AMP: Program Context Specific Buffer Caching

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 disk caching (AMP), which uses multiple caching policies within one application, and adapts both which policies to use and their relative fraction of the cache, based on program-context specific information. AMP differentiate disk requests based on the program contexts, or code locations, that issue them. Compared to recent work, AMP is unique in that it employs a new 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 by up to 50% in miss rate reduction. Compared to other detection-based schemes, we show that AMP detects access patterns more accurately for a series of synthesized workloads, and incurs up to 15% fewer misses for one application trace.

1 Detection (classification) Based Caching

Modern applications access increasing amounts of disk data and have widely varying access patterns. These access patterns deviate from traditional OS workloads with temporal locality. Recent work on *detection* or *classification* based caching schemes [2, 5, 4] exploits consistent access patterns in applications by explicitly detecting the patterns and applying different policies. In this paper, we present Adaptive Multi-Policy caching, a detection-based caching system with several novel features.

One major aspect of the design of a detection-based caching scheme is the *pattern detection algorithm*. UBM [5] detects loops by looking for accesses to physically consecutive blocks in files. One obvious problem is that some loops are to blocks not physically consecutive. DEAR [2] detects loops and probabilistic patterns by sorting blocks by last-access time and frequency. Although it detects non-consecutive loops, it's more expensive and brittle against fluctuations in block ordering. PCC [4] resorts to a simple counter-based pattern detection algorithm. Essentially a stream with many repeating accesses is classified as looping. This scheme, although simple, risks classifying temporally clustered streams with high locality as loops.

AMP features a new access pattern detection algorithm. It detects consecutive and non-consecutive loops and is robust against small reorderings. 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.

One decision differentiating these approaches is the *definition of an I/O stream* to do detection on. For example, UBM [5] does per-file detection and DEAR is based on per-process detection. AMP and PCC [4] do per-program context (referred to as program counter in [4]) detection. The basic idea is to separate access streams by the *program context* (identified by the function call stack) when the I/O access is made, with the assumption that a single program context is likely to access disk files with the same pattern in the future. This PC-based approach differs radically from previous methods and is the key idea in both PCC and AMP, although the two systems were developed concurrently and independently. Interested readers are referred to [4] and [7] for further motivations for this approach and application traces showing its effectiveness.

2 Design

AMP is composed of two components. The first component, *Access Pattern Detector*, uses an access-pattern detection algorithm to determine the access pattern for each program context issuing I/O calls. The other component, *AMP Cache Manager*, subsumes the original OS cache manager. It maintains a default cache partition that holds all "normal" blocks using a default policy such as ARC or LRU (assumed to be ARC later on). Then, 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.

2.1 Access-Pattern Detection

The AMP access pattern detector assigns one of the following block access patterns to each program context, similar to UBM and PCC: **One-shot** for one-time accesses; **Looping** for repeated accesses to a series of blocks in the same or roughly the same order, either physically consecutive or not. **Temporally clustered** [1] for accesses characterized by the property that blocks accessed recently are more likely to be accessed in the future. **Others** when none of the above apply.

PCs that always issue one-shot accesses are easy to identify, by simply observing if no blocks accessed by it are accessed again. The other patterns are distinguished based on a metric we call *average reference recency*. The intuition is that if a sequence is temporally clustered, then the more recently a block was accessed, the more likely it is to be accessed again. The exact contrary holds for looping sequences. Hence, one way to distinguish these access patterns is to measure the *recency* of blocks accessed. Concretely, we measure this *reference recency* of an access by looking at the *relative position* of the previous appearance of the same block in the list of all previously accessed 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, we have $L_6 = \{4, 3, 1, 2\}$, and $L_7 = \{4, 1, 2, 3\}$, although we don't yet know the seventh block. 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 $|L_i| - 1$.

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

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

Then 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 $p_4 = 0$, $R_4 = \frac{0}{3-1} = 0$. Also $L_5 = \{2, 3, 1\}$ 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]. With the calculations omitted, we get $\bar{R} = 0.79$.

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 $\bar{R} = 0.5$. 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, vice versa.

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 found $T = 0.4$ to be a good value in experiments. 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.

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. A quick calculation reveals that by sampling $1/m$ pages, the total overhead could be reduced to roughly $1/m^2$ of the original. We could also adapt the sampling rate such that $|L|$ is bounded, thus limits the per-access overhead to a constant upper-bound. For details, see [7].

2.2 Low-overhead Partitioned Cache Management

The *Cache Manager* manages the cache according to the access pattern of each program context. AMP bases cache partition sizing decisions marginal gain estimation [5]. However, it differs from previous work in the way marginal gains are used. Both UBM and PCC evict from the partition with the smallest estimated marginal gain when a free block is needed. Unfortunately, finding this block can be expensive. Moreover, gain estimations are often inaccurate and delayed, which may lead to a large number of wrong evictions and overcorrections. AMP avoids these problems by using a *randomized* eviction policy and allowing cache partitions compete for new blocks. When a cache miss occurs, AMP forces some *other* randomly chosen partition to evict a page to free memory for the new page. This serves to increase the size of each partition proportional to it's marginal gain; over time the partition sizes move towards a balance of equal marginal gain. This achieves the same local optimum as previous approaches but with far lower overhead.

The actual adaptation algorithm is briefly as follows (See [7] for complete specification and justification). When a cache miss happens, a free block is allocated if one is available. Otherwise, an occupied block needs to be evicted. If the missed block is in the ghost queues of ARC ($B1$ and $B2$ in [6]), a block from a random MRU partition is evicted. If instead the access is from a looping context, we evict from a random partition with probability *arc_ghost_len*/*loop_size*$_i$, where *arc_ghost_len* is the length of the ghost queues of the ARC partition and *loop_size*$_i$ is the estimated number of blocks in every loop of the missed MRU partition. Otherwise, we just evict the MRU block of the missed partition.

2.3 Linux Implementation

We have implemented AMP for Linux 2.6.8.1. The program contexts are identified by walking the user-level stack and hashing together function return addresses. The AMP cache manager is implemented by extending the Linux buffer cache, called *page cache*. The fact that buffer caching is tightly integrated into the virtual memory system in Linux poses some challenge to the implementation, as discussed in [7]. The pattern detector is implemented at user-level and operates periodically. It calls upon a kernel trace collector periodically to collect sampled disk

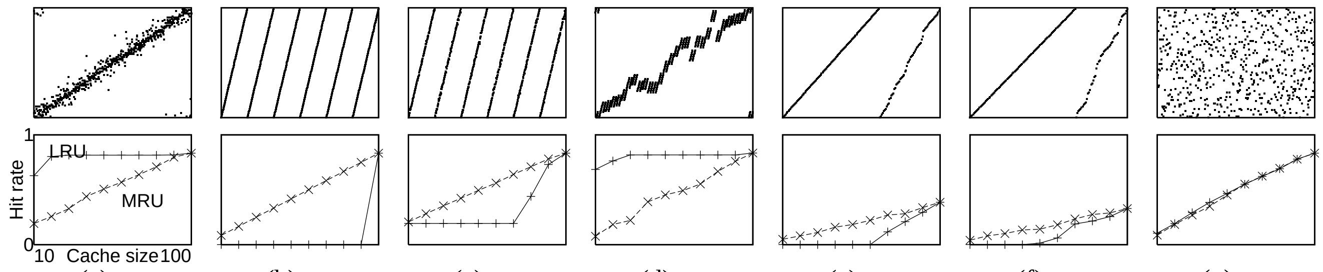

Figure 1: Traces used to compare detection schemes and their hit rates with LRU and MRU

Stream	AMP (R)	DEAR	PCC
a	other (0.755)	other	*loop*
b	loop (0.001)	loop	loop
c	loop (0.347)	*other*	loop
d	other (0.617)	other	*loop*
e	loop (0.008)	loop	loop
f	loop (0.010)	*other*	*other*
g	other (0.513)	other	*loop*

Table 1: Access pattern detection results of streams in Figure 1. *Incorrect* results are underlined.

I/O trace, along with corresponding program context information. The detection results are fed back to the kernel using the `/proc` file system interface.

3 Evaluation

3.1 Detection Scheme

We compared the AMP access detection scheme to DEAR[2] and PCC[4] using simulation. We implemented these based on the specifications in [2] and [4], respectively. All 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 1. Each stream accesses blocks numbered from 1-100. (a) shows temporally clustered accesses slowly moving through a file. (b) is pure loops and (c) is loops with each block moved around randomly (expected distance 0.5). (d) is temporally clustered with local loops. (e) and (f) are loops in which a block is accessed again with probability 0.6 and 0.5 respectively. (g) is uniformly random. The MRU and LRU hit rates in Figure 1 clearly indicate the best caching policy for each stream. Detection results of the algorithms are shown in Table 1.

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. 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. As discussed in section 1, the PCC detection scheme tends to mix locality with looping. Here it misclassifies three non-loop streams as loops. Actually it detects the highly temporally clustered stream (a) as looping.

3.2 Caching Performance

We used trace-driven simulation to study caching performance of AMP. Only a subset of our results are shown here; our full results are presented in [7]. All traces were collected on a 2.4 GHz Pentium 4 Xeon PC server using the tracing functionality of our AMP implementation. One difficulty we encountered was that [4] does not 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*.

Figure 2 shows results for using `cscope` to look up 5 symbols in the index (sized 106MB) of Linux kernel source code. Because `cscope` does big looping accesses to the index file, DEAR, PCC* and AMP all perform similarly and much better than LRU and ARC, which see no improvement until the cache is large enough to hold the whole index file.

The trace *scan* (Figure 3) shows a pathological case for PCC*. In this trace, a test program walks 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 3 shows that PCC* performs much worse than all others, including LRU.

Figure 4 shows performance while building the Linux kernel. Since the accesses in this trace show a high degree of locality and include many small loops, detection based methods would cannot improve things by applying MRU. In reality, PCC* and DEAR show degradation compared to LRU/ARC. PCC classifies some "small loop" contexts as loops and loses hits. AMP detects these correctly and shows slight improvements over LRU and ARC.

Our final simulation test was of DBT3 [3], an open-source implementation of the commercial TPC-H database benchmark. This a relatively large trace. The whole database is about 4GB, with each query against a large portion of the database. It is run on PostgreSQL 7.4.2. We ran only 16 of the 22 queries because the other 6 take too long to finish (hours to days) due to known issues

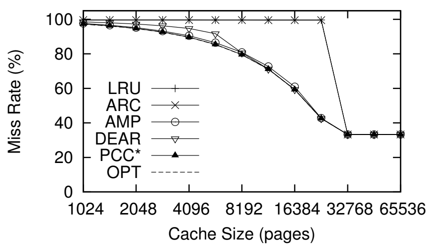

Figure 2: *cscope*

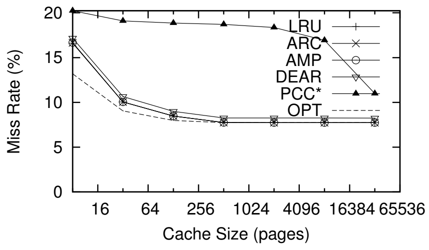

Figure 3: *scan*

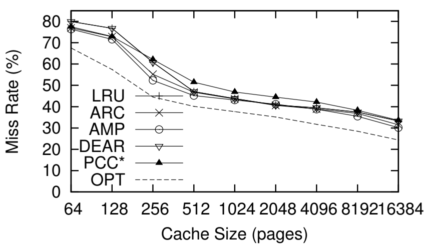

Figure 4: *linuxkernel*

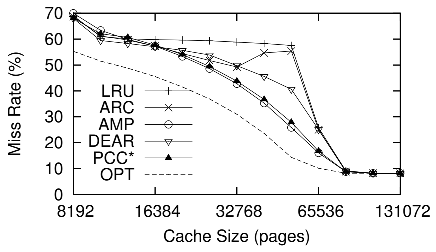

Figure 5: *dbt3*

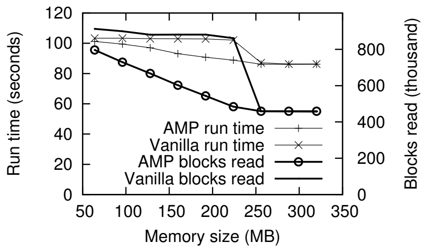

Figure 6: Glimpse on AMP and Linux

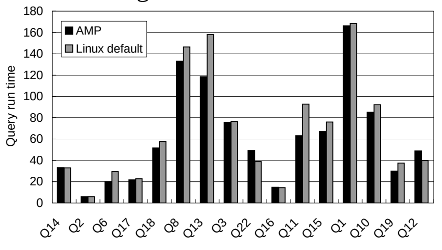

Figure 7: Query exec. times of *dbt3*

with PostgreSQL 7. At over 700M samples, this trace was too large for our simulator. Hence we down-sampled this trace by a factor of 7, reducing it's working set to $1/7$ of the original. Figure 5 shows detection based methods out-perform ARC/LRU again here. DEAR shows less improvement, probably because the the complex mix of accesses from the database process makes DEAR's process-based detection less accurate. AMP and PCC* yield greater improvements and perform similarly. 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%.

3.3 Measurement

Here we compare our AMP implementation with the default Linux page cache by benchmarking real applications. Our test machine was a Pentium 4 Xeon 2.4GHz server with 1GB of memory running Linux 2.6.8.1 with and without our AMP modifications.

Our first test application was `glimpse`, a text-retrieval tool. We use the `glimpseindex` command to index the Linux 2.6.8.1 kernel source files (sized 222MB). The execution times and number of blocks read from disk are shown in Figure 6. AMP shows significant performance improvement over the Linux page cache. Run time decreases by up to 13% and the number of blocks read from disk decreases by up to 43%, both when memory size is 224 MB.

We also ran the DBT3 database workload, in the same configuration as our *dbt3* trace. Figure 7 shows 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 by 9.6%, from 1091 seconds to 986 seconds. Disk read traffic decreased by 24.8% from 15.4 to 11.6 GB. Write traffic decreased by 6.5%, from 409 MB to 382 MB, probably due to lower cache contention.

4 Conclusion

We have presented AMP, an adaptive caching system that deduces information about an application's structure and uses it 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 a randomized partition size adaptation algorithm. Simulation confirms the effectiveness and robustness of the pattern detection algorithm. And measurement results on the Linux prototype are promising.

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] 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.

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

[4] 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)*, 2004.

[5] 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.

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

[7] Feng Zhou, Rob von Behreh, and Eric Brewer. Program context specific buffer caching with AMP. Technical report, CS Division, University of California, Berkeley, 2005.