

Performance Optimization Problem in Speculative Prefetching

Nor Jaidi Tuah, Mohan Kumar, *Senior Member, IEEE*,
Svetha Venkatesh, *Senior Member, IEEE*, and Sajal K. Das, *Member, IEEE*

Abstract—Speculative prefetching has been proposed to improve the response time of network access. Previous studies in speculative prefetching focus on building and evaluating access models for the purpose of access prediction. This paper investigates a complementary area which has been largely ignored, that of performance modeling. We analyze the performance of a prefetcher that has uncertain knowledge about future accesses. Our performance metric is the *improvement in access time*, for which we derive a formula in terms of *resource* parameters (time available and time required for prefetching) and *speculative* parameters (probabilities for next access). We develop a prefetch algorithm to maximize the improvement in access time. The algorithm is based on finding the best solution to a *stretch knapsack problem*, using theoretically proven apparatus to reduce the search space. An integration between speculative prefetching and caching is also investigated.

Index Terms—Speculative prefetching, caching.

1 INTRODUCTION

CACHING and prefetching of data have been used to improve the speed of information access across a network. In *caching*, copies of remote data are kept locally to reduce access time of repeatedly accessed data. In *prefetching*, access to remote data is anticipated and the data is fetched before it is required. This is in contrast to *demand fetch* where data is fetched only when it is actually requested.

Prefetching can either be *speculative*, where the knowledge about future accesses is not perfect, or *informed*, where the look-ahead to future accesses is certain. In this paper, we investigate speculative prefetching. Previous studies in speculative prefetching (see Section 1.1) focus on building access models and evaluating the performance of such models in predicting future accesses. While these models are important, they do not constitute a complete framework on which to build optimal prefetching strategies. To complement an access model, simple heuristics are usually resorted to, such as prefetching an item if the probability of its access is larger than a fixed threshold. Though these heuristics might be intuitively sound and their usefulness is empirically confirmed, more analytical treatment is required to understand their performance and merits.

We believe that, in addition to an access model, a prefetcher requires a resource model and a performance model. A *resource model* allows a prefetcher to predict the

amount of available and required resources. A *performance model* allows a prefetcher to optimize the usage of resources and adapt well to changing resource conditions, whereas heuristics using empirically tuned parameters may settle for a less optimal performance. In this paper, we develop a prefetch algorithm based on a performance model, assuming the existence of an access model and a resource model to provide the necessary predictions.

Prefetching competes for memory resources with caching. We found excellent work on the integration of informed prefetching and caching [13], [2], but no analogous work in speculative prefetching.

1.1 Related Work

Many recent studies in speculative prefetching assume persistence in trends of user request patterns. Tait [17] and Lei and Duchamp [8] use file access pattern based on the features of UNIX-style operating system where every program gives rise to a tree of forked processes that access some files. Each time a program runs, its access tree is constructed and compared against previously saved trees to detect a repeating pattern. When a good match is found, the nodes that would complete the currently constructed tree are prefetched.

Speculative prefetching has been proposed for improving web access [1], [9], [12]. Padmanabhan and Mogul [12] suggested server-side access prediction. The server builds a dependency graph where a link from item *A* to *B* means that *B* is likely to be accessed within a short interval after an access to *A*. Each link is labeled with the probability of the follow-up access being made, possibly intervened by accesses to other items. The authors in [12] also described a resource model for the purpose of estimating the time for retrieving files across the network, but this is simply to drive the simulation and not incorporated into the prefetcher.

The ETEL electronic newspaper project [1] proposed a client-side prediction in which the client builds a patterned frequency graph that contains a path for each sequence of

- N.J. Tuah is with the Faculty of Science, Universiti Brunei Darussalam, Gadong BE 1410, Brunei. E-mail: jaidi@fos.ubd.edu.bn.
- M. Kumar and S.K. Das are with the Department of Computer Science and Engineering, University of Texas at Arlington, Box 19015, Arlington, TX 76019-0015. E-mail: {kumar, das}@cse.uta.edu.
- S. Venkatesh is with the School of Computing, Curtin University of Technology, GPO BOX U1987, WA 6845, Australia. E-mail: svetha@cs.curtin.edu.au.

Manuscript received 7 July 1999; accepted 10 May 2001.

For information on obtaining reprints of this article, please send e-mail to: tpd@computer.org, and reference IEEECS Log Number 110186.

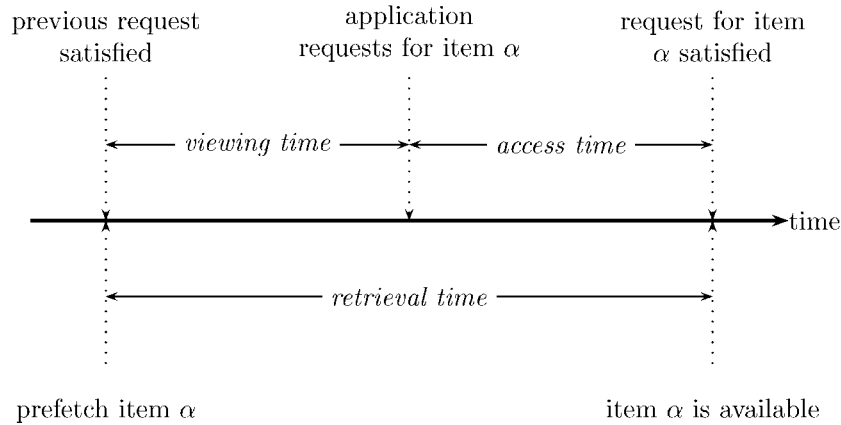


Fig. 1. Time durations.

accesses. Each arc in the graph is labeled with a probability of the successor being accessed immediately after the predecessor. ETEL monitors the accuracy of its prediction. It prefetches single files when the accuracy is high. When the accuracy is low, it prefetches multiple files.

Jiang and Kleinrock [5] combine the server-side and the client-side predictions for web browsing with the help of a dynamic threshold scheme. A formula for prefetch threshold is derived within a performance model that considers the costs of access delay and resource usage. To minimize the total cost, a file is prefetched only if its access probability is greater than or equal to the prefetch threshold. The threshold changes dynamically based on the system load, capacity, and the cost to the user.

Vitter and Krishnan [19] use data compression techniques to build an access tree that can make optimal predictions if the accesses are generated by a Markov process.

Integrated prefetching and caching policies are investigated in Transparent Informed Prefetching (TIP) [13] and Application-Controlled File System (ACFS) [2]. However, both TIP and ACFS assume look-ahead of future accesses.

1.2 Contributions of this Paper

With regard to performance of speculative prefetching, most of the previous works do not attempt any analytical study. Most of the results are essentially empirical, obtained either through Monte-Carlo simulation [1], [9], trace-driven simulation [8], [12], [17], or actual implementation [8], [17]. The theoretical analysis in [19] assumes zero delay in retrieving data and performance is measured in terms of fault rates.

Empirical studies have the disadvantage that fine-tuning for performance is based on trial and error. Furthermore, projecting an empirically derived result to a different network condition may not be possible. This is because it can be difficult to pinpoint and weigh the parameters that contribute to the prefetcher performance. We believe that the incorporation of an analytical performance model would allow a prefetcher to be more efficient as well as more adaptive.

In this paper, we study the performance of speculative prefetching and develop algorithms to maximize the expected improvement in access time (i.e., the network response time as perceived by the user/application). Our

model presupposes some knowledge about future accesses. In particular, it has a list of candidate items for the next access. It also presupposes some knowledge about the available and the required resources. In particular, the available time for prefetching, the retrieval time for each candidate item, and cache availability are known. Because we include cache as a parameter, our model leads to an algorithm that integrates prefetching with cache replacement.

The rest of the paper is organized as follows: Section 2 describes the parameters and the performance metrics used in our analysis. In Section 3, specific cases of prefetching are investigated. In particular, we consider the cases of prefetching only one and two items. In Section 4, we derive formulae for improvement in access time when an arbitrary number of items are prefetched. When the cache is assumed empty, the problem of maximizing this improvement reduces to a *stretch knapsack problem* described in Section 5. In Section 6, we consider the problem of integrating the prefetch and cache replacement decision to maximize the improvement in access time. Section 7 highlights the main points of this research and scope for further work.

2 FRAMEWORK OF ANALYSIS

We use the generic term “item” to refer to the object being accessed. This might be a file, a document, an SQL reply, etc. Each item is identified by a unique number, e.g., item 2. We use $\langle \rangle$ to enclose a list of items and use this font (ABC) for list names. RS is the concatenation of R and S . $|\mathcal{R}|$ is the number of elements in \mathcal{R} . The symbols for set operations, such as \in , \subset , \subseteq and \setminus , are used for list operations with their usual meanings.

2.1 Retrieval Model

The opportunity for prefetching comes when an application is waiting for the user input or carrying out some processing. For convenience, we shall refer to the time of such opportunity as the *viewing time*. We use the term *retrieval time* to refer to the time to fully retrieve an item. When a remote item is actually requested, the network may appear to be more responsive if the requested item has been prefetched and is already partially or fully retrieved. We use the term *access time* to refer to the response time to an actual request. Fig. 1 illustrates our terminology for the time durations.

We assume that the application can learn the probabilities of items being requested in the future and their retrieval times. Probabilities of future accesses can be derived from an access model constructed using statistics of past accesses, as in [1], [12]. In [19], a data compression technique is used to create an access model. A technique for estimating retrieval time in the presence of variable network latency and bandwidth is given in [20].

Since the decision to prefetch is based on speculation, there is no guarantee that a prefetched item is the one that will be accessed next. The user may issue a request while a prefetch for a different item is still in progress. When this happens, we assume that the communication bandwidth is shared evenly between the ongoing prefetch and the demand fetch for the newly issued request. For example, suppose we have two items, 1 and 2, which have individual retrieval times of 100 seconds (secs) and 20 secs, respectively. Assume that item 1 is prefetched at time 0, but, at time 30 secs item 2 is requested and retrieval for item 2 thus ensues. In the first 30 secs, the bandwidth is fully deployed for the retrieval of item 1. In the next 40 secs, the bandwidth is shared between the two items, at the end of which item 2 will arrive completely. After that the bandwidth will once again be fully deployed to item 1 which will arrive after the next 50 secs. This model does not consider the possibility of overlapping the start-up latency of request for item 2 with ongoing transmission of item 1.

2.2 Performance Metrics

Prefetch is carried out to improve the access time. We define the improvement in access time, referred to simply as *access improvement* and denoted by G , as follows:

$$G = E[t \mid \text{no prefetch}] - E[t \mid \text{prefetch}], \quad (1)$$

where t is the access time and $E[\cdot]$ denotes an “expected” value. A positive value of G implies an expected decrease in access time.

In speculative prefetching, since knowledge about future accesses is imperfect, there is bound to be an increase in network load resulting from incorrect prefetches. We refer to the increase in network load as the *retrieval excess cost*, denoted by C . The retrieval excess cost due to prefetching is defined as the expected retrieval time with prefetching minus the expected retrieval time without prefetching. A simple formula for C is:

$$C = \sum_{i \in \mathcal{F}} (1 - p_i) r_i, \quad (2)$$

where \mathcal{F} is the list of prefetched items, p_i is the probability of item i getting accessed, and r_i is its retrieval time.

We also use *relative access improvement*, denoted by G_r , and *access improvement over excess cost*, denoted by G_c . These are defined as follows:

$$G_r = \frac{G}{E[t \mid \text{no prefetch}]} \quad \text{and} \quad G_c = \frac{G}{C}. \quad (3)$$

3 ANALYSIS OF SPECIFIC CASES

In this section, we analyze the specific cases of prefetching one and two items. The current cache content is assumed irrelevant for the next request.

3.1 Prefetching One Item

We first analyze the performance of prefetching a single item when the following are known: the duration of viewing time (v), the probability and the retrieval time of the most probable item to be accessed next, and the expected retrieval time for other items. Let the list of accessible items be $\langle 1, \dots, n \rangle$, with item 1 being the top candidate for the next access. Let α denote the actual request. Let the probability $p_i = P\{\alpha = i\}$ and r_i be the retrieval time for item i . Then, v , p_1 , r_1 , and $\sum_{i=2}^n P\{\alpha = i \mid \alpha \neq 1\} r_i = R$ are known parameters. We make a simplification in our analysis by using R for retrieval time of items other than item 1.

If prefetch is not performed, then the expected access time is given by

$$E[t \mid \text{no prefetch}] = p_1 r_1 + \sum_{i=2}^n p_i R. \quad (4)$$

If prefetch is carried out, then the expected access time is given by

$$E[t \mid \text{prefetch item 1}] = p_1 t_{\text{hit}} + \sum_{i=2}^n p_i t_{\text{miss}}. \quad (5)$$

$$\text{where} \quad t_{\text{hit}} = \begin{cases} 0 & \text{if } r_1 \leq v \\ r_1 - v & \text{otherwise} \end{cases} \quad (6)$$

$$\text{and} \quad t_{\text{miss}} = \begin{cases} R & \text{if } r_1 \leq v \\ R + r_1 - v & \text{if } v < r_1 \leq v + R \\ 2R & \text{otherwise.} \end{cases} \quad (7)$$

In (6), access time is zero when the retrieval is completely masked by the viewing time. The three conditions in (7) correspond, respectively, to the following three cases: 1) The prefetch is completed before the demand fetch begins, 2) the demand fetch is partially slowed down by the prefetch, and 3) the demand fetch is completely slowed down by the prefetch.

Using (4), (5), (6), and (7), we derive the access improvement as

$$G = \begin{cases} p_1 r_1 & \text{if } r_1 \leq v \\ v - \sum_{i=2}^n p_i r_1 & \text{if } v < r_1 \leq v + R \\ p_1 v - \sum_{i=2}^n p_i R & \text{otherwise.} \end{cases} \quad (8)$$

Fig. 2 plots G , G_r , and G_c against r_1 for various combinations of R and p_1 , where time measures are relative to v . In all cases, G and G_r are maximum when $r_1 = v$, i.e., when the viewing time is fully used for prefetching and the item is ready when the application requires it.

The improvement indicated by G can be deceiving. Though G may indicate considerable access improvement when prefetching an item that has a slow retrieval time (for brevity, henceforth we use the terms “slow item” and “fast item”), the improvement can in fact be insignificant. This is because the ratio G_r quickly diminishes as r_1 gets larger. The upper limit for the value of G_r when $r_1 > v$ is the asymptotic v/r_1 . The ratio G_r also gets smaller as R gets larger with respect to v . For web browsing, this means prefetching does not give much improvement to a casual

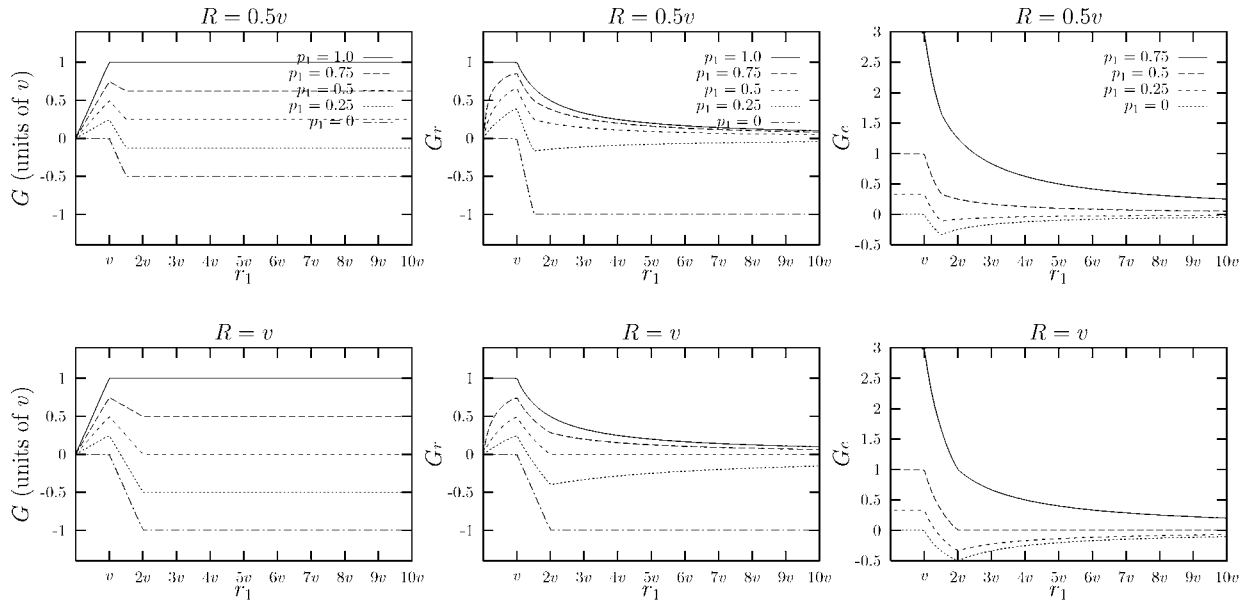


Fig. 2. Access improvement due to prefetching.

user who does not spend much time in reading documents and to a user who has a very slow network link. The latter case may be surprising because prefetching is performed especially to help us manage with slow network retrieval. Now, we show that it will not help much if the link is very slow! Let us take an extreme case to illustrate this point. If the network is infinitely slow (i.e., disconnected), then prefetching will not help at all.

The retrieval excess cost for the prefetch is $\sum_{i=2}^n p_i r_i$. Hence, $G_c = G / (\sum_{i=2}^n p_i r_i)$. The plots for G_c show that the best improvement relative to the retrieval excess cost is achieved when $r_1 \leq v$. When $p_1 = 1$, G_c is infinity because no extra retrieval time is spent for the prefetch.

The window within which G_r and G_c indicate improvement is rather critical. Initially, we tend to think that it is useful to prefetch a slow item. However, the results show that the improvement is small and the cost is high. Generally, there is a more significant improvement in prefetching a fast item, but, in absolute terms, it might be a difference of just a few seconds of the user's waiting time.

When the viewing time is large, it can be exploited to prefetch more than one item. We shall consider two different types of prefetch methods for multiple items, namely, *mainline prefetch* and *branch prefetch*, as explained below.

3.2 Mainline Prefetch

When the prefetcher is informed that "item 1 and then item 2 would be accessed," it may prefetch both items within the same viewing time. We refer to this as *mainline prefetch*. The advantage of mainline prefetch is that we are maximizing the overlap of prefetch time with viewing time. Another advantage, which we will not consider in our analysis, is the possibility of eliminating or reducing the start-up latency of retrieving item 2. The disadvantage is that item 2 is further into the future and, hence, is less certain.

Let us formally state the information required to perform mainline prefetch and study its performance. Let α_1 and α_2 be the sequence of the next two accesses. We assume that the possible items for α_1 are distinct from those for α_2 . Let the lists of candidates be $\langle 1, \dots, n_1 \rangle$ for α_1 and $\langle n_1 + 1, \dots, n_2 \rangle$ for α_2 . Let $p_1 = P\{\alpha_1 = 1\}$, $p_z = P\{\alpha_2 = n_1 + 1 \mid \alpha_1 = 1\}$, $r_z = r_{n_1+1}$ and $R =$ the average retrieval time. The following are known: p_1, p_z, r_1, r_z, v , and R . For later reference, we denote this piece of information as M . To simplify our analysis, we assume a uniform viewing time and use R to approximate both $\sum_{i=2}^{n_1} P\{\alpha_1 = i \mid \alpha_1 \neq 1\} r_i$ and

$$\sum_{i=n_1+2}^{n_2} P\{\alpha_2 = i \mid \alpha_1 = 1, \alpha_2 \neq n_1 + 1\} r_i.$$

If item 1 cannot be retrieved within the viewing time, we can ignore the information regarding item $n_1 + 1$. This is because, after the application makes a request, this information becomes obsolete. So, we consider only the case where item 1 can be retrieved within the viewing time, i.e., $r_1 < v$.

Assuming $r_1 < v$, the mainline prefetch can be performed by prefetching item 1 followed immediately by item $n_1 + 1$. Another alternative is to prefetch item 1 only and wait for the next access before deciding on another prefetch. We shall refer to this as *single prefetch*. Let G_{r_M} and $G_{r_M^1}$, respectively, denote the relative access improvement for mainline prefetch and single prefetch, given M . Similarly, G_{c_M} and $G_{c_M^1}$ denote the respective access improvement over retrieval excess cost.

Fig. 3 shows plots for G_{r_M} , $G_{r_M} - G_{r_M^1}$, and $G_{c_M} - G_{c_M^1}$ against r_z . The plots for G_{r_M} have two maximums—one at $r_1 + r_z = v$ and another at $r_1 + r_z = 2v$. The first maximum point is due to a possibility in hiding the access latency of item $n_1 + 1$ with the viewing time of the current item. The second maximum is due to a possibility in hiding the access

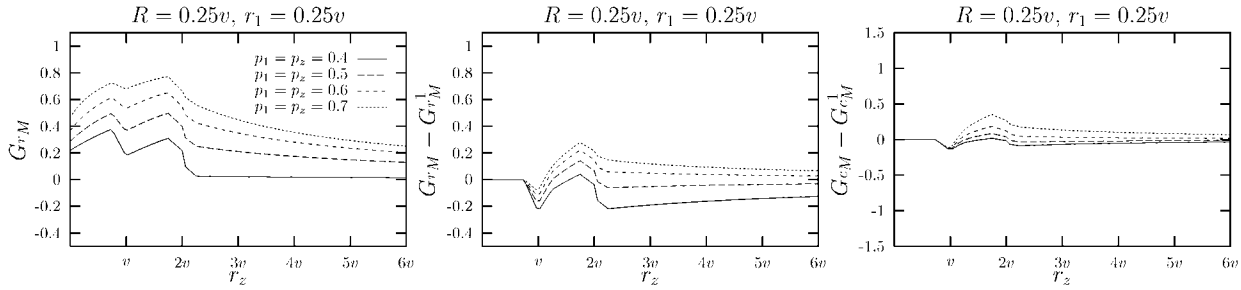


Fig. 3. Difference of mainline and single prefetch.

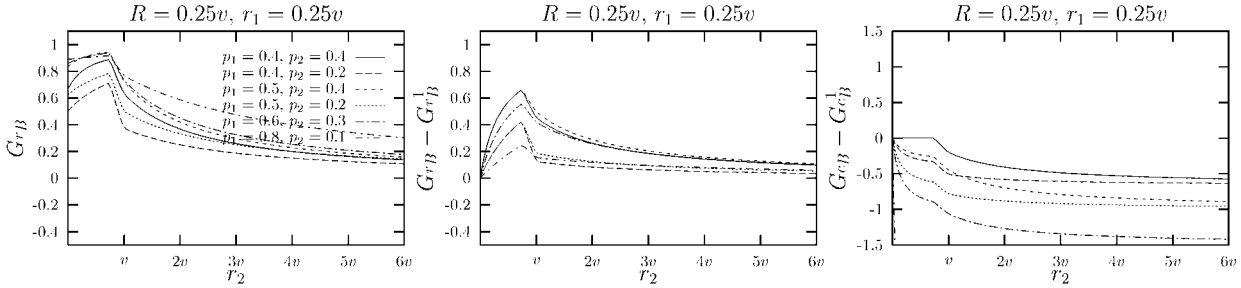


Fig. 4. Difference of branch and single prefetch.

latency of item $n_1 + 1$ with the viewing time of the current item and the viewing time of item 1 if indeed $\alpha_1 = 1$.

There is no point in performing mainline prefetch if it does not give any improvement over single prefetch; hence, we are also interested in $G_{r_M} - G_{r_M}^1$. When $r_1 + r_z < v$, there is no difference in access time between mainline and single prefetch. The curves then slope down, indicating that it is better to wait for the next viewing time to give room for the prefetch of the next item. However, the curves slope up when $r_1 + r_z$ approaches $2v$, indicating an increase in the advantage of utilizing the residual viewing time of the current item. For the smaller probability values, the improvement increase is not high enough to make mainline prefetch worthwhile.

We are also interested in how much additional retrieval time is required to achieve the improvement in mainline prefetch as opposed to single prefetch and, hence, the plots for $G_{c_M} - G_{c_M}^1$. A prefetch for multiple items cannot improve on retrieval excess cost over a prefetch of a single item because more uncertainty is involved in the former. The plots for $G_{c_M} - G_{c_M}^1$ run close to the horizontal zero axis. The negative plots indicate that, if mainline prefetch brings further improvement in access time over single prefetch, at the worst it is only with nominal extra retrieval time. The positive plots indicate that the mainline prefetch may bring improvement in access time over single prefetch for the same or slightly more retrieval cost.

3.3 Branch Prefetch

Consider the information that “either item 1 or 2 would be accessed.” When both items are prefetched within the same viewing time, we refer to this as the *branch prefetch* as two different possible continuations are prefetched.

Let us formally state the information required to perform branch prefetch and study its performance. Let α be the next access which can be any of items $1, 2, \dots, n$. The following parameters are known: v , p_1 , p_2 , r_1 , r_2 , and

$\sum_{i=3}^n P\{\alpha = i \mid \alpha > 2\}r_i = R$. We shall refer to this information as B .

If item 1 is prefetched and there is residual viewing time after its arrival, i.e., $r_1 < v$, then branch prefetch may be performed. As with mainline prefetch, we will assess branch prefetch against single prefetch.

Let G_{r_B} and $G_{r_B}^1$ denote the relative access improvement for branch prefetch and single prefetch, respectively, given B . The respective access improvement over excess cost are denoted by G_{c_B} and $G_{c_B}^1$.

Fig. 4 shows graphs for G_{r_B} , $G_{r_B} - G_{r_B}^1$, and $G_{c_B} - G_{c_B}^1$ against r_2 . The plots for G_{r_B} peak at the point where $r_1 + r_2 = v$. The plots for $G_{r_B} - G_{r_B}^1$ are positive for most of the cases considered, indicating an improvement in access time due to branch prefetch over single prefetch. However, in all cases, the plots for $G_{c_B} - G_{c_B}^1$ go from zero to negative. This indicates that any improvement due to branch prefetch is gained at the expense of considerably more network time.

3.4 Mainline versus Branch Prefetches

Direct comparison between G_{r_M} and G_{r_B} is not meaningful because the parameters are different. To simplify the analysis, we have ignored information pertaining to branch prefetch when we formulated mainline prefetch and vice versa. We may indirectly compare them by observing their improvement over single prefetch given the same respective parameters, i.e., by comparing the graphs of $G_{r_M} - G_{r_M}^1$ with $G_{r_B} - G_{r_B}^1$. Using a rough calibration, it appears that branch prefetch is expected to result in a better access time than mainline prefetch. For example, compare mainline prefetch, where $p_1 = p_2 = 0.7$ (Fig. 3), to branch prefetch, where $p_1 = 0.6$ and $p_2 = 0.3$ (Fig. 4). The latter gives more improvement over the single prefetch for the

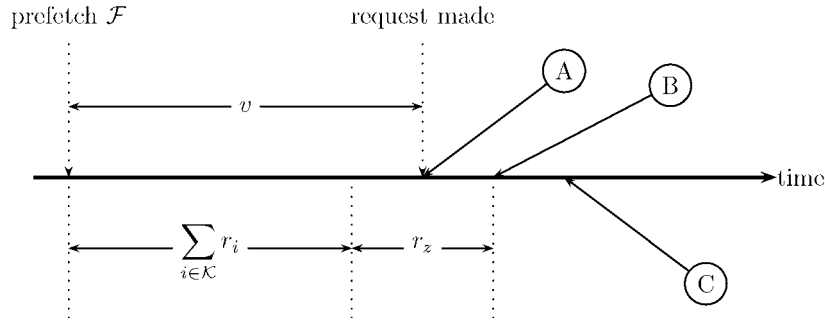


Fig. 5. Access time. (A) If $\alpha \in \mathcal{K}$, access time is zero. (B) If $\alpha = z$, access time equals the time to complete the prefetch. (C) If $\alpha \notin \mathcal{F}$, access time equals the time to complete the prefetch, plus the time to retrieve the item actually requested.

same parameters, even though the probabilities seem to favor the former (unconditional probabilities of 0.7 and 0.49 versus 0.6 and 0.3). This can be explained by considering the situation when item 1 does not get accessed next. In the case of branch prefetch, the probability of item 2 getting accessed next becomes $\frac{p_2}{1-p_1}$, whereas, in mainline prefetch, if item 1 is not accessed, then the probability of item $n_1 + 1$ getting accessed is considered zero.

The advantage of branch prefetch over mainline prefetch is not clear cut. It is at the expense of much extra retrieval cost.

4 ACCESS IMPROVEMENT

Though we have used a few performance metrics (two of which are ratios of G) in the previous section, the rest of the paper elaborates mainly on G .

In this section, we shall formulate the access improvement due to prefetching an arbitrary number of items. To simplify the analysis, we only consider a single access. This obviously means that we are dealing with a branch prefetch. We assume that, when a request is made while a prefetch for a different item is still in progress, the prefetch completes before the demand fetch.

Let $\mathcal{N} = \langle 1, \dots, n \rangle$ be the list of all accessible items. Let α be the next request. Assume that the access probabilities $p_i = P\{\alpha = i\}$ are known for all i . (Note that $P\{\alpha = i\}$ should be understood as $P\{\alpha = i \mid \text{current context}\}$.) Let r_i be the retrieval time of item i , which is also assumed known for all i . Let the duration of the viewing time be v .

Let the list of items to be prefetched be \mathcal{F} . When $\mathcal{F} = \langle \rangle$, the access improvement is trivially zero, i.e., $G(\langle \rangle) = 0$. Henceforth, in this section, we ignore this trivial case and proceed with the assumption that \mathcal{F} contains at least one element. Assume that items are prefetched in sequence so that, at the end of the viewing time, at most one item is still awaiting completion. Let \mathcal{K} be constructed as follows:

$$\mathcal{F} = \mathcal{K}\langle z \rangle \quad \text{where } \mathcal{K} \subset \mathcal{N}, z \in \mathcal{N} \setminus \mathcal{K}, \text{ and } \sum_{i \in \mathcal{K}} r_i < v. \quad (9)$$

Here, \mathcal{K} consists of items that are prefetched entirely within the viewing time. The prefetch for the last element, z , may (but not necessarily) continue past the viewing time. Note that this construction is general and requires only that \mathcal{F} is not empty and all prefetches are initiated before the next

request is made. We specify the construction for \mathcal{F} more for convenience of notation rather than restriction.

The amount by which the retrieval time of \mathcal{F} exceeds the viewing time will be called the *stretch time* and denoted as $st(\mathcal{F})$. This is defined as

$$st(\mathcal{F}) = \max \left\{ 0, \sum_{i \in \mathcal{F}} r_i - v \right\}. \quad (10)$$

4.1 Prefetch Only

Assume the cache is empty (or its current content can be disregarded). Thus, any access improvement will be solely due to prefetching. We shall use the symbols t° and G° for access time and access improvement, respectively, under this assumption.

When no prefetch is performed and a request is then made for item i , that item must be retrieved. This obviously gives an access time of r_i . Hence, the expected access time is

$$E[t^\circ \mid \text{no prefetch}] = \sum_{i \in \mathcal{N}} p_i r_i$$

Now, consider when items \mathcal{F} are prefetched. At the end of the viewing time, when the actual request is made, items \mathcal{K} are already fully prefetched. Thus, if the next request is made for item $i \in \mathcal{K}$, the access time is $t = 0$. If the request is made for item z , then we must wait for z to completely arrive, resulting in an access time of $t = st(\mathcal{F})$ (which may be 0). If the request is for item $i \notin \mathcal{F}$, then the prefetch is completely off the mark and, consequently, the requested item must be retrieved. This results in an access time of $t = r_i + st(\mathcal{F})$. These three different cases for access time are shown in Fig. 5. The expected access time is

$$\begin{aligned} E[t^\circ \mid \text{prefetch } \mathcal{F}] &= p_z st(\mathcal{F}) + \sum_{i \notin \mathcal{F}} p_i (r_i + st(\mathcal{F})) \\ &= \sum_{i \notin \mathcal{F}} p_i r_i + \sum_{i \notin \mathcal{K}} p_i st(\mathcal{F}). \end{aligned}$$

Hence, the access improvement when the cache is empty and \mathcal{F} is prefetched is,

$$\begin{aligned} G^\circ(\mathcal{F}) &= E[t^\circ \mid \text{no prefetch}] - E[t^\circ \mid \text{prefetch } \mathcal{F}] \\ &= \sum_{i \in \mathcal{F}} p_i r_i - \sum_{i \notin \mathcal{K}} p_i st(\mathcal{F}). \end{aligned} \quad (11)$$

4.2 Prefetch and Cache

We now consider the case where the cache is not empty and its contents may be relevant for serving the next request. Let the current list of items in the cache be \mathcal{C} .

When no prefetch is performed, request for item $i \notin \mathcal{C}$ requires the retrieval of that item. Hence, the expected access time is given by

$$E[t \mid \text{no prefetch}] = \sum_{i \notin \mathcal{C}} p_i r_i.$$

Now, consider when prefetch is performed. Let \mathcal{F} be constructed as in (9), except that now \mathcal{F} cannot have any elements in common with the cache, \mathcal{C} . A list of victims, \mathcal{D} , need to be ejected from the cache to give room to the incoming items. Disregarding the last item in the prefetch list, z , the new cache content at the time the request is made is $\mathcal{C}_{\text{new}} = \mathcal{K}\mathcal{C} \setminus \mathcal{D}$. There are three cases for access time to consider, similar to the three cases shown in Fig. 5: 1) If $\alpha \in \mathcal{C}_{\text{new}}$, $t = 0$, 2) if $\alpha = z$, $t = st(\mathcal{F})$, and 3) if $\alpha \notin \mathcal{C}_{\text{new}}(z)$, $t = r_i + st(\mathcal{F})$. Hence, the expected access time is

$$\begin{aligned} E[t \mid \text{prefetch } \mathcal{F}, \text{ eject } \mathcal{D}] &= p_z st(\mathcal{F}) + \sum_{i \notin \mathcal{C}_{\text{new}}(z)} p_i (r_i + st(\mathcal{F})) \\ &= \sum_{i \notin \mathcal{C}_{\text{new}}(z)} p_i r_i + \sum_{i \notin \mathcal{C}_{\text{new}}} p_i st(\mathcal{F}). \end{aligned}$$

Hence, when \mathcal{F} is prefetched and \mathcal{D} ejected, the access improvement is

$$\begin{aligned} G(\mathcal{F}, \mathcal{D}) &= E[t \mid \text{no prefetch}] - E[t \mid \text{prefetch } \mathcal{F}, \text{ eject } \mathcal{D}] \\ &= \sum_{i \in \mathcal{F}} p_i r_i - \sum_{i \in \mathcal{D}} p_i r_i - \sum_{i \notin \mathcal{K}\mathcal{C} \setminus \mathcal{D}} p_i st(\mathcal{F}) \\ &= G^\circ(\mathcal{F}) - \left(\sum_{i \in \mathcal{D}} p_i r_i - \sum_{i \in \mathcal{C} \setminus \mathcal{D}} p_i st(\mathcal{F}) \right). \end{aligned} \quad (12)$$

5 STRETCH KNAPSACK PROBLEM

Assuming the cache is empty and has no size restriction, the optimal list of items to prefetch can be obtained by solving the following problem:

$$\begin{aligned} \text{Find } \mathcal{F} \text{ to maximize } G^\circ(\mathcal{F}) &= \sum_{i \in \mathcal{F}} p_i r_i - \sum_{i \notin \mathcal{K}} p_i st(\mathcal{F}) \\ \text{subject to } \sum_{i \in \mathcal{K}} r_i &< v. \end{aligned} \quad (13)$$

The problem in (13) is *almost* a binary knapsack problem (KP). The KP has been extensively investigated (for example, see the references in [11]). If we restrict $st(\mathcal{F}) = 0$, the problem can be expressed as

$$\begin{aligned} \text{Maximize } \sum_{i \in \mathcal{N}} p_i r_i x_i \\ \text{subject to } \sum_{i \in \mathcal{N}} r_i x_i &\leq v \\ \text{and } x_i &= 0 \text{ or } 1, \end{aligned} \quad (14)$$

which in fact is a KP, where the profit and the weight of item i are $p_i r_i$ and r_i , respectively, and the knapsack

capacity is v . The main difference between the problems in (13) and (14) is that, in the former, the knapsack capacity may be exceeded. Recall that $\mathcal{F} = \mathcal{K}(z)$. Only \mathcal{K} needs to be completely inside the knapsack, while z may be partly (but not completely) outside and, thus, stretching the capacity. For this reason, we use the term *stretch knapsack problem* (SKP) to refer to the problem in (13).

Our algorithm for solving the SKP is a modification of Horowitz-Sahni's KP algorithm [4], which uses a branch-and-bound approach where a search through a binary decision tree is made coupled with bound checking to allow pruning of branches.

5.1 Anatomy of Search Space

There is a gap between KP and SKP that we need to bridge if we want to use the structure of a KP algorithm to search for the SKP solution. The gap is in the difference between their search spaces. For the KP, the solution is a *set* of items. Its search space therefore consists of all possible combinations of items. This can be completely covered by a traversal through a binary decision tree where the two branches from each nonleaf node represent the inclusion and the exclusion of an item from the solution. On the other hand, the solution for the SKP is a *list* of items. Its search space therefore consists of all permutations of items. Thus, the search space of the SKP is larger than that of the KP. In particular, if $st(\mathcal{F}) > 0$ and \mathcal{F}^* is a permutation of \mathcal{F} , then it is possible that $G^\circ(\mathcal{F}) \neq G^\circ(\mathcal{F}^*)$.

Theorem 1. *If $\bar{\mathcal{F}}$ is an optimal solution to the problem in (13) and $st(\bar{\mathcal{F}}) > 0$, then $\min\{p_f : f \in \bar{\mathcal{F}}\} = p_{\bar{z}}$, where \bar{z} is the last element in $\bar{\mathcal{F}}$.*

Proof. Suppose $\exists f \in \bar{\mathcal{F}}$ such that $p_{\bar{z}} > p_f$. We will show that this cannot be true if $\bar{\mathcal{F}}$ is an optimal solution and, hence, by contradiction, the theorem will be proven. Let $\bar{\mathcal{K}}$ be the list of all elements in $\bar{\mathcal{F}}$ excluding \bar{z} . (In other words, $\bar{\mathcal{F}} = \bar{\mathcal{K}}(\bar{z})$.) Form a list \mathcal{K}^* which is the same as $\bar{\mathcal{K}}$ except that the element f is replaced with \bar{z} . Let $\mathcal{F}^* = \mathcal{K}^*(f)$. If \mathcal{F}^* is not a feasible solution, then it is not consequential to the proof. So, let us further suppose that \mathcal{F}^* is indeed a feasible solution. From (11),

$$G^\circ(\bar{\mathcal{F}}) = \sum_{i \in \bar{\mathcal{F}}} p_i r_i - \sum_{i \notin \bar{\mathcal{K}}} p_i st(\bar{\mathcal{F}})$$

and

$$G^\circ(\mathcal{F}^*) = \sum_{i \in \mathcal{F}^*} p_i r_i - \sum_{i \notin \mathcal{K}^*} p_i st(\mathcal{F}^*).$$

Since $\bar{\mathcal{F}}$ and \mathcal{F}^* contain the same items, we get $\sum_{i \in \bar{\mathcal{F}}} p_i r_i = \sum_{i \in \mathcal{F}^*} p_i r_i$ and $st(\bar{\mathcal{F}}) = st(\mathcal{F}^*)$. From our assumption that $p_{\bar{z}} > p_f$ and the way \mathcal{K}^* is constructed, we get

$$\sum_{i \in \mathcal{K}^*} p_i > \sum_{i \in \bar{\mathcal{K}}} p_i \implies \sum_{i \notin \bar{\mathcal{K}}} p_i > \sum_{i \notin \mathcal{K}^*} p_i.$$

Hence, $G^\circ(\bar{\mathcal{F}}) < G^\circ(\mathcal{F}^*)$. \square

Theorem 1 allows us to confine the search space to permutations where the items are sorted in the descending order of probability. Even though the search space is reduced, it is guaranteed to contain an optimal solution. Note that this ordering is equivalent to descending order

of profits per unit weight, which is presumed by several KP algorithms (for example, [4], [10]).

When $p_i = p_j$, the relative ordering of items i and j cannot be arbitrary. For example, consider the problem defined by: $n = 2, p_1 = 0.5, p_2 = 0.5, r_1 = 100, r_2 = 2, v = 4$. The optimal solution to this problem, which is $\langle 2, 1 \rangle$, will be missed if the search is confined to the given ordering (because item 2 comes before item 1 in the optimal list). To avoid this problem, equally probable items are subsorted in increasing retrieval times. Henceforth, we shall assume

$$(p_i > p_{i+1}) \text{ or } (p_i = p_{i+1} \text{ and } r_i \leq r_{i+1}). \quad (15)$$

With the ordering in (15), we can now use a binary decision tree to search for the SKP solution.

5.2 Relaxation and Upper Bound

In this section, we derive an upper bound for the SKP solution. The upper bound will be used to prune the search tree.

The SKP is an integer programming problem. In the context of prefetching, an item is either entirely prefetched or not at all. By allowing items to be partially prefetched, we obtain the linear programming relaxation of SKP (linear SKP, for short).

Let x_i , where $0 \leq x_i \leq 1$, be the proportion of item i that is prefetched. We use x without the subscript to refer to the entire array x_1, \dots, x_n . Let z be the last item to be prefetched. Let \mathcal{K} be the list of wholly prefetched items not including z , i.e., $\mathcal{K} = \{i : x_i = 1, i \neq z\}$. The linear SKP is

$$\begin{aligned} \text{Maximize } \tilde{G}^\circ(x) &= \sum_{i \in \mathcal{N}} p_i r_i x_i - \sum_{i \notin \mathcal{K}} p_i \tilde{st}(x) \\ \text{where } \tilde{st}(x) &= \max \left\{ 0, \sum_{i \in \mathcal{N}} r_i x_i - v \right\}. \end{aligned} \quad (16)$$

The function \tilde{st} is the stretch time function, reformulated for the relaxed constraint of the linear SKP.

Suppose that the items, sorted according to (15), are consecutively inserted into the knapsack until the first item, \tilde{z} , is found which causes the knapsack to stretch. That is,

$$\tilde{z} = \min \left\{ j : \sum_{i=1}^j r_i > v \right\}.$$

Theorem 2. *The optimal solution \bar{x} of the linear SKP is obtained as:*

$$\bar{x}_i = \begin{cases} 1 & \text{if } 1 \leq i \leq \tilde{z} - 1 \\ (v - \sum_{i=1}^{\tilde{z}-1} r_i) / r_{\tilde{z}} & \text{if } i = \tilde{z} \\ 0 & \text{if } \tilde{z} + 1 \leq i \leq n. \end{cases}$$

Proof. By Dantzig's Theorem [3], \bar{x} is the solution to the linear programming relaxation of KP where the profit and the weight of item i are $p_i r_i$ and r_i , respectively, and the knapsack capacity is v . Hence, this is also the solution to (16) when $\tilde{st}(x) = 0$. We are left with the case of $\tilde{st}(x) > 0$ to prove.

Suppose the optimal solution is x^* where $\tilde{st}(x^*) > 0$, and the last item inserted into the knapsack is z^* . But, if we decrease the value of $x_{z^*}^*$ by ε while maintaining $\tilde{st}(x^*) > 0$, the value of \tilde{G}° changes by an amount of

$(1 - \sum_{i \in \mathcal{K}} p_i - p_{z^*}) r_{z^*} \varepsilon \geq 0$. Thus, we have obtained a better or an equally good solution. We can iterate to obtain an even better (or equally good) solution. At the limit of $\tilde{st}(x^*) \rightarrow 0$, we can apply Dantzig's Theorem. \square

Since the SKP solution space is a subset of the linear SKP solution space, a tight upper bound on G° is given by,

$$U_{G^\circ} = \tilde{G}^\circ(\bar{x}) = \sum_{i=1}^{\tilde{z}-1} p_i r_i + \left(v - \sum_{i=1}^{\tilde{z}-1} r_i \right) p_{\tilde{z}}. \quad (17)$$

5.3 An Algorithm for Exact Solution

Our algorithm for the exact solution of the SKP is shown in Fig. 6. We shall refer to it simply as the *SKP algorithm*. It is based on Horowitz-Sahni's algorithm for KP [4].

The SKP algorithm assumes that the items are sorted according to (15). It performs *forward moves* and *backtracking moves*. A forward move consists of inserting as many consecutive items as possible to raise the value of G° . The following recursive formula is used to calculate G° incrementally.

$$\begin{aligned} G^\circ(\langle \rangle) &= 0 \\ G^\circ(\mathcal{K} \langle z \rangle) &= G^\circ(\mathcal{K}) + p_z r_z - \sum_{i \notin \mathcal{K}} p_i \text{st}(\mathcal{K} \langle z \rangle). \end{aligned}$$

When an item causes G° to decrease, it is excluded and the upper bound of the currently constructed solution is computed. If the upper bound is lower than the value of the best solution so far, the algorithm backtracks; otherwise, it performs a deeper forward move. When the knapsack stretches to accommodate an item (or when no item remains), the current solution is complete with the remaining items deemed excluded. A backtracking move consists of removing the last inserted item from the solution.

6 MAXIMIZING ACCESS IMPROVEMENT

In Section 4, we derive a formula for access improvement given in (12). Any prefetch strategy based solely on this formula will be a greedy one in the sense that it tries to optimize the performance of the next single access without considering the effect of its decision further into the future. There are three negative aspects to this greediness. First, we will prefetch only candidates for the next request (branch prefetch). If candidates are exhausted before the viewing time is up, then the residual viewing time, which is an important asset as far as prefetching is concerned, is simply wasted. Even when the viewing time is completely utilized, there might be items more worthy of being prefetched but do not appear within the one-access window that we are considering. Second, the selection of victims for cache replacement also suffers from the shortsightedness of the one-access window. Third, the stretch time may intrude into the next viewing time and, thus, reduce the asset for the next prefetch.

In this section, we will discuss how to maximize the access improvement. We are not merely maximizing G of (12), which is only an approximation using a one-access branch analysis, but we also deal with the negative aspects mentioned above. In particular, we will investigate if the use of stretch time actually helps in the long run or perhaps the possible intrusion into the next viewing time prevails.

```

input:  $n, p, r, v$   output:  $\mathcal{F}$ 

1.  $j := 1$  (* initialise *)
    $x := 0, \dots, 0$  (* best item selectors *)
    $G := 0$  (*  $G^o$  (best solution) *)
    $\hat{x} := 0, \dots, 0$  (* current item selectors *)
    $\hat{G} := 0$  (*  $G^o$  (current solution) *)
    $\hat{v} := v$  (* current residual capacity ( $v - \sum_{i=1}^n r_i \hat{x}_i$ ) *)
    $p_{n+1} := 0$ 
    $r_{n+1} := \infty$ 

2. Find  $\tilde{z} = \min \left\{ k : \sum_{i=j}^k r_i > \hat{v} \right\}$  (* compute upper bound *)
    $U := \sum_{i=j}^{\tilde{z}-1} p_i r_i + (\hat{v} - \sum_{i=j}^{\tilde{z}-1} r_i) p_{\tilde{z}}$ 
   if  $G \geq \hat{G} + U$  then goto 5 endif

3. while  $j \leq n$  and  $\hat{v} > 0$  do (* perform a forward step *)
    $\delta := p_j r_j - \sum_{i=j}^n p_i \max \{0, r_j - \hat{v}\}$ 
   if  $\delta \leq 0$  then
      $\hat{x}_j := 0$ 
      $j := j + 1$ 
     if  $j < n$  then goto 2 endif
   else
      $\hat{v} := \hat{v} - r_j$ 
      $\hat{G} := \hat{G} + \delta$ 
      $\hat{x}_j := 1$ 
      $j := j + 1$ 
   endif
endwhile

4. if  $\hat{G} > G$  then (* update the best solution *)
    $G := \hat{G}$ 
    $x := \hat{x}$ 
endif

5. Find  $k = \max \{i < j : \hat{x}_i = 1\}$  (* backtrack *)
   if no such  $k$  then goto 6 endif
    $\hat{x}_k := 0$ 
    $\hat{v} := \hat{v} + r_k$ 
    $\delta := p_k r_k - \sum_{i=k}^n p_i \max \{0, r_k - \hat{v}\}$ 
    $\hat{G} := \hat{G} - \delta$ 
    $j := k + 1$ 
   goto 2

6.  $\mathcal{F} := \langle i : x_i = 1 \rangle$  (* final solution *)

```

Fig. 6. The SKP algorithm.

We will also discuss a cache replacement policy that removes some of the short-sightedness in choosing victims.

For the purpose of caching, we assume that items have equal sizes.

6.1 Effect of Stretch Time

The use of stretch time allows more items to be prefetched. However, the stretch time may intrude into the next viewing time, reducing the asset for the next prefetch. We want to know to which side the balance tips.

To investigate the effect of the use of stretch time, we have performed “prefetch only” simulation in which the cache is used only for prefetching items. Once a request is

satisfied, the cache is flushed out. (Alternatively, we can assume that the requests come from a highly transient process.)

The simulation consists of running 50,000 iterations through the following steps:

1. Generate n, p, r and v randomly,
2. Prefetch,
3. Generate a random request,
4. Calculate access time, and
5. Output v and t .

The values for p are generated using two different methods: *skewy* method and *flat* method. In the *skewy* method, we

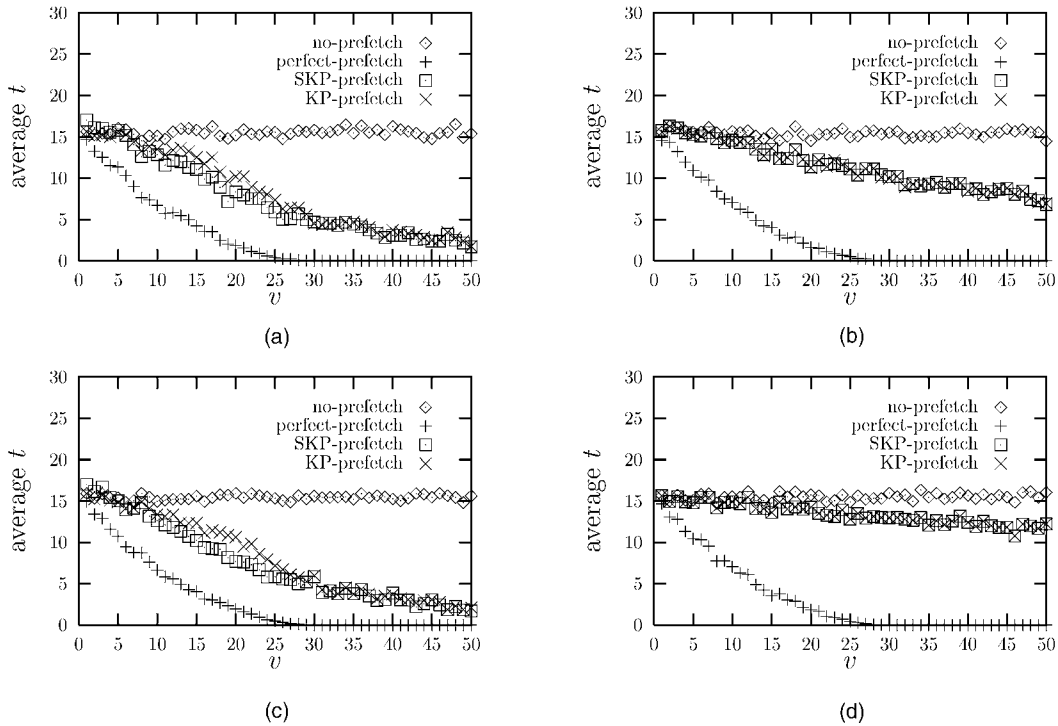


Fig. 7. Performance of prefetch. Each plot is obtained by running the “prefetch only” simulation for 50,000 iterations. The value for v ranges from 1 to 100 (though the plot is clipped at $v = 50$) and r ranges from 1 to 30.

start with a pie of 1.0. A random portion of the pie is allocated to an item and a random portion of the remainder is allocated to another item. This is repeated until one item remains which takes whatever is left of the pie. In the flat method, each item is given a random number taken from the same distribution. The numbers are then normalized so that they total 1. The skewy method is used to generate a situation where the next request is highly predictable and the top candidate for the next access strongly dominates. The use of the flat method results in a less predictable situation, where the most likely item to get accessed is not very far ahead from the second candidate.

Four different prefetch methods are employed in the simulation: *SKP-prefetch*, *KP-prefetch*, *perfect-prefetch*, and *no-prefetch*. The SKP-prefetch and the KP-prefetch use, respectively, the SKP solution and the KP solution to select items for prefetch. We already discussed how to find the SKP solution in Section 5. The perfect-prefetch always prefetches the correct item. For SKP-prefetch, a nonzero stretch time may reduce the viewing time in the next iteration.

Fig. 7 shows the average access time against v . In Figs. 7a and 7c, for which the skewy method is used, the performance of the SKP-prefetch is slightly better than that of the KP-prefetch. The exception is when v is small (approximately less than 5) where the SKP-prefetch performs worse than no prefetch. Increasing the number of items from 10 to 25 has little effect as the probabilities of additional items are vanishingly small.

In Figs. 7b and 7d, for which the flat method is used for generating the probabilities p , the performances of the SKP-prefetch and the KP-prefetch are almost the same. Indeed, for the case of $n = 25$, if there is any difference at all, it is too small to perceive from the graph. Increasing the number of items from 10 to 25 has the effect of increasing the average

access time. The increase is expected; in the extreme case when $n = \infty$ and there are no clearly dominating items, any speculative prefetch will be in vain.

The evidence that we get from our simulation results is in favor of using stretch time as it allows fuller utilization of the viewing time. However, there is always room for improvement. The SKP algorithm maximizes access improvement without any constraints. It inserts an item if it increases G° , even by an insignificant amount and even if it results in a large stretch time. Consider the problem with these parameters: $n = 2, p_1 = 0.99, p_2 = 0.01, r_1 = 5, r_2 = 101, v = 6$. The SKP solution to this problem is $\langle 1, 2 \rangle$. This is only 2 percent better (in terms of the value of G°) than the next best solution, $\langle 1 \rangle$. However, the stretch time, which is 100 units, is a liability in 99 percent of the cases. The SKP-prefetch only looks ahead one access and does not recognize this liability. This causes the performance deficiency that we see in Figs. 7a and 7c for small values of v .

6.2 Integrated Prefetch and Cache

We now consider the combined problem of choosing items to prefetch into and items to remove from the cache. We already derived in (12) the access improvement, $G(\mathcal{F}, \mathcal{D})$, when \mathcal{F} is prefetched and \mathcal{D} ejected. So, to begin with, we will solve the following problem:

Find $\langle \mathcal{F}, \mathcal{D} \rangle$ to maximize

$$G(\mathcal{F}, \mathcal{D}) = G^\circ(\mathcal{F}) - \left(\sum_{i \in \mathcal{D}} p_i r_i - \sum_{i \in \mathcal{C} \setminus \mathcal{D}} p_i st(\mathcal{F}) \right)$$

subject to $|\mathcal{D}| = |\mathcal{F}|$.

(18)

The condition $|\mathcal{D}| = |\mathcal{F}|$ arises from our assumption that items have equal sizes. A free block in the cache is considered to contain an item with zero access probability. So, the number of ejected items must always be equal to the number of incoming items.

The complexity of the search space for the problem in (18) is the same as that of the SKP. However, we have not been able to come up with a bounding technique to subdue its combinatorial explosion. So, we will settle for a suboptimal solution.

The expression for G suggests the following method. First, using the SKP algorithm, find \mathcal{F} to maximize $G^\circ(\mathcal{F})$. Then, find \mathcal{D} to minimize $\sum_{i \in \mathcal{D}} p_i r_i - \sum_{i \in \mathcal{C} \setminus \mathcal{D}} p_i st(\mathcal{F})$, which we will refer to as anti- g . The minimization problem can simply be solved by sorting \mathcal{C} in the ascending order of $p_i(r_i + st(\mathcal{F}))$ and taking the first $|\mathcal{F}|$ elements. Effectively, we are splitting the problem in (18) into two totally separate subproblems—what to prefetch and what to eject. This nonintegrated approach has the following problem: There may exist $f \in \mathcal{F}$ and $d \in \mathcal{D}$ such that the contribution of f to $G^\circ(\mathcal{F})$ is less than the contribution of d to anti- g . In this case, $\langle \mathcal{F} \setminus \{f\}, \mathcal{D} \setminus \{d\} \rangle$ is a better solution than $\langle \mathcal{F}, \mathcal{D} \rangle$. So, we must include an arbitration step to prevent ejection of an item from the cache by a less-worthy replacement.

A perfect arbitration must consider $st(\mathcal{F})$. The larger the value of $st(\mathcal{F})$, the more difficult it should be to eject items from the cache. However, if an item, having insufficient value for cache occupancy, is eliminated from \mathcal{F} , then $st(\mathcal{F})$ will change. The arbitration that causes the elimination might not be valid anymore because, as a result of this change, the price for cache occupancy may also change. To simplify matters, we assume, only for the purpose of arbitration, that $st(\mathcal{F}) = 0$ so that item $f \in \mathcal{F}$ contributes $p_f r_f$ to $G^\circ(\mathcal{F})$ and item $d \in \mathcal{D}$ contributes $p_d r_d$ to anti- g . Item f can only be prefetched if it can find a victim d such that $p_d r_d = \min_{i \in \mathcal{C}} \{p_i r_i\}$ and $p_f r_f > p_d r_d$. We call this *pr-arbitration*.

The *pr-arbitration* protects cached items that contribute significantly to the performance of the next access, but it is blind to accesses further into the future. Cached items that do not appear in the list of candidates for the next access have the same *pr* value, which is zero. To choose from among potential victims that have the same *pr* value, we employ a subarbitration. For this purpose, we define, for each item, a value called *delay-saving profit* as $freq(i) \times r_i$, where $freq(i)$ is the frequency of accesses to item i . This formula is a simplified form of the one used by WATCHMAN cache [15] and its web-related spawn [16]. When subarbitration is required, we choose the item with the lowest delay-saving profit. We call this *DS-arbitration*.

While everything else is meticulously derived to solve the problem in (18), the inclusion of the subarbitration seems ad hoc. However, we have a simple justification for this. The current prefetch-relevant context is known and this leads us to the *pr-arbitrated* SKP algorithm. However, the future contexts are probabilistic. The further we look into the future, the less certain we get about them. To simplify matters, we assume that the future contexts are

completely unknown. The access probability of item i when the context is unknown is

$$\approx \frac{freq(i)}{\sum_k freq(k)}.$$

So the total contribution to the cache value (given unknown context) is

$$\sum_{i \in \mathcal{C}} \frac{freq(i) \times r_i}{\sum_k freq(k)}.$$

Since $\sum_k freq(k)$ is a constant with respect to \mathcal{C} , our problem is to maximize $\sum_{i \in \mathcal{C}} freq(i) \times r_i$.

6.3 Simulation Results

We have carried out Monte Carlo simulations to see how SKP with arbitration performs. A 1,000-state Markov source is used for this purpose. When going to state i , the Markov source generates a request for item i and, after the request is served, it waits for the duration of v_i seconds, where $30 \leq v_i \leq 300$, before changing to another state. The state transition matrix is constructed such that there are 10 to 20 possible transitions from any state. The retrieval time for item i is $s_i/\text{bandwidth} + \text{latency}$, where s_i is the size of item i . The values of s_i are taken from a uniform integer distribution ranging from 10 Kbyte to 1 Mbyte. Cache capacity is in terms of number of items. This means that, for the purpose of caching, each item fits exactly into one cache slot.

We simulate five different prefetch-cache policies:

1. *No + pr*: No prefetch is performed and *pr*-arbitration is used to select cache victims,
2. *KP + pr*: KP solution is used with *pr*-arbitration,
3. *SKP + pr*: SKP solution is used with *pr*-arbitration,
4. *SKP + pr + LFU*: Same as the previous one, with subarbitration using LFU (least frequently used), and
5. *SKP + pr + DS*: Same as the previous one, except DS-arbitration is used instead of LFU.

Fig. 8 shows the results of the simulations for bandwidth values of 5, 10, and 20 kbit/sec and cache sizes ranging from 10 to 100 items. From this figure we can verify that the SKP prefetch performs better than the KP prefetch for the two lower bandwidth values. Adding subarbitration clearly improves results in most cases. We are not surprised that the SKP + *pr* + DS outperforms the other SKP variations. While the *pr*-arbitration protects immediate candidates, the DS-arbitration keeps in cache the items that would otherwise consume too much network time. There are two interesting outcomes in the case of bandwidth = 20 kbit/sec. First, the KP prefetch is leading, but very slightly. Probably this is due to the following reason. The higher the bandwidth, the more items can be prefetched within any viewing time. At this bandwidth, the top candidates will most likely arrive within the viewing time. Any stretch time that results is most likely caused by a candidate with a very small access probability, so small that the stretch time becomes a liability most of the time (see the end of Section 6.1). The other interesting outcome is that the SKP + *pr* + LFU policy performs worse than the SKP + *pr* policy. Obviously,

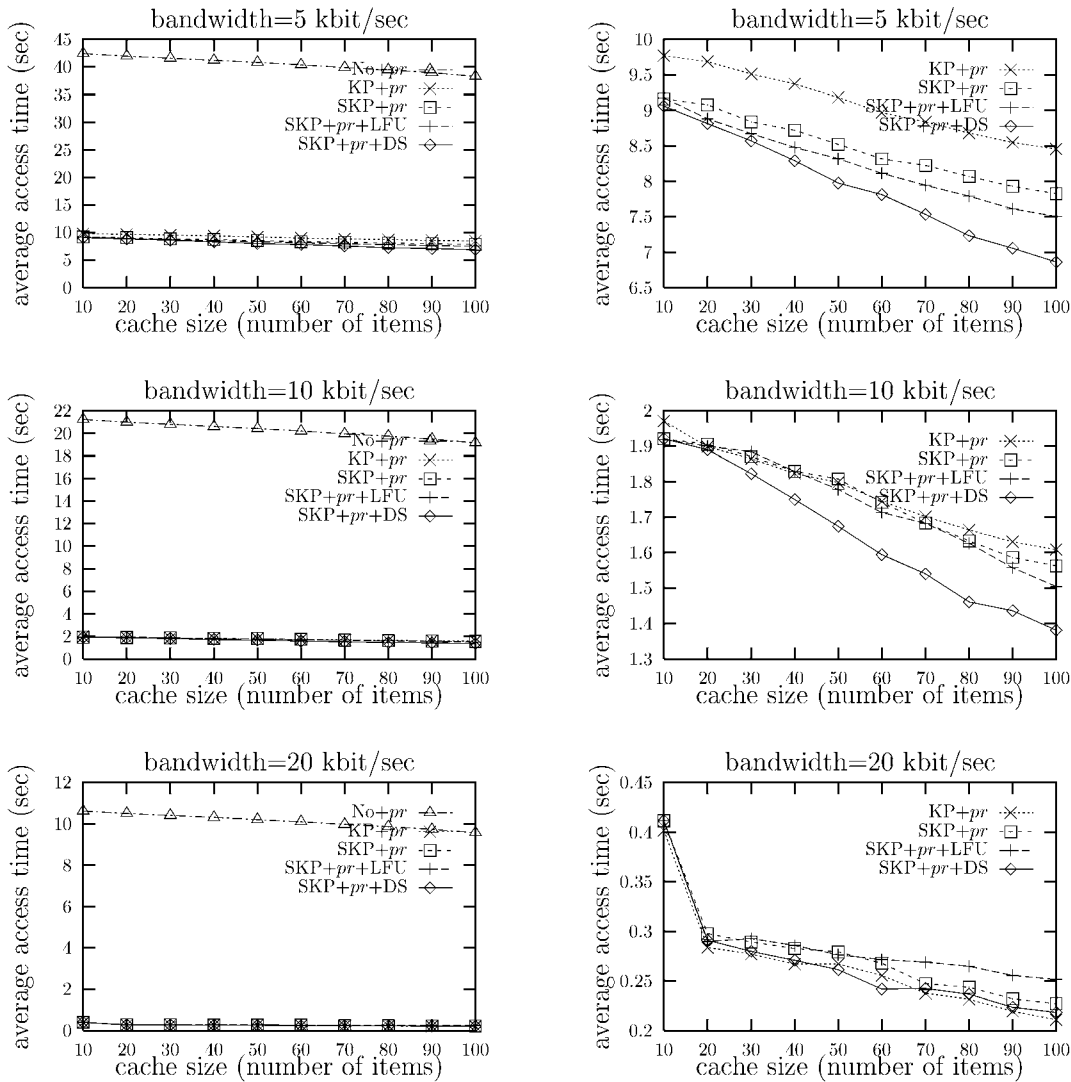


Fig. 8. Comparing different prefetch-cache policies. Each point is obtained by generating 5,000 requests from a Markov source. The latency is zero. The graphs in the left column include plots for the case of no prefetch (No + pr). In the right column, we zoom in to the range covered by the different prefetch-cache policies.

the LFU subarbitration is not reliable because it is missing out an important parameter, namely the retrieval time.

Fig. 9 shows more results for the SKP + pr + DS policy. Results for the No + pr policy are also shown for comparison. We vary bandwidth from 5 to 40 kbit/sec and latency from 0 to 4 seconds. The most important thing to note is that, for the case of no prefetch, doubling the bandwidth reduces the access time approximately by half. However, for the case of prefetch, the access time becomes approximately five times faster. This means that the relative improvement gets more significant for higher bandwidths. If we project the plots backward, halving the bandwidth will make the access time twice as slow for no prefetch, but five times slower for prefetch.

7 CONCLUSIONS AND FURTHER WORK

We have presented a performance model for speculative prefetching, incorporating resources and access prediction. Assuming the existence of an access model and a resource

model to provide necessary knowledge, we analyze the performance of a prefetcher that utilizes this knowledge.

Prefetching mainly benefits a user who accesses relatively fast items. Hence, the provision of hoarding [6], [7] would greatly help, not only for disconnected operation for which it is mainly intended, but also to complement prefetching under low bandwidth operation.

The trade-offs between mainline prefetch and branch prefetch present a challenge for implementation. Mainline prefetch may slightly improve on access time over single prefetch with only nominal extra retrieval cost. Branch prefetch is likely to improve on access time over single prefetch, but with considerably more retrieval cost. Ideally, a prefetcher can adapt its strategy depending on the available resources and the target performance. In a resource-rich environment, the prefetcher can even combine mainline and branch prefetches.

Disregarding retrieval cost, we develop a prefetch algorithm to maximize access improvement. The algorithm uses theoretically proven apparatus to reduce its search

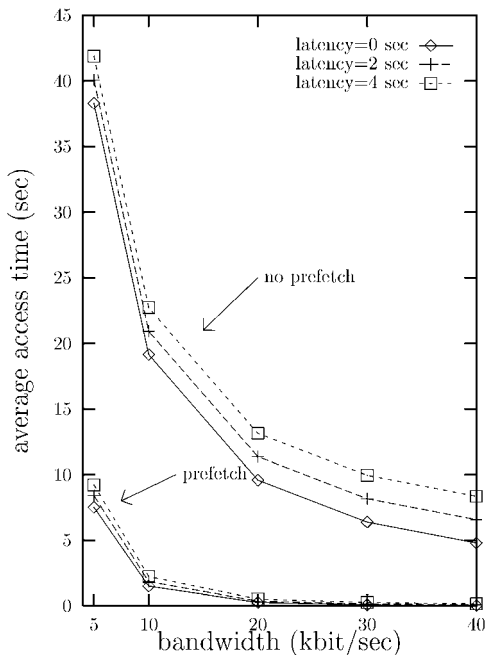


Fig. 9. Performance of SKP + pr + DS for different network conditions. Each point is obtained by generating 5,000 requests from a Markov source. For the no prefetch cases, the No + pr policy is used. For the prefetch cases, the SKP + pr + DS policy is used. The cache size is 100 times.

space. We integrate the prefetch algorithm with cache replacement using a two-stage arbitration. Our algorithm assumes uniform sizes for all items.

The SKP algorithm considers only one access ahead. Obviously, looking ahead deeper will improve the performance. The SKP algorithm with arbitration maximizes access improvement without regard to the increase in the network usage. Even if the most probable items are already in the cache, it will prefetch the lesser candidates if, by doing so, it can improve the expected access time even by an insignificant amount. This may be undesirable when network usage is costly.

In Section 1, we mention that a prefetcher requires three separate models. These are the access model, the performance model, and the resource model. We have assumed a generic (nonspecific) access model and a generic resource model to feed relevant information to the performance model. However, the generic models are not truly generic and, hence, the performance model is not really universal. Further, theoretical work should consider other performance factors for more generality. For example, in a subsequent work [18], we reformulate the performance model to fit an access model that we believe is more appropriate for web access.

An important factor that we have not included in our model is the expected cache lifetime of individual items. If an item goes stale quickly, obviously it should not be given a high priority even if it is accessed frequently. Another issue that we have avoided is the functional priority of each item. For example, an HTML file is more important than a graphics file used for bullets or banners because the document is still readable without the latter, but not without the former. Furthermore, when an HTML file is

available, the browser can start responding by displaying its textual content even before its embedded graphics arrive. The other way round, displaying embedded graphics outside their context, does not make sense. Other possibly relevant factors include access types (read, write, continuous media), client mobility, and provision for batch retrievals.

ACKNOWLEDGMENTS

The work of N.J. Tuah was done under scholarship of the Brunei government through Universiti Brunei Darussalam. The work presented in this paper was carried out during his doctoral study at the Curtin University of Technology in Australia.

REFERENCES

- [1] M. Banâtre, V. Issarny, F. Leleu, and B. Charpiot, "Providing Quality of Service over the Web: A Newspaper-Based Approach," *Computer Networks*, vol. 29, nos. 8-13, pp. 1457-1465, Sept. 1997.
- [2] P. Cao, "Application-Controlled File Caching and Prefetching," PhD thesis, Dept. of Computer Science, Princeton Univ., Jan. 1996.
- [3] G.B. Dantzig, "Discrete Variable Extremum Problems," *Operations Research*, vol. 5, pp. 266-277, 1957.
- [4] E. Horowitz and S. Sahni, "Computing Partitions with Applications to the Knapsack Problem," *J. ACM*, vol. 21, pp. 277-292, 1974.
- [5] Z. Jiang and L. Kleinrock, "An Adaptive Network Prefetch Scheme," *IEEE J. Selected Areas in Comm.*, vol. 16, no. 3, pp. 358-368, Apr. 1998.
- [6] J.J. Kistler, "Disconnected Operation in a Distributed File System," PhD thesis, Carnegie Mellon Univ., May 1993.
- [7] G.H. Kuenning, "Seer: Predictive File Hoarding for Disconnected Mobile Operation," PhD thesis, Univ. of California, Los Angeles, 1997.
- [8] H. Lei and D. Duchamp, "An Analytical Approach to File Prefetching," *Proc. USENIX Ann. Technical Conf.*, Jan. 1997.
- [9] E.P. Markatos and C.E. Chronaki, "A Top-10 Approach to Prefetching on the Web," Technical Report TR 173, ICS-FORTH, Greece, Aug. 1996.
- [10] S. Martello and P. Toth, "An Upper Bound for the Zero-One Knapsack Problem and a Branch and Bound Algorithm," *European J. Operational Research*, vol. 1, pp. 169-175, 1977.
- [11] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementation*. Wiley, 1990.
- [12] V.N. Padmanabhan and J.C. Mogul, "Using Predictive Prefetching to Improve World Wide Web Latency," *ACM SIGCOMM Computer Comm. Rev.*, pp. 22-36, July 1996.
- [13] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *Proc. 15th ACM Symp. Operating System Principles*, pp. 79-95, Dec. 1995.
- [14] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network File System," *Proc. Summer 1985 USENIX Conf.*, pp. 119-130, June 1985.
- [15] P. Scheuermann, J. Shim, and R. Vingralek, "WATCHMAN: A Data Warehouse Intelligent Cache Manager," *Proc. 22nd Very Large Databases (VLDB) Conf.*, 1996.
- [16] P. Scheuermann, J. Shim, and R. Vingralek, "A Case for Delay-Conscious Caching of Web Documents," *Computer Networks*, vol. 29, nos. 8-13, pp. 997-1005, Sept. 1997.
- [17] C.D. Tait, "A File System for Mobile Computing," PhD thesis, Graduate School of Arts and Sciences, Columbia Univ., 1993.
- [18] N.J. Tuah, M. Kumar, and S. Venkatesh, "Performance Modelling of Speculative Prefetching for Compound Requests in Low Bandwidth Networks," *Proc. Third ACM Int'l Workshop Wireless Mobile Multimedia*, pp. 83-92, 2000.
- [19] J.S. Vitter and P. Krishnan, "Optimal Prefetching via Data Compression," *Proc. IEEE 32nd Ann. Symp. Foundation of Computer Science*, pp. 121-130, 1991.
- [20] R.P. Wooster and M. Abrams, "Proxy Caching that Estimates Page Load Delays," *Computer Networks*, vol. 29, nos. 8-13, pp. 977-986, Sept. 1997.



Nor Jaidi Tuah received the BSc degree from Thames Polytechnic, United Kingdom, in 1989. He received the MSc degree from Queen Mary and Westfield College, United Kingdom, in 1990. He obtained the PhD degree from Curtin University of Technology, Western Australia, in 2000. He is currently with the Universiti Brunei Darussalam in the Department of Mathematics. His research interests

include artificial intelligence, functional programming, and parallel/distributed computing.



Mohan Kumar is an associate professor in the Department of Computer Science at the University of Texas at Arlington. From 1992 to 2000, he was a faculty member at the School of Computing, Curtin University of Technology in Western Australia. His research areas include mobile computing, wireless networks, pervasive computing, and parallel and distributed computing. He has published more than 70 articles in refereed journals and conference proceedings in the above areas. He is on the editorial board of

The Computer Journal and co-guest-edited special issues of the *Computer Journal*, *The IEEE Transactions on Computers*, and others. He is a senior member of the IEEE.



Svetha Venkatesh is a professor at the School of Computing at Curtin University of Technology, Perth, Western Australia. Her research is in the areas of active vision, biological-based vision systems, image understanding, and applications of computer vision to image and video database indexing and retrieval. She is the author of approximately 180 research papers in these areas and is a senior member of the IEEE.



Sajal K. Das received the PhD degree in computer science in 1988 from the University of Central Florida, Orlando. Currently, he is a full professor of computer science and engineering and the founding director of the Center for Research in Wireless Mobility and Networking (CReWMan) at the University of Texas at Arlington (UTA). Prior to 1999, he was a professor of computer science at the University of North Texas (UNT), Denton, where he founded the Center for Research in Wireless Computing (CReW) in 1997 and served as the director of the Center for Research in Parallel and Distributed Computing (CRPDC) during 1995-1997. He is a recipient of the UNT Student Association's Honor Professor Award in 1991 and 1997 for best teaching and scholarly research, UNT's Developing Scholars Award in 1996 for outstanding research, and UTA's Outstanding Senior Faculty Research Award in Computer Science in 2001. He has visited numerous universities, research organizations, and industry research labs for collaborative research and invited seminar talks. He was a visiting scientist at the Council of National Research in Pisa, Italy, and Slovak Academy of Sciences in Bratislava, and was also a visiting professor at the Indian Statistical Institute, Calcutta. He is frequently invited as a keynote speaker at international conferences and symposia. His current research interests include resource and mobility management in wireless networks, mobile computing, QoS provisioning and wireless multimedia, mobile Internet, network architectures and protocols, distributed/parallel processing, performance modeling, and simulation. He has published more than 185 research papers in these areas, directed several projects funded by industry and government, and filed four US patents in wireless mobile networks. He received the Best Paper Awards for significant research contributions at the ACM Fifth International Conference on Mobile Computing and Networking (MobiCom'99), the Third ACM International Workshop on Modeling, Analysis, and Simulation of Wireless and Mobile Systems (MSWiM 2000), and the ACM/IEEE International Workshop on Parallel and Distributed Simulation (PADS'97). He serves on the editorial boards of the *Journal of Parallel and Distributed Computing*, *Parallel Processing Letters*, *Journal of Parallel Algorithms and Applications*, and *Computer Networks*. He serves on numerous IEEE and ACM conferences as a technical program committee member, program chair, or general chair. He is a member of the IEEE TCPP Executive Committee and advisory boards of several cutting-edge companies. He is a member of the IEEE and the IEEE Computer Society.

► For more information on this or any computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.