**The Computer Journal Advance Access published October 14, 2005**

doi:10.1093/comjnl/bxh140

# Caching and prefetching algorithms for programs with looping reference patterns

GIANLUCA DINI, GIUSEPPE LETTIERI AND LANFRANCO LOPRIORE

*Dipartimento di Ingegneria della Informazione: Elettronica, Informatica, Telecomunicazioni, Università degli Studi di Pisa, via Diotisalvi 2, 56126 Pisa, Italy*
*Email: {g.dini, g.lettieri, l.lopriore}@iet.unipi.it*

**We present a thorough analysis of the memory behaviour of page caching and prefetching algorithms. The analysis is restricted to programs whose execution consists of iteration of a sequence of page accesses. Program activity is characterized in terms of utilization of system resources. A graphical model of program execution is used to describe both page placement in the primary memory and the actions of page fetch and replacement. The algorithms are compared from the point of view of a number of performance indexes that include program response time and utilization of the secondary memory system. Special attention is paid to transient program behaviour and the effects of the time necessary for the processor to control the disk activities of page fetch. The results of a large set of measurement experiments are used to validate the analytical model and acquire significant indications concerning the extent of the simplifying assumptions made in the theoretical analysis. The discussion of the relation to previous work makes special reference to two classes of algorithms that received much attention in the past, aggressive prefetching and informed prefetching.**

*Received 22 October 2004; revised 14 July 2005*

## 1. INTRODUCTION

In a demand-paged virtual memory system, when the running program references a page that is not available in the primary memory, a page fault exception is generated. The program enters the *stall state* and control is transferred to the memory management system, which fetches the missing page from disk to primary memory. Actual transfer of the page contents is performed by a complex of hardware/software resources that is, typically, several orders of magnitute slower than the processor and that we shall call the *secondary memory system* (or *sms*, for short). We shall use the term *block* to indicate the logical unit of secondary memory space and the term *frame* to indicate the logical unit of primary memory space. Blocks and frames have the same size, equal to the size of a page.

In a *global* strategy of storage allocation, all programs share a single pool of frames, whereas in a *local* strategy each program is associated with a specific primary memory area that we shall call the program *buffer*. We shall restrict our analysis to local strategies whereby the free frame necessary on the occurrence of a page fault to load the missing page is taken from the buffer of the program generating the fault. If no free frame is available in the buffer, a *victim page* is selected for replacement, by using a page replacement algorithm that is part of the memory management system. The victim page is evicted from the buffer and the buffer frame made free in this way is used for the missing page. Finally, execution of the stalling process is resumed.

A ubiquitous example of a page replacement algorithm is the *least-recently-used* (*LRU*) algorithm, which replaces the page whose previous reference is farthest in the past. LRU produces good performance in a variety of contexts. Approximations of this algorithm are largely used in time-sharing operating systems, and variants have been proposed to improve performance, in specific application environments, for instance [1, 2, 3, 4, 5, 6]. On the other hand, consider a program whose execution consists of iteration of a sequence of accesses to the pages of a memory area larger than the buffer. *Looping reference patterns* [7, 8, 9, 10, 11] of this type are very common in practice. They are representative of the memory behaviour of large classes of application programs. Examples are scientific applications and libraries [12], applications involving simulation based modelling [13], image processing, graph generation, integer intensive programs [14] and run-time supports (e.g. a garbage collector that reuses memory [4]). In the presence of a looping reference pattern, the page that will be referenced next in a given iteration is one whose previous reference is farthest in the past. This means that LRU evicts pages just before they are referenced again, thereby causing the fetch of all pages at each iteration. As a result, LRU produces the worst possible performance. A better strategy is to evict a

page that has been referenced recently, as the next reference to this page is furthest in the future. An example is the *most-recently-used* (*MRU*) page replacement strategy, which evicts the page whose previous reference is nearest in the past.

Memory caching techniques keeping recently used pages in the buffer are effective in saving a page fetch when one of these pages is referenced again. However, a program referencing a page that is missing in the buffer has to stall in wait for completion of the page fetch from secondary memory into the buffer. This page latency negatively affects program response time. A possible solution is to anticipate the fetch of the pages the program is going to use, so that each of these pages is available in the buffer before being accessed by the program. *Prefetch* techniques of this type have been recognised as the best method beyond caching to reduce storage access times [15]. In sequential file management, a widely used prefetch approach is the read-ahead of the file pages [16]. The main motivation for page read-ahead is the practical consideration that most application programs access files in strict page sequence. On the other hand, the read-ahead strategy has no knowledge of the actual program pattern of memory reference. Being based on spatial locality, this strategy is prone to unsatisfactory performance for programs exhibiting locality on a temporal basis. Consider a program that iteratively accesses the elements of a circular list, for instance. If the elements are not contiguous in memory, data locality is temporal rather than spatial. The read-ahead mechanism will prefetch pages that will not be used, thereby causing useless sms activity while producing no advantage in terms of program response time.

A central issue in any prefetching strategy is the interaction with the activities of page replacement [17, 18]. If the buffer is full, the prefetch of a given page causes selection of a victim page and de-allocation of the corresponding buffer frame. Allocation of a frame for prefetch may reduce disk latency, whereas de-allocation of a frame may cause a new page fetch, if the evicted page is referenced again later. A prefetch started too early may cause replacement of a page that will be used soon, thereby increasing the number of page movements between primary memory and secondary memory. A prefetch initiated too late may cause the program to stall in wait for termination of the load of the missing page. In these situations, prefetch may lead to little advantage or even decrease overall performance.

This paper presents a thorough analysis of the memory behaviour of a number of caching and prefetching algorithms. We shall restrict our study to application programs with looping reference patterns, whose execution consists of the iteration of a sequence of accesses to a set of memory pages. This sequence is called the *base stream*. We shall consider *read-once* base streams where all pages are accessed for read only. An example is a query in a large database. An action of this type causes the read of a possibly large number of pages, and does not modify the page contents. In our analysis, this hypothesis allows us to concentrate our attention on the program pattern of page reference, whereas in the presence of write accesses further factors should be considered. Let us refer to a situation in which a new page must be loaded into the buffer and no free frame is available, for instance. If we select a page for replacement that was accessed for write, the page contents must be copied back to secondary memory before reusing the corresponding buffer frame. This action of page copy alters the normal sequence of page accesses as implied by the base stream. Movements of the disk arm may follow, which modify the time required to fetch the missing page, for instance. Of course, no action of page copy is ever required if all pages are accessed for read only. From the point of view of program performance, we can find large classes of application programs that stall in wait for completion of read operations from disk to primary memory for a significant fraction of their total execution time. Text searches and relational database queries are examples of these read-intensive application programs. Write operations are less critical [19]. An application accessing the disk for write is seldom forced to stall in wait for completion of the write [20].

We shall take two page replacement algorithms into consideration, LRU and MRU. For a program with a looping reference pattern, LRU always produces the worst replacement and the worst program response time, whereas MRU is provably optimal [9]. Together, LRU and MRU limit the range of the possible response times in the absence of page prefetch. Furthermore, we shall consider two page prefetching algorithms, which we call *early prefetch* (*EP*) and *late prefetch* (*LP*). Both these algorithms implement a combination of page caching and prefetching, by replacing the page whose next reference is furthest in the future while prefetching the non-buffered page whose next reference is nearest in the future. They differ on the time instant when a new prefetch is started up, and consequently, on the duration of the time interval between completion of the prefetch of a given page and occurrence of the next program reference to this page. This factor is prone to important effects on program performance [18]. Essentially, EP begins a new prefetch as soon as the sms becomes idle. Therefore, EP implements a form of aggressive prefetch that generates a heavy sms activity. The software overhead related to this activity can have a negative impact on the program execution time. LP produces lighter sms utilization by delaying the prefetch of a missing page until the latest time instant allowing the prefetch to be completed before occurrence of the reference to this page, so that the program does not stall. However, this may cause the prefetch to complete at a time that is too late to start the prefetch of the next missing page, thus causing the program to stall at a later time.

In the rest of this paper:

- We characterize the activities of programs with looping reference patterns in terms of utilization of system resources (Section 2). A graphical model of program execution is introduced to describe both the page placement in the buffer and the dynamic behaviour of the page caching and prefetching algorithm, as far as the activities of page fetch and replacement are concerned.
- We perform an analytical evaluation of the memory behaviour of the two page replacement algorithms,

LRU and MRU (Section 3), as well as of the two prefetching algorithms, EP and LP (Section 4). A comparative analysis of these four algorithms is carried out from the point of view of a number of performance indexes that include program response time and sms utilization (Section 5). Special attention is paid to transient program behaviour and the time necessary for the processor to control the activities of the sms.

- We present the results of a large set of measurement experiments that have been carried out to validate the analytical model (Section 6). These experiments give us significant indications concerning the extent of the simplifying assumptions made in the theoretical analysis.
- Finally, we discuss the relation of our work with previous work (Section 7). We make special reference to two classes of algorithms that received much attention in the recent past, aggressive prefetching [21] and informed prefetching [19]. The results of research concerning these algorithms are closely related with our results. Furthermore, with reference to the analytical evaluation of the latter class of algorithms, we argue that the analysis results provide a support to forms of application-controlled memory management where the application program specifies both the names of the pages that form the base reference stream and the caching and prefetching algorithm that is best suited to support program execution.

## 2. THE REFERENCE STREAM

We shall hypothesize that the sms includes the hardware resources necessary to move a single page at a time, from secondary memory to primary memory. This means that a new fetch starts after termination of the previous fetch. The processor executes a single sequential program, the *target program*, which is entirely resident in primary memory. When the processor is not idle, it either executes the target program or is involved in an activity of sms control, as is required to start up a page fetch operation, for instance. We abstract the behaviour of the target program to the following model:

- The target program produces a *reference stream* consisting of $m$ iterations of a *base stream*. The base stream takes the form of a sequence of $n$ *page references*. A reference to a given page consists of one or more read accesses to the storage cells that form this page. All accesses are for read only. The name (virtual address) of the page involved in the $i$-th page reference of the base stream is denoted by $p_i$. The base stream contains at most one reference to any given page, so we have
$$p_i \neq p_j \quad \forall i, j \in \{0, \ldots, n-1\},\ i \neq j. \qquad (1)$$
- The pages of the base stream are contained in secondary memory and must be loaded into primary memory before being accessed by the processor. A primary memory buffer of size $c$ frames is reserved for the target program. When program execution is started up, the buffer is empty. At each reference to a page that is not in the buffer, the program stalls in wait for completion of the fetch of this page from secondary memory into a free buffer frame. If no free frame is available in the buffer, the page fetch is preceded by the selection of a victim page for replacement. This page is evicted from the buffer and the frame made free in this way is used for storage of the new page. All page references are for read, and consequently, there is no need to copy the victim page back to secondary memory.

The following factors quantify the activity of the target program in terms of utilization of system resources:

- *Page reference time* $t_{\mathrm{pr}}$: period of time (number of time units) necessary for the processor to accomplish a page reference, by carrying out all the memory accesses that form this reference.
- *Sms control time* $t_{\mathrm{sms}}$: period of time necessary for the processor to accomplish the activity of sms control relevant to an operation of page fetch. $t_{\mathrm{sms}}$ includes the computational overhead of allocating a buffer frame, sending the fetch request to the sms and servicing the interrupt request from the sms on completion of the fetch.
- *Page fetch time* $t_{\mathrm{f}}$: period of time elapsed between the beginning of the processor activity of sms control for the fetch of a given page and the actual accessibility of this page to the target program in the primary memory. Note that $t_{\mathrm{sms}}$ is a component of $t_{\mathrm{f}}$.
- *Normalized page fetch time* $\phi = t_{\mathrm{f}}/t_{\mathrm{pr}}$: number of page references (for $t_{\mathrm{f}} \geq t_{\mathrm{pr}}$) or fraction of a page reference (for $t_{\mathrm{f}} < t_{\mathrm{pr}}$) that the program can accomplish in the period of time necessary to fetch a page in the hypothesis of $t_{\mathrm{sms}} = 0$.
- *Normalized sms control time* $\sigma = t_{\mathrm{sms}}/t_{\mathrm{f}}$: fraction of page fetch time necessary for the processor to control an sms activity of page fetch. In other words, $\sigma$ represents the fraction page fetch time that cannot be overlapped with the execution of the target program.
- *Normalized buffer capacity* $\beta = c/n$: fraction of the pages of the base stream that the buffer can contain.

We shall hypothesize that quantities $t_{\mathrm{pr}}$, $t_{\mathrm{sms}}$ and $t_{\mathrm{f}}$ are all independent of the specific page.

Let us define the *minimum execution time* $T_{\mathrm{min}}$ of the target program as the period of time that elapses between the beginning of the earliest page reference in the first iteration of the base stream and the completion of the latest page reference in the last iteration in the hypothesis that the pages of the base stream are all contained in the buffer. In a situation of this type, the program never stalls, and we have
$$T_{\mathrm{min}} = nmt_{\mathrm{pr}}. \qquad (2)$$

Let $x$ denote the generic caching/prefetching algorithm. In our analysis, we shall pay special attention to the following program performance indexes:

- *Response time* $T_x$: total number of time units necessary to execute the target program when algorithm $x$ is used for page caching/prefetching.

- *Normalized response time* $R_x = T_x / T_{\min}$.
- *Sms busy time* $S_x$: total number of time units necessary for the sms to accomplish all the activities of page fetch requested by the target program. Let $f_x$ denote the average number of pages fetched in a single iteration of the base stream when algorithm $x$ is used for page caching/prefetching. By definition we have $S_x = mf_x t_{\mathrm{f}}$
- *Sms utilization* $U_x = S_x / T_x$. We have

$$U_x = \frac{mf_x t_{\mathrm{f}}}{T_x} = \frac{f_x \phi}{R_x n}. \quad (3)$$

### 2.1. A graphical representation

We shall model the processor activities in the buffer and the actions of page fetch and replacement by taking advantage of a graphical representation having the form of a clock. The clock dial is partitioned into $n$ sectors, one sector for each page reference that forms the base stream. Figure 1 shows an example of this graphical representation. The example is relevant to the memory behaviour of the LRU algorithm, which we shall consider in detail shortly.

At any given time, a clock hand, the *reference hand* $\mathcal{R}$, points to the page being referenced by the processor at this time. If the target program is stalling and $\mathcal{R}$ points to a given page, then the program terminate the reference to this page just before entering the stall state. The other clock hand, called the *fetch hand* $\mathcal{F}$, points to the page being fetched from secondary memory into the buffer. If the sms is idling, $\mathcal{F}$ points to the page to be fetched next. The lengths of the two hands indicate the relative speed. Just like in a real clock, the longer hand moves faster. If $t_{\mathrm{f}} > t_{\mathrm{pr}}$, i.e. $\phi > 1$, $\mathcal{R}$ is longer than $\mathcal{F}$, for instance. Two hands of the same length indicate that the relationship between their speeds is inessential. We shall use notations $p_{\mathcal{R}}$ and $p_{\mathcal{F}}$ to indicate the pages pointed by $\mathcal{R}$ and $\mathcal{F}$, respectively. The page preceding $p_{\mathcal{R}}$ is denoted by $p_{\mathcal{R}-1}$.

A box surrounding a given page indicates that at present a buffer frame is reserved for storage of this page. We shall use the term *buffered page* to denote a page of this type. A box surrounding a sequence of several pages indicates a zone of the base stream whose corresponding pages are all contained in the buffer. We shall use the term *buffered zone* to denote a zone of this type. As long as the reference hand moves within the boundaries of a buffered zone, the program does not stall. In a situation of this type, the processor alternates activities of page reference with activities of sms control.

A sequence of one or more pages with no surrounding box indicates a zone of the base stream whose corresponding pages are only contained in secondary memory. Before starting up a reference to one of these pages, the program enters the stall state in wait for the completion of the loading of this page into the buffer. We shall use the term *stall zone* to denote a zone of this type.

As pointed out previously, when a page should be loaded into the buffer and no free frame is available, a page is selected for replacement. Graphically, an action of this type causes the movement of a box of size one page. This movement is indicated by an arrow. A white dot denotes the page being evicted from the buffer, and a black dot denotes the page being fetched from secondary memory into the buffer frame made free in this way.

### 2.2. Assumptions and limitations

For the sake of clarity, we briefly list the assumptions and limitations we introduced in the previous sections:

- the computer system executes a single program;
- the program is entirely resident in primary memory;
- the program execution consists of the iteration of a fixed sequence of accesses to a set of memory pages;
- all page accesses are for read, and, consequently, there is no need to copy a page back to secondary memory;
- the page reference time ($t_{\mathrm{pr}}$), the sms control time ($t_{\mathrm{sms}}$) and the page fetch time ($t_{\mathrm{f}}$) are constant and independent of the specific page;
- the sms moves a single page at a time, from secondary memory to primary memory.

A discussion of these assumptions and limitations will follow during the paper with particular reference to their impact of the analysis results.

## 3. NON-PREFETCHING ALGORITHMS

The two non-prefetching algorithms considered in this paper, LRU and MRU, differ on the page that is selected for replacement when a new page must be fetched from secondary memory into the buffer and no free frame is available in the buffer. LRU is defined by the following rule:

- Least recently used: every page replacement evicts the buffered page whose previous reference is farthest in the past.

MRU is defined by the following rule:

- Most recently used: every page replacement evicts the buffered page whose previous reference is nearest in the past.

In the graphical representation introduced in Section2.1, LRU implies that we evict the farthest page we encounter moving counterclockwise in the buffered zone from the present position of the reference hand. MRU implies that we evict the nearest page we encounter moving counterclockwise in the buffered zone from the present position of the reference hand.

### 3.1. The LRU algorithm

We shall now take advantage of our graphical representation to model the memory behaviour of the target program in the hypothesis that LRU is used for page replacement. At the beginning of program execution, the first $c$ pages of the base stream are loaded into the buffer. In this initial phase, the reference hand points to page $p_{\mathcal{F}-1}$ (Figure 1a). When the buffer becomes full, execution enters a steady state. The fetch of page $p_{c+i}$ causes replacement of page $p_i$, as this is the buffered page that was referenced least recently. Figure 1b shows the memory configuration when the first page replacement takes place, i.e. $i = 0$.

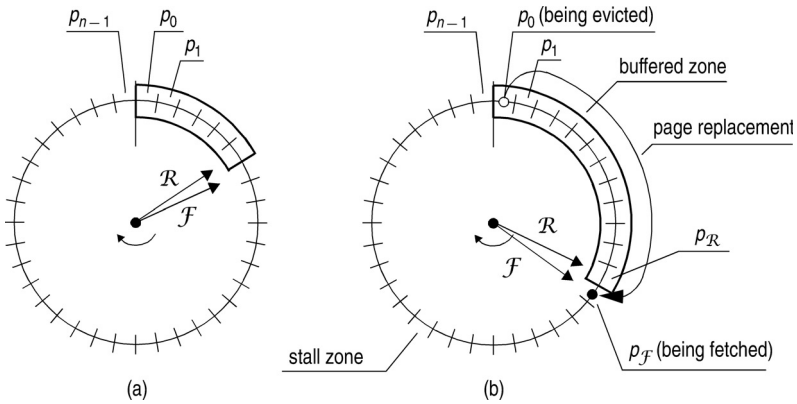


**FIGURE 1.** Memory behaviour of LRU: (a) transient configuration: (b) steady-state configuration.

Thus, in steady state, a buffered zone of size $c$ pages moves clockwise at the speed of the fetch hand. The program stalls on completion of every page reference, and we have

$$f_{\mathrm{LRU}} = n. \tag{4}$$

Each fetch operation adds $t_{\mathrm{f}}$ time units to program response time; thus we have

$$T_{\mathrm{LRU}} = T_{\mathrm{min}} + mf_{\mathrm{LRU}}t_{\mathrm{f}} = T_{\mathrm{min}} + mnt_{\mathrm{f}}. \tag{5}$$

Dividing by $T_{\mathrm{min}}$ and manipulating, we obtain

$$R_{\mathrm{LRU}} = 1 + \phi. \tag{6}$$

### 3.2. The MRU algorithm

The transient behaviour of MRU is similar to that of LRU (see Figure 1a). A steady state is reached when the buffer becomes full, on completion of the reference to page $p_{c-1}$. In the stall zone, the fetch of page $p_i$ causes replacement of page $p_{i-1}$, as this is the buffered page that was referenced most recently (Figure 2). At each iteration of the base stream, the buffered zone moves one position counterclockwise. This is a consequence of the fact that the crossing of the stall zone begins by replacing the buffered page involved in the latest reference and terminates by fetching the page involved in the reference preceding the earliest reference.

Each iteration causes the fetch of $n - c$ pages from secondary memory into the buffer,[1] that is

$$f_{\mathrm{MRU}} = n - c. \tag{7}$$

Each fetch operation adds $t_{\mathrm{f}}$ time units to program response time; thus we have

$$T_{\mathrm{MRU}} = T_{\mathrm{min}} + mf_{\mathrm{MRU}}t_{\mathrm{f}} = T_{\mathrm{min}} + m(n-c)t_{\mathrm{f}} \tag{8}$$

and

$$R_{\mathrm{MRU}} = 1 + (1-\beta)\phi. \tag{9}$$

[1] In fact, some iterations produce the fetch of $n-c+1$ pages. The results of our evaluation, as expressed by forthcoming Relation 9, are still valid if $n \gg 1$.

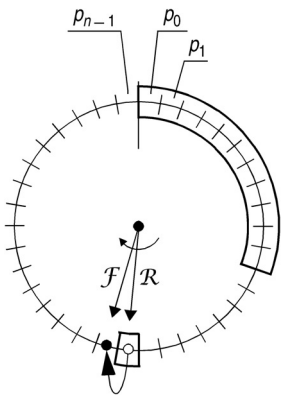


**FIGURE 2.** Memory behaviour of MRU in the stall zone.

## 4. PREFETCHING ALGORITHMS

The two prefetching algorithms considered in this paper, EP and LP, differ on the time when a new prefetch begins in a situation of idle sms. EP is defined by the following rules:

- *Next reference*: every page fetch loads the non-buffered page whose next reference is nearest in the future.
- *Fair replacement*: every page replacement evicts the buffered page whose next reference is furthest in the future, provided this page has been referenced since its most recent load from secondary memory into the buffer.
- *Early prefetch*: let $t_1$ be the time instant when the sms becomes idle. The next page fetch begins at the earliest time instant $t \geq t_1$ that satisfies the fair replacement rule (Figure 3a)

In our graphical representation, the next reference rule implies that, when a fetch operation is being carried out, the fetch hand points to the first non-buffered page we encounter moving clockwise from the present position of the reference hand. The fair replacement rule implies that every page replacement involves the first buffered page we encounter moving counterclockwise from the present position of the reference hand, provided this page has already been referenced. If no such page exists, the sms stalls.

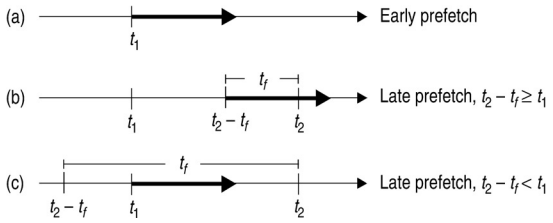


**FIGURE 3.** Timings of the prefetching algorithms.

LP is defined by the next reference and fair replacement rules and by the following rule:

- *Late prefetch*: let $t_1$ be the time instant when the sms becomes idle and $t_2$ be the earliest subsequent time instant at which a page is referenced that is missing in the buffer. The fetch of this page begins at the earliest time instant $t \geq \max(t_1, t_2 - t_f)$ that satisfies the fair replacement rule (Figure 3b and c).

In the situation of Figure 3b, $t_2 - t_f$ is the latest time instant making it possible to complete the fetch of a missing page before occurrence of the next reference to this page.

### 4.1. The EP algorithm

In our analysis of the memory behaviour of the EP algorithm, we shall first consider the case of $t_{sms} = 0$ and $\sigma = 0$. In this ideal situation of *no sms overhead*, full parallelism is possible between the processor activities of page reference and the sms activities of page fetch.

#### *4.1.1. Fast fetch*

We must distinguish between *fast fetch* and *slow fetch* situations. We are in the presence of a fast fetch when $t_f \leq t_{pr}$, i.e. $\phi \leq 1$. Graphically, this means that the movement of the fetch hand around the dial is faster than that of the reference hand. To simplify the presentation without any loss of generality, we shall hypothesize that $t_{pr}$ is a multiple of $t_f$, i.e. quantity $1/\phi$ is an integer.

At the beginning of the execution of the target program, page $p_0$ is loaded into the buffer. When $p_0$ becomes available, the processor begins to reference this page and the sms begins to fetch page $p_1$. Owing to the difference between the speeds of the two hands, in the period of time necessary to complete the reference to $p_0$, the sms fetches $1/\phi$ pages into the buffer, that is, pages $p_1$ to $p_{1/\phi}$. Thus, on the beginning of the reference to page $p_1$, this page is already available in the buffer.

Figure 4a shows an action of page fetch and replacement taking place in this early phase of program execution. The distance between the two hands, $\mathcal{F}$ and $\mathcal{R}$, increases until it becomes equal to $c$ buffer frames. At this point, the program enters a steady state. No further action of page fetch is possible that does not violate the fair replacement rule. Consequently, the sms stops in wait for completion of the reference to page $p_{\mathcal{F}-c}$, when this page can be selected for replacement (Figure 4b). Behaviour of this type characterises every subsequent page reference. Thus,

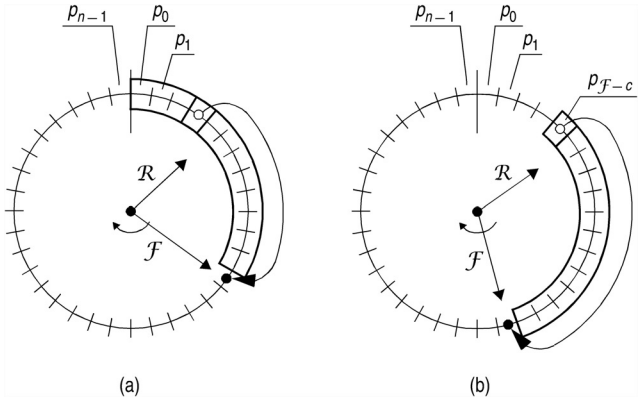


**FIGURE 4.** Memory behaviour of EP when $\phi \leq 1$: (a) transient configuration; (b) steady-state configuration.

in steady state, we have a single buffered zone of size $c$ pages that rotates clockwise around the dial at the speed of the reference hand. The program never stalls, and consequently $T_{EP} = T_{min} = mnt_{pr}$ and $R_{EP} = 1$. In each iteration of the base stream, $n$ pages are fetched from secondary memory into the buffer; thus we have $f_{EP} = n$.

If quantity $1/\phi$ is not an integer, minor alterations follow in the movement of the buffered zone around the dial. We shall not consider this issue at any further length.

#### *4.1.2. Slow fetch, small buffer*

We are in the presence of a slow fetch when $t_f > t_{pr}$, i.e. $\phi > 1$. Graphically, this means that the movement of the reference hand around the dial is faster than that of the fetch hand. We shall hypothesize that $t_f$ is a multiple of $t_{pr}$, i.e. quantity $\phi$ is an integer.

In the first iteration of the base stream, the difference between the speeds of the two hands causes the program to stall on termination of each page reference. When the buffer becomes full, the fetch hand points to page $p_c$ and the reference hand points to page $p_{c-1}$. According to the fair replacement rule, page $p_{c-2}$ is evicted from the buffer. Behaviour of this type characterizes every subsequent reference to a page in the stall zone. In this zone, the reference hand points to page $p_{\mathcal{F}-1}$, and page $p_{\mathcal{R}-1}$ is selected for replacement (Figure 5a).

On termination of the first iteration, we have a single buffered zone of size $c$ pages that covers pages $p_{n-2}, p_{n-1}, p_0, p_1, \ldots, p_{c-3}$. In steady state, the configuration of this zone is characterized by holes placed at a distance of $\phi$ pages from each other (Figure 5b). At each iteration of the base stream, the buffered zone moves one position counterclockwise. The motivations of this complex buffer configuration can be summarized as follows. When the sms starts a new page fetch, it replaces page $p_{\mathcal{R}-1}$, thereby generating a hole in $p_{\mathcal{R}-1}$. In the period of time required to accomplish the fetch, the reference hand moves $\phi$ pages forward. Consequently, when the next fetch is started up, a new hole is generated at a distance of $\phi$ pages from the previous hole. In the buffered zone the program never stalls. This is

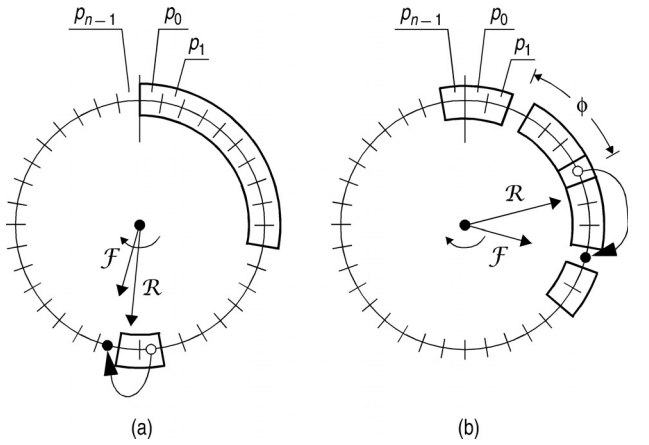


**FIGURE 5.** Memory behaviour of EP when $\phi > 1$ and $\beta \leq \beta'_{\sigma=0}$: (a) transient configuration; (b) steady-state configuration.

a consequence of the fact that in the period of time in which the processor accomplishes the page references between two holes, the sms fetches the page corresponding to the first subsequent hole.

Quantitatively, let $n_{\mathrm{h}}$ be the number of holes in the buffered zone, given by relation

$$n_{\mathrm{h}} = \frac{c}{\phi - 1}. \tag{10}$$

The size of the buffered zone (including the holes) is $c + n_{\mathrm{h}}$ pages. EP produces $n_{\mathrm{h}}$ fetch operations in the buffered zone and $n - (c + n_{\mathrm{h}})$ fetch operations in the stall zone; thus we have $f_{\mathrm{EP}} = n - c$. The fair replacement rule is always satisfied. The sms never idles, and consequently $T_{\mathrm{EP}} = mf_{\mathrm{EP}}t_{\mathrm{f}} = m(n-c)t_{\mathrm{f}}$, $R_{\mathrm{EP}} = (1-\beta)\phi$.

#### *4.1.3. Slow fetch, large buffer*

The memory behaviour described above characterizes slow fetch situations in which the size of the buffered zone, including the holes, is smaller than or equal to the size of the base stream. Let us now consider the opposite, *large buffer* situation where

$$c + n_{\mathrm{h}} > n. \tag{11}$$

Let us define quantity $\beta'_{\sigma=0} = 1 - \phi^{-1}$ for $\phi > 1$ and $\sigma = 0$. Substituting Equation (10) into (11) and manipulating, we obtain the *large buffer condition* $\beta > \beta'_{\sigma=0}$. In a situation of this type, the buffered zone extends throughout the dial and $n - c$ holes are placed at a distance of $\phi$ pages from each other (Figure 6). Each movement of the fetch hand produces an apparent movement of all the holes, and the extent of this movement is $\phi$ positions clockwise. The motivations are as follows. As a consequence of the fair replacement rule, every new page fetch replaces page $p_{\mathcal{R}-1}$, thereby producing a hole. In the period of time necessary to accomplish the fetch, the reference hand moves $\phi$ pages forward. Thus, the new hole is generated at a distance of $\phi$ pages from the previous hole. As a consequence of the next reference rule, the fetch hand always points to the first hole after the reference hand. The program never stalls and we have $T_{\mathrm{EP}} = T_{\min} = mnt_{\mathrm{pr}}$ and $R_{\mathrm{EP}} = 1$.

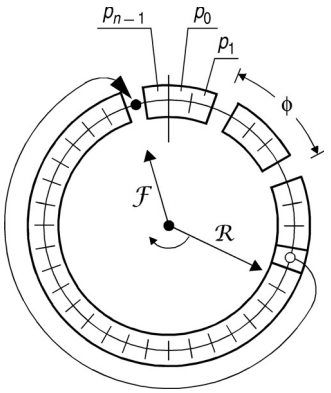


**FIGURE 6.** Memory behaviour of EP when $\phi > 1$ and $\beta \leq \beta'_{\sigma=0}$: steady-state configuration.

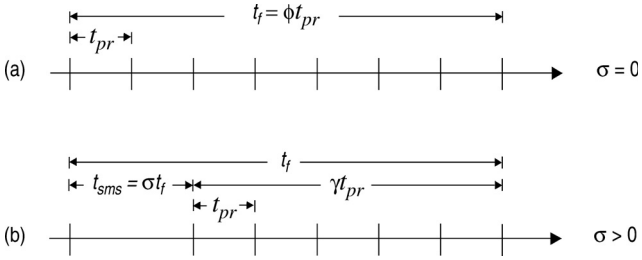


**FIGURE 7.** Relations existing between quantities $t_{\mathrm{pr}}$, $t_{\mathrm{sms}}$ and $t_{\mathrm{f}}$ when (a) $\sigma = 0$, and (b) $\sigma > 0$.

### 4.2. Effects of sms overhead in EP

Up to now, we have hypothesized that page fetch operations have no cost in terms of processor time, i.e. $t_{\mathrm{sms}} = 0$ and $\sigma = 0$. In this hypothesis, up to $\phi$ page references can be accomplished in the period of time of a single page fetch (Figure 7a). Of course, it may well be the case that less page references are accomplished, if the page to be referenced next is not available in the buffer and must be fetched from secondary memory, for instance.

Let us now consider a situation of $t_{\mathrm{sms}} > 0$ and $\sigma > 0$. In this case, for each page fetch, a fraction $t_{\mathrm{sms}} = \sigma t_{\mathrm{f}}$ of total processor time is spent to control the sms activity inherent to this fetch. Consequently, less page references are processed unless sms control activities take place when the processor would be otherwise idle. With reference to the period of time of a page fetch and Figure 7b, let us now introduce quantity $\gamma$ defined by relation $t_{\mathrm{f}} = \sigma t_{\mathrm{f}} + \gamma t_{\mathrm{pr}}$ as a generalization of normalized fetch time $\phi$. Manipulating we obtain

$$\gamma = (1 - \sigma)\phi. \tag{12}$$

Memory behaviour of EP is determined by the value of $\gamma$.

*4.2.1. Fast fetch*

When $\gamma \leq 1$ we are in a situation of fast fetch. Memory configuration is the same as in Section 4.1.1 for the case $\sigma = 0$ and $\phi \leq 1$ (see Figure 4). In steady state, the program never stalls and each page fetch adds $\sigma t_f$ time units to program response time. We have $f_{EP} = n$, and consequently

$$T_{EP} = T_{min} + mn\sigma t_f \tag{13}$$

and

$$R_{EP} = 1 + \sigma\phi. \tag{14}$$

*4.2.2. Slow fetch*

When $\gamma > 1$ we are in a situation of slow fetch. Memory behaviour is determined by the relation between normalized buffer capacity $\beta$ and quantity $\beta' = 1 - \gamma^{-1}$, defined for $\gamma > 1$ and $\sigma > 0$. When $\beta \leq \beta'$, we are in the presence of a small buffer. Memory configuration is the same as in Section 4.1.2 for the case $\sigma = 0$ and $\phi > 1$ (see Figure 5). Hence, $R_{EP} = (1 - \beta)\phi$. With respect to the case $\sigma = 0$, in the buffered zone the slowing down of the movement of the reference hand, resulting from sms overhead, is taken into account by the fact that $\beta' < \beta'_{\sigma=0}$ for every given $\phi$. In the stall zone, the processor is always idle when the need arises for an action of sms control, and, consequently, this action does not increase program response time.

When $\beta > \beta'$, we are in the presence of a large buffer. Memory configuration is the same as in Section 4.1.3 for the case $\sigma = 0$ and $\phi > 1$ (see Figure 6). The program never stalls when a new page fetch is started up, and consequently, each page fetch adds $\sigma t_f$ time units to program response time. A fetch operation occurs every time $\gamma$ page references (see Figure 7b); thus we have

$$T_{EP} = T_{min} + \frac{mn\sigma t_f}{\gamma} \tag{15}$$

and

$$R_{EP} = \frac{1}{1 - \sigma}. \tag{16}$$

### 4.3. The LP algorithm

Finally, let us consider the memory behaviour of LP. In this case, too, we must distinguish between situations of fast fetch and slow fetch.

*4.3.1. Fast fetch*

In a situation of fast fetch ($\gamma \leq 1$), the buffer becomes full in the first iteration of the base stream, on completion of the loading of page $p_{c-1}$, when the reference hand points to page $p_{c-1}$ and the fetch hand points to page $p_c$. Page $p_{c-2}$ is selected for replacement and the frame made free in this way is used to load page $p_c$ (Figure 8a). Behaviour of this type characterizes the crossing of the stall zone.

Each iteration of the base stream produces $f_{LP} = n - c$ page fetches. The program never stalls, so each page fetch adds $\sigma t_f$ time units to program response time. Hence

$$T_{LP} = T_{min} + m(n - c)\sigma t_f \tag{17}$$

and

$$R_{LP} = 1 + (1 - \beta)\sigma\phi. \tag{18}$$

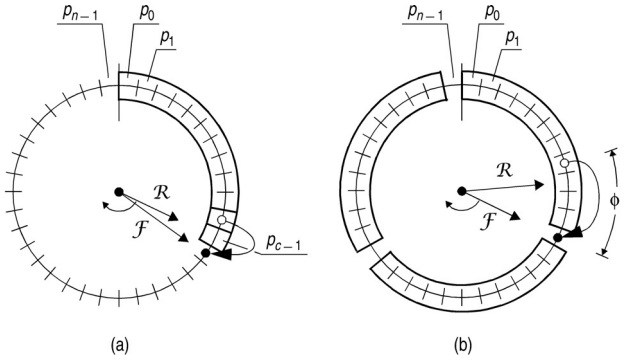


**FIGURE 8.** Memory behaviour of LP: steady-state configuration when (a) $\gamma \leq 1$, and (b) $\gamma > 1$ and $\beta > \beta'$.

*4.3.2. Slow fetch*

In a situation of slow fetch ($\gamma > 1$) and small buffer ($\beta \leq \beta'$), steady-state memory configuration of LP is the same as in Section 4.1.2 for EP in the case $\sigma = 0$ and $\phi > 1$ (see Figure 5). Thus, in steady state, $R_{LP} = (1 - \beta)\phi$.

An important difference exists between transient memory behaviour of EP and LP. In EP, all the holes in the buffered zone are generated in the first iteration of the base stream. In LP, the first iteration generates a buffered zone of size $c$ pages. The holes are generated by the $n_h$ subsequent iterations, one hole for each iteration. In transient state, the time necessary to execute an iteration is higher than in steady state. This is a consequence of the fact that fewer holes are available in the buffered zone, with negative effects on parallelism between processor activity and sms activity. If $m \ll n_h$, the extent of these effects can be evaluated easily. In a situation of this type, $P$ never reaches a steady state. By ignoring the few fetch operations issued in the buffered zone, we have

$$T_{LP} = mct_{pr} + m(n - c)t_f \tag{19}$$

and

$$R_{LP} = \beta + (1 - \beta)\phi. \tag{20}$$

Finally, in the presence of a large buffer ($\beta > \beta'$), the first iteration of the base stream produces a buffered zone of size $c$ pages. Each subsequent iteration generates a hole. In steady state, the holes are equidistant. At each iteration, they move $\phi$ positions counterclockwise (Figure 8b). Program response time is the same as for $\gamma \leq 1$, and is expressed by Equation (18).

## 5. THEORETICAL RESULTS

Table 1 summarizes the results of our analytical evaluation of performance indexes $R_x$ and $U_x$ for the caching/prefetching

**TABLE 1.** Expressions for program response time and sms utilization.

| | |
|---|---|
| $R_{\mathrm{LRU}} = 1 + \phi$ | $U_{\mathrm{LRU}} = \dfrac{\phi}{1+\phi}$ |
| $R_{\mathrm{MRU}} = 1 + (1-\beta)\phi$ | $U_{\mathrm{MRU}} = \dfrac{(1-\beta)\phi}{1+(1-\beta)\phi}$ |
| $R_{\mathrm{EP}} = \begin{cases} 1+\sigma\phi & \text{for } \gamma \le 1 \\ \dfrac{1}{1-\sigma} & \text{for } \gamma > 1, \beta \ge \beta' \\ (1-\beta)\phi & \text{otherwise} \end{cases}$ | $U_{\mathrm{EP}} = \begin{cases} \dfrac{\phi}{1+\sigma\phi} & \text{for } \gamma \le 1 \\ 1 & \text{otherwise} \end{cases}$ |
| $R_{\mathrm{LP}} = \begin{cases} 1+(1-\beta)\sigma\phi & \text{for } \gamma \le 1 \\ & \gamma > 1, \beta \ge \beta' \\ (1-\beta)\phi & \text{otherwise} \end{cases}$ | $U_{\mathrm{LP}} = \begin{cases} \dfrac{(1-\beta)\phi}{1+(1-\beta)\sigma\phi} & \text{for } \gamma \le 1 \\ & \gamma > 1, \beta \ge \beta' \\ 1 & \text{otherwise} \end{cases}$ |

$\phi = t_{\mathrm{f}}/t_{\mathrm{pr}}, \sigma = t_{\mathrm{sms}}/t_{\mathrm{f}}, \beta = c/n, \gamma = (1-\sigma)\phi, \beta' = 1-\gamma^{-1}$

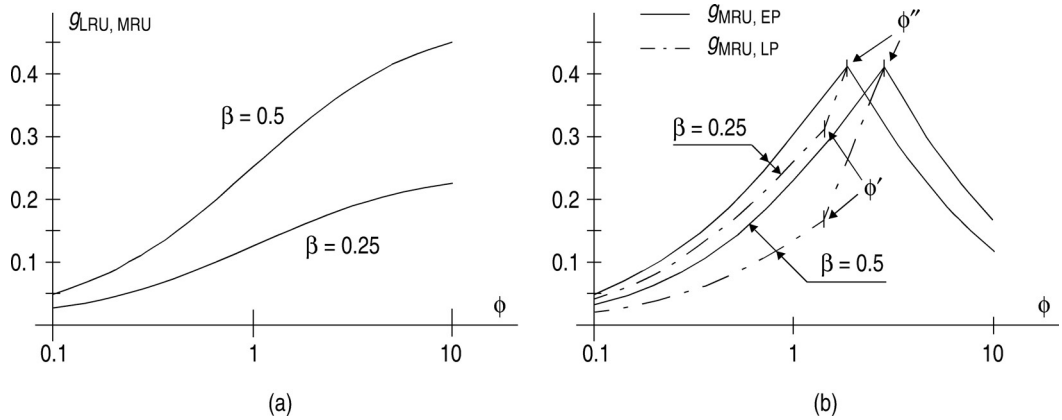


**FIGURE 9.** Relative performance gain as a function of $\phi$ for different values of $\beta$: (a) $g_{\mathrm{LRU,MRU}}$; (b) $g_{\mathrm{MRU,EP}}$ and $g_{\mathrm{MRU,LP}}$ when $\sigma = 0.3$.

algorithms considered in this paper. All the indexes are expressed in terms of quantities $\sigma$, $\phi$ and $\beta$. In this section, we shall analyze the meaning of each of these quantities in a comparative analysis of the memory behaviour of the four algorithms.

### 5.1. Fetch time

Let $x'$ and $x''$ denote two caching/prefetching algorithms. The *relative performance gain* $g_{x',x''}$ we obtain by using $x''$ instead of $x'$ is given by relation

$$g_{x',x''} = \frac{R_{x'} - R_{x''}}{R_{x'}}. \tag{21}$$

We shall take advantage of relative performance gains to analyse the effects of normalized fetch time $\phi$ on program performance. First, let us consider performance gain $g_{\mathrm{LRU,MRU}}$ resulting the from utilization of MRU instead of LRU. Substituting the expressions of $R_{\mathrm{LRU}}$ and $R_{\mathrm{MRU}}$ from Table 1 into Equation (21) and manipulating we obtain

$$g_{\mathrm{LRU,MRU}} = \frac{\beta}{1+1/\phi}. \tag{22}$$

Figure 9a shows the diagrams of $g_{\mathrm{LRU,MRU}}$ as a function of $\phi$ for $\beta = 0.25$ and $\beta = 0.5$. Response time of MRU is always shorter than that of LRU; however, for small values of $\phi$ the difference is small. In fact, the better behaviour of MRU is due to less operations of page fetch. If the time costs of these operations are low, the performance gain is marginal. The maximum value of $g_{\mathrm{LRU,MRU}}$ is $\beta$ and is produced by high values of $\phi$.

Let us now consider the performance gain resulting from the utilization of a prefetching algorithm instead of MRU. Figure 9b shows the diagrams of $g_{\mathrm{MRU,EP}}$ and $g_{\mathrm{MRU,LP}}$ as functions of $\phi$ for $\sigma = 0.3$ and two values of $\beta$, 0.25 and 0.5. Let $\phi'$ denote quantity $(1-\sigma)^{-1}$ and $\phi''$ denote quantity $(1-\sigma)^{-1} \cdot (1-\beta)^{-1}$. The diagrams of $g_{\mathrm{MRU,EP}}$ are characterized by two sharp variations, which occur at $\phi = \phi'$ and $\phi = \phi''$. The diagrams of $g_{\mathrm{MRU,LP}}$ feature a single variation, taking place at $\phi = \phi''$.

Both $g_{\mathrm{MRU,EP}}$ and $g_{\mathrm{MRU,LP}}$ are small for small values of $\phi$. Indeed, if the time required to fetch a page is negligible, we save little time by prefetching pages. Larger buffers, corresponding to high values of $\beta$, reduce the number of page fetches, thereby improving performance of non-prefetching algorithms and lowering the two performance gains.

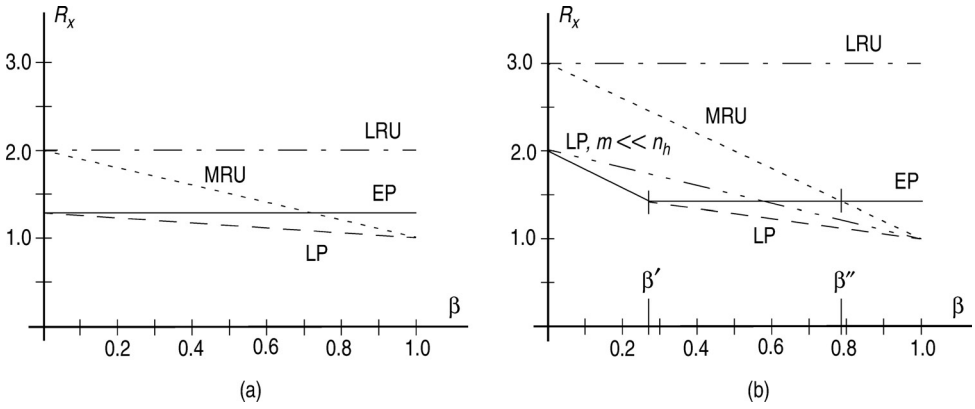


**FIGURE 10.** Normalized response time as a function of $\beta$ when $\sigma = 0.3$ and: (a) $\phi = 1$ and $\gamma = 0.7$; (b) $\phi = 2$, $\gamma = 1.4$ and $\beta = 0.28$.

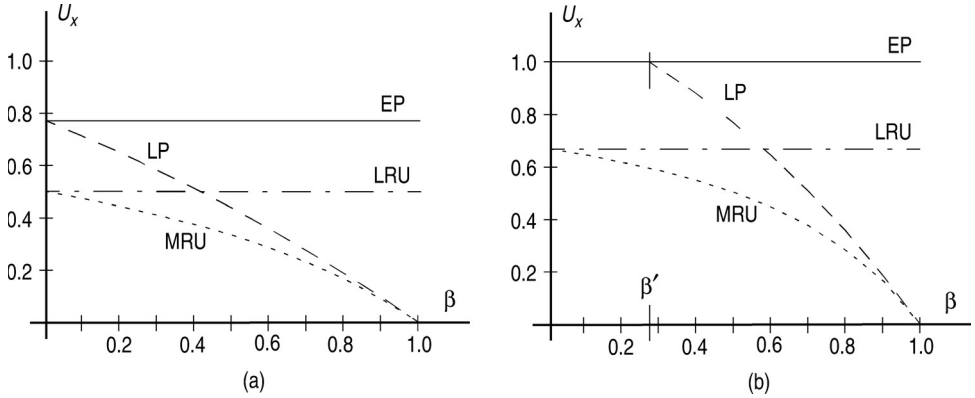


**FIGURE 11.** Sms utilization as a function of $\beta$ when $\sigma = 0.3$ and: (a) $\phi = 1$ and $\gamma = 0.7$; (b) $\phi = 2$, $\gamma = 1.4$ and $\beta = 0.28$.

When $\phi = \phi''$, we have the maximum performance gain $g_{\max}$, equal to $(1-\sigma)/(2-\sigma)$ for both EP and LP. This result is independent of the buffer size. Thus, in an ideal system featuring $\sigma = 0$, we obtain a 50% response time reduction by using a prefetching algorithm instead of MRU. A non-zero value of $\sigma$ lowers $g_{\max}$ and raises the corresponding value of $\phi''$.

Performance gains decrease for high values of $\phi$. In situations of this type, the time required to reference a page is a fraction of the time required to accomplish a page fetch. The ensuing situations of program stall reduce the advantages of page prefetching.

### 5.2. Buffer capacity

For each given caching/prefetching algorithm $x$, Figures 10 and 11 show the diagrams of normalized response time $R_x$ and sms utilization $U_x$ as a function of normalized buffer capacity $\beta$ for $\sigma = 0.3$ and two values of $\phi$, 1 and 2. $R_{\text{LRU}}$ is independent of the buffer size. In fact, LRU always makes the worst replacement decision, evicting the buffered page whose next reference is nearest in the future. LP always produces the best response time. Essentially, better usage of buffer space implies correct replacement decisions, which are more likely to occur if they are delayed as much as possible. In both MRU and LP, program response time decreases when the buffer size increases. In terms of our graphical representation, a larger buffer causes the stall zone to shrink, thereby reducing the number of page fetches (see Figure 2).

In a non-prefetching algorithm, no parallelism is possible between the processor activities of page reference and the sms activities of page fetch. Consequently, the problem of reducing response time is equivalent to that of reducing the number of page fetches and sms utilization. Prefetching algorithms decrease program response times by causing the sms to operate in parallel with the processor, and indeed, the diagrams of Figure 11 show that sms utilization in EP and LP is higher than in MRU for all buffer capacities. However, a large number of page fetches increases processor overhead for sms control in EP and LP. Observe that, by definition, $T_x \geq T_{\min} + mf_x t_{\text{sms}}$. Dividing by $T_{\min}$ and manipulating we obtain $R_x \geq (1 - \sigma U_x)^{-1}$. This relation expresses a lower bound on program response time. An increase of sms utilization $U_x$ raises this lower bound, and indeed when $\sigma > 0$ the two prefetching algorithms are unable to produce minimum execution time $T_{\min}$.

EP uses the sms intensively, and, consequently, for $\sigma > 0$ and large buffers, MRU produces better response times than EP (see Figure 10). Manipulating the expressions of $R_{\text{EP}}$ and $R_{\text{MRU}}$ from Table 1 we find that the response time of EP is

equal to that of MRU when $\beta = \beta''$, where quantity $\beta''$ is defined as follows:

$$\beta'' = \begin{cases} 1 - \sigma & \text{for } \gamma \leq 1 \\ 1 - \dfrac{\sigma}{\gamma} & \text{otherwise.} \end{cases} \quad (23)$$

Indications concerning worst-case behaviour of the four algorithms can be obtained by considering a situation of $\sigma = 1$, when $R_{\mathrm{LRU}} = R_{\mathrm{EP}}$, $U_{\mathrm{LRU}} = U_{\mathrm{EP}}$, $R_{\mathrm{MRU}} = R_{\mathrm{LP}}$ and $U_{\mathrm{MRU}} = U_{\mathrm{LP}}$. In fact, quantity $\sigma$ is an index of the degree of potential parallelism between page referencing activities and page fetch activities. When $\sigma = 1$, no parallelism is possible, and we lose all advantages inherent in page prefetching. Any performance difference between the two prefetching algorithms derives from different replacements. The better behaviour of LP is a consequence of deferred selections of the victim pages.

### 5.3. Transient behaviour

LRU and MRU reach a steady-state buffer configuration after a single iteration of the base stream, and EP reaches a steady-state configuration after two iterations. For these algorithms, transient behaviour is marginal, and the effects of the initial situation of empty buffer can be largely ignored. This is not the case for LP. As seen in Section 4.3.2, this algorithm reaches a steady state after a number of iterations of the base stream that is a function of quantity $\phi$ and can be a significant fraction of the total. This fact must be taken into careful consideration when comparing program response times.

As can be seen in Figure 10b, the steady-state response time of LP is lower than that of EP for every value of $\beta$. This is not the case if we consider transient behaviour. We derived Relation 20 as an approximation of response time $R_{\mathrm{LP}}$ in the case $m \ll n_{\mathrm{h}}$. The diagram shows that, in a situation of this type, LP is faster than MRU for every value of $\beta$ (see Figure 10). On the other hand, EP outperforms LP if the buffer is small. Essentially, LP is unable to exploit the whole sms bandwidth. While execution of the target program is traversing the buffered zone, LP causes the sms to idle even in the presence of a large stall zone. This is not the case for EP, which produces better utilization of the sms by anticipating the fetch operations needed in the stall zone.

## 6. EXPERIMENTAL RESULTS

We carried out a large set of measurement experiments aimed at validating the analytical results illustrated in the previous sections. These experiments allowed us to evaluate the extent of the simplifying assumptions made in our theoretical analysis. The testbed was a prototype of a distributed system developed within the framework of a large research effort in the field of single-address-space architectures [22, 23]. This prototype consists of a set of identical Intel Pentium-based workstations (*nodes*) connected by a fast network. The configuration of each node includes a 128-Mbyte primary memory, a Samsung SV0844 hard disk, an Intel PIIX 4 ATA33 controller and a Digital 21140A fast Ethernet adapter. While doing our measurements, the single-address-space features were disabled, and each node hosted the FreeBSD 4.2 operating system running in single-user mode.

Our target program was a synthetic benchmark program written in the C++ programming language. It was compiled by using the `gcc` compiler version 2.95.3 and forcing optimization level 2 (the `-O2` compiler option, producing optimization not involving forms of space–speed trade-offs). The program iterates execution of a base stream consisting of an ordered sequence of references to a set of data pages. A reference to a given page consists of one or more read accesses to each storage unit that forms this page. We accomplish the allocation of the buffer frames in the primary memory by taking advantage of the `mmap()` and `mlock()` system calls. So doing, we prevent the host operating system from selecting these frames for replacement. A system component, called the *buffer manager*, is deputed to implement the specific page caching/prefetching strategy. In the experiments using a prefetching algorithm, a process, called the *prefetcher*, runs in parallel with the benchmark program. The prefetcher is responsible for interactions with the sms. Process scheduling is accomplished by using the SCHED_FIFO POSIX thread scheduling policy (first-in/first-out scheduling). The priority of the prefetcher is higher than that of the benchmark program. Consequently, every request for a page fetch causes execution of the prefetcher to be immediately resumed and the request to be served as soon as the sms becomes idle. In the LP algorithm, the buffer manager estimates the latest time instant making it possible to complete the fetch of a missing page before the occurrence of the next program reference to this page. At this time instant, the buffer manager sends a synchronization signal to the prefetcher, thereby causing the page fetch to begin.

The parameters of a benchmark run include the page reference time, the number of pages in the base stream, the number of iterations of the base stream and the buffer size. Measurements were made for each caching/prefetching algorithm. To normalize program response times, we needed a suitable approximation of quantity $T_{\mathrm{min}}$. To this aim, we measured program response time in an experiment using a buffer of the same size as the base stream. We found that the algorithm has little impact on this measurement ($<2\%$). The approximation of $T_{\mathrm{min}}$ utilized in the results presented here was obtained by using MRU.

We implemented two secondary memory subsystems, a *local sms* and a *remote sms*. In the experiments using the local sms, the data pages referenced by the target program were stored in a dedicated partition of the hard disk that was accessed in raw mode. Intervention of the FreeBSD buffer cache was inhibited, and every request for a block read was served by the hard disk controller via direct memory access. An intrinsic difference exists between secondary memory behaviour of a system of this type and that hypothesized in our analytical model. In the model, page fetch time $t_{\mathrm{f}}$ is considered constant and independent of both the time instant when the request for a page fetch is issued, the specific page to be prefetched, and the sequence of previous page fetches. In fact, as a consequence of the physical attributes of disks,

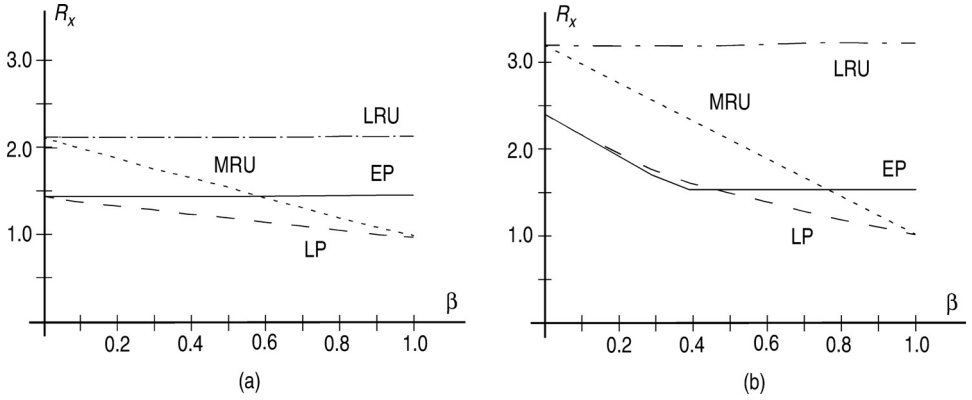


**FIGURE 12.** Normalized response time as a function of $\beta$ in the measurement experiments using the remote sms when $t_f = 600\,\mu s$ and: (a) $t_{pr} = 500\,\mu s$; (b) $t_{pr} = 256\,\mu s$. The page size was 4 KBytes.

these factors are prone to affect sms behaviour deeply. Page fetch time $t_f$ is the result of three components: (i) seek time $t^{seek}$, i.e. the time to move the disk arm to the track containing the page $p$ to be fetched; (ii) rotation latency $t^{latency}$, i.e. the time for the first sector containing $p$ to rotate under the read-write head; and (iii) page transfer time $t^{transfer}$, i.e. the time necessary to transfer a block of bits of size one page under the read–write head. Both seek time and rotation latency depend on the position of the head and the disk platters when a new page fetch is started up.

In the experiments using the remote sms, we took advantage of the distributed architecture of our testbed system. A node, called the *client node*, was dedicated to run the benchmark program, whereas the data pages accessed by this program were stored in the primary memory of a different node, the *page server*. A portion of the primary memory of the client node was reserved for the program buffer. Thus, the fetching of a page $p$ into the buffer produced transmission of the page contents across the network, from the primary memory area reserved for $p$ in the server node to the buffer frame reserved for $p$ in the client node. Transmission was accomplished by using a simple protocol of page request, delivery and acknowledgment of receipt, layered over the User Datagram Protocol. In these experiments, the time necessary to copy a page from the page server to the program buffer in the client node was essentially constant. This situation well matches our theoretical hypothesis of a constant page fetch time. By eliminating the effects of hard disk mechanics, these experiments allowed us to validate the analytical model. Furthermore, the remote sms experiments can be effectively used to characterize program behaviour in a distributed environment featuring a page server equipped with a large primary memory cache [24], [25], [26]. In a situation of this type, all page fetches in steady state are accomplished from the cache rather than from disk [27].

### 6.1. Remote sms

In the following, we shall present the results of two experiments carried out in a system configuration using the remote sms. These experiments are representative of situations of fast fetch ($\gamma \leq 1$) and slow fetch ($\gamma > 1$), respectively. In both experiments the page size was 4 KBytes, page fetch time $t_f$ was 600 μs, the base stream generated 100 page references and its execution was iterated 100 times.

#### *6.1.1. Fast fetch*

In the first experiment, a page reference consisted of ∼160 thousand instructions, corresponding to a page reference time $t_{pr}$ of 500 μs. The instructions were equally partitioned between register loads, register increments, arithmetic adds, comparisons and jumps. The value of normalized page fetch time resulting from ratio $t_f/t_{pr}$ is $\phi' = 1.2$. According to Relation 12, the fast fetch condition $\gamma \leq 1$ is satisfied when normalized sms control time $\sigma$ is >0.17. We shall validate this assumption shortly.

Figure 12a shows the experimental diagrams of normalized response time as a function of normalized buffer capacity. A close correspondence exists between these diagrams and those resulting from our analytical evaluation (see Figure 10a). The value $\phi''$ of normalized page fetch time that best approximates the measurement results can be interpolated by applying the minimum square method to the analytical expressions of $R_{LRU}$ and $R_{MRU}$ (see Table 1). The result is $\phi'' = 1.12$. The small difference (<7%) between this value and value $\phi'$ computed above is due to the fact that the analytical expressions of $R_{LRU}$ and $R_{MRU}$ do not take into account the cold-start situation that takes place at the beginning of the execution of the target program when the buffer is empty.

We shall hypothesize that value $\phi''$ also applies to the two prefetch algorithms, EP and LP. Indeed, normalized page fetch time is defined in terms of two quantities, $t_f$ and $t_{pr}$, which are independent of the algorithm. In this hypothesis, the value of normalized sms control time can be derived from the measurement results. We obtain $\sigma = 0.4$ for both EP and LP, thereby validating our initial hypothesis of $\gamma \leq 1$.

#### *6.1.2. Slow fetch*

In the second experiment, the page reference time $t_{pr}$ was 256 μs. The value of normalized page fetch time resulting from ratio $t_f/t_{pr}$ is $\phi' = 2.34$. If we hypothesize a value of $\sigma$

close to that found in the fast fetch situation (a hypothesis that we shall validate shortly), then we have $\gamma > 1$ corresponding to a slow fetch situation. Figure 12b shows the experimental diagrams of normalized response time as a function of normalized buffer capacity. In this case, too, a close correspondence exists between these diagrams and those resulting from our analytical study (see Figure 10b). The value $\phi''$ of normalized page fetch time that best approximates the measurement results is 2.18. In this case, too, the difference between $\phi'$ and $\phi''$ is quite small (<7%).

The analytical results for $\sigma = 0.3, \phi = 1$ and $\beta \leq \beta'$ indicate a difference of 1 between normalized response times $R_{\mathrm{EP}}$ and $R_{\mathrm{LP}}$ and normalized response time $R_{\mathrm{MRU}}$ (see Figure 10b). In our measurements, the average difference is 0.7. This discrepancy is a consequence of a delay existing in our experiments between the end of a page prefetch and the beginning of the next prefetch. This delay is mainly due to the context switches between the target program and the prefetcher, and to the activities of the prefetcher. We quantified this delay in 50μs. By adding this delay to page fetch time $t_{\mathrm{f}}$, we obtain a close correspondence between the measurement results and the analytical results.

For high values of $\beta$, both the analytical model and the experimental measurements indicate that $R_{\mathrm{EP}}$ is independent of $\beta$ (see Figures 10b and 12b). An estimate of normalized sms control time can be obtained by using Relation 16 and the experimental value of $R_{\mathrm{EP}}$. The result is $\sigma = 0.34$.

### 6.2. Local sms

In the experiments using local sms, we clustered the pages of the base stream according to criteria of temporal locality. We placed page $p_{i+1}$ involved in the $(i+1)$-th reference in the disk sectors contiguous to those reserved for page $p_i$. So doing, we pursued stabilization of page fetch time. In detail, in the stall zone the missing pages are fetched in sequence. Let $t_{\mathrm{f,stall}}$ denote the time required to fetch a page in this zone. If $t_{\mathrm{f}} > t_{\mathrm{pr}}$ we have

$$t_{\mathrm{f,stall}} = t^{\mathrm{transfer}}. \tag{24}$$

In the buffered zone, page fetch activities take place only if we are in the presence of holes (see Section 4.1.2 and Figure 5b, for instance). Let $t_{\mathrm{f,buffered}}$ denote the time required to fetch a page in this zone. If the holes are placed at a distance of $\phi$ pages from each other and the latest fetch involved page $p_i$, the next fetch will involve page $p_{i+\phi}$. If page $p_{i+\phi}$ is stored in the same disk track as page $p_i$, we have $t^{\mathrm{seek}} = 0$. A rotation latency $t^{\mathrm{latency}} = (\phi - 1)t^{\mathrm{transfer}}$ is necessary for the disk arm to move across the disk sectors storing pages $p_{i+1}, p_{i+2}, \ldots, p_{i+\phi-1}$. Thus,

$$t_{\mathrm{f,buffered}} = \phi t^{\mathrm{transfer}}. \tag{25}$$

At the beginning of a new iteration of the base stream, if the size of the base stream exceeds the track size a movement of the disk arm takes place from the disk position of the last page referenced in the previous iteration to the position of the first page referenced in the new iteration. If the base stream has sufficient length, this situation is comparatively rare, and we can safely ignore the resulting seek time and rotation latency.

Now consider the memory behaviour of LRU. As pointed out in Section 3.1, in each iteration this algorithm causes the fetch of all the pages that form the base stream. As a consequence of page clustering in contiguous disk sectors, if $t_{\mathrm{f}} > t_{\mathrm{pr}}$ we have no rotation latency. Furthermore, a seek time is paid only once for each track and can be ignored. We may conclude that a suitable approximation of page transfer time $t^{\mathrm{transfer}}$ can be obtained by measuring the page fetch time in the absence of page prefetching, by using LRU.

We shall now present the results of two experiments carried out in a system configuration using the local sms. These experiments are representative of situations of fast fetch ($\gamma \leq 1$) and slow fetch ($\gamma > 1$), respectively. In both experiments, the page size was 8 KBytes, the base stream was composed of 2000 page references and its execution was iterated 50 times.

#### 6.2.1. *Fast fetch*

In the first experiment, the page reference time $t_{\mathrm{pr}}$ was 518 μs. By using LRU, we measured an average page fetch time $t_{\mathrm{f}}$ of 487 μs. Thus, normalized page fetch time $\phi'$ was 0.94, corresponding to a situation of fast fetch ($\gamma \leq 1$). Figure 13a shows the experimental diagrams of normalized response time as a function of normalized buffer capacity. The value $\phi''$ of normalized page fetch time that best approximates the measurement results is 0.91. The difference between $\phi'$ and $\phi''$ is <4%.

As seen in Sections 4.2.1 and 4.3.1, when $\gamma \leq 1$ the steady state configurations of both EP and LP feature no holes in the buffered zone. Consequently, disk access times are essentially constant and the analytical model is accurate. Indeed, the experimental diagrams of Figure 13a match the theoretical diagrams of Figure 10a well. By using the experimental diagrams, we obtain $R_{\mathrm{EP}} = 1.32$, with a standard deviation of 0.1%, and, from Relation 14, $\sigma = 0.31$.

#### 6.2.2. *Slow fetch*

In the second experiment, the page reference time $t_{\mathrm{pr}}$ was 145 μs. By using LRU, we measured an average page fetch time $t_{\mathrm{f}}$ of 450 μs. Thus, normalized page fetch time $\phi'$ was 3. Condition $\gamma > 1$ corresponding to a slow fetch situation is satisfied when $\sigma < 0.67$. In this hypothesis, the diagrams resulting from the theoretical analysis are those of Figure 10b. The number of iterations of the base stream is much smaller than the number of pages in the sequence, resulting in $m \ll n_{\mathrm{h}}$. As seen in Section 5.3, this consideration is important as far as the LP algorithm is concerned.

Figure 13b shows the results of our measurements in diagram form. The experimental behaviour of LRU and MRU is very similar to the theoretical behaviour, whereas this is not the case for EP and LP. As seen in Section 4.3.2, our model indicates that, in the buffered zone, LP causes a negligible number of page prefetches. In the stall zone, every page fetch takes place in parallel with the processor activity of page reference, and this is the main reason for the

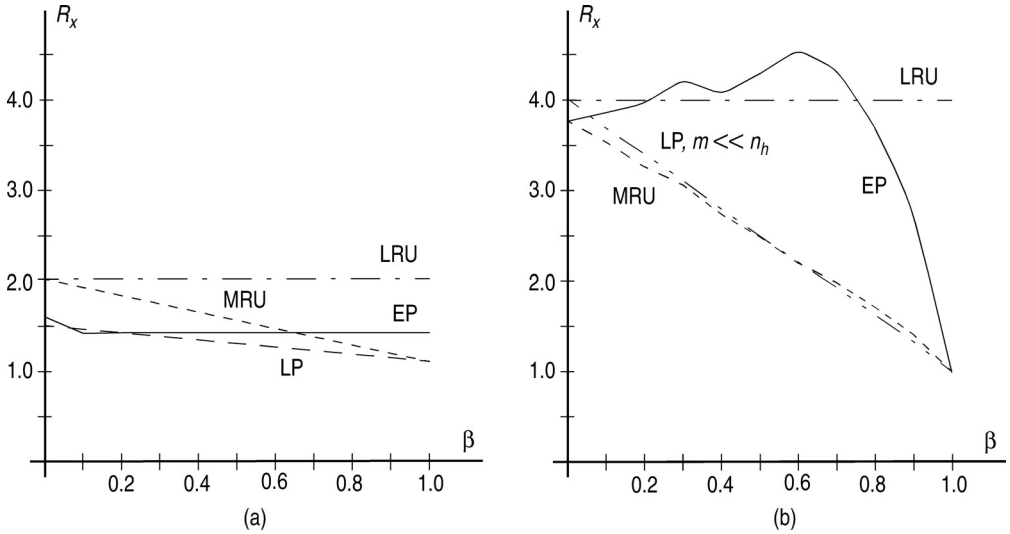


**FIGURE 13.** Normalized response time as a function of $\beta$ in the measurement experiments using the local sms when: (a) $t_{\rm pr} = 500\,\mu$s and $t_{\rm f} = 487\,\mu$s; (b) $t_{\rm pr} = 145$ and $t_{\rm f} = 450\,\mu$s. The page size was 8 KBytes.

better performance of LP with respect to MRU. On the other hand, our measurements indicate that MRU and LP produce essentially the same response times. This is a consequence of the form of data prefetch that the disk controller carries out at the hardware level. On termination of the read of a given sector, the controller autonomously accesses the subsequent sectors and copies their contents into an internal buffer, thereby reducing disk access times in the case of subsequent read operations involving these sectors. The rationale is that most of the file accesses are highly sequential.

The behaviour of EP deserves special attention. In our experiments, the response time of this algorithm is significantly worse than in the theoretical analysis. For small buffer sizes, the response time is roughly independent of buffer capacity, and the algorithm seems unable to take advantage of a buffer size increase. The causes of this anomaly are related to the significant number of holes that EP produces in the buffered zone when the buffer is small. Our analytical model indicates that the holes are at a distance of $\phi$ pages from each other (see Section 4.2.2). Our experimental measurements indicate that the distance $\bar{\phi}$ between the holes is determined by the first iteration of the base stream, when the pages are fetched sequentially into the buffer. In a situation of this type, the page fetch time features a single component, $t^{\rm transfer}$; thus we have $\bar{\phi} = t^{\rm transfer}/t_{\rm pr}$. Let $n_{\rm buffered} = c/(\bar{\phi} - 1)$ denote the number of holes in the buffered zone and $n_{\rm stall} = n - c - n_{\rm buffered}$ denote the number of pages in the stall zone. The following relation expresses the time $t$ necessary to accomplish a single iteration of the base stream:

$$t = n_{\rm stall} \cdot t_{\rm f,stall} + n_{\rm buffered} \cdot t_{\rm f,buffered}. \tag{26}$$

Substituting Equations (24) and (25) into (26) and manipulating we obtain

$$t = nt^{\rm transfer}. \tag{27}$$

This quantity is independent of the buffer capacity.

Intuitively, the addition of $\bar{\phi}$ buffer frames allows us to save as many page fetches in the stall zone. However, the new hole that is generated in the buffered zone increases time cost $t$ by a quantity roughly equal to $\bar{\phi}$ page fetch times, thereby nullifying the advantages deriving from the larger buffer. In contrast, our analytical model hypothesizes a constant page fetch time. Consequently, a new hole in the buffered zone increases $t$ by the time cost of a single page fetch, and this time cost is more than counterbalanced by the time savings in the stall zone. The overall result is a decrease of $t$.

For large buffers, our measurements for the EP algorithm indicate that program response time is no longer constant and diminishes when the buffer size increases. Indeed, when the buffered zone (including the holes) extends throughout the base stream, the program never stalls (see Section 4.1.3). On the other hand, page fetch activities are still necessary to load the pages corresponding to the holes into the buffered zone. The resulting sms overhead has a negative impact on overall performance, causing response time of EP to be worse than those of MRU and LP.

## 7. RELATION TO PREVIOUS WORK

In this section, we shall consider two classes of algorithms that received much attention in the recent past, the aggressive prefetching and the informed prefetching algorithms. As will be made clear shortly, the results of research concerning these algorithms are closely related with our results.

### 7.1. Aggressive prefetching

Cao *et al.* [21] identified a set of conditions that must be obeyed by any optimal prefetching and caching strategy. The *optimal prefetching* and *optimal replacement* conditions state that every prefetch action must involve the non-buffered page whose next reference is nearest in the future while evicting the buffered page whose next reference is furthest in the future. The *do no harm* condition forbids replacement of page $p_i$

with page $p_j$ if $p_i$ will be referenced before $p_j$. Finally, the *first opportunity* condition forbids a given activity of page fetch and replacement if this activity could have been carried out previously. The first two conditions univocally identify the pages involved in any activity of page fetch and replacement, while the other conditions provide some advice concerning when to prefetch.

Investigation focused on the memory behaviour of an integrated strategy for prefetching and caching, called *aggressive*. The aggressive strategy performs the fetch of the next page missing in the buffer at the earliest opportunity that does not violate the four conditions. Trace-driven simulation experiments were performed to compare the memory behaviour of the aggressive prefetching strategy with the behaviour of six non-prefetching strategies, which were essentially variants of LRU and Belady's optimal off-line replacement strategy [28].

If applied to the context of programs with looping reference patterns, our EP algorithm is equivalent to (i.e. it produces the same sequence of page fetches and replacements as) Cao *et al.*'s aggressive algorithm. As far as the LP algorithm is concerned, in a fast fetch situation of $\gamma \leq 1$ the fetch of a new page is delayed until the latest time instant compatible with completion of the fetch before beginning of the reference to this page (Figure 8a). In this time interval, the two hands do not move and the first opportunity condition is violated. On the other hand, we demonstrated that LP may behave better than EP, which complies with the four conditions. This is essentially a consequence of the overhead of issuing prefetch requests in terms of processor times for sms control.

In fact, Cao *et al.*'s analytical model ignores sms overhead. Consequently, the costs connected with page prefetching activities are underestimated. In our terminology, situations of this type correspond to $t_{\text{sms}} = 0$ and $\sigma = 0$. The results of our theoretical analysis and experimental measurements demonstrate that assumptions of this type are undue simplifications. For instance, EP and LP produce the same response times when $\sigma = 0$, whereas when $\sigma > 0$ LP is faster than EP. This result is true even if $\gamma > 1$. Furthermore, when $\sigma = 0$ EP is faster than MRU, whereas this is not the case when $\sigma > 0$ and $\beta > \beta''$, and the value of $\beta''$ diminishes when $\sigma$ increases.

Cao *et al.*'s simulation experiments assume a constant disk access time, equal to the average of the disk access times resulting from the program traces. We considered a situation of this type in our remote sms experiments. On the other hand, the electromechanical nature of hard disks leads to variations in page fetch times, and our experience with the local sms evidences the significant effects of these variations on the behaviour of the EP algorithm, for instance. Considerations of this type suggest that a hypothesis of a constant disk access time can be effectively used to model specific system configurations, e.g. a remote file server, but is unable to produce correct results for an application program accessing a large amount of data partially kept in the local secondary storage devices.

In a subsequent paper [29], Cao *et al.* extended their analysis to interactions between the three aspects, application-controlled caching, prefetching and disk scheduling. Both single-process and multi-process environments were taken into consideration. As pointed out by the authors, page prefetching provides new opportunities to improve disk performance, if several prefetch requests are generated at the same time instead of just one request, for instance. In a situation of this type, the disk driver has an opportunity to reduce both disk seek time and rotation latency by ordering the prefetch requests according to increasing block numbers [30].

Our analysis approaches the disk-scheduling problem from a different point of view. Rather than letting the sms dynamically reorder disk access requests to improve disk access times, we hypothesize that the pages are placed on the disk surface by taking the program patterns of page reference into consideration. In particular, if two pages are contiguous in the base stream, they are placed in contiguous disk areas. As far as disk latency is concerned, we evaluated the negative effects produced by the holes in the buffered zone in situations of slow fetch. It is worth noticing that the addition of dynamic disk scheduling will not alleviate this problem. Moreover, the holes move one position counterclockwise at each iteration of the base stream, and, consequently, we cannot devise a different placement of pages on the disk surface capable of reducing latency.

Cao *et al.*'s study formulates no hypothesis concerning program access patterns to the memory pages. Consequently, the authors were not in a position to extend the results of their analysis beyond general indications concerning algorithm behaviour in memory, e.g. lower and upper bounds on program response times. In their simulation experiments, they selected application programs characterized by substantially different memory reference patterns. The results of these experiments were used to validate the theoretical results and evaluate the impact on program performance ensuing from the integration of various combinations of application-controlled caching, prefetching and disk scheduling.

Rather than considering a collection of different programs, our experimental evaluation took advantage of a synthetic benchmark program to investigate the behaviour of a specific class of application programs, i.e. programs exhibiting a looping pattern of memory reference. So doing, we were in a position to vary both the caching/prefetching algorithm and a number of parameters characterizing program utilization of system resources. As a result, we obtained a thorough knowledge of program and system reactions to these variations.

Glass and Cao [14] analysed the memory traces of a set of benchmark programs exhibiting very different execution characteristics. As a result of this analysis, they propose a classification of programs into three categories, according to program behaviour in memory. The first category includes memory-intensive programs whose memory access patterns are on a large scale. In each execution interval, a program in this category references a large number of memory pages. The second category includes memory-intensive programs whose memory access patterns are on a small scale. The third category includes programs whose memory references

span a large memory area in highly regular, sequential access patterns. The application programs considered in our analysis belong to this third category. It should be pointed out that a program of this type not necessarily accesses the memory pages in address order. An example is a program that repeatedly references the elements of a circular list. A program of this type may well exhibit a strong looping behaviour that complies well with our model, even if the list elements occupy non-contiguous memory positions.

### 7.2. Informed prefetching and caching

#### *7.2.1. Application-controlled memory management*

Traditional systems implement reactive forms of memory management whereby a disk access is started up in response to an application demand for a missing page. In these systems, the algorithm for page replacement is fixed and independent of the program generating the page fault. This approach produces satisfactory results only if a close correspondence exists between the memory access pattern of the running program and the memory access paradigm hypothesized by the page replacement algorithm. If this correspondence is lacking, significant performance degradations may follow [31].

In an alternative, *application-controlled* approach [32, 33, 34, 35], the application program supplies the memory management system with indications concerning future behaviour in terms of requests of memory resources. Patterson *et al.* call these indications *disclosing hints* [19, 20]. A disclosing hint takes the form of the name of a file and an access pattern for this file, for instance. The program transmits the hints to the memory management system via system calls. The programmer inserts these calls explicitly at appropriate points of the program code. Alternatively, automatic compiling techniques can be used for the generation of the hints in the object code [36, 37]. The application can be modified to predict future memory accesses and generate the hints. To this aim, the application will use the processor cycles that would otherwise be wasted as a consequence of stalls [38]. Disclosing hints support a proactive vision of memory management in which the system improves overall utilization of memory resources by prefetching pages aggressively.

In a different view of application-controlled memory management, hints contain indications concerning the memory management strategy that is more suitable for the given application program. In Patterson *et al.*'s terminology, these are called *hints that give advice*. They are especially effective when the memory reference pattern of the program is known and predictable.

The analysis of caching and prefetching algorithms presented in this paper is intended to support a form of application-controlled memory management in which the application program supplies the memory management system with both disclosing hints and hints that give advice. The disclosing hints concern the names of the pages that form the base stream. The advice specifies the caching and prefetching algorithm that is best suited to support program execution. The algorithm will be chosen by characterizing program activity and system configuration in terms of quantities $t_{\mathrm{pr}}$, $t_{\mathrm{sms}}$ and $t_{\mathrm{f}}$. These quantities will be used to evaluate and compare the resulting program performance indexes. In this phase, we shall take advantage of the relations of Table 1 as well as the diagrams of Figures 9–11.

As a first example, let us consider a database query application. Database management systems usually include a software component, called the buffer manager, which is responsible for selection of the page to be evicted from the buffer when a free frame is lacking [9]. The buffer manager uses its own page replacement algorithm, and the common choice is LRU [35]. Let us now consider two tables, $t_1$ and $t_2$, and let $t_3 = t_1 \bowtie t_2$ be the result of their natural join. Memory behaviour of the natural join application complies with our model of a looping reference pattern. For this program, our analysis demonstrates that, in the absence of prefetch, MRU produces both a fast response time and low sms utilization (see Figures 10 and 11). In an application-controlled approach, we shall modify the code implementing the query by adding instructions to transmit a hint to the buffer manager. In this case, the advice is to replace the ubiquitous LRU with MRU.

As a further example, let us refer to a program with a looping reference pattern, and suppose that the page reference time of the program is shorter than the page fetch time, so that $\gamma > 1$. Suppose that the program produces a number $m$ of iterations of the base stream that is much smaller than the number $n_{\mathrm{h}}$ of iterations necessary to reach a steady state. As shown in Section 5.3, in a situation of this type LP produces unsatisfactory response times, and the hint will be to use EP (see Figure 10b). On the other hand, if the buffer is small, prefetch algorithms utilize the sms intensively (see Figure 11). In a multiprogrammed environment featuring a high level of contention for secondary memory resources, sms utilization may be an important factor, and a non-prefetching algorithm may have a more appropriate result, at least for low-priority programs. In this case, the hint will be to use MRU.

#### *7.2.2. The fixed-horizon algorithm*

Patterson *et al.* [19] propose a form of cost-benefit analysis supporting integration of proactive memory management with LRU caching.

Page prefetching takes advantage of an algorithm of informed prefetching, which in a subsequent paper [39] is called the *fixed horizon* algorithm. The horizon is the time instant when to prefetch a given page before the occurrence of the next reference to this page. The algorithm determines the horizon by considering the time necessary to carry out the preceding references. Prefetching a page at any time instant before the horizon leads to no advantage in terms of program response time.

Of course, a close relationship exists between the concept of a prefetch horizon and the LP estimate of the earliest time instant suitable for prefetch (see Figure 3b and c). However, in both [19] and [39], unlimited availability of disk arms is supposed to provide enough disk parallelism so that the

need for disk queuing never arises. On the contrary, we hypothesize that the sms can honour no more than a single page fetch request at a time. Consequently, every request for a page prefetch must be delayed unless the sms is idling. A delayed prefetch will be able to mask program perception of page fetch time only partially. The program will stall, albeit for a time interval shorter that the entire duration of a page fetch.

*7.2.3. An example of implementation*

Huizinga and Desai [16] report the results of an implementation effort concerning integration of informed prefetching and caching techniques within the framework of the Linux operating system, as a replacement of the traditional read-ahead mechanism of the Linux kernel. They introduce a new system primitive, called *prefetch*(), having the same format as the *read*() primitive. An application program will take advantage of *prefetch*() to transmit a hint to the memory management system before issuing the corresponding *read*().

The authors carried out a wide set of measurement experiments on a Linux system compiled to include informed prefetching features. With respect to read-ahead, the results of these experiments evidence noticeable performance improvements for all programs whose file accesses are not strictly sequential and no performance penalty for programs exhibiting highly sequential access patterns. These results confirm the suitability of application-controlled memory management strategies for integration in a widely used, off-the-shelf system.

*7.2.4. Predictive prefetch*

Vellanki and Chervenak [40] distinguish between deterministic prefetching systems and probabilistic systems. A deterministic system decides which secondary memory blocks to prefetch next by taking advantage of application-provided hints taking the form of an ordered list of blocks that will be accessed in the future. An example of such a system is the Patterson *et al.*'s [19, 20] informed prefetching implementation. In contrast, a probabilistic system tries to derive a list of future memory accesses from an analysis the past accesses.

Vellanki and Chervenak propose a form of predictive prefetching as a variant of the informed prefetching scheme that takes advantage of a cost-benefit analysis in a probabilistic framework. In contrast with informed prefetching, predictive prefetching does not require that application programs be modified to insert system calls for transmission of the hints to the memory management system. Instead, the system takes the decision of which block to prefetch autonomously, by using a probability prefetch tree based on past accesses.

The analysis of the predictive prefetching algorithm differs from our analysis from a number of important viewpoints, while presenting a few interesting analogies. In both cases, much attention is paid to the costs of the actions of page prefetch in terms of the time necessary for the processor to start up the disk activity. This important factor was often neglected in similar studies. On the other hand, the predictive prefetch analysis assumes unlimited availability of disk units, no disk congestion and a constant disk access time. Conversely, we allow a single page fetch at a time while using disk access time as a parameter of our evaluation.

Vellanki and Chervenak [40] evidence that the advantages resulting from block prefetch activities decrease for large buffer sizes. This result is confirmed by our study.

The predictive prefetching study pays special attention to the cache miss rate for different cache sizes and page reference times. Of course, cache miss rates are especially well suited to compare effectiveness of different probabilistic algorithms to identify the future block references [41]. It is worth noticing that in the presence of block prefetch, the miss rate of the disk cache is only one of the factors that determine overall program performance. In fact, the cost of a cache miss in terms of program execution times decreases significantly if the prefetch of the missing page was started up before the generation of the miss.

## 8. CONCLUDING REMARKS

We have carried out a thorough analysis of the memory behaviour of a number of page caching and prefetching algorithms. The analysis was restricted to programs whose paradigms of memory access are known in advance. More specifically, we considered programs with looping reference patterns, whose execution consists of the iteration of a sequence of read accesses to a set of memory pages. We considered two page replacement algorithms, LRU and MRU, selected to limit the range of all possible program response times in the absence of page prefetch, and two prefetch algorithms, EP and LP, characterized by very different degrees of sms activity.

The theoretical analysis was mainly aimed at expressing program response time and sms utilization in terms of system configuration and program activity. We paid special attention to transient program behaviour and the effects of the time necessary for the processor to control the disk activities of page fetch. To this aim, we introduced a graphical representation of program execution modelling both page placement in the primary memory buffer and the actions of page fetch and replacement. The results of a large set of measurement experiments were used to validate the analytical model while acquiring significant indications concerning the extent of the simplifying assumptions made in the theoretical analysis. In the discussion of the relation to previous work, we paid special attention to two classes of algorithms that received much attention in the past, aggressive prefetching and informed prefetching.

A number of concluding observations follow:

- An analysis of page caching and prefetching algorithms should always pay special attention to the effects of the mechanical limitations of hard disks. In this respect, a number of simplifying assumptions that are often considered quite acceptable in theoretical studies should always be validated carefully by using data resulting from experimental measurements in real systems. An

example is the hypothesis that the page fetch time be independent of the specific page to be fetched as well as the sequence of previous fetches. We have seen that a hypothesis of this type, albeit quite acceptable to model the behaviour of a page server, is prone to significant underestimation of program response times in a system configuration featuring a local sms, e.g. in the case of the EP algorithm (see Section 6.2.2).

- The time necessary for the processor to control the sms has a significant impact on program response times. In the corresponding time interval, no parallelism is possible between processor activity and sms activity. Consequently, the effects of sms control time cannot be simply assimilated to those of a longer page fetch time, for instance. A comparative analysis of prefetching algorithms should consider sms overhead carefully (see Section 4.2, for instance).
- Steady state behaviour of programs in memory may be significantly different from transient behaviour. Transient analysis is especially important if the caching and prefetching algorithm reaches its steady state after a number of iterations spanning a large fraction of program execution. As seen in Section 5.3, this is the case for LP, whose transient behaviour in a situation of small buffer is characterized by significantly worse performance than steady state behaviour.
- Page placement in the buffer may result in configurations of the buffered zone of the base stream that are both odd and to some extent counterintuitive. Examples are the configurations produced by EP and LP if the page fetch time is greater than the page reference time. As seen in Sections 4.1.2, 4.1.3 and 4.3.2, in situations of this type the buffered zone features a number of equidistant holes. The results of the experimental measurements made in a local sms configuration indicate that these holes are responsible for significant increases in program response time (see Section 6.2.2).

In-depth understanding of program behaviour in memory may give suggestions for significant enhancements of program performance. Techniques of this type are of special interest for time-critical applications that are worth the effort of modifying the program code, to include hint for the memory management system according to informed prefetch techniques, for instance. We hope that our work will result in a significant contribution in this direction.

## ACKNOWLEDGEMENT

The work described in this paper was supported in part by the Italian Ministero dell'Istruzione, dell'Università e della Ricerca in the framework of a research project entitled 'High Quality Web Systems'.

## REFERENCES

[1] Johnson, T. and Shasha, D. (1994) 2Q: a low overhead high performance buffer management replacement algorithm. In *Proc. Twentieth Int. Conf. on Very Large Data Bases*, Santiago, Chile, September 12–15, pp. 439–450. Morgan Kaufmannn, San Mateo, CA.

[2] Lee, D., Choi, J., Kim, J.-H., Noh, S. H., Min, S. L., Cho, Y. and Kim, C. S. (1999) On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. *Perf. Eval. Rev.*, **27**, 134–143.

[3] O'Neil, E. J., O'Neil, P. E. and Welkum, G. (1993) The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, **22**, 297–306.

[4] Smaragdakis, Y., Kaplan, S. and Wilson, P. (1999) EELRU: simple and effective adaptive page replacement. *Perform. Eval. Rev.*, **27**, 122–133.

[5] Megiddo, N. and Modha, D. S. (2003) ARC: a self-tuning, low overhead replacement cache. In *Proc. Second USENIX Conf. on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 31–April 2, pp. 115–130. USENIX Association, Berkeley, CA.

[6] Bansal S. and Modha, D. S. (2004) CAR: clock with adaptive replacement. *Proc. Third USENIX Conf. on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 31–April 2, pp.187–200. USENIX Association, Berkeley, CA.

[7] Choi, J., Noh, S. H., Min, S. L. and Cho, Y. (1999) An implementation study of a detection-based adaptive block replacement scheme. In *Proc. 1999 USENIX Annual Technical Conf.*, Monterey, CA, June 6–11, pp. 239–252. USENIX Association, Berkeley, CA.

[8] Choi, J., Noh, S. H., Min, S. L. and Cho, Y. (2000) Towards application/file-level characterization of block references: a case for fine-grained buffer management. *Perf. Eval. Rev.*, **28**, 286–295.

[9] Faloutsos, C., Ng, R. and Sellis, T. (1995) Flexible and adaptable buffer management techniques for database management systems. *IEEE Trans. Comput.*, **44**, 546–560.

[10] Kim, J. M., Choi, J., Kim, J., Noh, S. H., Min, S. L., Cho, Y. and Kim, C. S. (2000) A low-overhead high performance unified buffer management scheme that exploits sequential and looping references. In *Proc. Fourth USENIX Symp. on Operating System Design and Implementation*, San Diego, CA, October 22–25, pp. 119–134. USENIX Association, Berkeley, CA.

[11] Gniady, C., Butt, A. R. and Hu, Y. C. (2004) Program counter based pattern classification in buffer caching. In *Proc. Sixth USENIX Symp. on Operating Systems Design and Implementation*, San Francisco, CA, December 6–8, pp. 395–408. USENIX Association, Berkeley, CA.

[12] Shalicky, T. *LASPack Reference Manual*. Available at http://www.tu-dresden.de/mwism/skalicky/laspack/laspack.html.

[13] Pasquale, B. K. and Polyzos, G. C. (1993) A static analysis of I/O characteristics of scientific applications in a production workload. In *Proc. 1993 ACM/IEEE Conf. on Supercomputing*, Portland, OR, November 15–19, pp. 388–397. IEEE Computer Soceity Press, Los Alamitos, CA.

[14] Glass, G. and Cao, P. (1997) Adaptive page replacement based on memory reference behavior. *Perf. Eval. Rev.*, **25**, 115–126.

[15] Patterson, R. H. and Gibson, G. A. (1994) Exposing I/O concurrency with informed prefetching. *Proc. Third Int. Conf. on Parallel and Distributed Information Systems*, Austin, TX, September 28–30, pp. 7–16. IEEE Computer Society Press, Los Alamitos, CA.

[16] Huizinga, D. M. and Desai, S. (2000) Implementation of informed prefetching and caching in Linux. *Proc. Int. Conf. on Information Technology: Coding and Computing*, Las Vegas,

NV, March 27–29, pp. 443–448. IEEE Computer Soceity, Los Alamitos, CA.

[17] Albers, S., Garg, N. and Leonardi, S. (2000) Minimizing stall time in single and parallel disk systems. *J. ACM*, **47**, 969–986.

[18] Kaplan, S. F., McGeoch, L. A. and Cole, M. F. (2002) Adaptive caching for demand prepaging. In *Proc. Third Int. Symp. on Memory Management*, Berlin, Germany, June 20–21, pp. 114–126. ACM Press New York, NY.

[19] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D. and Zelenka, J. (1995) Informed prefetching and caching. *Oper. Syst. Rev.*, **29**, 79–95.

[20] Patterson, R. H., Gibson, G. A. and Satyanarayanan, M. (1993) A status report on research in transparent informed prefetching. *Oper. Syst. Rev.*, **27**, 21–34.

[21] Cao, P., Felten, E. W., Karlin, A. R. and Li, K. (1995) A study of integrated prefetching and caching strategies. In *Proc. ACM SIGMETRICS Joint Int. Conf. on Measurement and Modeling of Computer Systems*, Ottawa, Ontario, Canada, May 15–19, pp. 188–197. ACM Press New York, NY.

[22] Dini, G., Lettieri, G. and Lopriore, L. (2000) An overview of Ulisse, a distributed single address space system. In Kirby, G. N. C., Dearle, A. and Sjøberg D. I. K. (eds) *Proc. Ninth Int. Workshop on Persistent Object Systems: Design, Implementation, and Use, POS-9*, Lillehammer, Norway, September 6–8, *LNCS*, **2135**, 215–227. Springer-Verlag GmbH, Berlin.

[23] Lopriore, L. (2002) Access control mechanisms in a distributed, persistent memory system. *IEEE Trans. Parall. Distr. Syst.*, **13**, 1066–1083.

[24] Dahlin, M. D., Wang, R. Y., Anderson, T. E. and Patterson, D. A. (1994) Cooperative caching: using remote client memory to improve file system performance. In *Proc. First USENIX Symp. on Operating Systems Design and Implementation*, Monterey, CA, November 14–17, pp. 267–280. USENIX Association, Berkeley, CA.

[25] Feeley, M. J., Morgan, W. E., Pighin, F. H., Karlin, A. R., Levy, H. M. and Thekkath, C. A. (1995) Implementing global memory management in a workstation cluster. *Oper. Syst. Rev.*, **29**, 201–212.

[26] Franklin, M. J., Carey, M. J. and Livny, M. (1992) Global memory management in client-server DBMS architectures. In *Proc. 18th Int. Conf. on Very Large Data Bases*, Vancouver, BC, Canada, August 23–27, pp. 596–609. Morgan Kaufmannn, San Mateo, CA.

[27] Voelker, G. M., Anderson, E. J., Kimbrel, T., Feeley, M. J., Chase, J. S., Karlin, A. R. and Levy, H. M. (1998) Implementing cooperative prefetching and caching in a globally-managed memory system. *Perform. Eval. Rev.,* **26**, 33–43.

[28] Belady, L. A. (1966) A study of replacement algorithms for virtual storage computers. *IBM Syst. J.*, **5**, 78–101.

[29] Cao, P., Felten, E. W., Karlin, A. R. and Li, K. (1996) Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, **14**, 311–343.

[30] Lumb, C. R., Schindler, J. and Ganger, G. R. (2002) Freeblock scheduling outside of disk firmware. In *Proc. USENIX Conf. on File and Storage Technologies (FAST 02)*, Monterey, CA, January 28–30, USENIX Ass., pp. 275–288.

[31] Shenoy, P. J., Goyal, P., Rao, S. S. and Vin, H. M. (1998) Symphony: an integrated multimedia file system. In *Proc. ACM/SPIE Multimedia Computing and Networking 1998*, San Jose, CA, January 26–28, pp. 124–138. SPIE—International Society for Optical Engineering.

[32] Bartoli, A., Dini, G. and Lopriore, L. (2001) Application-controlled memory management in a single address space environment. *Int. J. Softw. Tools Technol. Transfer*, **3**, 235–245.

[33] Cox, M. and Ellsworth, D. (1997) Application-controlled demand paging for out-of-core visualization. In *Proc. Conf. on Visualization'97*, Phoenix, AZ, October 19–24, pp. 235–244. IEEE, New York, NY.

[34] Harty, K. and Cheriton, D. R. (1992) Application-controlled physical memory using external page-cache management. *SIGPLAN Notices*, **27**, 187–197.

[35] Krueger, K., Loftesness, D., Vahdat, A. and Anderson, T. E. (1993) Tools for the development of application-specific virtual memory management. *ACM SIGPLAN Notices*, **28**, 48–64.

[36] Brown, A. D. and Mowry, T. C. (2001) Compiler-based I/O prefetching for out-of-core applications. *ACM Trans. Comput. Syst.*, **19**, 111–170.

[37] Mitra, T., Yang, C.-K. and Chiueh, T. (2000) Application-specific file prefetching for multimedia programs. In *Proc. Int. Conf. on Multimedia and Expo*, New York, NY, July 30–August 2, pp. 459–462. IEEE, Piscataway, NJ.

[38] Chang, F. and Gibson, G. A. (1999) Automatic I/O hint generation through speculative execution. In *Proc. Third Symp. on Operating Systems Design and Implementation*, New Orleans, LA, February 22–25, pp. 1–14. USENIX Association, Berkeley, CA.

[39] Kimbrel, T., Tomkins, A., Patterson, R. H., Bershad, B., Cao, P., Felten, E. W., Gibson, G. A., Karlin, A. R. and Li, K. (1996) A trace-driven comparison of algorithms for parallel prefetching and caching. *ACM Oper. Syst. Rev.*, **30**, 19–34.

[40] Vellanki, V. and Chervenak, A. L. (1999) A cost-benefit scheme for high performance predictive prefetching. In *Proc. SCC99 Conf. on High Performance Networking and Computing*, Portland, OR, November 14–19, ACM, New York, NY.

[41] Yeh, T., Long, D. D. E. and Brandt, S. A. (2001) Performing file prediction with a program-based successor model. In *Proc. Ninth Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Cincinnati, OH, August 15–18, pp. 193–202, IEEE Computer Society Press, Los Alamitos, CA.