corresponding accuracy for all prefetching algorithms. For TPC-W^{10} with a 512MB cache, shown in Figure 8(a), *QuickMine* issues 2M prefetches, while *C-Miner**, SEQ, and RUN issue less than 500K prefetches. The RUN scheme is the least accurate ($< 50\%$) since many prefetches are spuriously triggered. The SEQ scheme exhibits a better prefetch accuracy of between 50% and 75% for the three cache sizes. Both *QuickMine*

and *C-Miner** achieve greater than 75% accuracy. While *C-Miner** has slightly higher accuracy than *QuickMine* for the rules issued, this accuracy corresponds to substantially fewer rules than *QuickMine*. This is because, many of the *C-Miner** correlation rules correspond to false correlations at the context switch boundary, hence are not triggered at runtime. As a positive side-effect, the higher number of issued prefetches in *QuickMine* allows the disk scheduler to re-order the requests for optimizing seeks, thus reducing the average prefetch latency. As a result, average promote latencies are significantly lower in *QuickMine* compared to *C-Miner**, specifically, 600μs versus 2400μs for the 512MB cache. For comparison, a cache hit takes 7μs on average and a cache miss takes 3200μs on average, for all algorithms.

For RUBiS[10] with a 512MB cache, shown in Figure 8(c), *QuickMine* issues 1.5M prefetches, which is ten times more than *C-Miner** and SEQ. In the RUN scheme, the spatial locality of RUBiS causes more prefetches (250K) to be issued compared to SEQ, but only 38% of these are accurate, as shown in Figure 8(d). As before, while *C-Miner** is highly accurate (92%) for the prefetches it issues, substantially fewer correlation rules are matched at runtime compared to *QuickMine*, due to false correlations. With larger cache sizes, there is less opportunity for prefetching, because there are fewer storage cache misses, but at all cache sizes *QuickMine* issues more prefetch requests than other prefetching schemes. Similar as for TPC-W, the higher number of prefetches results in a lower promote latency for *QuickMine* compared to *C-Miner** i.e., 150μs versus 650μs for RUBiS in the 512MB cache configuration.

Benefit of Context Awareness: We compare the total number of correlation rules generated by frequent sequence mining, with and without context awareness. In our evaluation, we isolate the impact of context awareness from other algorithm artifacts, by running *C-Miner** without rule pruning, on its original access traces of RUBiS and DBT-2, and the *de-tangled* access traces of the same. In the *de-tangled* trace, the accesses are separated by thread identifier, then concatenated. We notice an order of magnitude reduction in the number of rules generated by *C-Miner**. Specifically, on the original traces, *C-Miner** without rule pruning generates 8M rules and 36M rules for RUBiS and DBT-2, respectively. Using the *de-tangled* trace, *C-Miner** without rule pruning generates 800K rules for RUBiS and 2.8M rules for DBT-2. These experiments show that context awareness reduces the number of rules generated, because it avoids generating false block correlation rules for the blocks at the context switch boundaries.

Another benefit of context-awareness is that it makes parameter settings in *QuickMine* insensitive to the concurrency degree. For example, the value of the lookahead/gap parameter correlates with the concurrency degree in context oblivious approaches, such as *C-Miner**, i.e., needs to be higher for a higher concurrency degree. In contrast, *QuickMine*'s parameters, including the value of the lookahead parameter, are independent of the concurrency degree; they mainly control the *prefetch aggressiveness*. While the ideal *prefetch aggressiveness* does depend on the application and environment, *QuickMine* has built-in dynamic tuning mechanisms that make it robust to overly aggressive parameter settings.

The ability to dynamically tune the prefetching decisions at run-time is yet another benefit of context-awareness. For example, in TPC-W, *QuickMine* automatically detects that the *BestSeller* and the symbiotic pair of *Search* and *NewProducts* benefit the most from prefetching, while other queries in TPC-W do not. Similarly, it detects that only the *StockLevel* transaction in DBT-2 benefits from prefetching. Tracking the prefetching benefit per context allows *QuickMine* to selectively disable or throttle prefetching for low performing query templates thus avoiding unnecessary disk congestion caused by useless prefetches. In particular, this feature allows *QuickMine* to provide a small benefit for DBT-2, while *C-Miner** degrades the application performance.

Benefit of Incremental Mining: We show the benefit of dynamically and incrementally updating correlation rules through the use of the LRU based rule cache as in *QuickMine* versus statically mining the entire sequence database to generate association rules as in *C-Miner** [24, 25]. Figure 9 shows the number of promotes, cumulatively, over the duration of the experiment for *C-Miner**, *C-Miner** with periodic retraining (denoted as *C-Miner*$^+$) and *QuickMine*. For these experiments, we train *C-Miner** on the *de-tangled* trace to eliminate the effects of interleaved I/O, hence make it comparable with *QuickMine*. As Figure 9 shows, the change in the access patterns limits the prefetching effectiveness of *C-Miner*$*$, since many of its mined correlations become obsolete quickly. Thus, no new promotes are issued after the first 10 minutes of the experiment. In *C-Miner*$^+$, where we retrain *C-Miner** at the 10 minute mark, and at the 20 minute mark of the experiment, the effectiveness of prefetching improves. However, *C-Miner*$^+$ still lags behind *QuickMine*, which adjusts its rules continuously, on-the-fly. By dynamically aging old correlation rules and incrementally learning new correlations, *QuickMine* maintains a steady performance throughout the experiment. The dynamic nature of *QuickMine* allows it to automatically and gracefully adapt to the changes in the I/O access pattern, hence eliminating the need for explicit retraining decisions.

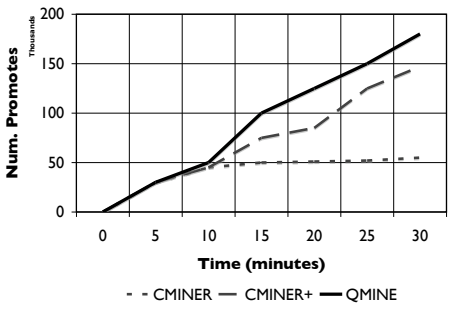


Figure 9: **Incremental Mining.** When access patterns change, the performance of static mining deteriorates over time.

7 Related Work

This section discusses related techniques for improving caching efficiency at the storage server, including: i) collaborative approaches like our own, which pass explicit hints between client and storage caches, or require more extensive code restructuring and reorganization, ii) gray-box approaches, which infer application patterns at the storage based on application semantics known *a priori*, and iii) black box approaches, which infer application patterns at the storage server in the absence of any semantic information.

Explicitly Collaborative Approaches. Several approaches pass explicit hints from the client cache to the storage cache [6, 12, 23, 28]. These hints can indicate, for example, the reason behind a write block request to storage [23], explicit demotions of blocks from the storage client to the server cache [41], or the relative importance of requested blocks [7]. These techniques modify the interface between the storage client and server, by requiring that an additional identifier (representing the hint) be passed to the storage server. Thus, similar to QuickMine, these techniques improve storage cache efficiency through explicit context information. However, as opposed to our work, inserting the appropriate hints needs thorough understanding of the application internals. For example, Li et al. [23] require the understanding of database system internals to distinguish the context surrounding each block I/O request. In contrast, we use readily available information within the application about preexisting contexts.

In general, collaboration between the application and storage server has been extensively studied in the context of database systems, e.g., by providing the DBMS with more knowledge of the underlying storage characteristics [32], by providing application semantic knowledge to storage servers i.e., by mapping database relations to objects [33], or by offloading some tasks to the storage server [31]. Other recent approaches in this area [4, 11, 15] take advantage of context information available to the database query optimizer [11, 15], or add new middleware components for exploiting explicit query dependencies e.g., by SQL code re-writing to group related queries together [4]. Unlike our technique, these explicitly collaborative approaches require substantial restructuring of the database system, code reorganization in the application, or modifications to the software stack in order to effectively leverage semantic contexts. In contrast, we show that substantial performance advantage can be obtained with minimal changes to existing software components and interfaces.

Gray-box Approaches. Transparent techniques for storage cache optimization leverage I/O meta-data, or application semantics known *a priori*. Meta-data based approaches include using file-system meta-data, i.e., distinguishing between i-node and data blocks explicitly, or using filenames [5, 20, 22, 35, 43], or indirectly by probing at the storage client [2, 3, 37]. Alternative techniques based on application semantics leverage the *program backtrace* [12], user information [43], or specific characteristics, such as in-memory addresses of I/O requests [8, 18] to classify or infer application patterns.

Sivathanu et al. [36] use minimally intrusive instrumentation to the DBMS and log snooping to record a number of statistics, such that the storage system can provide cache exclusiveness and reliability for the database system running on top. However, this technique is DBMS-specific, the storage server needs to be aware of the characteristics of the particular database system.

Graph-based prefetching techniques based on discovering correlations among files in filesystems [13, 21] also fall into this category, although they are not scalable to the number of blocks typical in storage systems [24].

In contrast to the above approaches, our work is generally applicable to any type of storage client and application; any database and file-based application can benefit from QuickMine. We can use arbitrary contexts, not necessarily tied to the accesses of a particular user [43], known application code paths [12], or certain types of meta-data accesses, which may be client or application specific.

Black Box Approaches. Our work is also related to caching/prefetching techniques that treat the storage as a black box, and use fully transparent techniques to infer access patterns [10, 17, 26, 27]. These techniques use sophisticated sequence detection algorithms for detecting application I/O patterns, in spite of access interleaving due to concurrency at the storage server. In this paper, we have implemented and compared against two such techniques, *run-based prefetching* [17], and *C-Miner** [24, 25]. We have shown that the high concurrency degree common in e-commerce applications makes these techniques ineffective. We have also argued

that *QuickMine*'s incremental, dynamic approach is the most suitable in modern environments, where the number of applications, and number of clients for each application, hence the degree of concurrency at the storage server vary dynamically.

8 Conclusions and Future Work

The high concurrency degree in modern applications makes recognizing higher level application access patterns challenging at the storage level, because the storage server sees random interleavings of accesses from different application streams. We introduce *QuickMine*, a novel caching and prefetching approach that exploits the knowledge of logical application sequences to improve prefetching effectiveness for storage systems.

QuickMine is based on a minimally intrusive method for capturing high-level application contexts, such as an application thread, database transaction, or query. *QuickMine* leverages these contexts at the storage cache through a dynamic, incremental approach to I/O block prefetching.

We implement our context-aware, incremental mining technique at the storage cache in the Network Block Device (NBD), and we compare it with three state-of-the-art context-oblivious sequential and non-sequential prefetching algorithms. In our evaluation, we use three dynamic content applications accessing a MySQL database engine: the TPC-W e-commerce benchmark, the RUBiS auctions benchmark and DBT-2, a TPC-C-like benchmark. Our results show that context-awareness improves the effectiveness of block prefetching, which results in reduced cache miss rates by up to 60% and substantial reductions in storage access latencies by up to a factor of 2, for the read-intensive TPC-W and RUBiS. Due to the write intensive nature and rapidly changing access patterns in DBT-2, *QuickMine* has fewer opportunities for improvements in this benchmark. However, we show that our algorithm does not degrade performance by pruning useless prefetches for low performing contexts, hence avoiding unnecessary disk congestion, while gracefully adapting to the changing application pattern.

We expect that context-aware caching and prefetching techniques will be of most benefit in modern data center environments, where the client load and number of co-scheduled applications change continuously, and largely unpredictably. In these environments, a fully on-line, incremental technique, robust to changes, and insensitive to the concurrency degree, such as QuickMine, has clear advantages. We believe that our approach can match the needs of many state-of-the-art database and file-based applications. For example, various persistence solutions, such as *Berkeley DB* or Amazon's Dynamo [9], use a mapping scheme between logical identifiers and physical block numbers e.g., corresponding to the MD5 hash function [9]. Extending the applicability of our QuickMine algorithm to such logical to physical mappings is an area of future work.

Acknowledgements

We thank the anonymous reviewers for their comments, Livio Soares for helping with the Linux kernel changes, and Mohamed Sharaf for helping us improve the writing of this paper. We also acknowledge the generous support of the Natural Sciences and Engineering Research Council of Canada (NSERC), through an NSERC Canada CGS scholarship for Gokul Soundararajan, and several NSERC Discovery and NSERC CRD faculty grants, Ontario Centers of Excellence (OCE), IBM Center of Advanced Studies (IBM CAS), IBM Research, and Intel.

References

[1] Transaction processing council. http://www.tpc.org.

[2] ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Information and Control in Gray-Box Systems. In *Proc. of the 18th ACM Symposium on Operating System Principles* (October 2001), pp. 43–56.

[3] BAIRAVASUNDARAM, L. N., SIVATHANU, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proc. of the 31st International Symposium on Computer Architecture* (June 2004), pp. 176–187.

[4] BOWMAN, I. T., AND SALEM, K. Optimization of Query Streams Using Semantic Prefetching. *ACM Transactions on Database Systems 30*, 4 (December 2005), pp. 1056–1101.

[5] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. A study of integrated prefetching and caching strategies. In *Proc. of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS* (1995), pp. 188–197.

[6] CHANG, F. W., AND GIBSON, G. A. Automatic I/O hint generation through speculative execution. In *Proc. of the 3rd USENIX Symposium on Operating Systems Design and Implementation* (1999), pp. 1–14.

[7] CHEN, Z., ZHANG, Y., ZHOU, Y., SCOTT, H., AND SCHIEFER, B. Empirical Evaluation of Multi-level Buffer Cache Collaboration for Storage Systems. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (June 2005), pp. 145–156.

[8] CHEN, Z., ZHOU, Y., AND LI, K. Eviction-based Cache Placement for Storage Caches. In *Proc. of the USENIX Annual Technical Conference, General Track* (June 2003), pp. 269–281.

[9] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proc. of the 21st ACM Symposium on Operating Systems Principles* (2007), pp. 205-220.

[10] DING, X., JIANG, S., CHEN, F., DAVIS, K., AND ZHANG, X. Diskseen: Exploiting disk layout and access history to enhance I/O prefetch. In *USENIX Annual Technical Conference* (2007), USENIX, pp. 261–274.

[11] GAO, K., HARIZOPOULOS, S., PANDIS, I., SHKAPENYUK, V., AND AILAMAKI, A. Simultaneous pipelining in QPipe: Exploiting work sharing opportunities across queries. In *Proc. of the 22nd International Conference on Data Engineering, ICDE* (April 2006), pp. 162–174.

[12] GNIADY, C., BUTT, A. R., AND HU, Y. C. Program-counter-based pattern classification in buffer caching. In *Proc. of the 6th Symposium on Operating System Design and Implementation* (2004), pp. 395–408.

[13] GRIFFIOEN, J., AND APPLETON, R. Reducing file system latency using a predictive approach. In *Proc. of the USENIX Summer Technical Conference* (1994), pp. 197–207.

[14] HAAS, L. M., KOSSMANN, D., AND URSU, I. Loading a cache with query results. In *Proc. of 25th International Conference on Very Large Data Bases* (1999), pp. 351–362.

[15] HARIZOPOULOS, S., AND AILAMAKI, A. StagedDB: Designing database servers for modern hardware. *IEEE Data Eng. Bull. 28*, 2 (2005), 11–16.

[16] HSU, W. W., SMITH, A. J., AND YOUNG, H. C. I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level. *ACM Transaction on Database Systems 26*, 1 (2001), 96–143.

[17] HSU, W. W., SMITH, A. J., AND YOUNG, H. C. The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems 23*, 4 (2005), 424–473.

[18] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 14–24.

[19] KEETON, K., ALVAREZ, G., RIEDEL, E., AND UYSAL, M. Characterizing I/O-intensive workload sequentiality on modern disk arrays. In *4th Workshop on Computer Architecture Evaluation using Commercial Workloads* (2001).

[20] KROEGER, T. M., AND LONG, D. D. E. Design and implementation of a predictive file prefetching algorithm. In *Proc. of the USENIX Annual Technical Conference* (2001), pp. 105–118.

[21] KUENNING, G. H., POPEK, G. J., AND REIHER, P. L. An analysis of trace data for predictive file caching in mobile computing. In *Proc. of the USENIX Summer Technical Conference* (1994), pp. 291–303.

[22] LEI, H., AND DUCHAMP, D. An analytical approach to file prefetching. In *Proc. of the USENIX Annual Technical Conference* (1997), pp. 21–21.

[23] LI, X., ABOULNAGA, A., SALEM, K., SACHEDINA, A., AND GAO, S. Second-Tier Cache Management Using Write Hints. In *Proc. of the Conference on File and Storage Technologies* (December 2005).

[24] LI, Z., CHEN, Z., SRINIVASAN, S. M., AND ZHOU, Y. C-Miner: Mining Block Correlations in Storage Systems. In *Proc. of the FAST '04 Conference on File and Storage Technologies* (San Francisco, California, USA, March 2004), pp. 173–186.

[25] LI, Z., CHEN, Z., AND ZHOU, Y. Mining Block Correlations to Improve Storage Performance. *ACM Transactions on Storage 1*, 2 (May 2005), 213–245.

[26] LIANG, S., JIANG, S., AND ZHANG, X. STEP: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *Proc. of the 27th IEEE International Conference on Distributed Computing Systems* (2007), p. 64.

[27] MADHYASTHA, T. M., GIBSON, G. A., AND FALOUTSOS, C. Informed Prefetching of Collective Input/Output Requests. In *Proc. of the ACM/IEEE Conference on Supercomputing: High Performance Networking and Computing* (1999).

[28] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proc. of the 15th ACM Symposium on Operating System Principles* (1995), pp. 79–95.

[29] PHILLIPS, L., AND FITZPATRICK, B. Livejournal's backend and memcached: Past, present, and future. In *Proc. of the 19th Conference on Systems Administration* (2004).

[30] RAAB, F. TPC-C - The Standard Benchmark for Online transaction Processing (OLTP). In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. 1993.

[31] RIEDEL, E., FALOUTSOS, C., GIBSON, G. A., AND NAGLE, D. Active disks for large-scale data processing. *IEEE Computer 34*, 6 (2001), 68–74.

[32] SCHINDLER, J., GRIFFIN, J. L., LUMB, C. R., AND GANGER, G. R. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proc. of the Conference on File and Storage Technologies* (January 2002), pp. 259–274.

[33] SCHLOSSER, S. W., AND IREN, S. Database Storage Management with Object-based Storage Devices. In *Workshop on Data Management on New Hardware* (June 2005).

[34] SHEN, K., YANG, T., CHU, L., HOLLIDAY, J., KUSCHNER, D. A., AND ZHU, H. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems* (March 2001), pp. 197–208.

[35] SIVATHANU, G., SUNDARARAMAN, S., AND ZADOK, E. Type-Safe Disks. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, USA, November 2006).

[36] SIVATHANU, M., BAIRAVASUNDARAM, L. N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Database-Aware Semantically-Smart Storage. In *Proc. of the FAST '05 Conference on File and Storage Technologies* (December 2005), pp. 239–252.

[37] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-Smart Disk Systems. In *Proc. of the FAST '03 Conference on File and Storage Technologies* (March 2003).

[38] SRIKANT, R., AND AGRAWAL, R. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proc. of the 5th International Conference on Extending Database Technology* (March 1996), pp. 3–17.

[39] WILKES, J. Traveling to Rome: QoS specifications for automated storage system management. In *Proc. of 9th International Workshop on Quality of Service* (2001), pp. 75–91.

[40] WONG, M., ZHANG, J., THOMAS, C., OLMSTEAD, B., AND WHITE, C. *Database Test 2 Architecture*, 0.4 ed. Open Source Development Lab, `http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_su%ite/osdl_dbt-2/dbt_2_architecture.pdf`, June 2002.

[41] WONG, T. M., AND WILKES, J. My Cache or Yours? Making Storage More Exclusive. In *Proc. of the USENIX Annual Technical Conference, General Track* (June 2002), pp. 161–175.

[42] YAN, X., HAN, J., AND AFSHAR, R. CloSpan: Mining closed sequential patterns in large databases. In *Proc. of the 3rd SIAM International Conference on Data Mining* (2003).

[43] YEH, T., LONG, D. D. E., AND BRANDT, S. A. Increasing predictive accuracy by prefetching multiple program and user specific files. In *Proc. of the 16th Annual International Symposium on High Performance Computing Systems and Applications* (2002), pp. 12–19.