Towards Self-Tuning Memory Management for Data Servers

Gerhard Weikum, Arnd Christian König
University of the Saarland, D-66041 Saarbrucken, Germany
{weikum,koenig}@cs.uni-sb.de

Achim Kraiss
Dresdner Bank AG, D-60301 Frankfurt, Germany
achim.kraiss@dresdner-bank.com

Markus Sinnwell
SAP AG, D-69185 Walldorf, Germany
markus.sinnwell@sap-ag.de

Abstract

Although today's computers provide huge amounts of main memory, the ever-increasing load of large data servers, imposed by resource-intensive decision-support queries and accesses to multimedia and other complex data, often leads to memory contention and may result in severe performance degradation. Therefore, careful tuning of memory mangement is crucial for heavy-load data servers. This paper gives an overview of self-tuning methods for a spectrum of memory management issues, ranging from traditional caching to exploiting distributed memory in a server cluster and speculative prefetching in a Web-based system. The common, fundamental elements in these methods include on-line load tracking, near-future access prediction based on stochastic models and the available on-line statistics, and dynamic and automatic adjustment of control parameters in a feedback loop.

1 The Need for Memory Tuning

Although memory is relatively inexpensive and modern computer systems are amply equipped with it, memory contention on heavily loaded data servers is a common cause of performance problems. The reasons are threefold:

- Servers are operating with a multitude of complex software, ranging from the operating system to database systems, object request brokers, and application services. Much of this software has been written so as to quickly penetrate the market rather than optimizing memory usage and other resource consumption.
- The distinctive characteristic and key problem of a data server is that it operates in multi-user mode, serving many clients concurrently or in parallel. Therefore, a server needs to divide up its resources among the simultaneously active threads for executing queries, transactions, stored procedures, Web applications, etc. Often, multiple data-intensive decision-support queries compete for memory.
- The data volumes that need to be managed by a server seem to be growing without limits. One part of this trend is that multimedia data types such as images, speech, or video have become more popular and are being merged into conventional-data applications (e.g., images or videos for insurance claims). The other

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

part is that diminishing storage costs have made it possible and business competition has made it desirable to keep extended histories of business events for marketing and other enterprise-critical decisions.

Like resource contention in general, memory contention can lead to performance disasters, also known as "thrashing". When memory becomes scarce in a multi-user server, performance does not degrade gracefully at all. For example, when too many join or grouping operations run in parallel and not all of them can be given sufficiently large in-memory hash tables or sort buffers, too much intermediate data needs to be written onto disk and read again too often. In this case, the memory bottleneck causes a dramatic increase of the disk load, leading to long queueing delays for disk accesses (or network queueing in other settings), and ultimately, such excessive queueing kills the application performance. The possible remedy of limiting the multiprogramming level (i.e., the maximum number of concurrently active operations) is not a real solution as this forces clients to wait until their requests are admitted for execution and may thus lead to poor user-perceived response times, too.

The critical nature of memory contention makes careful tuning of the server's memory management a necessity. As today's IT operation costs are increasingly dominated by the costs of human administration staff rather than hardware or software, automating tuning decisions will become mandatory in the long run. In particular, we need dynamically self-tuning methods that can adjust the policies and control parameters of a server's memory management to evolving workload characteristics in an online manner without human intervention.

This paper gives an overview on the state of the art of self-tuning memory management methods for data servers, as a critical piece along the path towards a new generation of zero-admin, trouble-free information systems [1]. The rest of the paper is organized as follows. Section 2 reviews approaches to traditional data caching, the aim being to maximize the cache hit ratio of a server. Section 3 extends the workload setting by adding long-running queries that require large amounts of private working memory. Section 4 extends the architectural model by considering how to exploit the distributed memory in a server cluster. Section 5 adds self-tuning prefetching techniques to the repertoire of memory management methods. Finally, Section 6 discusses how to exploit advanced forms of both prefetching and caching in a widely distributed system like the Internet. Section 7 concludes the paper with some general lessons learned.

2 Self-tuning Server Caching

The traditional usage of memory in a data server is for caching frequently accessed data to avoid disk I/O. Such a cache is typically organized to hold fixed-size pages (e.g., of size 32 KBytes) and is managed by a cache replacement policy that drops a page whenever a new page needs to be brought in. The goal of the cache manager is to maximize the *cache hit ratio*, i.e, the fraction of page accesses that can be served from the cache. The most widely used replacement policies are variations of the LRU (Least Recently Used) algorithm which selects the page whose last reference time is oldest among all currently cache-resident pages as the replacement victim.

LRU is a robust policy that performs well under a wide variety of workload characteristics. However, it is suboptimal with regard to specifically idiosyncratic, albeit fairly frequent page reference patterns:

- Sequential scans over a large set of pages which compete for memory with random accesses to individual pages. Plain LRU would rank pages that are freshly touched during a scan as worthy, although the probability for accessing them again in the near future is low.
- Random accesses to different page sets with highly skewed cardinalities. For example, LRU treats data and index pages as equally worthy when both exhibit the same access pattern, say for a batch of exact-match queries where index and data accesses alternate, although the access probability for an index page is much higher.

To overcome these deficiencies of plain LRU, researchers and system architects have developed, during the past decades, a number of tuning methods for managing a page cache. It turns out, however, that most of them are troublesome in that they are not fully self-tuning: either they rely on human intuition or experience for calibration, or they are geared only for relatively static workloads where access patterns remain stable over a long time period. We can categorize the various approaches as follows:

- The "Pandora" approach relies on explicit tuning hints from programs, either application programs of file servers or the query-processing layer of a database server. For example, a query-processing engine knows about sequential access patterns and can inform the cache manager so that pages referenced only once during a scan are attached to the low-priority end of the LRU chain [22]. However, this hint-passing approach is limited; trying to generalize it to other access patterns bears a high risk that "local" hints issued by individual operations are detrimental for global performance metrics (e.g., by monopolizing memory on behalf of a single operation). So hints open up both potential blessing and curse, like Pandora's box.
- The "Sisyphus" approach aims to tune the cache manager by partitioning the overall cache into separate "pools" and assigns different page classes to different pools [22]. This approach works very well for partitioning index versus data pages or discriminating pages from different tables. However, determining appropriate pool sizes and a proper assignment of page classes to pools is a highly delicate tuning task, and when the workload characteristics change significantly after some time, one may have to start the tuning all over again. So this is a real Sisyphus work for the administration staff.
- Finally, the "Sphinx" approach takes a completely different approach in that it abandons LRU and rather adopts a replacement policy based on access frequencies. Such an LFU (Least Frequently Used) policy would indeed be optimal for a static workload where pages have independent reference probabilities that do not vary over time. With evolving access patterns, however, one crucially needs some kind of aging scheme for reference counters to recognize when previously hot pages become cold and vice versa. For example, in practically used variants such as LRD (Least Reference Density) replacement [7], the reference counter of a page is decremented by a specific value once every so many references; so the aging is controlled by two parameters. Now the crux is to find appropriate values for such fine-tuning parameters, which may be as hard as it was for Oedipus to solve the Sphinx's cryptic riddles.

The state-of-the-art solution for truly self-tuning cache management follows the Sphinx approach, but avoids the delicate aging parameters by using an appropriate form of online statistics which has the aging automatically built-in. We coin this approach the "Nike" approach (to stay in Greek mythology); concrete implementations are the LRU-k algorithm [15] or its variant 2Q [11]. The LRU-k method is organized into three steps according to an *observe-predict-react* paradigm [24]:

- *Observation:* The method is based on keeping a limited form of each relevant page's reference history: the k last reference timepoints. Here "relevant" means all pages that are currently in the cache plus some more pages that are potential caching candidates. Determining the latter kind of relevant pages may be seen as a tricky fine-tuning problem again, but one can easily make a pragmatic and practically robust choice based on the five-minute rule [10]: the history of pages that have not been referenced within the last five minutes can be safely discarded from the method's bookkeeping.
- *Prediction:* The time interval between the k-th last reference of a page p and the current time gives us a statistical estimator for the reference probability of the page: having observed k, $k \geq 2$, references to p in the time period between t_k and now and assuming independent "arrivals" of page accessses, we can infer a page-specific access rate, also known as the page's *heat*, $heat(p) = \frac{k}{now - t_k}$, and the probability for accessing the page (at least once) within the next T time units is given by $1 - e^{-heat(p)T}$. It can be shown that it is optimal to rank pages according to these near-future access probabilities [16].
- *Reaction:* When space needs to be freed up in the cache, LRU-k replaces the page with the smallest value for the above (estimated) probability. For low-overhead implementation, it is sufficient to simply maintain the heat of the pages or merely the timepoints of the k-th last references, and determine the page with

the lowest heat or, equivalently, the oldest k-th last reference as a replacement victim. Such an implementation needs a priority queue for the relevant pages; so it requires logarithmic time for page-replacement decisions. An approximative variant, the 2Q algorithm, makes its decisions even in constant time [11].

This family of cache replacement algorithms can be easily generalized to cope also with variable-size caching granules, say documents rather than pages, as one would have to deal with, for example, in a Web server. Now the benefit of caching a document, as measured by the document heat, needs to be normalized by the occupied cache space. So we derive a benefit/cost ratio for each document as the quotient of document heat and document size, which has been coined the *temperature* of the document. Then, for caching decisions, documents are simply ranked by their temperature.

LRU-k and its variants cope well with a wide spectrum of page access patterns including mixes of sequential and random references and also patterns that evolve over time. They are completely self-tuning in that they neither need administrator help for fine-tuning nor hints from programs. Their core assets, *online statistics for tracking access patterns* and *probabilistic reasoning for predicting near-future accesses*, are indeed fundamental for memory management in general, as the discussion of the following sections will show.

3 Automatic Tuning of Cache and Working Memory

In addition to caching disk-resident data in memory, a data server needs to manage also working memory that is needed by long-running operations such as hash- or sort-based joins or video/audio streams. With a highly diverse workload, memory management should no longer focus on a single global performance metric such as the mean response time of all operations. Rather differences in the resouce consumption and response-time-criticality of different *workload classes* should be taken into account. As the system has no way of automatically inferring the relative importance of the various classes, a human administrator must explicitly specify class-specific performance goals, which can then be monitored and enforced automatically. For example, all medium-sized join operations of standard users could be one class, large joins issued on behalf of business executives could be another class, and the most common class would be "rifle-shot" exact-match queries; the first two classes need working memory and the last one operates on the server's shared page cache. Performance goals could then be upper bounds for the per-class mean response time or the 90th percentile of the class's response time distribution for a given throughput (i.e., class-specific arrival rates) to be sustained. The number of such classes would typically be small, maybe, five or ten, and it often makes sense to have an additional "free" class with all operations for which no goal is specified. The objective function of the server should then be to minimize the mean response time of the free class under the constraint that the response-time goals of all classes are met.

The mechanism for handling multiple workload classes is a conceptual partitioning of the available memory into *class-specific memory areas*; these include operation-type-specific working memory (e.g., hash tables) as well as a shared page cache or even multiple page caches dedicated to specific data partitions. The partitioning is merely conceptual, not physical; so a memory area can be shared by multiple workload classes. For ease of explanation, however, we assume in what follows that each class accesses only one memory area; for example, m-1 classes of decision-support queries operate on private working memory, and one class of exact-match queries makes use of the shared page cache. Further note that we still need a replacement algorithm like LRU-k within an area if the area serves as a page cache (as opposed to working memory). Approaches for automatically maintaining performance goals within this memory model can generally be described as a *feedback loop*, again consisting of three steps for observation, prediction, and reaction [4, 3, 20].

Observation: This step involves collecting online statistics about the response time of the various workload classes, e.g., using moving time-window averaging. The length of the observation window must be carefully chosen, for too small intervals may result in the system overreacting to statistical fluctuations, whereas with overly long intervals, the system is not sufficiently responsive to evolving changes in the workload characteristics. Therefore, the observation phase needs some form of calibration, but this can be accomplished automatically based on the ratio of the observed metric's mean value and variance.

Prediction: Whenever the performance goal of one class of operations is not satisfied, the server has to determine an appropriate re-adjustment of its memory usage, for example, increasing the working memory for a certain class of join operations and dynamically decreasing the size of the shared page cache. Assuming algorithms for the various operations (e.g., joins or sorting) that can cope with time-varying memory resources, this calls for predictions of how performance would change if the memory assignment were modified in a certain way. So we are interested in predicting the response time R_i of class i as a function of the sizes of different memory areas $M_1, \ldots M_m$ assigned to the various classes.

Unfortunately, deriving or even adequately approximating the function $R_i(M_1, \ldots, M_m)$ is very difficult. Prior approaches that aimed at a completely analytical, accurate solution have not had much practical impact. Rather a simpler and practically viable approach [4, 3, 20] is to model each R_i function approximatively as a linear function of its memory-area size M_i. Although this is a fairly crude approximation as far as absolute hit-ratio or response-time predictions are concerned, it is sufficient to drive the feedback loop for gradually approaching the optimum memory assignment. Based on this approximate model one can determine the minimum amount of memory M_i such that the predicted performance for class i will be acceptable.

Reaction: After the desirable sizes of memory areas have been determined, the appropriate adjustments have to be enforced by the server, by increasing the area of the class whose response-time goal has been violated and taking away memory from the "free" class. If no such excess memory can be reclaimed without violating another class's goal, the server is obviously in an overload situation. In this case, it would re-initiate the prediction phase under the new goal of finding a memory assignment that mimimizes the maximum goal violation among all classes, $max\{\frac{R_i}{G_i} \mid 1 \leq i \leq m\}$, where R_i is the response time and G_i the response-time goal of class i. In addition, one could introduce an admission control (i.e., a limitation of the multiprogramming level) for preventing such situations or making them unlikely; methods along these lines again build crucially on a prediction step similar to what we have sketched above (see, e.g., [8, 3]).

If it turns out, after another observation cycle, that the re-adjusted memory area for the goal-violating class is still not sufficiently large for achieving the goal, the entire feedback loop is iterated. This feedback loop is guaranteed to converge (quickly) to a stable state if the actual response-time functions (i.e., not the approximative linear ones) are concave [3]. Even for non-concave functions, the approach typically reaches a new stable state fairly quickly, as new feedback is incorporated in each iteration of the feedback loop.

4 Exploiting Distributed Memory

High-end data services are often implemented on server clusters with a (small) number of computers, each with its own memory, sharing disks through a high-speed network (e.g., fibre channel). A functionally equivalent architecture would be a collection of independent servers (each with its own disks) with all data replicated across all of them; this is attractive for read-only applications such as Web information services. In both cases, a data request arrives at one of the servers and will primarily be executed there. During the execution it is possible that the server has to fetch some data from disk although another server holds that data in its memory. This is the situation where data shipping can be beneficial: with current low-latency interconnect technology accessing the remote memory of another server is often significantly faster than performing the disk access.

Exploiting this option in a systematic manner amounts to a distributed caching algorithm, which essentially controls the dynamic replication of data objects (i.e., fixed-size pages or variable-size documents) in the caches of the underlying computers [6, 9, 14]. Two simple heuristics for this purpose are "egoistic" caching, where each server simply runs a local cache replacement algorithm such as LRU or LRU-k, and the other extreme, called "altruistic" caching. Egoistic caching views remotely cached data that is not locally cached as an opportunity, but does not actively aim to make this situation more likely. In fact, it may easily end up with situations where the hottest data is fully replicated in all caches with only little space left for the rest of the data. Altruistic caching, on the other hand, aims exactly at minimizing this replication by giving preference in the local cache replacement to data that is not already cache-resident at a different server. Obviously, this method needs some distributed bookkeeping about the global cache contents. With a high-bandwidth network this is an affordable overhead, and altruistic caching outperforms the egoistic heuristics. However, even the fastest interconnect may become congested under high load and then turns into an effectively-low-bandwidth network. So one needs some careful tuning for reconciling the two distributed caching approaches.

The key to such an online tuning is, once again, a mathematical cost model for assessing whether egoistic or altruistic behavior is preferable under the current workload and system setting. Such a model has been developed in [21] (see also [23] for a closely related approach), driven by online statistics about the local and global heat of data objects, and taking into account memory-access costs and detailed costs of both the network and the computers' disks including queueing effects. Then cache replacement decisions are essentially optimization problems: drop the page or document that contributes least to the system-wide caching benefit where benefit is proportional to the mean response time of data requests over all servers. It is crucial, however, that this optimization model can be evaluated online with low overhead, so that it can drive the online decisions of the local cache managers. To this end, the cost model is simplified and itself "tuned" in [21] by various approximations and lazy, threshold-driven evaluations of cost formulas, as well as lazy estimation of load parameters and dissemination of cache-contents bookkeeping data in the network.

The resulting algorithm always performs at least as good as the better one of the two simple heuristics and clearly outperforms the other one. Most importantly, this property holds even for evolving workloads and time-varying network utilization. As the underlying cost model includes disk queueing, the entire approach can even contribute to disk load balancing, for example, by caching a higher fraction of an overloaded disk's data in remote caches. As a further extension of this method and a generalization of the algorithm of Section 3, it is even possible to accomodate workload-class-specific response-time goals in a server-cluster architecture [20].

5 Integrating Speculative Prefetching with Caching

Caching reduces the overall disk I/O load and thus helps throughput, but since caching can usually not eliminate disk I/O completely, data-intensive operations may still exhibit long response times. To reduce response time, prefetching can be employed as an additional measure to mask the latency of the disk accesses by bringing the relevant data into memory already before it is explicitly requested. Prefetching pays off particularly well for high-latency data requests, for example, for large images or video fragments that reside on disk, or when the data resides in a tertiary-storage archive or needs to be fetched from another server over a high-latency WAN.

Prefetching generally leverages some anticipation of near-future data requests. In special cases such as sequential scans, exact knowledge about what and when to prefetch can be obtained. In full generality, however, a server would have to speculate about the near-future access patterns of ongoing operations or client sessions. Such speculation is necessarily stochastic in nature, so that prefetching is beneficial only with a certain (hopefully high) probability. The server has to be careful so as not to initiate too much speculative prefetching, as this could possibly replace other, more worthy data from the cache and affect performance adversely. Reconciling prefetching and caching therefore involves quantitative benefit-cost considerations to throttle overly aggressive prefetching (see, e.g., [17]). To this end, a self-tuning method is once again called for.

A natural idea is to estimate near-future access probabilities on the basis of the stationary heat statistics that an LRU-k-like cache manager would keep anyway (see Section 2), or the corresponding temperature values when data units have variable size. Such a method, coined *temperature-based vertical data migration* in [12, 13], keeps a list of the top-temperature non-cached data units and considers their prefetching in descending order of temperature, but the prefetching is initiated only when the corresponding document's temperature exceeds the temperature of the document(s) that would need to be dropped from the cache instead.

When the latencies of fetching non-cached documents vary significantly, especially when tertiary storage is involved where "offline volumes" (i.e., tapes or (magneto-)optical platters) may first have to be mechanically loaded into a drive, the above benefit-cost consideration should be further refined by explicitly considering the fetch costs of non-cached documents (see also [18, 19] for similar considerations regarding cache replacement alone in settings like data warehouses and Web proxy servers). With a planning horizon of length T (e.g., the next second for prefetching from secondary storage or the next minute for tertiary storage), the expected number of accesses to document d within time T is $N_{spec}(d) = heat(d) \times T$. Then the *benefit* of prefetching document d is the product $\frac{N_{spec}(d)}{size(d)} \times fetch\_time(d, v)$, where $fetch\_time(d, v)$ is the estimated time for accessing d on its "home location" v where v can be secondary storage, an online volume in tertiary storage, or an offline volume, and the division by $size(d)$ is a normalization per cache space unit. The *cost* of this prefetching, on the other hand, is essentially the cost for accessing the corresponding cache replacement victim d' on its slower home location: $\frac{N_{spec}(d')}{size(d')} \times fetch\_time(d', v')$. Prefetching is considered beneficial only if its benefit exceeds its cost.

A cache manager that operates under this regime is self-tuning as it can automatically infer when aggressive prefetching should be throttled and when speculative prefetching pays off. The formulas for estimating the benefit and cost of a potential prefetching step can be made even more accurate by incorporating scheduling and queueing effects for the involved storage devices [13]. The method has been shown to be highly effective in experiments, especially for tertiary-storage archives, but it is attractive also for speculative prefetching from secondary-storage disks. Note that its overhead is low, comparable to the LRU-k bookkeeping.

6 Self-tuning Caching and Prefetching for Web-based Systems

The temperature-based prefetching method of the previous section has striven for a practically viable compromise between achieving a high prefetching benefit and keeping its bookkeeping overhead acceptably low. In particular, the bookkeeping metadata must not consume more than a trivial amount of cache space itself, and the time for making prefetching and cache replacement decisions should be at least an order of magnitude less than the time for a non-cached data access. When servers are accessed over the Web or use tertiary storage for infrequently requested but still "alive" data, the WAN or the tertiary storage incur a very high latency (and possibly also data transfer time, especially when the WAN is congested). This may call for an even higher fraction of data for which near-future accesses are anticipated to be downloaded to the client or a proxy server near the client in a timely manner (i.e., before the client explicitly asks for that data). So the stochastic prediction for near-future requests should be more "aggressive" but also needs to be more "accurate", as the cost of speculative wrong decisions also becomes higher. On the other hand, with a latency of several seconds that one aims to mask, one can afford both more space and more computation time for the stochastic predictions themselves.

A "richer" class of models for capturing reference patterns and predicting near-future accesses are Markov chains. Each active client connected to a proxy server can be viewed as a stochastic process that proceeds through states such that the current state corresponds to the document that the client has accessed last. State transitions then reflect the co-access patterns induced by the client. In a (first-order) Markov chain the probability for accessing document j right after document i does not depend on the prior history of the client's session and can thus be viewed as constant, unconditional probability p_{ij} that can be estimated by keeping online statistics. Then, knowing the current state of a client, one can compute the probability of a given document being requested within the next so many steps by a standard mathematical apparatus for the transient analysis of the Markov chain. These predictions would then drive the proxy server's prefetching and caching algorithm and/or the management of the data server's hierarchical storage (assuming tertiary storage is used).

Markov-chain-based algorithms have been intensively investigated for prefetching and caching (e.g., [5, 2]), but have previously overlooked the critical interdependencies between these two issues. Moreover, the prior methods have focused on the reference pattern of a single client and assumed a discrete time (i.e., simply counted access steps). In contrast, the recently developed *McMIN* method (Markov-chain based Migration for Near-line

Storage) [12, 13] takes into account the different interaction speeds of clients by using a continuous-time Markov chain (CTMC) for each client, where the residence times of states reflect the time that the client needs to issue its next request once it has obtained the document that corresponds to the current state. In Web-based access to a digital library, for example, bibliographies, abstracts, contents tables, results from search engines, or images may be browsed quickly whereas the user may spend much longer time on a hypertext document with a mathematical formula before proceeding along one of the outgoing links. The CTMC captures this variability; its transient analysis is a bit more involved than that of a discrete-time Markov chain. As shown in [12], it is possible to approximatively compute both the expected number of near-future accesses to a document d, $N_{spec}(d)$ (see Section 5), by appropriate precomputations and truncating negligible paths in the underlying traversal of the Markov chain. Moreover, both of these values can be aggregated over multiple CTMC models, one for each active client session, and the "arrivals" of new client sessions as well as "departures" of terminating sessions can be incorporated, too. Once the near-future access prediction is enhanced by the CTMC-based stochastic analysis, the same kind of benefit/cost assessment that we outlined in the previous section can be applied again. By taking into account the current states of sessions and the context-dependency of reference patterns, the effectivity of the prefetching is significantly increased, though [13].

7 Lessons Learned

This overview paper has sketched a suite of self-tuning methods for memory management, geared for centralized, high-speed interconnected, and widely distributed data servers. These methods are, to a large extent, complementary and may co-exist in a concrete system setting; for example, LRU-k as page replacement algorithm is a building block in goal-oriented distributed caching or temperature-based prefetching. The various tuning problems have been viewed as online optimization problems with dynamically evolving, statistically fluctuating input parameters. Consequently, the common paradigm that all approaches follow is that of an observe-predict-react cycle. Observation requires *online statistics*, prediction needs *mathematical models*, and reaction can often be organized as a *feedback loop*. These are the fundamental assets towards self-tuning memory management. For practically viable methods, however, one needs additional careful considerations on the following tradeoffs:

- The space needed for online statistics must be carefully controlled. In most situations, it is mandatory that the statistics are kept in memory for fast lookup. All methods presented in this paper can therefore be calibrated to adjust the amount of memory used for statistical load tracking. The actual setting of this calibration parameter has been driven by pragmatic considerations, however. More insight is needed on how much memory should be invested in the bookkeeping in order to auto-tune the remaining memory most intelligently, optimizing the overall use of memory.
- The complexity of the mathematical models used for predicting near-future accesses has ranged from lookups in the online statistics (e.g., in LRU-k) all the way to the transient analysis of Markov chains. Also, the metrics to be predicted have progressed from access probabilities to more complex notions of benefit and cost. So the CPU time overhead of predictions may be a critical factor, too. All methods have been devised so that this overhead can be controlled in a flexible manner, but one would also wish to have design guidelines for when it pays off to use more advanced prediction models.
- The reaction phase needs to adjust the memory usage according to the predicted reference patterns. This bears the risk of overreacting and ultimately ending up with oscillating, unstable adjustments of cache contents, memory-area sizes, etc. The presented algorithms have been designed so as to be robust in this regard. In particular, feedback-driven iterative adjustments have been shown to be well-behaved in this regard. Most strikingly, feedback-based methods work well even though the dependency of response time on the memory adjustment is merely approximated in a crude manner based on prior observations, without an explicit causal model. It would be desirable to obtain better insight in this issue from a control-theoretic viewpoint and be able to *guarantee* (fast) convergence after shifts in the workload patterns.

Much of this work is mature enough to be adopted by commercial data servers, and we believe that self-tuning algorithms will penetrate products and contribute towards zero-admin, trouble-free servers. With the explosion of data collection and generation on the Internet and the advent of mobile "gizmos" equipped with digital camera, speech recorder etc., distributed memory management and storage management in general will remain as great challenges, though. Individuals may record everything they hear or see [1] (e.g., during business meetings or vacation trips), most of which would be uploaded and ultimately archived on servers via wireless communication. One would ultimately expect that all data is virtually ubiquitous at acceptable speed and cost.

References

[1] Bernstein, P.A., et al., The Asilomar Report on Database Research, ACM SIGMOD Record 27(4), 1998.

[2] Bestavros, A., Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time in Distributed Information Systems, Data Engineering Conf., 1996.

[3] Brown, K., Carey, M., Livny, M., Goal Oriented Buffer Management Revisited, SIGMOD Conf., 1996.

[4] Chen, C.M., Roussopoulos, N., Adaptive Database Buffer Allocation Using Query Feedback, VLDB Conf., 1993.

[5] Curewitz, K.M., Krishnan, P., J.S. Vitter, Practical Prefetching Via Data Compression, SIGMOD Conf., 1993.

[6] Dahlin, M.D., Wang, R.Y., Anderson, T.E., Patterson, D.A., Cooperative Caching: Using Remote Client Memory to Improve File System Performance, Symp. on Operating Systems Design and Impl., 1994.

[7] Effelsberg, W., Haerder, T., Principles of Database Buffer Management, ACM TODS 9(4), 1984.

[8] Faloutsos, C., Ng, R.T., Sellis, T., Flexbile and Adaptable Buffer Management Techniques for Database Management Systems, IEEE Transactions on Computers 44(4), 1995.

[9] Franklin, M.J., Client Data Caching, Kluwer Academic Publishers, 1996.

[10] Gray, J., Graefe, G., The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb, ACM SIGMOD Record 26(4), 1997.

[11] Johnson, T., Shasha, D., 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm, VLDB Conf., 1994.

[12] Kraiss, A., Weikum, G., Vertical Data Migration in Large Near-Line Document Archives Based on Markov-Chain Predictions, VLDB Conf., 1997.

[13] Kraiss, A., Weikum, G., Integrated Document Caching and Prefetching in Storage Hierarchies Based on Markov-Chain Predictions, VLDB Journal 7(3), 1998.

[14] Leff, A., Wolf, J.L., Yu, P.S., Efficient LRU-based Buffering in a LAN Remote Caching Architecture, IEEE Transactions on Parallel and Distributed Systems 7(2), 1996.

[15] O'Neil, E.J., O'Neil, P.E., Weikum, G., The LRU-K Page Replacement Algorithm for Database Disk Buffering, SIGMOD Conf., 1993.

[16] O'Neil, E.J., O'Neil, P.E., Weikum, G., An Optimality Proof of the LRU-K Page Replacement Algorithm, Journal of the ACM 46(1), 1999.

[17] Patterson, R.H., Gibson, G.A., Ginting, E., Stodolsky, D., Zelenka, J., Informed Prefetching and Caching, ACM Symp. on Principles of Operating Systems, 1995.

[18] Scheuermann, P., Shim, J., Vingralek, R., WATCHMAN: A Data Warehouse Intelligent Cache Manager, VLDB Conf., 1996.

[19] Shim, J., Scheuermann, P., Vingralek, R., Proxy Cache Design: Algorithms, Implementation, and Performance, IEEE Transactions on Knowledge and Data Engineering, 1999.

[20] Sinnwell, M., König, A.C., Managing Distributed Memory to Meet Multiclass Workload Response Time Goals, Data Engineering Conf., 1999.

[21] Sinnwell, M., Weikum, G., A Cost-Model-Based Online Method for Distributed Caching, Data Engineering Conf., 1997.

[22] Teng, J.Z., Gumaer, R.A., Managing IBM Database 2 Buffers to Maximize Performance, IBM Systems Journal 23(2), 1984.

[23] Venkataraman, S., Livny, M., Naughton, J.F., Remote Load-Sensitive Caching for Multi-Server Database Systems, Data Engineering Conf., 1998.

[24] Weikum, G., Hasse, C., Moenkeberg, A., Zabback, P., The COMFORT Automatic Tuning Project, Information Systems 19(5), 1994.