Trace Reduction for LRU-based Simulations

Yannis Smaragdakis
University of Texas at Austin

Abstract

LRU buffer management is a policy under which an element buffer of size k always stores the k most recently used elements. Many variants of the policy are widely used in memory systems (e.g., virtual memory subsystems). We study a simple algorithmic problem (called the *trace reduction* problem): given a sequence of references to elements (a *reference trace*), compute the shortest sequence with identical LRU behavior for buffers of at least a certain size. Despite the straightforward statement of the problem, its solution is non-trivial. We offer an algorithm and discuss its practical applications. These are mostly in the area of trace-driven program simulation: for quite realistic buffer sizes, reference traces from real programs can be reduced by a factor of up to several orders of magnitude. This compares favorably with all previous trace reduction techniques in the literature.

1 Introduction

An *LRU stack* of size k is a data structure storing the k most recently accessed elements of a larger set. References to set elements may cause the subset stored in an LRU stack to change. A full record of these changes is said to constitute the *(LRU) behavior* of the stack under the sequence of references. We will discuss the *LRU trace reduction problem*: given a sequence of references, find the shortest sequence with identical behavior for an LRU stack of (at least) a given size.

As an example, consider a set with four elements (**a**, **b**, **c**, and **d**) and the reference sequence:

$$\mathbf{a\ b\ c\ b\ a\ c\ d\ a\ b\ d} \tag{1}$$

For an LRU buffer of size $k = 3$, the behavior of the sequence is:

$$\langle \mathbf{a}, NF\rangle, \langle \mathbf{b}, NF\rangle, \langle \mathbf{c}, NF\rangle, \langle \mathbf{d}, \mathbf{b}\rangle, \langle \mathbf{b}, \mathbf{c}\rangle, \textsc{Last} \tag{2}$$

The above behavior consists of pairs of elements entering and leaving the LRU stack. The special value Last signals the end of the sequence, while NF denotes that the LRU stack is not full and, hence, the insertion of one element does not cause the eviction of another. It is easy to confirm that the sequence:

$$\mathbf{a\ b\ c\ a\ d\ b} \tag{3}$$

has identical LRU behavior to (1) for a stack of size $k = 3$ (or greater). In fact, something more can be said: there is no sequence shorter than (3) with the same LRU behavior. Hence, sequence (3) is a solution to the LRU trace reduction problem for input (1) and a stack size of 3.

In this paper we will describe a general algorithm for computing such solutions. The algorithm has already found application in the area of *trace-driven simulation* of replacement policies for *storage hierarchies.* Storage hierarchies are used in various parts of modern computer systems (CPU, disk, and network caches, for instance). In such hierarchies, one level acts as a fast buffer that stores elements from the next level of storage. When the buffer is full, a decision needs to be made about the buffer element that will get replaced. The *Least Recently Used (LRU)* policy (to which the "LRU stack" data type owes its name) is the most extensively studied replacement policy for storage hierarchies. Under LRU the element replaced is the one used least recently (i.e., furthest in the past) — the buffer is essentially an LRU stack. Replacement policies can be studied cost-effectively using trace-driven simulation: reference traces from actual programs are collected and re-used in simulations of various policies. The size of such traces is usually enormous. As a result, traces are often hard to store and process (simulation time is commonly proportional to the length of the trace). Using our algorithm to solve the LRU trace reduction problem, traces can be drastically reduced in size and the results of the simulation are entirely accurate, provided the simulated LRU stack is of at least a certain size. What makes this application of even greater interest are the many variations of LRU that have been studied (e.g.,[FeLW78, GlCa97]). These policies handle a small part of the buffer using LRU and the rest using a different policy. Under the guarantees offered by our algorithm, simulation with reduced traces is exact for all such policies (provided that the LRU part of the buffer is larger than the reduction stack size of k elements).

The results of trace reduction using our algorithm are significant: for quite realistic buffer sizes, reference traces can be reduced by several orders of magnitude. All previous techniques that achieve similar compression results have one of three drawbacks: they are either inaccurate (i.e., introduce error by dropping information from the trace), or require a decompression phase (and, hence, do not speed up simulation which is performed on the decompressed result), or require modifications to the simulator to accept information in a different format (i.e., the result of the reduction is not itself a reference trace).

2 LRU Trace Reduction

2.1 Background

In this section we will define more formally the LRU trace reduction problem as well as some helpful concepts.

An *LRU stack* of size k is a data type defining a single operation $touch_k(m)$, where m identifies a "block" (we assume no properties for blocks other than identity). The semantics of the operation is defined as follows:

- If there have been less than k distinct blocks "touched" in the past, $touch_k(m)$ returns the special value NF.
- If m is among the k most recently "touched" blocks, $touch_k(m)$ returns the special value *none*.
- Otherwise $touch_k(m)$ returns an identifier of the $k+1$-th most recently "touched" block.

A reference trace is a finite sequence of references to blocks. We say that a reference trace r is *applied* to an LRU stack by performing the operation $touch_k$ for each block of the trace in increasing order (i.e., $touch_k(r(0)), touch_k(r(1)), \ldots, touch_k(r(\#r-1))$ [1]). We define the *complete (LRU) trace* of a reference trace r to be a sequence $c_{k,r}$ with $c_{k,r}(i) = \langle r(i), touch_k(r(i))\rangle$, $0 \leq i < \#r$. The complete trace is terminated by a special element LAST (that is, $c_{k,r}(\#r) = \text{LAST}$). We also define the *relevant event sequence* $re_{k,r}$ to consist of all indices i (in increasing order) such that $touch_k(r(i)) \neq none$. In the following, we will drop subscripts when they can be deduced from the context (both for the above definitions and for ones still to be introduced). Thus we may often refer to $touch_k$, $c_{k,r}$ and $re_{k,r}$ as $touch$, c and re, for simplicity.

We define the *behavior* of a reference trace r relative to an LRU stack to be the subsequence b of the complete sequence c, consisting only of relevant events. That is, $b(i) = c(re(i)) = \langle r(re(i)), touch(re(i))\rangle$, for $0 \leq i < \#re$. By analogy to the complete trace, $b(\#re) = \text{LAST}$. Thus, the behavior of a reference trace contains all references made to blocks that were not among the k most recently touched, and the results of the respective touch operations. The first component $r(re(i))$ of a pair $\langle r(re(i)), e(re(i))\rangle$ will be called the *fetch part* (since it corresponds to a block "fetched" in the LRU stack) and the second component $e(re(i))$ will be called the *evict part* (since it corresponds to a block "evicted" from the stack). We will write $b(i).fetch$ and $b(i).evict$ for the fetch and evict parts of the i-th element of a sequence b.

Data structures conforming to the LRU stack specification are usually studied in the context of storage hierarchies, where they correspond to the Least Recently Used replacement policy. This policy is known to belong in the general class of *stack algorithms* [MGST70]. In fact, LRU belongs to the subset of stack

[1]With $\#r$ we denote the length of a sequence r.

algorithms defined in [MGST70] that base their replacement decisions on a priority list for elements. A property of these algorithms is that if two reference traces have identical behavior for a buffer of size k, they will also have identical behavior for all buffers larger than k. We can now define the problem we are trying to solve:

The LRU Trace Reduction Problem: Given a reference trace, compute a trace such that it has the same behavior for an LRU stack of size k and no shorter trace has that behavior.

Due to the aforementioned property of LRU stacks, the resulting trace will also have identical behavior for larger buffers. A trace satisfying the conditions of the LRU trace reduction problem will be called a *reduced* trace. In general, there will be more than one reduced trace, but all of them will be of the same length.

2.2 Developing a Lower Bound

Before we present an algorithm for the LRU trace reduction problem, it is useful to examine some of the characteristics of the reduced trace.

A simple observation regarding the reduced reference trace is that it depends only on the behavior of the original trace: two input traces with the same behavior will have the same set of reduced traces. Hence, our problem is to compute a reduced trace from a given behavior sequence (derived from the input trace through LRU simulation). Consider the sequence formed by taking the "fetch" parts (i.e., first elements of pairs) of the elements in a behavior sequence. Clearly, this needs to be a subsequence of any reference trace having this behavior. The LRU trace reduction problem is equivalent to adding the smallest possible set of elements to the behavior sequence, so that it becomes a complete LRU trace. We will attempt to develop a lower bound for the contents of this set. Later we will show that this bound is tight: our algorithm produces a trace containing exactly the extra references specified by the lower bound.

Consider a sequence of pairs (of the form found in a complete LRU trace or a behavior sequence). Any consecutive subsequence of a sequence *seq* is called an *interval* of *seq*. Two concepts that are important for our later development are those of a "run" and a "tight run":

Definition 1 (run) *A* fetch-evict run *(or just* run*) in a sequence of pairs of the form $\langle r(i), \mathit{touch}(r(i))\rangle$ is an interval, such that:*

- *it begins with a pair $\langle r(s), \mathit{touch}(r(s))\rangle$ and ends either with a pair $\langle r(f), r(s)\rangle$ or with* LAST
- *no other element in the run has block $r(s)$ in its fetch part (i.e., for every i, $s < i < f$ it is $r(i) \neq r(s)$).*

Intuitively, a run describes the behavior of an LRU stack between the point where an element is last referenced before it is evicted, and the corresponding eviction point. We will describe intervals (and, hence, runs) using the starting and finishing indices in the sequence where they occur. A run (s_0, f_0) (that is, a run starting at index s_0 and finishing at index f_0) will be said to *contain* another run (s_1, f_1) iff $s_0 < s_1$ and $f_1 < f_0$ (note that this definition prescribes that runs extending till the end of a sequence do not contain one another).

Definition 2 (tight run) *A* tight *fetch-evict run (or just* tight run*) is a run that contains no other runs.*

For illustration purposes, consider the example reference trace presented in the Introduction. Its complete LRU trace appears below (indexed for easier reference):

$$\begin{array}{ccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \langle \mathbf{a}, NF\rangle & \langle \mathbf{b}, NF\rangle & \langle \mathbf{c}, NF\rangle & \langle \mathbf{b}, \mathit{none}\rangle & \langle \mathbf{a}, \mathit{none}\rangle & \langle \mathbf{c}, \mathit{none}\rangle & \langle \mathbf{d}, \mathbf{b}\rangle & \langle \mathbf{a}, \mathit{none}\rangle & \langle \mathbf{b}, \mathbf{c}\rangle & \langle \mathbf{d}, \mathit{none}\rangle & \textsc{Last} \end{array}$$

There are five runs in this trace: $(3, 6)$, $(5, 8)$, $(7, 10)$, $(8, 10)$, and $(9, 10)$. All of them are tight. According to the following lemma, this is not a coincidence.

Lemma 1 *All runs in a complete LRU trace are tight.*

Proof: Assume that a run (s_0, f_0) in c is not tight. Then there exists a run (s_1, f_1) with $s_0 < s_1$ and $f_1 < f_0$. The last inequality means that $c(f_1) \neq \textsc{Last}$ (because there is a later element in the trace). Hence, the result of the operation $\mathit{touch}(r(f_1))$ must have been $r(s_1)$ (by definition of a run). By definition of *touch*, this is the $k+1$-th most recently touched element. This is, however, impossible since $r(s_0)$ is less recently last touched than $r(s_1)$ (no reference to $r(s_0)$ can exist between indices s_0 and f_1 by definition of a run) and it is still among the k most recently touched elements (since it has not been evicted from the stack by point f_1). Hence, every run in c is tight. $\square$

Continuing our example, consider the behavior sequence (2) from the Introduction (reproduced below with indices):

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ \langle \mathbf{a}, NF \rangle & \langle \mathbf{b}, NF \rangle & \langle \mathbf{c}, NF \rangle & \langle \mathbf{d}, \mathbf{b} \rangle & \langle \mathbf{b}, \mathbf{c} \rangle & \textsc{Last} \end{array} \tag{4}$$

Not all runs in this sequence are tight: for instance, run $(0, 5)$ contains both runs $(1, 3)$ and $(2, 4)$.

To derive a complete LRU trace from a behavior sequence, we need to add "extra" references to ensure that all runs are tight. The following lemma is straightforward:

Lemma 2 *If a run (s_0, f_0) of a behavior sequence b contains another run (s_1, f_1), then the reduced trace r contains a reference to $b(s_0).\mathit{fetch}$ between $re_{k,r}^{-1}(s_1)$ and $re_{k,r}^{-1}(f_1)$. (Recall that $re_{k,r}$ was defined to be monotonically increasing, hence its inverse $re_{k,r}^{-1}$ always exists.)*

Proof: Consider the complete LRU trace c produced from r. From Lemma 1, we have that all runs in c are tight. Since b is a subsequence of c, either interval $(re^{-1}(s_0), re^{-1}(f_0))$ or interval $(re^{-1}(s_1), re^{-1}(f_1))$ is not a run in c. Quick inspection of all possible cases for reference addition shows that the only way the tightness constraint can be preserved is if there exists i, such that $c(i) = b(s_0).\mathit{fetch} = c(re^{-1}(s_0)).\mathit{fetch}$, $re^{-1}(s_1) < i < re^{-1}(f_1)$. $\square$

In the example of sequence (4), run $(0, 5)$ contains run $(1, 3)$ and run $(2, 4)$. Lemma 2 implies that there needs to be a reference to $b(0).\mathit{fetch} = \mathbf{a}$ between indices 1 and 3, as well as between 2 and 4. A reference to $\mathbf{a}$ after position 2 satisfies both requirements, as seen in sequence (3). The corresponding complete LRU trace is:

$$\begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \langle \mathbf{a}, NF \rangle & \langle \mathbf{b}, NF \rangle & \langle \mathbf{c}, NF \rangle & \langle \mathbf{a}, \mathit{none} \rangle & \langle \mathbf{d}, \mathbf{b} \rangle & \langle \mathbf{b}, \mathbf{c} \rangle & \textsc{Last} \end{array} \tag{5}$$

In the general case, we are looking for the smallest number of extra references that if added to a behavior sequence will turn it into a complete LRU trace. Lemma 2 can be applied to all pairs of runs contained within one another to give a set of constraints for the extra references. In this way, we obtain for every block a set of constraints represented as intervals. The meaning of every constraint is that a reference to the block must appear within the given interval. (In the following, we will not distinguish between a constraint and the interval representing it.) As we saw in our example, several constraints can be satisfied by adding a single reference. We can recast the problem in a more general form: given intervals $(s_0, f_0), \ldots, (s_k, f_k)$, compute a minimal set of points S, such that for each (s_i, f_i), there exists $p \in S$ with $s_i < p < f_i$. It is not hard to show that the problem is almost identical to the *activity-selection problem* (e.g., [CoLR90], p.330): given *activities* represented as intervals, compute the maximum-size set of non-overlapping activities.

The extra requirement in our case is that we also want to compute the intersection of intervals that a certain point in the above minimal set will satisfy. Because of this requirement, a slightly different algorithm than that presented in [CoLR90], p.330, is more suitable. Clearly, a reference satisfies constraints $(s_0, f_0), \ldots, (s_k, f_k)$ iff it belongs to the interval $(\max\{s_0, \ldots, s_k\}, \min\{f_0, \ldots, f_k\})$. The interval is empty if $\max\{s_0, \ldots, s_k\} \geq \min\{f_0, \ldots, f_k\}$. Examining the constraints in increasing starting index order (so that $\max\{s_0, \ldots, s_k\} = s_k$) leads to the following lemma:

Lemma 3 *Consider all constraints for a single block (assuming at least one exists), represented as intervals* $(s_0, f_0), \ldots, (s_n, f_n)$ *with* $s_i < s_j$ *for* $i < j$*.*

1. *The least number of extra references required to satisfy all such constraints is equal to* $\#e - 1$*, where* e *is the sequence with* $e(0) = 0$ *and* $e(i+1)$ *is the least index, such that* $s_{e(i+1)} \geq \min\{f_{e(i)}, \ldots, f_{e(i+1)}\}$ *or, if none exists,* $e(i+1) = n+1$ *and* $e(i+1)$ *is the last element of the sequence* e*.*

2. *The corresponding intervals (*$\#e - 1$ *in total) where the extra references must lie are* $(s_{e(i+1)-1}, min\{f_{e(i)}, \ldots, f_{e(i+1)-1}\})$*, for* $0 \leq i < \#e - 1$*. For a sequence seq and a block identifier bl we call the set of these intervals,* $int_{seq,bl}$*.*

Proof by simple induction (omitted). Very similar in spirit to Theorem 17.1 in [CoLR90], p.332.

We will also use the name *conjunctive constraints* for the elements of $int_{seq,bl}$ (again, we may drop any subscripts whose value is clear from the context).

Lemma 3 essentially describes an algorithm for computing a minimal set of intervals where references satisfying all constraints must lie (the same algorithm can be applied to the activity-selection problem). The algorithm gives a lower bound for the set of extra references required to produce a reduced trace. Notice that it is not clear that there is always a trace with no more extra references than these specified by the lower bound. Also, the procedure does not determine the exact place in an interval where extra references need to be inserted, or the relative positions of extra references. The algorithm described in the next section ensures that the lower bound is tight and provides a very efficient way of computing a reduced trace (in essence, without having to recognize long runs, which would be computationally costly).

2.3 The Algorithm

We will present an algorithm to solve the LRU trace reduction problem. The algorithm takes a behavior sequence as input (derived from a reference trace through simple LRU simulation) and outputs a reduced trace. Additionally, it uses a data structure *queue*, which is an LRU stack augmented by two operations:

- *blocks_after(block)*: returns the set of blocks, touched less recently than *block*, but still in the data structure (i.e., within the last k distinct blocks touched). If *block* has the special value NF, the returned set is empty (this is useful for uniform treatment of the boundary case where the structure is being filled up).

- *more_recent*$(block_1, block_2)$: returns a boolean value indicating whether $block_1$ was touched more recently than $block_2$. If $block_1$ has the special value LowerLimit, or $block_2$ has the special value NF, False is returned.

We use the form *queue.<operation>* to designate the invocation of an operation on *queue*. The input sequence, *behavior*, is represented as an array for simplicity. The full algorithm (called OLR for *optimal LRU reduction*) appears below:

```
OLR(behavior, k)
 1  $lookahead \leftarrow 0, current \leftarrow 0, fetched\_in\_future \leftarrow \emptyset, previous\_evict \leftarrow$ LowerLimit
 2  $queue \leftarrow \emptyset(k)$
 3  ▷ $\emptyset(k)$ denotes an empty LRU stack of size $k$
 4  while $behavior[current] \neq$ Last
 5      do $must\_touch \leftarrow queue$.BLOCKS_AFTER($behavior[current].evict$)
 6         $lookahead\_done \leftarrow$ False
 7         while $behavior[lookahead] \neq$ Last and $\neg lookahead\_done$
 8             do if $behavior[lookahead].evict \in fetched\_in\_future$
 9                   then $lookahead\_done \leftarrow$ True
10                   else if $queue$.MORE_RECENT($previous\_evict$, $behavior[lookahead].evict$)
11                          then PRODUCE_REFERENCE($behavior[lookahead].evict$)
12                               $must\_touch \leftarrow must\_touch \setminus \{behavior[lookahead].evict\}$
13                        $previous\_evict \leftarrow behavior[lookahead].evict$
14                        $fetched\_in\_future \leftarrow fetched\_in\_future \cup \{behavior[lookahead].fetch\}$
15                        $lookahead \leftarrow lookahead + 1$
16         for $x \in must\_touch$
17             do PRODUCE_REFERENCE($x$)
18         PRODUCE_REFERENCE($behavior[current].fetch$)
19         $fetched\_in\_future \leftarrow fetched\_in\_future \setminus \{behavior[current].fetch\}$
20         $current \leftarrow current + 1$
```

```
PRODUCE_REFERENCE(block)
 1  $queue$.TOUCH($block$)
 2  OUTPUT($block$)
```

OLR essentially outputs the fetch parts of pairs in the behavior sequence, interspersed with references to elements in the stack. It works by advancing two pointers (*current*, *lookahead*) through the input (behavior sequence). After each iteration of the loop in lines 7-15, the *lookahead* pointer has advanced to the end of the first tight run beginning at or after position *current*. With each iteration of the loop in lines 4-20, the *current* pointer advances to the next pair in the behavior sequence. The output (reduced) trace is created through successive calls to function PRODUCE_REFERENCE (lines 11, 17, and 18). Before a reference to a block is output, it gets applied to the *queue* data structure. This ensures that *queue* reflects the state of an LRU stack of size k, to which the output trace is being applied.

During the execution of the algorithm, two sets of blocks are maintained: $fetched\_in\_future$ contains the set $\{behavior[i].fetch : current \leq i < lookahead\}$ (this property holds everywhere other than between lines 14 and 15). The *must_touch* set is initialized to contain the blocks touched less recently than $behavior[current].evict$. Finally, *previous_evict* is introduced for convenience: it holds the same block as $behavior[lookahead - 1].evict$, except in the first iteration when it holds the value LowerLimit.

An important observation concerns the **if** statement in lines 10-12 of the algorithm:

Observation 1 After the execution of lines 10-12 the following property holds: if $current \leq i < j \leq lookahead$, then $behavior[j].evict$ was touched more recently than $behavior[i].evict$ (in the current state of *queue*). That is, lines 10-11 make sure (by adding references to blocks) that eviction order matches touch order by increasing recency (up to the *lookahead* point). Line 12 removes any touched elements from the *must_touch* set (if the element was in the set).

It is illustrative to follow the execution of the algorithm's outer loop (lines 4-20) in a simple example (for a stack of size $k = 7$). Consider the following state of execution (the contents of *queue* appear on the left, ordered by recency of touches, with the rightmost element being the most recently touched).

$$\begin{array}{rcc} \textit{queue} & = & \mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d},\mathbf{e},\mathbf{f},\mathbf{g} \\ \textit{fetched\_in\_future} & = & \{\mathbf{h},\mathbf{i}\} \\ \textit{must\_touch} & = & \{\mathbf{a}\} \end{array} \qquad \ldots\ \underset{\widehat{\textit{current}}}{\langle \mathbf{h},\mathbf{b}\rangle}\ \ \langle \mathbf{i},\mathbf{c}\rangle\ \ \underset{\widehat{\textit{lookahead}}}{\langle \mathbf{j},\mathbf{f}\rangle}\ \ \langle \mathbf{k},\mathbf{d}\rangle\ \ \langle \mathbf{l},\mathbf{g}\rangle\ \ \langle \mathbf{b},\mathbf{i}\rangle\ \ldots$$

The state shown is reached after the execution of line 5 (computation of the *must_touch* set). This set represents the blocks in the stack that need to be referenced before *behavior*[*current*].*fetch*. The loop in lines 7-15 will iterate till *lookahead* reaches the end of the first tight run after position *current* (that is, till *lookahead* points to the $\langle \mathbf{b},\mathbf{i}\rangle$ element). For each element that *lookahead* visits (namely $\langle \mathbf{j},\mathbf{f}\rangle$, $\langle \mathbf{k},\mathbf{d}\rangle$, $\langle \mathbf{l},\mathbf{g}\rangle$, $\langle \mathbf{b},\mathbf{i}\rangle$) the algorithm examines the blocks identified by their evict parts. Extra references are added to out-of-order blocks, so that Observation 1 holds. Note that once a block is referenced, it becomes the most recently touched block. Hence, the check of line 10 will always succeed till the end of the next tight run, causing all other blocks examined in the course of the inner loop (lines 7-15) to also be referenced. In our example, the output would be **d**, **g**.

After the end of the inner loop, lines 16-17 will add references to all remaining blocks in the *must_touch* set. Finally, a reference for the fetch part of the current behavior element is output. This completes the output sequence for this iteration of the outer block. The complete output and final state after the execution of line 20 are:

$$\begin{array}{rcc} \textit{queue} & = & \mathbf{c},\mathbf{e},\mathbf{f},\mathbf{d},\mathbf{g},\mathbf{a},\mathbf{h} \\ \textit{fetched\_in\_future} & = & \{\mathbf{i},\mathbf{j},\mathbf{k},\mathbf{l},\mathbf{b}\} \\ \textit{output} & = & \ldots\mathbf{d},\mathbf{g},\mathbf{a},\mathbf{h}\ldots \end{array} \qquad \ldots\ \langle \mathbf{h},\mathbf{b}\rangle\ \ \underset{\widehat{\textit{current}}}{\langle \mathbf{i},\mathbf{c}\rangle}\ \ \langle \mathbf{j},\mathbf{f}\rangle\ \ \langle \mathbf{k},\mathbf{d}\rangle\ \ \langle \mathbf{l},\mathbf{g}\rangle\ \ \underset{\widehat{\textit{lookahead}}}{\langle \mathbf{b},\mathbf{i}\rangle}\ \ldots$$

Note that the *queue* structure is not fully ordered according to eviction order (for instance, there is an **e** between the **c** and **f**). The need to add just enough references to preserve *some* order in the structure is what makes the LRU trace reduction problem challenging. Related problems (e.g., minimum reference trace to convert an LRU stack from one configuration to another) have trivial solutions (which cannot be translated into a solution for the reduction problem).

2.4 Algorithm Correctness

To argue that the algorithm is correct, we must show that the behavior of the produced trace is identical to the algorithm's input and that no shorter trace can have that behavior. Due to its length, the proof can be found in the Appendix. Nevertheless, we will try to give some intuition on the proof development here. Proving that the behavior of the output is identical to the input is straightforward. We will concentrate on the more interesting task of proving that the produced trace is minimal:

- Extra references produced by lines 11 and 17 of the algorithm correspond to non-tight runs. Additionally, in both cases the references satisfy the requirements of Lemma 3: the next constraint on the block referenced has $s_{e(i+1)} \geq \min\{f_{e(i)}, \ldots, f_{e(i+1)}\}$ (and thus cannot be satisfied by the same reference as the previous constraints). This guarantees that the extra references are as few as possible.

- The previous steps are meaningless if we cannot establish a correspondence between non-tight runs in the behavior sequence and in the produced trace. In particular, the difficulty is that the algorithm adds references to a behavior trace to produce the reduced trace. These references can introduce new runs, and these runs entail more constraints (in the sense of Lemma 3). Fortunately, we can prove that the new constraints do not affect the computation of conjunctive constraints (i.e., do not increase the lower bound).

Proving the latter point is quite interesting. Part of the value of the OLR algorithm lies in its efficiency: the algorithm only needs to examine at most k references at any given time. Thus, it does not explicitly detect the endpoints of long non-tight runs. This is, for instance, reflected in lines 16-17, where new references are output in any relative order (i.e., without regard to the subsequent eviction order of the blocks). As it turns out, this order does not matter: extra references to the same blocks need to be inserted in the future before the blocks are actually evicted. Lines 10-11 ensure that the order of final references is such that no new non-tight runs are introduced. Note that this freedom in the relative ordering of extra references by line 17 results into multiple traces that are all solutions to the trace reduction problem[2].

2.5 Time and Space Requirements

We will first examine the running time of OLR. Its asymptotic complexity depends on the complexity of operations on our augmented LRU stack (*queue* data structure). Assume that $h(k)$ is the complexity of operations *touch* and *more_recent*, for a stack of k elements. It is then reasonable to assume that the complexity of operation *blocks_after* is $O(h(k) + n)$, if n is the size of the returned set. For most practical applications, $h(n)$ will be the function $\log n$ and the *queue* structure will be implemented as an augmented self-balancing tree. Another factor is the cost of maintaining simple sets ($fetched\_in\_future$ and *must_touch*) under additions of a single element and deletions. It is quite reasonable to assume that this cost is also bounded by $h(n)$.

Under the above assumptions, the running time of OLR is $O(m \cdot h(k) + n)$, where m is the size of the input (behavior sequence) and n the size of the output (the inequality $m \leq n \leq m \cdot k$ holds). This is not hard to see: Loop 7-15 of the algorithm will be executed exactly m times and the complexity of every operation is at most $h(n)$, for a total running time of $O(m \cdot h(n))$. The outer loop (lines 4-20) will also be executed m times and the total cost of the m $blocks\_after$ operations is $O(m \cdot h(k) + n)$ (the total size of sets returned cannot be more than the length of the output since a reference is produced for each element). Similarly, the running time of the loop in lines 16-17 is $O(n)$ and the cost of lines 18 and 19 is $O(m \cdot h(n))$. Thus the total running time of OLR is $O(m \cdot h(k) + n)$. Note that the input of OLR (behavior sequence) is produced by applying a reference trace to an LRU stack. If we assume $h(k)$ to also be the cost of maintaining a simple LRU stack of k elements (i.e., one with only the operation *touch*), then for a reference trace of length l, producing the behavior sequence has a runtime cost of $O(l \cdot h(k))$. Thus, producing the behavior sequence is generally much more costly than applying OLR.

Another interesting aspect of OLR is that the size of its working storage is bounded by k (i.e., all structures used, including the portion of the input being examined, have at most k elements). This makes the algorithm ideal for online implementations (i.e., reduction can take place while a trace is being produced).

3 Applications

We implemented the reduction algorithm and applied it to actual memory reference traces. The reduced traces were then used in experiments with adaptive replacement policies based on LRU and proved quite valuable. These results provide a good preliminary indication of the applicability of the algorithm.

Traces in our experiment came from six actual programs [3] ("espresso", "gcc", "grobner", "ghostscript", "lindsay", and "p2c"). The traces were collected using the VMTrace utility [Kapl98]. The original traces were already in a "blocked" form (i.e., contained references to pages and not to individual addresses, and contiguous references to the same page were replaced by a single reference). The sizes of the blocked traces appear on the table below:

Trace name	*Original size (KBytes)*	*Trace name*	*Original size (KBytes)*
`espresso`	2,288,568	`gcc-2.7.2`	262,670
`grobner`	54,514	`gs3.33`	940,603
`lindsay`	865,835	`p2c`	215,057

[2]There are, however, solutions that cannot be produced by OLR (due to its bounded lookahead).

[3]These traces are the ones to which we had immediate access for our initial experiments. Many more traces have been reduced since, but the above results are highly typical.