On List Update with Locality of Reference *

Susanne Albers† Sonja Lauer‡

Abstract

We present a comprehensive study of the list update problem with locality of reference. More specifically, we present a combined theoretical and experimental study in which the theoretically proven and experimentally observed performance guarantees of algorithms match or nearly match.

In the first part of the paper we introduce a new model of locality of reference that closely captures the concept of runs, representing sequences of requests to the same item. Using this model we develop refined theoretical analyses of popular list update algorithms. The second part of the paper is devoted to an extensive experimental study in which we have tested the algorithms on traces from benchmark libraries. It shows that the theoretical and experimental bounds differ by just a few percent.

Our new theoretical bounds are substantially lower than those provided by standard competitive analysis. Another result is that the well-known Move-To-Front strategy exhibits the best performance. Its refined competitive ratio tends to 1 as the degree of locality in a request sequence increases. This confirms that Move-To-Front is the method of choice in practice.

1 Introduction

The list update problem is one of the most extensively studied online problems, with a tremendous body of literature published over the past 40 years. The problem has been investigated with respect to both average-case and worst-case competitive analysis. We refer the reader to [1, 4, 5, 6, 12, 18, 27, 30, 32, 35, 36, 38, 41] for a selection of some key results.

The list update problem consists in maintaining a set of items as an unsorted linear list. More specifically, a linear linked list of items is given. A list update algorithm is presented with a sequence of *requests* that must be served in their order of occurrence. Each request specifies an item in the list. In order to serve a request, a list update algorithm must *access* the requested item, i.e. it has to start at the front of the list and search linearly through the items until the desired item is found. Accessing the i-th item in the list incurs a cost of i. Immediately after an access, the requested item may be moved at no extra cost to any position closer to the front of the list. These exchanges are called *free exchanges*. All other exchanges of two consecutive items in the list cost 1 and are called *paid exchanges*. The goal is to serve the request sequence so that the total cost is as small as possible. We emphasize that this is the standard cost model, see also [38]. Of particular interest are *online algorithms* that serve each request without knowledge of any future requests.

While early work on the list update problem evaluated online algorithms assuming that requests are generated according to probability distributions, research over the past 20 years has focused on *competitive analysis* [38]. Here an online algorithm is compared to an optimal offline algorithm. Given a request sequence σ,

*Work supported by the Deutsche Forschungsgemeinschaft and the European Research Council (ERC), Grant Agreement No. 691672.

†Department of Computer Science, Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany, albers@in.tum.de.

‡Department of Computer Science, University of Freiburg, Georges Köhler Allee 79, 79110 Freiburg, Germany. sonja.lauer@informatik.uni-freiburg.de

let $A(\sigma)$ denote the cost incurred by online algorithm A in serving σ, and let $OPT(\sigma)$ denote the optimum offline cost. Algorithm A is called c-competitive if there exists a constant α such that $A(\sigma) \leq c \cdot OPT(\sigma) + \alpha$ holds for all σ and all size lists.

In 1985 Sleator and Tarjan proved that the *Move-To-Front* algorithm is 2-competitive [38]. This elegant strategy simply moves an item to the front of the list whenever it is requested. Since then, algorithms with an improved competitiveness have been developed. While the competitive ratios are of course constant, there is a substantial gap between the theoretical bounds and the performance ratios of the algorithms observed in practice. Moreover, *Move-To-Front* often outperforms other strategies, see e.g. [10, 12]. The reason is that competitive analysis considers arbitrary request sequences, whereas sequences arising in practice have a special structure: They exhibit locality of reference, meaning that at any point in time only a small set of items is referenced.

There has been considerable research interest in studying the paging problem with locality of reference [3, 7, 14, 20, 25, 26, 31, 33] because, in paging, the gap between the theoretical and experimental performance values is even super-constant. However, hardly any work has been presented for the classical list update problem. In fact, references [10, 29] point out that locality is an essential aspect in the list update problem and that a good model is required to properly evaluate the performance of algorithms.

Previous results: We focus on the results that have been developed in the framework of competitive analysis. As mentioned above Sleator and Tarjan [38] showed that *Move-To-Front* is 2-competitive. This is the best factor deterministic online algorithms can achieve [32]. Bachrach and El-Yaniv [9] devised deterministic *MRI* and *PRI* families of algorithms. These families attain competitive ratios of 2 and 3, respectively. We next turn to randomized algorithms. The first randomized strategy was presented by Irani [30]. Her *Split* algorithm is 1.9375-competitive. Reingold et al. [35] presented an elegant *BIT* algorithm that is 1.75-competitive. This factor is substantially below the deterministic bound of 2. The *BIT* algorithm can be generalized to a family of *Counter* strategies [35]. A *Timestamp* family of algorithms was developed in [1]. It achieves a competitiveness equal to the Golden Ratio $\Phi \approx 1.62$. The best randomized algorithm currently known is *COMB* which is 1.6-competitive [4]. Interestingly, *COMB* is a combination of *BIT* and a (deterministic) element of the *Timestamp* family. The factor of 1.6 is close to best lower bound of 1.50084 developed by Ambühl et al. [6] on the performance of randomized list update algorithms.

Experimental studies for the list update problem have been presented by Rivest [36], Bentley and McGeoch [12] and Bachrach et al. [10]. They analyzed popular algorithms on request sequences generated by probability distributions and Markov sources, on sequences derived from text and Pascal files as well as on sequences extracted from the Calgary Corpus [17]. The results are not unanimous. A conclusion is that the ranking of algorithms depends on the degree of locality in the input.

The only prior work addressing list update with locality of reference was a paper by Angelopoulos et al. [8]. They adapted a locality model [3] introduced for the paging problem and proved that *Move-To-Front* is superior to other algorithms.

Our contribution: We present a comprehensive study of the list update problem with locality of reference. The goal is to provide a refined analysis of the problem in which theoretical and empirical results match or nearly match. To this end our study integrates theoretical and experimental work.

First, in Section 2, we introduce a new model of locality of reference that is based on the natural concept of *runs*. A run is a sequence of requests to the same item. We define a number of parameters that characterize request sequences in terms of the occurrence of long runs. Using these parameters we will be able to accurately estimate the performance of list update algorithms. We also define a model of so-called λ-*locality* that characterizes classes of input sequences with respect to their degree of locality. Loosely speaking, the more long runs there are, the higher the locality. As we shall see, our new concepts properly capture locality of reference in the list update problem, both from a theoretical and practical point of view.

In Section 3 we present refined theoretical analyses of list update algorithms. We concentrate on the most

popular strategies that have received the most attention recently, namely *Move-To-Front*, *BIT* and *COMB*. In order to be able to analyze *COMB*, we have also evaluated a member of the *Timestamp* family. Of course, we have also investigated an optimal offline strategy. For each algorithm we have analyzed the total service cost incurred on a request sequence, where cost is expressed in terms of our new locality parameters. Interestingly, for *Move-To-Front* our cost analysis is exact, i.e. our locality model is powerful enough to exactly quantify *Move-To-Front*'s service cost on any request sequence. Furthermore, for each online algorithm, we have evaluated its performance relative to that of an optimal offline algorithm. Here *Move-To-Front* achieves an excellent performance ratio and responds well to locality of reference: The competitiveness even tends down to 1 as the degree of locality increases. This does not hold true for the other online algorithms.

In Section 4, we present a comprehensive experimental study in which we have evaluated our list update algorithms on real-world traces from benchmark libraries. Obviously, the list update problem is a solution to the classic dictionary problem. In this context, in practice, requests are memory accesses. Secondly, list update has interesting applications in data compression, see e.g. [13, 16]. For instance, the open source data compression program `bzip2` relies on *Move-To-Front* encoding in combination with a preceding Burrows Wheeler transformation. Therefore, in our experiments we consider as input (a) memory access strings (47 traces) and (b) sequences arising in data compression routines (44 traces). In our tests we first analyze the traces with respect to their locality characteristics. It shows that the parameters introduced in Section 2 are indeed sensible.

Next, in the experiments, for each algorithm and each input sequence, we have computed the total service cost. Furthermore, for each online algorithm and each input, we have determined the *experimentally observed competitiveness*, which is the total service cost of the algorithm divided by the total cost incurred by an optimal offline strategy. Since the offline version of the list update problem is NP-hard, we have approximated the optimum service cost by that of the pairwise optimum, see Section 4 for details. In general, our theoretically proven and experimentally observed bounds are very close and differ by just a few percent. As for the total service cost, *Move-To-Front* exhibits an error of 0 because our theoretical bound is exact. For the other three online algorithms *BIT*, *Timestamp* and *COMB*, the average relative error between the theoretical and experimentally observed values is 3–4% on the memory access traces and 7–9% on the data compression sequences. As far as performance ratios relative to the optimum are concerned, the average relative errors between our theoretical bounds and the experimentally observed competitive ratios are a bit higher. *Move-To-Front* exhibits average relative errors of 0.3% on the memory traces and of 0.7% on the data compression sequences. The other three strategies incur average errors of 3–4% on the memory traces and of 8–10% on the data compression sequences.

In our study, the theoretical and experimental performance ratios of the algorithms are much lower than the corresponding standard competitive ratios. In particular, *Move-To-Front* shows the best performance with ratios in the range of 1.2–1.3. This confirms that *Move-To-Front* is the method of choice in practice.

We finally remark that our study does not address the algorithms *Transpose* and *Frequency Count* as they to not achieve constant competitive ratios [38].

Subsequent work: After the conference publication of our paper, list update with locality of reference has been studied in [19, 22, 21]. Dorrigiv et al. [19] propose a locality model based on Denning's working sets and analyze the performance of algorithms using a certain non-locality parameter. They show that *Move-To-Front* is an optimal strategy with respect to this parameter. Moreover, the analysis separates *Move-To-Front* from other strategies. However, no experimental study is conducted, comparing the theoretical bounds to those observed in practice. Dorrigiv and López-Ortiz [22, 21] study list update algorithms under probability distributions that exhibit locality of reference. They analyze the expected service cost of various algorithms and show that *Move-To-Front* is the best strategy in this framework. A study of list update algorithms in data compression was presented by Dorrigiv et al. [23, 24].

2 A new model for locality of reference

Informally speaking, a request sequence exhibits locality of reference if, at any time, it references only a small set of items. If an item is requested, it is likely to be requested again soon. This description suggests the concept of *runs*, where a run is a subsequence of requests to the same item. In the best case, when there is a high degree of locality, an item is requested many times in a row before a different element is referenced. Unfortunately, real-world request sequences may contain only few of these pure long runs. However, long runs may occur if we focus on small item sets and, in particular, on item pairs: If, at any time, item x is more relevant than y, then this relation is likely to hold also in the near future and we encounter several requests to x before the next reference to y arises. Thus long runs occur if we project request sequences to smaller item sets or item pairs. A request sequence exhibits a high degree of locality if a substantial portion of the requests belongs to long runs. In the following we introduce a formal model of locality of reference based on this generalized notion of runs.

Let L be the set of items in the list to be maintained. Consider a fixed request sequence σ. For any two items $x, y \in L$ with $x \neq y$, let σ_{xy} be the request sequence that is derived from σ when deleting all requests that are neither to x nor to y, i.e. only the requests to x and y survive. Any maximal subsequence of consecutive requests to the same item in σ_{xy} is called a *run*. Let $r(\sigma_{xy})$ be the number of runs in σ_{xy}. A run is *short* if it consists of one request only. A run consisting of at least two requests is *long*. Let $s(\sigma_{xy})$ and $l(\sigma_{xy})$ denote the number of short and long runs, respectively, in σ_{xy}. Then $s(\sigma_{xy}) + l(\sigma_{xy}) = r(\sigma_{xy})$.

On long runs online algorithms typically perform well, relative to an optimal offline algorithm. In order to properly evaluate our algorithms, we need some further definitions. A long run ρ is called a *prefixed long run* if it is preceded by one or more short runs; otherwise ρ is called an *independent long run*. Again let $l_p(\sigma_{xy})$ and $l_i(\sigma_{xy})$ be the number of prefixed and independent long runs, respectively. We have $l_p(\sigma_{xy}) + l_i(\sigma_{xy}) = l(\sigma_{xy})$.

Consider two long runs ρ' and ρ such that ρ' occurs earlier than ρ in σ_{xy}. Run ρ' *immediately precedes* ρ if the last request of ρ' is followed by the first request of ρ in σ_{xy} or if ρ' and ρ are separated by short runs only. A long run ρ which is not equal to the first long run in σ_{xy} represents a *long run change* if ρ and the immediately preceding long run ρ' reference different items. The first long run ρ in σ_{xy} represents a long run change if ρ and the first request of σ_{xy} reference the same item (imagining that σ_{xy} was preceded by a long run to just the other item of $\{x, y\}$). Let $l_c(\sigma_{xy})$ be the number of long run changes in σ_{xy}. We have $l_c(\sigma_{xy}) \leq l(\sigma_{xy})$. Furthermore, $l_i(\sigma_{xy}) \leq l_c(\sigma_{xy})$ because each independent long run, except for possibly the first one, is immediately preceded by another long run, which references a different item. The number of long run changes will be particularly important in lower bounding the cost incurred by an optimal offline algorithm. Hence, it will allow us to derive good upper bounds on the relative performance ratios of online strategies. Table 1 summarizes the various parameters for a projected sequence σ_{xy}.

As an example, consider $\sigma_{xy} = xxyyyxyxyxxxxyxxx$. The sequence consists of nine runs, five of them are short and the remaining four are long. That is $r(\sigma_{xy}) = 9$, $s(\sigma_{xy}) = 5$ and $l(\sigma_{xy}) = 4$. The runs $xxxx$ and xxx are prefixed long runs so that $l_p(\sigma_{xy}) = 2$ and $l_i(\sigma_{xy}) = 2$. All long runs, except for the last one, represent a long run change, i.e. $l_c(\sigma_{xy}) = 3$. On the other hand, if σ_{xy} started with a short run to x, then the following long run yyy would not be a long run change.

So far we have defined a number of values for a particular request sequence σ_{xy}. We now sum these values over all pairs of items x and y. For any pair $x, y \in L$ with $x \neq y$ and for any value $v \in \{r, s, l, l_i, l_p, l_c\}$, let $v(\sigma) = \sum_{\{x,y\} \subseteq L, x \neq y} v(\sigma_{xy})$. For instance, $r(\sigma)$ is the total number of runs in σ, while $s(\sigma)$ and $l(\sigma)$ represent the total number of short and long runs, respectively, in σ. In the experiments (Section 4) it shows that all of the parameters are sensible. Typically, 60% of the runs are long runs. Most of them are independent long runs. Moreover, the value l_c is quite expressive. The ratio l_c/l is usually 5% to 10% higher than l_i/l.

All the definitions presented so far refer to a given request sequence σ and, using these definitions, we will be able to accurately evaluate the performance of list update algorithms on such a σ. Next, we introduce

Parameter	Count
$r(\sigma_{xy})$	runs
$s(\sigma_{xy})$	short runs
$l(\sigma_{xy})$	long runs
$l_p(\sigma_{xy})$	prefixed long runs
$l_i(\sigma_{xy})$	independent long runs
$l_c(\sigma_{xy})$	long run changes

Table 1: The parameters of the locality model, for any σ_{xy}. For a request sequence σ, the parameters are summed over all pairs $x \neq y$.

a model of locality of reference that applies to *classes* of request sequences which may be generated by a particular application. Intuitively, request sequences exhibit a high degree of locality if there are many long runs. However, in order to obtain meaningful results we have to work with a refined definition. Again, the number of long run changes is crucial. We say that a class Σ of request sequences exhibits λ-*locality*, for some $0 \leq \lambda \leq 1$, if for any $\sigma \in \Sigma$ inequality $l_c(\sigma)/r(\sigma) \geq \lambda$ holds, i.e. the number of long run changes represents at least a fraction of λ among all the runs. Note that, for a given request sequence, $l_c(\sigma)$ accounts for all the independent long runs and, depending on the input, for a smaller or larger fraction of the prefixed long runs. If a request sequence consists of long runs only, we have $\lambda = 1$.

An alternative, perhaps more intuitive definition would be to set λ as $l(\sigma)/r(\sigma)$. However with this definition we would not be able to derive good bounds because the analysis of an optimal offline algorithm OPT crucially depends on $l_c(\sigma)$. A reader may wonder why our locality model does not incorporate the length of long runs. This parameter is irrelevant for algorithms performance because after the second request of a long run competitive algorithms have moved the referenced item ahead of the other item in the list and no further cost is incurred on the run. We finally remark that our new locality model, based on runs, is different from models introduced for paging, see again [3, 7, 14, 20, 33]. The latter usually model working sets and, in particular, working set sizes over certain time intervals.

3 Analyzing online and offline algorithms

In this section we present refined theoretical analyses of list update algorithms. We first revisit a general analysis framework based on item pairs. Then we lower bound the cost of an optimal offline algorithm. This estimate will be crucial to evaluate the performance of online algorithms.

3.1 Basic cost analysis

We show that the cost incurred by any online or offline list update algorithm A on a request sequence σ can be evaluated by considering pairs of items. This reduction is not new, but the reduction shown here incorporates for the first time paid exchanges that an algorithm may perform.

Let $m = |\sigma|$ be the length of σ and $\sigma(t)$ be the request posed at time t, $1 \leq t \leq m$. The cost incurred by A in accessing $\sigma(t)$ is 1 plus the number of items that precede item $\sigma(t)$ in the list at time t. Additionally, A may perform paid exchanges. For any item $x \in L$ and any time t, define $A_x(t, \sigma) = 1$ if x precedes item $\sigma(t)$ in the list at time t; otherwise $A_x(t, \sigma) = 0$. We note that $A_x(t, \sigma) = 0$ for any t with $\sigma(t) = x$. Furthermore, let $A_p(t, \sigma)$ denote the number of paid exchanges performed by A at time t. Using these definitions, the cost

incurred by any online or offline list update algorithm A can be expressed as

$$\begin{aligned} A(\sigma) &= \sum_{t=1}^{m}(\sum_{x\in L} A_x(t,\sigma) + A_p(t,\sigma)) + m \\ &= \sum_{x\in L}\sum_{t=1}^{m} A_x(t,\sigma) + \sum_{t=1}^{m} A_p(t,\sigma) + m \\ &= \sum_{x\in L}\sum_{y\in L}\sum_{t:\sigma(t)=y} A_x(t,\sigma) + \sum_{t=1}^{m} A_p(t,\sigma) + m \\ &= \sum_{\substack{\{x,y\}\subseteq L \\ x\neq y}}\sum_{\substack{t: \\ \sigma(t)\in\{x,y\}}} (A_x(t,\sigma) + A_y(t,\sigma)) + \sum_{t=1}^{m} A_p(t,\sigma) + m. \end{aligned}$$

Let $A_{p,xy}(\sigma)$ be the total number of paid exchanges performed by A to change the relative order of x and y in the list while serving σ. Then

$$A(\sigma) = \sum_{\substack{\{x,y\}\subseteq L \\ x\neq y}} (\sum_{\substack{t: \\ \sigma(t)\in\{x,y\}}} (A_x(t,\sigma) + A_y(t,\sigma)) + A_{p,xy}(\sigma)) + m.$$

Furthermore, let $A_{xy}(\sigma) = \sum_{t:\sigma(t)\in\{x,y\}}(A_x(t,\sigma) + A_y(t,\sigma)) + A_{p,xy}(\sigma)$. This term intuitively represents the cost incurred by items x and y on requests that are to either x or y, plus the number of paid exchanges involving both x and y. We remark that, for any t with $\sigma(t) \in \{x, y\}$, the sum $A_x(t,\sigma) + A_y(t,\sigma)$ is either 0 or 1 depending on whether or not the requested item precedes the other item of the pair $\{x, y\}$. With the abbreviation $A_{xy}(\sigma)$ we obtain

$$A(\sigma) = \sum_{\substack{\{x,y\}\subseteq L \\ x\neq y}} A_{xy}(\sigma) + |\sigma|. \tag{1}$$

Suppose that algorithm A serves σ_{xy} on the two-item list that consists of x and y only and let $A(\sigma_{xy})$ be the incurred cost in the *partial cost model*. In this model the cost of serving a request is equal to the number of items that precede the requested item in the current list, i.e. a request to the first item in the list costs 0 and a request to the second item in the list costs 1. All online algorithms proposed in the literature for list update have the property that $A_{xy}(\sigma) = A(\sigma_{xy})$, for any $x, y \in L$ with $x \neq y$. We will verify this property when studying online algorithms in the following sections. As for an optimal offline strategy OPT, inequality $OPT_{xy}(\sigma) \geq OPT(\sigma_{xy})$ holds. Again we will verify this property in the sequel. Therefore, it will be convenient to study $A(\sigma_{xy})$ instead of analyzing the (online or offline) cost $A_{xy}(\sigma)$.

For the analysis of $A(\sigma_{xy})$ we will often partition σ_{xy} into *phases* such that each phase ends with a long run; the last phase ends with the last request of σ_{xy} if the last run happens to be short. Let p_{xy} denote the number of phases in this partition and let $\pi(i)$ be the ith phase, $1 \leq i \leq p_{xy}$. If $\pi(i)$ starts with a request to x, then the phase has one of the following two structures, depending on whether the last request of the phase is to x or y.

$$\text{(a) } (xy)^k x^l \quad k \geq 0,\ l \geq 1 \qquad\qquad \text{(b) } (xy)^k y^l \quad k \geq 1,\ l \geq 0$$

If $\pi(i)$ starts with a request to y, the structures are symmetric. If the phase ends with a long run, we have $k \geq 0, l \geq 2$ in case (a) and $k \geq 1, l \geq 1$ in case (b).

Based on this phase partitioning, we introduce some definitions regarding the beginning and end of σ_{xy} and A's list configuration. Let $f_b(\sigma_{xy})$ be equal to 1 if the item first requested in σ_{xy} precedes the other item of $\{x, y\}$ in the initial list; otherwise let $f_b(\sigma_{xy}) = 0$. Moreover, let $f'_b(\sigma_{xy})$ be equal to 1 if $f_b(\sigma_{xy}) = 1$ and σ_{xy} starts with a short run followed by a long run; otherwise $f'_b(\sigma_{xy}) = 0$. Finally, let $f_e(\sigma_{xy})$ be equal to 1 if

the last phase consists of a single request and the referenced item is stored after the other item of $\{x, y\}$ in the current list when A processes σ_{xy} on the two-item list and reaches the beginning of the last phase; otherwise $f_e(\sigma_{xy}) = 0$. Table 2 summarizes these additional parameters.

As an example, let $\sigma_{xy} = xyyyxxy$ and assume that A is the *Move-To-Front* algorithm. Suppose that in the initial list x is stored before y. In this case $f_b(\sigma_{xy}) = 1$ and $f'_b(\sigma_{xy}) = 1$. Moreover, $f_e(\sigma_{xy}) = 1$ because immediately before *Move-To-Front* serves the last run, x is stored before y in the current list.

Indicator	Value
$f_b(\sigma_{xy})$	1 if item requested first is in front
$f'_b(\sigma_{xy})$	1 if additionally σ_{xy} starts with short run followed by long run
$f_e(\sigma_{xy})$	1 if σ_{xy} ends with short run and item is in the back

Table 2: Summary of the additional parameters for σ_{xy}. Again they are to be summed over all pairs $x \neq y$.

Again we sum these definitions over item pairs. For any $x, y \in L$ with $x \neq y$ and for any value $v \in \{f_b, f'_b, f_e\}$, let $v(\sigma) = \sum_{\{x,y\} \subseteq L, x \neq y} v(\sigma_{xy})$. We remark that values $v(\sigma)$, with $v \in \{f_b, f'_b, f_e\}$, will be needed to properly evaluate the cost of our investigated algorithms. However, when determining the performance of online algorithms relative to the optimal offline strategy, many of the terms will cancel. Furthermore, we observe that values $v(\sigma)$ with $v \in \{r, s, l, l_c, l_p, l_i\}$ may grow arbitrarily large as the length of σ increases, whereas the other values $v(\sigma)$ with $v \in \{f_b, f'_b, f_e\}$ are bounded by $|L|(|L| - 1)/2$.

3.2 The cost of an optimal offline algorithm

We lower bound the cost incurred by an optimal offline algorithm OPT on a request sequence σ.

Lemma 1 *The cost incurred by OPT is at least $OPT(\sigma) \geq \frac{1}{2}(r(\sigma) + l_c(\sigma) + f_e(\sigma)) - f_b(\sigma) + |\sigma|$.*

Proof. We first argue that $OPT_{xy}(\sigma) \geq OPT(\sigma_{xy})$ holds for any fixed pair of items $x, y \in L$ with $x \neq y$, cf. also [30]. Suppose that we serve σ_{xy} on the two-item list consisting of x and y by mimicking OPT's behavior when servicing σ on the entire list of all items in L. More specifically, we change the relative order of x and y in the two-item list, using free or paid exchanges, whenever OPT does so in the entire list. This service schedule incurs a cost of $OPT_{xy}(\sigma)$ in the partial cost model, and the latter value cannot be smaller than the optimal cost $OPT(\sigma_{xy})$ of serving σ_{xy} on the two-item list. Hence we find $OPT_{xy}(\sigma) \geq OPT(\sigma_{xy})$.

Using (1) we find $OPT(\sigma) \geq \sum_{\{x,y\} \subseteq L, x \neq y} OPT(\sigma_{xy}) + |\sigma|$. In the following we consider a fixed pair of items $x, y \in L$ with $x \neq y$ and will prove

$$OPT(\sigma_{xy}) \geq \frac{1}{2}(r(\sigma_{xy}) + l_c(\sigma_{xy}) + f_e(\sigma_{xy})) - f_b(\sigma_{xy}). \tag{2}$$

Summing (2) over all pairs of items and taking into account that $OPT(\sigma) \geq \sum_{\{x,y\} \subseteq L, x \neq y} OPT(\sigma_{xy}) + |\sigma|$, we derive the lemma. An optimal offline algorithm for serving σ_{xy} on a two-item list consisting of x and y is easy to state: On the first request of each long run move the requested item to the front of the list. On any other request, do not change the position of the referenced item. No paid exchanges are used.

We will show that, essentially, $OPT(\sigma_{xy}) \geq \frac{1}{2}(r(\sigma_{xy}) + l_c(\sigma_{xy}))$. However, in order to establish a correct and accurate lower bound, we have to consider the list configuration at the beginning and end of σ_{xy}, which is captured by $f_b(\sigma_{xy})$ and $f_e(\sigma_{xy})$, see again Table 2.

In order to analyze $OPT(\sigma_{xy})$ we partition σ_{xy} into phases as described in Section 3.1. We analyze an arbitrary phase $\pi(i)$, $1 \leq i \leq p_{xy}$, and assume w.l.o.g. that the phase starts with a request to item x. Recall that $\pi(i)$ has one of the following two structures: (a) $(xy)^k x^l$ with $k \geq 0,\ l \geq 1$ or (b) $(xy)^k y^l$ with $k \geq 1,\ l \geq 0$.

If σ_{xy} consists of a single request to item, say x, then (2) is easy to see. We have $r(\sigma_{xy}) = 1$ and $l_c(\sigma_{xy}) = 0$. If x is stored in front of y in the initial list, then the cost incurred by OPT is 0. Since $f_b(\sigma_{xy}) = 1$ and $f_e(\sigma_{xy}) = 0$, the right hand side of (2) is in fact smaller than 0. On the other hand, if x is stored after y in the initial list, then the cost incurred by OPT is 1. We have $f_b(\sigma_{xy}) = 0$ and $f_e(\sigma_{xy}) = 1$ such that the right hand side of (2) is also equal to 1. In the remainder of this analysis we assume that σ_{xy} consists of more than one request.

For any i with $1 \leq i \leq p_{xy}$, let $r(\pi(i))$ be the number of runs in $\pi(i)$. Furthermore, let $l_c(\pi(i)) = 1$ if phase i ends with a long run and this long run represents a long run change. Otherwise we set $l_c(\pi(i)) = 0$. A long run change occurs in phase i if and only if the phase has structure (a) with $l \geq 2$: This holds because, if $i \geq 2$, phase $i-1$ ended with a long run of requests to item y and only phase structure (a) ends with x. If $i = 1$, then a long run ending phase 1 is the first long run in σ_{xy} and represents a long run change if the referenced item is equal to the item first referenced in σ_{xy}. Finally, let $OPT(\pi(i))$ denote the cost incurred by OPT when serving $\pi(i)$, $1 \leq i \leq p_{xy}$. We note that if $f_e(\sigma_{xy}) = 1$, then $p_{xy} > 1$ because σ_{xy} consists of more than one request and a last phase containing a single request cannot be equal to the first phase. We will show that for any number i, $1 < i < p_{xy}$,

$$OPT(\pi(i)) \geq \frac{1}{2}(r(\pi(i)) + l_c(\pi(i))) \tag{3}$$

as well as

$$OPT(\pi(1)) \geq \frac{1}{2}(r(\pi(1)) + l_c(\pi(1))) - f_b(\sigma_{xy}) \tag{4}$$

and, if $p_{xy} > 1$,

$$OPT(\pi(p_{xy})) \geq \frac{1}{2}(r(\pi(p_{xy})) + l_c(\pi(p_{xy})) + f_e(\sigma_{xy})). \tag{5}$$

If σ_{xy} consists of only one phase, then the desired inequality (2) follows from (4), taking into account that $f_e(\sigma_{xy}) = 0$. If σ_{xy} consists of at least two phases, then we obtain (2) by summing (3), for all $i = 2, \ldots, p_{xy} - 1$, as well as (4) and (5).

We first analyze any phase that is not equal to the first phase and prove inequalities (3) and (5). Consider a phase i with $1 < i \leq p_{xy}$. Recall that the previous phase $i-1$ ended with a long run of requests to item y such that x is stored at position two in the list when phase i starts. If $f_e(\sigma_{xy}) = 1$, then $\pi(p_{xy})$ consists of a single request to x and (5) is easy to see: We have $l_c(\pi(p_{xy})) = 0$. Furthermore OPT's cost is 1, which is equal to $\frac{1}{2}(r(\pi(p_{xy})) + f_e(\sigma_{xy}))$. Therefore we may assume $f_e(\sigma_{xy}) = 0$ when considering phase number p_{xy} and the proof inequalities (3) and (5) reduces to showing $OPT(\pi(i)) \geq \frac{1}{2}(r(\pi(i)) + l_c(\pi(i)))$.

If phase i has structure (a), then the cost incurred by OPT is equal to $k+1$ because each request to x in the phase prefix $(xy)^k x$ costs 1. If $l \geq 2$, then OPT moves x to the front of the list on the first request of the long run x^l and no further cost is incurred. The number of runs in the phase is $2k+1$. If the phase ends with a long run, we have a long run change, i.e. $l_c(\pi(i)) = 1$. If the phase ends with a short run, i.e. $l = 1$, then $l_c(\pi(i)) = 0$. We obtain $OPT(\pi(i)) = k+1 = \frac{1}{2}(2k+1+1) \geq \frac{1}{2}(r(\pi(i)) + l_c(\pi(i)))$. If phase i has structure (b), then OPT pays a cost of k because each request to x costs 1. The number of runs in the phase is $2k$ and $OPT(\pi(i)) = k = \frac{1}{2}(2k) = \frac{1}{2}(r(\pi(i)) + l_c(\pi(i)))$ because no long run change occurred and hence $l_c(\pi(i)) = 0$.

We finally take care of the first phase. If x is stored after y in the initial list, then $f_b(\sigma_{xy}) = 0$ and the arguments presented in the last paragraph immediately carry over. Recall that $l_c(\pi(1)) = 1$ if and only if the phase has structure (a) with $l \geq 2$. We obtain $OPT(\pi(1)) \geq \frac{1}{2}(r(\pi(1)) + l_c(\pi(1))) = \frac{1}{2}(r(\pi(1)) + l_c(\pi(1))) - f_b(\sigma_{xy})$. If x is stored in front of y in the initial list, then $f_b(\sigma_{xy}) = 1$. We have $OPT(\pi(1)) = \lfloor r(\pi(1))/2 \rfloor$ because runs alternate between items x and y. The latter expression is at least $\frac{1}{2}(r(\pi(1)) + l_c(\pi(1))) - f_b(\sigma_{xy})$ because $f_b(\sigma_{xy}) = 1$. □

3.3 Online algorithms

We first study deterministic online algorithms and then address randomized strategies. The most popular online algorithm for list update is *Move-To-Front*.

Algorithm Move-To-Front (MTF): Move the requested item to the front of the list.

Lemma 2 *The cost incurred by MTF is $MTF(\sigma) = r(\sigma) - f_b(\sigma) + |\sigma|$.*

Proof. We first argue that $MTF_{xy}(\sigma) = MTF(\sigma_{xy})$ holds for any item pair x, y with $x \neq y$. When MTF serves a request sequence, at any time item x precedes item y in the current list if and only if the last request made to an item from $\{x, y\}$ was to x rather than to y. This holds when MTF serves σ on the entire list consisting of all the items in L as well as when MTF serves σ_{xy} on the two-item list consisting of x and y only. Sequence σ consists of $|\sigma_{xy}|$ requests to x and y. Thus, for any i with $1 \leq i \leq |\sigma_{xy}|$, the ith request made to an item from $\{x, y\}$ in σ contributes 1 in $MTF_{xy}(\sigma)$ if and only if the ith request in σ_{xy} incurs a cost of 1 in $MTF(\sigma_{xy})$. Note that MTF does not use paid exchanges. We conclude, as desired $MTF_{xy}(\sigma) = MTF(\sigma_{xy})$.

Hence, using (1), we find $MTF(\sigma) = \sum_{\{x,y\} \subseteq L, x \neq y} MTF(\sigma_{xy}) + |\sigma|$. Consider a fixed pair of items $x, y \in L$ with $x \neq y$. We prove $MTF(\sigma_{xy}) = r(\sigma_{xy}) - f_b(\sigma_{xy})$. Summing this equation over all pairs of items and using $MTF(\sigma) = \sum_{\{x,y\} \subseteq L, x \neq y} MTF(\sigma_{xy}) + |\sigma|$, we obtain the lemma. When *MTF* serves σ_{xy} on the two-item list, on the first request of each run the referenced item is moved to the front of the list. Hence on each run, except for possibly the first one, MTF incurs a cost of exactly 1. On the first run, the cost is 1 if the requested item is stored behind the other item of $\{x, y\}$ in the initial list. The cost is 0 if the requested item precedes the other item of $\{x, y\}$ in the initial list and in this case $f_b(\sigma_{xy}) = 1$. These arguments yield the equation to be proven. □

For any request sequence σ, let $\alpha(\sigma) = (|\sigma| - f_b(\sigma))/r(\sigma)$. Furthermore, let $\beta(\sigma) = l_c(\sigma)/r(\sigma)$ be the fraction of the long run changes relative to the total number of runs. The following theorem gives a refined bound on the performance ratio of MTF. It implies, in particular, that MTF is 2-competitive.

Theorem 1 *For any request sequence σ, the cost incurred by MTF is at most $\frac{2+2\alpha(\sigma)}{1+2\alpha(\sigma)+\beta(\sigma)}$ times that payed by OPT.*

Proof. Applying Lemmas 1 and 2 we find that the ratio of the cost incurred by MTF to that payed by OPT is upper bounded by

$$\begin{aligned} c &= \frac{r(\sigma) - f_b(\sigma) + |\sigma|}{\frac{1}{2}(r(\sigma) + l_c(\sigma) + f_e(\sigma)) - f_b(\sigma) + |\sigma|} \leq \frac{r(\sigma) - f_b(\sigma) + |\sigma|}{\frac{1}{2}(r(\sigma) + l_c(\sigma)) - f_b(\sigma) + |\sigma|} \\ &= \frac{2 + 2(|\sigma| - f_b(\sigma))/r(\sigma)}{1 + l_c(\sigma)/r(\sigma) + 2(|\sigma| - f_b(\sigma))/r(\sigma)} = \frac{2 + 2\alpha(\sigma)}{1 + 2\alpha(\sigma) + \beta(\sigma)}. \end{aligned}$$

□

An immediate consequence of the above theorem is the following corollary. It implies that MTF can achieve a competitiveness as low as 1 on request sequences that exhibit a high degree of locality, i.e. that satisfy λ-locality with values of λ close to 1.

Corollary 1 *On request sequences exhibiting λ-locality, MTF achieves a competitive ratio of at most $\frac{2}{1+\lambda}$.*

A second important deterministic online strategy is *Timestamp*. The algorithm is used, in particular, to construct the best randomized online strategy currently known.

Algorithm Timestamp (TS): Insert the requested item, say x, immediately in front of the first item in the list that precedes x in the current list and was requested at most once since the last request to x. If there is no such item or if x is requested for the first time, do not change the position of x.

Lemma 3 *The cost incurred by TS is $TS(\sigma) \leq r(\sigma) + l_i(\sigma) - l_p(\sigma) + f'_b(\sigma) + |\sigma|$.*

Proof. Again, we first argue that $TS_{xy}(\sigma) = TS(\sigma_{xy})$ for any item pair x, y with $x \neq y$. In the following paragraph we will show that when TS serves a request sequence σ on the full list, the referenced item, say x, never passes an item in the list that was requested at least twice since the last request to x. Thus the following property holds: When TS serves the ith request to x or y in σ, the relative order of the two items in the full list is the same as that in the two-item list consisting of x and y when TS serves the ith request in σ_{xy}, $1 \leq i \leq |\sigma_{xy}|$. We conclude $TS_{xy}(\sigma) = TS(\sigma_{xy})$.

It remains to show that when TS serves a request sequence σ on the full list, the referenced item, say x, never passes an item in the list that was requested at least twice since the last request to x. We prove the statement by induction on t. Consider an arbitrary request sequence σ. The statement holds at time $t = 1$ because the referenced item $\sigma(1)$ is requested for the first time and TS does not change its position in the list. Suppose that the desired statement holds up to time $t - 1$, and let $x = \sigma(t)$ be the item referenced at time t. If x is requested for the first time, then again the position of x in the list does not change. Otherwise let y be the first item in the current list that precedes x and was requested at most once since the last request to x. (In case no such y exists, the position of x remains unchanged.) Let z be any item that was requested at least twice since the last request to x. The last three requests made to either y or z are of the following form: (a) $z \ldots z \ldots y$; (b) $z \ldots y \ldots z$ or (c) $y \ldots z \ldots z$. In each case z precedes y in the current list after the above subsequence of requests. In case (a) this follows from induction hypothesis. In cases (b) and (c) this holds because on requests to z this item passes all items that were referenced at most once since the last request to z. We conclude that when x is inserted in front of y, item z is not passed.

Thus, using (1), we obtain

$$TS(\sigma) = \sum_{\substack{\{x,y\} \subseteq L \\ x \neq y}} TS(\sigma_{xy}) + |\sigma|. \tag{6}$$

We consider a fixed pair of items $x, y \in L$ with $x \neq y$ and will prove

$$TS(\sigma_{xy}) \leq s(\sigma_{xy}) + 2l_i(\sigma_{xy}) + f'_b(\sigma_{xy}). \tag{7}$$

As $r(\sigma_{xy}) = s(\sigma_{xy}) + l(\sigma_{xy}) = s(\sigma_{xy}) + l_p(\sigma_{xy}) + l_i(\sigma_{xy})$ we find $TS(\sigma_{xy}) \leq r(\sigma_{xy}) + l_i(\sigma_{xy}) - l_p(\sigma_{xy}) + f'_b(\sigma_{xy})$. Summing the latter inequality over all pairs of items, taking into account (6), we obtain the lemma.

For the analysis of $TS(\sigma_{xy})$ we partition σ_{xy} again into phases as described in Section 3.1. We consider an arbitrary phase $\pi(i)$, $1 \leq i \leq p_{xy}$, and assume w.l.o.g. that the phase starts with a request to item x. Recall that phase $\pi(i)$ has one of the following two structures.

$$\text{(a)}\ \ (xy)^k x^l \quad k \geq 0,\ l \geq 1 \qquad\qquad \text{(b)}\ \ (xy)^k y^l \quad k \geq 1,\ l \geq 0$$

The independent long runs in σ_{xy} are exactly the phases of structure (a) with $k = 0$ and $l \geq 2$. When TS serves σ_{xy} on the two-item list, on the second request of each long run the algorithm moves the referenced item to the front of the list if it is not already there. Hence on each long run TS pays a cost of at most 2. This proves that on all independent long runs, i.e. on all phases of structure (a) with $k = 0$ and $l \geq 2$, TS's total cost is upper bounded by $2l_i(\sigma_{xy})$. In the following we prove that on all other phases, TS incurs a cost of at

most $s(\pi(i))$, where $s(\pi(i))$ denotes the number of short runs in $\pi(i)$. In the first phase the cost is by 1 larger if $f'_b(\sigma_{xy}) = 1$.

Consider an arbitrary phase $\pi(i)$ and suppose first that $i > 1$. At the beginning of the phase item x is stored behind y in the list because the last two requests were made to y. Thus TS incurs a cost of 1 on the first request of the phase. If the phase consists of only one request, we are done because in this case the number of short runs is also 1. Therefore assume that the phase consists of at least two requests. When serving the first request of $\pi(i)$, TS leaves x's position in the list unchanged because y was requested at least twice since the last reference to x. Hence the second request of the phase, which is to y, does not incur cost and we have identified a short run that does not incur cost. We will use this fact in the next paragraph. If the phase is composed of short runs only, we are done because each further short run can incur a cost of at most 1; in fact the cost is exactly 1.

If the phase ends with a long run, then in phase structure (a) we have $k \geq 1$ because $k = 0$ would imply that we deal with an independent long run, which was already analyzed above. Hence, on the first request of the long run x^l TS moves x to the front of the list so that the run incurs a cost of only 1. Recall that the second short run of the phase does not incur any cost. Hence the cost paid by TS in $\pi(i)$ is exactly equal to the number $s(\pi(i))$ of short runs. If the phase ends with a long run and has structure (b), then the suffix y^l does not incur cost: This holds true if $k = 1$ because, as argued above, x is not moved in front of y on the first request of the phase. If $k > 1$, then on the last request of $(xy)^k$ TS moves y in front of x in the list because x was referenced at most once since the last request to x. Hence TS pays a cost of $2k - 1$ in the phase, which is equal to the number of short runs.

We finally study the first phase $\pi(1)$. The arguments presented in the previous paragraph immediately carry over if x is stored behind y in the initial list because on the first request to x algorithm TS does not change the position of x in the list and a second request of the phase, which is to y, does not incur cost. Therefore assume that x is stored in front of y in the initial list. In this case the first run of the phase does not incur cost, and we have again identified a run not causing any cost. If the phase is composed of short runs only, we are done because the service cost of TS in $\pi(1)$ is upper bounded by $s(\pi(1))$. Therefore suppose that the phase ends with a long run. This implies that the phase consists of at least two runs because a single long run would represent an independent long run, whose cost was studied before. If the phase is composed of two runs, then its structure is $xy^{l'}$, for some $l' \geq 2$. TS pays a cost of 2 in the phase because on the first request to y it does not change the position of y in the list as the item is referenced for the first time. On the second request to y, the item is moved to the front of the list and incurs no further cost in the phase. This cost of 2 is equal to the number of short runs plus $f'_b(\sigma_{xy})$.

If the phase is composed of at least three runs, the item referenced in the final long run ρ was requested at least once before and hence moves to the front of the list after the first request of ρ, if not already there, because the other item of the pair was requested only once in between. Thus the final long run incurs a cost of at most 1. Recalling that the first run of $\pi(1)$ does not generate cost, we conclude that TS's total cost in $\pi(1)$ is bounded by the number of short runs of the phase. □

We observe that TS is better than MTF if the number of prefixed long runs is larger than the number of independent long runs plus $f'_b(\sigma)$.

Recall that $\alpha(\sigma) = (|\sigma| - f_b(\sigma))/r(\sigma)$ and $\beta(\sigma) = l_c(\sigma)/r(\sigma)$. Furthermore, let $\alpha'(\sigma) = (f_b(\sigma) + f'_b(\sigma))/r(\sigma)$ and $\gamma(\sigma) = (l_i(\sigma) - l_p(\sigma))/r(\sigma)$.

Theorem 2 *For any request sequence σ, the cost incurred by TS is at most $\frac{2+2\alpha(\sigma)+2\alpha'(\sigma)+2\gamma(\sigma)}{1+2\alpha(\sigma)+\beta(\sigma)}$ times that payed by OPT.*

Proof. Applying Lemmas 1 and 3 we find that the ratio of the cost incurred by TS to that payed by OPT is

upper bounded by

$$\begin{aligned} c &= \frac{r(\sigma)+l_i(\sigma)-l_p(\sigma)+f'_b(\sigma)+|\sigma|}{\frac{1}{2}(r(\sigma)+l_c(\sigma)+f_e(\sigma))-f_b(\sigma)+|\sigma|} \leq \frac{r(\sigma)+l_i(\sigma)-l_p(\sigma)+f'_b(\sigma)+|\sigma|}{\frac{1}{2}(r(\sigma)+l_c(\sigma))-f_b(\sigma)+|\sigma|} \\ &\leq \frac{r(\sigma)+l_i(\sigma)-l_p(\sigma)+f_b(\sigma)+f'_b(\sigma)+|\sigma|-f_b(\sigma)}{\frac{1}{2}(r(\sigma)+l_c(\sigma))+|\sigma|-f_b(\sigma)} \\ &= \frac{2+2(l_i(\sigma)-l_p(\sigma))/r(\sigma)+2(f_b(\sigma)+f'_b(\sigma))/r(\sigma)+2(|\sigma|-f_b(\sigma))/r(\sigma)}{1+l_c(\sigma)/r(\sigma)+2(|\sigma|-f_b(\sigma))/r(\sigma)} \\ &= \frac{2+2\alpha(\sigma)+2\alpha'(\sigma)+2\gamma(\sigma)}{1+2\alpha(\sigma)+\beta(\sigma)}. \end{aligned}$$

□

We have $l_i(\sigma) \leq l_c(\sigma)$ and $f'_b(\sigma) \leq f_b(\sigma) \leq |L|(|L|-1)/2$. Therefore, $\gamma(\sigma) \leq \beta(\sigma)$ and $\alpha'(\sigma)$ is upper bounded by a constant that is independent of σ. Hence the above theorem also yields that TS is 2-competitive. On the other hand, $l_p(\sigma)$ can be 0 and $l_i(\sigma)$ can be as high as $l_c(\sigma)$. In this case $\gamma(\sigma) = \beta(\sigma)$. Thus our refined analysis does not yield an improved competitive ratio for TS on request sequences satisfying λ-locality.

We next turn to randomized algorithms and consider the popular *Bit* strategy.

Algorithm BIT: Maintain a bit $b(x)$ for each item $x \in L$. These bits are initialized independently and uniformly at random to a value in $\{0,1\}$. On a request, complement the bit of the referenced item. If the bit value changes to 1, move the item to the front of the list.

For the proof of Lemma 4 below we need the following claim.

Claim 1 *Assume that BIT serves σ_{xy} on a list consisting of x and y only.*

(1) Suppose that x is stored after y in the two-item list and that x is requested. Then the expected cost of the following request is equal to $1/2$.

(2) Suppose that BIT has just served a subsequence xyx of requests. If the next request is to y, then BIT's expected cost on that request is equal to $3/4$. If the next request is to x, the expected cost is equal to $1/4$.

Analogous statements hold if the roles of x and y are interchanged.

Proof. Consider any item x. After BIT has served i requests to x, the bit $b(x)$ has value $(b_0(x)+i) \bmod 2$, where $b_0(x)$ is the initial bit value. Thus, whenever BIT serves a request to x, the value of $b(x)$ is equally likely to be 0 or 1. This was also shown in [35].

To verify part (1) of the claim we simply observe that when serving the request to x, with probability $1/2$ BIT moves x to the front of the list. Thus, after the request, x precedes y in the list with probability exactly $1/2$. To verify part (2) we remark that after the service of the subsequence xyx, item x precedes y in the list if and only if (a) x is moved to the front of the list on the second request to x or if (b) x is not moved to the front of the list on the second request to x (such that it was moved to the front on the first request to x) and y is not moved to the front on its reference in xyx. Event (a) occurs with probability $1/2$ while (b) occurs with probability $1/4$. □

Lemma 4 *The expected cost incurred by BIT is $BIT(\sigma) \leq \frac{3}{4}r(\sigma)+\frac{1}{4}l(\sigma)+\frac{1}{2}l_i(\sigma)+\frac{1}{4}f_e(\sigma)+|\sigma|$.*

Proof. The question whether or not BIT moves a requested item, say x, to the front of the list only depends on the initial bit value $b_0(x)$ and the number of requests to x processed so far. On the ith request to x, the item

is moved to the front of the list if $(b_0(x) + i) \bmod 2 = 1$. Thus when BIT serves the ith request to either x or y in σ, item x precedes y in the full list if and only if x precedes y in the two-item list when BIT serves the ith request in σ_{xy}, $1 \leq i \leq |\sigma_{xy}|$. Hence $BIT_{xy}(\sigma) = BIT(\sigma_{xy})$, for any pair $x, y \in L$ with $x \neq y$, and we obtain

$$BIT(\sigma) = \sum_{\substack{\{x,y\} \subseteq L \\ x \neq y}} BIT(\sigma_{xy}) + |\sigma|. \tag{8}$$

Fix an item pair $x, y \in L$ with $x \neq y$. We will prove that the expected cost incurred by BIT is

$$BIT(\sigma_{xy}) \leq \frac{3}{4} r(\sigma_{xy}) + \frac{1}{4} l(\sigma_{xy}) + \frac{1}{2} l_i(\sigma_{xy}) + \frac{1}{4} f_e(\sigma_{xy}). \tag{9}$$

We obtain the lemma by summing the last inequality over all item pairs, taking into account (8).

We evaluate $BIT(\sigma_{xy})$ by partitioning, as usual, σ_{xy} into phases as described in Section 3.1. Consider an arbitrary phase $\pi(i)$, $1 \leq i \leq p_{xy}$, and assume w.l.o.g. that the phase starts with a request to item x. Phase $\pi(i)$ has one of the following two structures.

$$\text{(a) } (xy)^k x^l \quad k \geq 0,\ l \geq 1 \qquad\qquad \text{(b) } (xy)^k y^l \quad k \geq 1,\ l \geq 0$$

If σ_{xy} consists of a single request to item, say x, we can easily verify (9): If x is stored in front of y in the initial list, then the cost incurred by BIT is 0 and the right hand side of (9) is lower bounded by 0. On the other hand, if x is stored after y in the initial list, then the cost incurred by BIT is 1. The right hand side of (9) is also equal to 1 because $r(\sigma_{xy}) = 1$ and $f_e(\sigma_{xy}) = 1$ while the other terms are 0.

In the remainder of this proof we assume that σ_{xy} consists of more than one request and analyze the various phases. For any i with $1 \leq i \leq p_{xy}$, let $r(\pi(i))$ be the number of runs in $\pi(i)$. Furthermore, let $l(\pi(i))$ be equal to 1 if the phase contains a long run; otherwise $l(\pi(i)) = 0$. Finally, we set $l_i(\pi(i))$ equal to 1 if the phase consists of an independent long run; otherwise $l_i(\pi(i)) = 0$. Let $BIT(\pi(i))$ denote the expected cost incurred by BIT in the phase. Note that if $f_e(\sigma_{xy}) = 1$, then $p_{xy} > 1$ because the request sequence consists of more than one request and the first phase cannot consist of a single request. We will show that for any phase number i satisfying $1 \leq i < p_{xy}$,

$$BIT(\pi(i)) \leq \frac{3}{4} r(\pi(i)) + \frac{1}{4} l(\pi(i)) + \frac{1}{2} l_i(\pi(i)) \tag{10}$$

and for index p_{xy},

$$BIT(\pi(p_{xy})) \leq \frac{3}{4} r(\pi(p_{xy})) + \frac{1}{4} l(\pi(p_{xy})) + \frac{1}{2} l_i(\pi(p_{xy})) + \frac{1}{4} f_e(\sigma_{xy}). \tag{11}$$

If σ_{xy} consists of only one phase, then the desired inequality (9) follows from (11). If σ_{xy} consists of at least two phases, we derive (9) by summing (10), for all $i = 1, \ldots, p_{xy} - 1$, and (11).

If $i = p_{xy}$ and $f_e(\sigma_{xy}) = 1$, then (11) follows easily: Phase $\pi(p_{xy})$ consists of a single request to an item, say x. The cost incurred by BIT is 1, and this is equal to $\frac{3}{4} r(\pi(p_{xy})) + \frac{1}{4} f_e(\sigma_{xy})$. Inequality (11) follows because $l(\pi(p_{xy})) = l_i(\pi(p_{xy})) = 0$. Therefore, when analyzing the last phase indexed $i = p_{xy}$, we may assume $f_e(\sigma_{xy}) = 0$ and proving (10) and (11) reduced to showing

$$BIT(\pi(i)) \leq \frac{3}{4} r(\pi(i)) + \frac{1}{4} l(\pi(i)) + \frac{1}{2} l_i(\pi(i)) \tag{12}$$

for any i with $1 \leq i \leq p_{xy}$. We first analyze a phase $\pi(i)$, where either (a) $i \geq 2$ or (b) $i = 1$ and $f_b(\sigma_{xy}) = 0$ hold. Then we study the remaining case that $i = 1$ and $f_b(\sigma_{xy}) = 1$.

So consider a phase $\pi(i)$, where either $i \geq 2$ or $i = 1$ combined with $f_b(\sigma_{xy}) = 0$ hold. We first argue that when $\pi(i)$ starts, x is stored behind y in the list. This obviously holds if $i = 1$ because $f_b(\sigma_{xy}) = 0$

indicates that the item first requested in σ_{xy} and hence in the first phase is stored after the other item in the initial list. If $i \geq 2$, then phase $\pi(i)$ is preceded by a long run of requests to y. After the service of the second request of this long run, y definitely precedes x in the list. Hence the first request of $\pi(i)$ incurs a cost of 1 and using Claim 1, part (1) we obtain that the second request of the phase costs $1/2$. If the phase has structure (a) with $k = 0$, the total cost of the phase is $3/2$, which is equal to $\frac{3}{4}r(\pi(i)) + \frac{1}{4}l(\pi(i)) + \frac{1}{2}l_i(\pi(i))$ because $r(\pi(i)) = l(\pi(i)) = l_i(\pi(i)) = 1$ as $\pi(i)$ represents an independent long run. Inequality (12) holds.

Therefore we may assume $k \geq 1$ in both structure (a) and (b), which implies $l_i(\pi(i)) = 0$. Again, the first two requests xy in the phase cause a total cost of $3/2$. Using Claim 1, part (2) we find that BIT pays an expected cost of $3/4$ for the service of any further short run in the phase. If the phase consists of only short runs, then BIT's expected cost is $\frac{3}{4}r(\pi(i))$ and (12) holds because $l(\pi(i)) = 0$. So suppose that $\pi(i)$ ends with a long run. If the phase has structure (a), then Claim 1, part (2) implies that the first two requests of the suffix x^l incur a cost of $3/4+1/4$. On the second request of this long run x is moved to the front of the list, if x is not already there, and the run incurs no further cost. Thus $BIT(\pi(i)) = \frac{3}{2}k + 1 = \frac{3}{4}r(\pi(i)) + \frac{1}{4}l(\pi(i))$ because $l(\pi(i)) = 1$. Inequality (12) holds. If the phase has structure (b), then Claim 1, part (2) with the roles of x and y interchanged implies that the first request of y^l incurs an expected cost of $1/4$. As y is moved to the front of the list, if not already there, the run incurs no further cost. We obtain $BIT(\pi(i)) = \frac{3}{2}k+\frac{1}{4} = \frac{3}{4}r(\pi(i))+\frac{1}{4}l(\pi(i))$ because, again, $l(\pi(i)) = 1$.

We finally have to study the first phase $\pi(1)$ if $f_b(\sigma_{xy}) = 1$, i.e. the item first requested in $\pi(1)$ precedes the other item of the pair in the initial list. In this case the first request of the phase incurs a cost of 0. If $\pi(1)$ consists of a single long run, then BIT's total cost in the phase is 0. Inequality (12) holds because the right-hand side is lower bounded by 0. So assume that $\pi(1)$ consists of at least two runs, which implies $l_i(\pi(1)) = 0$. In this case the second request of the phase, which is to y, incurs a cost of 1. If the phase consists of exactly two runs, then $\pi(1) = xy^l$ where $l \geq 1$, and using Claim 1, part (1) we obtain that BIT's expected cost is at most $3/2$, which in turn is upper bounded by $\frac{3}{4}r(\pi(1)) + \frac{1}{4}l(\pi(1))$ because $r(\pi(1)) = 2$ and $l(\pi(1)) = 1$. We finally focus on the case that $\pi(1)$ is composed of at least three runs. In this case we can simply consider a truncated phase $\pi'(1)$ derived from $\pi(1)$ by removing the first request. In this truncated phase a final long run is still preceded by a short run and the item first requested in $\pi'(1)$ precedes the other item of the pair when the phase starts. Applying all the arguments of the previous paragraph we obtain (12), taking into account that the original phase $\pi(1)$ contains even one run more than $\pi'(1)$. □

Again, $\alpha(\sigma) = (|\sigma|-f_b(\sigma))/r(\sigma)$ and $\beta(\sigma) = l_c(\sigma)/r(\sigma)$. Define $\delta(\sigma) = (\frac{1}{2}l(\sigma)+l_i(\sigma)+2f_b(\sigma))/r(\sigma)$.

Theorem 3 *For any request sequence σ, the expected cost incurred by BIT is at most $\frac{1.5+2\alpha(\sigma)+\delta(\sigma)}{1+2\alpha(\sigma)+\beta(\sigma)}$ times that payed by OPT.*

Proof. Applying Lemmas 1 and 4 we find that the ratio of the expected cost incurred by BIT to that payed by OPT is upper bounded by

$$\begin{aligned}
c &= \frac{\frac{3}{4}r(\sigma) + \frac{1}{4}l(\sigma) + \frac{1}{2}l_i(\sigma) + \frac{1}{4}f_e(\sigma) + |\sigma|}{\frac{1}{2}(r(\sigma) + l_c(\sigma) + f_e(\sigma)) - f_b(\sigma) + |\sigma|} \\
&\leq \frac{\frac{3}{4}r(\sigma) + \frac{1}{4}l(\sigma) + \frac{1}{2}l_i(\sigma) + |\sigma|}{\frac{1}{2}(r(\sigma) + l_c(\sigma)) - f_b(\sigma) + |\sigma|} \\
&\leq \frac{1.5r(\sigma) + \frac{1}{2}l(\sigma) + l_i(\sigma) + 2f_b(\sigma) + 2(|\sigma| - f_b(\sigma))}{r(\sigma) + l_c(\sigma) + 2(|\sigma| - f_b(\sigma))} \\
&= \frac{1.5 + (\frac{1}{2}l(\sigma) + l_i(\sigma) + 2f_b(\sigma))/r(\sigma) + 2(|\sigma| - f_b(\sigma))/r(\sigma)}{1 + l_c(\sigma)/r(\sigma) + 2(|\sigma| - f_b(\sigma))/r(\sigma)} \\
&= \frac{1.5 + 2\alpha(\sigma) + \delta(\sigma)}{1 + 2\alpha(\sigma) + \beta(\sigma)}.
\end{aligned}$$

□

It is not hard to show that the above theorem also implies that BIT is 1.75-competitive. Taking into account that $l(\sigma) \leq r(\sigma)$ and $l_i(\sigma) \leq l_c(\sigma)$, we obtain the following corollary, which yields that BIT attains a competitiveness of 1.5 on request sequences with a high degree of locality, i.e. with values of λ close to 1.

Corollary 2 *On request sequences exhibiting λ-locality, BIT achieves a competitive ratio of* $\min\{1.75, \frac{2+\lambda}{1+\lambda}\}$.

We finally turn to *COMB*, the list update algorithm achieving the smallest competitive ratio currently known.

Algorithm Combination (COMB): With probability $4/5$ serve the request sequence using BIT, and with probability $1/5$ serve the sequence using TS.

Lemma 5 *The expected cost incurred by $COMB$ is $COMB(\sigma) \leq \frac{1}{5}(4r(\sigma)+4l_i(\sigma)+f_e(\sigma)+f'_b(\sigma))+|\sigma|$.*

Proof. By the definition of $COMB$, on any request sequence, the algorithm's expected cost is $\frac{4}{5}BIT(\sigma) + \frac{1}{5}TS(\sigma)$ such that, using Lemmas 3 and 4, we obtain

$$\begin{aligned} COMB(\sigma) &\leq \frac{1}{5}(3r(\sigma)+l(\sigma)+2l_i(\sigma)+f_e(\sigma)) + \frac{1}{5}(r(\sigma)+l_i(\sigma)-l_p(\sigma)+f'_b(\sigma))+|\sigma| \\ &= \frac{1}{5}(4r(\sigma)+l(\sigma)+3l_i(\sigma)-l_p(\sigma)+f_e(\sigma)+f'_b(\sigma))+|\sigma| \\ &= \frac{1}{5}(4r(\sigma)+4l_i(\sigma)+f_e(\sigma)+f'_b(\sigma))+|\sigma|. \end{aligned}$$

The last line follows because $l(\sigma) = l_i(\sigma) + l_p(\sigma)$. □

Let $\zeta(\sigma) = (l_i(\sigma) + \frac{5}{4}f_b(\sigma) + \frac{1}{4}f'_b(\sigma))/r(\sigma)$.

Theorem 4 *For any request sequence σ, the expected cost incurred by COMB is at most $\frac{1.6+2\alpha(\sigma)+1.6\zeta(\sigma)}{1+2\alpha(\sigma)+\beta(\sigma)}$ times that payed by OPT.*

Proof. Using Lemmas 1 and 4 we obtain that the ratio of the expected cost of $COMB$ to that of OPT is upper bounded by

$$\begin{aligned} c &= \frac{\frac{1}{5}(4r(\sigma)+4l_i(\sigma)+f_e(\sigma)+f'_b(\sigma))+|\sigma|}{\frac{1}{2}(r(\sigma)+l_c(\sigma)+f_e(\sigma))-f_b(\sigma)+|\sigma|} \\ &\leq \frac{\frac{1}{5}(4r(\sigma)+4l_i(\sigma)+f'_b(\sigma))+|\sigma|}{\frac{1}{2}(r(\sigma)+l_c(\sigma))-f_b(\sigma)+|\sigma|} \\ &\leq \frac{1.6r(\sigma)+1.6l_i(\sigma)+0.4f'_b(\sigma)+2|\sigma|}{r(\sigma)+l_c(\sigma)-2f_b(\sigma)+2|\sigma|} \\ &= \frac{1.6+1.6(l_i(\sigma)+\frac{5}{4}f_b(\sigma)+\frac{1}{4}f'_b(\sigma))/r(\sigma)+2(|\sigma|-f_b(\sigma))/r(\sigma_{xy})}{1+l_c(\sigma)/r(\sigma)+2(|\sigma|-f_b(\sigma))/r(\sigma_{xy})} \\ &= \frac{1.6+2\alpha(\sigma)+1.6\zeta(\sigma)}{1+2\alpha(\sigma)+\beta(\sigma)}. \end{aligned}$$

□

We have $l_i(\sigma) \leq l_c(\sigma)$ and $f'_b(\sigma) \leq f_b(\sigma) \leq |L|(|L|-1)/2$. Hence the above theorem also implies that $COMB$ is 1.6-competitive. Since $l_i(\sigma)$ can be as high as $l_c(\sigma)$, Theorem 4 does not give an improved competitive ratio for $COMB$ on request sequences satisfying λ-locality.

4 Experimental study

In this section we report on the results of our experimental study in which we have implemented the algorithms analyzed in Section 3. The main purpose of our study is to compare the experimentally observed performance of the algorithms to the bounds stated in Theorems 1–4 as well as Corollaries 1 and 2. In order to get meaningful results we have tested the algorithms on real-world request sequences from benchmark libraries. Clearly, self-organizing linear lists represent a solution to the classic dictionary problem, where we have to maintain a set of elements so as to efficiently perform search and possible update operations. When a dictionary is maintained by a computer/CPU, requests are actually memory accesses. A second important application of self-organizing linear lists is data compression, where we wish to store a file using few bits. A first, general finding is that the experimental results are consistent for both data sets. In fact, the results are even slightly more positive for the memory access traces.

Since data compression has gained considerable popularity and importance, as far as the application of list update algorithms is concerned, we report on the corresponding results first and then present the results for the memory access strings.

4.1 Data compression

Bentley et al. [13] showed that self-organizing linear lists can be used to build locally adaptive data compression schemes. The approach was further developed and studied in [2, 16, 28]. The best compression results are achieved when the scheme is combined with the famous Burrows-Wheeler transformation [16], yielding compression rates that are comparable or even better than that of Lempel-Ziv based schemes. In fact the common open source data compression program `bzip2` consists of a Burrows-Wheeler transformation followed by Move-To-Front and Huffman encodings. We briefly describe the basic data compression scheme by Bentley et al. [13] and the Burrow-Wheeler transformation [16].

Data compression schemes: In data compression we are given a string S that shall be *compressed*, i.e. that shall be represented using fewer bits. The string S consists of *symbols*, where each symbol is element of an alphabet $X = \{x_1, \ldots, x_n\}$. The idea of data compression schemes using linear lists is to convert the string S of symbols into a string I of integers. An *encoder* maintains a linear list of symbols contained in X and reads the symbols in the string S. Whenever the symbol x_i has to be compressed, the encoder looks up the current position of x_i in the linear list, outputs this position and updates the list using a list update algorithm. Here one can use MTF or any other list update strategy. If symbols to be compressed are moved closer to the front of the list, then frequently occurring symbols can be encoded with small integers. Clearly, when the string I is actually stored or transmitted, each integer in the string should be coded again using a variable length prefix code.

The refined compression scheme by Burrows and Wheeler first applies a transformation to the string S. The purpose of this transformation is to group together instances of a symbol x_i occurring in S so that the resulting string S' exhibits a high degree of locality of reference. Of course the transformation is reversible so that, given S', the original string can be recovered. We refer the reader to [16] for details on the Burrows-Wheeler transformation and its efficient implementation. The transformed string S' is then encoded using the algorithm by Bentley et al. [13] as described in the previous paragraph. Alphabet X, i.e. the set of entries in the self-organizing linear list, is the ASCII alphabet with its 256 different characters. The initial list is given by the initial numerical order of the ASCII characters. In the string S, each byte represents a symbol. For large files to be compressed, the Burrows-Wheeler transformation is applied not to the entire file but rather to blocks of uniform size; the block size may be chosen by a user.

Data sets and their locality characteristics: In our experiments we selected files available at the repository named *Canterbury Corpus* [17]. This collection was developed as an extension of the widely-used *Calgary Corpus* and represents the standard benchmark library for evaluating data compression algorithms. It

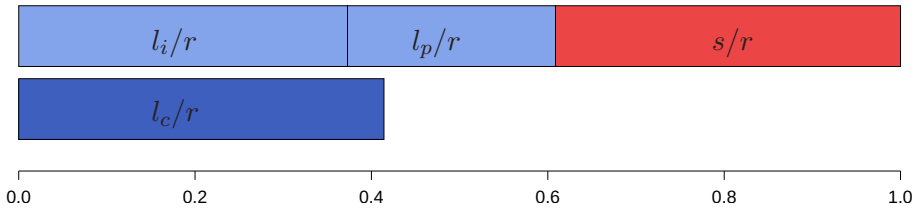

Figure 1: Average distribution of the runs into short runs and long runs and of the long runs into independent and prefixed long runs. Furthermore, the average ratio of long run changes is given.

consists of different corpora such as the true *Canterbury Corpus*, the *Large Corpus* and the *Calgary Corpus*. The corpora contain files of different types. In addition to text files such as books and papers there are source code files, pictures, office documents, object files, and many more. A description of the corpora can be found at [17].

In our tests we selected all the files from the Canterbury Corpus, the Large Corpus and the Calgary Corpus. We applied the Burrows-Wheeler transformation to each of these files. We chose a block size of $9 \cdot 10^5$ bytes, which is the default and also the maximum allowable block size in `bzip2`. If the file size exceeds the block size, then, as described above, the file is split into several blocks of the chosen size. This was the case for all the files of the Large Corpus. The sequences obtained from the Burrows-Wheeler transformation are the request sequences on which the list update algorithms have to be evaluated. Recall that each byte of the sequence forms a request. We remark that we actually do not compress files; instead we evaluate list update algorithms on these realistic benchmark sequences.

Table 3 in Appendix A shows the characteristics of our (transformed) request sequences. A graphical summary is depicted in Figure 1. The length of the request sequences differs vastly among the test instances. There are short sequences consisting of only 3721 requests (`grammar.lsp`) and long sequences of up to $9 \cdot 10^5$ references, which occur when a file is split into several blocks. The third column of Table 3 shows how many blocks were generated. Moreover, the number of different bytes (ASCII characters) requested differs vastly. In text files typically about 80 to 90 different characters are requested. In object files such as `obj1` and `obj2` of the Calgary Corpus all the 256 ASCII characters are referenced.

For each of the request sequences we have computed the values r, s, l, l_p, l_i, l_c as introduced in Section 2. We have also computed values f_b, f'_b, f_e defined in Section 3.1 but, for brevity, do not show them. For better reading, instead of giving the absolute values of r, s, l, l_p, l_i, l_c we show the interesting relations. Table 3 contains the exact values for each file while Figure 1 depicts the average values over all the files. First, it shows that among all the runs of a request sequence, about 60% to 65% are long runs. The fraction can go as high as 80% in the case of file `trans` in the Calgary Corpus. An exception is the file `kennedy.xls` in which only 10%–15% of the runs are long, indicating that the Burrows-Wheeler transformation does not work well on this file. Columns 7 and 8 of Table 3 and Figure 1 show the distribution of the prefixed and independent long runs among all the long runs. Again, in most cases, the majority of the long runs are independent long runs (fractions can go as high as 80%), indicating that long runs are usually followed by long runs. Finally, the last column of Table 3 and Figure 1 show the number of long run changes relative to the number of long runs. Here the interesting observation is that the ratio l_c/l is typically 5% to 10% higher than l_i/l. This demonstrates that the definition of l_c was indeed sensible in Section 2 as it yields more expressive lower bounds on the cost of OPT, compared to l_i. A final remark is that, for files split into several blocks, all the numbers for the various blocks are consistent, i.e. the characteristics do not change within the file.

Performance results: We have executed the online algorithms analyzed in Section 3 on all the request sequences described above and recorded their cost. As for the randomized strategies BIT and $COMB$, they were executed 16 times on each sequence and, for any sequence, the average cost was taken. Since the

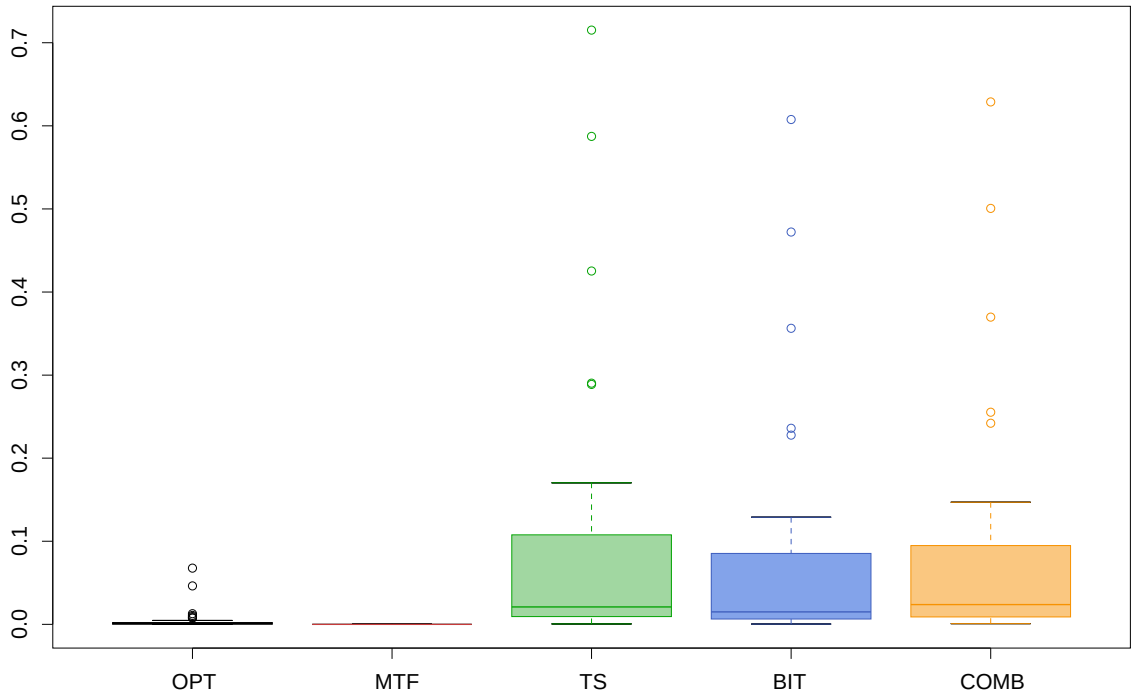

Figure 2: Relative errors of the upper bounds on the service costs given by Lemmas 1–5 when compared to the actual costs observed.

offline version of the list update problem is NP-hard [5] and the best known offline algorithm takes $O(2^n n! m)$ time [34], where $n = |L|$ and $m = |\sigma|$, computing the true optimum offline cost is impossible for our request sequences. Therefore, we computed the *pairwise optimum*, which is equal to $\sum_{\{x,y\} \subseteq L, x \neq y} OPT(\sigma_{xy}) + |\sigma|$. This expression is usually used as approximation of the optimum offline cost, even in standard competitive analysis, see e.g. [1]. To evaluate the pairwise optimum, for each request sequence σ and each pair x, y of ASCII characters we derived the projected sequence σ_{xy} and computed $OPT(\sigma_{xy})$. The latter cost is easy to determine because an optimal offline algorithm for sequences on two-item lists is simple to state, cf. proof of Lemma 1. In the following figures and tables, OPT always refers to the pairwise optimum.

Table 4 in Appendix A and Figure 2 above represent the costs incurred by the algorithms on all the sequences. For each algorithm we compare the costs observed in the experiments to those implied by the theoretical bounds of Lemmas 1–5. For any $A \in \{MTF, TS, BIT, COMB, OPT\}$, the columns headed A^* in Table 4 record the experimentally observed cost. Using values $r, s, l, l_i, l_p, l_c, f_b, f'_b, f_e$ for the various request sequences, Lemmas 1–5 yield theoretical bounds. Instead of reporting these bounds, we give the relative errors. The relative error, for a given strategy A and request sequence σ, is the absolute value of the difference between the experimental and theoretical costs, divided by the experimental value. Figure 2 depicts the relative errors using box plots, which is a standard method to display numerical data. For each algorithm, the bold line within the box represents the median data point. The box includes 50% of the data points, where 25% is located above and 25% below the median. The upper (respectively lower) whisker is the maximum (respectively minimum) data point that can be found within a distance of 1.5 of the inter-quartile range. All other points are outliers. It shows that all the errors are very small, indicating that the bounds developed in Lemmas 1–5 very well approximate the experimentally observed cost. As for OPT, or the pairwise optimum, the average relative error is below 0.5%. The incurred error for MTF is 0 because the bound given in Lemma 2 is exact. For the other three online strategies, the average relative errors are between

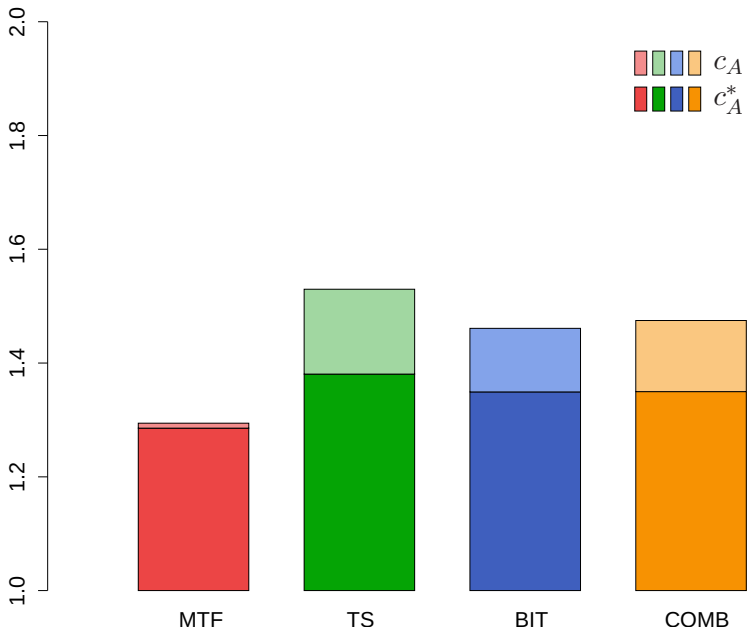

Figure 3: Average upper bounds c_A on the performance ratios as implied by Theorems 1–4 compared to the average experimentally observed competitiveness c_A^*.

7% and 8%; the medians are even below 2%. Observe that the errors cannot be 0 because Lemmas 3–5 give upper bounds on the service cost of the respective algorithms.

For comparison with previous experimental studies [9, 10] Table 5 shows the average service cost incurred by the algorithms on a single request. These average costs are all very small, typically in the range between 1.8 and 5, which confirms that the request sequences exhibit a high degree of locality and that the algorithms respond well to this property. In files such as `kennedy.xls`, `geo` or `obj1`, which access the full ASCII character set, the access cost is a bit higher.

Table 6 in Appendix A as well as Figures 3 and 4 contain the main results of our experimental study. Table 6 presents, for each online algorithm and each of our request sequences, the experimentally observed competitiveness and compares it to the theoretical bounds developed in Theorems 1–4. For each algorithm $A \in \{MTF, TS, BIT, COMB\}$, expression c_A^* refers to the experimentally observed competitiveness on a given sequence, which is the actual cost incurred by A divided by the cost of the pairwise optimum; the latter values are reported in Table 4. Expression c_A refers to the theoretical bound. For each algorithm, the average performance values, over all files, are depicted in dark color (exp3rimental bounds) and light color (theoretical bounds) in Figure 3. Table 6 also shows the relative error between the experimental and theoretical expressions; a graphical representation using box plots is given in Figure 4. A first, very positive finding is that the experimentally observed and theoretical performance ratios are very close to each other. Hence our locality model and theoretical analyses are indeed sensible. The best results are achieved for MTF. Here the average relative error is below 0.7%. For almost all of the files, the actual error is substantially smaller because four files in the Canterbury Corpus (`fields.c`, `grammar.lsp`, `sum` and `xargs.1`) contribute very higher errors to the average value. For the other three online algorithms the average relative errors are higher, ranging between 8% and 10%, but these values are still reasonable. Again, for many files we have very small errors; high contributions in the average relative errors just come from the files `fields.c`, `grammar.lsp` and `xargs.1`. Figure 4 illustrates that the median error values are even below 3%.

A second important result is that the experimentally observed competitiveness as well as the performance

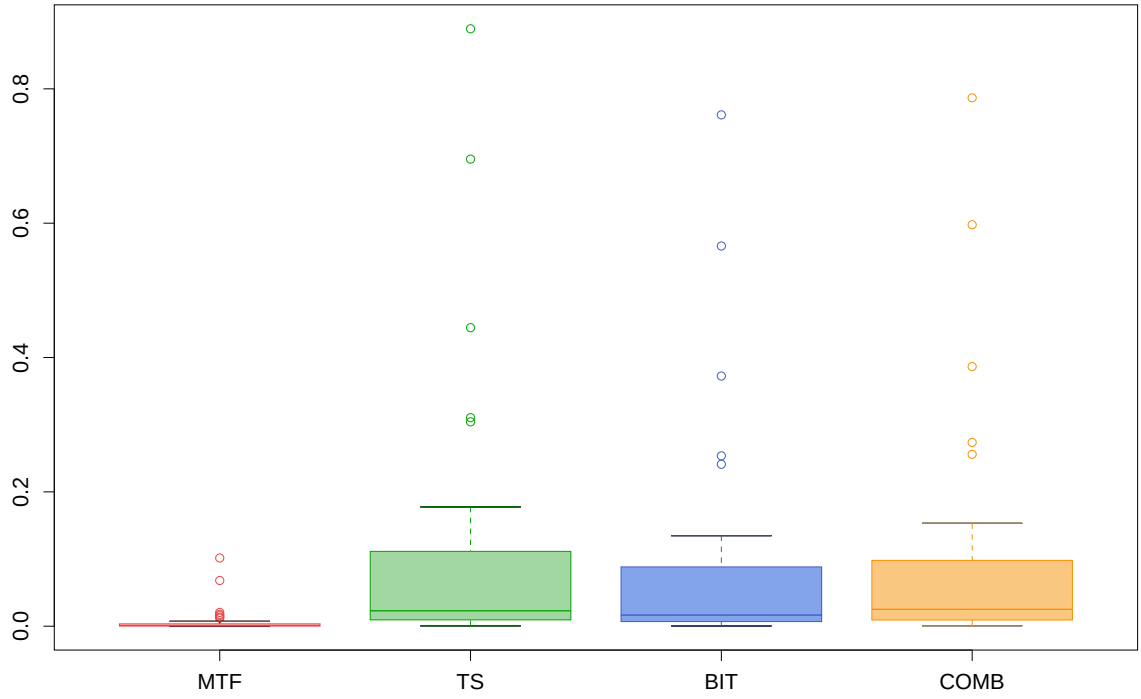

Figure 4: Relative errors of the upper bounds on the performance ratios as implied by Theorems 1–4 when compared to the experimentally observed competitiveness.

ratios implied by Theorems 1–4 are much lower than the standard competitive ratios of the algorithms. Recall that MTF and TS are 2-competitive while BIT and $COMB$ achieve competitive ratios of 1.75 and 1.6, respectively. In our experiments, MTF shows the best behavior with performance ratios between 1.2 and 1.3. An exception is the file `kennedy.xls` where the transformed request sequences do not exhibit a substantial degree of locality. The other three algorithms TS, BIT and $COMB$ are slightly worse, with ratios that are typically in the range between 1.3 and 1.6. There is no clear winner among these three algorithms. For each strategy there are some sequences where this strategy outperforms the other two. We note that for a few files such as `fields.c`, `grammar.lsp` or `xargs.1`, the theoretical bounds implied by Theorem 2–4 are higher than the standard competitive ratios of the algorithms. This is no contradiction. On these relatively short files, consisting of only a few thousands of requests, the initial, apparently unfortunate list ordering has a high influence of the overall service cost. The terms $f_b(\sigma)$ and $f'_b(\sigma)$, which take into account such effects, are ignored in classical competitive analysis that evaluates the performance of algorithms on long sequences.

Table 7 in Appendix A and Figure 5 report on the performance of MTF and BIT in terms of λ-locality. As argued in Section 3.3, TS and $COMB$ achieve no improved competitiveness and hence are not listed. For algorithm $A \in \{MTF, BIT\}$, expression c_A^λ refers to the theoretical bounds implied by Corollaries 1 and 2. We compare this value to the experimentally observed competitiveness c_A^*, reported in Table 6, and compute relative errors. In order to get a reasonable set of numerical results, which can also be compared to that of Table 6, we treat each sequence as being a representative member of a class of inputs. (Of course, alternatively, one could also group sequences to form classes.) For both MTF and BIT the competitive performance under λ-locality is higher than the theoretical bounds of Theorems 1 and 3. This is not surprising because Corollaries 1 and 2 consider asymptotic algorithm performance, ignoring the request sequence length $|\sigma|$, i.e. an additive 1 per request, in both the online and offline cost. However, since the average service cost of the algorithms is small (cf. Table 5), the additive values of 1 make a difference in performance. For

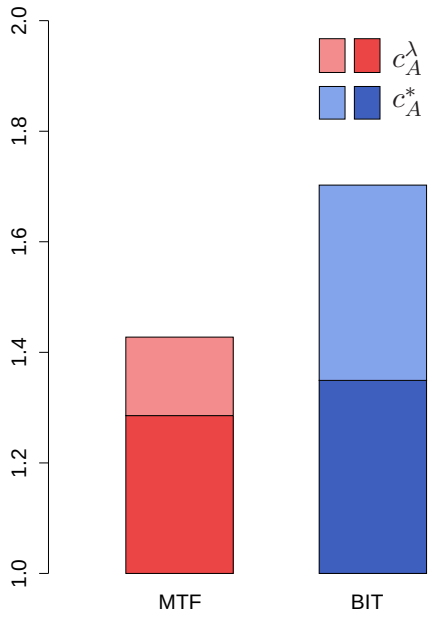

Figure 5: Average bounds on the performance ratios c_A^λ for $A \in \{MTF, BIT\}$ as implied by Corollaries 1 and 2. For comparison, the corresponding average values of the experimentally observed ratios are represented by the darker colored bars.

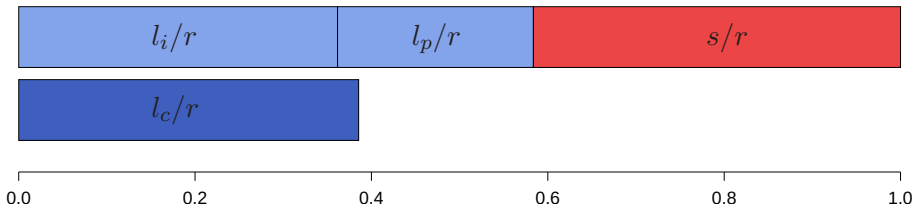

Figure 6: Average distribution of the runs of the memory access traces into short runs and long runs and of the long runs into independent and prefixed long runs. Furthermore, the average ratio of long run changes is given.

MTF we have competitive values, in terms of λ-locality, that are typically between 1.2 and 1.5, which is still considerably lower than the standard competitive ratio of 2. For BIT the values are usually above 1.6 and often reach even 1.75.

4.2 Memory accesses

We now report on our experiments with memory accesses, thereby simulating dictionary operations. In these request sequences, each request is an access to a memory page. The alphabet is the set of different pages ever referenced in a given sequence. In contrast to the data compression experiments, where the list of 256 ASCII characters was fixed, here our self-organizing lists grow dynamically. Whenever a page is referenced for the first time, its address is inserted as new item into the list. This is simply done by appending the new item at the end of the list and executing the chosen list update strategy assuming a search request was issued.

Data sets and their locality characteristics: We selected traces that were generated by SPEC (Standard Performance Evaluation Corporation) which maintains widely-used benchmark collections for performance

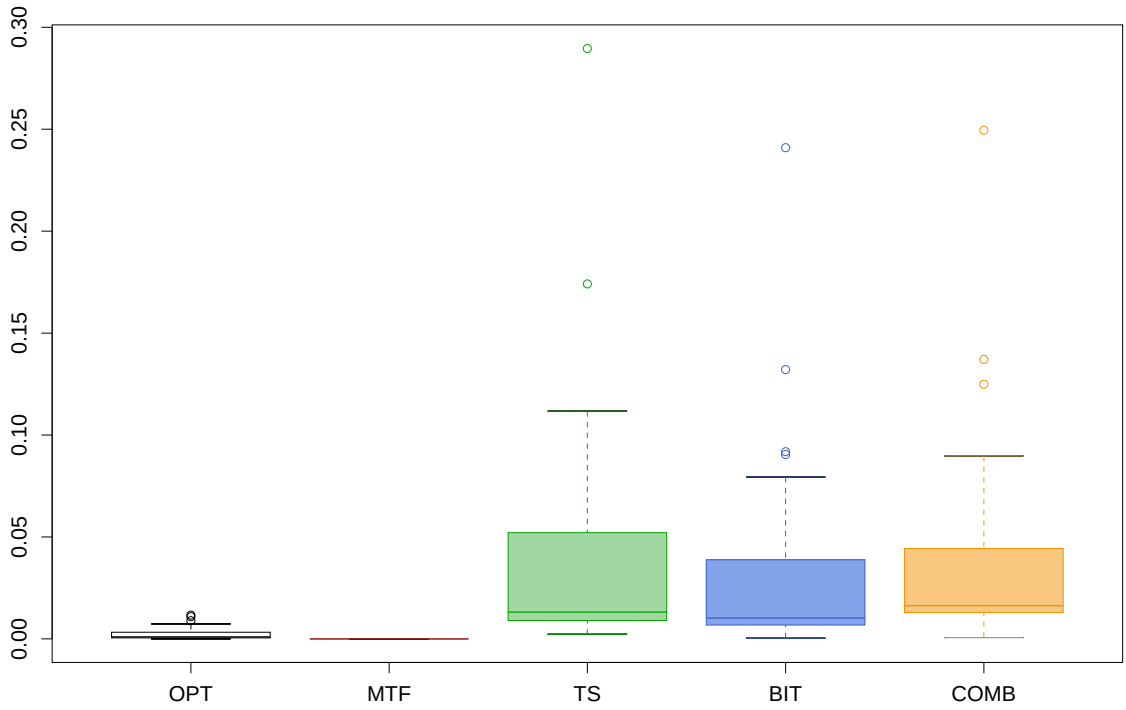

Figure 7: Relative errors of the upper bounds on the service costs on the memory access traces as implied by Lemmas 1–5.

evaluation [40]. The traces were generated by execution of the SPEC CPU2000 benchmark suite that contains a large variety of programs ranging from compilers, to word processors, to computer visualization programs, to name just a few. The programs were executed on a Pentium II processor running Redhat Linux 6.0. The resulting set of 47 memory access traces can be downloaded at the Bringham Young University Trace Distribution Center [15]. Instead of working with the whole traces which partly contain several hundreds of millions of requests, we considered the samples of the traces provided. These samples consist of the first section of the traces up to nearly 10.5 million requests. More information about the traces can be found at [40].

Table 8 in Appendix B and Figure 6 show the characteristics of the traces. The alphabet size differs vastly for the various traces and ranges from 251 (`ammp`) to 3151 (`perl makerand`). Interestingly, the locality characteristics are very similar to those of the data compression traces. This confirms the known fact that memory accesses do exhibit locality of reference. Among the runs of a sequence, typically about 60% are long runs. Recall that the numbers were about 10% to 15% higher in the case of data compression. In our test set only file `mesa` forms an exception with a fraction of 60% of short runs. Among the long runs, the majority forms independent long runs (fractions can be as high as 75%). The ratio l_c/l is typically 5% higher than l_i/l. We recall that in data compression this increase as well as the proportion of independent long runs was a bit higher. The reason is that the Burrows-Wheeler transformation generates sequences with a high degree of locality.

Performance results: As usual we have executed all our algorithms on each of the traces. The randomized algorithms were again executed 16 times on each input and the average cost was recorded. Table 9 in Appendix B and Figure 7 present the actual service costs of the algorithms and the relative errors incurred by the corresponding theoretical bounds of Lemmas 1–5. The average error of OPT, or the pairwise optimum, is less than 0.2%. As for MTF, the error is 0 because our theoretical bound is exact. For the other three algorithms the average relative error is around 3%, and hence even lower than in the case of the data compression

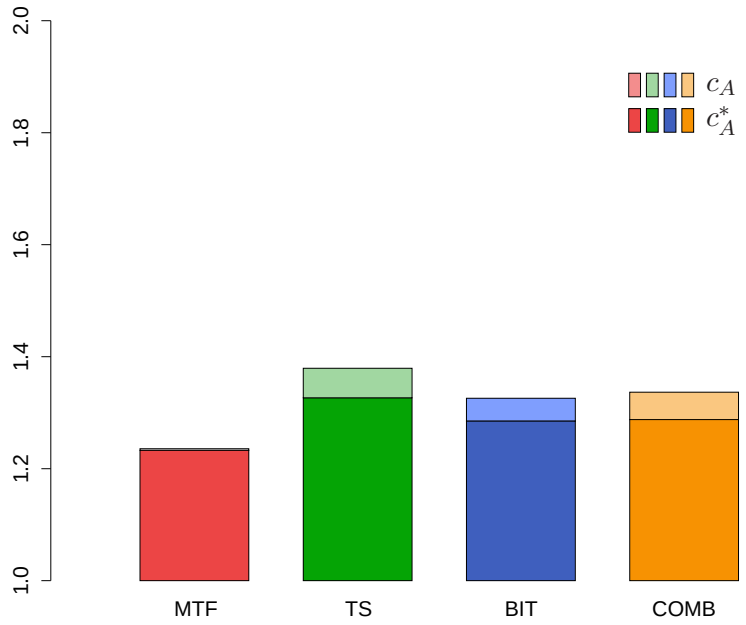

Figure 8: Average upper bounds c_A on the performance ratios on the memory access traces, as implied by Theorems 1–4, compared to the average experimentally observed competitiveness c_A^*.

traces. The median error values are below 2%, cf. Figure 7. Table 10 depicts the average service cost per request of the algorithms. We observe very small values between 2 and 3 uniformly over all the algorithms and traces. Again these values are even slightly lower than those of the data compression traces.

The most important results are presented in Table 11 in Appendix B as well as in Figures 8 and 9 below, which compare the experimentally observed competitiveness of the algorithms to the performance guarantees of Theorems 1–4. The relative errors are also computed. These errors are surprisingly small and again even lower than in the experiments with the data compression traces. For MTF we observe an average relative error of 0.2%. For the other algorithms the average relative errors are between 3% and 4%; the media values are below 2%. Furthermore, the performance ratios of the algorithms are very low. Algorithm MTF exhibits theoretical and experimental performance ratios that are typically in the range between 1.2 and 1.25. Hence these values are substantially smaller than MTF's competitiveness of 2. For the other three algorithms the performance ratios are a bit higher, typically ranging between 1.25 and 1.4. There are two exceptions, namely files `mcf` and `perl makerand` where the ratios are higher. In general, BIT appears to be slightly better than TS and $COMB$ but the difference is marginal. For all the latter three algorithms, too, the experimentally observed and theoretical performance ratios are well below the competitive ratios of the strategies.

Finally Table 12 in Appendix B and Figure 10 below show the results for λ-locality. We assume again that each trace is a representative sample from a given application. The table shows the competitiveness of the algorithms, expressed in terms of λ, and the relative errors when the value is compared to the experimentally observed competitiveness shown in Table 12. As in the case of the experiments with data compression traces, MTF exhibits performance guarantees that are well below the competitive ratio of 2; for BIT the guarantees are in the range 1.65–1.75 and hence not substantially below the standard competitiveness of 1.75. We remark again that the performance ratios in Table 12 are higher than the bounds of Table 11 because Corollaries 1 and 2 ignore the request sequence length $|\sigma|$, i.e. an additive 1 per request, which makes a difference because the average cost of the algorithms is very low. Since the average service cost on the memory traces is even lower than on the data compression traces, the relative errors here are higher.

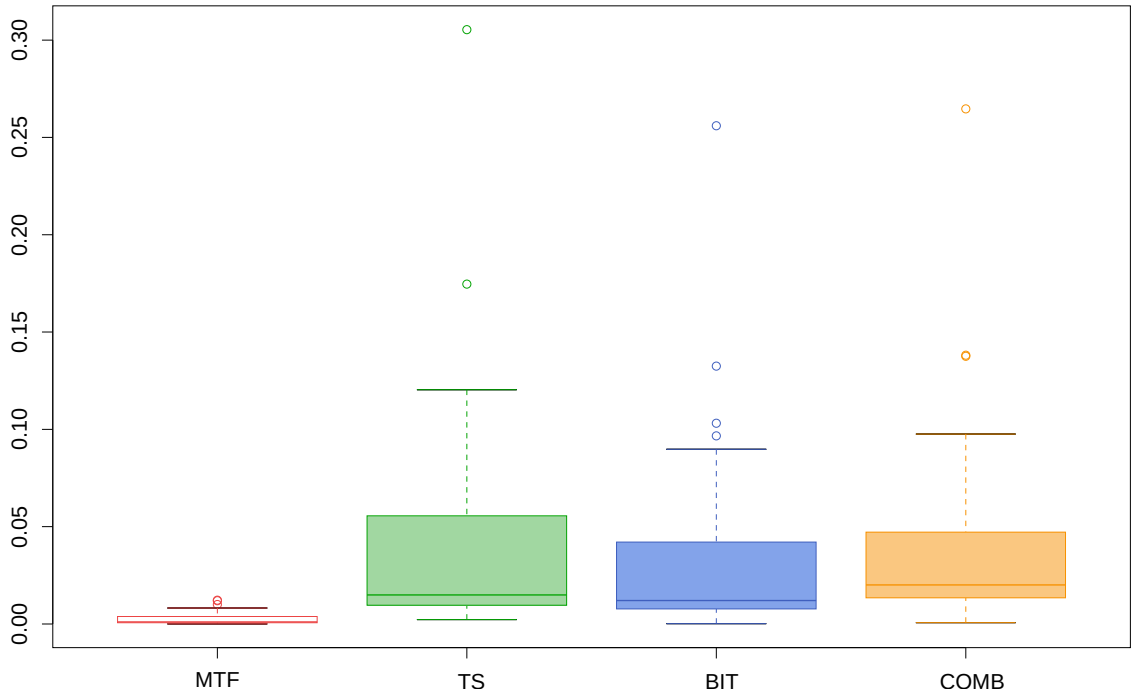

Figure 9: Relative errors of the upper bounds on the performance ratios on the memory access traces, as implied by Theorems 1–4.

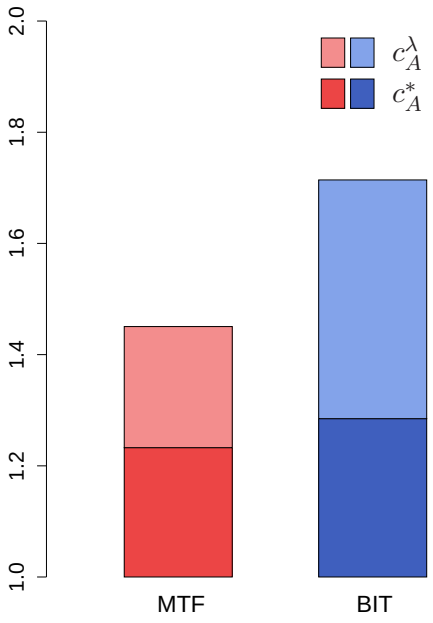

Figure 10: Average bounds on the performance ratios c_A^λ for $A \in \{MTF, BIT\}$ for the memory access traces as implied by Corollaries 1 and 2. For comparison, the corresponding average values of the experimentally observed ratios are represented by the darker colored bars.

Overall we observe the same qualitative results as in the tests with the data compression sequences. As mentioned before the results for the memory access traces are even more positive, i.e. the average relative errors between the theoretical an experimental bounds are smaller.

5 Conclusions

In this paper we have presented a new, powerful model of locality of reference that allows us analyze list update algorithms. We have developed refined theoretical performance guarantees for popular online algorithms. A main result is that *Move-To-Front* responds very well to locality of reference, which does not hold true for the other strategies examined. We have complemented the theoretical investigations by an extensive experimental study. It shows that the theoretically proven and experimentally observed performance guarantees match up to very small relative errors. We conjecture that, with respect to the competitive ratios, the gap between the theoretical and experimental bounds could be tightened even further if it were possible to derive better lower bounds on the optimum offline cost. More generally, an interesting working direction is to apply our locality model to other data structures problems. A natural candidate are splay trees [39]. A challenge is to analyze the cost of the search tree operations in terms of locality parameters.

References

[1] S. Albers. Improved randomized on-line algorithms for the list update problem. *SIAM Journal on Computing*, 27:670–681, 1998.

[2] S. Albers and M. Mitzenmacher. Average case analyses of list update algorithms, with applications to data compression. *Algorithmica*, 21:312–329, 1998.

[3] S. Albers, L.M. Favrholdt and O. Giel. On paging with locality of reference. *Journal of Computer and System Sciences*, 70:145–175, 2005.

[4] S. Albers, B. von Stengel and R. Werchner. A combined BIT and TIMESTAMP algorithm for the list update problem. *Information Processing Letters*, 56:135–139, 1995.

[5] C. Ambühl. Offline list update is NP-hard. *Proc. 8th Annual European Symposium on Algorithms*, Springer LNCS 1879, 42–51, 2000.

[6] C. Ambühl, B. Gärtner and B. von Stengel. A new lower bound for the list update problem in the partial cost model. *Theoretical Computer Science*, 268:3–16, 2001.

[7] S. Angelopoulos, R. Dorrigiv and A. López-Ortiz. On the separation and equivalence of paging strategies. *18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 229–237, 2007.

[8] S. Angelopoulos, R. Dorrigiv and A. López-Ortiz. List update with locality of reference: MTF outperforms all other algorithms. *Proc. 8th Latin American Symposium on Theoretical Informatics*, Springer LNCS 4957, 399-410, 2008.

[9] R. Bachrach and R. El-Yaniv. Online list accessing algorithms and their applications: Recent empirical evidence. *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, 53–62, 1997.

[10] R. Bachrach, R. El-Yaniv and M. Reinstädtler. On the competitive theory and practice of online list accessing algorithms. *Algorithmica*, 32:201–245, 2002.

[11] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11:2–14, 1994.

[12] J.L. Bentley and C.C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Communication of the ACM*, 28:404–411, 1985.

[13] J.L. Bentley, D.S. Sleator, R.E. Tarjan and V.K. Wei. A locally adaptive data compression scheme. *Communication of the ACM*, 29:320–330, 1986.

[14] A. Borodin, S. Irani, P. Raghavan and B. Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50:244–258, 1995.
[15] BYU Trace Distribution Center, `http://tds.cs.byu.edu/tds/index.jsp`
[16] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. DEC SRC Research Report 124, 1994.
[17] The Canterbury Corpus, `http://corpus.canterbury.ac.nz/`
[18] F.R.K. Chung, D.J. Hajela and P.D. Seymour. Self-organizing sequential search and Hilbert's inequality. *Proc. 17th Annual Symposium on the Theory of Computing*, 217–223, 1985.
[19] R. Dorrigiv, M.R. Ehmsen and A. López-Ortiz. Parameterized analysis of paging and list update algorithms. *Algorithmica*, 71(2):330–353, 2015.
[20] R. Dorrigiv and A. López-Ortiz. On certain new models for paging with locality of reference. *Proc. 2nd International Workshop on Algorithms and Computation (WALCOM)*, Springer LNCS 4921, 200–209, 2008.
[21] R. Dorrigiv and A. López-Ortiz. A new perspective on list update: Probabilistic locality and working set. *Proc. 9th International Workshop on Approximation and Online Algorithms*, Springer LNCS 7164, 150–163, 2011.
[22] R. Dorrigiv and A. López-Ortiz. List update with probabilistic locality of reference. *Information Processing Letters*, 112(13):540–543, 2012.
[23] R. Dorrigiv, A. López-Ortiz and J.I. Munro. List update algorithms for data compression. *Proc. IEEE Data Compression Conference (DCC)*, 512, 2008.
[24] R. Dorrigiv, A. López-Ortiz, and J.I. Munro. An application of self-organizing data structures to compression. *Proc. 8th International Symposium on Experimental Algorithms*, Springer LNCS 5526, 137–148, 2009.
[25] A. Fiat and A. Karlin. Randomized and multipointer paging with locality of reference. *Proc. 27th Annual ACM Symposium on Theory of Computing*, 626–634, 1995.
[26] A. Fiat and M. Mendel. Truly online paging with locality of reference. *Proc. 38rd Annual Symposium on Foundations of Computer Science*, 326–335, 1997.
[27] G.H. Gonnet, J.I. Munro and H. Suwanda. Towards self-organizing linear search. *Proc. 19th Annual IEEE Symposium on Foundations of Computer Science*, 169–174, 1979.
[28] D. Grinberg, S. Rajagopalan, R. Venkatesan and V.K. Wei. Splay trees for data compression. *Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, 522–530, 1995.
[29] J.H. Hester and D. S. Hirschberg. Self-organizing linear search. *ACM Computing Surveys*, 17:295–312. 1985.
[30] S. Irani. Two results on the list update problem. *Information Processing Letters*, 38:301–306, 1991.
[31] A. Karlin, S. Phillips und P. Raghavan. Markov paging. *Proc. 33rd Annual Symposium on Foundations of Computer Science*, 24–27, 1992.
[32] R. Karp and P. Raghavan. Personal communication cited in [35], 1990.
[33] E. Koutsoupias and C.H. Papadimitriou. Beyond competitive analysis. *Proc. 35th Annual Symposium on Foundations of Computer Science* 394–400, 1994.
[34] N. Reingold and J. Westbrook. Optimum off-line algorithms for the list update problem. Technical Report YALEU/DCS/TR-805, Yale University, 1990.
[35] N. Reingold, J. Westbrook and D.D. Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11:15–32, 1994.
[36] R. Rivest. On self-organizing sequential search heuristics. *Communication of the ACM*, 19:63–67, 1976.
[37] J. Seward. *bzip2* home page, `http://www.bzip.org/`
[38] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202-208, 1985.

[39] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.
[40] SPEC - Standard Performance Evaluation Corporation, `http://www.spec.org/`
[41] E. Torng. A unified analysis of paging and caching. *Algorithmica*, 20:175-200, 1998.

Appendix A

file	size	block	diff. bytes	s/r	l/r	l_p/l	l_i/l	l_c/l	λ
The Canterbury Corpus									
alice29.txt	152089	1	74	0.3840	0.6160	0.4328	0.5672	0.6327	0.3898
asyoulik.txt	125179	1	68	0.3959	0.6041	0.4447	0.5553	0.6267	0.3786
cp.html	24603	1	86	0.3032	0.6968	0.3196	0.6804	0.7280	0.5072
fields.c	11150	1	90	0.1801	0.8199	0.1546	0.8454	0.8671	0.7109
grammar.lsp	3721	1	76	0.2287	0.7713	0.1328	0.8672	0.8819	0.6802
kennedy.xls	1029744	1	256	0.8817	0.1183	0.6483	0.3517	0.6612	0.0782
		2	236	0.8500	0.1500	0.6026	0.3974	0.5977	0.0896
lcet10.txt	426754	1	84	0.3963	0.6037	0.4607	0.5393	0.6069	0.3664
plrabn12.txt	481861	1	81	0.4336	0.5664	0.5108	0.4892	0.5695	0.3226
ptt5	513216	1	159	0.4434	0.5566	0.5505	0.4495	0.5237	0.2915
sum	38240	1	255	0.3820	0.6180	0.3934	0.6066	0.6740	0.4165
xargs.1	4227	1	74	0.2503	0.7497	0.1801	0.8199	0.8440	0.6328
The Large Corpus									
E.coli	4638690	1	4	0.4930	0.5070	0.5196	0.4804	0.6250	0.3169
		2	4	0.4972	0.5028	0.5224	0.4776	0.6245	0.3140
		3	4	0.4973	0.5027	0.5259	0.4741	0.6206	0.3120
		4	4	0.4946	0.5054	0.5209	0.4791	0.6250	0.3159
		5	4	0.4904	0.5096	0.5172	0.4828	0.6271	0.3195
		6	4	0.4970	0.5030	0.5184	0.4816	0.6312	0.3175
bible.txt	4047392	1	62	0.3971	0.6029	0.4608	0.5392	0.6102	0.3679
		2	62	0.4002	0.5998	0.4665	0.5335	0.6054	0.3631
		3	63	0.4217	0.5783	0.4968	0.5032	0.5815	0.3363
		4	61	0.4046	0.5954	0.4707	0.5293	0.6027	0.3589
		5	63	0.4142	0.5858	0.4797	0.5203	0.5965	0.3494
world192.txt	2473400	1	90	0.3830	0.6170	0.4409	0.5591	0.6270	0.3868
		2	84	0.3793	0.6207	0.4331	0.5669	0.6341	0.3936
		3	88	0.3650	0.6350	0.4151	0.5849	0.6474	0.4111
The Calgary Corpus									
bib	111261	1	81	0.3402	0.6598	0.3675	0.6325	0.6891	0.4547
book1	768771	1	82	0.4391	0.5609	0.5271	0.4729	0.5533	0.3104
book2	610856	1	96	0.3835	0.6165	0.4395	0.5605	0.6250	0.3853
geo	102400	1	256	0.4829	0.5171	0.5446	0.4554	0.5741	0.2969
news	377109	1	98	0.3498	0.6502	0.3964	0.6036	0.6585	0.4282
obj1	21504	1	256	0.4049	0.5951	0.4339	0.5661	0.6453	0.3840
obj2	246814	1	256	0.3433	0.6567	0.3852	0.6148	0.6736	0.4424
paper1	53161	1	95	0.2966	0.7034	0.3035	0.6965	0.7405	0.5208
paper2	82199	1	91	0.3535	0.6465	0.3849	0.6151	0.6729	0.4351
paper3	46526	1	84	0.3323	0.6677	0.3504	0.6496	0.7042	0.4702
paper4	13286	1	80	0.2773	0.7227	0.2674	0.7326	0.7728	0.5585
paper5	11954	1	91	0.2690	0.7310	0.2569	0.7431	0.7811	0.5710
paper6	38105	1	93	0.2757	0.7243	0.2786	0.7214	0.7611	0.5512
pic	513216	1	159	0.4434	0.5566	0.5505	0.4495	0.5237	0.2915
progc	39611	1	92	0.2876	0.7124	0.3004	0.6996	0.7427	0.5291
progl	71646	1	87	0.2652	0.7348	0.2742	0.7258	0.7613	0.5594
progp	49379	1	89	0.2135	0.7865	0.2086	0.7914	0.8179	0.6433
trans	93695	1	99	0.1964	0.8036	0.1949	0.8051	0.8278	0.6652

Table 3: Characteristics of the files and the transformed request sequences. File size (in bytes), partitioning in blocks, number of different bytes per block and statistics on the runs.

file	OPT^*	f_{OPT}	MTF^*	f_{MTF}	TS^*	f_{TS}	BIT^*	f_{BIT}	$COMB^*$	f_{COMB}
The Canterbury Corpus										
alice29.txt	392895	0.0013	502868	0	522306	0.0431	526456.9375	0.0319	524126.3125	0.0371
asyoulik.txt	362574	0.0002	474180	0	487663	0.0436	494001.6875	0.0317	492105.3125	0.0353
cp.html	104158	0.0031	133635	0	151906	0.1704	147630.9375	0.1289	147249.4375	0.1469
fields.c	42524	0.0075	49582	0	65868	0.4253	58897.1250	0.3563	60363.6875	0.3698
grammar.lsp	19391	0.0676	22572	0	30722	0.7153	27150.8125	0.6077	27910.2500	0.6287
kennedy.xls	12472268	0.0010	22354897	0	21597020	0.0012	18080322.1875	0.0004	18274572.2500	0.0285
	939959	0.0047	1621774	0	1558178	0.0250	1340384.5000	0.0202	1356081.5625	0.0423
lcet10.txt	1027207	0.0003	1311293	0	1341239	0.0196	1361943.1875	0.0147	1355707.1250	0.0172
plrabn12.txt	1282558	0.0003	1698414	0	1671403	0.0150	1729810.9375	0.0108	1715711.5000	0.0130
ptt5	942890	0.0019	1180995	0	1131896	0.0224	1186887.9375	0.0155	1177437.5625	0.0155
sum	337864	0.0128	460760	0	505699	0.0771	498365.5625	0.0533	496601.2500	0.0650
xargs.1	23181	0.0463	28233	0	35967	0.5873	33074.9375	0.4722	33568.1875	0.5006
The Large Corpus										
E.coli	1711786	$< 10^{-4}$	2133228	0	2108144	0.0006	2131150.5625	0.0004	2125126.1250	0.0011
	1727309	$< 10^{-4}$	2159514	0	2130515	0.0006	2153816.5625	0.0004	2147982.0000	0.0010
	1717674	$< 10^{-4}$	2146793	0	2113718	0.0006	2139920.1250	0.0005	2133448.0625	0.0011
	1719056	$< 10^{-4}$	2145202	0	2118336	0.0006	2141468.1250	0.0005	2135649.0000	0.0011
	1702239	$< 10^{-4}$	2116237	0	2094276	0.0006	2116540.4375	0.0004	2111176.0625	0.0008
	267290	$< 10^{-4}$	334220	0	329993	0.0037	333196.8750	0.0030	332557.5625	0.0032
bible.txt	1690008	$< 10^{-4}$	2059346	0	2105108	0.0091	2129224.1250	0.0065	2121837.0625	0.0082
	1756850	$< 10^{-4}$	2161595	0	2203033	0.0087	2233457.5625	0.0062	2224340.5000	0.0081
	1837709	$< 10^{-4}$	2308138	0	2303754	0.0085	2360623.0625	0.0060	2346456.5000	0.0077
	1793291	$< 10^{-4}$	2219158	0	2256058	0.0084	2289041.1250	0.0064	2280334.7500	0.0077
	975056	0.0001	1233963	0	1243358	0.0157	1268445.0000	0.0113	1261998.3750	0.0133
world192.txt	2013201	0.0002	2510591	0	2616268	0.0103	2629476.6875	0.0075	2617647.7500	0.0116
	2031383	0.0001	2528823	0	2652556	0.0096	2655963.0000	0.0072	2645196.6875	0.0115
	1558549	0.0002	1933008	0	2057427	0.0129	2046894.1875	0.0098	2040235.3125	0.0148
The Calgary Corpus										
bib	310092	0.0007	388932	0	427473	0.0587	418736.6250	0.0451	418474.8125	0.0529
book1	1998680	0.0003	2651634	0	2582324	0.0096	2689370.6875	0.0069	2662206.8750	0.0096
book2	1559892	0.0002	1986645	0	2076699	0.0138	2086792.3750	0.0103	2075630.1875	0.0155
geo	2499234	0.0020	3797973	0	3615473	0.0098	3785313.6250	0.0062	3780945.8125	0.0010
news	1396286	0.0002	1809619	0	1990178	0.0150	1960421.5625	0.0112	1950057.9375	0.0204
obj1	372864	0.0093	529522	0	554319	0.0739	560958.7500	0.0488	556354.1250	0.0600
obj2	2085629	0.0015	2796806	0	3172590	0.0106	3089426.6250	0.0085	3050721.6250	0.0272
paper1	182349	0.0019	226971	0	265066	0.1097	252272.8750	0.0867	252736.3125	0.1006
paper2	244978	0.0027	313512	0	337465	0.0813	334597.8750	0.0614	334155.4375	0.0687
paper3	157773	0.0024	201921	0	223426	0.1148	218574.8125	0.0877	218969.8750	0.0961
paper4	57123	0.0080	72257	0	84729	0.2888	80642.8750	0.2278	81351.9375	0.2421
paper5	61211	0.0107	77312	0	92995	0.2903	87205.7500	0.2359	87804.1875	0.2553
paper6	138675	0.0018	171298	0	204572	0.1417	192904.6250	0.1122	193633.4375	0.1277
pic	942890	0.0019	1180995	0	1131896	0.0224	1186887.9375	0.0155	1177437.5625	0.0155
progc	154882	0.0017	194241	0	228571	0.1238	217648.1250	0.0966	217821.8125	0.1124
progl	180236	0.0014	213958	0	252752	0.1057	238902.5625	0.0840	240581.9375	0.0935
progp	135749	0.0017	156964	0	199299	0.1364	181676.0000	0.1125	183934.0000	0.1253
trans	245167	0.0018	278104	0	361727	0.0850	325851.5625	0.0709	331187.3750	0.0799

Table 4: Actual service costs incurred by the algorithms. Comparison of the actual costs to the bounds of Lemmas 1–5 yields the relative errors f_A, for the various strategies A.

file	$\overline{OPT^*}$	$\overline{MTF^*}$	$\overline{TS^*}$	$\overline{BIT^*}$	$\overline{COMB^*}$
The Canterbury Corpus					
alice29.txt	2.5833	3.3064	3.4342	3.4615	3.4462
asyoulik.txt	2.8964	3.7880	3.8957	3.9464	3.9312
cp.html	4.2335	5.4317	6.1743	6.0005	5.9850
fields.c	3.8138	4.4468	5.9074	5.2823	5.4138
grammar.lsp	5.2112	6.0661	8.2564	7.2966	7.5007
kennedy.xls	13.8581	24.8388	23.9967	20.0892	20.3051
	7.2447	12.4998	12.0096	10.3310	10.4520
lcet10.txt	2.4070	3.0727	3.1429	3.1914	3.1768
plrabn12.txt	2.6617	3.5247	3.4686	3.5899	3.5606
ptt5	1.8372	2.3012	2.2055	2.3126	2.2942
sum	8.8354	12.0492	13.2243	13.0326	12.9864
xargs.1	5.4840	6.6792	8.5089	7.8247	7.9414
The Large Corpus					
E.coli	1.9020	2.3703	2.3424	2.3679	2.3613
	1.9192	2.3995	2.3672	2.3931	2.3866
	1.9085	2.3853	2.3486	2.3777	2.3705
	1.9101	2.3836	2.3537	2.3794	2.3729
	1.8914	2.3514	2.3270	2.3517	2.3458
	1.9272	2.4098	2.3794	2.4025	2.3978
bible.txt	1.8778	2.2882	2.3390	2.3658	2.3576
	1.9521	2.4018	2.4478	2.4816	2.4715
	2.0419	2.5646	2.5597	2.6229	2.6072
	1.9925	2.4657	2.5067	2.5434	2.5337
	2.1794	2.7581	2.7791	2.8352	2.8208
world192.txt	2.2369	2.7895	2.9070	2.9216	2.9085
	2.2571	2.8098	2.9473	2.9511	2.9391
	2.3144	2.8705	3.0553	3.0396	3.0298
The Calgary Corpus					
bib	2.7871	3.4957	3.8421	3.7636	3.7612
book1	2.5998	3.4492	3.3590	3.4983	3.4629
book2	2.5536	3.2522	3.3997	3.4162	3.3979
geo	24.4066	37.0896	35.3074	36.9660	36.9233
news	3.7026	4.7987	5.2775	5.1986	5.1711
obj1	17.3393	24.6243	25.7775	26.0863	25.8721
obj2	8.4502	11.3316	12.8542	12.5172	12.3604
paper1	3.4301	4.2695	4.9861	4.7455	4.7542
paper2	2.9803	3.8141	4.1055	4.0706	4.0652
paper3	3.3911	4.3400	4.8022	4.6979	4.7064
paper4	4.2995	5.4386	6.3773	6.0698	6.1231
paper5	5.1205	6.4675	7.7794	7.2951	7.3452
paper6	3.6393	4.4954	5.3686	5.0624	5.0816
pic	1.8372	2.3012	2.2055	2.3126	2.2942
progc	3.9101	4.9037	5.7704	5.4946	5.4990
progl	2.5156	2.9863	3.5278	3.3345	3.3579
progp	2.7491	3.1788	4.0361	3.6792	3.7249
trans	2.6166	2.9682	3.8607	3.4778	3.5347

Table 5: Average service cost per request incurred by the algorithms.

file	c_{MTF}	c^*_{MTF}	$f_{c_{MTF}}$	c_{TS}	c^*_{TS}	$f_{c_{TS}}$	c_{BIT}	c^*_{BIT}	$f_{c_{BIT}}$	c_{COMB}	c^*_{COMB}	$f_{c_{COMB}}$
The Canterbury Corpus												
alice29.txt	1.2820	1.2799	0.0016	1.3890	1.3294	0.0449	1.3847	1.3399	0.0334	1.3856	1.3340	0.0387
asyoulik.txt	1.3084	1.3078	0.0004	1.4043	1.3450	0.0441	1.4061	1.3625	0.0320	1.4058	1.3573	0.0357
cp.html	1.2908	1.2830	0.0061	1.7174	1.4584	0.1776	1.6084	1.4174	0.1347	1.6302	1.4137	0.1531
fields.c	1.1816	1.1660	0.0134	2.2373	1.5490	0.4444	1.9008	1.3850	0.3724	1.9681	1.4195	0.3864
grammar.lsp	1.2822	1.1640	0.1015	2.9935	1.5843	0.8894	2.4660	1.4002	0.7612	2.5715	1.4393	0.7866
kennedy.xls	1.7944	1.7924	0.0011	1.7357	1.7316	0.0023	1.4518	1.4496	0.0015	1.5086	1.4652	0.0296
	1.7380	1.7254	0.0073	1.7116	1.6577	0.0325	1.4642	1.4260	0.0268	1.5137	1.4427	0.0492
lcet10.txt	1.2770	1.2766	0.0004	1.3317	1.3057	0.0199	1.3458	1.3259	0.0150	1.3430	1.3198	0.0176
plrabn12.txt	1.3248	1.3242	0.0004	1.3233	1.3032	0.0154	1.3638	1.3487	0.0112	1.3557	1.3377	0.0134
ptt5	1.2570	1.2525	0.0036	1.2317	1.2005	0.0261	1.2821	1.2588	0.0185	1.2720	1.2488	0.0186
sum	1.3918	1.3637	0.0206	1.6453	1.4968	0.0992	1.5818	1.4750	0.0724	1.5945	1.4698	0.0848
xargs.1	1.3009	1.2179	0.0681	2.6305	1.5516	0.6953	2.2343	1.4268	0.5659	2.3135	1.4481	0.5976
The Large Corpus												
E.coli	1.2462	1.2462	$< 10^{-4}$	1.2322	1.2315	0.0006	1.2455	1.2450	0.0004	1.2428	1.2415	0.0011
	1.2502	1.2502	$< 10^{-4}$	1.2341	1.2334	0.0006	1.2475	1.2469	0.0005	1.2448	1.2435	0.0010
	1.2498	1.2498	$< 10^{-4}$	1.2313	1.2306	0.0006	1.2464	1.2458	0.0005	1.2434	1.2421	0.0011
	1.2479	1.2479	$< 10^{-4}$	1.2330	1.2323	0.0006	1.2464	1.2457	0.0005	1.2437	1.2423	0.0011
	1.2432	1.2432	$< 10^{-4}$	1.2310	1.2303	0.0006	1.2438	1.2434	0.0004	1.2413	1.2402	0.0008
	1.2504	1.2504	$< 10^{-4}$	1.2391	1.2346	0.0037	1.2504	1.2466	0.0030	1.2481	1.2442	0.0032
bible.txt	1.2186	1.2185	0.0001	1.2571	1.2456	0.0092	1.2681	1.2599	0.0065	1.2659	1.2555	0.0083
	1.2304	1.2304	$< 10^{-4}$	1.2650	1.2540	0.0088	1.2793	1.2713	0.0063	1.2764	1.2661	0.0081
	1.2561	1.2560	0.0001	1.2643	1.2536	0.0085	1.2923	1.2845	0.0061	1.2867	1.2768	0.0077
	1.2375	1.2375	$< 10^{-4}$	1.2687	1.2581	0.0084	1.2846	1.2764	0.0064	1.2814	1.2716	0.0077
	1.2657	1.2655	0.0002	1.2954	1.2752	0.0159	1.3158	1.3009	0.0114	1.3117	1.2943	0.0135
world192.txt	1.2475	1.2471	0.0003	1.3134	1.2996	0.0107	1.3163	1.3061	0.0078	1.3158	1.3002	0.0119
	1.2451	1.2449	0.0002	1.3186	1.3058	0.0098	1.3170	1.3075	0.0073	1.3173	1.3022	0.0116
	1.2406	1.2403	0.0003	1.3375	1.3201	0.0132	1.3265	1.3133	0.0100	1.3287	1.3091	0.0150
The Calgary Corpus												
bib	1.2556	1.2542	0.0011	1.4610	1.3785	0.0598	1.4126	1.3504	0.0461	1.4223	1.3495	0.0539
book1	1.3272	1.3267	0.0004	1.3049	1.2920	0.0100	1.3553	1.3456	0.0073	1.3453	1.3320	0.0100
book2	1.2741	1.2736	0.0004	1.3502	1.3313	0.0142	1.3520	1.3378	0.0107	1.3517	1.3306	0.0158
geo	1.5244	1.5197	0.0031	1.4653	1.4466	0.0129	1.5282	1.5146	0.0090	1.5156	1.5128	0.0018
news	1.2966	1.2960	0.0004	1.4473	1.4253	0.0154	1.4202	1.4040	0.0115	1.4256	1.3966	0.0208
obj1	1.4453	1.4201	0.0177	1.6248	1.4867	0.0929	1.6017	1.5045	0.0646	1.6063	1.4921	0.0766
obj2	1.3444	1.3410	0.0025	1.5412	1.5212	0.0132	1.4971	1.4813	0.0107	1.5059	1.4627	0.0295
paper1	1.2486	1.2447	0.0032	1.6182	1.4536	0.1132	1.5075	1.3835	0.0897	1.5297	1.3860	0.1037
paper2	1.2843	1.2798	0.0036	1.4948	1.3775	0.0852	1.4544	1.3658	0.0649	1.4625	1.3640	0.0722
paper3	1.2840	1.2798	0.0032	1.5838	1.4161	0.1184	1.5113	1.3854	0.0909	1.5258	1.3879	0.0994
paper4	1.2801	1.2649	0.0120	1.9345	1.4833	0.3042	1.7521	1.4117	0.2411	1.7886	1.4242	0.2559
paper5	1.2828	1.2630	0.0156	1.9910	1.5193	0.3105	1.7860	1.4247	0.2536	1.8270	1.4345	0.2736
paper6	1.2402	1.2352	0.0040	1.6909	1.4752	0.1462	1.5523	1.3911	0.1159	1.5800	1.3963	0.1316
pic	1.2570	1.2525	0.0036	1.2317	1.2005	0.0261	1.2821	1.2588	0.0185	1.2720	1.2488	0.0186
progc	1.2577	1.2541	0.0028	1.6632	1.4758	0.1270	1.5448	1.4053	0.0993	1.5685	1.4064	0.1153
progl	1.1912	1.1871	0.0035	1.5560	1.4023	0.1096	1.4408	1.3255	0.0870	1.4638	1.3348	0.0967
progp	1.1599	1.1563	0.0032	1.6736	1.4681	0.1400	1.4929	1.3383	0.1155	1.5290	1.3550	0.1285
trans	1.1370	1.1343	0.0024	1.6046	1.4754	0.0876	1.4264	1.3291	0.0732	1.4621	1.3509	0.0823

Table 6: Upper bounds c_A on the performance ratios achieved by $A \in \{MTF, TS, BIT, COMB\}$ as implied by Theorems 1–4. The values are compared to the experimentally observed competitiveness c^*_A, which is the actual cost incurred by A to that of the pairwise optimum. The comparison yields a relative error f_{c_A}.

file	λ	c^{λ}_{MTF}	$f_{c^{\lambda}_{MTF}}$	c^{λ}_{BIT}	$f_{c^{\lambda}_{BIT}}$
The Canterbury Corpus					
alice29.txt	0.3898	1.4391	0.1244	1.7195	0.2833
asyoulik.txt	0.3786	1.4508	0.1093	1.7254	0.2664
cp.html	0.5072	1.3270	0.0343	1.6635	0.1736
fields.c	0.7109	1.1689	0.0025	1.5845	0.1440
grammar.lsp	0.6802	1.1903	0.0226	1.5952	0.1393
kennedy.xls	0.0782	1.8549	0.0349	1.7500	0.2072
	0.0896	1.8355	0.0638	1.7500	0.2272
lcet10.txt	0.3664	1.4637	0.1466	1.7319	0.3062
plrabn12.txt	0.3226	1.5122	0.1419	1.7500	0.2975
ptt5	0.2915	1.5486	0.2364	1.7500	0.3902
sum	0.4165	1.4120	0.0354	1.7060	0.1566
xargs.1	0.6328	1.2249	0.0057	1.6125	0.1301
The Large Corpus					
E.coli	0.3169	1.5188	0.2187	1.7500	0.4056
	0.3140	1.5220	0.2174	1.7500	0.4035
	0.3120	1.5244	0.2197	1.7500	0.4047
	0.3159	1.5199	0.2180	1.7500	0.4048
	0.3195	1.5157	0.2192	1.7500	0.4074
	0.3175	1.5180	0.2140	1.7500	0.4038
bible.txt	0.3679	1.4621	0.1998	1.7310	0.3740
	0.3631	1.4672	0.1925	1.7336	0.3637
	0.3363	1.4967	0.1916	1.7483	0.3611
	0.3589	1.4718	0.1894	1.7359	0.3600
	0.3494	1.4821	0.1711	1.7410	0.3383
world192.txt	0.3868	1.4421	0.1564	1.7211	0.3177
	0.3936	1.4351	0.1528	1.7176	0.3137
	0.4111	1.4173	0.1428	1.7087	0.3010
The Calgary Corpus					
bib	0.4547	1.3749	0.0962	1.6874	0.2496
book1	0.3104	1.5263	0.1504	1.7500	0.3006
book2	0.3853	1.4438	0.1336	1.7219	0.2871
geo	0.2969	1.5422	0.0148	1.7500	0.1554
news	0.4282	1.4004	0.0805	1.7002	0.2109
obj1	0.3840	1.4451	0.0176	1.7225	0.1450
obj2	0.4424	1.3866	0.0340	1.6933	0.1431
paper1	0.5208	1.3151	0.0565	1.6575	0.1981
paper2	0.4351	1.3937	0.0890	1.6968	0.2423
paper3	0.4702	1.3604	0.0629	1.6802	0.2128
paper4	0.5585	1.2833	0.0145	1.6416	0.1628
paper5	0.5710	1.2731	0.0080	1.6366	0.1487
paper6	0.5512	1.2893	0.0437	1.6446	0.1823
pic	0.2915	1.5486	0.2364	1.7500	0.3902
progc	0.5291	1.3080	0.0429	1.6540	0.1770
progl	0.5594	1.2826	0.0804	1.6413	0.2382
progp	0.6433	1.2171	0.0526	1.6085	0.2019
trans	0.6652	1.2011	0.0588	1.6005	0.2042

Table 7: Competitive ratios c^{λ}_{MTF} and c^{λ}_{BIT} according to Corollaries 1 and 2. The values are compared to c^{*}_{MTF} and c^{*}_{BIT}, respectively, yielding relative errors $f_{c^{\lambda}_{MTF}}$ and $f_{c^{\lambda}_{BIT}}$.

Appendix B

file	seq. size	diff. pages	s/r	l/r	l_p/l	l_i/l	l_c/l	λ
bzip2 g7	10481530	353	0.4871	0.5129	0.5017	0.4983	0.5584	0.2864
bzip2 g9	10481504	825	0.4715	0.5285	0.4760	0.5240	0.5819	0.3076
bzip2 p7	10481167	346	0.4790	0.5210	0.4866	0.5134	0.5742	0.2992
bzip2 p9	10481213	826	0.4641	0.5359	0.4628	0.5372	0.5957	0.3193
bzip2 s7	10481540	340	0.4819	0.5181	0.4935	0.5065	0.5675	0.2940
bzip2 s9	10481386	826	0.4663	0.5337	0.4677	0.5323	0.5903	0.3151
crafty	10481629	1007	0.3199	0.6801	0.2525	0.7475	0.7712	0.5245
eon cook	10481801	533	0.4976	0.5024	0.3749	0.6251	0.7495	0.3766
eon kajiya	10482257	397	0.5114	0.4886	0.3721	0.6279	0.7776	0.3800
eon rush	10482256	374	0.5067	0.4933	0.3766	0.6234	0.7678	0.3788
gap	10481513	901	0.4011	0.5989	0.3965	0.6035	0.6417	0.3843
gcc 166	10481586	706	0.3663	0.6337	0.3676	0.6324	0.6749	0.4277
gcc 200	10479248	542	0.3438	0.6562	0.3333	0.6667	0.7002	0.4595
gcc expr	10481541	778	0.3557	0.6443	0.3527	0.6473	0.6871	0.4427
gcc integ	10481491	716	0.3573	0.6427	0.3546	0.6454	0.6857	0.4407
gcc scilab	10481568	809	0.3510	0.6490	0.3357	0.6643	0.7018	0.4555
gzip graphic	10481443	576	0.4377	0.5623	0.4566	0.5434	0.5933	0.3336
gzip log	10481298	429	0.4871	0.5129	0.5680	0.4320	0.4607	0.2363
gzip program	10481317	509	0.4302	0.5698	0.4706	0.5294	0.5681	0.3237
gzip random	10480337	526	0.4320	0.5680	0.4565	0.5435	0.5927	0.3367
gzip source	10481850	482	0.4402	0.5598	0.4920	0.5080	0.5420	0.3034
mcf	10481598	3003	0.2487	0.7513	0.2436	0.7564	0.7713	0.5794
parser	10481694	1353	0.4209	0.5791	0.3811	0.6189	0.6634	0.3841
perl diffmail	10481359	996	0.3354	0.6646	0.3044	0.6956	0.7326	0.4869
perl makerand	10482174	3151	0.3296	0.6704	0.1901	0.8099	0.8142	0.5459
perl perfect	10481539	580	0.3172	0.6828	0.2808	0.7192	0.7553	0.5157
perl splitmail	10481588	702	0.3332	0.6668	0.3015	0.6985	0.7353	0.4903
twolf	10481015	469	0.3467	0.6533	0.3148	0.6852	0.7190	0.4697
vortex one	10481880	1589	0.3629	0.6371	0.2420	0.7580	0.7839	0.4994
vortex two	10481864	1647	0.3603	0.6397	0.2393	0.7607	0.7864	0.5031
vortex three	10481875	1642	0.3585	0.6415	0.2369	0.7631	0.7888	0.5060
vpr place	10481572	782	0.4098	0.5902	0.4591	0.5409	0.5624	0.3319
vpr route	10481581	605	0.4476	0.5524	0.4256	0.5744	0.6254	0.3455
ammp	10481899	251	0.4102	0.5898	0.4050	0.5950	0.6541	0.3858
applu	10482411	1394	0.3156	0.6844	0.2410	0.7590	0.7640	0.5229
apsi	10474135	517	0.3426	0.6574	0.2804	0.7196	0.7448	0.4896
art	10481866	1133	0.3962	0.6038	0.4240	0.5760	0.5997	0.3621
equake	10481760	747	0.5088	0.4912	0.5480	0.4520	0.5346	0.2626
facerec	10481552	527	0.4081	0.5919	0.4130	0.5870	0.6360	0.3765
fma3d	10479601	520	0.4946	0.5054	0.5527	0.4473	0.4821	0.2436
galgel	10481912	516	0.5967	0.4033	0.4931	0.5069	0.5618	0.2266
lucas	10481561	1570	0.4984	0.5016	0.4829	0.5171	0.5266	0.2641
mesa	10482000	1398	0.6044	0.3956	0.5280	0.4720	0.4774	0.1889
mgrid	10482496	1977	0.4200	0.5800	0.3325	0.6675	0.6741	0.3909
sixtrack	10481307	1668	0.4867	0.5133	0.4811	0.5189	0.5300	0.2720
swim	10481645	2098	0.3741	0.6259	0.4067	0.5933	0.6188	0.3872
wupwise	10481968	569	0.3581	0.6419	0.3062	0.6938	0.7229	0.4640

Table 8: Characteristics of the memory access traces. Length of the sequence, number of different pages requested and statistics about the runs.

file	OPT^*	f_{OPT}	MTF^*	f_{MTF}	TS^*	f_{TS}	BIT^*	f_{BIT}	$COMB^*$	f_{COMB}
bzip2 g7	19041305	0.0005	23808453	0	23750319	0.0045	23959189.9375	0.0003	23753803.9375	0.0075
bzip2 g9	19595974	0.0034	24482053	0	24708075	0.0217	24638102.3750	0.0197	24747882.5000	0.0161
bzip2 p7	19119813	0.0004	23800332	0	23948967	0.0044	23914290.3125	0.0056	23881583.7500	0.0070
bzip2 p9	19651460	0.0033	24441294	0	24859257	0.0221	24643607.8125	0.0221	24697333.6875	0.0216
bzip2 s7	19072205	0.0003	23779070	0	23830100	0.0043	23879476.6875	0.0042	23825573.8750	0.0061
bzip2 s9	19625107	0.0034	24447705	0	24800612	0.0218	24724719.9375	0.0175	24791288.1250	0.0163
crafty	27435162	0.0022	32797854	0	40157785	0.0220	36614133.6250	0.0182	36696794.4375	0.0364
eon cook	18340223	0.0014	21921452	0	23311621	0.0099	22278605.8750	0.0074	22444844.0000	0.0097
eon kajiya	17994231	0.0009	21377727	0	22718353	0.0055	21662773.6875	0.0035	21878515.6250	0.0037
eon rush	18055725	0.0008	21477451	0	22794425	0.0049	21731393.9375	0.0053	21956685.3125	0.0047
gap	23116558	0.0033	28791857	0	30930257	0.0211	30213037.6875	0.0164	30213024.0625	0.0222
gcc 166	26454187	0.0016	32890776	0	36572110	0.0112	35253853.1250	0.0100	35417589.0000	0.0131
gcc 200	26383889	0.0010	32290397	0	37023743	0.0066	35163962.5625	0.0054	35338105.2500	0.0113
gcc expr	26633066	0.0022	32902894	0	37085519	0.0131	35523302.3750	0.0113	35643474.0000	0.0171
gcc integ	26721936	0.0017	33055848	0	37202802	0.0112	35646241.1250	0.0102	35795867.1250	0.0150
gcc scilab	26224155	0.0019	32165133	0	36679428	0.0151	34901247.1875	0.0116	34971861.3125	0.0206
gzip graphic	20977689	0.0009	26272194	0	26938644	0.0109	26888501.5000	0.0090	26759993.5000	0.0146
gzip log	16265504	0.0004	19883978	0	19148234	0.0089	19761100.0000	0.0056	19500459.8750	0.0134
gzip program	20323992	0.0005	25396805	0	25800601	0.0092	26019709.0625	0.0061	25824171.1875	0.0127
gzip random	21990881	0.0005	27749992	0	28506825	0.0089	28555206.1875	0.0049	28307865.8125	0.0142
gzip source	19305104	0.0004	24065375	0	24093040	0.0091	24424265.2500	0.0082	24243553.8125	0.0131
mcf	47240992	0.0110	57523649	0	74415158	0.1072	67398608.1250	0.0904	66914378.2500	0.1249
parser	20951524	0.0073	25770547	0	27559490	0.0544	26735276.7500	0.0433	26805792.6250	0.0492
perl diffmail	26608785	0.0019	32269697	0	37719043	0.0236	35334990.3125	0.0203	35657725.0625	0.0254
perl makerand	22763892	0.0117	27474091	0	32428760	0.2896	29487743.8750	0.2409	30123287.3125	0.2495
perl perfect	28207213	0.0006	33898407	0	40845230	0.0073	37799755.3750	0.0050	38288086.5000	0.0086
perl splitmail	27771215	0.0011	33718801	0	39781745	0.0107	37130853.0625	0.0094	37484745.6875	0.0144
twolf	25405630	0.0007	30805751	0	35694923	0.0051	33581876.6250	0.0041	34062627.7500	0.0026
vortex one	27623852	0.0051	33569584	0	40733985	0.0549	36794640.1250	0.0465	37343411.5625	0.0550
vortex two	27735820	0.0050	33692025	0	40946422	0.0593	37016741.3750	0.0486	37500413.4375	0.0594
vortex three	27807823	0.0043	33764481	0	41092298	0.0596	37094391.6250	0.0497	37664598.8750	0.0582
vpr place	23362831	0.0010	29937974	0	30660466	0.0183	30914425.8125	0.0149	30650092.8125	0.0226
vpr route	20484025	0.0009	25410421	0	26525432	0.0123	26063059.5000	0.0091	26000790.5000	0.0158
ammp	21137925	$< 10^{-4}$	25873293	0	27571783	0.0022	26974465.7500	0.0020	27132639.3750	0.0007
applu	17402848	0.0005	19857152	0	22578118	0.0853	21244261.1250	0.0686	21447265.0625	0.0753
apsi	20282849	0.0002	23680360	0	27411563	0.0094	25593928.0000	0.0091	25805319.8125	0.0151
art	19928722	0.0005	24635070	0	25379249	0.0498	25510315.3750	0.0343	25364097.3125	0.0423
equake	16962520	0.0010	20879076	0	20147541	0.0259	20590410.0000	0.0192	20445848.3750	0.0234
facerec	21395553	0.0008	26375120	0	27938951	0.0087	27440575.9375	0.0082	27505873.6875	0.0095
fma3d	17865239	0.0002	22430936	0	21664281	0.0121	22241428.3750	0.0088	22043067.6875	0.0133
galgel	17136944	0.0003	21405539	0	21347139	0.0121	20824361.2500	0.0094	20708296.8750	0.0207
lucas	16872346	0.0089	21045514	0	20465514	0.1053	20702051.8125	0.0794	20606735.4375	0.0870
mesa	18514269	0.0004	24643701	0	23352719	0.0830	23283006.1250	0.0629	23050166.8125	0.0784
mgrid	17736426	0.0002	21757472	0	22386216	0.1741	21980486.9375	0.1321	22129566.2500	0.1371
sixtrack	31058863	0.0046	43390757	0	43099241	0.0577	43264831.9375	0.0443	43256346.0625	0.0464
swim	27568308	0.0042	35908171	0	37284035	0.1117	37237966.3750	0.0918	37451208.6875	0.0898
wupwise	16766967	0.0007	19107193	0	21169868	0.0140	20246223.3750	0.0096	20232872.6250	0.0204

Table 9: Actual service costs incurred by the algorithms on the memory access traces. Comparison of the actual costs to the bounds of Lemmas 1–5 yields the relative errors f_A, for the various strategies A.

file	$\overline{OPT*}$	$\overline{MTF*}$	$\overline{TS*}$	$\overline{BIT*}$	$\overline{COMB*}$
bzip2 g7	1.8167	2.2715	2.2659	2.2858	2.2663
bzip2 g9	1.8696	2.3357	2.3573	2.3506	2.3611
bzip2 p7	1.8242	2.2708	2.2850	2.2816	2.2785
bzip2 p9	1.8749	2.3319	2.3718	2.3512	2.3563
bzip2 s7	1.8196	2.2687	2.2735	2.2782	2.2731
bzip2 s9	1.8724	2.3325	2.3662	2.3589	2.3653
crafty	2.6175	3.1291	3.8313	3.4932	3.5011
eon cook	1.7497	2.0914	2.2240	2.1255	2.1413
eon kajiya	1.7166	2.0394	2.1673	2.0666	2.0872
eon rush	1.7225	2.0489	2.1746	2.0732	2.0947
gap	2.2055	2.7469	2.9509	2.8825	2.8825
gcc 166	2.5239	3.1380	3.4892	3.3634	3.3790
gcc 200	2.5177	3.0814	3.5331	3.3556	3.3722
gcc expr	2.5409	3.1391	3.5382	3.3891	3.4006
gcc integ	2.5494	3.1537	3.5494	3.4009	3.4152
gcc scilab	2.5019	3.0687	3.4994	3.3298	3.3365
gzip graphic	2.0014	2.5065	2.5701	2.5653	2.5531
gzip log	1.5519	1.8971	1.8269	1.8854	1.8605
gzip program	1.9391	2.4231	2.4616	2.4825	2.4638
gzip random	2.0983	2.6478	2.7200	2.7246	2.7010
gzip source	1.8418	2.2959	2.2985	2.3301	2.3129
mcf	4.5070	5.4881	7.0996	6.4302	6.3840
parser	1.9989	2.4586	2.6293	2.5507	2.5574
perl diffmail	2.5387	3.0788	3.5987	3.3712	3.4020
perl makerand	2.1717	2.6210	3.0937	2.8131	2.8738
perl perfect	2.6911	3.2341	3.8969	3.6063	3.6529
perl splitmail	2.6495	3.2170	3.7954	3.5425	3.5762
twolf	2.4240	2.9392	3.4057	3.2041	3.2499
vortex one	2.6354	3.2026	3.8861	3.5103	3.5627
vortex two	2.6461	3.2143	3.9064	3.5315	3.5776
vortex three	2.6529	3.2212	3.9203	3.5389	3.5933
vpr place	2.2289	2.8562	2.9252	2.9494	2.9242
vpr route	1.9543	2.4243	2.5307	2.4866	2.4806
ammp	2.0166	2.4684	2.6304	2.5734	2.5885
applu	1.6602	1.8943	2.1539	2.0267	2.046
apsi	1.9365	2.2608	2.6171	2.4435	2.4637
art	1.9013	2.3503	2.4213	2.4338	2.4198
equake	1.6183	1.9919	1.9222	1.9644	1.9506
facerec	2.0413	2.5163	2.6655	2.6180	2.6242
fma3d	1.7048	2.1404	2.0673	2.1224	2.1034
galgel	1.6349	2.0421	2.0366	1.9867	1.9756
lucas	1.6097	2.0079	1.9525	1.9751	1.9660
mesa	1.7663	2.3510	2.2279	2.2212	2.1990
mgrid	1.6920	2.0756	2.1356	2.0969	2.1111
sixtrack	2.9633	4.1398	4.1120	4.1278	4.1270
swim	2.6302	3.4258	3.5571	3.5527	3.5730
wupwise	1.5996	1.8229	2.0196	1.9315	1.9303

Table 10: Average service cost per request incurred by the algorithms on the memory access traces.

file	c_{MTF}	c^*_{MTF}	$f_{c_{MTF}}$	c_{TS}	c^*_{TS}	$f_{c_{TS}}$	c_{BIT}	c^*_{BIT}	$f_{c_{BIT}}$	c_{COMB}	c^*_{COMB}	$f_{c_{COMB}}$
bzip2 g7	1.2510	1.2504	0.0005	1.2535	1.2473	0.0050	1.2585	1.2583	0.0002	1.2575	1.2475	0.0080
bzip2 g9	1.2544	1.2493	0.0040	1.2935	1.2609	0.0259	1.2869	1.2573	0.0236	1.2882	1.2629	0.0201
bzip2 p7	1.2453	1.2448	0.0004	1.2586	1.2526	0.0048	1.2582	1.2508	0.0060	1.2583	1.2490	0.0074
bzip2 p9	1.2487	1.2437	0.0040	1.2980	1.2650	0.0261	1.2865	1.2540	0.0259	1.2888	1.2568	0.0255
bzip2 s7	1.2473	1.2468	0.0004	1.2553	1.2495	0.0047	1.2578	1.2521	0.0046	1.2573	1.2492	0.0065
bzip2 s9	1.2506	1.2457	0.0039	1.2963	1.2637	0.0258	1.2867	1.2599	0.0213	1.2886	1.2632	0.0201
crafty	1.1982	1.1955	0.0023	1.4994	1.4637	0.0244	1.3620	1.3346	0.0205	1.3895	1.3376	0.0388
eon cook	1.1971	1.1953	0.0016	1.2857	1.2711	0.0115	1.2255	1.2147	0.0089	1.2376	1.2238	0.0112
eon kajiya	1.1893	1.1880	0.0010	1.2708	1.2625	0.0065	1.2092	1.2039	0.0045	1.2215	1.2159	0.0047
eon rush	1.1905	1.1895	0.0008	1.2697	1.2624	0.0058	1.2110	1.2036	0.0062	1.2227	1.2161	0.0055
gap	1.2502	1.2455	0.0037	1.3714	1.3380	0.0250	1.3331	1.3070	0.0200	1.3408	1.3070	0.0259
gcc 166	1.2455	1.2433	0.0018	1.4004	1.3825	0.0130	1.3483	1.3326	0.0118	1.3587	1.3388	0.0149
gcc 200	1.2251	1.2239	0.0010	1.4139	1.4033	0.0076	1.3414	1.3328	0.0064	1.3559	1.3394	0.0123
gcc expr	1.2383	1.2354	0.0023	1.4140	1.3925	0.0155	1.3520	1.3338	0.0136	1.3644	1.3383	0.0195
gcc integ	1.2394	1.2370	0.0019	1.4105	1.3922	0.0131	1.3500	1.3340	0.0120	1.3621	1.3396	0.0168
gcc scilab	1.2290	1.2265	0.0020	1.4226	1.3987	0.0171	1.3490	1.3309	0.0136	1.3637	1.3336	0.0226
gzip graphic	1.2537	1.2524	0.0011	1.2996	1.2842	0.0120	1.2946	1.2818	0.0100	1.2956	1.2756	0.0156
gzip log	1.2230	1.2225	0.0004	1.1883	1.1772	0.0094	1.2222	1.2149	0.0060	1.2154	1.1989	0.0138
gzip program	1.2505	1.2496	0.0007	1.2821	1.2695	0.0100	1.2889	1.2802	0.0068	1.2876	1.2706	0.0133
gzip random	1.2627	1.2619	0.0007	1.3086	1.2963	0.0095	1.3057	1.2985	0.0055	1.3063	1.2873	0.0148
gzip source	1.2473	1.2466	0.0006	1.2601	1.2480	0.0096	1.2762	1.2652	0.0087	1.2729	1.2558	0.0136
mcf	1.2322	1.2177	0.0119	1.7648	1.5752	0.1204	1.5738	1.4267	0.1031	1.6120	1.4164	0.1381
parser	1.2401	1.2300	0.0082	1.3984	1.3154	0.0631	1.3418	1.2761	0.0515	1.3531	1.2794	0.0576
perl diffmail	1.2153	1.2127	0.0021	1.4540	1.4175	0.0257	1.3576	1.3279	0.0223	1.3769	1.3401	0.0275
perl makerand	1.2217	1.2069	0.0122	1.8596	1.4246	0.3054	1.6269	1.2954	0.2560	1.6735	1.3233	0.2646
perl perfect	1.2026	1.2018	0.0007	1.4597	1.4480	0.0080	1.3477	1.3401	0.0057	1.3701	1.3574	0.0093
perl splitmail	1.2158	1.2142	0.0013	1.4498	1.4325	0.0121	1.3513	1.3370	0.0107	1.3710	1.3498	0.0157
twolf	1.2134	1.2126	0.0007	1.4132	1.4050	0.0058	1.3281	1.3218	0.0048	1.3451	1.3408	0.0033
vortex one	1.2218	1.2152	0.0054	1.5640	1.4746	0.0607	1.4013	1.3320	0.0521	1.4339	1.3519	0.0607
vortex two	1.2212	1.2147	0.0053	1.5722	1.4763	0.0650	1.4068	1.3346	0.0541	1.4399	1.3521	0.0650
vortex three	1.2198	1.2142	0.0046	1.5730	1.4777	0.0645	1.4065	1.3340	0.0544	1.4398	1.3545	0.0630
vpr place	1.2829	1.2814	0.0011	1.3378	1.3124	0.0194	1.3444	1.3232	0.0160	1.3431	1.3119	0.0237
vpr route	1.2417	1.2405	0.0009	1.3121	1.2949	0.0133	1.2851	1.2724	0.0100	1.2905	1.2693	0.0167
ammp	1.2241	1.2240	$< 10^{-4}$	1.3073	1.3044	0.0023	1.2788	1.2761	0.0021	1.2845	1.2836	0.0007
applu	1.1419	1.1410	0.0008	1.4092	1.2974	0.0862	1.3054	1.2207	0.0693	1.3261	1.2324	0.0760
apsi	1.1679	1.1675	0.0003	1.3646	1.3515	0.0097	1.2737	1.2619	0.0094	1.2919	1.2723	0.0154
art	1.2369	1.2362	0.0006	1.3377	1.2735	0.0504	1.3247	1.2801	0.0349	1.3273	1.2727	0.0429
equake	1.2323	1.2309	0.0012	1.2200	1.1878	0.0271	1.2386	1.2139	0.0204	1.2349	1.2054	0.0245
facerec	1.2338	1.2327	0.0009	1.3183	1.3058	0.0096	1.2941	1.2825	0.0090	1.2989	1.2856	0.0104
fma3d	1.2559	1.2556	0.0003	1.2277	1.2126	0.0124	1.2563	1.2450	0.0091	1.2506	1.2339	0.0135
galgel	1.2496	1.2491	0.0004	1.2612	1.2457	0.0125	1.2271	1.2152	0.0098	1.2339	1.2084	0.0211
lucas	1.2598	1.2473	0.0100	1.3541	1.2130	0.1163	1.3371	1.2270	0.0897	1.3405	1.2213	0.0976
mesa	1.3318	1.3311	0.0006	1.3668	1.2613	0.0836	1.3373	1.2576	0.0634	1.3432	1.2450	0.0789
mgrid	1.2272	1.2267	0.0004	1.4826	1.2622	0.1746	1.4035	1.2393	0.1325	1.4193	1.2477	0.1375
sixtrack	1.4039	1.3970	0.0049	1.4750	1.3877	0.0629	1.4617	1.3930	0.0493	1.4644	1.3927	0.0514
swim	1.3084	1.3025	0.0045	1.5102	1.3524	0.1167	1.4813	1.3508	0.0966	1.4871	1.3585	0.0947
wupwise	1.1406	1.1396	0.0009	1.2814	1.2626	0.0149	1.2200	1.2075	0.0104	1.2323	1.2067	0.0212

Table 11: Upper bounds c_A on the performance ratios achieved by $A \in \{MTF, TS, BIT, COMB\}$ on the memory access traces, as implied by Theorems 1–4. The values are compared to the experimentally observed competitiveness c^*_A, which is the actual cost incurred by A to that of the pairwise optimum. The comparison yields a relative error f_{c_A}.

file	λ	c^{λ}_{MTF}	$f_{c^{\lambda}_{MTF}}$	c^{λ}_{BIT}	$f_{c^{\lambda}_{BIT}}$
bzip2 g7	0.2864	1.5547	0.2434	1.7500	0.3908
bzip2 g9	0.3076	1.5296	0.2243	1.7500	0.3919
bzip2 p7	0.2992	1.5394	0.2367	1.7500	0.3991
bzip2 p9	0.3193	1.5160	0.2189	1.7500	0.3955
bzip2 s7	0.2940	1.5456	0.2396	1.7500	0.3977
bzip2 s9	0.3151	1.5208	0.2208	1.7500	0.3891
crafty	0.5245	1.3119	0.0974	1.6560	0.2408
eon cook	0.3766	1.4529	0.2155	1.7264	0.4212
eon kajiya	0.3800	1.4493	0.2199	1.7247	0.4326
eon rush	0.3788	1.4506	0.2195	1.7253	0.4335
gap	0.3843	1.4448	0.1600	1.7224	0.3178
gcc 166	0.4277	1.4008	0.1267	1.7004	0.2760
gcc 200	0.4595	1.3703	0.1197	1.6852	0.2644
gcc expr	0.4427	1.3863	0.1221	1.6932	0.2694
gcc integ	0.4407	1.3882	0.1222	1.6941	0.2700
gcc scilab	0.4555	1.3741	0.1203	1.6871	0.2676
gzip graphic	0.3336	1.4997	0.1975	1.7498	0.3652
gzip log	0.2363	1.6177	0.3233	1.7500	0.4404
gzip program	0.3237	1.5109	0.2091	1.7500	0.3669
gzip random	0.3367	1.4963	0.1857	1.7481	0.3463
gzip source	0.3034	1.5344	0.2309	1.7500	0.3832
mcf	0.5794	1.2663	0.0399	1.6331	0.1447
parser	0.3841	1.4449	0.1747	1.7225	0.3498
perl diffmail	0.4869	1.3450	0.1091	1.6725	0.2595
perl makerand	0.5459	1.2938	0.0720	1.6469	0.2714
perl perfect	0.5157	1.3195	0.0980	1.6598	0.2386
perl splitmail	0.4903	1.3420	0.1053	1.6710	0.2498
twolf	0.4697	1.3608	0.1223	1.6804	0.2713
vortex one	0.4994	1.3339	0.0976	1.6669	0.2515
vortex two	0.5031	1.3306	0.0954	1.6653	0.2478
vortex three	0.5060	1.3280	0.0937	1.6640	0.2474
vpr place	0.3319	1.5016	0.1718	1.7500	0.3225
vpr route	0.3455	1.4865	0.1983	1.7432	0.3701
ammp	0.3858	1.4432	0.1790	1.7216	0.3491
applu	0.5229	1.3133	0.1510	1.6567	0.3571
apsi	0.4896	1.3426	0.1500	1.6713	0.3245
art	0.3621	1.4683	0.1878	1.7341	0.3547
equake	0.2626	1.5840	0.2869	1.7500	0.4417
facerec	0.3765	1.4530	0.1787	1.7265	0.3462
fma3d	0.2436	1.6082	0.2809	1.7500	0.4057
galgel	0.2266	1.6306	0.3054	1.7500	0.4401
lucas	0.2641	1.5821	0.2684	1.7500	0.4263
mesa	0.1889	1.6823	0.2639	1.7500	0.3916
mgrid	0.3909	1.4379	0.1721	1.7189	0.3870
sixtrack	0.2720	1.5723	0.1254	1.7500	0.2563
swim	0.3872	1.4417	0.1069	1.7209	0.2740
wupwise	0.4640	1.3661	0.1988	1.6830	0.3938

Table 12: Competitive ratios c^{λ}_{MTF} and c^{λ}_{BIT} for the memory access traces, according to Corollaries 1 and 2. The values are compared to c^{*}_{MTF} and c^{*}_{BIT}, respectively, yielding relative errors $f_{c^{\lambda}_{MTF}}$ and $f_{c^{\lambda}_{BIT}}$.