AIDOA: An Adaptive and Energy-Conserving Indexing Method for On-Demand Data Broadcasting Systems

Jiun-Long Huang
Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan, ROC
E-mail: jlhuang@cs.nctu.edu.tw

Abstract

Since only a modest improvement in battery lifetime is expected in the next few years, energy conservation is raised as a key factor of the design of mobile devices. In view of this, we propose in this paper an energy-conserving on-demand data broadcasting system employing data indexing technique. Different from the prior work, power consumption of turning on and turning off the wireless network interfaces is considered. In addition, we also employ server cache to reduce the effect of the time to retrieve data items from the corresponding data servers. Specifically, we first analyze the access time and tuning time of data requests and propose algorithm AIDOA to adjust the degree of buckets according to system workload. Several experiments are then conducted to evaluate the performance of algorithm AIDOA. Experimental results show that algorithm AIDOA is able to greatly reduce power consumption at the cost of a slight increase in average access time and adjust the index and data organization dynamically to adapt to change of system workload.

Keywords: data indexing, on-demand data broadcasting, energy conservation, mobile information system

1 Introduction

Owing to the constraints resulting from power-limited mobile devices and low-bandwidth wireless networks, designing a power conserving mobile information system with high scalability and high bandwidth utilization becomes an important research issue, and hence attracts a significant amount of research attention. In recent years, data broadcasting is proposed to address such challenge and has been recognized as a promising data dissemination technique in mobile computing environments [1][4][5][10][11]. Most research works on data broadcasting focus on generating a proper broadcast program or designing scheduling algorithms to minimize the *average access time*, which is defined as the average time elapsed from the moment a client issues a query to the point the desired data item is read.

As shown in [17][19], only a modest improvement (about 20% $\sim$ 30%) in battery lifetime is expected in the next few years. Hence, energy conservation is raised as a key factor of the design of mobile devices. Consider a Nokia 5510 which supports AAC and MP3 playing. Compared to the power consumed on music playing, the wireless network interface (abbreviated as WNI) consumes much more energy (as much as 70% of the total power in Nokia 5510) [22]. Hence, reducing the power consumption on WNIs is an effectively means to reduce the overall power consumption. Most devices can operate in two modes: *active* mode and *doze* mode. Many studies show that the power consumed in active mode is much higher than that consumed in doze mode. For example, a typical wireless PC card, ORiNOCO, consumes 60 mW during the doze mode and 805 $\sim$ 1400 mW during the active mode [19]. As a consequence, in order to reduce power consumption, the mobile devices should stay in doze mode as long as possible.

To evaluate the effect of data indexing algorithms on energy conservation, *tuning time*, which is defined as the time that a mobile device operates in active mode in order to retrieve a data item, is introduced in [12]. Since employing data indexing will unavoidably introduce some overhead in access time, data indexing algorithms should reduce tuning time as much as possible at the cost of producing an acceptable increase in access time. Since the size of an index item is usually much smaller than that of a data item, the increment in access time is usually small. As a result, many research works study the design of data indexing algorithms in push-based data broadcasting environments [20][21]. However, most studies on on-demand data broadcasting focus on the design of scheduling algorithms [2][5] to reduce average access time, and only few of them consider the employment of data indexing in on-demand data broadcasting

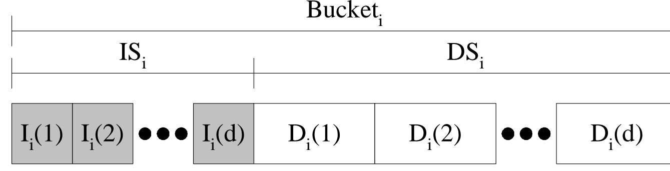


Figure 1: Index structure

environments [13] to reduce average tuning time.

In [13], Lee et al. proposed an indexing algorithm for on-demand data broadcast systems. As shown in Figure 1, the proposed broadcast program is made up of a series of buckets and each bucket consists of one index segment and one data segment. A data segment contains a series of data items, while an index segment consists of the index items of the data items in the corresponding data segment. For a bucket, the number of data items in the corresponding data segment is called the *degree* of the bucket. The information in an index item, say $I_i(1)$, consists of the identifier of the corresponding data item $D_i(1)$, the data size of $D_i(1)$ and the time that $D_i(1)$ in bucket i will be broadcast on the broadcast channel. In addition, by the information in the current index segment, a mobile device is able to determine the broadcast time of the index segment of the next bucket.

Although inserting index items into the broadcast program is able to significantly reduce the average tuning time at the cost of a slight increase in average access time [13], however, the proposed data indexing method proposed in [13] has the following drawbacks:

- *Does not consider power consumption of turning on and turning off the WNIs.*

  As pointed out in [18], turning on and turning off the WNIs consume some time and energy, and the transition times of a WNI from active mode to doze mode and from doze mode to active mode are both on the order of tens milliseconds. Consider two organizations of index and data items shown in Figure 2.[1] Suppose that a mobile device tunes to the broadcast channel at time t_{Start} and finishes the retrieval of the desired data item at time t_{End}. Without considering power consumption of turning on and turning off the WNI, the power consumptions of organization one and organization two are equal. However, when power consumption of turning on and turning off the WNIs is considered, organization two outperforms organization one.

[1] The descriptions of symbols ‘A’, ‘D’, ‘F’ or ‘N’ will be given in Table 1 in Section 3.2.

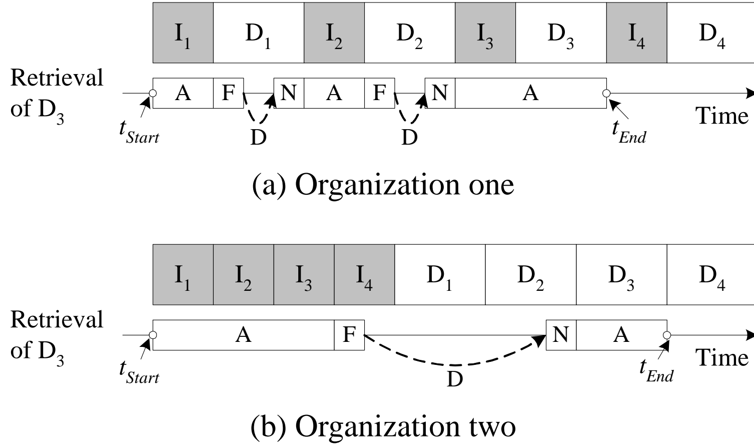


Figure 2: Example organizations of index and data items

Therefore, we argue that the design of an energy-conserving data indexing method should take power consumption of turning on and turning off the WNIs into account to obtain precise power consumption estimation. To the best of our knowledge, there is no prior work on data indexing in on-demanding broadcast considering power consumption of turning on and turning off the WNIs, thereby distinguishing our paper from others.

- *Does not consider the data fetch time*

  Most studies on indexing in on-demand data broadcasting are under the premise that all data items are *immediately* available for a data broadcasting system [13]. However, as pointed out in [6], the data fetch time cannot be neglected since it is infeasible to store all data items in the local cache of the system. Hence, the traditional data broadcasting systems [5] may not perform well. As a consequence, we argue that the indexing algorithm in on-demand data broadcasting should also consider the data fetch time in order to attain higher efficiency.

- *Does not adapt to change of system workload*

  In mobile computing environments, schemes with static degree may not be able to adapt to change of system workload. Such phenomenon shows the necessity of designing an adaptive algorithm to dynamically adjust the degree of buckets to adapt to the change of system workload. To the best of our knowledge, all prior works on data indexing in on-demanding broadcast employ static degree and none of them is able to adapt to change of system workload.

In view of this, we propose in this paper an energy-conserving on-demand data broadcasting system by employing the data indexing technique. Different from the prior work on data indexing on on-demand data broadcasting, power consumption of turning on and turning off the WNIs is considered. Specifically, we first analyze the access time and tuning time of data requests and propose algorithm AIDOA to adjust the degree of buckets according to system workload. In essence, algorithm AIDOA consists of two phases, statistics collection phase and adjustment phase, and switches back and forth between these two phases periodically. The system collects some statistic information of all served data requests in statistics collection phase, and the collected information is used to adjust the degree of buckets in adjustment phase according to the derived analytical results. In addition, we employ server cache to eliminate the performance degradation caused by the data fetch time. We also propose a program generation algorithm and a cache replacement policy to cooperate with algorithm AIDOA. Several experiments are then conducted to evaluate the performance of algorithm AIDOA. Experimental results show that due to the dynamic adjustment on degree of buckets, scheme using algorithm AIDOA outperforms other schemes with static degree in most cases.

The rest of this paper is organized as follows. Section 2 describes the proposed system architecture and the power consumption model used in this paper. Section 3 shows the analytical model of the proposed system architecture. Based on the analytical model, we propose algorithm AIDOA in Section 4. In addition, the companion program generation algorithm and cache replacement policy are proposed in Section 5. Experimental results are shown in Section 6 to evaluate the performance of algorithm AIDOA, and finally, Section 7 concludes this paper.

2 Preliminaries

2.1 System Architecture

We adopt the index structure proposed in [13] and the adopted index structure is shown in Figure 1. As shown in Figure 3, the proposed system architecture consists of the following components.

- **Scheduler:** The scheduler is in charge of receiving and processing the data requests submitted by mobile devices. After receiving a data request, say Req_i, the scheduler will search the ready queue,

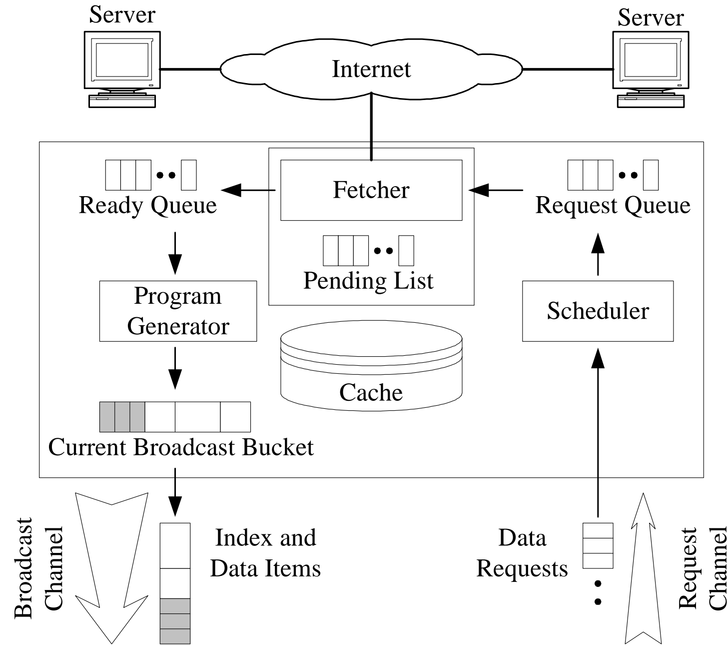


Figure 3: System architecture

the pending list and the request queue sequentially to check whether there exists a data request, say Req_j, with the same required data item as Req_i. When Req_j is in the pending list, the scheduler merges Req_i into Req_j. When Req_j is in the ready queue (respectively, the request queue), the scheduler will merges Req_i into Req_j and updates the priorities of all data items in the ready queue (respectively, the request queue) according to the employed scheduling algorithm such as FIFO, LWF, RxW and so on. Otherwise, when Req_j does not exist, the scheduler will insert Req_i into the request queue and update the priorities of all data items in the request queue according to the employed scheduling algorithm.

- **Fetcher:** The fetcher repeatedly retrieves the data request with highest priority from the request queue, and fetches the required data item from the corresponding data server via Internet. Cache is employed to reduce the performance degradation caused by the data item fetch time. To fetch a data item, the fetcher first checks whether the required data item is cached in the local cache. If yes, the fetcher will mark the cached data item as LOCKED and insert the data request into the ready queue. Then, the fetcher will retrieve the data request with highest priority from the request queue and repeat the above procedure.

Otherwise, when the desired data item is not cached, the fetcher will submit a data request message to the data server of the required data item and insert the data request into the pending list. Then, the fetcher will check the number of pending data requests and will stop if the number of pending data requests is equal to a predetermined threshold. Otherwise, the fetcher will repeat the above procedure until the number of pending data requests is equal to a predetermined threshold or the request queue is empty.

When a data server responds with a data item, the fetcher will retrieve the corresponding data request from the pending list and insert the data request into the ready queue. In addition, the fetcher will insert the received data item into the cache. Several cached data items may be replaced by the employed replacement policy when the free space of the cache is not enough to store the received data item.

- **Program generator:**

  The program generator employs a program generation algorithm to compose all buckets of broadcast programs. After a bucket is generated, the index and data items in the current bucket are broadcast sequentially. The program generator will start to compose another bucket after all index items and data items in the current bucket have been broadcast.

2.2 Power Consumption Model

Denote the time for a mobile device to switch the WNI from active mode to doze mode as T_{On} and the time to switch the WNI from doze mode to active mode as T_{Off}. To evaluate the power consumption of turning on and turning off the WNIs, we assume that the power consumption of a mobile device spending in time intervals T_{On} (respectively, T_{Off}) is equal to that of a mobile device staying in active mode for time $\alpha_1 \times T_{On}$ (respectively, time $\alpha_2 \times T_{Off}$). Similar to [22], the values of α_1 and α_2 can be obtained by profiling.

Denote the traditional (i.e., without considering the turning-on and turning-off time of WNIs) average tuning time of a data request as T_{Tuning}. To evaluate the overall power consumption, we define the *effective tuning time* of a data request as $T_{Tuning}^{Eff.} = T_{Tuning} + n_1 \times \alpha_1 \times T_{On} + n_2 \times \alpha_2 \times T_{Off}$, where n_1 and n_2 are the numbers of times of turning on and turning off the WNI, respectively, and T_{Tuning} is the traditional tuning

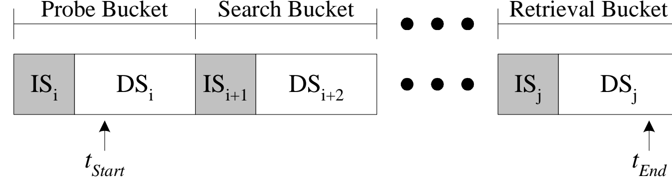


Figure 4: Categories of buckets

time. To ease the presentation, we use the term tuning time to represent effective tuning time, and assume $\alpha_1 = \alpha_2 = 1$ in the rest of this paper.

3 Analytical Model

3.1 Client Access Protocol

After submitting a data request, a mobile client will retrieve the desired data item according to the employed client access protocol. We adopt the client access protocol described in [20], and the protocol consists of the following phases.

- *Initial probe phase:* After submitting a data request, the mobile device tunes to the broadcast channel and listens on the broadcast channel to wait for the appearance of an index segment.
- *Index search phase:* The mobile device enters index search phase after retrieving an index segment. In index search phase, the mobile device determines whether the desired data item will be broadcast in the corresponding data segment. If not, the mobile device will switch to doze mode and then switch back to active mode when the next index segment is broadcast. Otherwise, the mobile device will enter data retrieval phase.
- *Data retrieval phase:* If the desired data item will be broadcast in the current data segment, the mobile device will retrieve the time that the desired data item will be broadcast from the current index segment and switch to doze mode. Then, when the desired data item is broadcast, the mobile device will switch back to active mode and retrieve the desired data item.

Consider the example shown in Figure 4 that a mobile device submits a data request. Let t_{Start} be the time that the mobile device starts to listen on the broadcast channel after submitting the data request, and

t_{End} be the time that the mobile device receives the desired data item. According to the employed client access protocol, the buckets within the time interval from t_{Start} to t_{End} can be divided into the following three categories:

- *Probe bucket:* The bucket which t_{Start} lies on is called the probe bucket. In Figure 4, $Bucket(i)$ is the probe bucket. There is only one probe bucket for each data request.
- *Search bucket:* The bucket whose index segment is retrieved by the mobile device and whose data segment is skipped by the mobile device is called search bucket. In Figure 4, $Bucket(i+1)$, $Bucket(i+2)$, $\cdots$, $Bucket(j-1)$ are all search buckets. For a data request, there may be zero, one or multiple search bucket(s).
- *Retrieval bucket:* The bucket which t_{End} lies on is called the probe bucket. That is, retrieval bucket is the bucket where the mobile device retrieves the desired data item. In Figure 4, $Bucket(j)$ is the retrieval bucket. For each data request, there is only one probe bucket. In addition, the probe bucket and the retrieval bucket of a data request may be the same or different.

3.2 Derivations of Access Time and Tuning Time

To facilitate the following derivations, we have the following assumptions:

- All data items are of equal size S_D.
- The time to broadcast a data item (i.e., $\frac{S_D}{B}$) is larger than $T_{On} + T_{Off}$

Note that both assumptions are not the limitations of algorithm ADIOA and are made only to ease the derivations in Section 3 and Section 4. Hence, they will be relaxed in Section 5 and Section 6.

In $Bucket(i)$, denote the moment that the mobile device starts to turn on and turn off the WNI as $t_{WakeUp}(i)$ and $t_{Sleep}(i)$, respectively. In addition, we also denote that the starting time and the ending time of $Bucket(i)$ as $Bucket(i).Start$ and $Bucket(i).End$, respectively. For a data request, we also partition the time interval from t_{Start} to t_{End} into several segments and each segment is marked as ‘A’, ‘D’, ‘F’ or ‘N’. The descriptions of these four symbols are given in Table 1.

According to the relationship of the probe and retrieval buckets, a data request may be belonging to one of the following two types.

Symbol	Description
A	The mobile device is in active mode
D	The mobile device is in doze mode
F	The mobile device is turning off its WNI
N	The mobile device is turning on its WNI

Table 1: The symbols of time frames

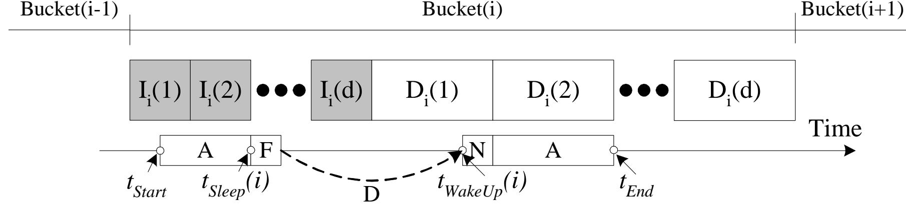


Figure 5: A probe bucket in a Type I data request

3.2.1 Type I: The probe and retrieval buckets are the same

As shown in Figure 5, in a Type I data request, t_{Start} and t_{End} are within the same bucket. In addition, according to the employed client access protocol, t_{Start} must be located in the index segment. Otherwise, t_{Start} and t_{End} will not be in the same bucket, and such result conflicts with the definition of Type I data requests. In order to minimize power consumption, $t_{Sleep}(i)$ is determined as the moment that the mobile device has finished the retrieval of the corresponding index item of the desired data item, and $t_{WakeUp}(i)$ is determined as the moment that the mobile device has to start to turn on the WNI in order to retrieve the desired data item.

We observe from Figure 5 that one Type I data request will increase the aggregate access time of all data requests by $t_{End} - t_{Start}$. On the other hand, the contribution of a Type I data request on the aggregate tuning time of all data requests is determined by the length of the time interval $(t_{Sleep}(i), t_{WakeUp}(i))$. When $t_{WakeUp}(i) - t_{Sleep}(i) > T_{Off}$, the data request will increase aggregate tuning time by

$$\begin{aligned} & t_{Sleep}(i) - t_{Start} + t_{End} - t_{WakeUp}(i) + T_{On} \\ = \quad & t_{Sleep}(i) - t_{Start} + \frac{S_D}{B} + T_{On} + T_{Off}. \end{aligned}$$

Otherwise, when $t_{WakeUp}(i) - t_{Sleep}(i) \leq T_{Off}$ (i.e., the mobile must start to turn on the WNI before the WNI has been turned off), the time interval $(t_{Sleep}(i), t_{WakeUp}(i))$ is too short to turn on and then turn off

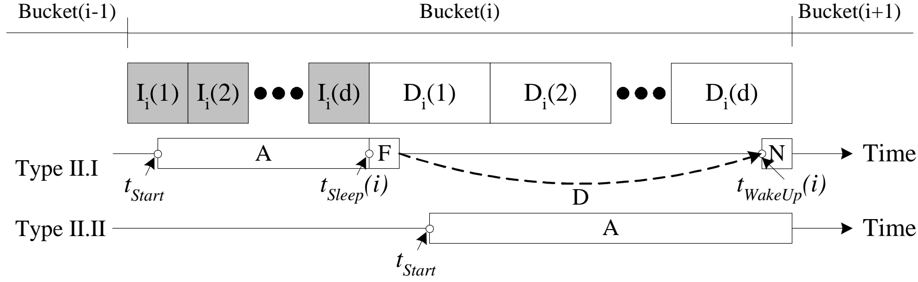


Figure 6: Probe buckets in a Type II.I and a Type II.II data requests

the WNI. Hence, the data request will increase aggregate tuning time by $t_{End} - t_{Start}$.

3.2.2 Type II: The probe and retrieval buckets are different

The time interval (t_{Start}, t_{End}) of a Type II data request consists of one probe bucket, zero, one or multiple search bucket(s) and one retrieval bucket. Next, we will derive the contributions of the probe bucket, the search buckets and the retrieval bucket of a Type II data request, separately, on the aggregate access time and aggregate tuning time of all data requests.

Probe bucket Consider the example shown in Figure 6. According to the location of t_{Start}, Type II data requests can be divided into the following two subtypes.

Type II.I: t_{Start} is in the index segment.

Consider a Type II.I data request. Since the desired data item is not in the probe bucket (i.e, $Bucket(i)$),the probe bucket of a Type II.I data request will increase the aggregate access time of all data requests by $Bucket(i+1).Start - t_{Start}$.

On the other hand, to maximize power-saving, the mobile device should start to turn off the WNI after retrieving the latest index item in IS_i, and must turn on the WNI on $Bucket(i+1).Start$ to retrieve the first index item in IS_{i+1}. Hence, $t_{WakeUp}(i)$ is equal to $Bucket(i+1).Start - T_{On}$. As a consequence, a Type II.I data request will increase the aggregate tuning time of all data requests by $t_{Sleep}(i) - t_{Start} + T_{Off} + T_{On}$.

Type II.II: t_{Start} is in the data segment.

When t_{Start} is in the data segment, according to the employed client access protocol, the mobile device has to listen on the broadcast channel to wait for the appearance of the index segment of the next bucket (i.e., IS_{i+1}). Hence, in $Bucket(i)$, the mobile device is in active mode from t_{Start} to $Bucket(i+1).Start$, and

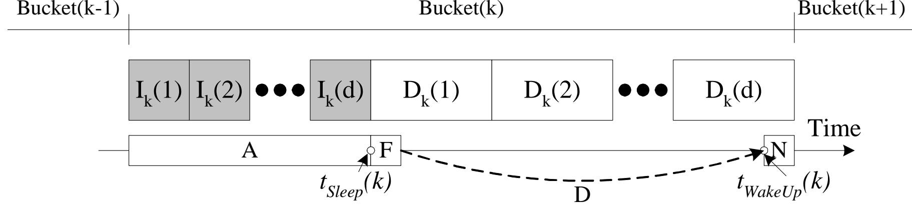


Figure 7: A search bucket in a Type II data request

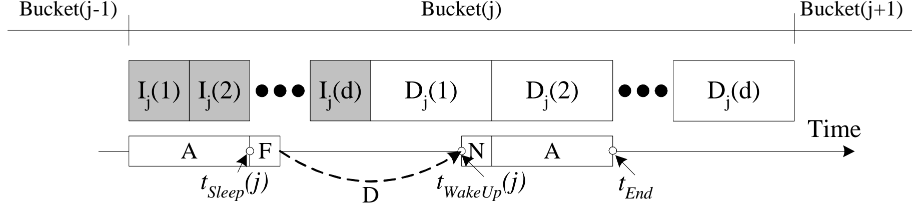


Figure 8: A retrieval bucket in a Type II data request

the contributions of the probe bucket of a Type II.II data request on aggregate access time and aggregate tuning time are both $Bucket(i+1).Start - t_{Start}$.

Search bucket Consider the example shown in Figure 7. In a search bucket, the mobile device operates in active mode to retrieve the index segment and starts to turn off the WNI after retrieving all index items in the index segment. Then, the mobile device has to start to turn on the WNI to ensure that the mobile device just enters active mode on $Bucket(k+1).Start$. Hence, in a search bucket $Bucket(k)$, the contributions on aggregate access time and aggregate tuning time of all data requests are

$$Bucket(k+1).Start - Bucket(k).Start = d \times \frac{S_I + S_D}{B},$$

and

$$t_{Sleep} - Bucket(k).Start + T_{On} + T_{Off} = d \times \frac{S_I}{B} + T_{On} + T_{Off},$$

respectively.

Retrieval bucket Consider the example shown in Figure 8. In the retrieval bucket, the mobile device retrieves the index items in the index segment sequentially until the index item of the desired data item has been retrieved. Then, the mobile starts to turn off the WNI to wait for the appearance of the desired

data item. In order to retrieve the desired data item, the mobile device has to start to turn on the WNI so that the mobile device is able to enter active mode in the moment that the desired data item is just being broadcast. Hence, the retrieval bucket of a Type II data request will increase aggregate access time by $t_{End} - Bucket(j).Start$. In addition, the retrieval bucket of a Type II data request will increase aggregate tuning time by

$$\begin{aligned} & t_{Sleep}(j) - Bucket(j).Start + t_{End} - t_{WakeUp}(j) + T_{On} + T_{Off} \\ = \; & t_{Sleep} - Bucket(k).Start + \frac{S_D}{B} + T_{On} + T_{Off}, \end{aligned}$$

when $t_{WakeUp}(j) - t_{Sleep}(j) > T_{On}$. Otherwise, the data request increases total tuning time by $t_{End} - Bucket(j).Start$.

With the above discussions, for a Type II data request, its contributions on aggregate access time and tuning time are equal to the summations of access time and tuning time, respectively, of its probe bucket, search buckets and retrieval bucket.

4 AIDOA: Adaptive Index and Data Organizing Algorithm

With the analysis in Section 3, we propose in this section algorithm AIDOA (standing for Adaptive Index and Data Organizing Algorithm) to dynamically adjust the degree of buckets according to the system workload. Basically, algorithm AIDOA consists of two phases: statistics collection phase and degree adjustment phase, and switches between statistics collection phase and degree adjustment phase periodically. In statistics collection phase, the server will keep track of information of all data requests and the recorded information will be used to guide the adaptation procedure in the successive execution of adjustment phase.

4.1 Statistics Collection Phase

In each execution of statistics collection phase, the server will collect statistic information of all data requests served in the current execution of statistics collection phase. A data request is *served* when the desired data item has been broadcasted.

Two data structures, $Stat_I$ and $Stat_{II}$, are defined to store the collected information of Type I and Type II (including Type II.I and Type II.II) data requests, respectively. The details of $Stat_I$ and $Stat_{II}$ are as follows.

Details of $Stat_I$

- *ReqNo:* The number of Type I data requests served in the current statistics collection phase
- *AggAT:* Aggregate Access Time of Type I data requests served in the current statistics collection phase
- *AggTT:* Aggregate Tuning Time of Type I data requests served in the current statistics collection phase

Details of $Stat_{II}$

- *ReqNo:* The number of Type II data requests served in the current statistics collection phase
- *AggATP/AggTTP:* Aggregate Access/Tuning Time of Probe buckets of Type II data requests served in the current statistics collection phase
- *AggATS/AggTTS:* Aggregate Access/Tuning Time of Search buckets of Type II data requests served in the current statistics collection phase
- *AggATR/AggTTR:* Aggregate Access/Tuning Time of Retrieval buckets of Type II data requests served in the current statistics collection phase

Each field, except *ReqNo*, of $Stat_I$ and $Stat_{II}$ has an *average* version with new names by replacing prefix *Agg* to *Avg*. For example, the field *AvgAT* of $Stat_I$ indicates the *average* access time of all Type I data requests served in the current statistics collection phase. We also define the structure *Request* to indicate data requests which are merged together. Elements in the request queue, pending list and the ready queue are all instance of structure *Request*. An instance of structure *Request* is said in the server when it is in the request queue, pending list or the ready queue. The details of structure *Request* are as follows.

Details of structure *Request*

- *ReqNo*: The number of data requests which are merged together and are represented by the instance of *Request*

- *AvgTIS*: Average Time In Search buckets of the data requests represented by the instance of *Request*

After receiving a data request, the server first determines the type of this data request. If the data request is belonging to Type I, the server calculates the contributions of the data request on aggregate average and tuning time based on the analysis in Section 3.2.1, and updates $Stat_I$ accordingly. Since being able to be served by the current bucket, a Type I data request will neither be merged into a structure *Request* nor be inserted into the request queue, the ready queue and the pending list.

On the other hand, when the data request is belonging to Type II, the server first checks whether it can be merged into an instance of structure *Request* in the server. If yes, the server updates the fields (i.e., $ReqNo$ and $AvgTIS$) of the instance of structure *Request* accordingly. Otherwise, the server creates a new instance of structure *Request* and inserts the instance into the request queue. Finally, the server calculates the contribution on aggregate access time and tuning time of the probe bucket of the data request according to the derivations in Section 3.2.2, and updates $Stat_{II}$ accordingly.

While an instance of $Request$, say r, is retrieved from the ready requests[2], the server first calculates the average number of search buckets that each data request in r has by

$$AvgSBNo \leftarrow \frac{Bucket(j).start - r.ATIS}{d \times (S_D + S_I)}.$$

The contributions of these search buckets on aggregate access time and aggregate tuning time can be obtained from the derivations in Section 3.2.2, and $Stat_{II}.AggATS$ and $Stat_{II}.AggTTS$ are updated accordingly. The server the calculates the time that the desired data item of r can be retrieved (i.e., t_{End}). Finally, with t_{End}, the server calculates the aggregate contributions of the retrieval buckets of all data requests in r on aggregate access time and tuning time according to the derivations in Section 3.2.2, and updates $Stat_{II}.AggATR$ and $Stat_{II}.AggTTR$ accordingly. The algorithmic form of the procedure to update $Stat_{II}$ when an instance of structure *Request* is served is as follows.

Procedure RequestServed(*Request r*)

1: $Stat_{II}.ReqNo \leftarrow Stat_{II}.ReqNo + r.ReqNo$
2: $AvgSNo \leftarrow \frac{Bucket(j).start - r.ATIS}{d \times (S_D + S_I)}$
3: $Stat_{II}.AggATS \leftarrow Stat_{II}.AggATS + \left(d \times \frac{S_I + S_D}{B}\right) \times AvgSBNo \times r.ReqNo$
4: $Stat_{II}.AggTTS \leftarrow Stat_{II}.AggTTS + \left(d \times \frac{S_I}{B} + T_{On} + T_{Off}\right) \times AvgSBNo \times r.ReqNo$

[2]Readers can refer to Section 5 to see how the system retrieves instances of *Request* from the ready queue.

5: Calculate t_{End} of r
6: $Stat_{II}.AggATR \leftarrow Stat_{II}.AggATR + (t_{End} - Bucket(j).Start) \times r.ReqNo$
7: Let TTR be the tuning time of r in the retrieval bucket
8: $Stat_{II}.AggTTR \leftarrow Stat_{II}.AggTTR + TTR \times r.ReqNo$

4.2 Degree Adjustment Phase

In each execution of degree adjustment phase, the server will adjust the degree (i.e., the value of d) of buckets according to the statistic information collected in the precedent execution of statistics collection phase. Let $T_{Access}(d)$ and $T_{Tuning}(d)$ be the average access time and average tuning time, respectively, when the degree of the broadcast programs is d. For each field, the value of the *average* version is equal to the value of the *aggregate* version divided by the number of data requests. For example, the value of $Stat_I.AvgTT$ is equal to $\frac{Stat_I.AggTT}{Stat_I.ReqNo}$. Then, according to the analysis in Section 3, we have

$$T_{Access}(d) = W_I \times (Stat_I.AvgAT) + W_{II} \times (Stat_{II}.AvgATP + Stat_{II}.AvgATS + Stat_{II}.AvgATR), \text{ and}$$

$$T_{Tuning}(d) = W_I \times (Stat_I.AvgTT) + W_{II} \times (Stat_{II}.AvgTTP + Stat_{II}.AvgTTS + Stat_{II}.AvgTTR),$$

where W_I and W_{II} are the weights of Type I and Type II data requests, respectively. The values of W_I and W_{II} are defined as the ratios of the numbers of Type I and Type II data requests. Hence, we have

$$W_I = \frac{Stat_I.ReqNo}{Stat_I.ReqNo + Stat_{II}.ReqNo}, \text{ and}$$

$$W_{II} = \frac{Stat_{II}.ReqNo}{Stat_I.ReqNo + Stat_{II}.ReqNo}.$$

In addition, $T_{OverAll}(d)$ is employed as the metric of the system performance, and is defined as

$$T_{OverAll}(d) = \beta \times T_{Access}(d) + (1 - \beta) \times T_{Tuning}(d).$$

In the above equation, β is an administrator-specified parameter to reflect the relative importance of average access time (i.e., $T_{Access}(d)$) and average tuning time (i.e., $T_{Tuning}(d)$). Hence, there is no optimal setting of β. The objective of degree adjustment phase is to determine the new value of d to minimize $T_{OverAll}(d)$. However, since globally minimizing $T_{OverAll}(d)$ is difficult, algorithm AIDOA is designed to

find the new value of d, say d_{Next}, where $T_{OverAll}(d_{Next})$ is local minimum. That is, we will find a value of d_{Next} so that $T_{OverAll}(d_{Next})$ is smaller than $T_{OverAll}(d_{Next}+1)$ and $T_{OverAll}(d_{Next}-1)$. Since the exact values of $T_{Access}(d_{Next})$ and $T_{Tuning}(d_{Next})$ when $d_{Next} \neq d_{Curr.}$ cannot be obtained from the collected statistic information, we adopt the following approximation method to estimate $T_{Access}(d_{Next})$ and $T_{Tuning}(d_{Next})$.

Let $Stat_I^{d_{Next}}$ and $Stat_{II}^{d_{Next}}$ be the approximations of the values of structure $Stat_I$ and $Stat_{II}$ when the degree of buckets is d_{Next}. Then, we have the following lemmas:

Lemma 1 $Stat_I^{d_{Next}}.AvgAT$ and $Stat_I^{d_{Next}}.AvgTT$ can be approximated by

$$Stat_I^{d_{Next}}.AvgAT = Stat_I.AvgAT + (d_{Next} - d_{Curr.}) \times \frac{S_I}{B}$$

and

$$Stat_I^{d_{Next}}.AvgTT = Stat_I.AvgTT,$$

respectively.

Lemma 2 $Stat_{II}^{d_{Next}}.AvgATP$ and $Stat_{II}^{d_{Next}}.AvgTTP$ can be approximated by

$$Stat_{II}^{d_{Next}}.AvgATP = \frac{S_I}{S_I + S_D} \times Stat_{II.I}^{d_{Next}}.AvgATP + \frac{S_D}{S_I + S_D} \times Stat_{II.II}^{d_{Next}}.AvgATP,$$

and

$$Stat_{II}^{d_{Next}}.AvgTTP = \frac{S_I}{S_I + S_D} \times Stat_{II.I}^{d_{Next}}.AvgTTP + \frac{S_D}{S_I + S_D} \times Stat_{II.II}^{d_{Next}}.AvgTTP,$$

respectively, where

$$Stat_{II.I}^{d_{Next}}.AvgATP = Stat_{II}.AvgATP + (d_{Next} - d_{Curr.}) \times \left(\frac{S_I}{B} + \frac{S_D}{B}\right),$$

$$Stat_{II.I}^{d_{Next}}.AvgTTP = Stat_{II}.AvgTTP + (d_{Next} - d_{Curr.}) \times \frac{S_I}{B},$$

$$Stat_{II.II}^{d_{Next}}.AvgATP = Stat_{II}.AvgATP + (d_{Next} - d_{Curr.}) \times \frac{S_D}{B}$$

and

$$Stat_{II.II}^{d_{Next}}.AvgTTP = Stat_{II}.AvgTTP + (d_{Next} - d_{Curr.}) \times \frac{S_D}{B}.$$

As mentioned in Lemma 2, setting the degree of buckets from $d_{Curr.}$ to d_{Next} will increase the numbers of index and data items in each probe bucket of Type II data requests by $d_{Next} - d_{Curr.}$. Suppose that these extra index and data items are from the search buckets. Then, we have

Lemma 3 $Stat_{II}^{d_{Next}}.AvgATS$ and $Stat_{II}^{d_{Next}}.AvgTTS$ can be approximated as

$$Stat_{II}^{d_{Next}}.AvgATS = AvgSBNo_{Next} \times d_{Next} \times \frac{(S_I + S_D)}{B},$$

and

$$Stat_{II}^{d_{Next}}.AvgTTS = AvgSBNo_{Next} \times \left(d_{Next} \times \frac{S_I}{B} + T_{Off} + T_{On} \right),$$

where

$$AvgSBNo_{Next} \quad = \quad \frac{Stat_{II}.AvgATS \times B}{d_{Next} \times (S_I + S_D)} - \frac{d_{Next} - d_{Curr.}}{d_{Next}},$$

respectively.

Lemma 4 $Stat_{II}^{d_{Next}}.AvgATR$ and $Stat_{II}^{d_{Next}}.AvgTTR$ can be approximated as

$$Stat_{II}^{d_{Next}}.AvgATR = Stat_{II}.AvgATR + (d_{Next} - d_{Curr.}) \times \frac{S_I}{B},$$

and

$$Stat_{II}^{d_{Next}}.AvgTTR = Stat_{II}^{d_{Next}}.AvgTTR,$$

respectively.

The approximations of $T_{Access}(d_{Next})$ and $T_{Tuning}(d_{Next})$ can be calculated based on the above approximations. From the above lemmas, we have the following observations:

1. Increasing the value of degree will increase average tuning time in probe buckets since the number of data items in a bucket increases. In addition, increasing the value of degree will also reduce the aggregate tuning time of search buckets since the average number of search buckets decreases. The

increase of average tuning time in probe buckets and the decrease of aggregate tuning time of search buckets are, respectively, the benefit and the cost of increasing the value of degree.

2. To minimize average tuning time, decreasing the value of degree is encouraged when average access time it short. It is because that decreasing the value of degree will reduce the average tuning time in probe buckets by slightly increasing aggregate tuning time of search buckets. Such increase results from the increase in the average number of the search buckets.

We then devise procedure DegreeAdjustment to find the value of d_{Next} where $T_{OverAll}(d_{Next})$ is local minimum. In procedure DegreeAdjustment, the server first checks whether increasing or decreasing the value of degree will reduce the value of $T_{OverAll}(d_{Next})$. After that, the server repeatedly increases or decreases the value of degree by one until $T_{OverAll}(d_{Next})$ is local minimum. Finally, the system sets the value of degree (i.e., $d_{Curr.}$) to the return value of procedure DegreeAdjustment. The algorithmic form of procedure DegreeAdjustment is as follows.

Procedure DegreeAdjustment

Note: The new value of d (i.e, d_{Next})is returned

1: **if** $(T_{OverAll}(d_{Curr.}+1) < T_{OverAll}(d_{Curr.}))$ **then**
2: $\quad \delta \leftarrow 1$
3: **else if** $(T_{OverAll}(d_{Curr.}-1) > T_{OverAll}(d_{Curr.}))$ **then**
4: $\quad \delta \leftarrow -1$
5: **else**
6: $\quad$ **return** $d_{Curr.}$
7: $d_{Next} \leftarrow d_{Curr.}$
8: **while** $(T_{OverAll}(d_{Next}+\delta)) < T_{OverAll}(d_{Next}))$ **do**
9: $\quad d_{Next} \leftarrow d_{Next}+\delta$
10: **return** d_{Next}

4.3 Complexity Analysis

To derive the worst time complexity of algorithm AIDOA, we consider the case that no request merge occurs. Suppose that the number of received requests in one execution of statistics collection phase is n. Then, the time complexity of one execution of statistics collection phase is $O(n)$ since the time complexity of one execution of procedure RequestServed is $O(1)$. Suppose that the maximal value of degree is d_{Max}. The time complexity of procedure DegreeAdjustment is $O(d_{Max})$. Since algorithm AIDOA executes procedure DegreeAdjustment once in each execution of degree adjustment phase, the time complexity of one execution of degree adjustment phase is $O(d_{Max})$. To implement algorithm AIDOA, we have to spend

storage space to store structures $Stat_I$ and $Stat_{II}$. Since the sizes of structures $Stat_I$ and $Stat_{II}$ are fixed and are independent of n, the space complexity of algorithm AIDOA is $O(1)$.

5 Design of Program Generation Algorithm and Cache Replacement Policy

After determining the new value of degree, the program generator will generate the successive buckets accordingly. Since data items may be cached in the server cache, the adopted program generation algorithm should cooperate with the employed cache replacement policy. Each cached data item is initially marked as LOCKED and only the cached data items in UNLOCK state are candidates of replacement. To facilitate the design of cache replacement policy, the system maintains a min heap *Cand* which stores all data items in UNLOCKED state according to their priorities. The definition of the priority of a data item will be given later in this section. Note that in this and the following section, we relax the assumption that all data items are of the same size, and denote the size of D_i as $size(D_i)$ and the average data size as S_D.

The server maintains a list *bucket* which contains the index items and data items of the current bucket. Initially, *bucket* is empty. Then, the server retrieves $d_{Curr.}$ data items from the head of the ready queue, inserts them into *bucket* and marks them as LOCKED. In addition, the corresponding index items of the data items in *bucket* are also inserted into *bucket*. Then, the server broadcasts the index items and data items in *bucket* sequentially. Once an item has been broadcast, it will be removed from *bucket*. If the item is a data item, it will be marked as UNLOCKED. Once *bucket* becomes empty, the server retrieves $d_{Curr.}$ data items from the head of the ready queue and repeats the above procedure. The algorithmic form of the proposed program generation algorithm is as follows.

Algorithm ProgramGeneration

```
1: while (true) do
2:     bucket ←BucketGeneration()
3:     while (bucket is not empty) do
4:         item ←the head of bucket
5:         Remove the head of bucket
6:         Broadcast item
7:         if (item is a data item) then
8:             Mark item as UNLOCKED
9:             Calculate the priority of item and insert item into Cand
```

Procedure BucketGeneration

```
1: budket ← empty
```

2: **for** (i=1 to $d_{Curr.}$) **do**
3: **if** (ready queue is empty) **then**
4: **break**
5: Fetch a data item (denoted as *item*) from the head of ready queue
6: Mark *item* as LOCKED
7: Append *item* into *bucket*
8: Insert the corresponding index items of the data items in *bucket* into the head of *bucket*
9: **return** *bucket*

We now consider the design of server cache. Similar to other cache replacement policies, we define an evict function to determine the cache priorities of all data items. The profit of caching a data item is defined as the overall data fetch time saving when the data item is cached, The cost of caching a data item is defined as the size of the data item. The cache replacement policy is designed to maximize the aggregate profit of all cached data items under the limitation on the aggregate cost (i.e., size) of all cached data items. Hence, the cache priority of a data item D_i is defined as below.

$$priority(D_i) = \frac{fetch(D_i) \times rate(D_i)}{size(D_i)},$$

where $fetch(D_i)$ is the time for the server to fetch D_i from the data server of D_i and $rate(D_i)$ is the request rate of D_i. When retrieving D_i from the corresponding data server, the server calculates the value of $fetch(D_i)$ and stores it for further uses. The server also stores the time of the previous cache hit of D_i, denoted as $t_{PrevHit}(D_i)$. In addition, for each cache hit of D_i, $rate(D_i)$ is set to

$$\frac{1}{t_{CurHit} - t_{PrevHit}(D_i)},$$

where t_{CurHit} is the time of the current cache hit of D_i. After the calculation of request rate of D_i, $t_{PrevHit}(D_i)$ is set to t_{CurHit}.

The proposed cache replacement policy is as follows. When a data item, say D_i, is retrieved from the data server, it will be inserted into the cache. When inserting D_i into the cache, the server first checks whether the cache is of enough free space for D_i. If yes, the system stores D_i into the cache, calculates $priority(D_i)$ and marks D_i as LOCKED. Otherwise, the system repeatedly removes “the data item with the smallest priority among all data items in *Cand*” from *Cand* until the free space of the cache becomes enough. Then, the system stores D_i into the cache, calculates $priority(D_i)$ and marks D_i as LOCKED. The algorithmic form of the proposed cache replacement policy is as follows.

Parameter	Value
Data object number	4000
Data object sizes	Lognormal dist. (mean 7 KB)
Data access probabilities	Zipf dist. with parameter 0.75
Cache capacity	$0.01 \times \sum$ object size
Object fetch delay	Exponential dist. with $\mu = 2.3$
Client number	250
Service holding time	Exp. dist. with $\mu = 10$ minutes
Service establishing time	Exp. dist. with $\mu =$ one hour

Table 2: Default system parameters

Algorithm CacheReplacement(D_i)

1: **while** ($FreeSpace < size(D_i)$) **do**
2: Let D_j be the data item with the smallest priority among all other data items in *Cand*
3: Remove D_j from cache
4: $FreeSpace \leftarrow FreeSpace + size(D_j)$
5: Insert D_i into cache
6: Calculate the $priority(D_i)$
7: Mark D_i as LOCKED

Suppose that the data items in *Cand* are organized as a min heap. In addition, let $n_{Replace}$ be the number of data items to be replaced. Therefore, the time complexity of one execution of algorithm CacheReplacement is $O(n_{Replace} \times \log |Cand|)$.

6 Performance Evaluation

6.1 Simulation Model

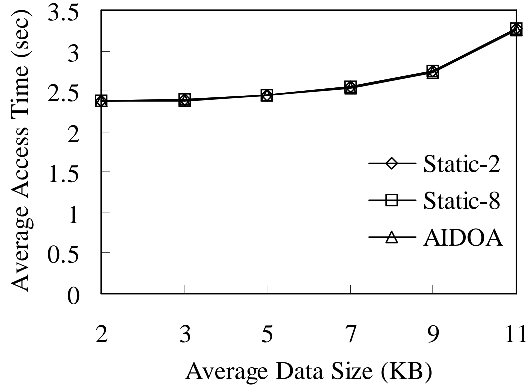
We take LWF (standing for Longest Wait First) as the underlying scheduling algorithm to prioritize the data requests in the request queue and the ready queue. The server provides one request channel and one broadcast channel with network bandwidth 38.4 Kbps and 384 Kbps, respectively. Analogously to [8], we assume that there are 4000 data objects and the sizes of data objects follow a lognormal distribution with a mean of 7 KBytes. The size of a data request message and an index item is set to 128 bytes. The times to turn on and turn off the WNIs are both set to 30ms. The access probability of data objects follows a Zipf distribution, which is widely adopted as a model for real Web traces [3][7]. The parameter of the Zipf distribution is set to 0.75 with a reference to the analyses of real Web traces [7][15]. Since small objects are

much more frequently accessed than large ones [9], we assume that there is a negative correlation between the object size and its access probability. The default capacity of the cache is set to $0.01 \times \sum$object size and the fetch delays of data objects follow an exponential distribution with mean 2.3 seconds [8]. Similar to [16], the number of users in the network is set to 250. Service holding time and service re-establishing time for each user are set to exponential distributions with means of 10 minutes and one hour, respectively. Service re-establishing time is defined as the time interval between the moment that a user terminates the service and the moment that the user establishes the service again. We also assume that the inter-arrival time of data requests of each user follow an exponential distribution with mean 10 seconds [14]. The value of β is set to 0.5 simulate the environment that average access time and average tuning time are of equal importance.

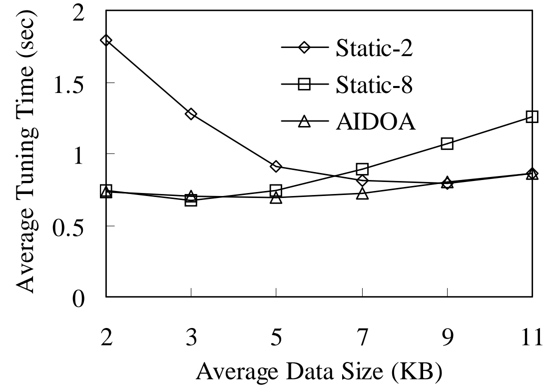
In order to evaluate the performance of the proposed degree adjustment method in algorithm AIDOA, the algorithm proposed in [13] (referred to as algorithm Static) is modified to cooperate with the cache replacement policy and the program generation algorithm proposed in Section 5. Hence, the difference between algorithm AIDOA and algorithm Static is only on the ability of adjusting the degree of buckets. Based on algorithm Static, we devise two schemes, Static-2 and Static-8 which set the degree of buckets to two and 8, respectively, and the values of degree of buckets are fixed throughout the simulation. In addition, scheme AIDOA employs algorithm AIDOA and initializes the degree of buckets to two. Hence, scheme AIDOA will dynamically adjust the degree of buckets according to system workload. Note that all these three schemes employ server cache to eliminate performance degradation caused by the data fetch time.

6.2 Effect of Average Data Size

In this experiment, we investigate the effect of average data size on average access time and average tuning time. Average data size is set from two KBytes to 11 KBytes and the experimental results are shown in Figure 9a and Figure 9b, respectively. Due to increasing the load of the broadcast channel, it is intuitive that increasing average data size results in the increase in average access time. In addition, when average data size is large enough, the load of the broadcast channel is high, and hence, a slight increase in average data size will cause significant increases in average access time. Since the sizes of index items are much smaller than those of data items, the effect of the degrees of broadcast programs in average access time is



(a) Average Access Time

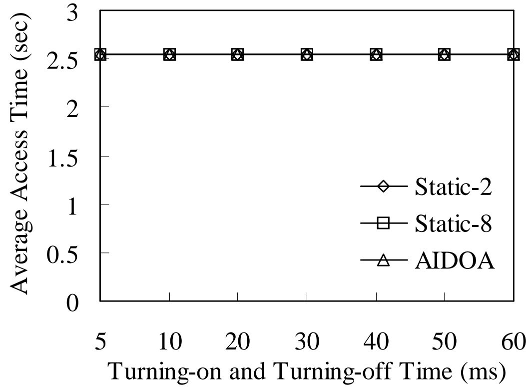


(b) Average Tuning Time

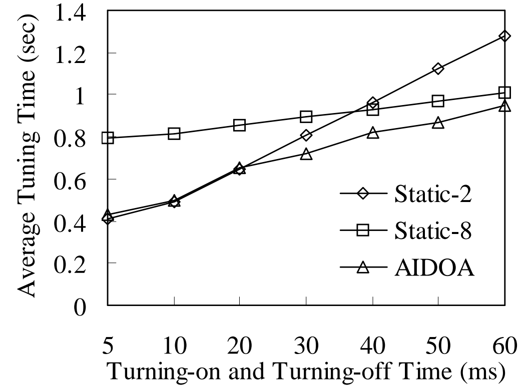
Figure 9: The effect of average data size

quite small.

Although the values of degrees of broadcast programs only slightly affect average access time of all schemes, they result in significant effects in average tuning time. As shown in Figure 9b, scheme Static-8 performs well only when average data size is small, and scheme Static-2 performs well only when average data size is large. As observed in Section 4.2, increasing the value of degree will increase average tuning time in the probe bucket. In addition, increasing the value of degree also decreases the number of search buckets and hence reduces average tuning time in the search buckets. When average data size is large, average access time is also long. Employing a large value of degree will reduce average tuning time in the search buckets by reducing the number of search buckets (i.e., reducing the number of times of turning-on and turning-off the WNIs), and increase the average tuning time in the probe bucket. However, due to the trade-off between average tuning time in the probe bucket and the search buckets, the value of degree cannot be set to be too large. In addition, we can also observe from Section 4.2 that decreasing the value of degree will decrease average tuning time in the probe bucket and increase the number of search buckets. Therefore, scheme with small values of degree outperforms schemes with large values of degree in the case with small average data size. Hence, the value of degree cannot be set to be too small, either. Different from scheme Static-2 and scheme Static-8, since scheme AIDOA is able to dynamically adjust the value of degree to a proper value according to system workload, scheme AIDOA outperforms scheme Static-2 and scheme Static-8 in most cases.



(a) Average Access Time



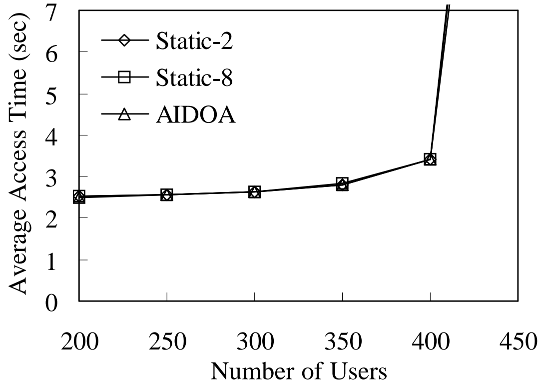
(b) Average Tuning Time

Figure 10: The effect of turning-on and turning-off time

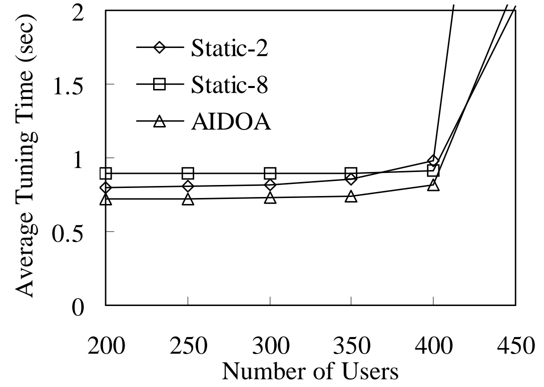
6.3 Effect of Turning-on and Turning-off Time of WNIs

The effect of turning-on and turning-off time of WNIs is measured in this subsection and the experimental results are given in Figure 10. In this experiment, we assume that $T_{On} = T_{Off}$ and set the value of T_{On} and T_{Off} from 5ms to 60ms. As shown in Figure 10a, the values of T_{On} and T_{Off} do not affect average access time of scheme Static-2 and scheme Static-8. It is because that in these two schemes, degrees of broadcast buckets are fixed, and the values of T_{On} and T_{Off} do not affect the organizations of broadcast programs. On the other hand, although scheme AIDOA is able to dynamically adjust the degree of buckets, the influence of T_{On} and T_{Off} on average access time of scheme AIODA is small since the size of index items is much smaller than that of data items.

Consider average tuning time of these schemes shown in Figure 10b. According to the observations in Section 4.2, increasing the value of degree will increase average tuning time in the probe bucket and reduce the aggregate tuning time in the search buckets. Since the benefit of increasing the value of degree is in proportion to the values of T_{On} and T_{Off}, scheme Static-2 performs well when T_{On} and T_{Off} are small. On the contrary, scheme Static-8 outperforms scheme Static-2 in the case with large T_{On} and T_{Off}. Although producing more power consumption in the probe bucket than scheme Static-2 does, scheme Static-8 is still able to reduce overall power consumption since being able to greatly reduce the power consumption on turning-on and turning-off the WNIs by reducing the average number of search buckets. On the other hand, with dynamic adjustment in the degree, scheme AIDOA is able to determine a suitable value of degree for current system workload, and hence outperforms scheme Static-2 and scheme Static-8 in most



(a) Average Access Time



(b) Average Tuning Time

Figure 11: The effect of the number of users

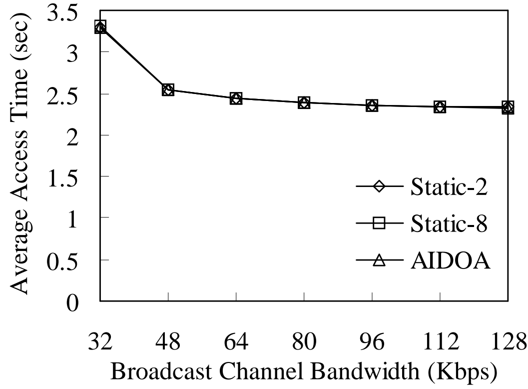
cases.

6.4 Effect of Number of Users

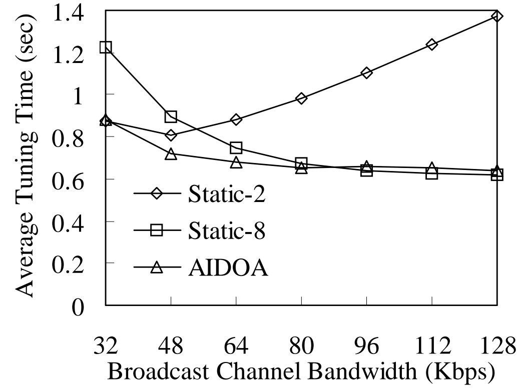
We evaluate in this subsection the scalability of these schemes in average access time and average tuning time by increasing the number of users from 200 to 450, and the experimental results are shown in Figure 11.

Due to the characteristics of data broadcasting, it is intuitive that increasing the number of users results in smoothly increasing in average access time. As shown in Figure 11a, when the number of users is small, increasing the number of users produces a slight increase in average access time since the system load is still light. However, when the number of users is large enough, the system load becomes high and increasing the number of users results in drastic increases in average access time. In addition, since the size of index items is much smaller than that of data items, average access time of these schemes is close.

Figure 11b shows average tuning time of these schemes with the number of users varied. We observe that when the number of users is not large, scheme Static-2 outperforms scheme Static-8. In addition, when the number of users is large, average tuning time of scheme Static-2 becomes longer than that of scheme Static-8. This phenomenon agrees to the observations in Section 4.2 that in average tuning time, the cases with short average access time favor schemes with small values of degree and the cases with long average access time favor schemes with large values of degree. Therefore, when the number of users is large enough, average tuning time of scheme Static-2 becomes much longer than that of scheme Static-



(a) Average Access Time



(b) Average Tuning Time

Figure 12: The effect of bandwidth of broadcast channel

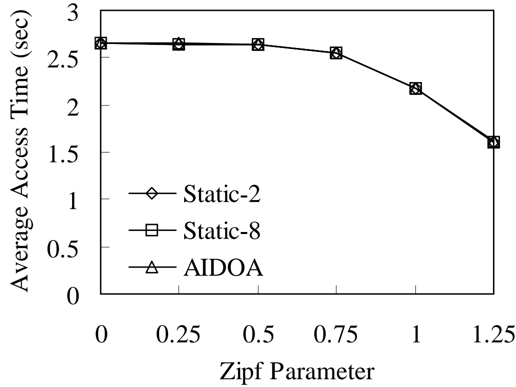
8 since average access time of these schemes both becomes drastically increasing. Since being able to adjust the values of degree according to system workload, scheme AIDOA outperforms scheme Static-2 and Static-8 when the number of users is not large. In addition, average tuning time of scheme AIDOA is close to that of scheme Static-8 when the number of users is large.

6.5 Effect of Bandwidth of the Broadcast Channel

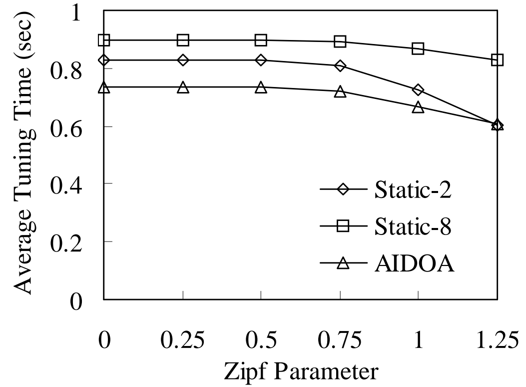
In this experiment, we investigate the effect of bandwidth of the broadcast channel on average access time and average tuning by setting the bandwidth from 32 Kbps to 128 Kbps. The experimental results are shown in Figure 12.

It is intuitive that increasing bandwidth of the broadcast channel decreases average access time. However, as shown in Figure 12a, the effect of increasing the bandwidth of the broadcast channel on average access time diminishes when the bandwidth is large enough. It is because that average access time comprises several components such as the fetch time and the broadcast time of data items, the waiting time of data requests spending in queues and the transmission time of data requests on the request channel. Since increasing bandwidth of the broadcast channel only reduces broadcast time of data items, the effect of increasing bandwidth of the broadcast channel on average access time is limited. Similar to the precedent experiments, average access time of these schemes is close.

Now, consider the experimental results on average tuning time shown in Figure 12b. As observed in Figure 12b, scheme Static-8 performs well only when the bandwidth of the broadcast channel is high.



(a) Average Access Time



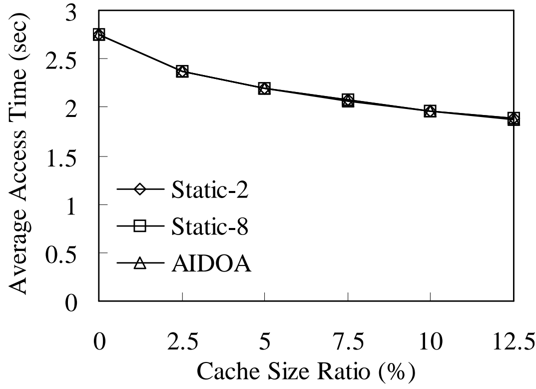
(b) Average Tuning Time

Figure 13: The effect of skewness of data requests

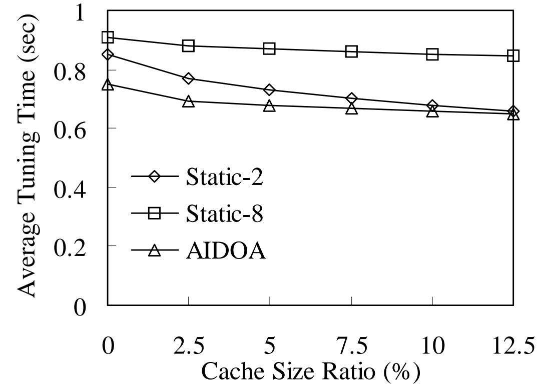
Similarly, scheme Static-2 performs well only when the bandwidth of the broadcast channel is low. Consider the scenario of increasing the value of degree. As observed in Section 4.2, with the same increment in the value of degree, increasing the bandwidth of the broadcast channel will reduce the cost of increasing the value of degree by reducing average tuning time in the probe bucket. Since most average tuning time in the search buckets comes from turning-on and turning-off the WNIs, average tuning time in the search buckets does not affect by the increase of the bandwidth of the broadcast channel. Therefore, increasing bandwidth of the broadcast channel favors increasing the value of degree. Similarly, decreasing bandwidth of the broadcast channel makes increasing the value of degree more costly. On the other hand, scheme AIDOA is able to adjust the value of degree according to system workload, and hence outperforms scheme Static-2 and scheme Static-8 in most cases. This phenomenon shows the advantage of scheme AIDOA due to its adaptability to system workload.

6.6 Effect of Skewness of Access Probabilities of Data Requests

We evaluate in this experiment the effect of skewness of access probabilities of data requests by setting the value of Zipf parameter from zero to 1.25. The larger the value of Zipf Parameter is, the more skewed access probabilities of data requests are. In addition, the value of Zipf parameter is set to zero to indicate the case that access probabilities of data requests are equal. It is intuitive that when the access probabilities become more skewed, more data requests are merged together, and average access time becomes shorter. The result shown in Figure 13a agrees to this intuition.



(a) Average Access Time



(b) Average Tuning Time

Figure 14: The effect of cache size ratio

Figure 13b shows average access time of all schemes with the value of Zipf parameter varied. As observed in Figure 13b, average tuning time decreases as the value of Zipf parameter increases. It can be explained by the observations in Section 4.2 that in average tuning time, cases with short average access time favors schemes with small values of degree. Therefore, scheme Static-2 outperforms scheme Static-8 especially in the cases with high value of Zipf parameter. With the change of skewness of access probabilities, scheme AIDOA is able to adjust the value of degree to adapt to such change, and hence, attains better performance.

6.7 Effect of Cache Size

This experiment evaluates the effect of cache size on average access time and average tuning time, and the experimental results are shown in Figure 14. Similar to [8], cache size is determined as "cache size ratio$\times$the summation of the sizes of all data items." The case that the value of cache size ratio is set to zero indicates the case that the server does not employ cache.

As shown in Figure 14a, average access time decreases as the value of cache size ratio increases. When the cache size is large, many data items are cached and can be obtained by the server without being fetched from the data servers. In addition, when the cache size is large enough, the benefit of increasing cache size diminishes since data items with high access rates are cached in the server cache. We also observe from Figure 14a that employing server cache, which is neglected in the prior studies on data indexing for on-demand data broadcasting [13], is able to effectively reduce average access time.

As the observations in Section 4.2, when it comes to average tuning time, cases with short average access time favor schemes with small values of degree. Hence, scheme Static-2 outperforms scheme Static-8 especially when the value of cache size ratio is large. This observation agrees with the results shown in Figure 14b. When the value of cache size ratio is small, both schemes do not perform well since the value of degree in scheme Static-2 is too small and the value of degree in scheme Static-8 is too large. On the other hand, scheme AIDOA is able to dynamically adjust the value of degree to attain better performance, showing the advantage of scheme AIDOA.

7 Conclusion

We proposed in this paper an energy-conserving on-demand data broadcasting system employing the data indexing technique. Different from the prior work, power consumption of turning on and turning off the wireless network interfaces was considered. In addition, we also employed server cache to reduce the effect of data fetch time. Specifically, we first analyzed the access time and tuning time of data requests and proposed algorithm AIDOA to adjust the degree of buckets according to system workload. We also devised an approximation method to estimate the effect of increasing and decreasing the values of degree, and employed the approximation method to guide the adjustment of algorithm AIDOA. In addition, the companion program generation algorithm and cache replacement policy were proposed to cooperate with algorithm AIDOA. Several experiments were then conducted to evaluate the performance of algorithm AIDOA. Experimental results showed that algorithm AIDOA is able to greatly reduce power consumption at the cost of a slight increase in average access time and dynamically adjust the index and data organization to adapt to change of system workload.

References

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast Disks: Data Management for Asymmetric Communication Environments. In *Proceedings of the ACM SIGMOD Conference*, pages 198–210, March 1995.

[2] S. Acharya and S. Muthukrishnan. Scheduling On-demand Broadcasts: New Metrics and Algorithms. In *Proceedings of the 4th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 43–94, October 1998.

[3] C. Aggarwal, J. L. Wolf, and P. S. Yu. Caching on the World Wide Web. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):94–107, 1999.

[4] M. Agrawal, A. Manjhi, N. Bansal, and S. Seshan. Improving Web Performance in Broadcast-Unicast Networks. In *Proceedings of the IEEE INFOCOM Conference*, March-April 2003.

[5] D. Aksoy and M. J. Franklin. Scheduling for Large-Scale On-Demand Data Broadcasting. In *Proceedings of IEEE INFOCOM Conference*, pages 651–659, March 1998.

[6] D. Aksoy, M. J. Franklin, and S. Zdonik. Data Staging for On-Demand Broadcst. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 571–580, September 2001.

[7] L. Breslau, P. Cao, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of the IEEE INFOCOM Conference*, March 1999.

[8] C.-Y. Chang and M.-S. Chen. Exploring Aggregate Effect with Weighted Transcoding Graphs for Efficient Cache Replacement in Transcoding Proxies. In *Proceedings of the 18th IEEE International Conference on Data Engineering*, Feburary 2002.

[9] S. Glassman. A Caching Relay for the World Wide Web. *Computer Networks and ISDN Systems*, 27, 1994.

[10] J.-L. Huang and M.-S. Chen. Dependent Data Broadcasting for Unordered Queries in a Multiple Channel Mobile Environment . *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1143–1156, September 2004.

[11] J.-L. Huang and W.-C. Peng. An Energy-Conserved On-Demand Data Broadcasting System. In *Proceedings of the 6th International Conference on Mobile Data Management*, May 2005.

[12] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Data on Air: Organization and Access. *IEEE Transactions on Knowledge and Data Engineering*, 9(9):353–372, June 1997.

[13] S. Lee, D. P. Carney, and S. Zdonik. Index Hint for On-demand Broadcasting. In *Proceedings of the 19th IEEE International Conference on Data Engineering*, March 2003.

[14] C.-W. Lin, H. Hu, and D. L. Lee. Adaptive Realtime Bandwidth Allocation for Wireless Data Delivery. *ACM/Kluwer Wireless Networks*, 10:103–120, 2004.

[15] V. Padmanabhan and L. Qiu. The Content and Access Dynamics of a Busy Web Site: Findings and Implications. In *Proceedings of the IEEE SIGCOMM Conference*, pages 293–304, August-September 2000.

[16] M. A. Sharaf and P. K. Chrysanthis. On-Demand Data Broadcasting for Mobile Decision Making. *ACM/Kluwer Mobile Networks and Applications*, 9(4):703–714, December 2004.

[17] E. Shih, P. Bahl, and M. J. Sinclair. Wake on Wireless: An Event Driven Energy Saving Strategy for Bettery Operated Devices. In *Proceedings of the 8th ACM/IEEE International Conference on Mobile Computing and Networking*, September 2002.

[18] T. Simunic, S. Boyd, and P. Glynn. Managing Power Consumption in Networks on Chips. *IEEE Transactions on Very Large Scale Integration Systems*, 12(1):96–107, January 2004.

[19] M. A. Viredaz, L. S. Brakmo, and W. R. Hamburgen. Energy Management on Handheld Devices. *ACM Queue*, 1(7):44–52, October 2003.

[20] J. Xu, W.-C. Lee, and X. Tang. Exponential Index: A Parameterized Distributed Indexing Scheme for Data on Air. In *Proceedings of the 2nd ACM/USENIX International Conference on Mobile Systems*, June 2004.

[21] J. L. Xu, B. Zheng, W.-C. Lee, and D. K. Lee. Energy Efficient Index for Querying Location-Dependent Data in Mobile Broadcast Environments. In *Proceedings of the 19th International Conference on Data Engineering*, March 2003.

[22] H. Zhu and G. Cao. A Power-Aware and QoS-Aware Service Model on Wireless Networks. In *Proceedings of IEEE INFOCOM Conference*, March 2004.

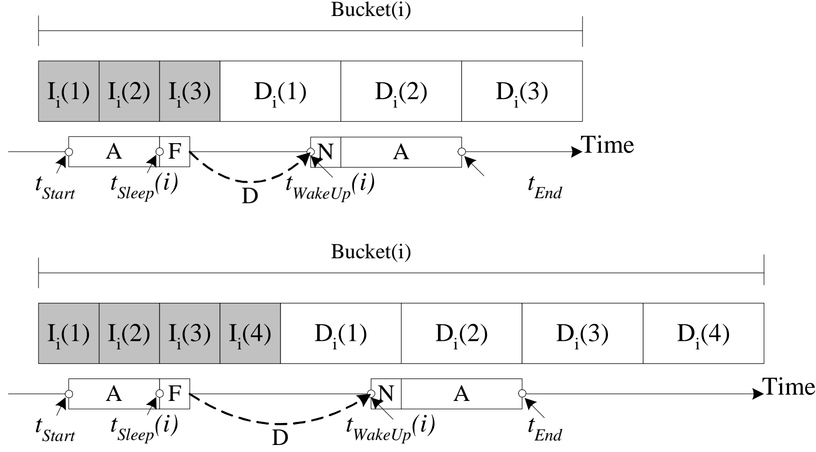


Figure 15: An example scenario of Type I data requests that $d_{Curr.} = 3$ and $d_{Next} = 4$

Appendix

Proof of Lemma 1:

Consider the cases that $d_{Next} > d_{Curr.}$. Figure 15 shows an example of Type I data requests. When the degree of buckets is set from $d_{Curr.}$ to d_{Next}, $d_{Next} - d_{Curr.}$ index item(s) and $d_{Next} - d_{Curr.}$ data item(s) will be appended to each index segment and data segment, respectively. As observed from Figure 15, setting the degree of buckets from $d_{Curr.}$ to d_{Next} increases the average access time of Type I data requests by $\frac{(d_{Next}-d_{Curr.})\times S_I}{B}$. In addition, we also observe that appending $d_{Next} - d_{Curr.}$ index items into each index segment does not affect the average tuning time of Type I requests. Hence, from the above observations, we have

$$Stat_I^{d_{Next}}.AvgAT = Stat_I.AvgAT + (d_{Next} - d_{Curr.}) \times \frac{S_I}{B}, \text{ and}$$

$$Stat_I^{d_{Next}}.AvgTT = Stat_I.AvgTT.$$

We then apply the above equations as the approximations of $Stat_I^{d_{Next}}.AvgAT$ and $Stat_I^{d_{Next}}.AvgTT$ in the cases that $d_{Next} < d_{Curr.}$, and hence, prove Lemma 1. **Q.E.D.**

Proof of Lemma 2:

Consider the example probe bucket of Type II data requests shown in Figure 16. Suppose that t_{Start} follows a uniform distribution between $Bucket_i.Start$ and $Bucket_i.End$. Therefore, as observed from Figure 16, the probabilities of a Type II data request to be Type II.I and Type II.II are $\frac{S_I}{S_I+S_D}$ and $\frac{S_D}{S_I+S_D}$,

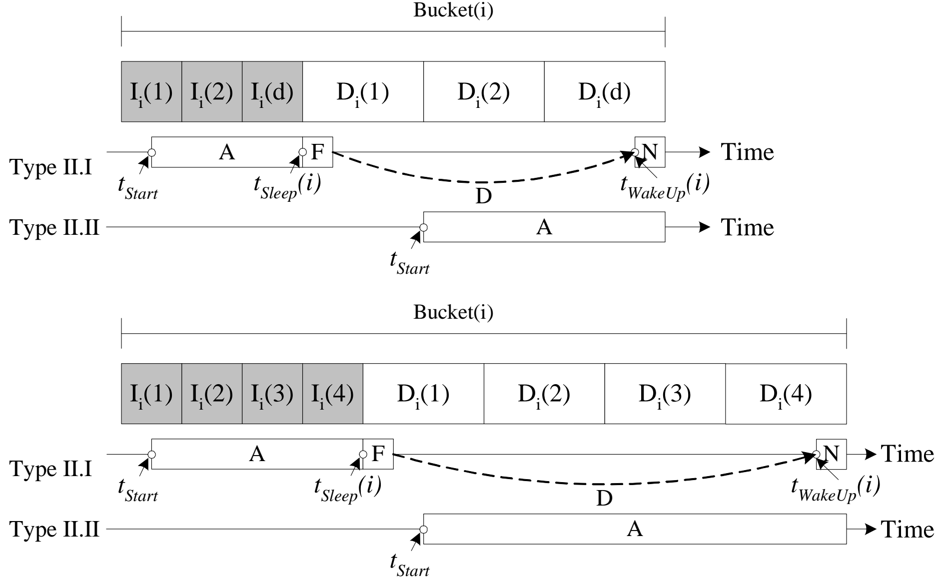


Figure 16: An example scenario of Type II data requests that $d_{Curr.} = 3$ and $d_{Next} = 4$ on the probe bucket

respectively. Therefore, by the definition of $Stat_{II}^{d_{Next}}.AvgATP$ and $Stat_{II}^{d_{Next}}.AvgTTP$, we have

$$Stat_{II}^{d_{Next}}.AvgATP = \frac{S_I}{S_I+S_D} \times Stat_{II.I}^{d_{Next}}.AvgATP + \frac{S_D}{S_I+S_D} \times Stat_{II.II}^{d_{Next}}.AvgATP, \text{ and}$$

$$Stat_{II}^{d_{Next}}.AvgTTP = \frac{S_I}{S_I+S_D} \times Stat_{II.I}^{d_{Next}}.AvgTTP + \frac{S_D}{S_I+S_D} \times Stat_{II.II}^{d_{Next}}.AvgTTP.$$

We now consider the cases that $d_{Next} > d_{Curr.}$ and derive the approximations of $Stat_{II.I}^{d_{Next}}.AvgATP$, $Stat_{II.I}^{d_{Next}}.AvgTTP$, $Stat_{II.II}^{d_{Next}}.AvgATP$ and $Stat_{II.II}^{d_{Next}}.AvgTTP$. When the degree of buckets is set to from $d_{Curr.}$ to d_{Next}, $d_{Next} - d_{Curr.}$ index item(s) and $d_{Next} - d_{Curr.}$ data item(s) will be appended to each index segment and data segment. As observed from Figure 16, setting the degree of buckets from $d_{Curr.}$ to d_{Next} increases the average access time and average tuning time of the probe buckets of Type II.I data requests (i.e., $Stat_{II.I}^{d_{Next}}.AvgATP$ and $Stat_{II.I}^{d_{Next}}.AvgTTP$) by $\frac{(d_{Next}-d_{Curr.})\times(S_I+S_D)}{B}$ and $\frac{(d_{Next}-d_{Curr.})\times S_I}{B}$, respectively. Hence, we have

$$Stat_{II.I}^{d_{Next}}.AvgATP = Stat_{II}.AvgATP + (d_{Next} - d_{Curr.}) \times \left(\frac{S_I}{B} + \frac{S_D}{B}\right), \text{ and}$$

$$Stat_{II.I}^{d_{Next}}.AvgTTP = Stat_{II}.AvgTTP + (d_{Next} - d_{Curr.}) \times \frac{S_I}{B}.$$

In addition, we also observe that increasing the degree of buckets from $d_{Curr.}$ to d_{Next} increases both

the average access time and average tuning time of the probe buckets of Type II.II data requests (i.e., $Stat_{II.II}^{d_{Next}}.AvgATP$ and $Stat_{II.II}^{d_{Next}}.AvgTTP$) by $\frac{(d_{Next}-d_{Curr.})\times(S_I+S_D)}{B}$. Therefore, we have

$$Stat_{II.II}^{d_{Next}}.AvgATP = Stat_{II}.AvgATP + (d_{Next} - d_{Curr.}) \times \frac{S_D}{B}, \text{ and}$$

$$Stat_{II.II}^{d_{Next}}.AvgTTP = Stat_{II}.AvgTTP + (d_{Next} - d_{Curr.}) \times \frac{S_D}{B}.$$

We then apply the above equations to the approximations of $Stat_{II.I}^{d_{Next}}.AvgATP$, $Stat_{II.I}^{d_{Next}}.AvgTTP$, $Stat_{II.II}^{d_{Next}}.AvgATP$ and $Stat_{II.II}^{d_{Next}}.AvgTTP$ in the cases that $d_{Next} < d_{Curr.}$, and hence, prove Lemma 2.

Q.E.D.

Proof of Lemma 3:

In the cases that the degree of buckets is $d_{Curr.}$, since one data item and the corresponding index item contribute $Stat_{II}.AvgATS$ by $\frac{S_I+S_D}{B}$, the average number of data items in Type II data requests is $Stat_{II}.AvgATS \times \frac{B}{S_I+S_D}$.

Consider the cases that set the degree of buckets to d_{Next}. For each Type II data request, on average, $d_{Next} - d_{Curr.}$ index items and $d_{Next} - d_{Curr.}$ data items move from the search buckets to the probe bucket. Therefore, the average number of index and data items in Type II data requests both become $Stat_{II}.AvgATS \times \frac{B}{S_I+S_D} - (d_{Next} - d_{Curr.})$. Since each search bucket contains d_{Next} index items and d_{Next} data items, the average number of search buckets of Type II data requests is

$$AvgSBNo_{Next} = Stat_{II}.AvgATS \times \frac{B}{S_I + S_D} - (d_{Next} - d_{Curr.}).$$

Finally, according to the derivations in Section 3.2.2, since each Type II data request contains $AvgSBNo_{Next}$ search buckets on average, we have

$$Stat_{II}^{d_{Next}}.AvgATS = AvgSBNo_{Next} \times d_{Next} \times \frac{(S_I + S_D)}{B}, \text{ and}$$

$$Stat_{II}^{d_{Next}}.AvgTTS = AvgSBNo_{Next} \times \left(d_{Next} \times \frac{S_I}{B} + T_{Off} + T_{On}\right).$$

Q.E.D.

Proof of Lemma 4:

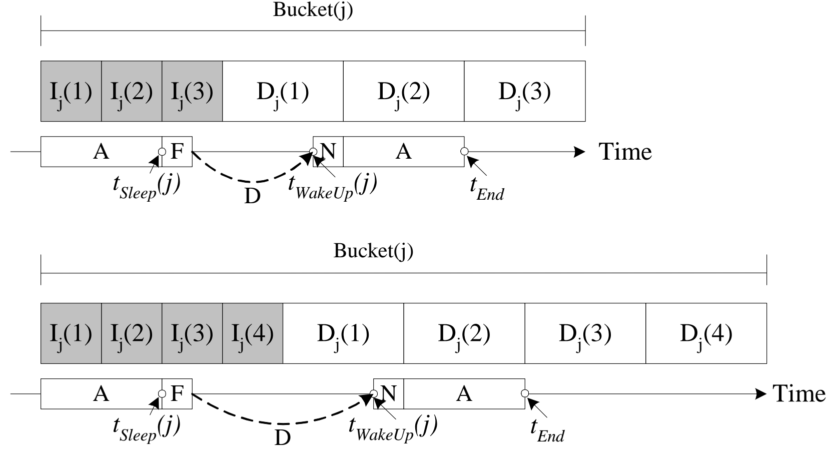


Figure 17: An example scenario of Type II data requests that $d_{Curr.} = 3$ and $d_{Next} = 4$ on the retrieval bucket

Consider the cases that $d_{Next} > d_{Curr.}$ and the example Type II data request shown in Figure 17. When the degree of buckets is set to from $d_{Curr.}$ to d_{Next}, $d_{Next} - d_{Curr.}$ index item(s) and $d_{Next} - d_{Curr.}$ data item(s) will be appended to each index segment and data segment, respectively. As observed from Figure 17, setting the degree of buckets from $d_{Curr.}$ to d_{Next} increases the average access time of each Type II data request on the retrieval bucket by $\frac{(d_{Next} - d_{Curr.}) \times S_I}{B}$. In addition, we also observe that appending $d_{Next} - d_{Curr.}$ index items into each index segment does not affect the average tuning time of each Type II request on the retrieval bucket. Hence, from the above observations, we have

$$Stat_{II}^{d_{Next}}.AvgATR = Stat_{II}.AvgATR + (d_{Next} - d_{Curr.}) \times \frac{S_I}{B}, \text{ and}$$

$$Stat_{II}^{d_{Next}}.AvgTTR = Stat_{II}.AvgTTR.$$

We then apply the above equations as the approximations of $Stat_{II}^{d_{Next}}.AvgATR$ and $Stat_{II}^{d_{Next}}.AvgTTR$ in the cases that $d_{Next} < d_{Curr.}$, and hence, prove Lemma 4. **Q.E.D.**