Performance and Stability Analysis of Multilevel Data Structures with Deferred Reorganization

Ing-Ray Chen, *Member*, *IEEE*, and Sayed A. Banawan, *Member*, *IEEE*

Abstract—We develop a methodology for analyzing the performance and stability of a server that maintains a multilevel data structure to service a set of access operations for (key, value) records. A subset of the operations executed by the server (e.g., insert and delete) require the multilevel data structure be *reorganized* so that the server can execute all subsequent requests efficiently. We study how often the server should carry out data reorganization (i.e., maintenance) to maximize its performance. If the server is frequently idle then there is no need to impose the reorganization overhead on the operation requests. The reorganization overhead may be completely *eliminated* by utilizing server-idling periods. If the server is frequently busy, then the reorganization overhead can be *minimized* by performing a complete reorganization only after the server has served a sufficient number of insert/delete operations so that the amortized cost per operation is small. Therefore, the issue of how often one should perform data reorganization to minimize the average service time depends not only on the multilevel data structure maintained by the server but also on the type and intensity of the system workload.

The proposed methodology is exemplified with a two-level sorted file with deferred maintenance. The performance and stability results are compared with those of a single-level binary tree data structure with on-the-fly maintenance. It is shown that deferred maintenance of the two-level sorted file outperforms on-the-fly maintenance of the single-level binary tree in both open and closed systems. Furthermore, deferred maintenance can sustain higher workload intensities without risking system stability.

Index Terms—Queueing servers, multilevel data structures, atomic operations, open and closed systems, modeling, deferred maintenance, periodic maintenance, performance analysis, stability analysis.

1 Introduction

The client-server paradigm has been widely accepted as an effective approach to structuring a computing system, especially in network-based environments. Many servers use a search table data structure to support lookup, update, deletion, and insertion of records. The records are accessed using a key field and updated atomically. Example applications include an authentication server that maintains a password file [5], [15], a database server that maintains bank account or employee information with the bank account or employee name being the search key [7], a name server that maintains object location information [18], a switch in a high speed computer network that maintains routing information for active virtual circuits [16], etc.

An issue that needs to be considered in the design of such servers is that after some insert/delete operations are executed, a data reorganization may be required. For example, performing an insert/delete operation on a sorted array data structure (a sorted file) requires the array be reorganized so that it remains sorted and compact. A balanced binary tree has a similar requirement. A B-tree must also reorganize its nodes in a bottom-up fashion when leaf nodes are split or merged.

- I.-R. Chen is with the Department of Computer Science, Virginia Polytechnic Institute and State University, Northern Virginia Center, Falls Church, VA 22043. E-mail: irchen@cs.vt.edu.
- S.A. Banawan is with the Department of Computer Science, Faculty of Science, University of Qatar, PO Box 2713, Doha, State of Qatar. E-mail: banawan@qu.edu.qa.

Manuscript received 26 Aug. 1993; revised 26 Aug. 1996.
Recommended for acceptance by R.A. Kemmerer.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 105891.

Because reorganization is usually a time-consuming operation, reorganizing the data structure on-the-fly, i.e., on a per insert/delete operation basis, needlessly increases the average service time of all operations because: 1) if the server is frequently idle then there is no need to impose the reorganization overhead on the operation requests; the reorganization overhead may be completely *eliminated* from the clients' point of view by utilizing idle periods for reorganization and 2) if the server is busy most of the time then the reorganization overhead can be *minimized* by performing a complete reorganization only after serving a sufficient number of insert/delete operations so that the amortized cost per operation is small. Therefore, the issue of when to perform data reorganization in order to minimize the average service time depends on not only the data structure maintained by the server but also the type and intensity of the system workload.

We have previously modeled and studied the stability and optimality conditions of a queueing server that maintains a single-level data structure (e.g., a linear list) with deferred maintenance [2], [4]. In this paper, we extend our previous work to model and analyze servers that use *multilevel* data structures, where the data is organized into a hierarchy of structures in order to provide efficient implementations for all access operations [14]. This approach requires the use of some maintenance methods to reorganize the multilevel data structure to achieve its goal. Background maintenance processes [6] were proposed to reorganize multilevel data structures concurrently with the server's foreground process [14]. Here, we study deferred maintenance [13] as a design alternative. The reason is that some experimental results [1] indicated deferred maintenance in the context of garbage collection can perform better than concurrent maintenance at high traffic situations because of the large context-switch overhead associated with concurrent maintenance, especially when there is no deterioration bound on concurrent maintenance (which makes it just a variant of deferred maintenance).

The paper develops a modeling methodology that can be applied to both open (e.g., transaction) and closed (e.g., batch and terminal) systems [12] for a queueing server maintaining a multilevel data structure with deferred maintenance. It does not make any assumptions regarding the workload intensity. Earlier work [14] dealt with concurrent maintenance only and made the stringent assumption that there is only a single terminal user in the whole system. Therefore, there was no need to consider the stability issue under high traffic situations. This paper proposes a deferred maintenance technique that can utilize server-idling periods to perform data reorganization. Using this technique, the server will: 1) perform a complete data reorganization *periodically* so as to minimize the average service time per operation; if a customer arrives before the maintenance is completed, the customer must wait in the queue; 2) attempt to reorganize the data structure whenever it becomes idle, i.e., when the server's queue is empty; however, if a customer arrives before the maintenance is completed, the maintenance is aborted (i.e., by discarding the scratch copy that would otherwise replace the data structure [3]) and the server resumes service immediately. Our goal is to first model this queueing system and then apply the model to uncover the stability and optimality conditions as a function of the system workload.

The paper is organized as follows. Section 2 provides some background information on queueing servers that maintain multilevel data structures with deferred maintenance, and introduces the notation used in the paper. Section 3 develops performance models to analyze the behavior of these servers in both open and closed systems. In doing so, a state reduction technique is applied to reduce the model complexity and to allow analytical solutions for the average service time to be derived as a function of the workload intensity. Section 4 explains how the modeling technique can be applied to study a server that maintains a two-level sorted array data structure in both open and closed systems. The results show that deferred maintenance can greatly improve system response time in comparison with on-the-fly maintenance of two other implementations, namely, a single-level sorted array and a single-level binary tree. Furthermore, deferred maintenance is able to sustain higher workload intensities in open systems without risking stability. An interpretation of the dependency of the optimal deferred maintenance period on the type and intensity of the system workload is given. Finally, Section 5 summarizes the paper and outlines some future research areas.

2 Background and Notation

Following the original definition in [14], a multilevel data structure is a hierarchy of data structures. The hierarchy is ordered by decreasing search efficiency so that if a requested key exists at the top level the request can be serviced efficiently (this by and large applies to insert operations that may require checking for duplicate keys); otherwise, the server must recursively search the lower levels, thus degrading its performance. A queueing server maintaining a multilevel data structure is characterized by a service rate that varies from one operation to another depending on the *degradation level* of the data structure. The degradation level relates to the amount of reorganization necessary to achieve minimum service time. To illustrate this concept, consider a two-level sorted array (e.g., a search table) defined as follows:

```
const n = ... {size of the table}
type entry = record
                 status:(empty, occupied);
                 key: KeyType;
                 info: InfoType
             end;
var A: array [1 .. n] of entry;
    B: array [1 .. n] of entry;
    SizeA: 0 .. n;
    SizeB: 0 .. n;
```

Array `A` is always sorted while array `B` is not. Therefore, searching for records in `A` is fast, although inserting new records into it is relatively slow. The opposite is true for `B`. The variables `SizeA` and `SizeB` store the largest indices after which entries are `empty` in arrays `A` and `B`, respectively. Initially, `SizeB` is zero.

The server supports four standard operations that may be invoked by clients: `lookup` or `update(key: KeyType, var info: InfoType)`, `insert(key: KeyType, info: InfoType)`, and `delete(key: KeyType)`. To execute a lookup or an update operation, the server performs a binary search on the sorted array from `A[1]` to `A[SizeA]`. If the record is not found and `SizeB` is greater than zero, then it performs a linear search from `B[1]` to `B[SizeB]`. If the record is found, the server reads or updates the array element accordingly. To execute a delete operation, the server performs a similar search to locate the record, then it marks the record "deleted" by changing the status field from `occupied` to `empty` without actually removing the record and reorganizing the data structure. To process an insert operation, the server simply stores the new record at location `B[SizeB + 1]` and increments `SizeB` by 1, again without performing a data reorganization. (If duplicate keys are allowed, a search may also be needed before inserting a new record.) As more and more operations are serviced, the service time per operation increases because more array elements, including those marked deleted, will have to be inspected by the server to execute a lookup/update/delete operation. Moreover, an overflow will eventually occur in array `B` unless a data reorganization is performed to stabilize the system.

When a data reorganization is performed, all `occupied` entries are sorted and copied into array `A` with `SizeA` marking the last one, and `SizeB` is reset back to 0. At this point, the server can again process a request efficiently because only a binary search on array `A` is required to service a lookup, update or delete operation. Note that for the two-level sorted array, the variable `SizeA` holds the size of the data structure just after a data reorganization while the variable `SizeB` holds the number of entries in excess of `SizeA` at any time before the next data reorganization starts. Clearly, the service rate of the server would continue to deteriorate if we let `SizeB` increase without purging deleted records. On the other hand, if we force `SizeB = 0` all the time, i.e., by reorganizing the data structure after every insert or delete operation, then the overhead associated with reorganization can potentially be large. In both cases, the system may not exhibit optimal performance because the effective service time (including the data reorganization overhead) is not minimal. Here we also note that if the implementation of insert operations is changed such that a new record is appended to array `A` as long as array `A` remains sorted, then array `A` might contain more than `SizeA` elements before the next maintenance starts, and `SizeB` would not hold the number of entries in excess of `SizeA` at any time before the next data reorganization starts. In this paper, we consider the case in which neither the structure nor the number of elements stored at the *top* level of the data structure will be changed as a result of insert operations. Thus, array `A` always stores only `SizeA` records before the next maintenance starts, although some of which may have been marked deleted.

As in [1], [8], [14], we assume that the server system is designed to run for an infinitely long period of time, thus making it necessary that the "average" probability of deleting an existing record be equal to the "average" probability of adding a new record in the steady state. Initially at the system start-up time, the probability of insert operations may be greater than that of delete operations and the search table grows. We will discuss this transient case in Section 5. After the system has run for a sufficiently long time, however, the search table gradually converges to a constant size and the steady state reaches. In the steady state, the data structure is neither empty nor overflown and contains on the average R occupied records, corresponding to the long-term, average value of `SizeA` in the two-level sorted array example. The *degradation level* of the data structure is defined as the "average" number of active and deleted records in excess of R, thus corresponding to the value held in `SizeB` variable in the average sense in the two-level sorted array example. Specifically, let R_i be the size of a multilevel data structure immediately after the ith data reorganization. Then, the average size of the multilevel data structure in the steady state is

$$R = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} R_i \quad (1)$$

When the system reaches the steady state, the server will perform data reorganization periodically in a cycle and all maintenance periods will be essentially indistinguishable and of the same length. Therefore, it suffices to describe the average behavior of the server in a single but repeated period. Note that the "average" degradation level of the data structure immediately after a maintenance in such a repeated period is zero since *Prob*(insert) = *Prob*(delete) in the steady state, viz.

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} (R_i - R) = 0 \quad (2)$$

Other variants of multilevel data structures include, for example, using a linked list at the second level while leaving the first level as a sorted array, or using a perfectly balanced binary tree at the first level and a binary tree for insertion at the second level, etc. A possible design of a three-level hierarchy may use a perfectly balanced tree at the top level, a sorted array at the middle level to which new records may be appended provided that it remains sorted, and an unsorted array at the bottom level that accommodates all other new records. The objective of these and other instances of multilevel data structures is to relax the requirement that data reorganization be performed after each insert/delete operation so as to make all access operations efficient. An existing record is deleted simply by changing its status field to `empty`, while a new element is inserted at a level that can accommodate it without affecting the search efficiency of higher levels.

We are interested in developing analytic models to assess the performance and stability of such designs and compare them with single-level designs that perform data reorganization on-the-fly. As mentioned before, the original work in [14] dealt only with concurrent data reorganization and made the assumption that there is only a single terminal user in the system. This paper studies deferred maintenance as a design alternative and makes no assumption regarding the workload intensity. The models developed here can potentially be modified to analyze concurrent maintenance as well.

Conceptually, insert operations to a multilevel data structure are considered *degrading* operations because they increase the degradation level. Delete operations, on the other hand, never change the total number of active and deleted records stored in the data structure since they only mark the status of a record from `occupied` to `empty`. Therefore, delete operations along with lookup and update operations are considered *nondegrading* operations. A data reorganization upgrades the data structure by purging all deleted records at all levels and moving all active records to the top level. This eliminates any degradation (i.e., the degradation level becomes zero) and restores maximum efficiency to the data structure. For the two-level sorted array example, after a data reorganization, all deleted records in arrays `A` and `B` are purged, all active records are moved to the top level array `A` and sorted, and array `B` will become empty, i.e., `SizeB = 0`. As mentioned earlier, a multilevel data structure is not reorganized after each degrading operation; hence, the degradation level may increase over time. The degradation level measures the extent to which the server has been degraded since the last reorganization because more (active and deleted) records must be scanned by the server to service clients' requests.

To develop a model for studying the behavior of a queueing server maintaining a multilevel data structure, it is necessary to characterize the system workload and express the performance characteristics of the server as a function of the degradation level of the data structure. The performance characteristics are described by the following parameters:

μ_i: the rate of servicing nondegrading operations (i.e., lookup/update/delete) at degradation level i.

μ_i^a: the rate of servicing degrading operations (i.e., insert) at degradation level i.

m_i: the *maintenance* rate of the server at degradation level i.

ρ_i: the probability that the queue is nonempty given that the degradation level is i.

P_i: the steady state probability that the server is at degradation level i.

d_p: the degradation level at which a data reorganization should be performed to minimize the average service time per operation.

In general, the rate of servicing degrading operations may be different from the rate of servicing nondegrading operations. For the two-level sorted array example above, if keys are never duplicated then the time to service an insert operation is equal to the time needed to copy a new record into location `B[SizeB + 1]` as no search is involved. We use μ_i^a to denote the service rate of degrading (i.e., insert) operations and μ_i to denote the service rate of other operations. We also assume that all records have the same probability of being accessed and operations requests are statistically independent. The effects of *hot keys* and serial correlation between successive operation requests are not considered in this paper.

The system workload is characterized by the following parameters:

λ: the arrival rate of client requests (in open systems).

N: the number of clients (in closed systems).

Z: the average think time of a client between issuing operations (in closed systems).

q: the probability of insert operations, assumed to be equal to the probability of delete operations in the steady state, that is, *Prob*(insert) = *Prob*(delete) = q in the steady state.

3 MODEL

Here, we are interested in the performance of the server in the steady state in which *Prob*(insert) = *Prob*(delete) = q. The transient case in which *Prob*(insert) > *Prob*(delete) will be discussed later in Section 5. In the steady state, the server performs data reorganization periodically in a cycle and all fixed-length maintenance periods are indistinguishable. Immediately after a data reorganization, the average size of the data structure is a constant as defined by (1) and the average degradation level of the server is zero as defined by (2).

It is assumed that all service and maintenance times are exponentially distributed. Thus, a Markov model can be used to describe the behavior of a queueing server maintaining a multilevel data structure with deferred maintenance. This behavior depends on the type of data structure maintained as well as on the following two quantities: the customer population and the degradation level. The customer population describes the queuing behavior of the system while the degradation level determines both the service and maintenance rates of the server in a maintenance period. The steady state solution of the underlying Markov model can be obtained using the global balance technique [12] which yields one equation per state; therefore, it is important to reduce the number of dimensions in the state representation, or the number of equations tends to grow combinatorially.

In our approach, the system state is initially represented by the pair (h, k) where h is number of pending requests (i.e., the queue length) and k is the degradation level of the server in a maintenance period. Then, a state reduction technique is used to remove the queue length from the state representation, yielding a one-dimensional (i.e., degradation only) Markov model. The latter can be used to derive a closed-form solution for the average service time per operation. We will demonstrate that the state reduction technique improves solution efficiency without compromising accuracy in both open and closed systems.

3.1 Open Systems

Given that the arrival rate of service requests is λ, the behavior of the server can be described by the two-dimensional Markov model shown in Fig. 1. Recall that (h, k) denotes a state in which the queue length is h and the degradation level is k.

The model is constructed as follows:

1) The arrival of a new request increments the population, but has no effect on the degradation level. (The vertical transitions at rate λ downward.)
2) The queue length is decremented but the degradation level remains at i, $i < d_p$, after the server completes a nondegrading operation (i.e., lookup, update, or delete) which occurs with probability $\bar{q} = 1 - q$ and is served at rate μ_i. (The vertical transitions at rate $\bar{q}\mu_i$ upward.)
3) The server attempts a data reorganization whenever it is idle at degradation level i, $0 < i < d_p$. (The horizontal transitions at rate m_i from the top row states.) However, any request that arrives during the maintenance aborts it and the server remains at degradation level i.
4) The queue length is decremented and the degradation is incremented from i to $i + 1$, $i < d_p$, after the server completes a degrading operation (i.e., insert) which occurs with probability q and is served at rate μ_i^a. (The diagonal transitions at rate $q\mu_i^a$.)
5) A data reorganization is performed once the degradation level reaches d_p. (The wraparound transitions at rate m_{d_p} from the right column states.) If a request arrives during the maintenance period, it waits in the queue until the maintenance is completed.

The above two-dimensional model is too large to be solved directly for the performance metrics of interest. Numerical solutions are possible, but they are often laborious (e.g., a few hours of computation time using Monte Carlo simulation [17]). In addition, because of limited memory capacity, states with large queue sizes must be truncated, especially when d_p has a large value. This makes numerical solutions inefficient and sometimes even inaccurate. Below, we apply a state reduction technique through which a closed-form solution can be derived. It is an approximate solution because of the use of the state reduction technique. Nevertheless, its accuracy is comparable to that of the numerical solution as demonstrated in Section 4.

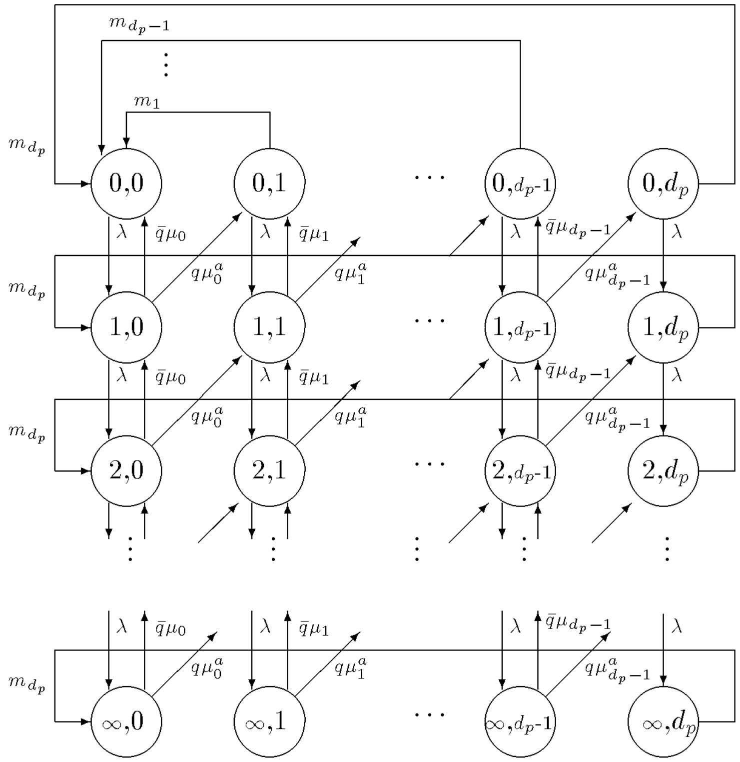

Fig. 1. A two-dimensional Markov model for open systems.

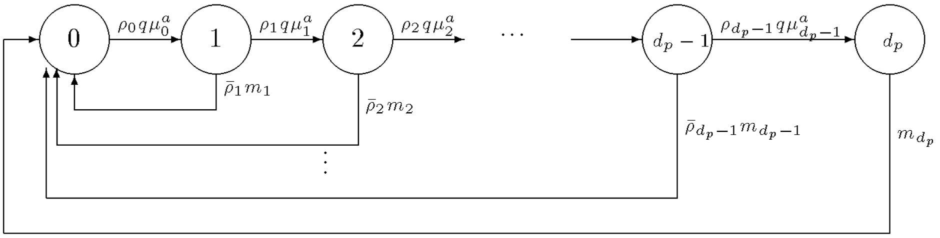

Fig. 2. The simplified model for open systems.

The approximate solution can be obtained by merging all states having the same degradation level, yielding a state diagram in which state i represents that the system is at degradation level i. The simplified model is shown in Fig. 2. It can be explained as follows.

From state i, $0 < i < d_p$, there are two possible transitions. One to state $i + 1$ and another to state 0. The former transition occurs after the server completes a degrading (i.e., insert) operation. The transition rate is $\rho_i q\mu_i^a$, where ρ_i is an approximate, independent estimate of the probability that the server is busy in state i, q is the probability of an insert operation and μ_1^a is its service rate. The latter transition to state 0, on the other hand, occurs after the server completes a successful maintenance given that it became idle in state i. The transition rate is $\bar{\rho}_i m_i$, where $\bar{\rho}_i$ is the probability that the server is idle in state i and m_i is its maintenance rate. In state d_p, the only possible transition is to state 0 at rate m_{d_p}, while in state 0, the only possible transition is to state 1 after completing a degrading operation.

To solve the simplified model, it is necessary to determine ρ_i (the probability that the server is busy at degradation level i). Consider the subset of states having the *same* degradation level i in Fig. 1. Requests arrive at rate λ and are served at rate $\bar{q}\mu_i$, where $\bar{q}$ is the probability of a nondegrading operation and μ_i is the corresponding service rate.[1] Using the utilization law of operational analysis [9], ρ_i is given by:

$$\rho_i = \begin{cases} \frac{\lambda}{\bar{q}\mu_i} & \text{if } \bar{q}\mu_i > \lambda \\ 1 & \text{otherwise} \end{cases}$$

and $\bar{\rho}_i = 1 - \rho_i$.

The above result has two implications. First, if $\bar{q}\mu_0 \le \lambda$, then $\forall i$, $\rho_i = 1$ (because μ_i is a monotonically decreasing function of i) and the server is busy in all states with probability 1. Second, if $\bar{q}\mu_0 > \lambda$ then, because the service rate μ_i

1. Note that $q\mu_i^a$ does not contribute to the service rate when the system stays at the same degradation level i; it does when the degradation level increases from i to $i + 1$.

is a decreasing function of i, there exists a degradation level i^* such that

$$\bar{q}\mu_{i^*} \le \lambda \tag{3}$$

and from which state onward the probability that the server is busy in state i is 1, i.e., $\forall j$, $d_p > j \ge i^*$, $\rho_j = 1$. The first condition is actually a special case of the second condition when $i^* = 0$.

The simplified model shown in Fig. 2 can be solved using global balance. Let P_i represent the steady state probability that the system is at degradation level i. The global balance equations are given by:

$$\forall j,\ 1 \le j \le d_p - 1 : (\rho_j q\mu_j^a + \bar{\rho}_j m_j)P_j = \rho_{j-1} q\mu_{j-1}^a P_{j-1}$$

$$m_{d_p} P_{d_p} = \rho_{d_p-1} q\mu_{d_p-1}^a P_{d_p-1}$$

Let $\rho_{d_p} = 0$ (hence $\bar{\rho}_{d_p} = 1$) for notational convenience. The above equations can be combined in the form:

$$\forall j,\ 1 \le j \le d_p : (\rho_j q\mu_j^a + \bar{\rho}_j m_j)P_j = \rho_{j-1} q\mu_{j-1}^a P_{j-1}\,.$$

which is a system of d_p linearly independent algebraic equations with $d_p + 1$ unknow variables P_i. Using the normalizing equation $\sum_{i=0}^{d_p} P_i = 1$, we get:

$$P_o = \frac{1}{\sum_{k=0}^{d_p} \prod_{j=1}^{k} \frac{p_{j-1} q\mu_{j-1}^a}{\rho_j q\mu_j^a + \bar{\rho}_j m_j}}\,, \tag{4}$$

where we have adopted the convention $\prod_{j=a}^{b} x = 1$ when $a > b$, and ρ_i is given by

$$\rho_i = \begin{cases} \frac{\lambda}{\bar{q}\mu_i} & \text{if } 0 \le i < i^* \\ 1 & \text{if } i^* \le i < d_p \\ 0 & \text{if } i = d_p \end{cases} \tag{5}$$

and i^* was defined in (3).

The average service rate of the server in open systems, denoted by X_o, can be computed using $X_o = \sum_{i=0}^{d_p-1} P_i \times (\bar{q}\mu_i + q\mu_i^a)$. Thus

$$X_o = \sum_{i=0}^{d_p-1} \frac{(\bar{q}\mu_i + q\mu_i^a)P_0}{\prod_{j=1}^{i} \left(\frac{\rho_j \mu_j^a}{\rho_{j-1}\mu_{j-1}^a} + \frac{\bar{\rho}_j m_j}{\rho_{j-1} q\mu_{j-1}^a} \right)} \tag{6}$$

The reciprocal of X_o, of course, is the average service time. (6) is a general closed-form solution for the average service rate of the server as a function of the workload intensity λ and d_p in the open system. Below, we shall see that the same equation can be modified to determine the average service rate for closed systems as well.

3.2 Closed Systems

Closed systems include both batch and terminal systems [12]. They differ from open systems in that their population cannot be arbitrarily large. The workload intensity in a batch system is specified by N, the average multiprogramming level. For a terminal system, the workload intensity is specified by N, the number of active terminal users, and Z, the average interarrival time of requests from terminal users (called the average *think* time).

3.2.1 Batch Systems

In a batch system, the population is assumed fixed. Each completed request is immediately replaced by another request from an infinite backlog of requests. Thus, a queueing server maintaining a multilevel data structure with deferred maintenance in a batch system is always busy at all degradation levels $i < d_p$. Furthermore, because the queue length is constant, the system state is completely specified by the degradation alone. Consequently, the system can be modeled by a state diagram similar to Fig. 2 with ρ_i equal to 1, $\forall i$, $0 \le i < d_p$. The closed-form solution for the average service rate in a batch system, X_b, is the same as (4) and (6) except that ρ_i is given by

$$\rho_i = \begin{cases} 1 & \text{if } 0 \le i < d_p \\ 0 & \text{if } i = d_p \end{cases} \tag{7}$$

3.2.2 Terminal Systems

In a terminal system, each user issues requests independently at rate $1/Z$. Users are logically and dynamically partitioned into two groups: those that have requests at the server's queue and those that do not (i.e., they are thinking). Let k denote the number of pending requests in the server's queue, $0 \le k \le N$, which is also the number of waiting users. Then, the arrival rate of requests issued by the other users is $(N - k)/Z$. The behavior of a queuing server maintaining a multilevel data structure with deferred maintenance in a terminal system is described by a two-dimensional Markov model as shown Fig. 3. It is similar to Fig. 1 except that it has a finite number of states and the arrival rates vary with the queue length. Here, we approximate ρ_i by the utilization of an M/M/1//N queuing system [10]. An M/M/1//N system is always *ergodic*.[2] Thus, for all degradation levels, the probability that the server is busy is strictly less than 1, i.e., $\forall i$, $0 \le i < d_p$, $\rho_i < 1$. This is exactly the opposite to batch systems where the queue is never empty and the server is always busy.

To find the average service rate in a terminal system, X_t, we again use (4) and (6) except that ρ_i is estimated from the solution of an M/M/1//N queuing system with a constant service rate $\bar{q}\mu_i$, i.e.,

$$\rho_i = 1 - \left[\sum_{k=0}^{N} \left(\frac{1}{Z\bar{q}\mu_i} \right)^k \frac{N!}{(N-k)!} \right]^{-1} \tag{8}$$

The optimal maintenance period can be determined by a binary search for the value of d_p that maximizes the expression in (6) given (4) and (8). Note the model for terminal systems agrees with that for the batch systems as $Z \to 0$, i.e., when $Z = 0$ in (8), ρ_i is equal to 1, $\forall i$, $0 \le i < d_p$, yielding (8) the same as (7).

2. Aperiodic, irreducible, and all states are positive recurrent.

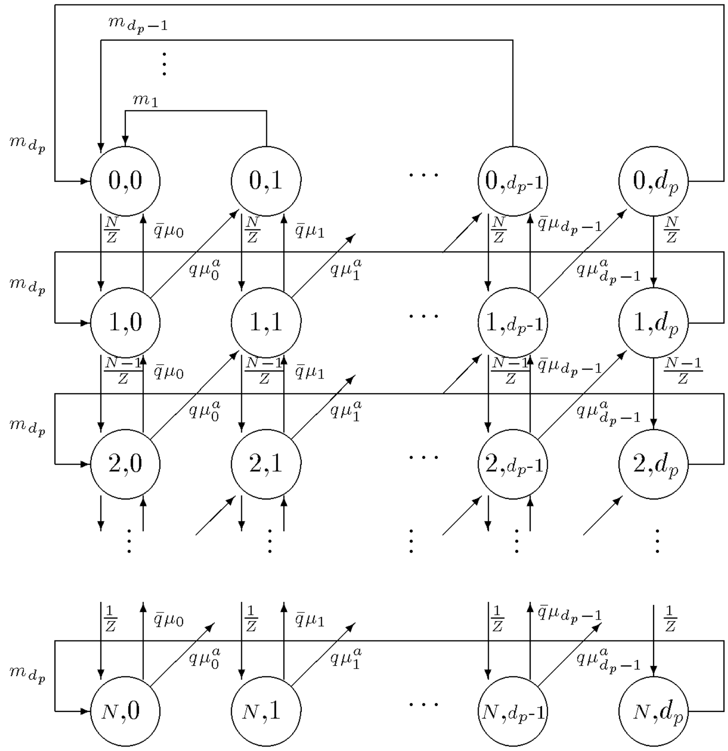

Fig. 3. A two-dimensional Markov model for closed terminal systems.

3.3 Detailed Modeling vs. Approximate Modeling

A key step in our methodology is that we use a single number to represent the degradation level of the data structure, that is, the number of records in excess of its average size in the steady state. Consequently, the reduced model is a one-dimensional Markov model that can be solved analytically. This approach falls into the category of *approximate modeling* as opposed to *detailed modeling* where the distribution of excessive elements in each level is considered explicitly. In the latter approach, the system state for a general n-level data structure would be represented by an $(n - 1)$-tuple that specifies the number of entries in each of the $(n - 1)$ lower levels of the hierarchy. For $n > 2$, an analytical solution of the corresponding $(n - 1)$-dimensional Markov model would be difficult, if not impossible, to obtain.

In our approach, the actual distribution of excessive records at the lower levels is not explicitly modeled in the Markov chain; instead it is accounted for implicitly during the *parameterization* process through which model parameters are assigned values (see Section 4.1). Specifically, if $n > 2$ then the service and maintenance rates can be estimated based on the knowledge regarding the probability that a requested key falls into each of the levels of the data structure. This probability depends on the type of the multilevel data structure being maintained by the server. This way, we can still use a one-dimensional Markov model that is simple and tractable, instead of a more complicated $(n - 1)$-dimensional model which may be intractable. This is a classical trade-off between the accuracy of a model and its solution complexity. Fortunately, a multilevel data structure with $n > 2$ is rarely used in practice (see [14]) and, therefore, in most cases, our model describes the system behavior accurately.

4 Application of the Model and Result Interpretation

Equations (4) and (6) derived above coupled with (5), (7), and (8) for open, batch, and terminal workload systems, respectively, are a general closed-form solution for X. The reciprocal of X is the average service time per operation. This analytical solution can be used to determine the value of d_p that optimizes the performance of a server maintaining a multilevel data structure with deferred maintenance under different workload conditions. This section demonstrates the utility of the closed-form solution by considering a detailed case study. Other instances can be treated similarly.

4.1 Example: A Database Server Maintaining a Two-Level Sorted File

The case study under consideration is a database server that maintains a sorted disk file [7]. Records in the file are accessed given a key such as the employee's social security number. There are no duplicate keys. The file is stored using the two-level sorted array data structure described in Section 2 in which the top level array, `A`, is always sorted while the lower level array, `B` is not. We define the following static parameters of this two-level sorted array data structure:

T: the average service time required to scan an array element.

T^a: the service time required to copy an array element from one location to another, e.g., copying an element to a location in array `B`.

T_h: the system *overhead* time associated with every operation.

R: the average size of the data structure, that is, the number of active records in the file assuming that in the steady state *Prob*(insert) = *Prob*(delete) = q.

For comparison purposes, we also define the following static parameter pertaining to a single-level binary tree with on-the-fly maintenance:

g: the system *overhead* time required to allocate or deallocate a node of a binary tree.

The database server is in an environment in which the average number of records is 500 (R = 500) in the steady state. Further, the probabilities of lookup, update, delete, and insert operations are 0.4, 0.2, 0.2, and 0.2, respectively, (q = 0.2). The server runs on a Sun 3/50 workstation for which T is estimated to be 0.0000577 sec including the time required to read an array entry and compare keys, T^a is estimated to be 0.0000277 sec, and T_h is estimated to be 0.0073 sec including the time required to read an operation from and send a response to a UNIX system-queue. Finally, g is estimated to be 0.0033 sec.

Given the above static parameters, the dynamic parameters of the system, i.e., the service and maintenance rates, can be estimated as follows:

1) The rate of servicing a nondegrading (i.e., lookup, update, or delete) operation at degradation level i is estimated by

$$\mu_i = \frac{1}{T\left(\frac{R}{R+i}\log_2 R + \frac{i}{R+i}\frac{i}{2}\right) + T_h},$$

because, on the average, the server needs to scan $\log_2 R$ elements in the sorted array (which occurs with probability $\frac{R}{R+i}$), and $i/2$ elements in the unsorted array (which occurs with probability $\frac{i}{R+i}$) to access a record. Recall that we assume all records have the same probability of being accessed and requests issued by users are independent. Furthermore, for delete operations, the time to change the status field of an active record to `empty` is negligible compared with the search time.

2) Since there are no duplicate keys, the rate of servicing a degrading (i.e., insert) operation at degradation level i is estimated by

$$\mu_i^a = \frac{1}{T^a + T_h}.$$

If duplicate keys exist, the denominator of the above expression should also include a term for the search time.

3) Assume that a data reorganization at degradation level i requires that the server first quicksorts the i elements in the unsorted array and then merges the two sorted arrays into one using a temporary storage. Then, the maintenance rate of the server at degradation level i is estimated by

$$m_i = \frac{1}{(T + T^a)(i \log_2 i + R) + iT},$$

because on the average the quicksort algorithm scans and copies $i \log_2 i$ elements while the merge algorithm scans $i + R$ of which R elements are copied.

The performance of the server is compared with two conventional designs: 1) a server that maintains a single-level sorted array with on-the-fly maintenance and 2) a server that maintains a single-level binary tree also with on-the-fly maintenance. Design 1) is less typical than design 2) and is included mainly for comparison purposes. In design 1), the array is always compact and sorted after every insert or delete operation. The average service rate of the server is estimated by:

$$\frac{1}{(1-2q)\times T\log_2 R + 2q\times\left(T\log_2 R + T^a\frac{R}{2}\right) + T_h}, \tag{9}$$

because on the average a lookup or an update operation (which occurs with probability $1 - 2q$) requires scanning $\log_2 R$ elements of the array, while a delete or an insert operation (which occurs with probability $2q$) requires the same amount of search time plus the time to move up or down the elements behind or ahead of the element to be deleted or inserted.

On the other hand, for design 2), the average service rate is estimated by:

$$\frac{1}{T\log_2 R + T_h + 2g + q[(2/3)\times 2T + (1/3)T\log_2 R]}, \tag{10}$$

where the average service time is estimated as the sum of:

1) the time to search for a requested key, $T \log_2 R$;
2) the overhead time per operation, T_h;
3) the time to allocate or deallocate a new node, g, if the operation is an insert or a delete operation which occurs with probability $2q$; and, finally, if the operation is a delete operation which occurs with probability q then
4) the time to reorganize the binary tree which requires time $2T$ with probability 2/3 (if the deleted node has zero or one child) and time $T\log_2 R$ with probability 1/3 (if the deleted node has two children).

4.2 Results for Open Systems

Fig. 4 plots the average service rate X_o vs. d_p in open systems as λ varies from 20 (corresponding to a lightly-loaded system) to 110 (corresponding to a heavily-loaded system), using the closed-form solutions for open systems given by (4), (5), and (6). The result obtained from simulating the original two-dimensional Markov model (Fig. 1) is shown in the same figure in dotted lines. The simulation results exhibit the same trend as analytical results and the average relative difference between the analytical and simulation results is only 3.5 percent. However, the numerical computation for the simulation results is quite laborious—it requires a computation time of several hours as opposed to a few minutes required for analytical results. Furthermore, because of memory capacity, it was necessary to limit the total number of states to a maximum of 1,024 × 500 (i.e., d_p × queue-size) during simulation, which may explain the small discrepancy between these two solutions.[3] These results show that

3. We use Monte Carlo simulation techniques for numerical computations. To simulate state transitions from a given state, we randomly simulate time values for all transitions emanating from that state based on their transition rates; we then take the smallest time value among all to determine the next state the system will go to. To keep track of the accumulated time and thus the probability that the server stays in a particular state, it requires the use of a two-dimensional state array. Therefore, in some simulation runs we truncate the number of states due to limited memory capacity.

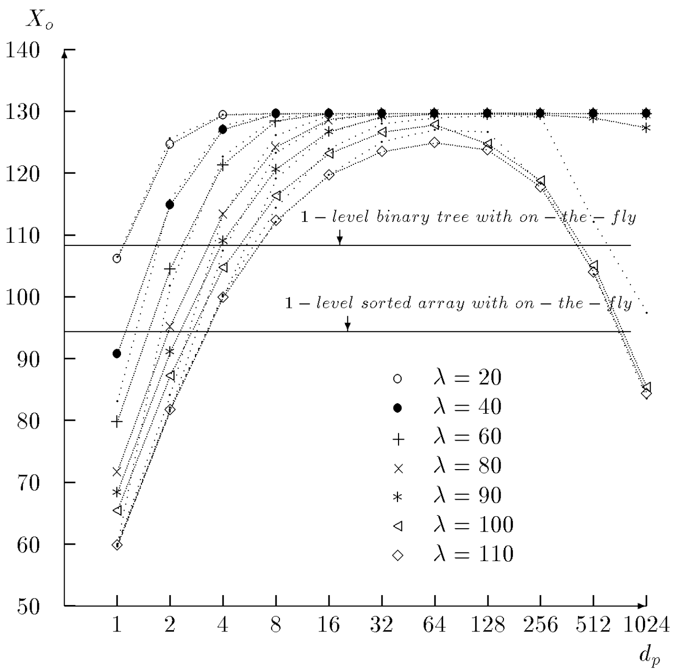

Fig. 4. Performance of deferred maintenance in open systems.

the state reduction technique can greatly improve solution efficiency without compromising accuracy. To compare the multilevel design with single-level designs, Fig. 4 also shows the average service rate in the case of a single-level sorted array based on (9), and the average service rate in the case a single-level binary tree based on (10), both with on-the-fly maintenance.

Because X_o is a concave function, there is an optimal value of d_p that maximizes the service rate as a function of the workload intensity λ. This optimal value of d_p decreases as λ increases. Then it remains constant without decreasing further.

The optimal value of d_p is determined by two different factors: "maintenance overhead minimization" and "maintenance overhead elimination." Recall that under the deferred maintenance policy, the server attempts to upgrade the data structure (i.e., purge deleted records) whenever it becomes idle even if the degradation level is below the optimal value of d_p. Such an attempt will be successful as long as no request arrives while the data structure is being reorganized. In this case, the maintenance overhead is eliminated with respect to subsequent arrivals because they do not wait for maintenance. The maintenance overhead elimination will contribute significantly to the reduction in service time when the workload intensity allows for a high idle-server probability. Fig. 4 shows that when λ is small, the maintenance should be deferred as late as possible because the server can use idling periods to clean-up the data structure and the rate at which the data structure is upgraded this way will always be higher than the rate at which the data structure is degraded due to insert/delete operations. In other words, when λ is below certain threshold value, maintenance overhead elimination totally dominates maintenance overhead minimization and consequently the optimal d_p value is infinity. This is the behavior of the system under light load conditions.

As λ increases, however, the server-idling probability is no longer high enough for the server to upgrade the data structure by merely utilizing its idling periods and still be able to catch up with the rate at which the data structure is degraded. Therefore, the server must periodically upgrade the data structure to optimize its performance. As a result, the average service rate decreases with increasing λ because some maintenance cost must be charged to clients as part of their response time due to extra queueing delay. This is the behavior of the system under medium load conditions.

When λ is high, maintenance overhead elimination is not possible at all because idle periods are too short for any maintenance to be performed successfully. Consequently, the optimal d_p value is determined by the maintenance overhead minimization factor alone. In this case, the deferred maintenance is performed after the server has served a sufficient number of clients so that the amortized cost per client is small. Note that above a high threshold value λ = 110, the optimal d_p has a fixed value (i.e., 64) and does not decrease any further with increasing λ. This is the behavior of the system under heavy load conditions.

It is easy to identify the conditions under which the system is stable. The system is stable as long as the average service rate is greater than the arrival rate, i.e., λ < 125, because the optimal service rate under heavy load conditions is only 125. The advantage of the two-level sorted array with deferred maintenance is obvious. While the single-level binary tree with on-the-fly maintenance allows for a maximum arrival rate of 108 (95 for the single-level sorted array with on-the-fly maintenance), the two-level sorted array with deferred maintenance allows for an arrival rate as high as 125 and the system is still stable.

Fig. 5 compares the mean response time per operation as a function of λ. For the two-level sorted array with deferred maintenance, the mean response time at each λ is computed based on the corresponding optimal d_p value. The improvement of deferred maintenance of the two-level sorted array over on-the-fly maintenance of the single-level binary tree is remarkable. In particular, as λ approaches 108, the server of the binary tree with on-the-fly maintenance becomes unstable and its response time approaches infinity. On the other hand, the server of the two-level sorted array with deferred maintenance remains stable and its response time increases slowly until λ ≈ 120.

4.3 Results for Closed Terminal Systems

Fig. 6 shows the analytical solution for a terminal system using (4), (8), and (6), assuming that the average think time, Z, is 0.0625 sec. The population varies from 1 (corresponding to a lightly-loaded system) to 16 (corresponding to a heavily-loaded system). The result obtained from simulating the original two-component Markov model (Fig. 3) is again shown in the same figure in dotted lines. These two sets of solutions match near perfectly with the average relative difference being only 1.6 percent.

Similar to the case of open systems, under light load conditions X_t is insensitive to the value of d_p beyond the optimal point because under these conditions maintenance overhead elimination dominates maintenance overhead minimization. Therefore, the optimal d_p can be arbitrarily

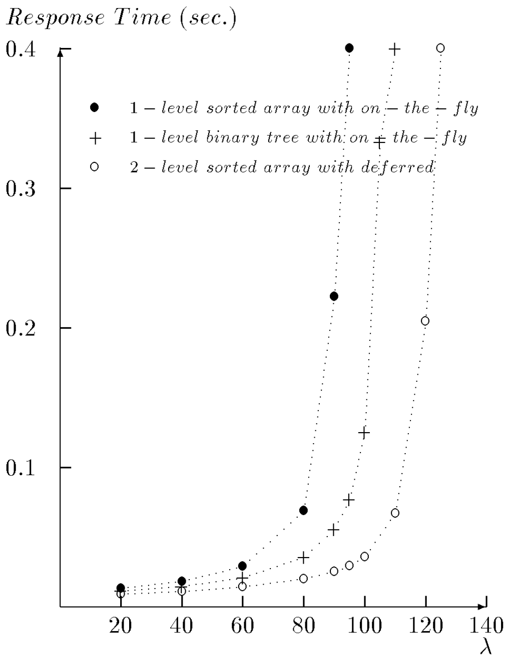

Fig. 5. Comparison of response times in open systems.

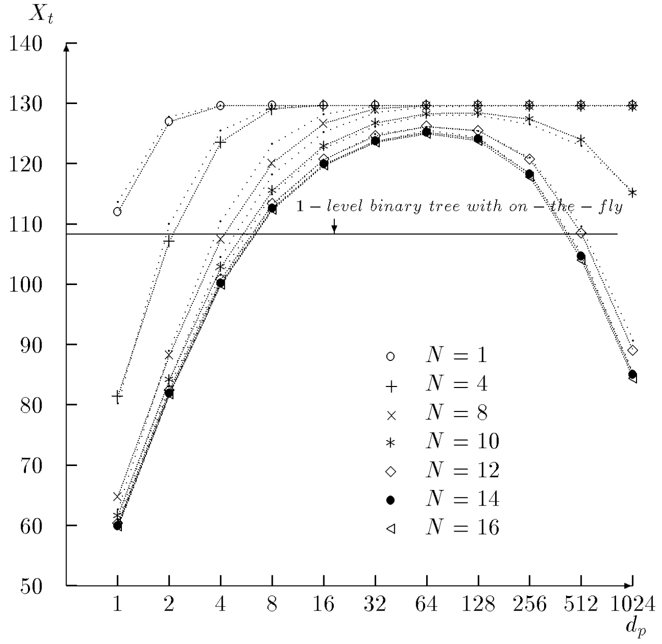

Fig. 6. Performance of deferred maintenance in closed terminal systems

large. Under medium to heavy load conditions, however, X_t is more sensitive to d_p because under these conditions maintenance overhead elimination is less important compared with maintenance overhead minimization. Consequently, X_t is optimized only at a particular d_p value. Specifically, when $N \approx 10$ to 12 (a medium load condition), the optimal d_p value is 128. When $N = 16$ (a heavy load condition), the optimal d_p value is 64 and stays so for higher values of N because the probability that the server can clean-up the data structure using server-idling periods approaches 0, despite the fact that the server-idling probability is nonzero in closed terminal systems.

4.4 Results for Closed Batch Systems

Fig. 7 shows the results for a closed batch system with the population N varying from 1 (corresponding to a lightly loaded system) to 16 (corresponding to a heavily loaded system.) Here, X_b is always optimized at the same value (i.e., $d_p = 64$), regardless of the value of N. This is because the server in a batch system is busy with probability 1, even if there is only a single request in the system. This is the case in which maintenance overhead minimization totally dominates maintenance overhead elimination. The simulation result in this case is identical to the analytical result.

5 Summary

It is important to identify conditions (i.e., workload types and intensities, and when to perform periodic maintenances) under which multilevel data structures with deferred maintenance can perform better than alternative designs before implementation. To that end, we have developed a Markov model and applied a state reduction technique to derive closed-form solutions for the average service time per operation of a deferred maintenance policy that utilizes server-idling periods for maintenance. The model is general enough to accommodate other deferred maintenance policies, such as the one that does not attempt maintenance when the server is idle. Unlike previous work, no assumption is made regarding the workload intensity or the type of data structure used by the server. Therefore, it can be applied to predict the performance of different multilevel data structures under various workload conditions.

To illustrate the utility of the model, it was applied to analyze a two-level sorted array with deferred maintenance. The performance of the two-level multilevel data structure, under the optimizing conditions identified by the closed-form solution, is compared with that of a single-level sorted array and a single-level binary tree, both with on-the-fly maintenance. The results suggest that a two-level sorted ar-

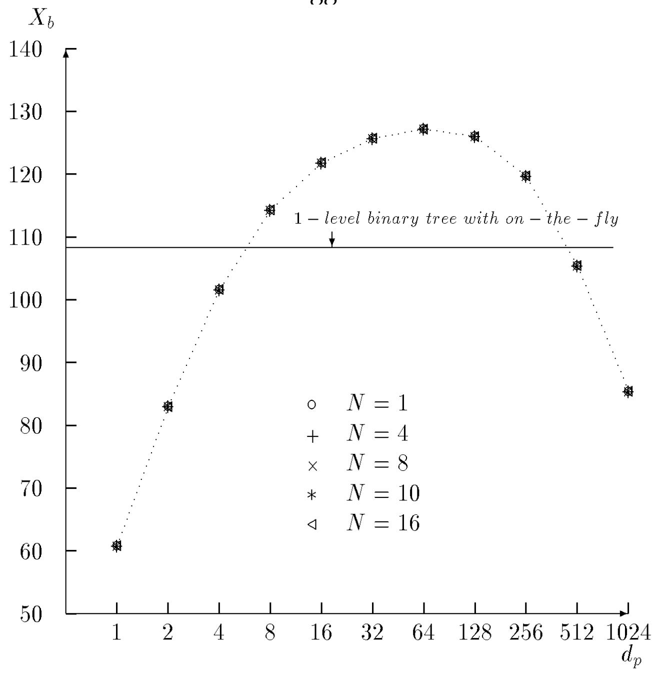

Fig. 7. Performance of deferred maintenance in closed batch systems.

ray with deferred maintenance can outperform single-level data structures with on-the-fly maintenance in both open and closed systems. Furthermore, because of a smaller service time per operation, deferred maintenance on the two-level sorted array allows the system to sustain a higher workload intensity before the system eventually becomes unstable as workload increases. These results imply that deferred maintenance is a viable alternative for improving the performance of queueing servers maintaining multilevel data structures.

The analysis performed in the paper assumes that delete and insert operations have equal probabilities. Therefore, a steady state exists in which the average size of the multilevel data structure is constant. Clearly, since in the steady state the system performs data reorganization periodically in a cycle, the initial size of the data structure in each fixed-length maintenance period is also equal to the average size of the data structure. Before the system reaches the steady state, however, it is possible that the system can be in a transient phase in which the probability of insert operations is larger than the probability of delete operations such that the size of the data structure increases. The models developed in the paper can also be applied to this case with one change. That is, the initial size of the data structure at the beginning of each maintenance period during the transient phase (and thus the maintenance and service rates in each period) will be varied on a period by period basis, as opposed to a constant size in the steady state. This may lead to a different optimizing value of d_p for each maintenance period (and thus a different period length) depending on how fast the data structure grows during the transient phase. A future research effort is required to observe if there is a trend or relation among these optimizing values of d_p.

Other possible future research areas include:

1) performing similar analyses to other multilevel data structures such as B-trees,
2) analyzing the effect of hot keys and serial correlation between requests on the optimizing value of d_p, and
3) investigating the applicability of the Markov model and the proposed state reduction technique to more sophisticated maintenance policies such as deferred and concurrent maintenance policies based on priority and interrupt-driven schemes.

ACKNOWLEDGMENTS

TBD

REFERENCES

[1] W.H. Bahaa-El-Din, F.B. Bastani, and J.-E. Teng, "Performance Analysis of Periodic and Concurrent Data Structure Maintenance Strategies for Network Servers," *IEEE Trans. Software Eng.*, vol. 15, no. 12, pp. 1,526-1,536, Dec. 1989.
[2] F.B. Bastani, I.R. Chen, and W. Hilal, "A Model for the Stability Analysis of Maintenance Strategies for Linear List," *The Computer J.*, vol. 34, no. 1, pp. 80-87, 1991.
[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. pp. 202-203, Addison-Wesley, 1987.
[4] I.R. Chen and S.A. Banawan, "A Reduced Markov Model for the Performance Analysis of Data Structure Servers with Periodic Maintenance," *The Computer J.*, vol. 35, pp. A363-A368, 1992.
[5] D.E. Denning, *Cryptography and Data Security*. pp. 161-164, Addison-Wesley, 1982.
[6] E.W. Dijkstra et al., "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Comm. ACM*, vol. 21, no. 11, pp. 966-975, Nov. 1978.
[7] R. Elmasri and S.B. Navathe, *Fundamentals of Database Systems*, second edition, Benjamin/Cummings, pp. 84-86, 1994.
[8] T. Hickey and J. Cohen, "Performance Analysis of On-the-Fly Garbage Collection," *Comm. ACM*, vol. 27, no. 11, pp. 1,143-1,154, Nov. 1984.
[9] R. Jain, *The Art of Computer Systems Performance Analysis*. pp. 555-556, John Wiley & Sons, 1991,
[10] L. Kleinrock, *Queueing Systems, vol. 1: Theory*. pp. 30-31, John Wiley & Sons, 1975.
[11] S.S. Lavenberg, *Computer Performance Modeling Handbook.* Academic Press, 1983.
[12] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* pp. 58-62; 162-169, Prentice Hall, 1984.
[13] U. Maruyama, and S. Smith, "Optimal Reorganization of Distributed Space Disk Files" *Comm. ACM*, vol. 19, no. 11, pp. 245-247, Nov. 1976.
[14] A. Moitra, S.S. Iyengar, F.B. Bastani, and I.L. Yen, "Multilevel Data Structures: Models and Performance," *IEEE Trans. Soft. Eng.*, vol. 14, no. 6, pp. 858-867, June 1988.
[15] R. Morris and K. Thompson, "Password Security: A Case Study," *Comm. ACM*, vol. 22, pp. 594-597, Nov. 1979.
[16] C. Patridge, *Gigabit Networking*. Addison-Wesley, 1994.
[17] R.Y. Rubinstein, *Simulation and the Monte Carlo Method.* John Wiley & Sons, 1981.
[18] A. Silberschatz and P.B. Galvin, *Operating System Concepts*, fourth edition. pp. 527-531, Addison-Wesley, 1994.

Ing-Ray Chen received the BS degree from the National Taiwan University, Taipei; and the MS and PhD degrees in computer science, both from the University of Houston, Texas. He is currently an associate professor in the Department of Computer Science at Virginia Polytechnic Institute and State University. His research interests are in reliability and performance analysis, mobile computing, and distributed systems. Dr. Chen is a member of the IEEE, IEEE Computer Society, and ACM.

no photo given

Sayed A. Banawan received the BSc degree in computer science and automatic control from Alexandria University, Egypt; and the MSc and PhD degrees in computer science from the University of Washington, Seattle. He is currently an associate professor in the Department of Computer Science at the University of Qatar. His research interests include performance modeling of computer systems, load sharing policies in distributed systems, simulation, and medical information systems. Dr. Banawan is a member of the IEEE and ACM.