

Low-overhead decision support for dynamic buffer reallocation

Karsten Schmidt · Sebastian Bächle

© Springer-Verlag 2011

Abstract Effective I/O buffering is a performance-critical task in database management systems. Accordingly, systems usually employ various special-purpose buffers to align, e.g., device speed, page size, and replacement policies with the actual data and workload. However, such partitioning of available buffer memory results in complex optimization problems for database administrators and also in fragile configurations which quickly deteriorate on workload shifts. Reliable forecasts of I/O costs enable a system to evaluate alternative configurations to continuously optimize its buffer memory allocation at runtime. So far, all techniques proposed for the prediction of buffer performance focus solely on hit ratio gains for increased buffer sizes to identify buffers which promise the greatest benefit. These approaches, however, assume that their forecast allows to extrapolate the effect for buffer downsizing, too. As we will show, this comes along with a severe risk of wrong tuning decisions, which may heavily impact system performance. Thus, we emphasize the importance of reliably forecasting the penalty to expect for shrinking buffers in favor of others. We explore the use of lightweight extensions for widely used buffer algorithms to perform on-the-fly simulation of buffer performance of smaller and larger buffer sizes simultaneously. Furthermore, we present a simple cost model and demonstrate how to compose these concepts into a self-tuning component for dynamic buffer reallocation.

Keywords Self-tuning · Buffer reallocation · Forecasting · Replacement algorithms

1 Introduction

Dynamic database management gained a lot of attention and visibility during recent years and led to various self-tuning approaches. As I/O reduction is one of the most important aspects, automatized buffer memory management has always been one of the building blocks for (self-)tuning of database systems. Data placement decisions but also variations in access patterns, page sizes, access speed, read/write characteristics, or prices of storage devices suggest the support of multiple buffers to optimally exploit existing I/O bandwidth. Memory partitioning, however, frequently entails memory waste, because some buffers may be underused while others are overused. Here, only continuous monitoring of system performance may assure adequate usage of the total memory budget and regular adjustment of buffer allocations at runtime, thereby enabling minimization of waste.

The decision when and which buffers have to be resized requires a cost-based model together with buffer techniques (i.e., page mapping, propagation algorithm) that are self-tunable at runtime. The quality of a decision depends on the cost model itself and the accuracy of forecasts. However, database buffers typically scale non-uniformly (i.e., in a non-linear fashion) and simple extrapolations of current performance figures can easily lead to wrong assumptions. In the worst case, the redistribution of buffer memory results in unintended buffer sweeps followed by excessive I/O thrashing, which again increases the time to pour oil on troubled waters. In our opinion, self-tuning components should therefore follow a strict “Don’t be evil” policy.

Most tuning approaches aim at maximum speedup, i.e., they focus on the identification of the greatest profiteer when

This is an extended version of the paper “Lightweight Performance Forecasts for Buffer Algorithms” selected for the special CSRD issue *Best papers BTW 2011* [1].

K. Schmidt (✉) · S. Bächle
Dept. of Computer Science, University of Kaiserslautern,
Kaiserslautern, Germany
e-mail: kschmidt@cs.uni-kl.de

more buffer memory can be assigned. Accordingly, they usually shift memory from buffers having low I/O traffic and/or low potential for performance gains to more promising ones. We believe that a sole focus on buffer growth is dangerous, because the risk of wrong decisions comes mainly from the inaccuracy of forecasts concerning smaller buffers. Once a buffer is shrunk too much, it may cause a lot of I/O and, in this way, also affect the throughput of all remaining buffers. Thus, reliable estimations for buffer downsizing are obviously as important as estimations for buffer upsizing. Good forecast quality is further urgently needed in dynamic environments which have to cope with many or intense workload shifts. Here, too cautious, i.e., too tiny adjustments, even when they are incrementally done, are not good enough to keep the system in a well performing state. Reliable forecasts help to justify more drastic reconfigurations which may be necessary to keep up with workload shifts.

1.1 Forecast of buffer behavior

Proposed forecast models for the performance of a resized buffer can be divided into two groups: The first group uses heuristics-based or statistical indicators to forecast buffer hit ratios, whereas the second group is based on simulation. Using heuristics-based approaches, the forecast quality is hard to determine. As a consequence, their use comes with the risk of wrong tuning decisions which may heavily impact system performance. Simulation-based approaches allow trustworthy estimations, but usually limited to the simulated buffer size. Outside already known or simulated ranges, hit ratios may change abruptly. For this reason, we need forecasts for *growing and shrinking* buffers.

The performance of a buffer does not scale linearly with its pool size, because mixed workloads containing scans and random I/O can cause abrupt jumps in the hit-ratio trend line as illustrated in Fig. 1. These jumps may also lead to differing speed-ups for varying buffer sizes, which again may cause wrong assumptions and decisions.

Performance prediction is always based on information gathered by monitoring, taking samples or (user) hints into account. Hit/miss ratios are the standard quality metrics for buffers, because they are cheap and express the actual goal of buffer use: I/O reduction. Unfortunately, they are useless for performance forecasts, i.e., they even do not allow to make simple extrapolations for growing or shrinking buffer sizes. To illustrate this fact, let us assume the following scenario for a given buffer size of 5 and LRU-based replacement. At the end of a monitoring period, we observed 5 hits and 10 misses. At least two different access patterns may have led to these statistics:

Scenario 1: 1, 2, 3, 4, 5, 1, 1, 1, 1, 1, 6, 7, 8, 9, 10, ...

Scenario 2: 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, ...

In the first scenario, 5 hits are attributed to repeated accesses of page 1, whereas, in the second scenario, the hits are attributed to 5 different pages (1, 2, 3, 4, 5). For the same scenarios and a buffer of size 2, we get completely different hit (h) and miss (m) statistics:

Scenario 1: $m, m, m, m, m, m, h, h, h, h, m, m, m, m, m, \dots$

Scenario 2: m, m, \dots

Obviously scenario 1 obtains a better hit rate with 4 hits to 11 misses than scenario 2 without any hit. If we increase the buffer instead to hold 6 pages in total, the picture turns again:

Scenario 1: $m, m, m, m, m, h, h, h, h, h, m, m, m, m, m, \dots$

Scenario 2: $m, m, m, m, m, h, h, h, h, h, m, h, h, h, h, \dots$

Now we observe 5 hits to 10 misses for scenario 1 and 9 hits to 6 misses for scenario 2. This example shows that hit/miss numbers or page/benefit metrics do not allow for correct extrapolations, because the order of page requests and the hit frequency distribution are important. Thus, self-tuning relies on monitoring and sampling of data where current buffer use is taken as an indicator for the future. Information relevant for resizing forecasts such as re-use frequencies, working set size, or noise generated by scans cannot be expressed in single numbers.

Instead, the ideal starting point for buffer forecasts is the replacement algorithm used for a buffer. Its statistics incorporate a lot more information about these relevant aspects than any other performance marker. Today, substantial research has already been performed to develop adaptive replacement algorithms, hence, it is safe to assume that such algorithms are operating “optimally” for the available memory. The question is now how to leverage this implicit

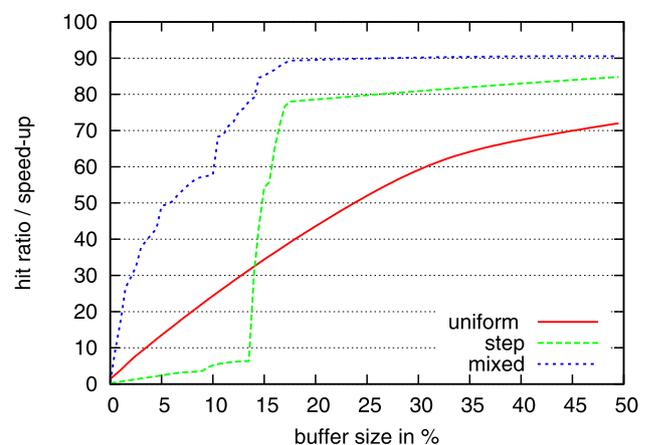


Fig. 1 Buffer speed-up trend for different access patterns

knowledge for performance forecasts. As we will demonstrate in the remainder of this paper, it is difficult but not impossible to get reliable estimates for buffer downsizing. In combination with already known simulation methods for the estimation of buffer upsizing, we can then build a lightweight framework for dynamic buffer management.

1.2 Related work

Optimal buffer management has been a key aspect in database system research since the very early days. Thus, various aspects such as the underlying disk model, search strategies within a buffer, replacement algorithms, concurrency issues and the implications of the page layout have been intensely studied [6]. Nevertheless, the complexity of buffer management did not allow to distill an optimal configuration for all different kinds of workloads and system environments. Instead, self-tuning mechanisms were explored to resolve performance bottlenecks at runtime.

One early self-tuning approach hints at specific access patterns like scans or index traversals to the buffer to optimize victim selection [7]. This allows to outperform standard LRU-based algorithms but addresses only a single aspect of dynamic buffer management. In [11], the authors give a theoretical base for the combined analysis of buffer sizing decisions and the influence of access patterns. Reference [4] models buffer load balancing as a constrained optimization problem and investigates the application of control theory and optimization theory.

In [14], control theory, runtime simulation, and cost-benefit analysis are integrated into a self-tuning framework. The presented forecast technique SBPX serves also as our baseline and is introduced in detail in Sect. 2. Some heuristic forecast techniques are presented in [3, 9]. The analytical work in [15] derives an equation to relate miss probability to buffer allocation. Finally, [5] proposes a brute-force step-by-step approach to determine the optimal configuration for an entire DBMS.

1.3 Contribution

In this work, we study two major prerequisites for self-tuning buffer memory allocation: cost determination and decision making. As the main objective of buffer tuning is I/O reduction and main memory management, decisions based on I/O costs are required to efficiently distribute available memory among all buffer pools. In particular, we look at overhead and quality for buffer undersizing and oversizing forecasts to estimate I/O costs for alternative configurations.

We present ideas to integrate low-overhead forecast capabilities for several common buffer algorithms and assess their feasibility in experiments. Furthermore, we show how

these forecasts can be used for nearly riskless self-tuning decisions. Eventually, a short evaluation is revealing prospects of simulation-based buffer tuning as well as its limitations.

The remainder of this paper is organized as follows: Sects. 2 and 3 discuss forecast techniques for buffer upsizing and downsizing, respectively. In Sect. 4, we present a decision model for a self-tuning component. The results of our experiments are shown in Sect. 5. Finally, Sect. 6 concludes the paper.

2 Forecast of buffer upsizing

The obvious way of accounting I/O costs for alternative buffer sizes is to fully simulate each of them for the same page reference string, i.e., page request sequence. Of course, a simulation of the propagation behavior for page numbers is sufficient; the actual payload data need not be kept in memory. Nevertheless, this approach requires additional data structures, such as hash maps for lookup, lists for the replacement algorithm, and virtual pages. Moreover, each buffer request has to be processed multiple times, i.e., page lookup and replacement maintenance for each simulated configuration. Obviously, the overhead of such a solution is prohibitive. In contrast, cheaper solutions may be less accurate, but still achieve meaningful results for resizing decisions.

Our buffer self-tuning refinements are inspired by the SBPX framework [14], which approximates the benefit of a larger buffer through “buffer extension”. This extension is simply an overflow buffer for the page identifiers of the most recently evicted pages. The overflow buffer must, of course, have its own strategy for victimization. The authors of SBPX recommend here a strategy “similar to that of the actual buffer pool” [14].

When a page miss in the actual buffer occurs, the extension checks if the page identifier is found in the overflow buffer, i.e., if the page would have been present in a larger buffer. In that case, we can account a “savings” potential for upsizing. Further, we must now maintain the overflow buffer. The page identifier of the actual evicted page is promoted to the overflow buffer, which in general requires to evict another page identifier from the overflow buffer. This replacement is not exactly the same as a real miss in the simulated larger buffer. The identifier of the requested page causing the miss could have been present in the larger buffer. In the course of continuous requests, however, also a larger buffer must evict pages. Thus, a replacement in the overflow buffer can be regarded as a “delayed” replacement effect. In the case of a page hit in the actual buffer, no further book-keeping is required, because the locality principle suggests that the replacement strategy in a larger buffer holds a superset of the pages present in a smaller one. Listing 1 shows a sketch of the modified page fix routine.

Listing 1 Modified page fix algorithm for upsize simulation

```

1 Frame fix(long pageNo) {
2   Frame f = mapping.lookup(pageNo);
3   if (f != null) {
4     strategy.refer(f); // update replacement strategy
5     ... // and statistics
6   } else {
7     Frame of = overflowMapping.lookup(pageNo);
8     if (of != null) {
9       overflowMapping.remove(of.pageNo);
10      ... // update overflow hit statistics
11    } else {
12      of = overflowStrategy.victim();
13      overflowBuffer.remove(of.pageNo);
14      ... // update overflow miss statistics
15    }
16
17    Frame v = strategy.chooseVictim();
18    strategy.copyStateTo(overflowStrategy);
19
20    v.copyStateTo(of);
21    // transfer page identifier to overflow
22    overflowMapping.put(of.pageNo, of);
23
24    mapping.remove(v.pageNo);
25    ... // replace page in frame v
26    strategy.referAsNew(v);
27    // update replacement strategy
28    ... // and statistics
29    mapping.put(pageNo, v);
30  }
31  return f;
32 }

```

The problem of this approach is that replacement decisions for two separate buffers in combination are not necessarily the same as for a single large buffer. Thus, the forecast quality of upsize simulations depends on one aspect: When a page is evicted from the actual buffer and promoted to the overflow area, we must be able to transfer “state” information (e.g., hit counters, chain position, etc.) from the actual replacement strategy into the overflow strategy (lines 17 and 20). Otherwise, the overflow strategy behaves differently.

3 Forecast of buffer downsizing

As shown above, knowledge about the performance gain through a larger buffer is useful to determine the greatest profiteer of more memory among several buffers. However, the question for the buffer(s), which may be safely shrunk without suffering from severe penalties, remains unanswered. The authors of SBPX extrapolated downsizing costs as the inverse of savings potential gained through upsize [14]. For buffer sizes close to the unknown (!) borders of working set sizes, however, this bears the risk of wrong decisions. Therefore, we developed a simple mechanism to find out if page hits would have been also page hits in a smaller buffer. In combination, the SBPX technique allows us now to determine which buffer profits the most from ad-

Listing 2 Modified page fix algorithm for downsize simulation

```

1 Frame fix(long pageNo) {
2   Frame f = mapping.lookup(pageNo);
3   if (f != null) {
4     if (!f.hotSet) {
5       Frame v = strategy.chooseHotSetVictim();
6       f.hotset = true; // swap frame to hotset
7       v.hotset = false;
8       strategy.swapHotset(f, v);
9       ... // update simulated statistics
10    }
11    strategy.refer(f); // update replacement strategy
12    ... // and statistics
13  } else {
14    Frame v = strategy.chooseVictim();
15    mapping.remove(v.pageNo);
16    ... // replace page in frame v
17    if (!v.hotset) {
18      Frame hv = strategy.chooseHotSetVictim();
19      hv.hotSet = false; // swap frame to hotset
20      v.hotSet = true;
21      strategy.swapHotset(f, v);
22    }
23    strategy.referAsNew(v);
24    // update replacement strategy
25    ... // and statistics
26    mapping.put(pageNo, v);
27  }
28  return f;
29 }

```

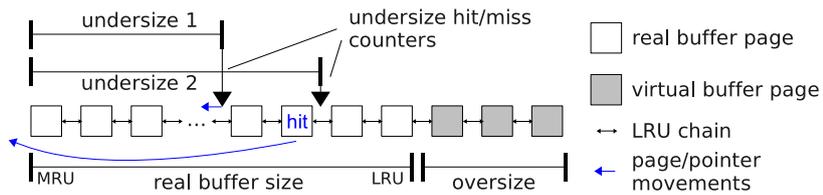
ditional memory, while our approach helps us to determine which buffer suffers least from downsizing.

The goal of buffer replacement algorithms is the optimized utilization of data access locality, i.e., to keep the set of the currently hottest pages that fits into memory. Accordingly, a small buffer is assumed to keep an “even hotter subset” of the pages that would be present in the actual buffer. Based on this assumption, we denote a subset of the pages in a buffer of size n as $hotset_k$, if it would be kept in a smaller buffer of size k . The key idea of our approach is to keep track of this hotset during normal processing. When a page is found in the buffer and belongs to the hotset, it would have been a hit in the smaller buffer, too. However, if a requested page is in the current buffer but not in the hotset, the smaller buffer would need to evict another page, which must be, of course, part of the current hotset and load the requested page from disk. Here, we only have to maintain the hotset. The page that would have been evicted from the smaller buffer is removed from the hotset and the requested page is added to the hotset. Each swap is accounted as a page miss for the simulated smaller buffer.

Of course, a page miss in the current buffer would also be a page miss in a smaller buffer. Accordingly, we have to select a replacement victim for both the current buffer and the (simulated) smaller buffer. The real victim page is now replaced with the new page and swapped with the virtual victim of the smaller buffer into the hotset. The modified page fix algorithm is shown in Listing 2.

Note that a real replacement victim is generally not expected to be part of the current hotset, because this would

Fig. 2 LRU-based buffer simulation with overflow extension



imply that the replacement strategy evicts a page more recently accessed. In some algorithms, however, such counter-intuitive decisions might be desired, e.g., to explicitly rule out buffer sweeps through large scans. Then, we must not maintain the hotset at all.

Obviously, the overhead of this approach is very small. We only need a single bit per buffer frame to flag the hotset membership and must determine a swap partner, when a new page enters the hotset. Furthermore, the simulation does not influence the quality of the current buffer, i.e., the strength of the replacement strategy is fully preserved. As said, the choice of the hotset victim is dependent on the used replacement strategy to reflect the behavior of the strategy in a smaller buffer correctly. In the following, we will investigate hotset victim determination for five popular families of replacement algorithms. In particular, we want to know if it is possible to predict replacement decisions for a smaller buffer based on the implicit knowledge present.

3.1 LRU

The LRU algorithm embodies a very simple, yet effective replacement strategy. It evicts always the *least recently used* page from a buffer. Typically, it is implemented as a doubly-linked list as shown in Fig. 2.

On request, a page is simply put to the head, i.e., MRU position of the chain. Thus, LRU finds its replacement candidate always at the tail, i.e., LRU position. Accordingly, the first k pages of the LRU chain in a larger buffer of size n are identical with the k pages in the simulated smaller buffer of size k and the hotset victim page is found at the k -th position from the head. The overhead of pointer dereferencing to position k can be avoided with a marker pointer (called *hotset LRU*), which is cheap to maintain. Hence, the hotset victim is guaranteed to be identical to the victim as in the smaller buffer and the simulation is fully precise. Evidently, the simplicity of LRU even allows to easily simulate at the same time the effects when the current buffer would be reduced to different smaller sizes, which is especially useful for precise step-wise tuning decisions. It is sufficient to place a marker at each desired position.

3.2 LRU-K

The LRU-K algorithm [13] follows a more general idea of LRU and takes the last K references of a page into ac-

Listing 3 LRU-K hotset victim selection

```

1 Frame chooseHotSetVictim() {
2     long min = t;
3     long minLast = Long.MAX_VALUE;
4     Frame v = null;
5     for (int i = 0; i < pages.length; i++) {
6         Frame p = pages[i];
7         History h = p.history;
8         if ((p.hotSet) && (t - last > CIP)) {
9             long last = h.last;
10            long dist = h.vector[k - 1];
11            if ((dist < min)
12                || ((dist == min) && (last < minLast))) {
13                victim = p;
14                min = hist.vector[k - 1];
15            }
16        }
17    }
18    return v;
19}

```

count. By doing so, it is “scan-resistant” and less vulnerable to workloads where frequently re-used pages mix with those having hardly any rereference. For each page, LRU-K maintains a history vector with the last K references and the timestamp of its last reference. Furthermore, history vectors of already evicted pages are retained for re-use if an evicted page is requested again within the so-called *retained information period (RIP)*. The replacement victim is only searched among those pages that have been buffered for at least a predefined *correlated reference period (CIP)*. The rationale behind this idea is to prevent a drop of pages immediately after their first reference. For further details on *CIP*, history maintenance, etc., we refer to the original paper.

The victim page is determined by the maximum backward K -distance, i.e., the page with the earliest reference in the history vector. Thus, although implemented differently, LRU-K behaves for $K = 1$ the same as LRU. The hotset victim is chosen accordingly as shown in Listing 3. Note that implementations of LRU-K usually maintain a search tree for that. For simplicity, we present here the modification of the unoptimized variant as in the original paper.

Due to the history update algorithm described in [13], more than one victim candidate can exist. This could become a problem for our simulation, because a real buffer might choose a different victim than simulated. Therefore, we simply evict the candidate with the least recent reference (line 12). As the timestamp of the last access is unique, our simulation will be accurate here. Instead, the choice of *RIP* turns out to become a problem. If the garbage collection for history entries is not aligned, pages that re-enter the

Listing 4 GCLOCK hotset victim selection

```

1 Frame chooseHotSetVictim() {
2   Frame v = null;
3   int h = clockHand;
4   for (int i = 0; i < size; i++) {
5     Page p = circle[(++h % size)];
6     if (p.hotSet) {
7       if (p.count == 0) {
8         return v;
9       } else if ((v == null) || (p.count < v.count)) {
10        v = p;
11      }
12    }
13  }
14  return v;
15 }

```

smaller buffer will be initialized differently than in simulation, which may affect future replacement decisions.

3.3 GCLOCK

The third strategy is GCLOCK [12], which stands for *generalized clock* algorithm. Like LRU-K, it takes the reference history of a page into account. In contrast to LRU-K, however, it is likely to degrade through scans but can be implemented with less computational and space overhead. The buffer itself is modeled as a circle of buffer frames, i.e., the clock. Each frame also maintains a simple reference counter, which is incremented for each reference to that specific page. For victim selection, the “clock hand” circles over all frames and decrements the reference counters. The clock hand stops at the first frame where the reference counter drops below zero. So, frequently referenced pages remain longer in the buffer, because they have higher reference counts.

The determination of a hotset victim is straightforward: We simply have to circle over the frames and look for the first hotset page whose reference counter would first drop below zero. Obviously, this is the page with the minimum reference counter. The algorithm is sketched in Listing 4.

Again, this only approximates the behavior of a smaller buffer with GCLOCK. There are two reasons: First, the angular velocity of the clock hand in a smaller buffer is higher because there are less frames. Second, the circular arrangement of buffer frames makes the algorithm inherently dependent on the initial order. Thus, victim selection is not only a matter of the page utilization, but also a matter of clock-hand position and neighborhood of frames. Using a second clock hand (i.e., pointer) walking solely over the hotset frames is necessary to address differing round trips. However, swapping of frame positions when the hotset is maintained would impact behavior of GCLOCK in the actual buffer—a circumstance, we want to avoid. To improve forecast quality, we implemented the smaller circle, i.e., the hotset, with forward pointers for hotset pages that point to the logical next one. In case of swapping (see lines 8 and 21

in Listing 2), only the forward pointer and a hotset counter for that page need to be maintained. In Sect. 5, we will show that these minor efforts can lead to almost perfect estimations.

3.4 2Q

The 2Q algorithm [8] is a simplified way of imitating LRU-2, which is noted for delivering good hit ratios but often poor performance due to its complex algorithm. In essence, 2Q is a combination of FIFO and LRU. On the first reference, 2Q places a page in a FIFO queue (denoted $a1$). The first re-reference of a page in the $a1$ queue promotes it to the LRU chain (denoted am). The effect of these two “stages” is that only hot pages are promoted to the LRU chain, which tends to keep cold pages longer than necessary. These cold pages, i.e., pages that are accessed only once within a longer time period are now dropped earlier by the FIFO queue. An extended version of 2Q splits the FIFO queue to keep track of rereferences to pages evicted from the FIFO queue [8]. The effect is similar to the history caching of LRU-K and comes with queue sizing problems for forecasts, too.

Sizing problems also arise for the FIFO queue and the LRU chain in the standard algorithm. Therefore, we used a simplified variation of 2Q where all buffer frames are assigned to the LRU chain and the FIFO queue only stores references to the pages in the LRU chain. So, it serves like an index for the LRU chain to identify pages referenced only once so far. Victims are primarily selected from the FIFO queue to replace those pages earlier. A subtlety of 2Q is here that the FIFO queue must not be drained to give new pages a chance for rereference and promotion to the LRU chain. The minimum fill degree of the FIFO queue is a configurable threshold. For simulation, we must therefore count the number of hotset entries in the queue, to be able to decide when a smaller buffer would pick a victim from the FIFO queue and not from the LRU chain. Also, the threshold must be the same for both sizes. Although this results in uniform retention times within the FIFO queue for differing LRU chain sizes, it is acceptable to some degree, because the threshold models the granted window for references of new pages. The hotset victim selection is sketched in Listing 5. Depending on the number of hotset entries in the $a1$ list (line 4), the victim is chosen either from the $a1$ list (lines 4–15) or from the am list (lines 17–22). Note that both LRU hotset pointers may refer to the same buffer page, which requires maintenance (i.e., moving the pointer) also for the unaffected queue (lines 5, 6 and lines 18, 19, respectively).

3.5 ARC and CAR

The ARC algorithm (Adaptive Replacement Algorithm) [10] employs a two-staged model like 2Q to achieve scan

Listing 5 2Q hotset victim selection

```

1 Frame chooseHotSetVictim() {
2   Frame v;
3   if ((a1.numberOfHotsetEntries() > threshold)) {
4     v = a1.hotsetLRU(); // victim from FIFO
5     if (v == am.hotsetLRU()) {
6       am.hotsetLRU() = am.hotsetLRU().amNext;
7       // if both list's hotset LRU pointers are equal
8       // move the am's too
9     }
10    a1.hotsetLRU() = a1.hotsetLRU().alNext;
11    while (!a1.hotsetLRU().hotSet) {
12      a1.hotsetLRU() = a1.hotsetLRU().alNext;
13      // skip non-hotset page to reposition
14      // a1's hotset LRU pointer
15    }
16  } else { // inverse logic for LRU list:
17    v = am.hotsetLRU(); // victim from LRU
18    if (v == a1.hotsetLRU())
19      a1.hotsetLRU() = a1.hotsetLRU().alNext;
20    am.hotsetLRU() = am.hotsetLRU().amNext;
21    while (!am.hotsetLRU().hotSet)
22      am.hotsetLRU() = am.hotsetLRU().amNext;
23  }
24  return v;

```

Listing 6 ARC page fix

```

1 Frame fix(long pageNo) {
2   Page p = buffer.get(pageNo);
3   if (p != null) {
4     if (p.getList() == B1) {
5       if (!p.overflow())
6         overflowMissCnt++;
7       else
8         overflowHitCnt++;
9     } else if (p.getList() == B2) {
10      if (!p.overflow())
11        overflowMissCnt++;
12      else
13        overflowHitCnt++;
14    } else {
15      overflowHitCnt++;
16      if (!p.hotSet())
17        hotsetMissCnt++;
18      else
19        hotsetHitCnt++;
20    }
21    p.refer();
22  } else {
23    p = strategy.victim(buffer);
24    buffer.remove(p.pageNo);
25    ... // load page
26    buffer.put(pageNo, p);
27    overflowMissCnt++;
28    hotsetMissCnt++;
29  }
30  return p;
31 }

```

residency. Its improved variant CAR (Clock with Adaptive Replacement) [2] provides similar performance in terms of hit ratios, but has the advantage that it “removes the cache hit serialization problem of LRU in ARC” [2]. While this is especially important for main memory caches, it is not in the center of interest for our discussion of disk-based DBMSs. Therefore, we focus here on ARC only.

ARC uses two LRU chains L_1 and L_2 to filter out scans and to retain hot pages for re-reference, respectively. The

Listing 7 ARC simulation functions for a Page

```

1 boolean hotSet(Page y, List lst) {
2   int steps = (lst == T1) ? T1.length : T2.length;
3   steps = steps - (steps * underSize / size);
4   int i = 0;
5   while (i < steps && y.next != null) {
6     i++;
7     y = y.next;
8   }
9   return (i >= steps);
10 }
11
12 // same as hotSet() but for B* lists
13 boolean overflow(Page y, List lst) {
14   int steps = (lst == B1) ? B1.length : B2.length;
15   steps = steps * overSize / size;
16   int i = 0;
17   while (i < steps && y.prev != null) {
18     i++;
19     y = y.prev;
20   }
21   return (i >= steps);
22 }

```

chain sizes are adapted at runtime through a parameter p . For that, the two chains are divided into top T_1 and bottom B_1 and T_2 , B_2 , respectively. The idea is similar to the SBPX approach. Pages referenced within T_1 and T_2 are in the cache while B_1 and B_2 only hold page identifiers of evicted pages. A page selected as victim from T_1 or T_2 is put into B_1 or B_2 , respectively. T_1 and T_2 are allowed to grow and shrink as long as their total amount of pages is below or equal to the buffer size, while parameter p yields the target size of T_1 . A hit in B_1 suggests a larger T_1 as “window” for a second reference to a page and, thus, increases p while a hit in B_2 suggests a growth of B_1 and decreases p . Victims are selected from the tail of T_1 as long as $|T_1| \geq p$ and from T_2 's tail otherwise.

The overhead for a separate SBPX-based oversize simulation can be avoided, when the history LRU chains B_1 and B_2 are taken into account. Therefore, the page fix routine is slightly adapted towards ARC-awareness. In Listing 6, the fix() algorithm is aware that pages found in the buffer (line 2–3) are either from a history LRU chain (line 4–13) or a T_* chain (line 14–20). We also added overflow and hotset counters for simulated misses and hits to illustrate the simple adaptation. Note, the refer() and victim() routines in line 21 and 23 are equal to the algorithms presented in [10].

Simulation of different buffer sizes in ARC does not use flags to indicate hotset or overflow membership of a page. Instead, two functions calculate these properties on-demand as shown in Listing 7. For hotset membership, we first estimate a valid hotset size depending on the current sizes of T_1 and T_2 (line 2 + 3), which determines the upper bound of page links to follow (called steps, line 5–8) towards the end of the T_* list. If the end can be reached within this bound, a page is not within the hotset range, otherwise it is.

The estimation of overflow membership looks similar. Only the chain traverse order is different because the steps

calculated to find the beginning of a B_* list set the upper bound. Usually, the hotset size is close to the actual size as well as the SBPX size. Therefore, following next and previous pointers into the direction of the real buffer size is favorable. Note, this way of simulation works for SBPX sizes is smaller than the history chains $B_1 + B_2$, i.e., SPBX size has to be less or equal to the cache size, which allows to forecast increase of up to 100% of the current size.

4 Buffer tuning

The crucial point in database tuning is the difficulty to precisely predict how a tuning decision will affect system performance. Even experienced database administrators with a deep knowledge of the workload and the database product itself regularly face this challenge. They rely on the assistance of sophisticated monitoring tools to prevent negative effects of their tuning decisions on the production system. Often they also run several observe-analyze-adjust cycles with reference workloads beforehand on dedicated test systems. Of course, this is time-consuming and expensive. Built-in self-monitoring and tuning components can ease this dilemma and reduce the risk of wrong decisions through rather small but continuous and incremental adjustments. In dynamic environments, however, those mechanisms may react too slow to keep up with the rate of workload shifts or short-term resource allocation for higher-level tuning decisions like auto-indexing. Therefore, we aim towards a re-formulation of the central question of automatic tuning from “Which adjustment *certainly will* give the greatest performance benefit?” to “Which adjustment *most likely will* give a performance benefit *but certainly not* result in a performance penalty?”. In other words, when we know that our reconfigurations will not harm, we get the freedom to react quicker and to apply more aggressive tuning.

In general, the total amount of buffer memory is limited and so the decision to assign more memory to a certain buffer is directly coupled with the decision of taking this memory from one or several others. Fortunately, the performance optimization heuristics for I/O-saving buffers (e.g. data pages, sorting) is straightforward: The more main memory can be used the better. Even an oversized buffer, i.e., a buffer larger than the actual data to be buffered, is less likely to become a performance bottleneck due to bookkeeping overhead. It is just a waste of main memory. The downsizing of a buffer, however, comes along with severe risks: the buffer’s locality may drastically decrease and even turn into thrashing causing excessive I/O, which also influences throughput of other buffers. Accordingly, we concentrate on the forecast of the negative effects of memory reallocations and base our tuning decisions not only, as common, on the estimated *benefits*, but also on vindicable forecasts of additional *costs*.

4.1 Cost model

Automatic tuning needs to derive costs from system state or from system behavior to quantify the quality of the current configuration. Additionally, it also needs to estimate the costs of alternative configurations to allow for comparison. Ideally, these costs comprise all performance-relevant aspects including complex dependencies between system components and future workload demands in a single number to allow for perfect decisions. Clearly, such a perfect cost model does not exist in practice. Instead, costs are typically derived from a mixture of cheaply accounted runtime indicators and heuristics-based or experience-based weight factors. The hope is to reflect at least the correct relationship between alternative setups w.r.t. to performance. The more precise this much weaker requirement can be met, the easier we can identify hazardous tuning decisions before they boomerang on the system.

In contrast to computational costs of a specific algorithm, costs expressing the quality of a buffer are inherently dependent on the current workload. Buffering 5% of the underlying data, for example, can be an optimal use of main memory at one moment, but become completely useless a few moments later. Therefore, each cost value is a snapshot over a window at a certain point in time with limited expressiveness for at most few periods ahead in the future. We define the general goal function for our tuning component as follows: At a given point in time t with a configuration c , find a configuration c' that has less accumulated I/O costs over the next n periods. The optimal window size and the number of forecast periods again depend on the actual workload; slowly changing workloads enable more precise cost estimations for longer periods, while rapidly changing workloads also decrease accuracy of future costs.

For simplicity, our cost model only considers buffer service time, i.e., the time needed to handle a page fix request. Of course, costs assigned to a specific buffer are dominantly determined by the number of I/Os performed. On a buffer miss (denoted m), a victim page has to be selected for replacement and flushed, if necessary, before the requested page is fetched from disk. Accordingly, a buffer miss causes at least one read operation, but may also cause several writes for flushing write-ahead log and victim page. The ratio between reads and synchronous writes is reflected by a weight factor f_{dirty} , which may vary over time and from buffer to buffer.

Depending on the characteristics of the underlying devices or blocking times under concurrent access, I/O times can also vary between various buffers. Hence, the costs of all buffers must be normalized to a common base to become comparable. We use here a second weight factor w_{buffer} for each buffer. As the time needed for a single I/O operation is easy to measure, these factors can be derived and adjusted at

runtime causing low overhead. Finally, the cost of a buffer at the end of time period t is expressed as:

$$c_{buffer}(t) = w_{buffer}(t) \cdot (1 + f_{dirty}(t)) \cdot m(t)$$

Note, we assume that CPU costs can be safely ignored, either because they are independent of whether an operation can be performed on buffered data or requires additional I/O, or because additional CPU cycles for search routines in larger buffers are negligible compared to an I/O operation. In the remainder of this paper, we also assume that read and write operations have symmetric costs and a low variance. However, it should be evident that the presented basic model can be easily extended to take asymmetric read/write costs (e.g. for solid state drives), different costs for random and sequential I/O, and also the apportionment of preparatory, asynchronous flushes of dirty pages into account.

4.2 Decision model

Our buffer balancing is based on the cost model of Sect. 4.1. In certain intervals, the buffer configuration is analyzed and reorganized if main memory reallocation promises reduced I/O costs for the entire system. Listing 8 shows the major algorithm for buffer balancing.

At the end of a monitoring period, we calculate a *save* and a *rise* ranking for all buffer pools based on their cost estimations. The higher a buffer pool is ranked in the *save* list, the more costs can be saved (i.e., this equals to providing a higher benefit) when its size is increased according to the simulated oversize. Similarly, the *rise* list ranks buffers by the cost estimations for undersize figures, where the minimum cost increase is ranked top (cf. Listing 8 lines 2–30).

With a greedy algorithm, buffer pool pairs are now picked from the top of both lists as long as the cost reduction on the *save* list is higher than the increase on the *rise* list (lines 32–45). A buffer may end up in both lists indicating a critical “jump” size which is easily recognized this way (line 38). A resize mechanism performs then the actual memory “shift”: The selected buffer from the *save* list is allowed to keep more page frames and references in the cache (line 50), while the buffer from the *rise* list is shrunken (line 51). For this, we repeatedly choose a victim page, flush it to disk if necessary and deallocate the page frame. Finally, the resized buffers are removed from both ranking lists (line 52–53) to avoid thrashing. Note, that we could compute an optimal solution for the resizing, but the greedy pair-wise resizing is much cheaper and delivers good results.

Of course, the simulated undersize and oversize areas of a buffer have to be adjusted as well. This is similar to the “regular” buffer resize. For instance, the number of hotset pages is reduced by selecting victims out of this subset and by switching their flags. Obviously, oversize areas can be

Listing 8 Balance algorithm

```

1 public void balance() {
2   RankList<Buffer> save = new RankList<Buffer>();
3   RankList<Buffer> rise = new RankList<Buffer>();
4   for (Buffer b : buffers) {
5     int saveIO = b.missCnt - b.overflowMissCnt;
6     int riseIO = b.hotsetMissCnt - b.missCnt;
7     boolean added = false;
8     for (Buffer sb : save) {
9       int cmp = sb.missCnt - sb.overflowMissCnt;
10      if (!added && saveIO > cmp) {
11        // insert into rank at current position:
12        sb.insertIntoRank(b);
13        added=true;
14      }
15    }
16    if (!added)
17      save.add(b); // buffer is added at the end
18
19    added = false;
20    for (Buffer rb : rise) {
21      int cmp = rb.hotsetMissCnt - rb.missCnt;
22      if (!added && riseIO < cmp) {
23        // insert into rank at current position:
24        rb.insertIntoRank(b);
25        added = true;
26      }
27    }
28    if (!added)
29      rise.add(b); // buffer is added at the end
30 }
31
32 Buffer i = null; // buffer to (i)ncrease
33 Buffer d = null; // buffer to (d)ecrease
34 while (true) {
35   i = null; d = null;
36   for (Buffer b : save) {
37     for (Buffer b2 : rise) {
38       if (d == null && b != b2) {
39         i = b; d = b2;
40         if ((d.hotsetMissCnt - d.missCnt) >
41             (i.missCnt() - i.overflowMissCnt))
42           red = null;
43       }
44     }
45   }
46   if (d == null)
47     break;
48
49   // amount depends on i's and d's simulation ranges
50   i.increase(amount);
51   d.decrease(amount);
52   save.remove(i); rise.remove(i);
53   save.remove(d); rise.remove(d);
54 }
55 }

```

kept or resized as desired. As oversize and undersize simulations for several buffer pools must not necessarily have the same size in bytes, gradual reallocations may become necessary. For that, we must extrapolate the buffer scaling behaviour between the real size and the simulated sizes (not shown).

Obviously, buffer resizing is a potentially expensive operation because it may require a forced flush of dirty pages when one buffer is shrunken in favor of another. However, the expected benefits justify this temporal overhead in general. If desired, the resize penalty for dirty pages could also

be included in the cost model to fully avoid this temporal negative effects.

4.3 Runtime aspects

Accounting hit/miss numbers for multiple simulated and real buffer sizes over a certain period of time induces estimations errors. For instance, a buffer smaller than the current buffer will usually have less hits and, in turn, requires more time to process the same sequence of buffer requests. Similarly, a buffer larger than the current one, will have a better hit ratio and potentially needs less time to process the same request sequence.

Therefore, simulation-based cost accounting needs to reuse the cost model's I/O weights for read and write operations to adjust the I/O cost in a (simulation) period to the different buffer sizes. That means, undersize simulation has to limit I/O accounting as soon as the maximum I/O budget for that period is reached and vice-versa for the SBPX part.

Adjusting memory assignments for buffer pools is also limited to the scalability prospects of a specific buffer algorithm. However, different buffer algorithms may perform differently and exchange of an algorithm would be an alternative tuning option without actually shifting memory. But different algorithms tend to use manifold figures such as access counters, timestamps, or history queues. The major problem is to carry over the current information when switching to a new algorithm. A poor alternative is to reset the entire propagation strategy. However, a practical way

is to initialize the new algorithm by evicting all the “old” pages into the new algorithm and continue to use the new algorithm. The decision to switch the algorithm can only be based on a full simulation of an alternative propagation algorithm relying on a similar cost model as presented in Sect. 4.1.

5 Evaluation

We assess the forecast accuracy of our extensions as well as the decision quality for buffer balancing with a generated set of benchmark workloads. As buffers do not scale uniformly, we generated page reference strings for various (common) patterns including random and sequential accesses of varying sizes.

5.1 Workload

In Figs. 3(a)–(d), we analyze the critical buffer size ranges for various access patterns whose characteristics are summarized in Table 1. Note, the total number of DB pages in a scenario is equal to the object size in the first column of the table. The only uniformly scaling buffer is measured for workloads dominated by random I/O (see Fig. 3(a)), where the overall hit ratio is—as expected—quite low. In this case, re-sizing extrapolations will work properly, but such an access behavior is unusual in databases. Dominating scans mixed with random access are modeled and measured in

Fig. 3 Buffer scalability for various workloads and replacement algorithms

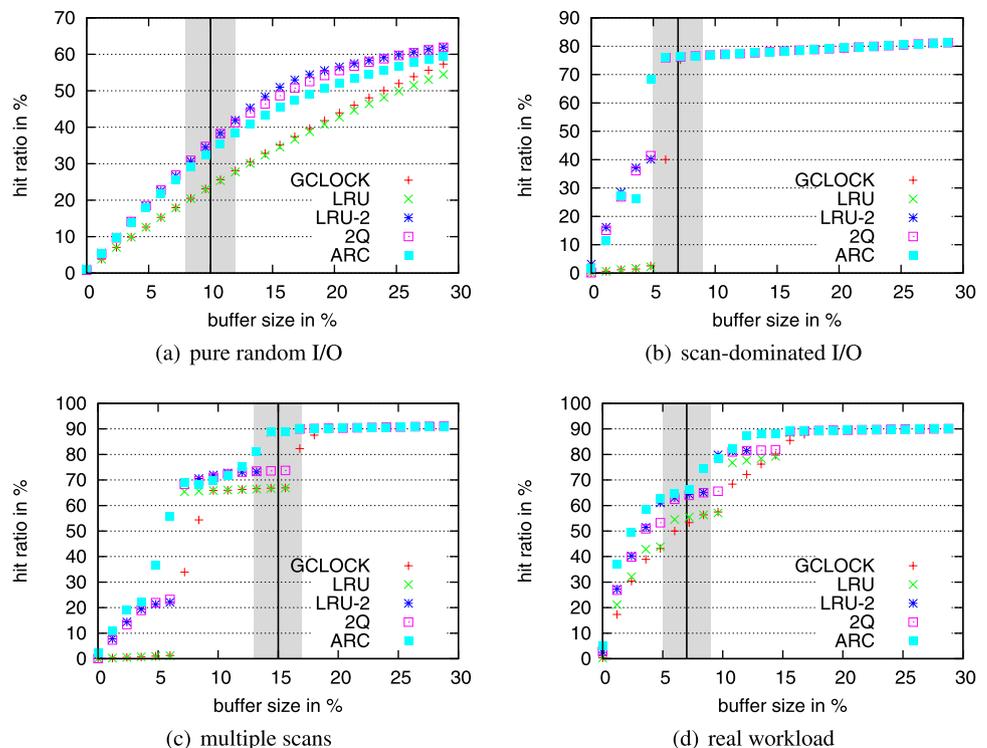


Table 1 Workload characteristics

Workload	Figure 3(a) (random)		Figure 3(b) (scan)		Figure 3(c) (jumps)		
Request share in %	50	50	25	75	10	65	25
\sum object size (pages)	150k	22k	150k	7k	150k	7k	13k
Access type	rnd	rnd	rnd	seq	rnd	seq	seq

Workload	Figure 3(d) (real)							
Request share in %	10	10	10	20	10	20	10	10
\sum object size (pages)	250k	5k	10k	10k	500	500	1k	2k
Access type	rnd	rnd	seq	seq	seq	seq	seq	seq

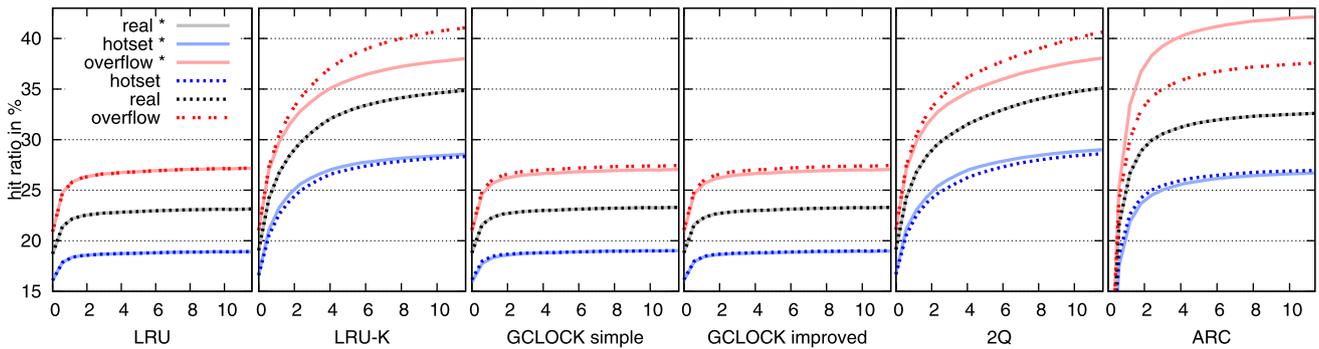


Fig. 4 Estimation accuracy for workload *random* (buffer calls $\times 100,000$ on *x*-axis)

Fig. 3(b). Although some of the replacement algorithms are “scan-resistant”, a dominant sequential access pattern easily provokes a “jump” in the buffer performance. In such cases, the buffer hit rate dramatically increases as soon as a frequent scan fits entirely into the buffer. Such “jumps” remain undetected if monitoring happens only at one side of the “jump”. The third workload shown in Fig. 3(c) is a mixture of multiple scans and random accesses in a single buffer. This scenario may represent a more typical buffer usage pattern which exhibits a realistic buffer scaling. In Fig. 3(c), several areas can be identified having different slopes, where each area boundary may cause uncertainty for extrapolations. In the last sample workload shown in Fig. 3(d), we have a mixture of high-locality scans and some noise generated by random accesses. This typical workload scenario causes several (small) jumps resulting in a stair-case pattern. In this case, fine-grained extrapolations necessary for buffer tuning may quickly fail, although the slope in the average is quite similar.

In the following sections, we investigate if our algorithms are capable of identifying and handling all of these (more or less) typical workload scenarios.

5.2 Accuracy

The quality of buffer balancing is based on the estimation quality of our extended buffer algorithms. Therefore, we

need to evaluate the estimation accuracy for the differing workloads. For the following experiments, the gray-shaded areas in Figs. 3(a)–(d) specify the simulated ranges centered around the actual buffer sizes indicated by the black lines. For simplicity, we always use a fixed range of $\pm 2\%$ of the total DB size. For each workload, we measure the accumulated undersize and oversize estimation accuracy. Each of the Figs. 4–7 contains the results of six algorithms using the same workload and up to 1.2 Mio buffer calls. The lines marked with an asterisk (*) illustrate the simulation-based hit ratios and, to enable comparison, the others show those of real buffers having the same sizes.

The first graphs are always showing the standard LRU behavior, which is always delivering perfect estimation accuracy; however, its hit ratio performance is not the best. But its lightweight simulation is definitely a plus. In contrast, the LRU-K results (second graphs) constantly indicate top hit ratios but show weaknesses in forecast quality. Especially, the downsize simulation of the *scan* workload fails with a dramatic overestimation.

The results for GCLOCK in Figs. 5 and 7 (third graphs) reveal its sensitivity to page order and clock-hand position for hotset simulations. By adding a second clock hand and forward pointers to simulate a separate clock for the hotset pages, we achieve considerably better accuracy (fourth

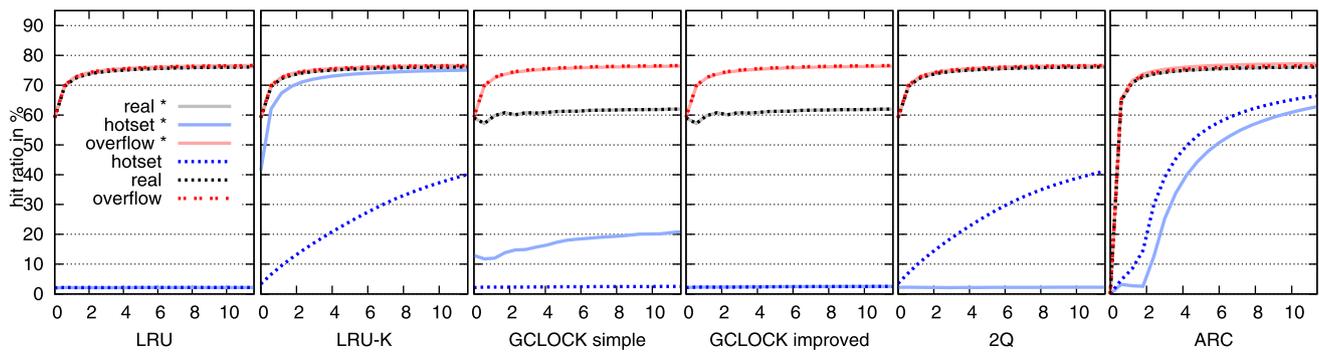


Fig. 5 Estimation accuracy for workload *scan* (buffer calls $\times 100.000$ on x -axis)

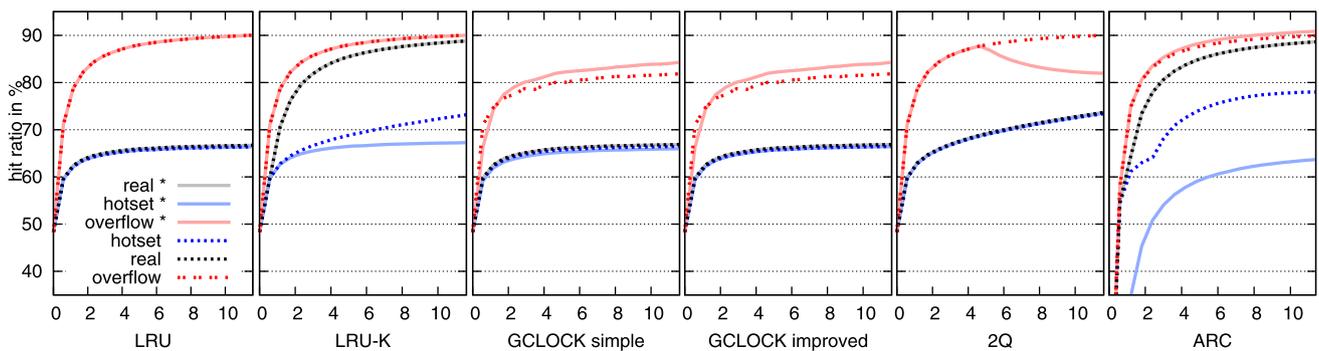


Fig. 6 Estimation accuracy for workload *jumps* (buffer calls $\times 100.000$ on x -axis)

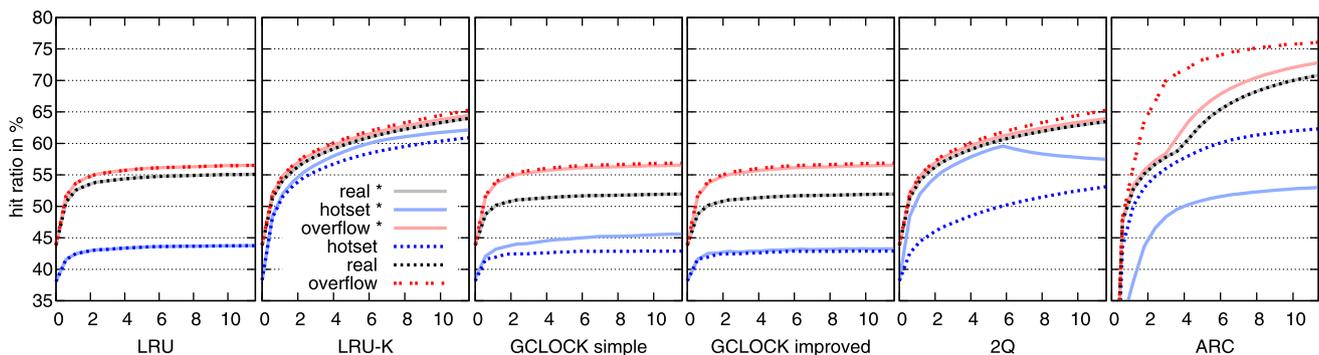


Fig. 7 Estimation accuracy for workload *real* (buffer calls $\times 100.000$ on x -axis)

graph), but its performance is always behind all other strategies.

On the second from right, we measure the forecast quality provided by the simplified 2Q algorithm. In all scenarios, it delivers top results while only requiring low maintenance overhead. However, forecast quality is disappointing in some scenarios. Similar to LRU-K, it fails for workload *scan*, but in the opposite direction with underestimation. Further, we observe a suddenly degrading forecast quality for the workloads *jumps* and *real*. Even worse, oversize estimations as well as undersize estimations are affected. Even

the use of a separate policy for the oversize buffer does not lead to better results.

The performance of ARC and its forecast accuracy is shown on the right-hand side. Besides always good hit ratios, ARC's forecast for downsizing underestimated the actual performance except for *random* workloads. Although the visual distance between estimation and actual hit ratio seems to be huge sometimes, this forecast error solely evolves from the algorithm's warm up. In Sect. 5.3, we are going to show that the permanent error is quite low, as the trend of the cumulative hit ratio forecast already indicates.

The experiments reveal that our simulations based on the locality principle lead to trustworthy estimations in many cases. On one side, simple algorithms like LRU and GCLOCK fit well into our framework. On the other side, more advanced algorithms such as LRU-K, ARC, and 2Q also allow lightweight estimations, but suffer from unpredictable estimation errors in some scenarios. The reasons are built-in mechanisms to achieve scan-resistancy, which are hard to model in simulations. Further, these algorithms do not allow logical composition of individual buffers.

5.3 Drift analysis

Some of the results shown in Sect. 5.2 reveal severe drifts leading to an apparent bad estimation quality. The simulations of ARC for example, show dramatic drifts for oversize and hotset estimation in Figs. 4 and 7, respectively. But the results show cumulated hit ratios, which means that even small but constant estimation errors sum up. Therefore, we analyzed the estimation error for the hit ratios of simulated buffer sizes for both extreme drift situations in Fig. 8. After a short warmup period, the error ratio settles between 5% for the overflow estimation of workload *random* and 10% for the hotset estimation of workload *real*.

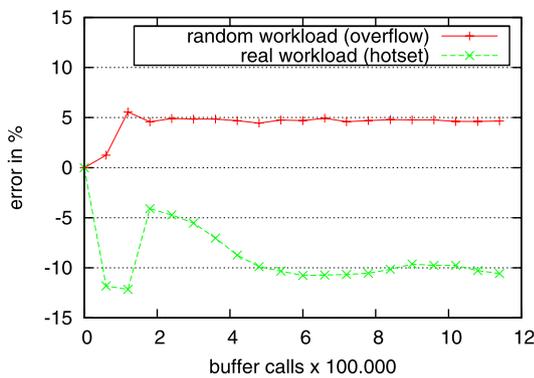


Fig. 8 Simulation error analysis explaining hit ratio drift

Obviously, a good choice for the simulation interval helps to reduce the estimation error rate. However, this is another tuning parameter which may also be adjusted during run-time.

5.4 Shifting workloads

Based on our accuracy and drift analysis we examined *workload shifts*. To guarantee that we hit “interesting” buffer (simulation) sizes for these shifts, we extended our predefined workloads from Sect. 5.1. Each result in Fig. 9 shows for 5 workload shifts the cumulative hit ratios for hotset and overflow simulations as well as their estimation errors. The vertical lines in the upper illustrations indicate the workload shifts, which (as expected) come along with kinks in the graphs. We omitted LRU simulations here because they always deliver fully accurate results.

On the one hand, we have the results for LRU-K and 2Q, which disclose serious problems for certain workloads, while hit ratio estimations for others are nearly perfect. For example, scan-dominated corner-cases such as the third and fifth workload shift cause hotset errors of up to 25% and 40% until the next shift. On the other hand, estimations for GCLOCK improved and ARC seem to be more reliable. Some workload shifts cause short peaks, where the simulation needs to adapt itself. However, most of the time, only marginal estimation errors below 2% are observed.

The workload shift analysis shows that for many cases estimation errors are quite low. Certain corner cases, i.e., extreme workloads causing a kind of thrashing for the simulation ranges, are hard to estimate and may lead to weak estimations. Because those corner cases are rather exceptional, we believe that they do not affect the benefit of our algorithmic extensions for resize simulations in general.

5.5 Buffer balance

In Fig. 10, the self-tuning mechanism presented in Sect. 4.2 automatically tunes two buffers, where buffer 0 was fed with

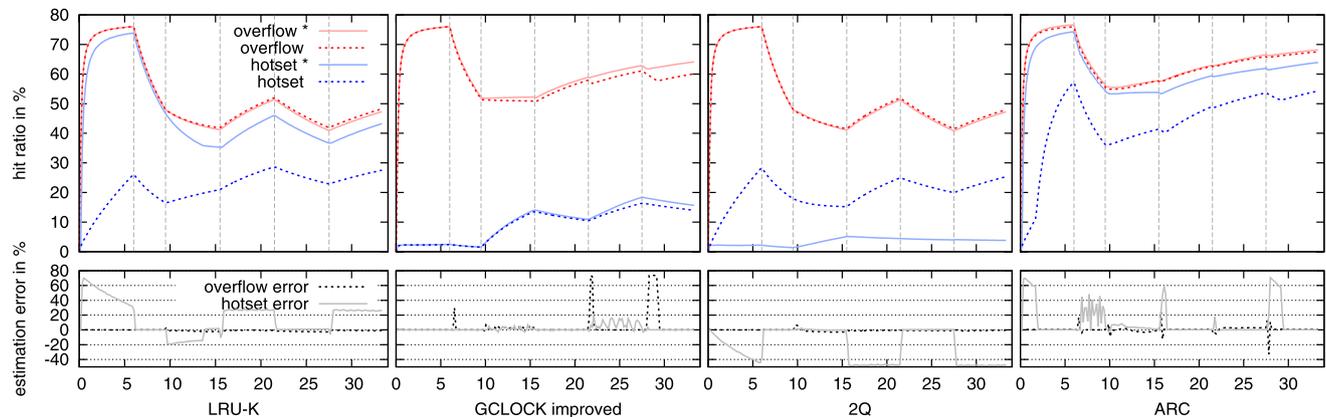


Fig. 9 Shifting workload analysis (buffer calls $\times 100,000$ on x-axis); LRU omitted due to zero error

random workload from Fig. 3(a) and buffer 1 with *scans* shown in Fig. 3(b). Buffer sizes (i.e., simulation and real) are chosen as described in Sect. 5.2. Due to space limitations, we present the results only exemplified for the improved GCLOCK and a fixed memory shift granularity of 2% of the DB size. After the buffers were warmed up (i.e., after 1.2 Mio buffer calls), the cost model triggers all memory shifts. The *random* workload buffer was shrunken according to its hotset simulation, whereas buffer 1 was in-

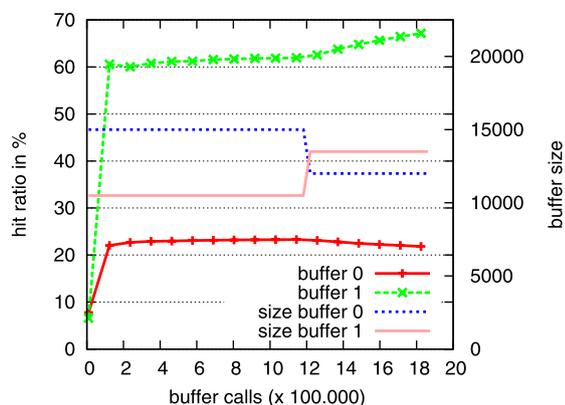


Fig. 10 Buffer balancing *random* vs. *scan*

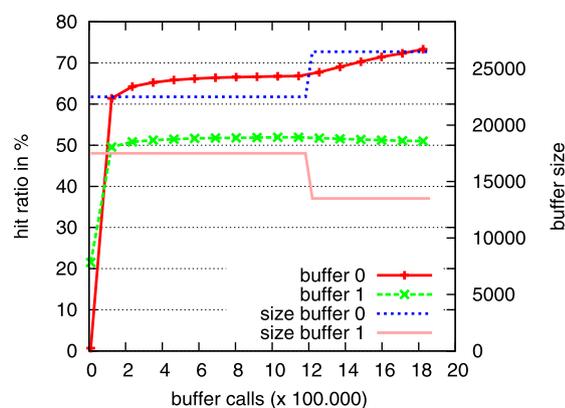
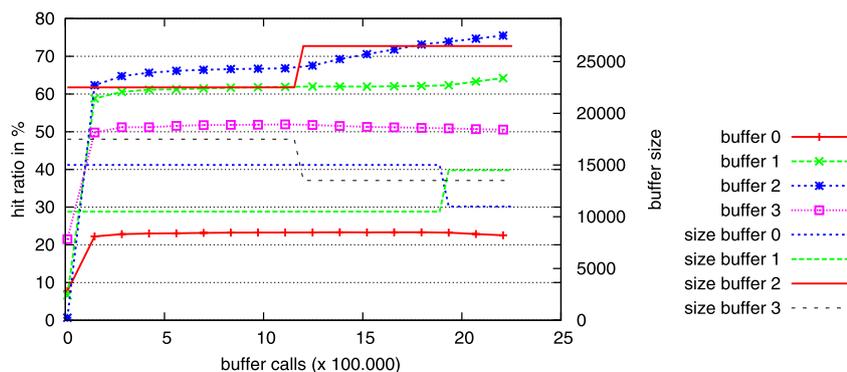


Fig. 11 Buffer balancing *jump* vs. *real*

Fig. 12 Balancing of four buffers under different workloads



creased. Although the hit ratio of buffer 0 slightly descends, the overall I/O performance improves, because the hit ratio of buffer 1 increases considerably.

Because the self-tuning decisions are based on a cost model, they are applicable for arbitrary scenarios. In our second example, we again use two buffers, one that is fed from the workload *jumps* and the other from the workload *real* as shown in Figs. 3(c) and 3(d). In this setting, SBPX fails because it does not recognize that the size of buffer 0 is close to a “jump” boundary. However, as indicated by Fig. 11, our downsizing simulation detects the pitfall and prevents buffer performance penalties.

As explained above, resizing two buffers is fairly simple. Therefore, we combine both experiments in a single setup shown in Fig. 12. The cut-out shows two memory shifts leading to minor descends of the hit ratio on the one side but clear improvements on the other side resulting in a steadily improved buffer performance.

In summary, we could experimentally prove that buffer balancing can be achieved at low cost, but it heavily depends on accurate and lightweight forecasts for both directions – upsize and downsize.

6 Conclusions

Even after decades of research on buffer management and optimization, the problem of a reliable, dynamic adaptation of buffer memory allocation is not fully solved. In this work, we studied opportunities to forecast buffer resizing effects to support harm-free self-tuning decisions. As downsizing a buffer is accompanied with severe risks of thrashing, we argued that reliable prediction of downsizing effects is a key point for self-tuning decisions. Furthermore, we argued that additional overhead for these forecasts must not add noticeable overhead to normal processing. Therefore, we focused on lightweight techniques to exploit knowledge from the buffer replacement strategies for forecasts and presented possible solutions for five families of replacement algorithms.

In our experiments, we could show that forecast quality is heavily dependent on the actual strategy. It seems that sophisticated strategies like LRU-K and 2Q make it hard or even impossible to get reliable forecasts for either upsizing, downsizing, or both. We found that there are two reasons for this: First, such algorithms use history-recording techniques, which are very costly to emulate for varying sizes. Second, they are extremely sensible to configuration parameters, which cannot be easily negotiated between differing buffer sizes. However, simpler, yet widely-used strategies like LRU and GCLOCK turned out to allow for cheap and highly accurate or even perfect forecasts. Especially our results gained for ARC presented a fairly good accuracy paired with an extreme low overhead. Our accuracy results for a mix of critical workloads and abrupt workload shifts have demonstrated the strength and weaknesses of all algorithms under test. In conjunction with a simple cost model and a greedy algorithm, we demonstrated the use of forecasts to improve buffer hit ratios without the risk of severe performance penalties. Following the idea of differing “stages” in 2Q to improve buffer behavior, our findings suggest to think about further partitioning of buffers with complex replacement strategies into several distinct buffers with simpler but more predictable strategies. This way, forecasts would generally become reliable and fragmentation issues were automatically resolved by the self-tuning capabilities.

References

- Bächle S, Schmidt K (2011) Lightweight performance forecasts for buffer algorithms. In: BTW 2011, LNI, vol P-180, pp 147–166
- Bansal S, Modha DS (2004) Car: clock with adaptive replacement. In: Proceedings of the 3rd USENIX conference on file and storage technologies. USENIX Association, Berkeley, pp 187–200. <http://portal.acm.org/citation.cfm?id=1096673.1096699>
- Brown KP, Carey MJ, Livny M (1993) Managing memory to meet multiclass workload response time goals. In: VLDB 1993. Morgan Kaufmann, San Mateo, pp 328–341
- Diao Y et al (2005) Comparative studies of load balancing with control and optimization techniques. In: ACC’05: proc 24th American control conf, pp 1484–1490
- Duan S, Thummala V, Babu S (2009) Tuning database configuration parameters with ituned. Proc VLDB Endow 2(1):1246–1257
- Effelsberg W, Härder T (1984) Principles of database buffer management. ACM Trans Database Syst 9(4):560–595. <http://doi.acm.org/10.1145/1994.2022>
- Jauhari R, Carey MJ, Livny M (1990) Priority-hints: an algorithm for priority-based buffer management. In: VLDB 1990, pp 708–721
- Johnson T, Shasha D (1994) 2q: a low overhead high performance buffer management replacement algorithm. In: VLDB 1994, pp 439–450
- Martin P, Li HY, Zheng M, Romanufa K,++ Powley W (2000) Dynamic reconfiguration algorithm: dynamically tuning multiple buffer pools. In: DEXA 2000, pp 92–101
- Megiddo N, Modha DS (2003) Arc: a self-tuning, low overhead replacement cache. In: Proceedings of the 2nd USENIX conference on file and storage technologies, pp 115–130. USENIX Association, Berkeley. <http://portal.acm.org/citation.cfm?id=1090694.1090708>
- Ng R, Faloutsos C, Sellis T (1995) Flexible and adaptable buffer management techniques for database management systems. IEEE Trans Comput 44(4):546–560. <http://dx.doi.org/10.1109/12.376169>
- Nicola VF, Dan A, Dias DM (1992) Analysis of the generalized clock buffer replacement scheme for database transaction processing. In: ACM SIGMETRICS 1992, pp 35–46
- O’Neil EJ, O’Neil PE, Weikum G (1999) An optimality proof of the LRU-K page replacement algorithm. J ACM 46(1):92–112
- Storm AJ, Garcia-Arellano C, Lightstone SS, Diao Y, Surendra M (2006) Adaptive self-tuning memory in db2. In: VLDB 2006, pp 1081–1092
- Tran DN, Huynh PC, Tay YC, Tung AKH (2008) A new approach to dynamic self-tuning of database buffers. Transf Storage 4(1):1–25. <http://doi.acm.org/10.1145/1353452.1353455>



Karsten Schmidt studied Computer Science from 1999 to 2006 at the Technical University of Ilmenau. Since June 2006 he is a scientific staff member in the XTC project at the DBIS research group lead by Prof. Härder. His main interests are storage structures and indexes for XML documents and adaptivity in database management systems.



Sebastian Bächle studied Computer Science from 2002 to 2007 at the University of Kaiserslautern. Since September 2007 he is a scientific staff member in the XTC project at the DBIS research group lead by Prof. Härder. His main interests are concurrency control in XML documents, database architecture, and parallel XQuery processing.