EELRU: Simple and Effective Adaptive Page Replacement

Yannis Smaragdakis, Scott Kaplan, and Paul Wilson
Deparment 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 of size larger than the main memory. In this paper we present *Early Eviction LRU (EELRU)*: an adaptive replacement algorithm based on the principle of detecting when the LRU algorithm underperforms (i.e., when the fetched memory pages are often the ones evicted lately). In simulations, EELRU proves to be quite effective for many memory sizes and several applications, often decreasing paging by over 30% for programs with large-scale reference patterns and by over 10% for programs with small-scale patterns. Additionally, the algorithm is very robust, rarely underperforming LRU. Our experiments are mostly with traces from the recent research literature to allow for easy comparison with previous results.

1 Introduction and Overview

Despite the lower prices and higher capacities of RAM, virtual memory replacement algorithms remain as relevant as ever. For one thing, the performance gap between memory and disks has increased—secondary storage speed is among the slowest growing parameters of modern computer systems. For another, the ubiquity of virtual memory has changed the way programs are written. More and more programs assume a plentiful and well-managed virtual memory. Representative workloads have changed significantly: old-style programs (e.g., TeX, which explicitly stages computation and produces intermediate results in files) are less common, while new-style programs (e.g., `javac`, which often requires 64Mbytes of memory for large inputs) become part of everyday use. To make matters worse, modern 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 operating system designers to improve virtual memory policies to obtain the best attainable performance, regardless of system configuration.

Getting good performance for all different applications and configurations is hard. Currently most operating systems use replacement algorithms that are approximations of Least Recently Used (LRU) replacement (e.g., segmented FIFO implementations [TuLe81, BaFe83]). Although very good in most cases, the performance of LRU-based algorithms suffers for regular access patterns larger than the size of main memory. Such patterns are quite common in programs. For instance, a linear loop "touching" more memory pages than the memory capacity incurs faults for every page: the page to be touched next is not among the most recently touched ones and, hence, has been evicted from memory. One way to look at this phenomenon is that LRU keeps pages in memory for long, but cannot keep all of them for long enough. Evicting some of the pages *early* allows other pages that will be used soon to remain in memory. This is the intuition behind Early Eviction LRU (EELRU). EELRU performs LRU replacement by default and diverges from LRU only when recent paging behavior indicates that LRU consistently underperforms.

EELRU is based on a novel combination of two ideas. The first is that replacement decisions should be made based on extensive **recency** information (i.e., information indicating how many other pages were touched since a page was last touched). EELRU uses recency information, even for pages not in memory, to detect large memory reference patterns. In particular, EELRU detects that LRU underperforms when *many of the fetched pages had just been evicted.* The second idea is that any adaptive algorithm should update its state information in a **timescale relative** way. 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 replacement policy. A loop over 600KB 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. Recency (and time, in general) is defined in EELRU as 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.

We argue that these two ideas represent sound principles upon which program 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 under this light (Section 2). Also, we propose that a special kind of plots, called *recency-reference* graphs are appropriate for studying program locality behavior (Section 4).

In this study we applied EELRU to fourteen program traces and examined 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 (in address terms) access patterns. 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 the first 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. 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 relevant references.

The two principles behind EELRU—recency and timescale relativity—offer distinct benefits for replacement studies. For instance, a recency-based standpoint (see also [Spi76, FeLW78, WoFL83]) 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.

Timescale relativity helps EELRU detect (and adapt to) phase changes. 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 (loops may perform different numbers of operations per step during different iterations—as is, for instance, the case with many nested loop patterns), the time between successive references will vary a lot. 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 online 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, FrGu30]). 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 furthest into the future?" Markov models express the probability of occurence 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 other words, the model tries to predict program behavior at the wrong (much more detailed) timescale. This makes Markov model-based replacement too brittle for actual use—the model cannot offer any accuracy at a large enough timescale (such as that of memory replacement decisions).

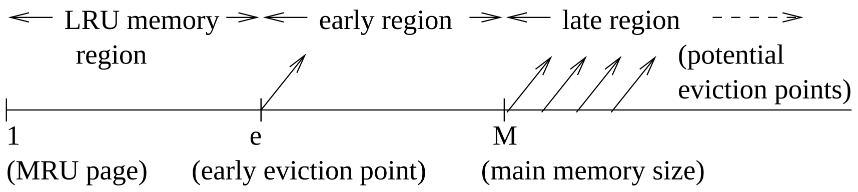

Figure 1: General EELRU scheme: LRU axis and correspondence to memory locations.

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. We have, however, experimented with several (both deterministic and probabilistic) fallback algorithms. 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. That is, the algorithm makes the assumption that the program recency behavior will remain the same for the near

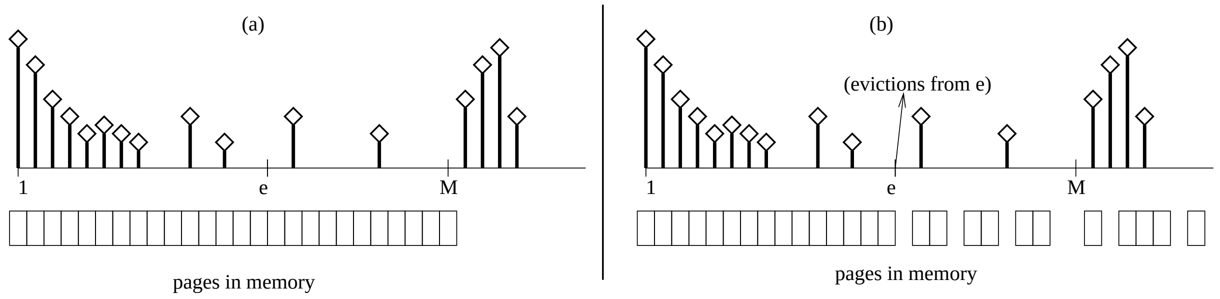

Figure 2: Example recency distribution: evicting early yields benefits.

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 occured *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 (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 (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 online 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

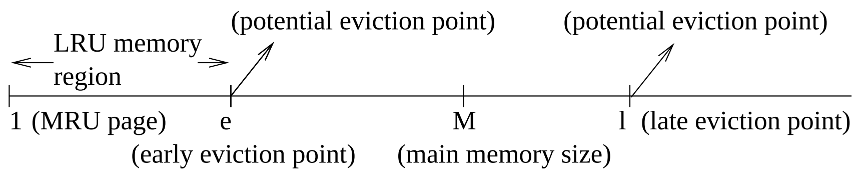

Figure 3: EELRU with WFL fallback: LRU axis and correspondence to memory locations.

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 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:

```
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 online 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).

[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).

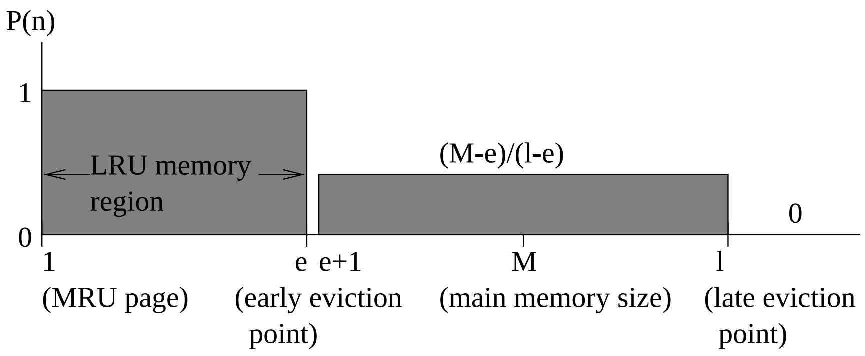

Figure 4: Probability of being in memory for a page with a given recency.

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:

```
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 paper.

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 us worth noting that the above set of traces contains representatives

[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]

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).

All of the simulations were performed using twelve combinations (pairs) of early and late eviction points. Three early points were used, at 20%, 40%, and 60% of the memory size (that is, the early region was at most 60% of the memory, with the other 40% 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 one third of the weight of the most recent one.

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 below. 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