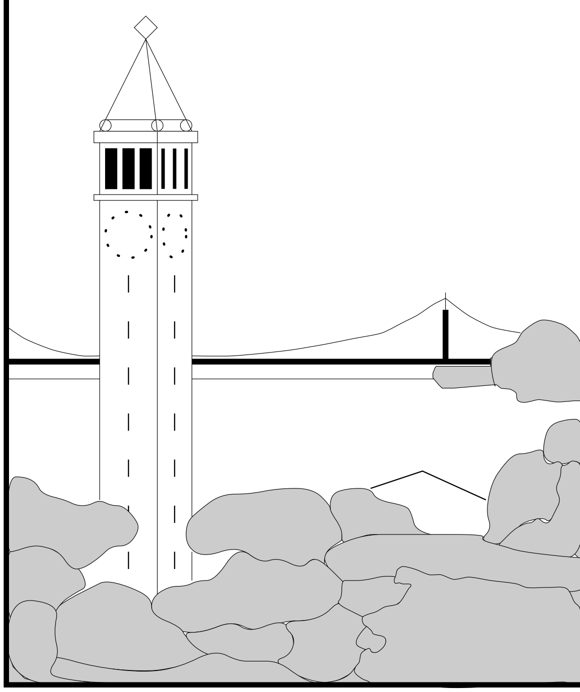
Program Context Specific Buffer Caching with AMP

Feng Zhou, Rob von Behren and Eric Brewer
Computer Science Division
University of California, Berkeley
{zf,jrvb,brewer}@cs.berkeley.edu



Report No. UCB/CSD-05-1379

April 2005

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Program Context Specific Buffer Caching with AMP

Feng Zhou, Rob von Behren and Eric Brewer
Computer Science Division
University of California, Berkeley
{zf,jrvb,brewer}@cs.berkeley.edu

Abstract

We present Adaptive Multi-Policy buffer caching (AMP), which uses multiple caching policies within one application, and adapts both which policies to use and their relative fraction of the cache. AMP differentiates disk requests based on the program contexts that issue them, and detects access pattern for each program context. This approach leads to detected patterns more stable than previous approaches that do detection based on processes and files. Compared to recent work, AMP is unique in that it employs a novel robust scheme for detecting looping patterns in access streams, as well as a low-overhead randomized way of managing many cache partitions. We show that AMP outperforms non-detection-based caching algorithms on a variety of workloads. For example, AMP reduces cache miss rate by up to 50% on a large database workloads. Other applications show about 20%-30% reduction in cache miss rate. Compared to other detection-based schemes, AMP detects access patterns more accurately in our experiment with a series of synthesized workloads, and incurs up to 15% fewer misses for one of the application traces. Experience with a Linux implementation is reported. Our prototype shortens the run time of a large database workload by 9.6%.

1 Introduction

Modern applications rely on increasing amounts of data with widely varying access patterns. Although traditional workstation OS workloads show temporal locality [1, 21], today's large disks and fast networks encourage both users and programs to create and save large amounts of data. Many end user applications stream large amounts of data with little reuse; others have complex mixes of sequential, hierarchical, and random accesses. Internet servers and their database backends often handle data for millions of users, with high degrees of concurrency. Disk access patterns for these modern applications deviate significantly from those of traditional workstation workloads.

Despite this increase in both I/O volume and complexity, disk access times continue to slow down relative to processor and memory speeds; there is no Moore's Law equivalent for disk access time. Thus, optimizing disk access is central to achieving good performance in modern applications.

In this paper, we take a new approach to disk caching, which we call adaptive multi-policy caching (AMP). Our goal is create a cache that adapts automatically not just to different applications, but to different behaviors within one application. It is *adaptive* because it observes these behaviors and adjusts as the workload changes, and it is *multi-policy* because it automatically uses the most appropriate policy for each different part of the program. For example, when AMP detects looping scans it will switch to MRU for that part of the application, while leaving the rest to a default policy (such as LRU).

The traditional focus of caching has been to find one policy that works well for all applications. This is well known to hurt some subset of applications, especially databases [24]. A possible solution is to examine *application-specific* caching, which aims to find the best policy for a given (small) set of applications, or even a single application. For example, the database community has a wealth of work on tailoring disk caching to the specific activities within the database [5, 18]. OS researchers have proposed various schemes for allowing application control of physical memory [9, 12]. Still others advocate per-application *detection-based* caches, to try to capture application-specific requirements automatically [4, 14]. These solutions all have significant drawbacks, either requiring too much intervention by the application programmer or simply not solving the problem well for important cases.

The AMP approach differs in a few significant ways. First, we automatically identify the different application *program contexts* that initiate each disk access. The program context from which a request originates often correlates with the logical purpose of that request. For example, a database may call `read()` both while scanning a table and traversing a tree, with clearly different access patterns for each[1]. Second, for each context we use a novel, robust algorithm to detect the access pattern and page reuse frequency. This allows us to pick the best policy for that context automatically; in particular, without any input from the application programmer. Finally, we automatically partition the cache space among contexts adaptively with a randomized method based on the idea of *equal marginal benefit* for each context. Each of these three facets of our approach required novel research, which we also present.

The results of our initial prototype are very promising. AMP is robust to changes in both workload and cache size, and consistently delivers performance equal to or better than other policies tested. In particular, AMP greatly outperforms ARC and LRU when the workload is much larger than the cache.

2 Disk Caching Algorithms

The widening gap between processor and disk performance makes disk cache hit rates critical to performance of I/O-bound applications. In this section we explore some of the fundamental problems that face cache algorithm designers. We start by examining the problems with simplistic replacement policies such as LRU, LFU, and MRU. Next, we discuss several ways in which recent algorithms have addressed these problems. Finally, we present the key insights that motivate our approach to disk cache management.

2.1 Simple Replacement Policies

Modern cache eviction policies are based on three simple strategies: least recently used (LRU), least frequently used (LFU) and most recently used (MRU). The problem with all three is that they work well for some workloads and poorly for others. For example, LRU works well when there is a high

[1]PCC [10] by Gniady et al. uses the same idea. AMP was developed concurrently with PCC.

degree of locality among requests but behaves badly if the workload includes large sequential scans. In fact, if the scan size exceeds the cache size, LRU performs pessimally with a zero percent hit rate. MRU has good performance for looping scans but poor performance when requests have high locality. LFU is well tailored to workloads following the Independent Reference Model [6] (accesses to each page are independent and have a fixed probability) such as random B-tree lookups. In these workloads some pages are likely to be accessed more frequently than others. However LFU performs poorly when the active set changes over time.

2.2 Recent Advances

Recent cache replacement algorithms avoid some of the problems of these simple strategies by employing a hybrid approach. In general, this entails using information about the workload to either overcome specific shortcomings in a basic algorithm or to modify the basic algorithm dynamically as the workload changes. Researchers have also employed a number of schemes to differentiate among applications or application subsystems, which may have different access patterns. We discuss several strategies below, along with representative examples of each.

Balancing recency and frequency. Of the basic algorithms, LRU performs well on the widest range of real workloads. Hence, many proposals start with LRU and include frequency information to make it more resistant to scans. Examples of such algorithms are LRFU [15], LRU-K [19], 2Q [13], ARC [17] and CAR [2].

ARC and CAR have been shown empirically to be superior to the other variants in both cache hit ratio and overhead. The basic idea of ARC/CAR is to partition the cache into two queues, each using either LRU (ARC) or CLOCK (CAR) replacement. Pages accessed twice recently are promoted from the first queue to the second. Therefore a one-pass scan will not evict all useful pages from the cache. Intuitively the first queue holds recently used pages, while the second queue holds frequently accessed pages. ARC/CAR adapts the sizes between two partitions using history information.

Like LRU, however, these policies focus on the microscopic details of the workload. Hence they cannot take advantage of larger trends in the workload or distinguish among different access patterns at different program contexts. For example, although both ARC and CAR are *scan-resistant*, they perform just as badly as LRU when faced with large, looping scans.

Application-specific policies. Another approach is to divide pages into logical *domains*, each with its own caching policy. Application writers can then provide hints to the caching system to determine which pages belong to each domain. Application-Controlled File Caching Policies [16] and DBMIN [5] are two such systems. Transparent Informed Prefetching [22] did the analog for prefetching. These all share the problem that they require significant effort on the part of the programmer, and tend to be brittle as the program evolves.

Detection-based caching. The final category of methods we will focus on in this paper is *detection or classification based approaches*, which explicitly try to identify specific patterns in I/O access streams and apply the most appropriate policies. The first design choice to make for a detection based approach is the *definition of an I/O stream* to do detection on. UBM [14] does per-file detection. DEAR is based on per-process detection. The recent PCC [10], as well as AMP, does per-program context (referred to as program counter in [10]) detection.

One key design decision then is the *pattern detection algorithm* given an access stream, where AMP differs from previous work. UBM watches for consecutive accesses to the file. If found, depending on whether repeated accessed are observed, the stream is classified as either looping or sequential, or 'other' is not found. One obvious problem with this approach is that only loops that access *consecutive* blocks will be detected, i.e. those accessing blocks i, $i+1$, $i+2$, ... DEAR sorts blocks by last-access time, and observes whether the more recently accessed block gets accessed farther in the future, indicating a loop. Although it detects non-consecutive loops, it's more expensive than UBM (involving keeping last access time for each block and sorting) and seems to be brittle to fluctuations in block ordering, according to our experiments. PCC resorts to a simple counter-based pattern detection algorithm. A stream is classified as looping if there are fewer non-repeating (`Seq` blocks than repeating (`Loop`) blocks. Otherwise, it is classified as sequential if `Seq` is larger than or equal to a threshold, or other otherwise. A major case when this fails as we see it is that it is absolutely normal for temporally clustered streams to also have less non-repeating blocks than repeating ones. These will be wrongly classified by PCC as looping.

AMP does program context based caching with a new access pattern detection algorithm. It is a simple algorithm that detects consecutive or non-consecutive loops, is robust against small reorderings and utilizes ordering information such that it distinguishes between loops and highly clustered accesses (as opposed to PCC). Its overhead is small and can be made constant per access, independent of working set or memory size.

AMP is also novel in the way it manages the cache partitions. Both UBM and PCC evict the block with the least estimated "marginal gain". Because the marginal gain estimation changes over time, finding this block can be expensive. AMP, in contrast, uses a randomized eviction policy that is much cheaper and robustly achieves similar effectiveness.

2.3 Our Approach

The fundamental hypotheses of our approach are that:

- *the program context from which an I/O request originates is highly correlated with the access pattern for that request, and*
- *the application's function call stack at the time of the I/O can effectively identify the program context.*

Figure 1 illustrates the intuition behind these hypotheses. Suppose we have two fictitious applications running at the same time, `foo_db`, a database, and `bar_httpd`, a web server. The arrows in the figure indicate function calls. The figure shows that a single `read` system call seen by the OS can come from three initiating functions. The function names give an important clue as to what pattern each access will have: `btree_scan()` will do looping scans; `btree_tuple_get()` causes probabilistic accesses, and accesses from `process_http_req()` will be temporally clustered (having

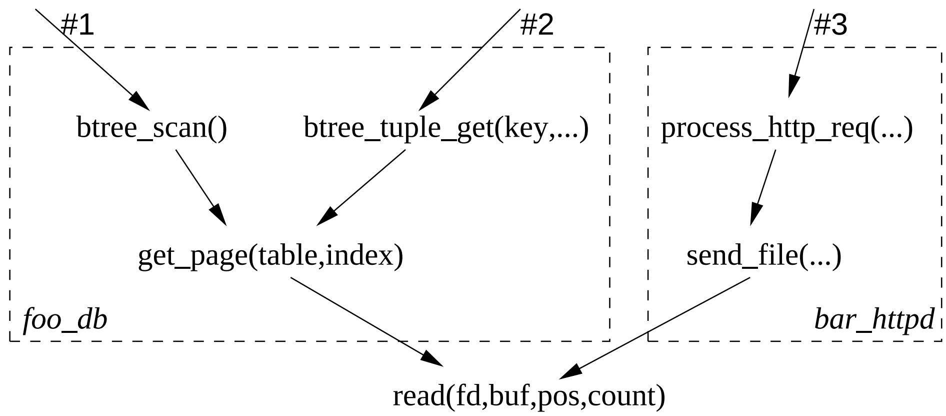


Figure 1: Call-stacks of two fictitious applications. Function names initiating requests suggest probable access patterns.

good locality). Hence, the best policy for each will be MRU, LFU and LRU respectively. If we can separate out the accesses from each of these functions, we have a good chance of detecting their access patterns and applying the best policy for each independently.

How do we determine which user-level function initiates the I/O request? The naive approach of simply using the user-level program counter fails because the function that "matters" may be way up in the call stack. As Figure 1 shows, `foo_db` issues all page requests in `get_page()`, so knowing that a request came from that function would not help.

In our solution, we do not try to find the one function that "conceptually" initiates the request, but instead use the whole call stack to identify the program context. This of course captures the real function. However, it may cause trouble for cases like recursive function calls, because each level of recursion will create a new program context. Fortunately in each of the cases we have explored, less than a dozen different call stacks dominate most I/O requests; that is, a few places issue most of the requests, and we can ignore locations with a small number of requests.

Our previous work on Capriccio [26] used runtime program context statistics to make better scheduling decisions. Program context IDs on most architectures can be obtained by simply following the stack frame-pointers and hashing together each frame's return address. We discuss practical issues about obtaining the program context ID in Section 3.3.

Figure 2 illustrates the power of separating I/O requests by program context. The figure shows the cumulative distribution of inter-reference gaps for major program contexts in our *glimpse* trace, which is of using *glimpseindex* to index the Linux kernel source code. The *inter-reference gap* is defined as the number of accesses to other pages that separate two accesses to the same page [23]. The vertical jumps in the graph indicate that a large number of pages at each program context have similar inter-reference gaps. Pages from program context 1, 2 and 3 (lines on the right of the figure) are referenced long after the first occurrence in most cases. The majority of pages from program context 4 and 5 both have inter-reference gaps of 4000 and 2000, respectively. So it is like that program contexts 4 and 5 are doing looping scans with those intervals.

All program contexts in this trace show a high degree of regularity in their block inter-reference gaps. By contrast, the combined CDF line is much smoother and does not show a clear pattern. This

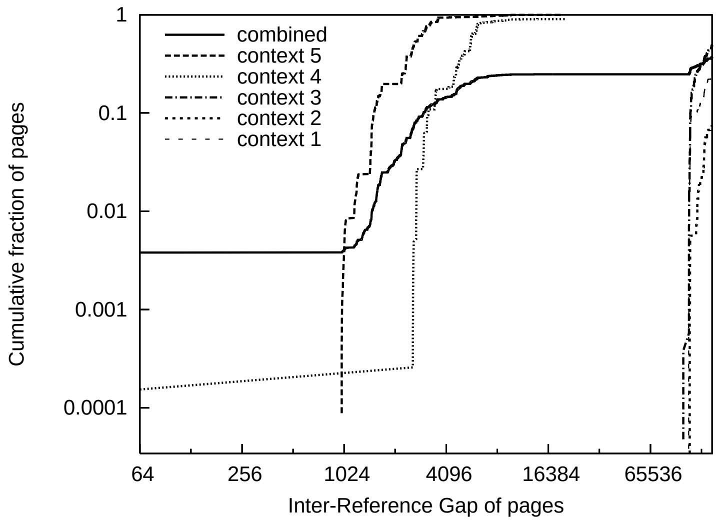


Figure 2: Inter-Reference Gap cumulative distribution of major program contexts in trace *glimpse*.

shows that differentiating accesses by program context can lead to much more accurate predictions of future accesses. This finding is analogous to the intuition behind the database caching algorithm DBMIN [5]. For the database buffer manager, the high-level actions of the database correlate much better with the overall access pattern than do the table or disk file.

As we shall show, our program context-based approach allows us to achieve the benefits of programmer-supplied caching hints by extracting this information automatically from running applications, requiring no programmer intervention at all. The result is a robust system that achieves similar or better performance to those with programmer hints, suffers lower risk of stale or incorrect information, and performs better than previous general caching policies.

3 Design and Implementation

In this section we present the Adaptive Multi-Policy framework (AMP), a simple, practical and general framework for program context-specific buffer caching. We also examine the in more detail the problems of detecting access patterns and allocating cache space among partitions.

The major features of this framework are:

- *Automatic*: AMP achieves application-specific caching without programmer-provided hints. This is a major advance from previous work.
- *Two-level adaptation*: The AMP framework employs both a high-level, relatively infrequent adaptation of policies and a low-level, continuous adaptation of cache partitioning among program contexts.
- *Opportunistic optimization*: The cache is organized such that when no obvious application-specific optimization is possible, AMP performs identically to the default cache replacement

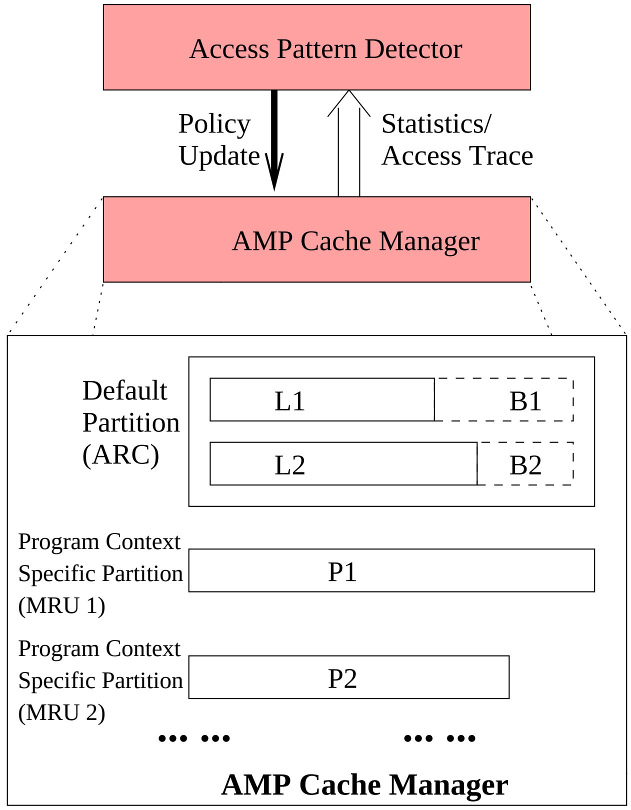


Figure 3: Adaptive Multi-Policy framework.

policy for the system. When components of an application show access patterns that AMP can exploit, AMP will perform better than the baseline policy, sometimes quite dramatically.

- *Adaptable to application boundaries*: AMP automatically discovers program contexts within an application and retains this information across runs. Hence, AMP can be successfully used on frequently-used, short-running processes (e.g. most Unix-style utilities), complicated long-running servers, and collections of programs working in concert.

AMP is composed of two components, shown in Figure 3. The *AMP Cache Manager* subsumes the original OS cache manager. It maintains a default cache partition that holds all "normal" blocks not subject to optimization, using a default policy such as ARC or LRU.[2] More importantly, it maintains one partition for each optimized program context, using the appropriate policy for that program context. It continuously adapts to the workload and adjusts the sizes of the partitions.

The *AMP Cache Manager* handles the low level details of deciding which pages to evict and apportioning pages among the various cache partitions. The *Access Pattern Detector* periodically runs AMPs access-pattern detection algorithm, using statistics and sampled access traces provided by the cache manager. It then feeds the policy decisions it made based on detection results back

[2]From now on, we assume the default policy is ARC in our discussion. Using other default policies requires slight changes to the size adaptation but the general approach applies.

to the cache manager. In general, the best interval between policy adaptations depends on the workload. However, for long-running servers, this high-level adaptation should be viewed as a "tuning" operation and thus run infrequently, e.g. once in several hours or even invoked manually by the administrator.

AMP's basic runtime algorithm is best illustrated through a simple example. When the OS starts an application it has never run before, all of its blocks go into the default cache partition. As the system collects statistics about the application's program contexts, the access pattern detector may decide that some of these contexts show exploitable access patterns and cause the cache manager to allocate new cache partitions for them.

Significantly, AMP's caching hints are workload-independent and hence can persist across application runs. For example, the hints might indicate that `/usr/bin/gcc` does sequential scans (perhaps reading source files) from several program contexts, identified by call-stack signatures $s_1, s_2, \ldots$ etc. After this is detected, AMP can use this information immediately in subsequent runs without requiring a detection phase. This applies until `gcc` itself is changed, when for example a new version of it is installed.

The following subsections explore the details of various aspects of the AMP design.

3.1 Access-Pattern Detection

The AMP access pattern detector periodically assigns one of the following block access patterns to each program context, in order to determine the policy to use for each context.

1. **One-shot**: one-time accesses.
2. **Looping**: repeated accesses to a series of blocks in the same or roughly the same order. These blocks could be physically sequential or not.
3. **Temporally clustered** [1]: accesses characterized by the property that blocks accessed recently are more likely to be accessed in the future.
4. **Others**: when none of the above apply.

Program contexts that always issue one-shot accesses are easy to identify. If none of the blocks accessed by a context are accessed again after a long time, the detector assigns the one-shot pattern to the context. We distinguish the other other patterns using a simple but robust scheme based on *average reference recency*.

If a sequence of accesses is temporally clustered, then the more recently a block was accessed, the more likely it is to be accessed again. In contrast, in a looping access the likely blocks to access next are those accessed the least recently. Hence, one way to distinguish these access patterns is to measure the *recency* of blocks accessed. The more recently accessed blocks the sequence accesses on average, the more it is likely to be temporally clustered, and vice versa. Concretely, we measure this *reference recency* of a block access by looking at the *relative position* of the last appearance of the same block, in the list of all previously appeared blocks, ordered by their last reference time.

Formally, for the i-th access in a sequence, we let L_i be the list of all previously accessed blocks, ordered from the oldest to the most recent by their last access time. Note that each block only appears in L_i at most once, at its position of last access. Thus if the access string is [4 2 3 1 2 3], with time increasing to the right, then the last access is number 6 (for block 3), and $L_6 = \{4, 3, 1, 2\}$, and $L_7 = \{4, 1, 2, 3\}$, although we don't yet know the seventh block. It is helpful to think of an access to block B as moving B to the right of L_i for the next access, analogous to a "move to front" list algorithm. Let the length of the list be $|L_i|$, and the position of the block of interest be p_i, with the oldest position being 0 and newest position being $|L_i| - 1$.

We define the *reference recency* R_i of the i-th access as:

$$R_i = \begin{cases} p_i/(|L_i| - 1), & |L_i| > 1 \\ 0.5, & |L_i| = 1 \\ \perp, & \text{undefined for first access} \end{cases}$$

After we get the reference recency of each access, the *average reference recency* $\bar{R}$ of the whole string is simply the average of all defined R_i.

Example 1. Consider looping access string [1 2 3 1 2 3]. For the second access to block 1, $i = 4$ and L_4 ={1 2 3}. Thus $p_4 = 0$, since the last access to block 1 occurs in position 0 of L_4. The access has recency $R_4 = \frac{0}{3-1} = 0$. For the next access to block 2, $L_5 = \{2, 3, 1\}$ and $p_5 = 0$, and thus $R_5 = 0$. Similarly, $R_6 = 0$ too, and in fact any pure looping pattern will have $R_i = 0$ and thus $\bar{R} = 0$.

Example 2. Consider temporally clustered string [1 2 3 4 4 3 4 5 6 5 6]. The calculation of R_i is:

i	Block	L_i	p_i	R_i
1	1	empty	$\perp$	$\perp$
2	2	1	$\perp$	$\perp$
3	3	1 2	$\perp$	$\perp$
4	4	1 2 3	$\perp$	$\perp$
5	4	1 2 3 **4**	3	1
6	3	1 2 **3** 4	2	0.67
7	4	1 2 **4** 3	2	0.67
8	5	1 2 3 4	$\perp$	$\perp$
9	6	1 2 3 4 5	$\perp$	$\perp$
10	5	1 2 3 4 **5** 6	4	0.8
11	6	1 2 3 4 **6** 5	4	0.8

$\bar{R} = 0.79$ (over all defined R_i)

In general, for pure loop sequences such as example 1, $\bar{R}$=0. For highly temporally clustered sequences $\bar{R}$ is close to 1. It is also easy to see a uniformly random access sequence has $\bar{R} = 0.5$, because for each access, the position of the last access is uniformly distributed over 0 to $|L_i| - 1$. In this sense, the average reference recency metric provides a measure of the correlation between recency and future accesses. If $\bar{R} > 0.5$, recently accessed blocks are more likely than average to be accessed in the near future. If $\bar{R} < 0.5$, recently accessed blocks are less likely than average to be accessed. When $\bar{R}$=0.5, overall recency is not correlated with future accesses.

The $\bar{R}$ values can be estimated either continuously, updating results as each I/O request is issued, or periodically, in a record-then-compute fashion. A continuous detector using exponential moving average of R values as $\bar{R}$ instead of the definition above can respond faster to changes in workload. In contrast, the periodical one can be easier to implement, because it could be done at user-level and needs less interaction with the cache manager.

The pattern detector categorizes all contexts with $\bar{R} < T$ as having looping patterns, where T is an adjustable threshold. We use $T = 0.4$ in all our experiments. Contexts with looping accesses are managed using MRU. AMP does not currently distinguish further between temporally clustered and other patterns; these contexts are all managed using the default policy (currently ARC).

The $\bar{R}$ metric is quite robust against small permutations of accesses. For example, the relative position of a block changes very little if access to it is exchanged with the access before or after it. Also, temporally clustered accesses having localized small loops will not be (incorrectly) classified as looping.

Block sampling. It is easy to see that the cost of computing $\bar{R}$ per access is $O(|L|)$. This calculation could hence become rather expensive for PCs accessing a large number of blocks. Sampling can be applied to reduce this cost. The sampling method we use is to hash the block number and only consider the block when the hash is within a range. Note that by reducing the block count by a factor of f, the execution time of the detection will roughly improve by f^2, because both m and n are reduced by f. This will have little effect on the detection result because each useful context will access many blocks, and sampling does not change the access pattern.

We are also experimenting with an adaptive sampling method. For each access, one hashes the block number to a 32-bit integer and only consider the block when the hash is smaller than a threshold Q, initially Q_0. Moreover, Q is halved whenever $|L|$ grows over the limit N_L and doubled when $|L|$ drops below $N_L/2$ and $Q > Q_0$. This adaptation effectively uses lower sampling rates for higher volume PCs. Because the blocks samples are always a random subset of all blocks, the detection result should be affected minimally.

3.2 Partitioned Cache Management and Low-overhead Adaptation

The *Cache Manager* manages the cache according to the access pattern of each program context. One-shot blocks get dropped immediately after use. Looping program contexts will each get a separate cache partition, running an MRU replacement policy. All program contexts with the temporally clustered pattern or no detected patterns go to the default cache partition, running the ARC replacement policy.

Now that blocks from different program contexts go into different cache partitions, we need a scheme to allocate the limited cache blocks among them. We would like the allocation scheme to have *low overhead*, *adapt* to fluctuations in workload and *asymptotically approach optimal behavior*. Although globally optimal allocation may be impossible to achieve in some cases, we should at least aim to reach a local optimum.

This type of partitioning is a well-studied resource allocation problem. Previous work used *miss-rate derivatives* [25] and *marginal-gains* [22, 18, 14], which are similar ideas, as guides to allocation of new buffers to partitions. The *marginal-gain* is defined as the expected number of extra hits over unit time if we allocate another block to a cache partition. For a specific partition and stable workload on that partition, it is a function of the current partition size. One application of the marginal-gain function is to use a greedy policy that always allocates a new buffer to the partition with largest current marginal-gain.

However, AMP differs in the way marginal gains are used in cache eviction from previous work. Both UBM and PCC evicts from the partition with the smallest estimated marginal gain when a new free cache block is needed. This is the right thing to do when the marginal gain estimations are accurate and timely. But it may lead to a large number of wrong evictions when the estimation is inaccurate or overcorrecting when it is delayed. Both cases are often true with the estimations. AMP tries to avoid these problems by doing the evicting probablistically. It lets the partitions compete for cache blocks, in a way such that whenever a cache miss happens, we give the partition one more block with a probability proportional to its marginal-gain. And this is done without computing the marginal-gain literally. In this way, when a dynamic balance among the competing partitions is achieved, all the partitions will have the same marginal-gain and no gain can be achieved by reallocating one block from one partition to another. Therefore in this case we achieve local optimum. This adaptation process goes on continuously over time and follows the changes in workload.

Figure 4 shows the actual cache allocation and partition size adaptation scheme; it also defines some of the values discussed below.

The ARC partition is expanded by one block whenever a hit in any of its ghost queues is observed. For MRU partitions, the c_i variable acts as a "coupon" counter. The partition is given a coupon for each access to it. Later a new block is allocated to it, when it has enough coupons.

Let's see why the equations in this scheme are correct and estimate marginal-gains, assuming stable access patterns. Over a period of time t, suppose there are m hits in the ARC partition's ghost buffers. Under the policy above, the ARC partition will be enlarged m times. Also observe that m hits in ghost buffers mean the measured marginal-gain is approximately:

$$\mathrm{MG}_{ARC} = \frac{m}{(|B_1| + |B_2|)t}$$

Then we have,

$$m = t(|B_1| + |B_2|) \cdot \mathrm{MG}_{ARC} \tag{1}$$

For an MRU partition, the expected number of enlargements it gets according to the policy is $n_i(|B_1| + |B_2|)/\mathit{loop\_size}$, where n_i is the number of accesses to the partition during time t. For a stable looping pattern, we have $n_i = \frac{t \cdot \mathit{loop\_size}}{\mathit{loop\_time}}$. Hence the number of enlargements, m', can also be written as

$$\begin{aligned} m' &= \frac{t \cdot \mathit{loop\_size}}{\mathit{loop\_time}} \cdot \frac{|B_1| + |B_2|}{\mathit{loop\_size}} \\ &= \frac{t(|B_1| + |B_2|)}{\mathit{loop\_time}} \\ &= t(|B_1| + |B_2|) \cdot \mathrm{MG}_i \end{aligned} \tag{2}$$

The last step holds because $\mathrm{MG}_i = 1/\mathit{loop\_time}$.

From Equations 1 and 2, we see that the number of expansions for both the ARC partition and MRU partitions are proportional to their marginal gains with the same coefficient. At the same

Definitions and variables:

- L_1: LRU queue of *inactive* in-core blocks in the ARC partition.
- L_2: LRU queue of *active* in-core blocks.
- B_1: Ghost queue for L_1. Hits in B_1 will result in expansion of L_1.
- B_2: Ghost queue for L_2.
- $loop\_size_i$: Average number of blocks in each loop of accesses to partition i.
- $loop\_time_i$: The average virtual system time duration of each loop for partition i.
- c_i: an integer for each partition

On each access to partition i:
  c_i++

On cache miss of block p *in partition* i:
  if (a free block available)
    allocate it and return
  if (i is ARC AND ($p \in B_1$ OR $p \in B_2$))
    if (there is another non-empty partition)
      evict from another non-empty partition randomly
      return
    else
      do adaptation according to ARC policy
  if (i is MRU && $c_i \geq loop\_size_i/(|B_1| + |B_2|)$
      AND there is another non-empty partition)
    evict from another non-empty partition randomly
    $c_i = c_i - loop\_size_i/(|B_1| + |B_2|)$
    return
  if (p is non-empty)
    evict a block from this partition (i)
  else
    evict from another non-empty partition randomly

Figure 4: Cache partition size adaptation.

time they are growing by taking blocks randomly from each other. Therefore the system moves towards equal marginal-gain for each partition.

One issue is MRU "garbage collection". Over time, although the access patterns of program contexts are the same, the working set may gradually shift. In this case vanilla MRU will eventually fail because it holds only old data. To solve this problem, we can periodically garbage collect long-time unused blocks. We use the estimated loop time and standard deviation to schedule these sweeps. Currently AMP sweeps every $loop\_time + 3loop\_time\_dev$. Another issue is what to do with blocks accessed by several different program contexts over time. They are simply moved among the partitions and the partition sizes adjusted accordingly.

To summarize, the design of the AMP framework focuses on simplicity, low-overhead and adaptability. As we shall see, the combination of relatively stable program context access patterns with

AMP's two-level adaptation yields high performance caching that is both practical and adaptive. Moreover, AMP's modular design can easily take advantage of new access pattern detection schemes as they are discovered.

3.3 Linux Implementation

We have implemented AMP for Linux 2.6.8.1 and glibc compiled with frame-pointers (available in most distributions). The program context is identified by walking the user-level stack by following the framepointers and hashing together function return addresses. The prototype is stable enough to run large applications like database benchmarks for hours and never crashed in our measurements.

The AMP cache manager is implemented by extending the Linux buffer cache, called *page cache* because a page is the smallest unit of caching, instead of a disk block. We added a field to the physical page structure to indicate the partition a page is in. Partition 0 is the original Linux page cache partition, with a CLOCK-like replacement policy. Partition 1 is the one-shot partition, where pages are freed whenever new pages are needed. Other partitions are MRU partitions. The fact that buffer caching is tightly integrated into the virtual memory system in Linux poses some challenge to the implementation. For example, a page in the cache could become a user page or a disk I/O buffer page at any time. Therefore, not every page can be freed immediately when new pages are needed. Also for performance reasons, Linux frees pages in batches instead of freeing exactly the number needed. Care is taken to make sure that we follow the size adaptation rules discussed above.

The pattern detector is implemented at user-level to ease development and thus operates periodically. It calls upon a kernel trace collector periodically to collect sampled disk I/O trace, along with corresponding program context information. The kernel trace collector collects I/O events in memory for a while and then feeds them to the pattern detector. After detecting the patterns, the detector tells the kernel to allocate an MRU partition for each popular (in terms of unique block count) MRU context, up to a fixed number, e.g 10. This is done because managing a single partition for each MRU context is unnecessary and a waste of memory, because many of these partitions will be empty almost all the time. An alternative way is to combine contexts into partitions, although how to pick contexts to combine is an open question.

4 Simulation Study

4.1 Detection Scheme

We compared the AMP access detection scheme to DEAR[4] and PCC[10] (from Gniady et al.) using simulation. Implementation of both schemes are done by us based on specification in [4] and [10] respectively. All of these schemes work well for detecting pure looping patterns. Therefore we focus our experiments on accesses with patterns but, more importantly, irregularities. We synthesize several such access streams, as shown in Figure 5. Each stream accesses blocks numbering from 1-100. Text description of these streams, and the detection results of the algorithms are shown in Table 1. Although we cannot claim these synthesized streams are representative of those from real applications, it is safe to say that in a lot of cases, access streams from real applications can be

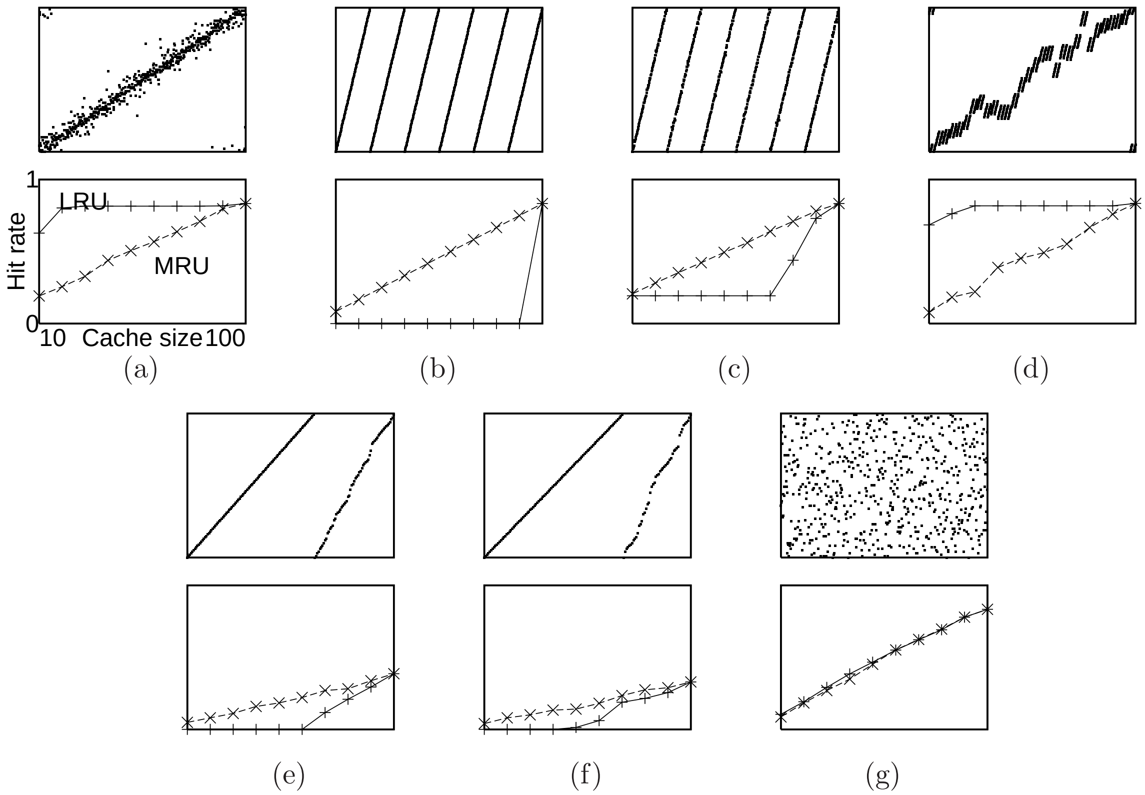


Figure 5: Traces used to compare detection schemes and their hit rates with LRU and MRU

much more irregular than these. Hence, a robust detection scheme should be able to handle these well.

The MRU and LRU hit rates in Figure 5 clearly indicate the best caching policy for each stream. Table 1 shows that AMP detects the correct pattern each time.

DEAR detects the correct patterns except for (c) and (f). The DEAR scheme requires two parameters: detection interval and group size. We set the detection interval to half of the stream length, so that DEAR does a single detection over the whole stream. Group size is set to 10. In general DEAR is quite sensitive to changes in the stream. For example, both (b)(c) and (e)(f) are pairs of similar streams. However DEAR succeeds for one but fails for the other in both cases. It is also sensitive to the parameters. We tried many different group sizes and found 10 to work best overall.

As discussed in section 2.2, the PCC detection scheme tends to mix locality with looping. Here it performs worse than DEAR and AMP, misclassifying 3 non-loop streams as loops. Actually it detects the highly temporally clustered stream (a) as looping.

Stream	AMP ($\bar{R}$)	DEAR	PCC	Description
a	*other* (0.755)	*other*	loop	temporally clustered accesses moving through the file
b	*loop* (0.001)	*loop*	*loop*	pure loops
c	*loop* (0.347)	other	*loop*	loops with each block moved around randomly
d	*other* (0.617)	*other*	loop	temporally clustered accesses with localized small loops
e	*loop* (0.008)	*loop*	*loop*	loops in which a block is accessed again with prob. 0.6
f	*loop* (0.010)	other	other	same as (e) except probability is 0.5
g	*other* (0.513)	*other*	loop	uniformly random

Table 1: Access pattern detection results of streams in Figure 5. Correct results are in italic.

Name	Total Pages	Unique Pages
cscope	80650	26936
gnuplot	24167	8783
scan	640718	49558
kernelbuild	403530	98699
glimpse	307324	71868
osdb	9610	3112
dbt3	1035152	84890

Table 2: Traces used in our evaluation.

4.2 Caching Performance

In the following subsections, we use trace-driven simulation as our primary means of studying caching performance of AMP. All traces are collected on a 2.4 GHz Pentium 4 Xeon PC server.[3] The traces are collected using the kernel tracing functionality of the AMP implementation. The kernel uses a compact in-memory event log during the tracing to avoid the interference that disk-based logging would require.

One difficulty we encountered when building the simulator is that [10] does not contain a detailed specification of PCC's partitioned cache manager. Hence, we implemented the PCC pattern detection algorithm and used AMP's cache management module. We believe this gives a fair evaluation of the detection algorithm because it should be orthogonal to the cache management algorithm. We call this hybrid scheme PCC*. Comparing the different partitioned cache management schemes is future work.

We collected a variety of traces to show different types of application workloads. Table 2 summarizes the accesses we observed. The Linux kernel manages buffer caches in pages. So we use pages as units in our discussion below. The details of each trace are described below. One note is that the *dbt3* trace is sampled in 1/7 by hashing the page number, as discussed in Section 3.1, because the original trace is too large (over 700M samples) for our simulator. Therefore its working set is reduced to 1/7 of the original.

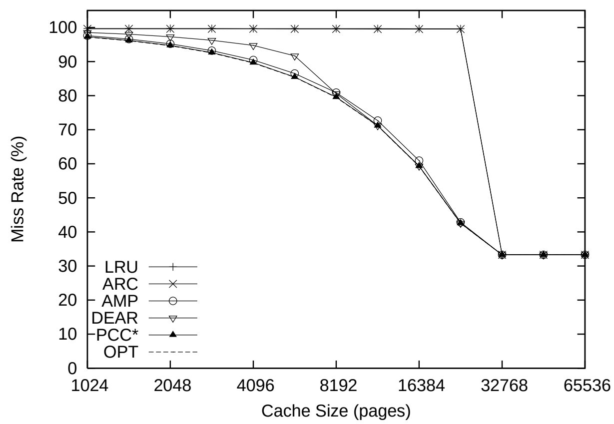


Figure 6: *cscope*

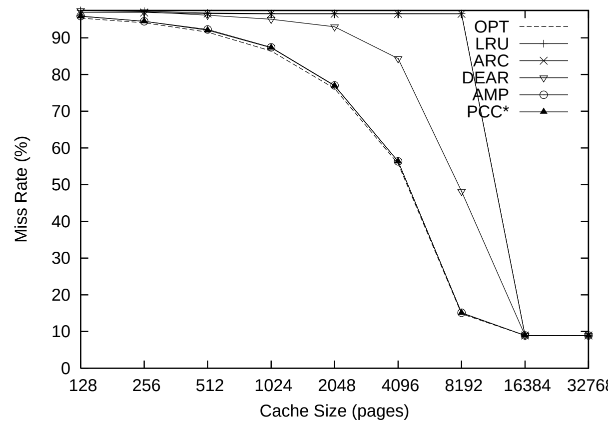


Figure 7: *gnuplot*

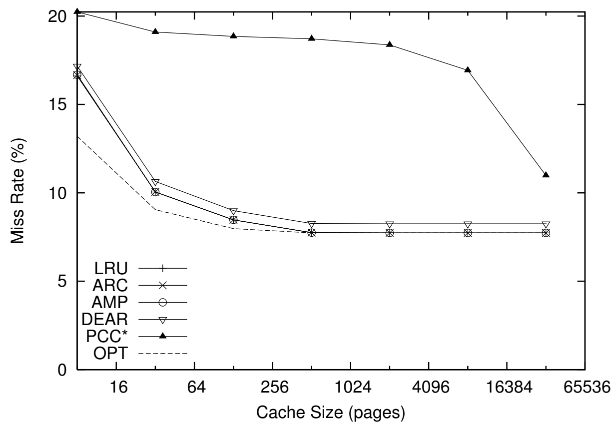


Figure 8: *scan* - simple workload where each file in a directory tree is read three times in whole.

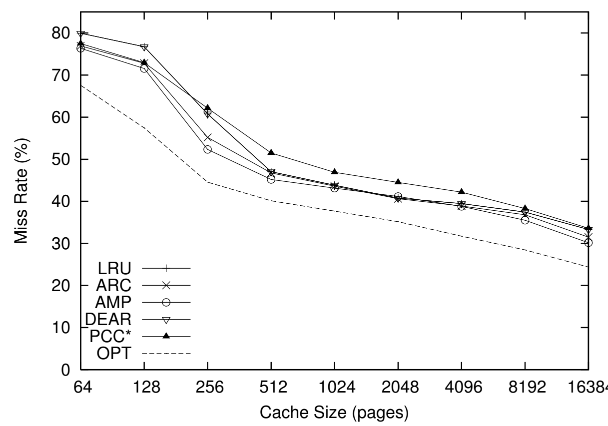


Figure 9: Linux kernel building. Most of this trace is temporally-clustered. AMP exploits looping accesses to program binaries and dynamic libraries.

4.2.1 Engineering Workloads

Figures 6 and 7 are results on two simple engineering workloads. The *cscope* trace records repeated queries to a C symbol index of the Linux kernel source code. The index file is 106MB in size. It also contains accesses to the matching source files. Because of the big looping accesses to the index file, neither ARC nor LRU sees any improvement until the cache is large enough to hold the whole index file. In contrast AMP performs very well. The *gnuplot* trace records the plotting of several large data files. The largest file is scanned multiple times to extract several data series. Therefore it exhibits both looping and sequential access patterns. Again, ARC and LRU perform poorly while AMP's miss rate decreases smoothly with cache size. In both traces, AMP and PCC* perform identically.

The trace *scan* (Figure 8) shows a pathological case for PCC*. In this trace, a test program walks

[3] We are unable to use publicly available traces because they don't include program context information.

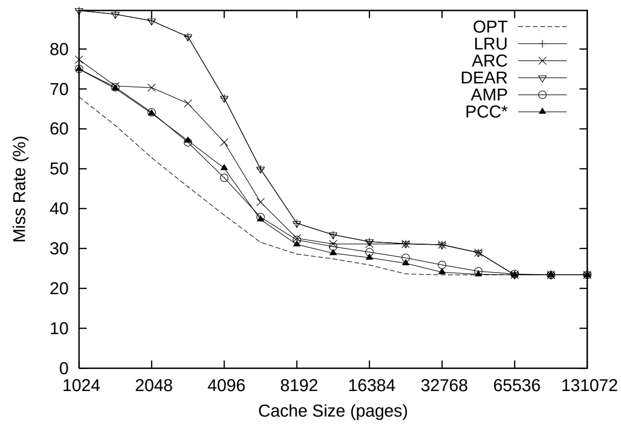


Figure 10: *glimpse*.

the Linux kernel source once and reads each file three times. These “small loops” are classified as “loop” by PCC and MRU is used. AMP classifies these as “temporally clustered” because more recent blocks get accessed. Figure 8 shows that PCC* performs much worse than all others, including LRU.

Figure 9 compares different cache policies for building the Linux kernel. Since the accesses in this trace show a high degree of locality and include many small loops, we expect that detection based methods would not improve things by applying MRU. In reality, PCC* and DEAR show degradation compared to LRU/ARC. However, PCC classifies some “small loop” contexts as loops and loses hits. AMP detects these correctly and shows slight improvements over LRU and ARC.

Figure 10 shows results for indexing the Linux kernel source code tree using the `glimpse` full text retrieval tool. It’s a relatively complex workload. AMP shows modest improvements over all other policies in most cases and comes closer to OPT. DEAR shows no improvement over LRU at all. PCC* perform similar to AMP, showing a bit of edge when cache size is large.

4.2.2 Database Workloads

Figure 11 compares two AMP variants with ARC and LRU on a database benchmark trace, OSDB [20]. OSDB is based on AS3AP, the ANSI SQL Scalable and Portable Benchmark, as documented in Chapter 5 of [11]. We used only the read-only “mixed-IR” portion of the test, because other parts contain many transactional writes and would not benefit much from buffer caching improvements. The part of interest uses a small (40MB) database and performs decision support queries. The database under test is MySQL 4.0.18. ARC and LRU perform identically, getting no hits at all until the cache is 1/3 the size of the working set. Both AMP variants perform dramatically better, improving smoothly as the cache size increases.

The LRU variant of AMP uses LRU as the default policy in place of ARC. The close performance of these two shows that AMP’s primary advantage comes from its program context analysis, rather than the choice of default policy.

Figure 12 shows cache miss rates of AMP, ARC and LRU on the trace *dbt3*, which is much larger than the previous trace. DBT3 [8] is an open-source implementation of the commercial TPC-H [7]

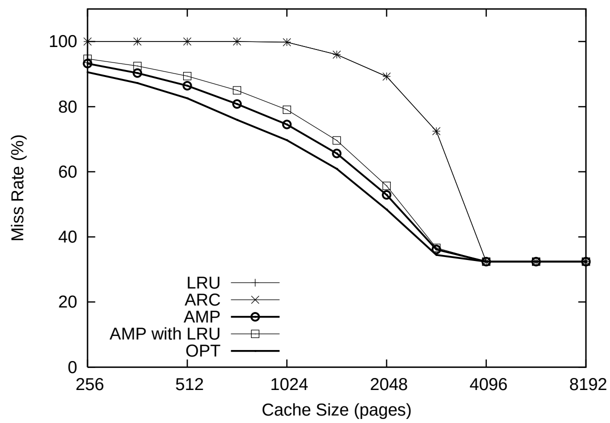


Figure 11: *osdb* (Open Source Database Benchmark)

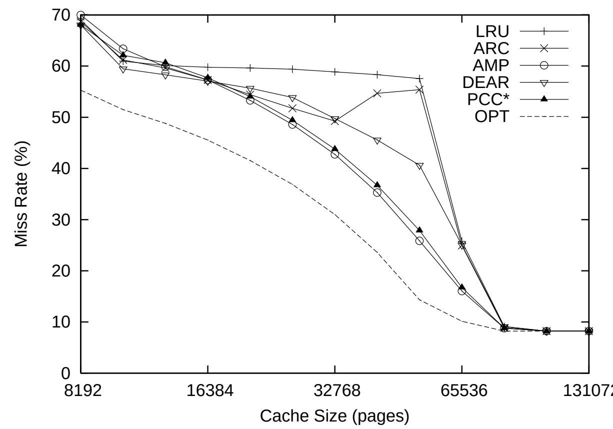


Figure 12: *dbt3* (a TPC-H benchmark implementation)

database benchmark. TPC-H models a decision support system. The main part of it a query stream composed of 22 pre-written, complex queries in random order. The data set is large compared to other experiments here. The whole database is about 4GB on disk. And each query is against a large portion, if not all, of the database. The benchmark is run on PostgreSQL 7.4.2. We only ran 16 of the 22 available queries because the other 6 queries take too long to finish given the time we have. Some queries taking too long (up to days) is a known issue discussed on the OSDL web site.

Figure 12 shows that AMP significantly out-performs ARC and LRU. The improvements are dramatic for several cache sizes. For example, for a cache size of 52016 pages, AMP achieves miss rate of 25.9%, compared with ARC at 55.4% and LRU at 57.6%, a reduction of more than 50%. In general, ARC and LRU perform poorly even if the working set is slightly larger than the cache size, because a lot of looping accesses are in the workload. AMP's hit rate increases smoothly with cache size.

It's worth pointing out that the fact that the PostgreSQL database relies on OS buffer caching makes this experiment possible. Most commercial databases do their buffer management at user-level by themselves. An in-kernel AMP implementation will not benefit them, although it would be a very interesting experiment to compare AMP to hand-tuned, complex database buffer managers.

4.3 Partition Size Adaptation

We now take a closer look at how the partition-size adaptation scheme works. Figure 13 is the history of size distributions between partitions in a run of the *glimpse* trace. MRU 1 (the top shaded area on the graph) corresponds to the scans through all indexed files. MRU 2-4 are accesses to the index files. ARC contains all other accesses. The jigsaws seen among MRU 2-4 are there because the two contexts are hitting each other's pages, not due to size adaptations. In this experiment, the working set of MRU 1 is much larger than MRU 2-4. But because the loops in MRU 2-4 have much shorter intervals than MRU 1, MRU 2-4 have larger marginal gains. Thus the graph shows that most cache space is allocated to the MRU 2-4. Over time the sum of MRU 2-4 is also growing because of the coupon mechanism we employed. The index files gets larger and larger

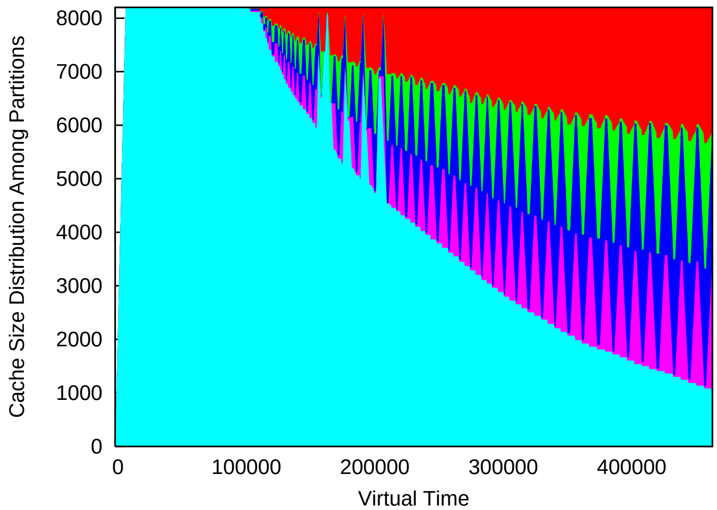


Figure 13: Partition size adaptation over time on *glimpse*. Each shaded area represents a partition. They are from top to bottom: MRU1, MRU2, MRU3, MRU4 and ARC.

over time. The number of accesses for each lookup in the indexes becomes larger. Therefore page access volumes of MRU 2-4 gets larger over time, which gives them more coupons and results in enlargement of the partitions.

4.4 Cost and Practical Issues

Our main overhead is that of generating the program context ID by walking the stack. On most machines this takes a few hundred cycles. Apart from that extra book-keeping, the cost per cache hit is about the same as the building-block policy, ARC, and MRU. Extra cost per miss may include that of adjusting partition sizes, which is also O(1) per page. If the stack-tracing overhead becomes a problem, a possible optimization is to only calculate it when a cache miss occurs, because each page in memory already has an attached program context ID. The downside is this may get pages accessed by multiple program contexts wrong. But this makes the stack-tracing overhead insignificant, since it will be dwarfed by the huge disk-access latency due to the cache miss.

Our high-level policy adaptation cost is minimal. As stated before, an important advantage of using relatively stable program context access patterns is that the access pattern detector does not need to be running all the time. This is much better than process-level adaptation, which requires constant adaptation. For example, [4] mentioned that DEAR requires one adaptation per 1000 requests or so.

GCC recently improved their no-framepointer compilation support and systems like Linux are gradually migrating to no-framepointer libraries and applications. Unwinding the call stack to get a list of function return addresses will be more expensive for these systems, but still possible. However, we can also use to compiler support or binary instrumentation to compute the identifier. We already have a simple algorithm to select function call sites to instrument, by either a modified compiler or the custom program loader, to compute the context identifier inexpensively. We plan to evaluate the necessity and effectiveness of this method in the future.

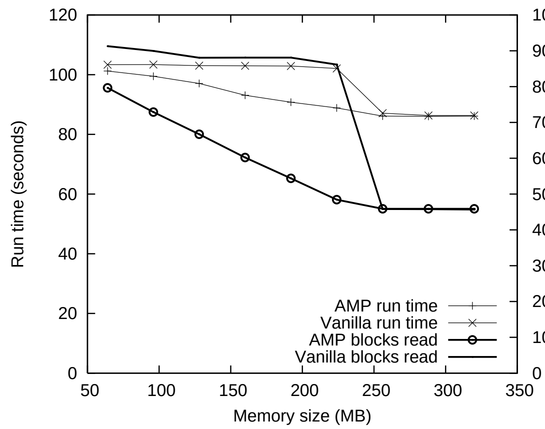


Figure 14: Glimpse on AMP and Linux page cache

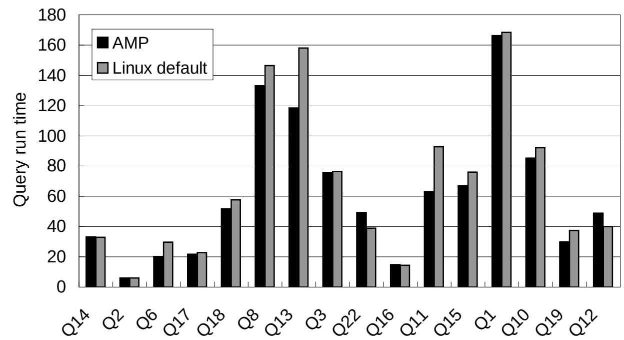


Figure 15: Execution time of queries in *dbt3* (a free *tpc-h* implementation)

5 Measurement

In this section, we compare our AMP implementation against the default Linux page cache implementation, by benchmarking real applications. We compared against Linux kernel 2.6.8.1, on which the AMP implementation is also based, so that other parts of the system is identical. The Linux page cache uses an improved CLOCK replacement policy [3]. There are two queues, *inactive* and *active*. All new pages go into the *inactive* queue. Pages are promoted to the *active* queue if they are accessed twice during their stay in the *inactive* queue. All replacement happens in the *inactive* queue. The kernel also moves old pages from the *active* queue back to *inactive* from time to time.

Our experiments are all done on a single-CPU 2.4 GHz Pentium 4 Xeon server with 1GB of memory. In order to vary the amount of memory available to the system, we wrote a simple program to allocate away the excess amount of memory and disable paging on them using the `mlock()` system call before the experiment begins. The program itself also uses about 10 KB of memory, which is negligible.

The first application we ran is `glimpse`. The setup is the same as in Section 4.2.1. The execution times and number of blocks read from disk are shown in Figure 14. AMP shows significant performance improvement over the Linux page cache. Run time is shortened by up to 13% and blocks read from disk is reduced by up to 43%, both when memory size is 224 MB. Execution time is not shortened as much as disk reads are saved because there are a lot of writes, which are the same for both system.

We also run the DBT3 [8] database workload, in the configuration as we collected the *dbt3* trace. We tuned the database parameters following instructions from both the database manual and DBT3. In the experiment we measure the execution time of each query and I/O activity of the system. The query stream is ran three times in the same order for each setting and average taken.

Figure 15 compares the execution times of queries on AMP and plain Linux. AMP did better in 11 queries and worse in 5 (Q14, Q2, Q8, Q22 and Q12) (reason under investigation). AMP shortens the total execution time of the query stream by 9.6%, from 1091 seconds to 986 seconds. Disk read traffic is reduced by 24.8% from 15.4 to 11.6 GB. Write traffic is also reduced by 6.5%, from 409 MB to 382 MB, probably due to lower cache contention.

6 Future Work

Our approach is complementary to innovations of building-block replacement policies. The individual caches for each program context will clearly benefit from using the most sophisticated caching policy for the specific access pattern. Moreover, our use of application information allows us to use specialized policies such as MRU that perform poorly in general. A number of additions can be made to the basic AMP design presented here to use other existing and new policies. For example, we plan to try LFU for probabilistic patterns like B-tree look-ups. It may also be interesting to see the results of using other policies such as 2Q [13] as the default policy. Although ARC seems to perform well in general, it may not be the best default policy for AMP, as the access pattern of the default partition is different from general workloads, since we remove one-shot and looping accesses (to other partitions).

We are also interested in exploring AMP's behavior when there are a large number of disparate application domains. We expect that aggregation of domains with similar access patterns will be useful to reduce the management overhead in this case. The specifics of how this aggregation should be done are an interesting research problem.

AMP makes it feasible to use narrow caching policies tailored to very specific situations. This may yield better performance for certain types of workloads. For example, a server that serves static content could improve overall latency by caching only the first portion of each file, and then using prefetching for the remaining portions. AMP makes it feasible to automatically detect this type of access pattern, and apply it only where it appears useful.

AMP's program context analysis lends itself well to other types of disk-access optimizations. For example, applications with a high degree of concurrency may be able to benefit from co-scheduling tasks likely to access the same data. This would yield better cache performance by decreasing the effective working set size at any one point.

Similarly, program context analysis could be used to make better prefetching decisions. For example Linux attempts to deduce appropriate read-ahead sizes on a per-file basis. it is likely that the accuracy of these predictions would be higher with program context information. In addition, AMP remembers patterns across runs, which should also help prefetching. We have preliminary results that suggest that program context analysis will be very effective at detecting and tuning prefetching behavior.

7 Conclusion

We have presented AMP, an adaptive caching algorithm that deduces information about an application's structure and uses this to pick the best cache replacement policy for each program context. Compared to recent and concurrent efforts, AMP is unique in that it uses a low-overhead and robust access pattern detection algorithm, as well as randomized cache partition size adaptation.

This potential is borne out in our experimental results. AMP minimally matches the performance of ARC, among the best general-purpose caching algorithms, regularly beats it by 25%, and can dramatically outperforms ARC for workloads that include looping scans (e.g. reduce miss rate by more than 50% for *dbt3* and even more for simpler application like *cscope*). For such workloads, AMP's hit rate increases steadily with the cache size, while ARC performs poorly unless a large fraction of the working set fits in the cache. We expect this to yield very smooth results for real applications that experience changes in working set size and workload frequency distribution.

AMP's program context analysis allows it to adapt well to the needs of particular application program contexts. Differentiating accesses by context is very effective at disambiguating overlapping access patterns. This allows AMP to adapt well to the needs of complicated applications and collections of simple applications. We achieve these benefits with low runtime overhead. Finally, AMP essentially produces a high-quality application-specific cache without any input from the programmer, and then adapts the policy automatically to changes in workload.

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 Thirteenth ACM Symposium on Operating Systems Principles*, pages 198–212. ACM Press, 1991.

[2] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST 04)*, 2004.

[3] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2nd edition, 2002.

[4] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. An implementation study of a detection-based adaptive block replacement. In *Proceedings of the 1999 Annual USENIX Technical Conference*, pages 239–252, 1999.

[5] Hong-Tai Chou and J. DeWitt David. An evaluation of buffer management strategies for relational database systems. *Readings in Database Systems*, pages 174–188, 1988.

[6] E. G. Coffman and P. J. Denning. *Operating System Theory*. Prentice-Hall, 1973.

[7] Transaction Processing Performance Council. TPC benchmark a standard specification, revision 2.0. June 1994.

[8] OSDL DBT3 database workload. `http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/osdl_dbt-3/`.

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266. ACM Press, 1995.

[10] Chris Gniady, Ali R. Butt, and Y. Charlie Hu. Program counter based pattern classification in buffer caching. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04) (to appear)*, 2004.

[11] Jim Gray. *Benchmark Handbook: For Database and Transaction Processing Systems*. Morgan Kaufmann Publishers Inc., 1992.

[12] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197. ACM Press, 1992.

[13] Theodore Johnson and Dennis Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 439–450, Santiago, Chile, 1994.

[14] Jongmin Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. A low-overhead high-performance unified buffer managament scheme that exploits sequential and looping reference. In *Symposium on Operating System Design and Implementation (OSDI'2000)*, 2000.

[15] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU (least recently/frequently used) replacement policy: A spectrum of block replacement policies. *IEEE Transactions on Computers*, 50(12):1352–1361, 2001.

[16] Kai Li, Edward W. Felten, and Pei Cao. Application-controlled file caching policies. In *Proc. of USENIX Summer 1994 Technical Conference*, pages 171–182, 1994.

[17] Nimrod Megiddo and Dharmendra S. Modha. Outperforming LRU with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[18] Raymond Ng, Christos Faloutsos, and Timos Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Trans. Comput.*, 44(4):546–560, 1995.

[19] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD Conference*, pages 297–306, 1993.

[20] The Open Source Database Benchmark. `http://osdb.sourceforge.net/`.

[21] John K. Ousterhout, Hervé; Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24. ACM Press, 1985.

[22] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95. ACM Press, 1995.

[23] Vidyadhar Phalke and Bhaskarpillai Gopinath. An Inter-Reference Gap model for temporal locality in program behavior. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 291–300. ACM Press, 1995.

[24] Michael Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, 1981.

[25] D. Thiebaut, H.S. Stone, and J.L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41:665–676, 1992.

[26] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 268–281. ACM Press, 2003.