# The EELRU Adaptive Replacement Algorithm

Yannis Smaragdakis
Georgia Institute of Technology
yannis@cc.gatech.edu

Scott Kaplan
Amherst College
sfkaplan@cs.amherst.edu

Paul Wilson
University of Texas at Austin
wilson@cs.utexas.edu

**Abstract**

The wide performance gap between processors and disks ensures that effective page replacement remains an important consideration in modern systems. This article presents *Early Eviction LRU (EELRU)*, an adaptive replacement algorithm. EELRU uses *aggregate* recency information to recognize the locality behavior of a workload and to adjust its speed of adaptation. An on-line cost/benefit analysis guides replacement decisions. This analysis is based on the *LRU stack model* (LRUSM) of program behavior. Essentially, EELRU is an on-line approximation of an optimal algorithm for the LRU stack model. We prove that EELRU offers strong theoretical guarantees of performance relative to the LRU replacement algorithm. EELRU can never be more than a factor of 3 worse than LRU, while in a common best case it can be better than LRU by a large factor (proportional to the number of pages in memory).

The goal of EELRU is to provide a simple replacement algorithm that adapts to reference patterns at all scales. Thus, EELRU should perform well for a wider range of programs and memory sizes than other algorithms. Practical experiments validate this claim. For a large number of programs and wide ranges of memory sizes, we show that EELRU outperforms LRU, typically reducing misses by 10% to 30%, and occasionally by much more—sometimes by a factor of two to ten. It rarely performs worse than LRU, and then only by a small amount.

**Content Indicators**

Categories and Subject Descriptors: D.4.2 [Operating Systems]:Storage Management—Storage Hierarchies, Virtual Memory

General Terms: Algorithms; Measurement; Performance

Additional Keywords and Phrases: replacement algorithms; LRU

# 1 Introduction and Overview

Modern operating systems come in a larger variety of configurations than ever before: the same personal computer OS is used in practice with physical memories ranging from 64Mbytes to well over 1Gbyte. It is a challenge for OS designers to improve virtual memory policies to obtain good performance, regardless of system configuration. For several decades, LRU has been the dominant replacement policy, either used directly or approximated in a variety of contexts. LRU has been shown empirically to often be very good—typically within a modest constant factor of optimal in misses for a fixed memory size. Nevertheless, LRU behavior can be bad, even for fairly common workloads. The simplest, well-known LRU failure mode occurs for regular access patterns larger than the size of main memory. For instance, consider any roughly cyclic (loop-like) pattern of accesses over modestly more pages than will fit in memory. Such cyclic patterns could be induced either by a loop in the program execution or by a runtime system, like a garbage collector that reuses memory. When pages are touched cyclically, and do not all fit in main memory, LRU will always evict the ones that have not been touched for the longest time, which are exactly the ones that will be touched again *soonest*.

The question is whether there can be a disciplined approach to replacement that prevents such problems, without resorting to *ad hoc* loop detection. One way to look at the behavior of LRU for large loops is to say that LRU keeps each page in memory for a long time, but cannot keep all of them for long enough. In contrast, an optimal algorithm will evict some pages shortly after they are touched in a given iteration of the loop. Evicting these pages *early* allows other pages to remain in memory until they are looped-over again. Thus, an optimal algorithm for large cyclic reference patterns is "unfair"—there is nothing particularly noteworthy about the pages chosen for "early" eviction relative to LRU. The pages are simply sacrificed because fairness is a disaster in such a situation.

The above observations form the intuition behind Early Eviction LRU (EELRU). EELRU adapts to reference patterns in order to ensure good performance regardless of memory size. By default, EELRU performs LRU replacement but diverges from LRU and evicts pages early when it notices that LRU behavior is suboptimal. This includes the case of too many pages being touched in a roughly cyclic pattern that is larger than main memory. The pattern detection is not *ad hoc*: program behavior is encoded as *recency* information (i.e., information indicating how many other pages were touched since a page was last touched). This is the same kind of information maintained by LRU, but EELRU maintains it for resident *and* (some) non-resident pages. Program behavior within a certain phase of execution exhibits consistent recency patterns and EELRU can make informed replacement decisions. In particular, EELRU can detect that LRU underperforms when *many of the fetched pages are among those evicted lately*.

This behavior of EELRU provides an interesting guarantee on its performance relative to LRU. The miss rate of EELRU can never be more than a small constant factor (less than 3) higher than that of LRU. The reason is that EELRU deviates from LRU only when the latter incurs many faults. EELRU reverts back to LRU as soon as EELRU starts incurring more faults—in other words, if EELRU is performing poorly, it will quickly return to LRU-like behavior. LRU, in contrast, can perform worse than EELRU by a factor proportional to the number of memory pages in the worst case. This factor is usually in the thousands. This guaranteed property of EELRU is interesting both because of the ubiquity of LRU (and its approximations, e.g., segmented FIFO [TuLe81, BaFe83]) and because of the commonality of the LRU worst-case pattern (a simple, large loop) in practice.

Additionally, EELRU is firmly based on a distinct principle of program locality studies, that of *timescale relativity* (see also [WKM94]). Program behavior can be studied at many timescales (for instance, real-time, number of instructions executed, number of memory references performed, etc.). Timescale relativity advocates that the timescale of a study should express only events that matter for the studied quantity. For instance, a typical hardware cache should examine different events than a virtual memory replacement policy. A loop over 600Kbytes of data is very important for the former but may be completely ignored by the latter. Timescale relativity comes into play because real programs exhibit strong phase behavior. EELRU tries to adapt to phase changes by assigning more weight to "recent" events that matter for replacement purposes. Intuitively, EELRU ignores all high-frequency references as these do not affect replacement decisions and may "dilute" time so much that important regularities are impossible to distinguish. In this article we argue that timescale relativity represents a sound principle upon which locality studies should be based. We examine some previous replacements algorithms in this light (Section 2). Also, we propose that a special kind of plot, termed a *recency-reference* graph, is appropriate for studying program locality behavior (Section 5). These plots capture regularities in program behavior and motivate EELRU.

To validate EELRU experimentally, we applied it to nineteen program traces and studied its performance. Eight of the traces are of memory-intensive applications and come from the experiments of Glass and Cao [GlCa97]. Glass and Cao used these traces to evaluate SEQ, an adaptive replacement algorithm that attempts to detect linear (not in recency but in *address* terms) *faulting* patterns. This set of traces contains representatives from all three trace categories identified in [GlCa97]: traces with large memory requirements but no clear memory access patterns, with small access patterns, and with large access patterns. We used another five traces from the Etch collection [LCBAB98]. These are also large applications, more representative of an engineering or program development workload. Finally, six more traces were collected as representatives of applications that are not memory-intensive but may have small-scale reference patterns.

The results of our evaluation are quite encouraging. EELRU performed at least as well as LRU in almost all situations and significantly better in most. Results of more than 30% fewer faults compared to LRU were *common* for a wide range of memory sizes and for applications with large-scale reference patterns. A comparison with the SEQ algorithm [GlCa97] was also instructive: SEQ is based on detecting patterns in the address space, while EELRU detects patterns in the recency distribution. Although our simulation was quite conservative (see Section 5), EELRU managed to obtain significant benefit even for traces for which SEQ did not. On the other hand, SEQ is by nature an aggressive algorithm and performed better for programs with very clear linear access patterns in the address space. Even in these cases, however, EELRU captured a large part of the available benefit.

A previous, shorter, study [SKW99] first described EELRU and the principles behind it. In the current article, the ideas have been elaborated to result in a more efficient algorithm—compared to the measurements of our earlier study, the new EELRU described here performs uniformly better and offers much more predictable behavior. Additionally, this article includes performance measurements for several more programs, as well as strong theoretical results on the performance of EELRU. Indeed, the new theoretical results provided the motivation for elaborating the algorithm to allow it greater flexibility in the choice of eviction points, which is the reason for the improved performance. Thus, we consider this study to supersede our earlier EELRU work [SKW99].

Overall, EELRU is a simple, soundly motivated, effective replacement algorithm. As a representative of an approach to studying program behavior based on recency and timescale relativity, it proves quite promising for the future.

## 2 Motivation and Related Work

The main purpose of this section is to compare and contrast the approach taken by EELRU to other replacement policies. This will help illustrate the rationale behind some of the design choices in EELRU. Management of memory hierarchies has been a topic of study for several decades. Because of the volume of work on the subject, we will limit our attention to some selected references.

Loop detection for replacement purposes has been proposed several times in order to address the shortcomings of LRU for large scale looping patterns. The well-known Atlas loop detector [BFH68] is commonly cited as the predecessor of more recent loop detection work (e.g., [MuNe80, GlCa97]). EELRU does not resort to loop detection: it is just recasting the program references in a different domain (recency), under the assumption that program regularities express themselves in that domain. As we will see in our recency-reference graphs, this assumption is justified. Additionally, the disciplined approach of EELRU is demonstrated in its theoretical guarantees compared to LRU. Unlike EELRU, traditional loop detectors can be "fooled" to perform arbitrarily worse

than LRU.

EELRU uses recency information to distinguish between pages. A recency-based standpoint (see also [Spi76, FeLW78, WoFL83]) dictates that the only way to differentiate between pages is by examining their past history of references, without regard to other information about the pages (e.g., proximity in the address space). This ensures that looping patterns of several different kinds are treated the same. Note that access patterns that cause LRU to page excessively do not necessarily correspond to linear patterns in the memory *address* space. For instance, a loop may be accessing records connected in a linked list or a binary tree. In this case, accesses are regular and repeated, but the addresses of pages touched may not follow a linear pattern. That is, interesting regularities do not necessarily appear in memory arrangements but in how recently pages were touched in the past. The SEQ replacement algorithm [GlCa97] is one that bases its decisions on address information (detecting sequential address reference patterns). Consequently, it is lacking in generality (e.g., cannot detect loops over linked lists connecting interspersed pages). Section 5 compares EELRU and SEQ extensively.

EELRU is based on the principle of timescale relativity, which helps it detect and adapt to phase changes. The first application of timescale relativity in EELRU is in determining that time advances at a slower rate for larger memories, or, equivalently, that the length of what constitutes a "phase" in program behavior is proportional to the memory size examined. This idea is by no means new. In fact, it is commonplace in many pieces of theoretical work on paging (e.g., [SlTa85, Tor98]), where an execution is decomposed into phases with working sets of size equal to that of memory.

The second application of timescale relativity in EELRU dictates that only events that matter for replacement decisions should count to advance time. In the past, several replacement algorithms based on good ideas have yielded rather underwhelming results because they were affected by events at the wrong timescale. For instance, EELRU uses reference recency information to predict future reference patterns. This is similar to the approach taken by Phalke [Pha95] with the inter-reference gap (IRG) model. Phalke's approach attempts to predict how soon pages will be referenced in the future by looking at the time between successive past references. A simpler version of the same idea is in the Atlas loop detector [BFH68], which examines only the last successive references. The loop detector fails because time is measured as the number of memory references performed. A timescale relative treatment would (for instance) define time in terms of the number of pages touched that have not been touched recently. Note the importance of this difference: time-based approaches, like IRG and the Atlas loop detector, do not filter out high-frequency information. If a loop repeats with significant variation per iteration the time between successive references will vary a lot. This is not unusual: loops may perform different numbers of operations per step during different iterations—as is, for instance, the case with many nested loop patterns. The Atlas loop detector would then

fail to recognize the regularity. More complex, higher order IRG models (such as those studied by Phalke) can detect significantly more regularities in the presence of variation. This complexity, however, makes them prohibitive for actual implementations. At the same time, the reference pattern in timescale relative terms may be extremely regular.

A view based on recency and timescale relativity can be applied to other work in the literature. Most work on replacement policies deals with specific formal models of program behavior. Indeed EELRU itself is inspired by the LRU stack model (LRUSM) [Spi76], as we will discuss in Section 3.2. LRUSM is an independent events model, where events are references to pages identified by their *recency* (i.e., the number of other pages touched after the last touch to a page). An optimal replacement algorithm for LRUSM is that of Wood, Fernandez, and Lang [WoFL83]. Unfortunately, programs cannot be modeled accurately as independent recency events. On the other hand, short program phases can be modeled very closely using LRUSM. Hence, a good on-line recency algorithm needs to be adaptive to detect phase changes. Timescale relativity (as in EELRU) is crucial for providing such adaptivity reliably.

Other well-known models are those in the class of Markov models (e.g., [CoVa76, FrGu74]). The straightforward case of a 0-th order Markov model corresponds to the well known *independent reference model (IRM)* [ADU71]. An optimal replacement algorithm for Markov models can be found in [KPR92]. Certainly Markov models can be used to codify several different aspects of the behavior of a system. Nevertheless, the standard use is to assume that memory accesses can be modelled using a Markov process. This approach tries to solve a far harder problem than that at hand. Instead of asking "which page will be referenced the farthest (or just far enough) into the future?", it asks "which pages will be referenced soon and in what order?". Deriving an answer to the former question from the answer to the latter requires accuracy that a model is very unlikely to exhibit. For instance, most of the pages referenced by real programs are re-referenced very soon and often, but the number and order of re-references is not relevant for replacement decisions. Trying to predict the re-reference patterns with accuracy is not only impossible but also irrelevant. From a timescale relativity standpoint, the common uses of Markov models attempt to predict program behavior at the wrong, much more detailed, timescale. This makes Markov model-based replacement too brittle for actual use—realistic models cannot offer any accuracy at a large enough timescale (such as that of memory replacement decisions).

Finally, EELRU can be viewed as a way to partially negate an often-stated assertion about the limits of actual eviction algorithms. Quoting from [Tor98]:

> Stated another way, the guaranteed performance of any deterministic on-line algorithm degrades sharply as the intrinsic working set size of an access sequence increases beyond [the memory size] whereas the performance of the optimal off-line algorithm degrades gracefully as

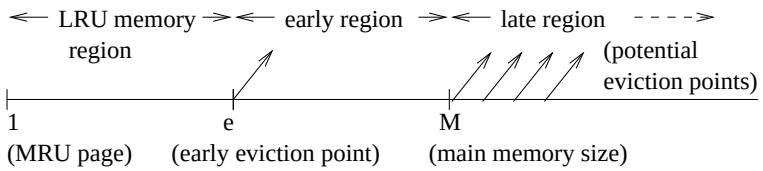


Figure 1: General EELRU scheme: LRU axis and correspondence to memory locations.

> the intrinsic working set size of an access sequence increases beyond [the memory size].

Even though this assertion holds under worst-case analysis, but, in practice, program reference sequences have enough regularity that an on-line algorithm can exploit to imitate the behavior of the optimal off-line algorithm. EELRU is an example of this approach and adds to LRU the ability to degrade gracefully its performance for large working sets when reference patterns are roughly cyclic.

# 3 The EELRU Algorithm

## 3.1 General Idea

The structure of the early-eviction LRU (EELRU) algorithm is quite simple:

1. Perform LRU replacement unless **many** pages fetched **recently** had just been evicted.
2. If **many** pages fetched **recently** had just been evicted, apply a *fallback algorithm*: either evict the least recently used page or evict the $e$-th most recently used page, where $e$ is a pre-determined recency position.

To turn this idea into a concrete algorithm, we need to define the notions of "many", "recently", etc., (highlighted above), as well as an exact fallback algorithm. By changing these aspects we obtain a family of EELRU algorithms, each with different characteristics. In this article we will only discuss a single fallback algorithm (one that is particularly simple and has a sound theoretical motivation). The algorithm is described in Section 3.2. In this section we describe the main flavor of the EELRU approach, which remains the same regardless of the actual fallback algorithm used.

Figure 1 presents the main elements of EELRU schematically, by showing the *reference recency axis* (also called the *LRU axis*) and the potential eviction points. The reference recency axis is a discrete axis where point $i$ represents the $i$-th most recently accessed page (written $r(i)$). As can be seen in Figure 1,

EELRU distinguishes three regions on the recency axis. The "LRU memory region" consists of the first $e$ blocks, which are always in main memory. (Note that the name may be slightly misleading: the "LRU region" holds the *most recently used* blocks. The name comes from the fact that this part of the buffer is handled as a regular LRU queue.) Position $e$ on the LRU axis is called the *early eviction point*. The region beginning after the early eviction point and until the memory size, $M$, is called the "early region". The "late region" begins after point $M$ and its extent is determined by the fallback algorithm used (e.g., see Section 3.2).

Recall that, at page fault time, EELRU will either evict the least recently used page or the page at point $e$ on the recency axis (i.e., the $e$-th most recently used page). The latter is called an *early eviction* and its purpose is to keep not-recently-touched pages in memory for a little longer, with the hope that they will soon be referenced again. The challenge is for EELRU to adapt to changes in program behavior and decide reliably which of the two approaches is best in every occasion.

EELRU maintains a queue of recently touched pages ordered by recency, in much the same way as plain LRU. The only difference is that the EELRU queue also contains records for pages that are *not* in main memory but were recently evicted. EELRU also keeps the total number of page references per recency region (i.e., two counters). That is, the algorithm counts the number of recent references in the "early" and "late" regions (see Figure 2a). This information enables a cost-benefit analysis, based on the expected number of faults that a fallback algorithm would incur or avoid. In essence, the algorithm makes the assumption that the program recency behavior will remain the same for the near future and compares the page faults that it would incur if it performed LRU replacement with those that it would incur if it evicted pages early.

Section 3.2 demonstrates in detail how this analysis is performed, but we will sketch the general idea here by means of an example. Consider Figure 2a: this shows the recency distribution for a phase of program behavior. That is, it shows for each position on the recency axis how many hits to pages on the position have occurred *lately*. The distribution changes in time, but remains fairly constant during separate phases of program behavior. The EELRU adaptivity mechanism is meant to detect exactly these phase changes.

If the distribution is monotonically decreasing, LRU is the best choice for replacement. Nevertheless, large loops could cause a distribution like that in Figure 2a, with many more hits in the late region than in the early region. This encourages EELRU to sacrifice some pages in order to allow others to stay in memory longer. Thus, EELRU starts evicting pages early so that eventually more hits in the late region will be on pages that have stayed in memory (Figure 2b).

EELRU is not the first algorithm to attempt to exploit such recency information for eviction decisions (e.g., see [FeLW78]). Its key point, however, is that it does so adaptively and succeeds in detecting changes in program phase

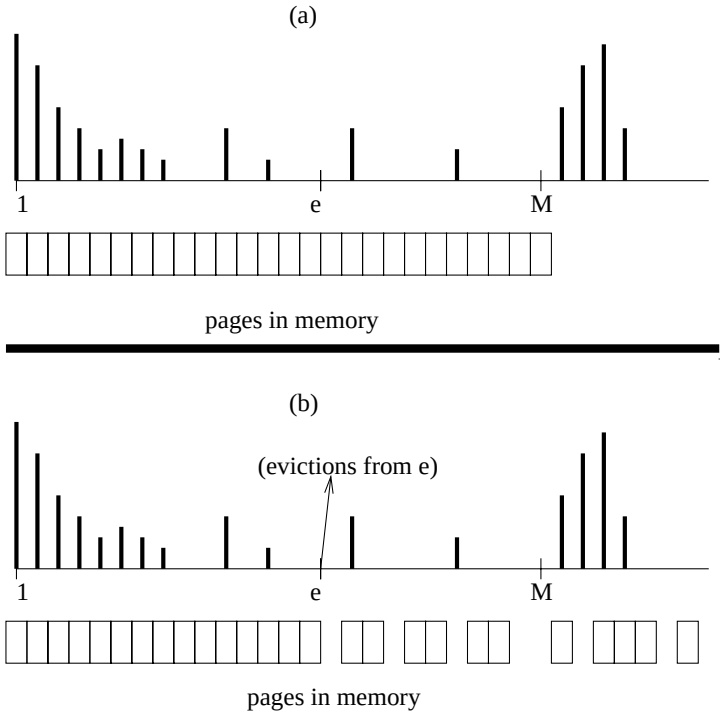


Figure 2: Example recency distribution of page touches: with LRU (left-hand side) many references are to pages not in memory. After evicting early (right-hand side) some less recent pages stay in memory. Since references to less recent pages are common (in this example distribution), evicting early yields benefits.

behavior. In the description of the general idea behind EELRU we used the word "recently". The implication is that the cost-benefit analysis performed by EELRU assigns more weight to "recent" faulting information (the weight decreases gradually for older statistics). The crucial element is the *timescale* of relevant memory references. The EELRU notion of "recent" refers neither to real time nor to virtual time (measured in memory references performed). Instead, time in EELRU is defined as the number of relevant events for the given memory size. The events considered relevant can only be the ones affecting the page faulting behavior of an application (i.e., around size $M$). These events are the page references (both hits and misses) in either the early or the late region. High-frequency events (i.e., hits to the $e$ most recently referenced pages) are totally ignored in the EELRU analysis. The reason is that allowing

high-frequency references to affect our notion of time dilutes our information to the extent that no reliable analysis can be performed. The same number of memory references may contain very different numbers of relevant events during different phases of program execution.

The basic EELRU idea can be straightforwardly generalized by allowing more than one instance of the scheme of Figure 1 in the same replacement policy. This can be viewed as having several EELRU eviction policies on-line and choosing the best for each phase of program behavior. For instance, multiple early eviction points may exist and only the events relevant to a point would affect its cost-benefit analysis. The point that yields the highest expected benefit will determine the page to be replaced. Section 3.2 discusses this in more detail.

Finally, we should point out that the simplicity of the general EELRU scheme allows for quite efficient implementations. Even though we have not provided an in-kernel version of EELRU, we speculate that it is quite feasible. In particular, EELRU can be approximated using techniques identical to standard in-kernel LRU approximations (e.g., segmented FIFO [TuLe81, BaFe83]). Just as in LRU approximations, references to the most recently used pages matter little for EELRU statistics and can be ignored. Compared to LRU, the only extra requirement of EELRU is maintaining recency information even for pages that have been evicted. Since this information only changes at page fault time, the cost of updating it is negligible.

## 3.2 A Concrete Algorithm

The first step in producing a concrete instance of EELRU is choosing a reasonable fallback algorithm. This will in turn guide our cost-benefit analysis, as well as the exact distribution information that needs to be maintained. An obvious candidate algorithm would be one that always evicts the $e$-th most recently used page. This is equivalent to applying Most Recently Used (MRU) replacement to the early region and clearly captures the intention of maintaining less recent pages in memory. Nevertheless real programs exhibit strong phase behavior (e.g., see the findings of [Den80]) which causes MRU to become unstable (pages which may never be touched again will be kept indefinitely).

The algorithm of Wood, Fernandez, and Lang [FeLW78, WoFL83] (henceforth called WFL [1] ) is a simple modification of MRU that eliminates this problem. The WFL replacement algorithm specifies two parameters representing an *early* and a *late* eviction point on the LRU axis. Evictions are performed from the early point, unless doing so means that a page beyond the late eviction point will be in memory. Thus, the algorithm can be written simply as:

[1] The WFL algorithm is called GLRU (for "generalized LRU") in [FeLW78]. To avoid confusion, we will not use this acronym, since it has been subsequently overloaded (e.g., to mean "global" LRU).

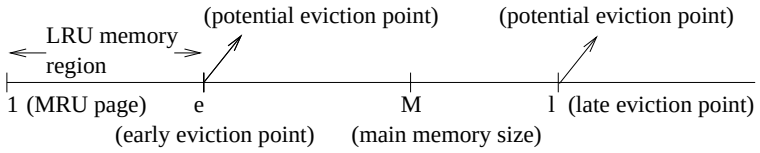


Figure 3: EELRU with WFL fallback: LRU axis and correspondence to memory locations.

if $r(l)$ is in memory
and the fault is on a less recently accessed page
then evict page $r(l)$
else evict page $r(e)$

(where $e$ is the early and $l$ the late eviction point). Figure 3 shows some elements of the WFL algorithm schematically.

It has been shown (see [WoFL83]) that there exist values for $e$ and $l$ such that the WFL algorithm is optimal for the LRU stack model of program behavior [Spi76] (that is, an independent-events model where the events are references to positions on the LRU axis). Again, however, program phase behavior (even for well-defined, long lasting phases) can cause the algorithm to underperform. This is not surprising: WFL is not an adaptive algorithm. Instead it presumes that the optimal early and late points are chosen based on a known-in-advance recency distribution. Thus, the adaptivity provided by EELRU is crucial: it is a way to turn WFL into a good on-line replacement algorithm. This is particularly true when multiple pairs of early and late eviction points exist and EELRU chooses the one yielding the most benefit (see subsequent discussion).

Even though entire programs cannot be modeled accurately using the LRU stack model, shorter phases of program behavior can be very closely approximated. Under the assumptions of the model, the WFL algorithm has the additional advantage of simplifying the cost-benefit analysis significantly. One of the properties of WFL is that when the algorithm reaches a steady state, the probability $P(n)$ that the $n$-th most recently accessed page (i.e., page $r(n)$) is in memory is:

$$P(n) = \begin{cases} 1 & \text{if } n \leq e \\ (M-e)/(l-e) & \text{if } e < n <= l \\ 0 & \text{otherwise} \end{cases}$$

The probability distribution is shown in Figure 4. Now the cost-benefit analysis for EELRU with WFL fallback is greatly simplified: we can estimate the number of faults that WFL would incur (at steady state) and compare that number to LRU. We will call *total* the number of recent hits on pages between $e$ and $l$ (in reference recency order). Similarly, we will call *early* the number of recent hits on pages between $e$ and $M$. The eviction algorithm then becomes:

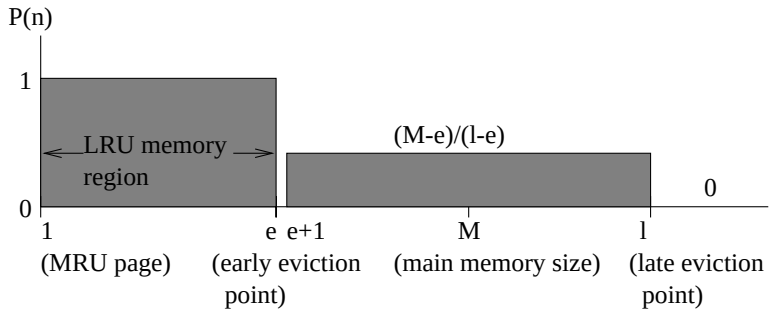


Figure 4: Probability of being in memory for a page with a given recency.

if $total \cdot (M-e)/(l-e) \leq early$
or ($r(l)$ is in memory
and the fault is on a less recently accessed page)
then evict the least recently accessed page
else evict page $r(e)$

We can now consider the obvious generalization of the algorithm where several instances of WFL, each with different values of $e$ and $l$, are active in parallel. By $e_i$, $l_i$, $total_i$, and $early_i$ we will denote the $e$, $l$, *total* and *early* values for the $i$-th instance of the algorithm. Then, the instance of WFL that will actually decide what page is to be evicted is the one that maximizes the expected benefit value $total_i \cdot (M-e)/(l-e) - early_i$. If all such values are negative, plain LRU eviction is performed. Note that in the case of multiple early and late eviction points, EELRU adaptivity performs a dual role. On one hand, it produces online estimates of the values of $e$ and $l$ for which the algorithm performs optimally (also, plain LRU is no more than another case for these values). On the other hand, the adaptivity allows detecting phase transitions and changing the values accordingly.

In the case of multiple early and late eviction points, one more modification to the basic WFL algorithm makes sense. Since not all late eviction points are equal, it is possible that when the $i$-th instance of WFL is called to evict a page, there is a page $r(n)$ in memory, with $n > l_i$. In that case, the algorithm should first evict all such pages (to guarantee that, in its steady state, all pages less recently referenced than $l_i$ will not be in memory). Note that this modification of the basic WFL algorithm does not affect its steady state behavior (and, consequently, its proof of optimality for the LRU stack model, as presented in [WoFL83]). Taking the change into account, our final eviction algorithm becomes:

let *benefit* be the maximum of the values
$total_i \cdot (M - e_i)/(l_i - e_i) - early_i$
and $j$ be the index for which this value occurs

if *benefit* $\leq 0$
or a page $r(n)$, $n > l_j$ is in memory
or ($r(l_j)$ is in memory
and the fault is on a less recently accessed page)
then evict the least recently accessed page
else evict page $r(e_j)$

This form of EELRU is the one used in all experiments described in this article.

# 4 Theoretical Results

Two main theoretical results are pertinent to EELRU. The first is the proof of robustness for the algorithm with respect to LRU: we show that EELRU can never incur more than 3 times as many faults as LRU, while LRU can incur up to $O(M)$ as many faults as EELRU. The second result shows that, with the right choice of parameters, the replacement algorithm of Wood, Fernandez, and Lang (WFL) is a *stack algorithm*.

## 4.1 EELRU vs. LRU

An interesting property of EELRU is that it is *robust* with respect to LRU under worst-case analysis. In particular, EELRU will never perform more than a small constant factor worse than LRU, while LRU can perform worse than EELRU by a factor proportional to the number of memory pages. The exact values of these factors depend somewhat on the parameters picked for the EELRU algorithm—e.g., the number and positions of early and late eviction points, and the speed of adaptation. Nevertheless, for any eviction point and a reasonable adaptation approach, it is easy to bound the worst case faults of EELRU to be 3 times the faults of LRU. Conversely, LRU can always incur $O(M - e)$ times as many faults as EELRU.

The proof is based on the observation that EELRU diverges from LRU only when the latter has incurred many faults lately and reverts back to LRU when it detects that LRU would have performed better. Thus, the only cases when LRU is better than EELRU are such that LRU incurs many faults (enough to tempt EELRU to diverge). In such cases the ratio of miss rates of the two algorithms is never worse than a constant. Conversely, a steady loop slightly larger than memory but within one of the late regions of EELRU will cause LRU to suffer misses for every page, while EELRU will suffer a constant number of misses per iteration.

For simplicity, we consider a version of EELRU that maintains only one early

eviction point, at recency $e$. A generalization of the proofs is straightforward (but tedious, especially for Theorem 2). The adaptivity mechanism maintains the $M$ most recent references beyond recency position $e$. If at least $M/2$ of these are in the early region, the algorithm does LRU evictions. Otherwise, it evicts early (i.e., the $e$-th most recently used page).

**Theorem 1** *LRU can incur $O(M-e)$ times as many faults as EELRU.*

*Proof*: Consider a linear loop over $M+1$ pages that iterates $O(M)$ times. The LRU algorithm will incur a fault for each reference in the loop. After the first $M/2+1$ references, EELRU will perform early evictions. From that point on, EELRU will incur one fault for each $M-e+1$ references. For $O(M)$ loop iterations, the ratio of LRU to EELRU faults will be $O(M-e)$. □

The next theorem gives a bound of 3 in the worst-case performance of EELRU relative to LRU. The bound can be tightened more (up to about $2.5-e/M$, we conjecture) but the conservative proof for the 3-fold bound is appealingly simple.

**Theorem 2** *EELRU will never incur more than 3 times as many faults as LRU.*

*Proof*: We will define a "potential" function on the state of EELRU, such that the potential can rise only if a reference would be an LRU fault, and if EELRU incurs a fault that is not an LRU fault, the potential will drop significantly. In this way, we can guarantee that EELRU can incur more faults than LRU only if LRU incurs enough faults (one third the number) in the first place. The challenge is in defining appropriately such a potential function, to capture the EELRU adaptivity.

Consider a function $f$ capturing the information of whether a reference (identified by recency) was in the early region (and, thus, would have been an LRU hit) or not:

$$f(i) = \begin{cases} 1 & \text{if } i\text{-th most recent reference was to a recency position} > M \\ 0 & \text{otherwise} \end{cases}$$

Then we can define the following quantities $p$ and $s$: $p = \sum_{i=1}^{M} (M+1-i) f(i)$, $s = \sum_{i=1}^{M} f(i)$. It is clear that $0 \le p \le \frac{M(M+1)}{2}$ and $0 \le s \le M$. That is, $s$ shows how many recent references encourage deviation from LRU and $p$ is a measure of the potential for deviation, such that more recent references are favored. In intuitive terms, $p$ captures the "decaying" of EELRU statistics that is fundamental for its adaptivity.

Additionally, we will define $d$ as the number of pages that are not resident in memory but are more recently accessed than the least recent page in memory. This corresponds to the total length of the "gaps" in the sequence of resident pages, as shown in Figure 2.

Now we will consider the effect of a reference with recency $r$ (i.e., to the $r$-th most recently accessed page) on the values of the quantities $p$ and $d$ during an execution of EELRU. The new values are designated $p'$ and $d'$.

- if $s > \frac{M}{2}$ (EELRU is in "early eviction mode"):
  - if $r \leq M$ then $p' = p - s$, $d' = d$
  - if $r > M$ then $p' = p + M - s$, $d' \leq d + 1$
- if $s \leq \frac{M}{2}$ (EELRU is in "LRU eviction mode"):
  - if $r \leq M$ then $p' = p - s$,if EELRU incurs a fault then $d' < d$, otherwise $d' = d$.
  - if $r > M$ then $p' = p + M - s$, $d' \leq d$

For simplicity, we can combine the above quantities into a single "potential" function, $t$, with $t = p + d\frac{M}{2}$. By definition of $t$, it is originally 0 and generally $t \geq 0$. From the above case analysis, one can see that for every fault incurred by EELRU that is not an LRU fault (i.e., $r \leq M$), $t$ is reduced by at least $\frac{M}{2}$ (either $p$ is reduced by more than $\frac{M}{2}$ while $d$ stays the same, or $d$ is reduced, while $p$ does not increase). The only way to increase $t$ is if $r > M$, that is LRU incurs a fault (which may also be a fault for EELRU). The increment is at most $M$.

To summarize, the potential can rise by at most $M$ if LRU incurs a fault, drops by at least $\frac{M}{2}$ if EELRU incurs a fault that is not an LRU fault, and has to stay non-negative. That is, every two faults of EELRU that are not faults for LRU can be incurred only if both EELRU and LRU incur one more fault. Thus, EELRU can incur at most 3 faults for each LRU fault. □

## 4.2 An Optimal Stack Algorithm for the LRUSM

The LRU Stack Model (LRUSM) is the simplest probabilistic model in the recency space. It is an independent events model, where the events are references to recency positions. That is, the model prescribes independent trials, all governed by the same fixed probability distribution. An outcome of $x$ in any trial, means that the $x$-th most recently accessed page is referenced. Wood, Fernandez, and Lang [WoFL83] showed an optimal replacement algorithm for the LRU stack model (i.e., for workloads that are random processes following the LRU stack model). In this section we first discuss a slight elaboration of their algorithm and show that it is a *stack algorithm*. Stack algorithms [MGST70] are a

well-known class of replacement algorithms. Their identifying property is the *inclusion property*: at any point in the execution of a workload, if a page is in a memory of size $M$, then it will also be in memory for any size $M' > M$. It follows that the miss curve (the curve formed by plotting the number of misses over a range of memory sizes) for a stack-algorithm-managed memory is non-increasing. This is a desirable property for a replacement algorithm: adding more memory should improve system behavior.

In fact, the "stack algorithm" property seems so fundamental that one may be tempted to speculate that all optimal algorithms for a reference model are stack algorithms. This is not so. Optimality of an on-line algorithm is defined only in respect to a reference model—nothing is guaranteed about the algorithm's behavior on arbitrary (i.e., non-conforming to the model) input. Being a stack algorithm is an intrinsic property of the algorithm, regardless of the input.

Wood, Fernandez, and Lang demonstrated that the optimal eviction algorithm for the LRUSM is of a form that has two parameters, which we call $e$ and $l$ (for "early eviction point" and "late eviction point", respectively). The $e$ most recently accessed pages will always be in memory. Pages with recency between $e$ and $l$ are in memory with probability $\frac{M-e}{l-e}$ (i.e., proportional to the amount of memory available to hold them). Pages that are not among the $l$ most recently accessed are guaranteed not to be in memory. Consequently, the expected hit ratio, $H(M)$, for memory size $M$ and the optimal replacement algorithm will be $H(M) = P(0,e) + \frac{M-e}{l-e}P(e,l)$, where $P(x,y), x < y$ is the probability that the page accessed is among the $y$ most recently accessed but not among the $x$ most recently accessed. The optimal algorithm is the one that makes the choice of $e$ and $l$ so that the hit ratio is maximized. There may be multiple values of $e$ and $l$ that yield the same hit ratio. The following algorithm fixes these values:

**Algorithm 1** *:*

- *Among all values of $e$ and $l$ that maximize $H(M)$, pick the highest value of $e$.*
- *For that $e$, pick the lowest value of $l$ that maximizes $H(M)$.*

We will show that the eviction algorithm resulting from this choice of parameters is a stack algorithm. The main property of the value pairs chosen by Algorithm 1 is that the intervals $e, \ldots, l$ (called the *eviction ranges*) cannot overlap (except for their endpoints) for different memory sizes. Thus, if for memory size $M_1$ the algorithm picks points $e_1$ and $l_1$, for memory size $M_2 > M_1$ the algorithm has to either pick the same points, or pick an early eviction point $e_2$ such that $e_2 \geq l_1$. In both cases, all pages that will be in a memory of size $M_1$, will also be in a memory of size $M_2$ at all points during the execution of a workload, thus making the algorithm be a stack algorithm.

Let us now show that the intervals $e, \ldots, l$ cannot overlap for different memory sizes. Consider the $e$ and $l$ picked by Algorithm 1 for a memory size $M$, and let us assume that $e < M < l$ (i.e., the optimal replacement for memory size $M$ is not LRU). Then for a recency point $x$, $x < e$, we have:

$$P(x,e) + P(e,l)\frac{M-e}{l-e} \geq (P(x,e) + P(e,l))\frac{M-x}{l-x}$$

(The left hand side is the hit ratio when $e$ is the early eviction point, in which cases references to positions $x$ through $e$ are hits. The right hand side is the hit ratio when $x$ is the early eviction point. Since $e$ yields an optimal hit ratio, the left hand side is greater than or equal to the right hand side.) From this, with some algebraic manipulation, we eventually get:

$$\frac{P(x,e)}{e-x} \geq \frac{P(e,l)}{l-e} \tag{1}$$

Inequality 1 essentially states that the average probability density in a range right before the early eviction point cannot be lower than that in the eviction range (i.e., the $e, \ldots, l$ interval).

Similarly, for a recency point $x$, $e < x < M$:

$$P(e,x) + P(x,l)\frac{M-x}{l-x} < (P(e,x) + P(x,l))\frac{M-e}{l-e}$$

(The inequality is strict because $e$ is the latest optimal early eviction point so no later point can yield the same hit ratio.) Again, from this we get:

$$\frac{P(e,x)}{x-e} < \frac{P(e,l)}{l-e}$$

Exactly the same is true of recency points $x$, $M \leq x < l$. There we have:

$$P(e,x)\frac{M-e}{x-e} < (P(e,x) + P(x,l))\frac{M-e}{l-e}$$

(The inequality is strict because $l$ is the earliest optimal late eviction point corresponding to $e$.) From this we get the same expression as above:

$$\frac{P(e,x)}{x-e} < \frac{P(e,l)}{l-e} \tag{2}$$

In other words, inequality 2 holds for all $x$, $e < x < l$ and states that the average probability density in a range starting at $e$ and ending before $l$ must be lower than the total average in the eviction range.

Finally, we consider the case of a recency point $x$, $l < x$. We get:

$$P(e,x)\frac{M-e}{x-e} \leq (P(e,x) - P(l,x))\frac{M-e}{l-e}$$

From which we have:

$$\frac{P(e,x)}{x-e} \leq \frac{P(e,l)}{l-e} \tag{3}$$

That is, the average probability density in a range starting at $e$ and extending beyond $l$ is at most equal to the average density in the eviction range.

Note that none of the above inequalities depend on the memory size, $M$. Using these inequalities it is easy to show that the eviction ranges $e_1, \ldots, l_1$ and $e_2, \ldots, l_2$ defined by Algorithm 1 for memory sizes $M_1$ and $M_2$, respectively, cannot overlap, except at their endpoints. Consider the case $e_1 < e_2 < l_1 < l_2$. (The only other case is $e_1 < e_2 \leq l_2 < l_1$, which is treated analogously.) If we use the notation $d(x,y)$ to represent the fraction $\frac{P(x,y)}{y-x}$ (i.e., the probability density in that range) we get:

$$d(e_1, e_2) < d(e_1, l_1)$$

(From applying inequality 2 with $e = e_1, l = l_1, x = e_2$.)

$$d(e_1, e_2) \geq d(e_2, l_2)$$

(From applying inequality 1 with $e = e_2, l = l_2, x = e_1$.) The two above inequalities imply that:

$$d(e_2, l_2) < d(e_1, l_1) \tag{4}$$

But similarly we get from the first inequality:

$$d(e_1, e_2) < d(e_1, l_1) \Rightarrow d(e_2, l_1) > d(e_1, l_1)$$

(From the definition of $d(x,y)$ and simple properties of fractions.) Also we have:

$$d(e_2, l_1) < d(e_2, l_2)$$

(From applying inequality 2 with $e = e_2, l = l_2, x = l_1$.) From the above we get $d(e_2, l_2) > d(e_1, l_1)$, which contradicts inequality 4.

This concludes the proof that eviction ranges chosen with Algorithm 1 do not have common internal points. This property directly implies that the WFL replacement algorithm with points $e$ and $l$ chosen through Algorithm 1 is a stack algorithm. To see this, consider that the algorithm can only possibly evict from recency point $e$ or $l$ and that the choice of point does not depend on memory size $M$. Consider two memory sizes $M_1$ and $M_2$ with $M_2 > M_1$. If the corresponding optimal eviction points, $e_1, e_2, l_1, l_2$ are equal (i.e., $e_1 = e_2$ and $l_1 = l_2$), then all eviction decisions for both memories will be identical (but some references resulting in evictions from the smaller memory will be hits in the larger memory). Similarly, if $e_2 \geq l_1$, the only pages that the larger memory may evict are those with recency above $l_1$, and these pages will never be stored in the smaller memory either.

# 5 Experimental Assessment

## 5.1 Settings and Methodology

To assess the performance of EELRU, we used program traces covering a wide range of memory access characteristics. There is no single widely acceptable set of memory-intensive applications for virtual memory studies. Nevertheless, our studied set is among the largest in the literature and is derived from three different sources. Eight of the traces are of memory-intensive applications and were used in the experiments of Glass and Cao [GlCa97]. Five more traces are part of the Etch traces collection [LCBAB98]. Another six traces are of programs that are commonly used in garbage collection and memory allocation studies. Our study sets contain some overlap (two executions of the gcc compiler, as well as two traces of a Perl interpreter, all for different inputs). We describe the traces in more detail below.

### 5.1.1 The SEQ Study Traces

The eight traces from [GlCa97] are only half of the traces used in that study. The rest of the experiments could not be reproduced because the reduced trace format used by Glass and Cao sometimes omitted information that was necessary for accurate EELRU simulation.[2] To see why this happens, consider the behavior of EELRU: at any given point, early evictions can be performed, making the algorithm replace the page at point $e$ on the LRU axis. Thus, the trace should have enough information to determine the $e$-th most recently accessed page. This is equivalent to saying that the trace should be sufficiently accurate for an LRU simulation with a memory of size $e$. The reduced traces of Glass and Cao have limitations on the memory sizes for which LRU simulation can be performed. Thus, the minimum simulatable memory size for EELRU (which is larger than the minimum simulatable size for LRU) may be too large for meaningful experiments. For instance, consider an EELRU simulation for which the "earliest" early eviction point is such that the early region is 60% of the memory (that is, 40% of the memory is managed using strict LRU). Then the minimum memory for which EELRU can be simulated will be 2.5 times the size of the minimum simulatable LRU memory. For some traces, this difference makes the minimum simulatable memory size for EELRU fall outside the memory ranges tested in [GlCa97]. For example, the "gcc" trace was in a form that allowed accurate LRU simulations only for memories larger than 475 pages (see [GlCa97]). Using the above early eviction assumptions, the minimum EELRU simulatable memory size would be 1188 pages, well outside the memory range for this experiment (the trace causes no faults for memories above 900 pages).

Figure 5 contains information on the eight traces for which EELRU could be

[2] In fact, two more traces from [GlCa97], "es" and "fgm", could have been used but were not made available to us.

| Program | Description | Min. simulatable LRU memory (4KB pages) |
|---|---|---|
| `applu` | Solve 5 coupled parabolic/elliptic PDEs | 608 |
| `gnuplot` | Postscript graph generation | 388 |
| `ijpeg` | Image conversion into JPEG format | 278 |
| `m88ksim` | Microprocessor cycle-level simulator | 491 |
| `murphi` | Protocol verifier | 533 |
| `perl` | Interpreted scripting language | 2409 |
| `trygtsl` | Tridiagonal matrix calculation | 611 |
| `wave5` | Plasma simulation | 913 |

Figure 5: Information on traces used in [GlCa97]

simulated, for easy reference. It is worth noting that this set of traces contains representatives from all three program categories identified by Glass and Cao. These are *programs with no clear patterns* (murphi, m88ksim), *programs with small-scale patterns* (perl), and *programs with large-scale reference patterns* (the rest of them).

### 5.1.2 The Etch Traces

The five Etch traces are of memory-intensive SPEC95 benchmark applications, with no clear large-scale reference patterns. The Etch traces collection [LCBAB98] also contains several traces from popular Windows applications (Word, Powerpoint, etc.). Nevertheless, these did not seem to exhibit interesting virtual memory behavior (at least for the inputs used to generate the traces). Indeed, the Etch traces were collected with processor cache studies in mind. The SPEC95 applications in the Etch traces, however, appeared more promising for virtual memory studies. (For completeness, we also simulated EELRU on the rest of the Etch traces and its performance was identical to LRU.)

The five traces studied are those of "go" (an AI program that plays the game of "Go"), "CC1" (the compiler core for the gcc compiler, building SPARC code), "compress" (a compression utility used to compress and decompress files in memory), "perl" (the well-known scripting language, with an input manipulating strings and prime numbers), and "vortex" (a database program).

### 5.1.3 The Small-scale Traces

An extra six traces were used to supply more data points. These are traces of executions that do not consume much memory. Hence, all their memory patterns are, at best, small-scale. The applications traced are "espresso" (a circuit simulator), "gcc" (the C compiler, compiling its own source code), "ghostscript" (a PostScript engine), "grobner" (a formula-rewrite program), "lindsay" (a communications simulator for a hypercube computer), and "p2c" (a Pascal to C translator).

### 5.1.4 Simulation Parameters

All of the simulations were performed with recency information maintained for 2.5 times more pages than fit in memory. This recency range was split in about 40 equal-size regions (rounded so that the end of a region coincided with the memory size, $M$). A histogram of references was maintained (i.e., the number of recent references for each region). For each region before recency $M$ and after $M$, its endpoints are possible early and late eviction points, repectively. The LRU part of memory was as small as possible, given the resolution of the traces. This was commonly above 100 pages for all large traces.

One more parameter affects simulation results significantly. Recall that replacement decisions should be guided by recent program reference behavior. To achieve this, distribution values need to be "decayed". The decay is performed in a memory-scale relative way: the values for all our statistics are multiplied by a weight factor progressively so that the $M$-th most recent reference ($M$ being the memory size) has 0.3 times the weight of the most recent one. The algorithm is not very sensitive with respect to this value. Values between 0.1 and 0.5 yield almost identical results. Essentially, we just need to ensure that old behavior matters exponentially less, and decays on a timescale comparable to the rate of replacement. For the purpose of decaying statistics, "time" was defined as the number of references to pages less recent than the last used eviction point. This enables adapting the "interesting region" to the locality of the current workload.

## 5.2 Locality Analysis

To show the memory reference characteristics of our traces, we plotted *recency-reference* graphs. Such graphs are scatter plots that map each page reference to the page position on the recency axis. High-frequency references (i.e., references to pages recently touched) are ignored, thus resulting in graphs that maintain the right amount of information at the most relevant timescale for a clear picture of program locality. For instance, consider the recency-reference graph for the wave5 trace, plotted in Figure 6. The graph is produced by *ignoring* references to the 1000 most recently accessed pages. If such references were taken into

account, the patterns shown in the graph could have been "squeezed" to just a small part of the resulting plot: interesting program behavior in terms of locality is usually very unevenly distributed in time.

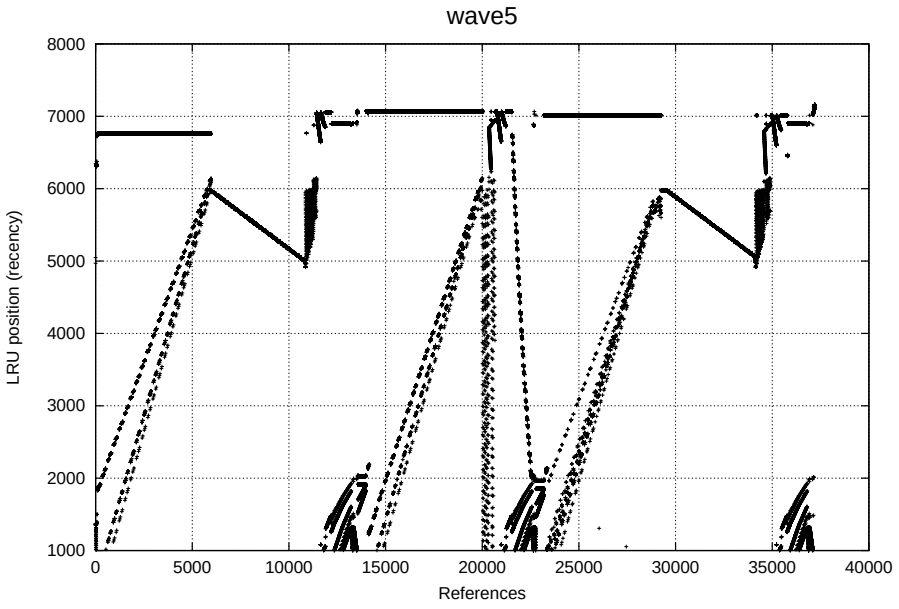


Figure 6: Recency-reference graph for wave5.

Based on this graph, we can make several observations. First, wave5 exhibits strong, large-scale looping behavior. There seem to be loops accessing over 6000 pages. The horizontal lines represent regular loops (i.e., pages are accessed in the same order as their last access after touching the same number of other pages). Note also the steep "upwards" diagonals which commonly represent pages being re-accessed in the *opposite* order from their last access.

Patterns in recency-reference graphs convey a lot of information of this kind and offer several advantages over the usual space-time graphs (e.g., see the plots in [Pha95, GlCa97]) for program locality analysis. To name a few:

- The information is more relevant. Instead of depicting which page gets accessed, recency-reference graphs show how recently a page was accessed before being touched again.
- High frequency information (e.g., hits to the few most recently accessed pages) dilutes time in space-time graphs. It is common that all interesting

behavior (with respect to faulting) occurs only in a small region of a space-time graph. Such information does not affect recency-reference graphs.

- First-time reference information may dominate a space-time graph (e.g., allocations of large structures). Such information is irrelevant for paging analysis and does not appear in a recency-reference graph.

Figures 7, 8, and 9 present recency-reference graphs for representative traces. There are several observations we can make:

- All programs exhibit strong phase behavior in recency terms. That is, their access patterns exhibit some clearly identified features that commonly persist. Comparing the values of the horizontal and vertical axes gives a good estimate of how long features persist (note that the aspect ratio of the plots is usually not 1:1). For all plots, features most commonly last for at least as many page references as their "size" in pages.

- The gnuplot graph exhibits a strong and large loop restricted to a very narrow recency region. All references in the gnuplot trace were either to the 200 most recently accessed pages (these are filtered out in the plot) or to pages above the 15000 mark on the recency axis! This is to be expected, as this trace shows gnuplot when run with a very large (8MB) input file, which causes it to create an internal data structure of over 64MB and iterate over it three times. This example is somewhat artificial but interesting: it shows what happens when a program written with no paging considerations in mind is "abused" with inputs large enough to make it page.

- The m88ksim graph initially displays a large loop (note the short horizontal line), followed by a "puff-of-smoke" feature, and an area without distinctive patterns. The initial loop is a simple linear loop over a little more than 4500 pages. Note that it lasts for approximately 4500 references, indicating that it is just a linear loop with two iterations. The "puff-of-smoke" pattern is characteristic of *sets of pages* that are accessed *all together* but *in "random" (uniform) order*. When some pages in the set get touched, pages before them on the recency axis become less-recently-touched (i.e., move "upward"). Gradually, all accesses concentrate to higher and higher points on the recency axis (with the size of the set being the limit). The feature-less part of the m88ksim graph represents "random" accesses to a large structure. Given the nature of the application we speculate that this could be a heavily used hash-table.

- The perl graph also exhibits "puff-of-smoke" features, together with steep diagonals (recall that these represent pages being re-accessed in the opposite order). The applu graph exhibits strong loops and steep upward diagonals, but also downward diagonals—a sign of selective iteration over

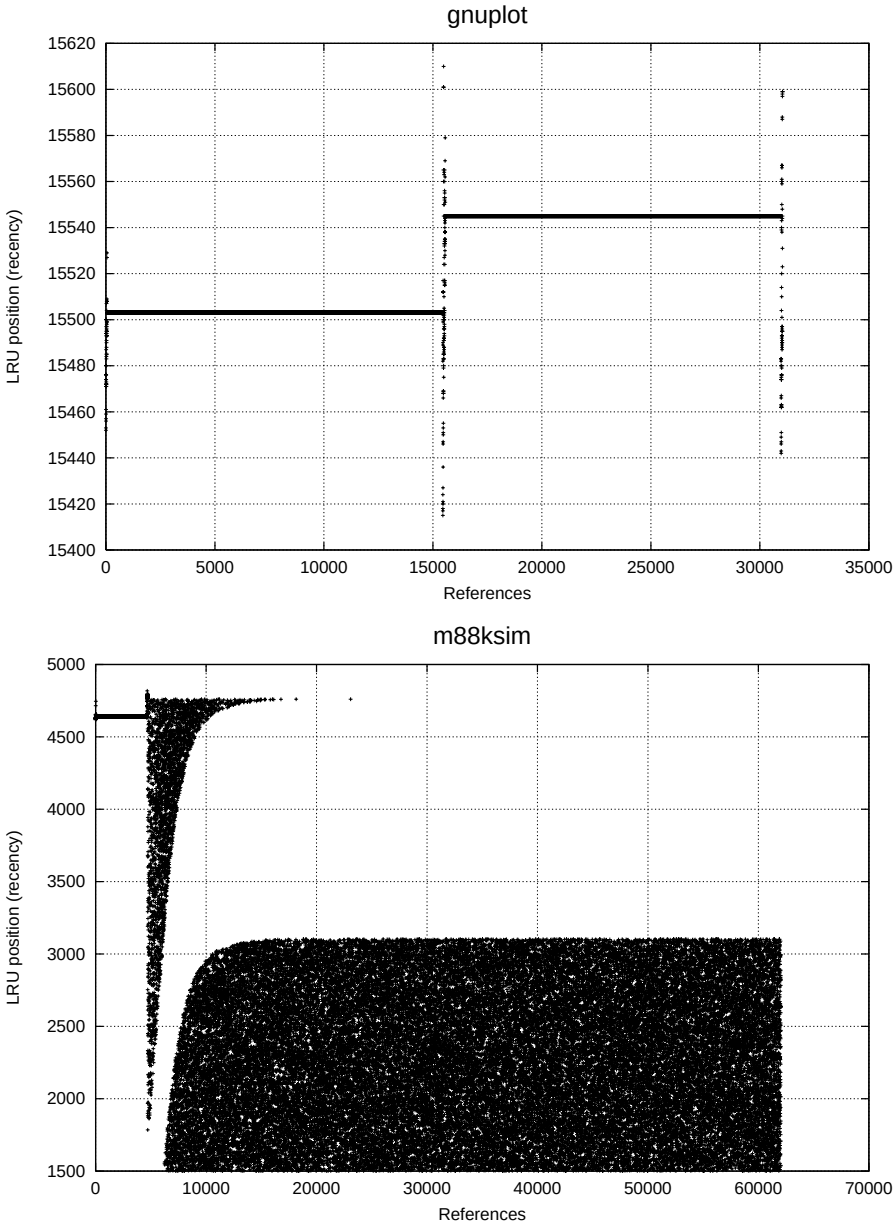


Figure 7: Recency-reference graphs for gnuplot and m88ksim.

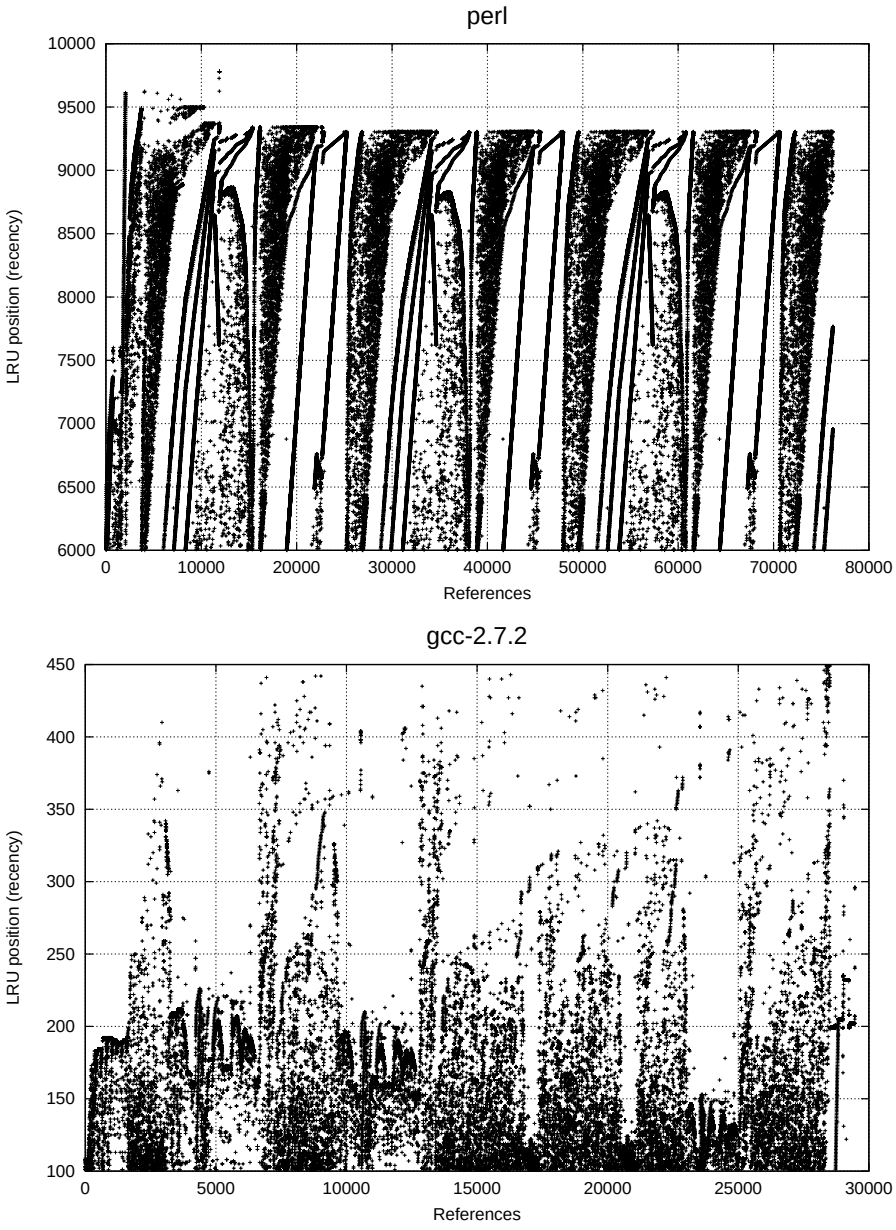


Figure 8: Recency-reference graphs for perl (from [GlCa97]) and gcc (from the small scale traces).

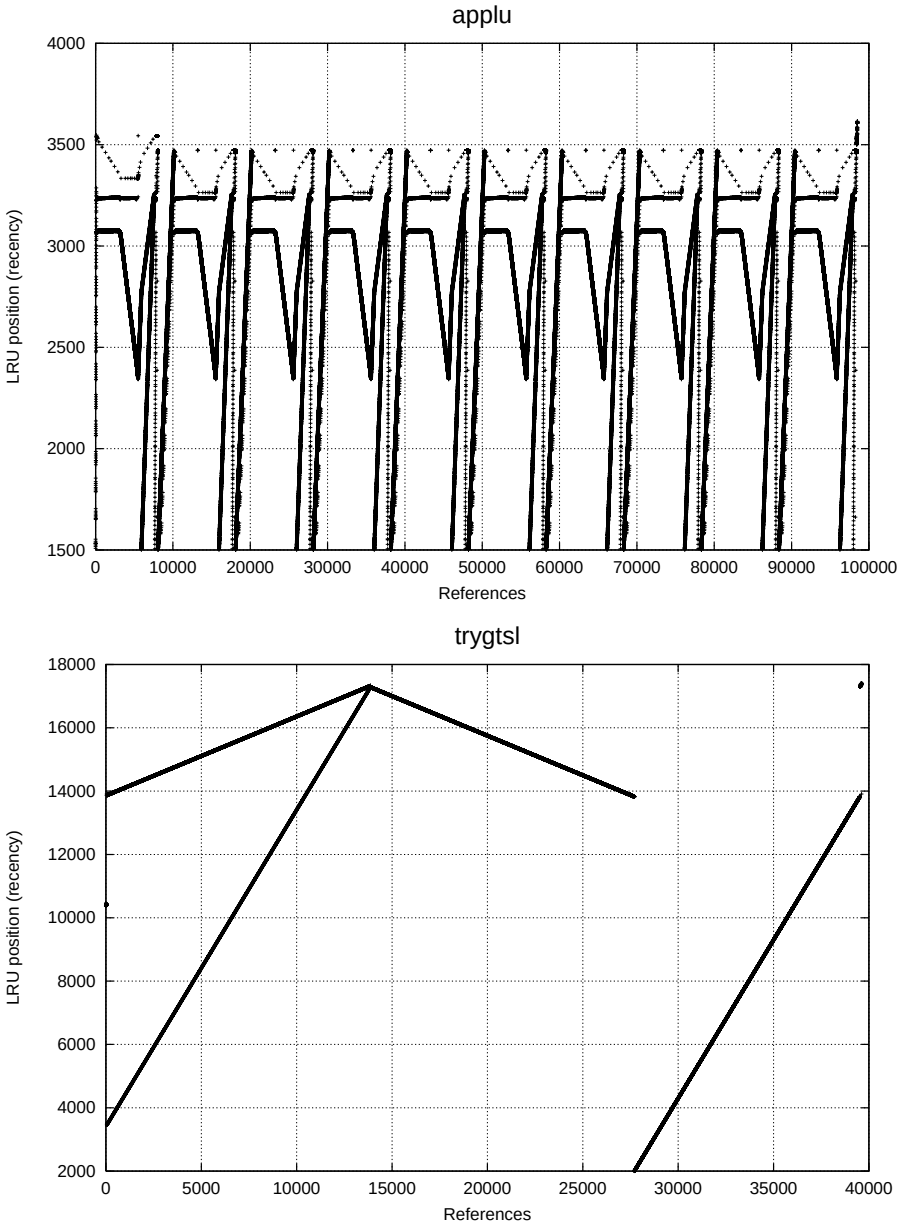


Figure 9: Recency-reference graphs for applu and murphi

some of the same data in the same order. The trygtsl graph shows linear patterns, but no loops that iterate unchanged.

Other recency-reference graphs are similar to the ones shown.

Based on the recency-reference graphs we can identify areas for which EELRU should exhibit a clear benefit. Thus, if a graph displays *high-intensity (dark) areas right above low-intensity (light) areas*, EELRU should be able to evict some pages early and keep in memory those that will be needed soon. Comparing these graphs with the results of our simulations (in the following sections) shows that this is indeed the case: the memory sizes for which EELRU is particularly successful are near such intensity boundaries.

## 5.3 Experimental Results

### 5.3.1 SEQ Study Traces

Figures 10 and 11 show the page faults incurred by EELRU, LRU, OPT (the optimal, off-line replacement algorithm), and SEQ for each of the eight memory-intensive traces. SEQ is the algorithm of Glass and Cao [GlCa97], with which these traces were originally tested. A detailed analysis of the behavior of LRU relative to OPT on these traces can be found in [GlCa97]. Here we will restrict our attention to EELRU.

As can be seen, EELRU consistently performs better than LRU for seven out of eight traces (for murphi, EELRU essentially performs LRU replacement). A large part of the available benefit (as shown by the OPT curve) is captured for all applications that exhibit clear reference patterns. A comparison with the SEQ algorithm is also quite instructive.[3]

The idea behind SEQ is to detect access patterns by observing *linear faulting sequences* (the linearity refers to the address space). Thus, SEQ is based on detecting address-space patterns, while EELRU is based on detecting (coarse-grained) recency-space patterns. Each approach seems to offer distinct benefits. EELRU is capable of detecting regularities that SEQ cannot capture. For instance, a linked list traversal may not necessarily access pages in address order, even though it could clearly exhibit strong looping behavior. Such traversals are straightforwardly captured in the recency information maintained by EELRU. On the other hand, SEQ can detect linear patterns more quickly than EELRU, and thus get more of the possible benefit in such cases. The reason is that recency information does not become available until a page is re-accessed (i.e., during the second iteration of the loop), while address information is available right away. The latter observation has consequences on the robustness of the two algorithms: EELRU is fairly conservative and only diverges from LRU in the case of persistent reference patterns. SEQ, on the other hand, is more risky

[3]The results for SEQ were obtained by running the simulator of Glass and Cao on the traces. Testing SEQ on other traces would require significant re-engineering of the simulator, as its replacement logic is tied to the trace reduction format used in [GlCa97].

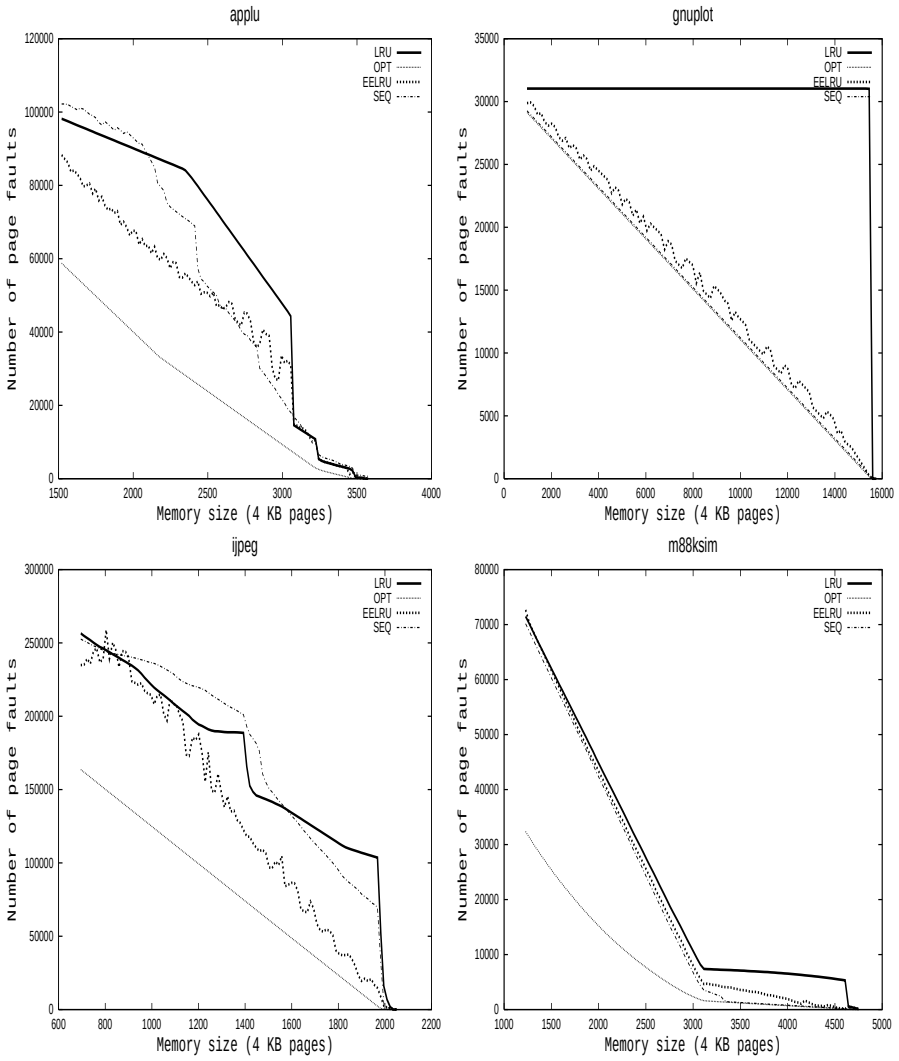


Figure 10: Fault plots for traces from the SEQ study. For gnuplot, the SEQ curve almost overlaps the OPT curve.

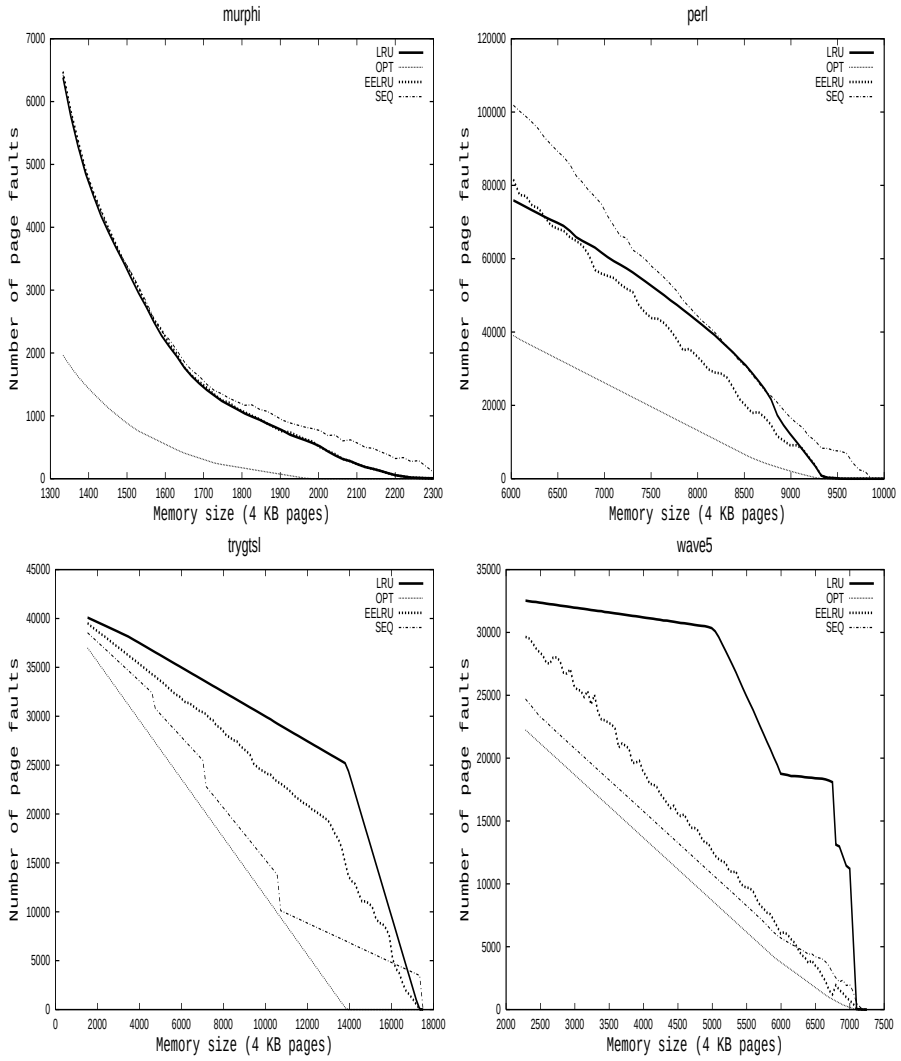


Figure 11: Fault plots for more traces from the SEQ study. For murphi, the EELRU curve almost overlaps the LRU curve.

and guesses about traversals which it has not encountered before (e.g., presumes that all sequential scans of more than 20 pages will be larger than main memory). Therefore, we expect that EELRU is much less likely to cause harm for randomly selected programs with no large-scale looping patterns.

The results of our experiments agree with the above analysis. EELRU outperformed SEQ for three of the traces (applu, ijpeg, perl), SEQ was better for another three (gnuplot, trygtsl, and wave5), while for m88ksim and murphi the difference was small (EELRU was slightly better in one case and slightly worse in the other). Note that SEQ performed better for programs with very clear linear access patterns. This is a result of the early loop detection performed by SEQ. Even in these cases, however, EELRU captured most of the available benefit. For all traces with recency patterns that could be exploited, EELRU consistently yielded improvements to LRU, even when the SEQ results seemed to indicate that few opportunities exist (e.g., ijpeg, perl).

Overall, we believe that the recency-based approach of EELRU is simpler, intuitively more appealing, and of more general applicability than address-based approaches like SEQ. The generality conjecture cannot, of course, be proven without extensive experiments and widely accepted "representative workloads" but the preliminary results of our experiments seem to confirm it.

Finally, as can be seen in the plots, EELRU is not a stack algorithm as it exhibits what is known as "Belady's anomaly": increasing the size of physical memory may increase the number of page faults. This is a result of the adaptive behavior of EELRU, which is intrinsic to the algorithm. As we saw in Section 4.2, the fallback algorithm used is a stack algorithm. Nevertheless, the recency histogram that EELRU keeps can change with time and the notion of "time" depends on the memory size. Thus, the exact adaptation decisions of EELRU are different for different memory sizes.

#### 5.3.2 Etch Traces

Figures 12 and 13 show the results of our experiments on the Etch traces. Note that the perl trace shown is for a different execution (much smaller input) than that of Figure 11. Nevertheless it exhibits very similar behavior. Overall, EELRU performs much better than LRU in three of these traces (compress, go, perl) and similarly to LRU in the other two (cc1, vortex). This supports the claim that EELRU can outperform LRU in common cases, and will never be "fooled" by much, as its adaptivity will help it revert to LRU when needed.

#### 5.3.3 Small-scale Traces

Our second experiment applied EELRU to traces without extensive memory requirements. Even though these traces are not interesting *per se* for a paging study, they help demonstrate that our approach is stable and handles a wide range of available memories. Additionally, these traces confirm that the patterns

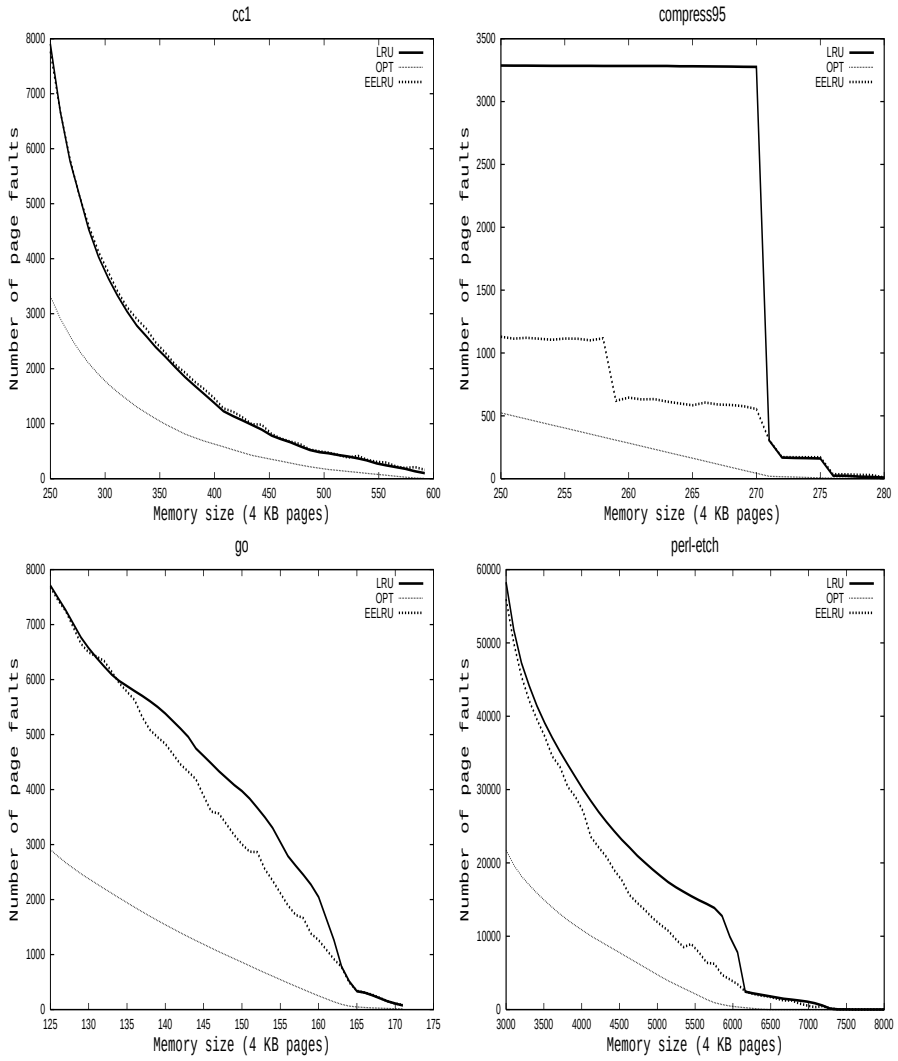


Figure 12: Fault plots for Etch traces. The perl trace is for a different execution than that of Figure 11, but exhibits similar behavior.

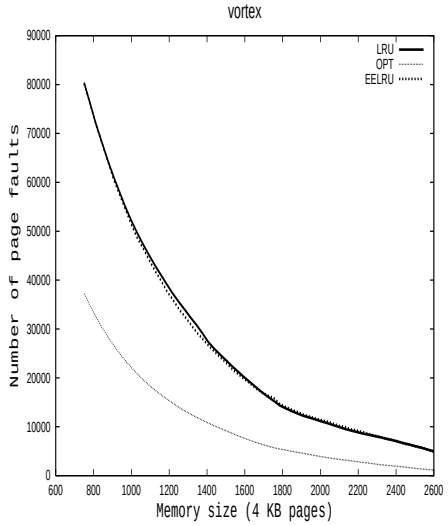


Figure 13: Fault plot for the vortex Etch trace.

that EELRU recognizes are not unique to programs written with paging in mind. Also, being able to adapt both to large-scale and to small-scale patterns is useful for any algorithm to be employed as a replacement algorithm in a multi-process environment. (A short discussion on the potential of EELRU as a replacement algorithm in time-sharing systems can be found in Section 5.4.)

Figures 14 and 15 show the results of applying EELRU to these traces. Because the traces are small and have no distinctive patterns (e.g., see the gcc recency-reference plot in Figure 8), we would expect EELRU to behave similarly to LRU. This confirms the robustness of the algorithm—EELRU is unlikely to perform worse than LRU if no regular patterns exist. As can be seen in the fault plots, this is indeed the case. Note, however, that even for some of these traces EELRU manages to get a small benefit compared to LRU (around 10% less faulting on average). In particular, EELRU seems to be capable of detecting and exploiting even very small-scale patterns. An examination of the gcc recency-reference plot (Figure 8) is quite interesting. We see that there are two small regions where high-intensity (dark) areas are directly above low-intensity (light) areas. EELRU is exploiting exactly these small-scale patterns, and exhibits most of its benefit for a memory size around 150—the boundary of the dark and light areas in the plot.

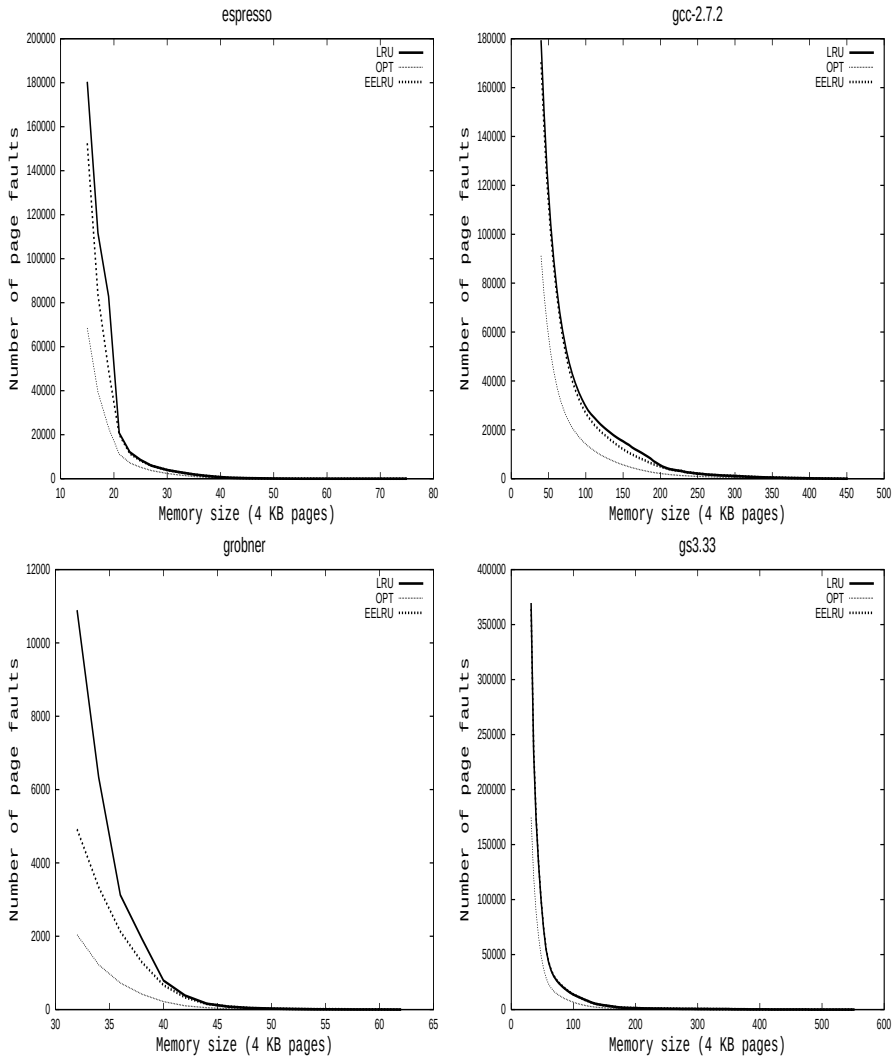


Figure 14: Fault plots for small-scale applications. For ghostscript (gs3.33), the EELRU line overlaps the LRU line.

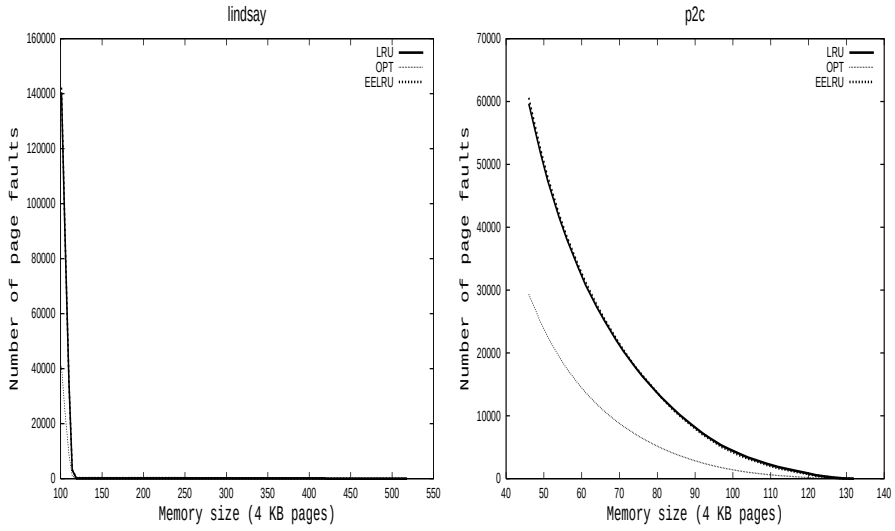


Figure 15: Fault plots for more small-scale applications. For lindsay, and p2c the EELRU line overlaps the LRU line.

## 5.4 EELRU in Actual Systems

There has been a long tradition of studying replacement algorithms in isolation, using program traces and analytical evaluations. Nevertheless, for a replacement algorithm to be deployed in practice, several other factors are important. Some important questions concern the efficiency of the algorithm, the implementation effort required, its performance in multiprocess workloads, etc.

With respect to such concerns, EELRU is well suited for a practical implementation. The algorithm does maintain more data than LRU, but the recency data structures (e.g., a linked list) are overall quite small. The bookkeeping performed is trivial, although somewhat tedious. On every memory reference to a not-too-recently-accessed page (i.e., above the first early eviction point) the appropriate recency region needs to be obtained and its hit number needs to be incremented. Of course, EELRU is not 100% realizable, as it suffers from the same performance drawback as LRU: in the ideal case, statistics need to be update for every single memory reference. Nevertheless, all standard in-kernel LRU approximations (e.g., segmented FIFO [TuLe81, BaFe83]) can be used with EELRU. These approximations have been shown to preserve the behavior characteristics of LRU with minimal per-reference overhead. A particularly promising approximation consists of a segmented queue where the first segment is maintained by a "clock algorithm", assisted by hardware-managed reference

bits for each page. The reference bits can provide coarse-grained information about the hits that preceed the earliest eviction point, allowing the consideration of an even earlier eviction point.

Additionally, there are several possibilities for applying EELRU (or a suitably adapted, recency-based variant) to multi-processing systems. First, EELRU itself could be useful as a "global" algorithm (i.e., managing all pages the same regardless of the process they belong to). Second, recency information of the kind maintained by EELRU could also help memory partitioning. That is, if a process incurs a lot of faults for recently evicted pages, a replacement algorithm could allocate more memory to that process, at the expense of a process for which a smaller memory space would not cause many faults.

# 6 Conclusions

Replacement algorithms are valuable components of operating system design and can affect system performance significantly. In this article we presented EELRU: an adaptive variant of LRU that uses recency information for pages *not in memory* to make replacement decisions. We believe that EELRU is a valuable replacement algorithm. It is simple, soundly motivated, intuitively appealing, and general. EELRU addresses the most common LRU failure modes for small memories, while remaining robust: its performance can never be worse than that of LRU by more than a small constant factor. Simulation results confirm our belief in the value of the algorithm. Additionally, EELRU demonstrates the virtues of a recency-based approach to replacement and advances the concept of timescale relativity for adaptive algorithms. We believe that both of these elements will be important for future research work in program locality.

# References


[ADU71] A.V. Aho, P.J. Denning, and J.D. Ullman, "Principles of Optimal Page Replacement", in *JACM* 18 pp.80-93 (1971).

[BaFe83] O. Babaoglu and D. Ferrari, "Two-Level Replacement Decisions in Paging Stores", *IEEE Transactions on Computers*, 32(12) (1983).

[BFH68] M.H.J. Baylis, D.G. Fletcher, and D.J. Howarth, "Paging Studies Made on the I.C.T. ATLAS Computer", *Information Processing 1968, IFIP Congress Booklet D* (1968).

[CoVa76] P.J. Courtois and H. Vantilborgh, "A Decomposable Model of Program Paging Behavior", *Acta Informatica*, 6 pp.251-275 (1976)

[Den80] P.J. Denning, "Working Sets Past and Present", *IEEE Transactions on Software Engineering*, SE-6(1) pp.64-84 (1980).

[FeLW78] E.B. Fernandez, T. Lang, and C. Wood, "Effect of Replacement Algorithms on a Paged Buffer Database System", *IBM Journal of Research and Development*, 22(2) pp.185-196 (1978).

[FrGu74] M.A. Franklin and R.K. Gupta, "Computation of pf Probabilities from Program Transition Diagrams", *CACM* 17 pp.186-191 (1974).

[GlCa97] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior", Proc. *SIGMETRICS '97.*

[KPR92] A.R. Karlin, S.J. Phillips, and P. Raghavan, "Markov Paging", in Proc. *IEEE Symposium on the Foundations of Computer Science (FOCS)* pp.208-217 (1992).

[LCBAB98] D.C. Lee, P.J. Crowley, J.L. Baer, T.E. Anderson, and B.N. Bershad, "Execution Characteristics of Desktop Applications on Windows NT", Proc. *Int. Symp. Comp. Arch. (ISCA) '98).*

[MGST70] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation Techniques for Storage Hierarchies", *IBM Systems Journal* 9 pp.78-117 (1970).

[MuNe80] H. Muramatsu, and H. Negishi, "Page Replacement Algorithm for Large-array Manipulation", *Software—Practice and Experience* 10 pp.575-587 (1980).

[Pha95] V. Phalke, *Modeling and Managing Program References in a Memory Hierarchy*, Ph.D. Dissertation, Rutgers University (1995).

[SlTa85] D.D. Sleator and R.E. Tarjan, "Amortized Efficiency of List Update and Paging Rules", *Communications of the ACM* 28(2), pp.202-208 (1985).

[SKW99] Y. Smaragdakis, S.F. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement", In Proc. *SIGMETRICS 1999.*

[Spi76] J.R. Spirn, "Distance String Models for Program Behavior", *Computer*, 9 pp.14-20 (1976).

[Tor98] E. Torng, "A Unified Analysis of Paging and Caching", *Algorithmica* 20, pp.175-200 (1998).

[TuLe81] R. Turner and H. Levy, "Segmented FIFO Page Replacement", In Proc. *SIGMETRICS 1981.*

[WKM94] P.R. Wilson, S. Kakkad, and S.S. Mukherjee, "Anomalies and Adaptation in the Analysis and Development of Prefetching Policies", *Journal of Systems and Software* 27(2):147-153, November 1994. Technical communication.

[WoFL83] C. Wood, E.B. Fernandez, and T. Lang, "Minimization of Demand Paging for the LRU Stack Model of Program Behavior", *Information Processing Letters*, 16 pp.99-104 (1983).