EELRU: Simple and Effective Adaptive Page Replacement

Yannis Smaragdakis, Scott Kaplan, and Paul Wilson*
Department of Computer Sciences
University of Texas at Austin
{smaragd, sfkaplan, wilson}@cs.utexas.edu

Abstract

Despite the many replacement algorithms proposed throughout the years, approximations of Least Recently Used (LRU) replacement are predominant in actual virtual memory management systems because of their simplicity and efficiency. LRU, however, exhibits well-known performance problems for regular access patterns over more pages than the main memory can hold (e.g., large loops). In this paper we present *Early Eviction LRU (EELRU)*. EELRU is a simple adaptive replacement algorithm, which uses only the kind of information needed by LRU—how recently each page has been touched relative to the others. It exploits this information more effectively than LRU, using a simple on-line cost/benefit analysis to guide its replacement decisions. In the very common situations where LRU is good, EELRU is good because it behaves like LRU. In common worst cases for LRU, EELRU is significantly better, and in fact close to optimal as it opts to sacrifice some pages to allow others to stay in memory longer. Overall, in its worst case, EELRU cannot be more than a constant factor worse than LRU, while LRU can be worse than EELRU by a factor almost equal to the number of pages in memory.

In simulation experiments with a variety of programs and wide ranges of memory sizes, we show that EELRU does in fact outperform LRU, typically reducing misses by ten to thirty percent, 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.

Overall, EELRU demonstrates several principles which could be widely useful for adaptive page replacement algorithms: (1) it adapts to changes in program behavior, distinguishing important behavior characteristics for each workload. In particular, EELRU is not affected by high-frequency behavior (e.g., loops much smaller than the memory size) as such behavior may obscure important large-scale regularities; (2) EELRU chooses pages to evict in a way that respects both the memory size and the aggregate memory-referencing behavior of the program; (3) depending on the aggregate memory-referencing behavior, EELRU can be "fair" (like LRU) or "unfair", selectively allowing some pages to stay in memory longer while others are evicted.

*This research was supported by IBM, Novell, and the National Science Foundation (under Research Initiation Award CCR-9410026).

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 main memories ranging from 32Mbytes to 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 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 exhibits a well known failure mode, which often affects adversely its performance for small memories. In particular, the performance of LRU suffers for regular access patterns larger than the size of main memory. Such patterns are quite common in programs. 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*.

One way to look at this phenomenon is 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 performs LRU replacement by default but diverges from LRU and evicts pages early when it notices that too many pages are being touched in a roughly cyclic pattern that is larger than main memory. Such patterns can be reliably detected using *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* non-resident pages. In particular, EELRU detects 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 constant factor higher than that of LRU (with the exact value depending on the parameters picked for the algorithm). The reason for this bound is that EELRU deviates from LRU only when the latter incurs many faults and 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. The notion of "recent", and time, in general, is defined in EELRU with respect to the number of "relevant events" that occur. Relevant events are only references to pages that are *not* among the most recently touched. 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.

Also according to timescale relativity, time in EELRU advances at a rate inversely proportional to the memory size. That is, for larger memories, a proportionally larger number of relevant events have to occur for the algorithm to decide to adapt its behavior. The reason behind this choice is that the timescale most relevant to caching decisions is the one at which eviction decisions turn out to be good or bad, i.e., roughly the time a page is likely to spend in memory, and either be touched or not before being evicted. Time in this sense depends on the rate at which a program touches a number of distinct pages comparable to the memory size, which may force evictions. If a program touches few pages (relative to the memory size), little or no time passes. If it touches many pages, time passes rapidly. At any given eviction, the replacement policy should attempt to choose a page which will not be touched for a long "time", i.e., until after "many" other pages are touched—more pages than will fit in memory. This timescale is also crucial for adaptation because an online adaptive replacement policy must detect program behaviors that last long enough to matter to its choice of caching strategies.

In this paper we argue that timescale relativity represents a sound principle upon which locality studies should be based. This includes not only the analysis of replacement algorithms but also the overall evaluation of program locality. 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 4).

To validate EELRU experimentally, we applied it to fourteen program traces and studied its performance. Most of the traces (eight) are of memory-intensive applications and come from the recent 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 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. An extra six 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 4), 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.

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.

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 4 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 the well-known Atlas loop detector [BFH68] that 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 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]. We believe that replacement algorithms based on Markov models fail in practice because they try to solve a far harder problem than that at hand. A replacement algorithm is a program using past data to answer a simple question: "which memory page will be first referenced farthest into the future?" Markov models express the probability of occurrence for specific sequences of references. Most of the pages referenced by real programs, however, are re-referenced very soon and often. The number and order of re-references is not relevant for replacement decisions. In our view, 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 the intrinsic working set size of an access sequence increases beyond [the memory size].

Even though this assertion holds under theoretical worst-case analysis, in practice program reference sequences exhibit 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 gracefully degrade 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 paper 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

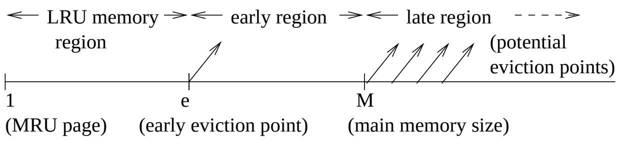

Figure 1: General EELRU scheme: LRU axis and correspondence to memory locations.

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 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]). References to the most recently used pages do not matter for EELRU statistics and incur no overhead. 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 modifica-

[1] The WFL algorithm is called GLRU (for "generalized LRU") in

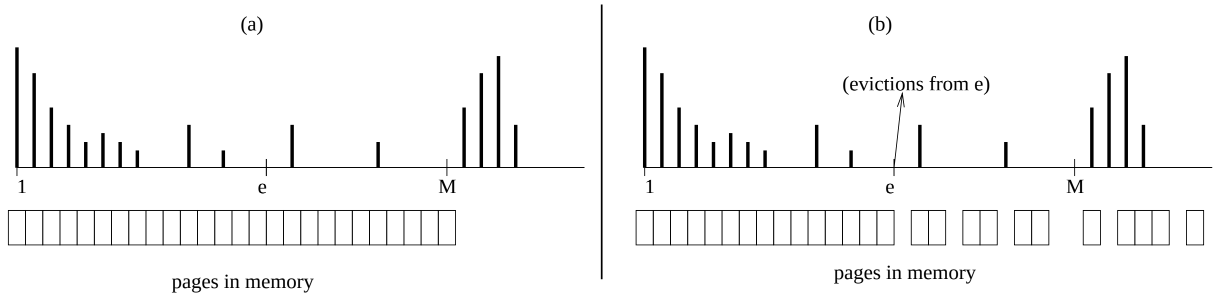

Figure 2: Example recency distribution of page touches: with LRU (left-hand side) many references are to pages not in memory. After evicting early (right-hand side) some less recent pages stay in memory. Since references to less recent pages are common (in this example distribution), evicting early yields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

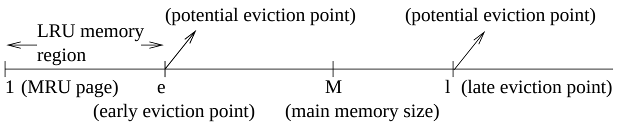

Figure 3: EELRU with WFL fallback: LRU axis and correspondence to memory locations.

tion 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:

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

[FeLW78]. To avoid confusion, we will not use this acronym, since it has been subsequently overloaded (e.g., to mean "global" LRU).

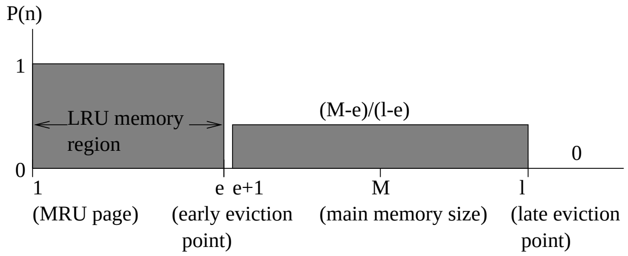

Figure 4: Probability of being in memory for a page with a given recency.

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:

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 on-line 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* ≤ 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 paper.

3.3 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 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 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.

Although we will not offer a rigorous proof of this claim, the argument is straightforward. 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 an illustrative example of this worst-case analysis, consider the extreme case where EELRU has a single early eviction point at position 1 of the recency axis (i.e., it may evict early the most recently used page) and a corresponding late eviction point at position $M + 1$ (with M being the memory size). In this case, a steady loop over $M + 1$ pages will cause LRU to suffer $M + 1$ faults, while EELRU will only suffer a single fault per iteration. At the same time, EELRU will never underperform LRU by a factor of more than 2 under *any* reference pattern. If it decides to diverge from LRU, it may incur at most 2 faults for each LRU fault that causes the divergence.

4 Experimental Assessment

4.1 Settings and Methodology

To assess the performance of EELRU, we used fourteen program traces, covering a wide range of memory access characteristics. Eight of the traces are of memory-intensive applications and were used in the recent experiments by Glass and Cao [GlCa97]. Another six traces were collected individually from programs that do not exhibit large memory reference patterns.

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. 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).

To reproduce as many experiments as possible, we picked early eviction points such that at least 40% of the memory was managed using strict LRU. This makes our simulations quite conservative: it means that EELRU cannot perform very early non-LRU evictions. As mentioned earlier, simulations for eight of the traces are meaningful for this choice of points [2] (i.e., the simulated memory ranges overlap significantly with those of the experiments of Glass and Cao). The table of Figure 5 contains information on these traces. It is worth noting that the above 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). 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 (a C compiler), ghostscript (a PostScript engine), grobner (a formula-rewrite program), lindsay (a communications simulator for a hypercube computer), and p2c (a Pascal to C translator).

[2] Two more traces from [GlCa97], "es" and "fgm", satisfy the restrictions outlined above 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]

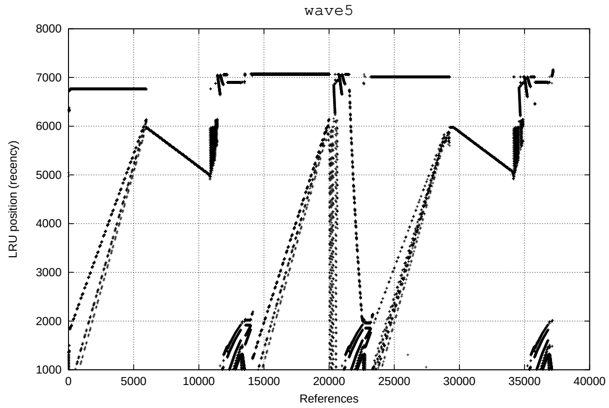

Figure 6: Recency-reference graph for wave5.

All of the simulations were performed using twelve combinations (pairs) of early and late eviction points. Three early points were used, at 40%, 60%, and 80% of the memory size (that is, at least 40% of the memory was handled by pure LRU). The late points were chosen so that the probability $P(n)$ for $e < n <= l$ took the values 2/3, 1/2, 1/3, and 1/4. 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.

4.2 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.

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.

Figure 7 presents 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 exhibits 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. For all plots, features most commonly last for at least as many references as their "size" in pages.
- The gnuplot graph exhibits strong and large loops 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!
- 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 the same number of references, indicating that it is just a loop with two iterations. The "puff-of-smoke"

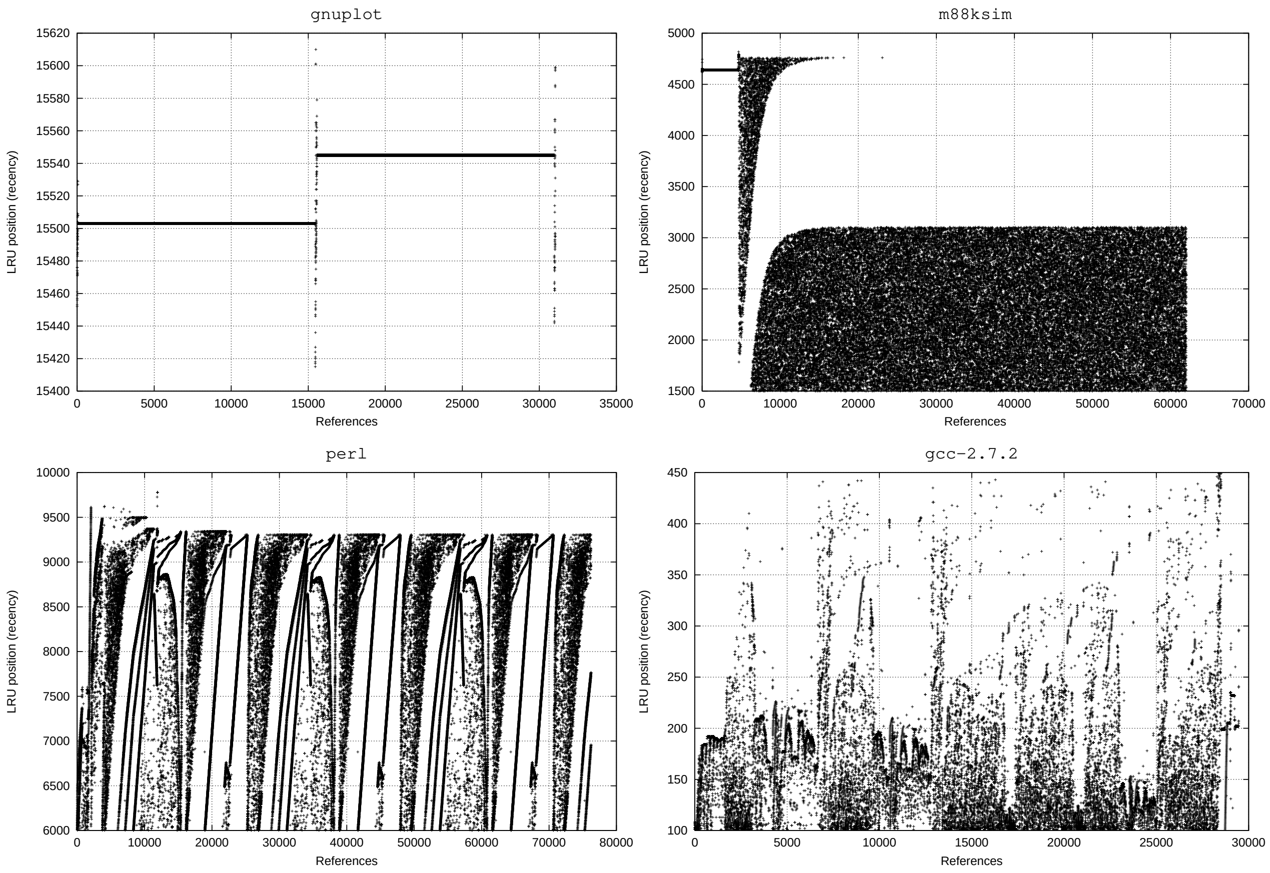

Figure 7: Recency-reference graphs for representative traces.

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 featureless 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).

Other recency-reference graphs are similar to the ones shown. All six "small-scale" traces (espresso, gcc, grobner, ghostscript, lindsay, and p2c) behave like gcc, having no distinctive patterns. Murphi displays random references, much like m88ksim. The rest of the traces have regular patterns. In the case of trygtsl, these are very clear, linear patterns (like gnuplot). In the cases of ijpeg and applu, the patterns look more like those of perl and wave5.

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.

4.3 First Experiment: Memory-intensive Applications

The first eight plots of Figure 8 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]

[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].

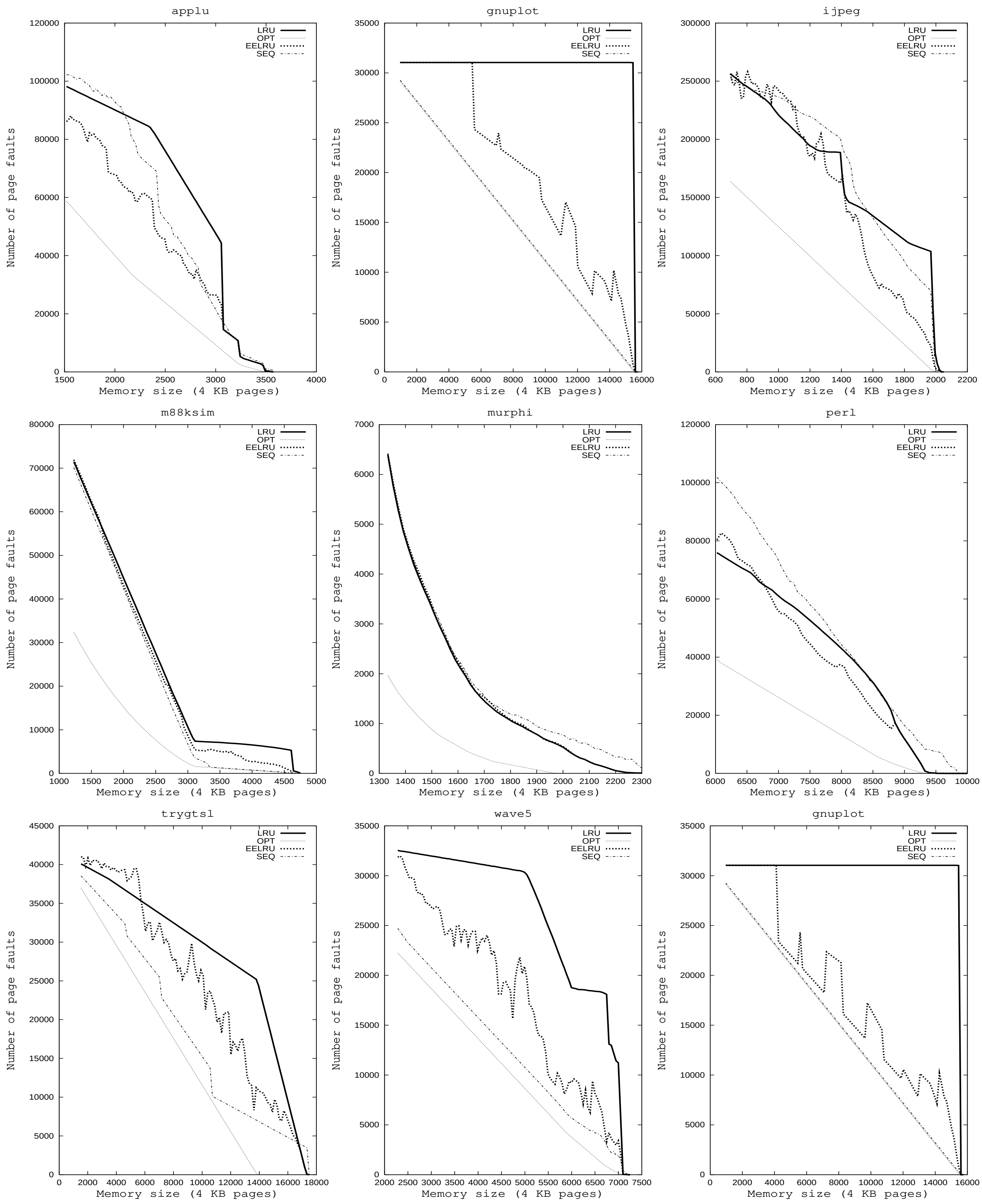

Figure 8: Fault plots for memory-intensive applications. For gnuplot, the SEQ curve almost overlaps the OPT curve. For murphi, the EELRU curve almost overlaps the LRU curve. The last plot (bottom right) is that of gnuplot with an extra, very early eviction point.

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 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).

Note also that, as mentioned in Section 4.1, the simulation of EELRU on these traces was conservative—40% of the memory buffer had to be handled using plain LRU. This accounts for some loss of effectiveness, especially for traces with clear access patterns. We can confirm this for gnuplot: the trace has enough information (see earlier table) to enable meaningful simulations for early eviction points corresponding to 10% of the memory size (i.e., only 10% of the buffer is strictly LRU). We added such an extra early eviction point and the result is shown in the last (bottom right) plot of Figure 8. As can be seen, the performance of EELRU in this case is much better, approaching that of SEQ and OPT. This is hardly unexpected. The gnuplot trace exhibits very regular behavior (in fact, it is dominated by a big loop). Hence, for memory sizes smaller than the size of the loop, allowing EELRU to evict many pages early, so that more other pages can remain longer in memory, is beneficial.

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 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. EELRU takes risks and tries to predict future program behavior based on past patterns. As an example, the decision of diverging from LRU depends on the memory size. For a larger memory size, EELRU may decide that it is worth evicting some pages early in order to capture a loop which is significantly larger than the memory size. If the loop does not persist for long enough, EELRU will incur more faults for this larger size than it would for a smaller memory size, for which the algorithm would perform simple LRU replacement. As another example of the risks taken by EELRU, recency patterns are often not steady (recall the diagonal lines in the plots of Section 4.2). Thus, the right eviction point may not be immediately evident when EELRU starts diverging from LRU. This error in the EELRU estimate will have different results for different memory sizes, depending on the positions of late eviction points relative to the shape of recency patterns.

4.4 Second Experiment: Small-scale Patterns

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 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 6.

Because these traces are small and have no distinctive patterns (e.g., see the gcc recency-reference plot in Figure 7), 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 of Figure 9, 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 7) 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.

5 Thoughts on Program Behavior and Replacement Algorithms

LRU serves as a good reference point for understanding the performance of other replacement algorithms. We have a rough taxonomy of program behavior into four classes: (1) *LRU-friendly*: In very good cases for LRU, any excellent replacement policy must resemble LRU, by and large preferring to keep recently-touched pages in memory. Programs that seldom touch more pages than will fit in memory, over fairly long periods, are LRU-friendly. If this regularity is sufficiently strong, LRU is nearly optimal, and many replacement policies (e.g., Random or FIFO) will also do well. This common case exhibits LRU's greatest strength. (2) *LRU-*

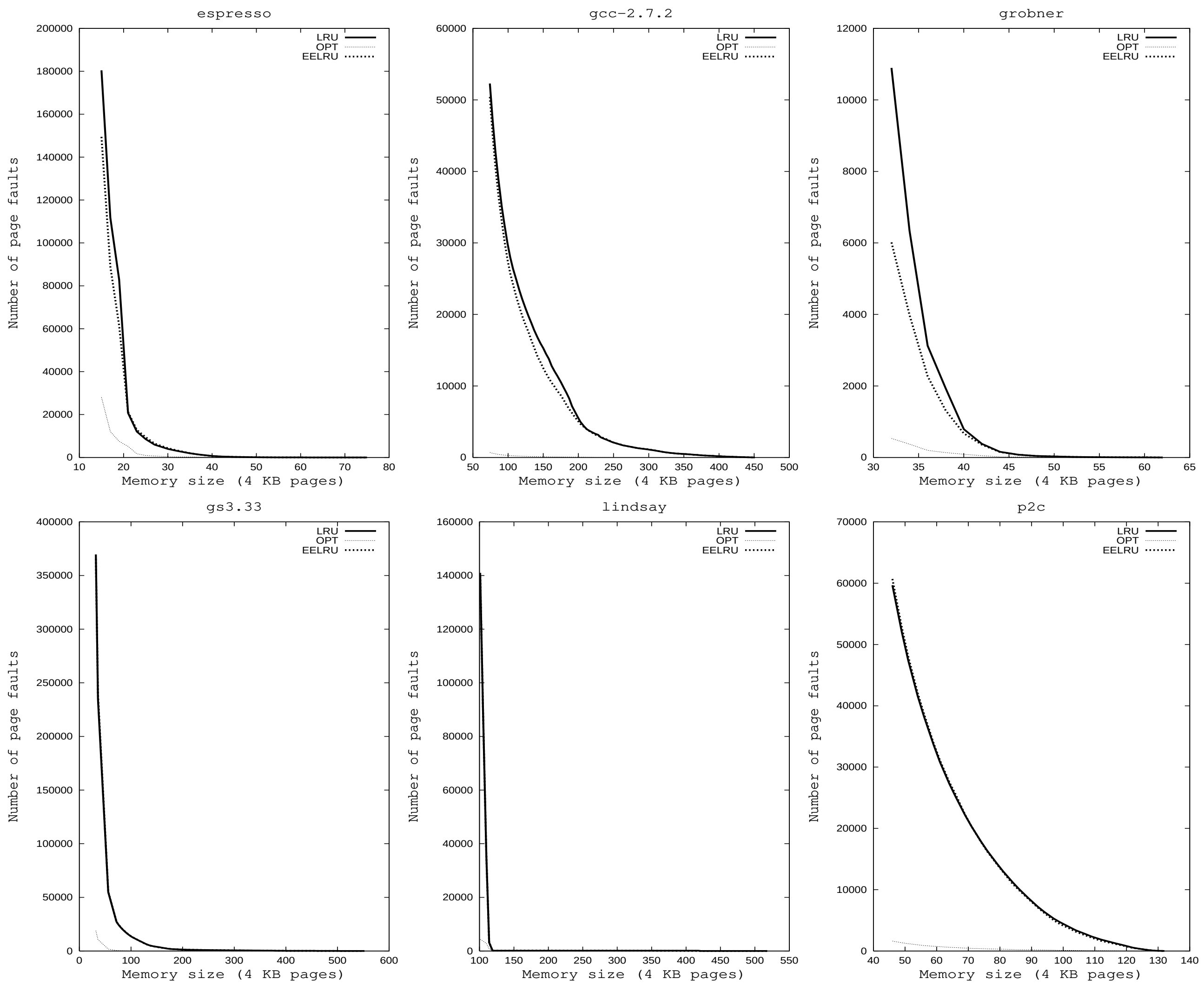

Figure 9: Fault plots for small-scale applications. For ghostscript (gs3.33), lindsay, and p2c the EELRU line overlaps the LRU line.

unfriendly: In very bad cases for LRU, where LRU chronically evicts pages shortly before they are touched again, any excellent replacement policy must evict some pages early to keep others in late, rather like MRU. If this regularity is sufficiently strong, MRU-like behavior will be nearly optimal. This relatively common case exhibits LRU's greatest weakness; addressing it properly is the main contribution of this paper, and we demonstrate that this is sufficient to substantially improve on LRU. (3) *unfriendly*: Programs which touch many pages that have not been touched for a very long time—i.e., pages that would have been evicted long ago by LRU—are unfriendly to *any* replacement policy, even an optimal one. LRU does "well" for them by default, because it is comparable to optimal in this case; our algorithm does the same. (4) *mixed*: This catch-all category covers programs that often touch pages that would soon be evicted by LRU *and* pages that LRU would have recently evicted. That is, the aggregate recency distribution is not a particularly good indicator, by itself, of what a replacement policy should do. This is an area for future work.

Notice that, in this last case, a nearly optimal replacement policy must evict some pages in preference to others, without simply being fair like LRU, or simply being unfair like MRU. It must make finer distinctions, recognizing more subtle aspects of program behavior than we deal with in this paper. Conversely, notice that in the other three cases, such fine distinctions are unnecessary for good performance, and may even be harmful if not made very well—if any of the three regularities is strong enough, there is little benefit to be gained by attempting to make fine distinctions, because the crucial information is primarily in the *aggregate* behavior of large numbers of pages, *not* in the details of particular pages' reference histories.

6 Conclusions and Future Work

Replacement algorithms are valuable components of operating system design and can affect system performance significantly. In this paper 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 constant factor. Simulation results seem to confirm our belief in the value of the algorithm. The main axes of our future experimentation will examine EELRU as a replacement algorithm for time-sharing systems and the cost and performance of an in-kernel approximation of EELRU.

Interestingly enough, there are several possibilities for a recency-based approach to a replacement algorithm in multiprocessing systems. For one thing, EELRU itself could be useful as a "global" algorithm (i.e., managing all pages the same regardless of the process they belong to). For another, 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. These ideas open up possibilities for quite sophisticated recency-based replacement algorithms. EELRU has already served to demonstrate some sound principles, like timescale-relative adaptivity, on which these algorithms should be based.

Acknowledgments

We would like to thank the anonymous referees for their detailed comments and advice on improving the paper.

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).

[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).

[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).