The VLDB Journal (2000) 8: 178–198

The VLDB Journal

Secure buffering in firm real-time database systems*

Binto George, Jayant R. Haritsa

Database Systems Lab, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560012, India;
e-mail: {binto,haritsa}@dsl.serc.iisc.ernet.in

Edited by O. Shmueli. Received March 1, 1999 / Accepted October 1, 1999

Abstract. Many real-time database applications arise in electronic financial services, safety-critical installations and military systems where enforcing **security** is crucial to the success of the enterprise. We investigate here the performance implications, in terms of killed transactions, of guaranteeing *multi-level secrecy* in a real-time database system supporting applications with *firm* deadlines. In particular, we focus on the *buffer management* aspects of this issue.

Our main contributions are the following. First, we identify the importance and difficulties of providing secure buffer management in the real-time database environment. Second, we present **SABRE**, a novel buffer management algorithm that provides *covert-channel-free* security. SABRE employs a fully dynamic one-copy allocation policy for efficient usage of buffer resources. It also incorporates several optimizations for reducing the overall number of killed transactions and for decreasing the unfairness in the distribution of killed transactions across security levels. Third, using a detailed simulation model, the real-time performance of SABRE is evaluated against unsecure conventional and real-time buffer management policies for a variety of security-classified transaction workloads and system configurations. Our experiments show that SABRE provides security with only a modest drop in real-time performance. Finally, we evaluate SABRE's performance when augmented with the GUARD adaptive admission control policy. Our experiments show that this combination provides close to ideal fairness for real-time applications that can tolerate covert-channel bandwidths of up to one bit per second (a limit specified in military standards).

Key words: Real-time database — Covert channels — Buffer management — Firm deadlines

1 Introduction

A **real-time database system (RTDBS)** is a transaction-processing system that is designed to handle workloads where transactions have individual timing constraints. Typically, the time constraint is expressed in the form of a *completion deadline*, that is, the application submitting the transaction would like it to be completed before a certain time in the future. In addition, from the application's performance perspective, a transaction that completes just before its deadline is no different to one that finishes much earlier. Therefore, in contrast to conventional DBMS where transaction throughput or response time is typically the primary performance metric, performance in an RTDBS is usually measured in terms of the ability of the system to complete transactions before their deadlines expire.

Many RTDBS applications arise in electronic financial services, safety-critical installations and military systems where enforcing **security** is crucial to the success of the enterprise. For example, consider the environment of an electronic (open-bid) auction on the World-Wide Web with online auctioneers and bidders. Typically, the auction database contains "secret" information such as bidders personal details, including private keys, credit-worthiness and past bidding patterns; the purchase price and ownership history of the items that are being auctioned; the list of "in-house bidders" – these are bidders planted by the auction house to provoke other bidders by artificially hiking the maximum bid; etc. The database also contains "public" information such as bidder public keys and authentication certificates; the starting bid price, the minimum bid increment and the time for delivery for each item; the sequence and state of bids for items currently under auction; etc. It is expected that the secret information is known only to the auctioneers, whereas the public information is available to both bidders and auctioneers.

In the above environment, the auction service provider faces a problem of *three-dimensional* complexity. (1) There is a considerable amount of data to be consulted, processed and updated, and while doing so, the *database consistency* should be maintained. (2) There are *time constraints* associated with various operations – for example, a bid is valid only if registered in the database within a pre-

* A partial and preliminary version of the results presented here appeared earlier in *Proc. of 24th VLDB Conf., August 1998.*
Correspondence to: J.R. Haritsa

specified time period after submission (in the Flash Auction at *http://www.firstauction.com*, bids that arrive more than *5 min* after the previous bid is registered are invalidated). (3) During every stage of the bidding process, *data security* must be ensured — unauthorized access to the secret information by bidders may help them gain unfair and financially lucrative advantages over other competitors.

Our current research has focused on the design of information systems that can simultaneously and effectively meet the above three challenges, that is, on the design of **secure real-time database systems (SRTDBS)**. In particular, our study is conducted in the context of real-time applications with "firm deadlines" [16]. For such applications, completing a transaction after its deadline has expired is of no utility and may even be harmful. Therefore, transactions that miss their deadlines are "killed", that is, immediately aborted and discarded from the system without being executed to completion. Accordingly, the performance metric is the *percentage of killed transactions*.[1] Our choice of firm-deadline applications is based on the observation that many real-time applications belong to this category. For example, in the 1-800 telephone service, a system responds with a "circuits are busy" message if a connection cannot be made before the deadline. Similarly, most Web-based services employ "stateless" communication protocols with timeout features.

For the above real-time context, we recently investigated the design and evaluation of high-performance secure *concurrency control* protocols [10]. We move on, in this paper, to considering the equally important and related issue of designing secure *buffer managers* that can both guarantee security and provide good real-time performance. To the best of our knowledge, this study represents the *first work* in the area of secure real-time buffer management.

1.1 Security mechanisms

For a DBMS to be considered secure, a number of requirements have been identified in the literature [4]. Among these, **secrecy**, that is, the prevention of unauthorized knowledge of secret data, is an especially important requirement for an RTDBS due to the sensitive nature of their application domains. In this paper, we focus exclusively on this issue and in the sequel use the term security synonymously with secrecy. More specifically, we consider environments wherein both database transactions and data items are assigned to a fully ordered set of disjoint security levels (e.g., Secret and Public) and the goal is to enforce *multilevel secrecy*: the data associated with a particular security level should not be revealed to transactions of lower security levels.

Security violations in a multilevel RTDBS can occur if, for example, information from the Secret database is transferred by corrupt high-security transactions to the Public database where they are read by conspiring low-security transactions. Such *direct* violations can be eliminated by implementing the classical Bell–LaPadula security model [20], which imposes restrictions on the data accesses permitted to transactions, based on their security levels. The Bell-LaPadula model is not sufficient, however, to protect from "covert channels". A covert channel is an *indirect* means by which a high-security transaction can transfer information to a low-security transaction [21]. For example, if a low-security transaction requests access to an exclusive resource, it will be delayed if the resource is already held by a high-security transaction, otherwise it will be granted the resource immediately. The presence or absence of the delay can be used as a "signaling" or encoding mechanism by a high-security transaction passing secret information to the low-security transaction (e.g., no delay could signify a 0, while delay could signify a 1). Note that, from the system perspective, nothing "obviously illegal" has been done in this process by the conspiring transactions.

Covert channels can be prevented by ensuring that low-security transactions do not "see" high-security transactions – this notion is formalized in [12] as *non-interference*, that is, *low-security transactions should not be able to distinguish between the presence or absence of high-security transactions*. This can be implemented, for example, by providing *higher priority to low-security transactions* whenever a conflict occurs between a low-security transaction and a high-security transaction. From a system perspective, it translates to implementing database managers that support the non-interference feature. Current database systems typically have a concurrency control manager, a buffer manager and a recovery manager. In our previous study [10], we presented a novel concurrency control manager that simultaneously provided covert-channel-free security and good real-time performance. We consider here the issue of designing *buffer managers* with similar properties.

1.2 Design challenges

Buffer managers take advantage of the temporal locality typically exhibited in database reference patterns to enhance system performance by minimizing disk traffic. In doing so, however, they open up possibilities for covert channels – in fact, *many more than those associated with concurrency control*. For example, the presence or absence of a delay in acquiring a free buffer slot, or the presence or absence of a specific data page in the buffer pool, or the allocation of a particular (physical) buffer slot, could all be used as channel mediums, whereas in concurrency control, data access time is the primary medium. Apart from this "multitude-of-channel-mediums" problem, there are several additional problems that arise while integrating security into the RTDBS framework in general, and into the buffer manager in particular.

1. An SRTDBS has to *simultaneously* satisfy two requirements, namely, provide security and minimize the number of killed transactions. Unfortunately, the mechanisms for achieving the individual goals often work at cross-purposes [14]. In an RTDBS, high priority is usually given to transactions with earlier deadlines in order to help their timely completion. On the other hand, in secure DBMS, low-security transactions are given high priority in order to avoid covert channels (as described above). Now consider the situation wherein a high-security process submits a transaction with a tight deadline in an

[1] Or equivalently, the percentage of missed transaction deadlines.

SRTDBS. In this case, priority assignment becomes difficult, since assigning a high priority may cause a security violation, whereas assigning a low priority may result in a missed deadline.

2. A major problem arising out of the preferential treatment of low-security transactions is that of "fairness" – high-security transactions usually form a disproportionately large fraction of the killed transactions. Note that this is an especially problematic issue, because it is the "VIPs", that is, the high-security transactions, that are being discriminated against in favor of the "common folk", that is, the low-security transactions.
   Unfortunately, designing dynamic mechanisms for achieving fairness that are completely free of covert channels appears to be *fundamentally impossible* [18]. The issue then is whether it is possible to design fair systems while still guaranteeing that the information leakage bandwidth (from covert channels) is within acceptable levels.
3. A simple method for eliminating covert channels is to enforce "static" resource allocation policies, wherein each transaction security class has a set of pre-assigned resources. The drawback of such strategies, however, is that they may result in very poor resource utilization under dynamic workloads. Similarly, employing "replication"-based data organizations wherein multiple copies of data pages are maintained will result in considerable storage overheads.
   The above approaches are *especially* problematic in the real-time context since they may result in a significant increase in the number of killed transactions. The challenge therefore is to design "dynamic" and "one-copy" policies that are demonstrably secure.
4. Unlike concurrency control, which essentially deals only with regulating data access, a buffer manager has *multiple* components – buffer allocation, buffer replacement and buffer pin synchronization [13], all of which have to be made secure.

In summary, for the above-mentioned reasons, making a real-time buffer manager implementation secure involves significant design complexity.

1.3 Contributions

We have conducted a detailed study on designing buffer managers that address the challenges described above, and report on the results here. Our main contributions are the following.

1. We identify the importance and difficulties of providing secure buffer management in the real-time transaction-processing environment.
2. We present **SABRE**, a new buffer management algorithm that provides complete covert-channel-free security.
   SABRE has been carefully designed to address all three components of buffer management – allocation, replacement and synchronization. Further, it is a dynamic one-copy policy and therefore ensures efficient usage of buffer resources.
3. Using a detailed simulation model of a firm-deadline RTDBS, the real-time performance of SABRE is evaluated against unsecure conventional and real-time buffer management algorithms for a variety of security-classified transaction workloads and system configurations. Both inter-transaction locality and intra-transaction locality, as well as restart locality, are modeled in our experiments. To isolate and quantify the effects of buffer management on the system performance, we also evaluate two baselines, **ALLHIT** and **ALLMISS**, in the simulations. ALLHIT models an ideal system where every page access results in a buffer pool "hit" , while ALLMISS models the other extreme – a system where every page access results in a buffer pool "miss".
4. We evaluate the performance of SABRE augmented with the **GUARD** adaptive transaction admission control policy, proposed in [11]. The GUARD policy (details in Sect. 9) is designed to achieve fairness in the distribution of the killed transactions across the various security levels, while remaining within the information leakage bandwidth limits specified in the US military's "Orange Book" [6], which defines security standards.

1.4 Organization

The remainder of this paper is organized as follows. Related work on real-time and secure buffer management is reviewed in Sect. 2. The security framework and the buffer model that we employ are described in Sects. 3 and 4, respectively. Our new SABRE secure real-time buffer management policy is presented in Sect. 5. The unsecure conventional and real-time buffer management policies evaluated in our study are described in Sect. 6. The performance model is described in Sect. 7 and the results of the simulation experiments are highlighted in Sect. 8. The GUARD admission control policy, its integration with SABRE, and its performance is presented in Sect. 9. Finally, in Sect. 10, we present the conclusions of our study.

2 Related work

Buffer management in traditional database systems has been extensively studied (see [7, 13] for surveys). In comparison, however, very little work is available with regard to either real-time buffer management or secure buffer management. Further, to the best of our knowledge, there has been no prior work regarding the *combination* of these areas, that is, secure real-time buffer management, which is the subject of this study. In the remainder of this section, we briefly review the existing literature on secure buffer management and on real-time buffer management.

2.1 Secure buffer management

The only prior research we are aware of on secure buffer management is the recent study by Warner et al. [28] in the context of conventional (i.e., non-real-time) DBMS. A number of design alternatives for secure buffer allocation, replacement and synchronization were explored in this study.

In particular, they *statically* divide the buffer pool among the various security levels,[2] and then employ a *dynamic* allocation scheme called "slot stealing", wherein buffers, currently underutilized at low-security levels, could be borrowed by high-security transactions. Later, if needed, these borrowed slots can be reclaimed by the low-security transactions. They also considered replication-based schemes wherein multiple copies of the same disk page could be present in the buffer pools of various security levels.

While their dynamic allocation scheme is an interesting approach, it has two limitations. First, it is only *partially dynamic*, since low-security transactions are not allowed to utilize the currently unused buffer slots of high-security transactions. Therefore, resource wastage could still result. Second, it must be ensured that the number of buffer slots in the high-security buffer pool that are clean and not performing I/O must *at all times* be as large as the number of slots borrowed from the low-security pool. This is necessary to support the immediate return of slots reclaimed by the low-security transactions and thereby to prevent covert channels. The utility of slot stealing is diminished by this constraint since it places restrictions on the high-security accesses.

A simulation-based performance evaluation of their various policies was also presented in [28], but this evaluation considered the buffer management component *in isolation*, not as part of an entire secure system. It is possible that the relative performance behavior of the policies may be impacted when integrated into a complete system.

Our work differs from the above in that, apart from addressing real-time applications, we consider (a) fully dynamic and unconstrained one-copy allocation policies, (b) the buffer manager's performance role in the context of an *entire system* where all remaining components are secure, and (c) the issue of performance fairness across security levels.

2.2 Real-time buffer management

Efforts to integrate buffer management with real-time objectives have been made for over a decade. The goal here is to ensure that the allocation and replacement of buffer slots is implemented so as to reflect the priorities of the transactions competing for this shared resource. For example, instead of using the standard global LRU policy for replacement, to allocate the LRU slot *among those currently held by the lowest priority transaction*.

A scheme for pre-fetching data pages in an RTDBS was presented in [29], but its applicability is limited since it assumes complete apriori knowledge of the workload on which static transaction pre-analysis techniques are applied. In [3], buffer management policies for prioritized transaction processing were introduced. In particular, prioritized versions of the popular LRU [7] and DBMIN [5] policies were presented and evaluated in conjunction with a prioritized allocation policy that involved suspension of low-priority transactions to create buffer space for higher priority transactions. These policies were designed for systems with *fixed* transaction priority classes and are therefore not directly applicable to the real-time environment where priority is usually a dynamic assignment. To address this lacuna, real-time versions of the allocation and replacement polices of [3] were developed and evaluated in [17] and in [19]. The results of these studies were inconclusive, however, since they arrived at contradictory conclusions regarding the utility of adding real-time information to buffer management – in [17], the results indicated that specialized real-time buffer management schemes have little performance impact, whereas in [19], incorporating priority was considered essential. Since that time, although many other RTDBS aspects have been extensively researched (see [27] for a survey), we are not aware of any subsequent literature on real-time buffer management.

A canonical real-time buffer management policy, called RT, against which we compare the performance of our new algorithm, SABRE, is described in Sect. 6.

3 Security model

In this section, we describe the security framework of our study. Database systems typically attempt to achieve multilevel secrecy by incorporating access control mechanisms based on the well-known Bell–LaPadula model [20]. This model is specified in terms of *subjects* and *objects*. An object is a data item, whereas a subject is a process that requests access to an object. Each object in the system has a *classification* level (e.g., Secret, Classified, Public, etc.) based on the security requirement. Similarly, each subject has a corresponding *clearance* level based on the degree to which it is trusted by the system.

The Bell–LaPadula model imposes two restrictions on all data accesses.

1. A subject is allowed **read** access to an object only if the former's clearance is *higher than or identical to* the latter's classification.
2. A subject is allowed **write** access to an object only if the former's clearance is *identical to or lower than* the latter's classification.

The Bell-LaPadula conditions effectively enforce a "read below, write above" constraint on transaction data accesses (an example is shown in Fig. 1), and thereby prevent *direct* unauthorized access to secure data. They are not sufficient, however, to protect from "covert channels", as described in the Introduction. For tackling covert channels, we use the *non-interference* formalism described in the Introduction, wherein low-security transactions do not "see" high-security transactions.

3.1 Orange security

For many real-time applications, security is an "all-or-nothing" issue, that is, it is a *correctness* criterion. In such "full-secure" applications, metrics such as the number of killed transactions or the fairness across transaction clearance levels are secondary *performance* issues. However, there are

[2] A scheme where the static buffer allocation is periodically reviewed and altered is also presented, but this scheme can result in covert channels since the redistribution mechanism itself can become the medium of information compromise – this issue is discussed in detail in Sect. 9.

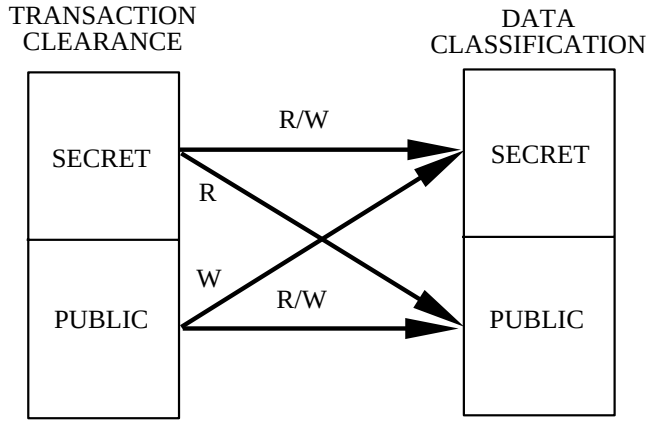

Fig. 1. Bell-LaPadula access restrictions

also applications for whom it is acceptable to have well-defined *bounded-bandwidth* covert channels in exchange for performance improvement [2]. For example, the US military's security standards, which are defined in the so-called "Orange Book" [6], specify that covert channels with bandwidth of less than *one bit per second* are typically acceptable – we will hereafter use the term **"orange-secure"** to refer to such applications.

The underlying premise of orange-secure applications is that covert channels based on such low bandwidths are acceptable because (a) these channels will take a long time to transfer a significant amount of information, (b) by the time the secret information is transferred it may well be obsolete, (c) the time taken to transfer information is long enough to considerably increase the possibility of detection, and (d) the performance or implementation cost associated with eliminating the covert channel is much higher than the cost of any leaked information.

In this study, we initially consider the design of buffer managers for full-secure real-time applications; later, in Sect. 9, we extend our scope to orange-secure applications. We assume, in all this work, that the buffer manager itself is a fully trusted component of the system.

3.2 Transaction priority assignment

As mentioned in the Introduction, assigning priorities in an SRTDBS is rendered difficult due to having to satisfy multiple functionality requirements. Given the paramount importance of security, the database system is forced to assign transaction priorities based primarily on clearance levels and only secondarily on deadlines. In particular, priorities are assigned as a vector $\text{P} = (\text{LEVEL}, \text{INTRA})$, where LEVEL is the transaction clearance level and INTRA is the value assigned by the priority mechanism used *within* the level. Clearance levels are numbered from one upwards, with one corresponding to the lowest security level. Further, priority comparisons are made in *lexicographic order* with lower priority values implying higher priority.

With the above scheme, transactions of a given clearance level have higher priority than all transactions with higher clearances, a necessary condition for non-interference. For the intra-level priority mechanism, any priority assignment that results in good real-time performance can be used. For example, the classical Earliest-Deadline-First assignment [24], wherein transactions with earlier deadlines have higher priority than transactions with later deadlines. For this choice, the priority vector would be $\text{P} = (\text{LEVEL}, \text{DEADLINE})$ – this priority assignment is used in most of our experimental evaluations.

4 Buffer model

Having described the security framework employed in our study, we move on in this section to presenting the buffer model. Buffer managers attempt to utilize the temporal locality typically exhibited in database reference patterns to maximize the number of buffer hits and thereby reduce disk traffic. Three kinds of reference localities are usually observed: *inter-transaction locality* (i.e., "hot spots"), *intra-transaction locality* (transaction's internal reference locality), and *restart locality* (restarted transactions make the same accesses as their original incarnation).

4.1 Buffer categories

At any given time, the slots in the buffer pool can be grouped into the following four categories: **pinned** – buffer slots containing valid pages that are currently being accessed (in read or write mode) by executing transactions; **active** – buffer slots containing valid pages that have been earlier accessed by currently executing transactions; **dormant** – buffer slots containing valid pages that have not been accessed by any currently executing transaction (these pages were brought in by earlier transactions); **empty** – buffer slots that are empty. Further, the first three categories (pinned, active, and dormant) can be further subdivided into **clean** and **dirty** groups, which contain the set of clean and dirty (i.e., modified) pages, respectively, of that category.

We also define the *level* of a buffer slot to be the lowest security level among all the transactions that are currently utilizing the page. For empty slots, the associated security level is undefined – levels are assigned only when a slot is in use.

4.2 Buffer components

There are three components commonly associated with buffer management [13]: *allocation*, *replacement* and *synchronization*. The allocation policy determines the assignment of buffer slots to transactions. The replacement policy determines which page is to be replaced in order to accommodate a new page if no empty buffer slots are currently available. The synchronization policy ensures that conflicting pins are not held on a data page at the same time.

An additional buffer management component, not normally seen in conventional DBMS, that arises specifically in the secure real-time domain is *pin pre-emption*. Consider the case, for example, where a low-clearance transaction requests buffer space, but the slots currently held by high-clearance transactions which would normally be candidates

for replacement cannot be forced out because they are all pinned. In this case, one option is for the low-clearance transaction to wait for slots to become unpinned, but this would immediately result in covert-channel possibilities. Therefore, there appears to be no choice but to "break the pin", that is, permit the replacement with the proviso that the high-clearance transaction holding the original pin is immediately notified that the page it had assumed to be stable had actually been replaced – the high-clearance transaction can then either abort or, preferably, redo the corrupted operation at a later time.

Security considerations are not the only reason for supporting pin preemption. Consider, for example, the case where a tight-deadline transaction wishes to utilize the slot currently pinned by a slack-deadline transaction. If the urgent transaction is blocked, this would mean that high-priority transactions are being blocked by low-priority transactions, a phenomenon known as *priority inversion* in the real-time literature [25]. Priority inversion can cause the affected high-priority transactions to miss their deadlines and is clearly undesirable. Therefore, pin pre-emption is useful from *real-time* considerations also.[3]

5 Secure real-time buffer management policy (SABRE)

In this section, we present **SABRE** (secure algorithm for buffering in real-time environments), our new secure real-time buffer management policy. SABRE provides complete covert-channel-free security and has optimizations built in both for reducing the overall number of killed transactions and for decreasing the unfairness in the distribution of killed transactions across security levels. Further, it is a dynamic and one-copy policy that ensures efficient use of the buffer resources.

Our new **SABRE** policy provides complete covert-channel-free security. Unlike the policies proposed in [28], SABRE is *fully dynamic* since there are no allocation quotas of buffer slots to security levels, and *unconstrained* in that no restrictions are placed on high- (or low-) clearance transaction accesses. Further, only *one copy* of a data page is maintained in the buffer pool. These features ensure efficient use of buffer resources. Finally, it incorporates several optimizations both for reducing the overall number of killed transactions and for decreasing the unfairness in the distribution of killed transactions across clearance levels. We describe its design in more detail in the remainder of this section.

5.1 Security features

The first step in enforcing security is to assign transaction priorities in the manner described in Sect. 3.2. However, note that this is *not sufficient* to make the buffer manager secure in terms of the non-interference requirement. We describe below the main additional security features incorporated in SABRE to ensure covert-channel-free security.

[3] Solely for intra-level pin conflicts, an alternative priority inversion-handling mechanism is *priority inheritance* [25].

1. The contents of a buffer slot are not "visible" to a transaction if the existence of the page in the slot is the consequence of the actions performed by higher security level transactions. Such a slot will appear as a *(pseudo-) empty* slot to the requesting transaction.
   More specifically, for a transaction of a specific clearance level, all the buffer pages of all higher clearance levels are not visible, whereas the pinned and active buffer pages of its own level and all lower clearance levels are visible (the visibility of dormant pages is discussed below in feature 3). This means that, for example, a low-clearance transaction is not given immediate access to a page that has been earlier brought into the buffer pool by a high-clearance transaction. Instead, it is forced to wait for the same period that it would have taken to bring the page from disk had the page not been in the buffer pool (the determination of this period is discussed in Sect. 5.3).
2. High-clearance transactions can replace the dormant slots of low-clearance transactions.
3. Dormant pages, of *any* level, are visible only to transactions of the *highest* clearance level – for no other transactions are they visible. That is, a low-clearance transaction is not given immediate access to a dormant page even at its own or lower clearance levels. Instead, it is forced to wait for the same period that it would have taken to bring the page from disk had the page not been in the buffer pool.
4. Pin pre-emption is supported (via transaction abort) at all security levels.[4]
5. When selecting from among the set of (really) empty slots, the slot is *randomly chosen* and not in any predefined order.

5.1.1 Discussion

Feature 1 above is an obvious requirement to ensure the absence of signaling between high-clearance and low-clearance transactions. Feature 2, which allows high-clearance transactions to "steal" the dormant slots of low-clearance transactions, is included for the following reason. If high-clearance transactions could not replace *any* pages of low-clearance transactions, then the buffer pool would very quickly fill up with the active and dormant pages of the low-clearance transactions and after this the high-clearance transactions would not be able to proceed further. That is, *starvation* of high-clearance transactions would occur.

Note, however, that there is a price to pay for ensuring that covert channels do not result in spite of slot stealing. This is expressed in feature 3, wherein low-clearance transactions are made to wait for access to dormant pages at even their *own and lower* clearance levels, thereby partially losing the benefits that could be gained from inter-transaction locality.[5] The reason that only the *highest clearance level*

[4] Supporting pin pre-emption at the *lowest* clearance level is not essential for security but is retained for performance reasons (see Sect. 5.2).

[5] While these delays will certainly increase the execution time of the individual low-clearance transaction, note that the effect on the disk throughput can be eliminated by using the proxy disk service optimization discussed in Sect. 5.3.

transactions are permitted to see dormant slots is to prevent "transitive" covert channels: Consider a three-security-level system composed of Secret, Classified and Public transactions. Here, if the classified transactions are allowed to see the public dormant slots, a transitive covert channel can arise out of classified transactions observing secret transactions replacing the public dormant slots.

Feature 4 ensures that high-clearance accesses can be unrestricted without causing covert channels since slots can be returned immediately, even if they are currently pinned. Finally, feature 5 ensures that low-clearance transactions cannot "guess" that they are being given a pseudo-empty slot by virtue of the fact that the slot they receive is not the first in the list of slots that they perceive to be empty.

We show next that the above design guarantees covert-channel-free security.

5.1.2 Proof that SABRE is covert-channel secure

The necessary and sufficient conditions for a secure interference-free scheduler are given in [18], and this framework is also applicable to the secure firm-deadline real-time transaction-processing environment. In this framework, there are three requirements to be satisfied for a system to be covert-channel secure – delay security, value security and recovery security – described below.

Delay security. A system is delay secure if the delay experienced by a subject is not affected by actions of subjects belonging to higher clearance levels.

Value security. A system is value secure if the value of objects accessed by a subject are not changed by actions of subjects belonging to higher clearance levels.

Recovery security. A system is recovery secure if the occurrence of a transaction restart appears the same for a subject regardless of the presence or absence of higher level subjects in the system.

Proof. We now consider the various kinds of security violations that are possible with regard to delay security, value security and recovery security, and show that none of these can occur with the SABRE algorithm.

Delay security

From a transaction's viewpoint, delay security can be violated in five different ways. (1) A page is absent in the buffer when it is expected to be present. (2) A page is present in the buffer when it is expected to be absent. (3) A free buffer slot is available when the buffer pool is expected to be full.[6] (4) A free buffer slot is unavailable when the buffer pool is expected to be not full. (5) A physical slot is allocated that is different than what was expected to be allocated.

The first possibility, a page is absent in the buffer when it is expected to be present, can occur only if a higher level transaction replaces the page.[7] SABRE disallows a high-security transaction from replacing the page brought into the buffer by a low-security transaction.

The second possibility, a page is present in the buffer when it is expected to be absent, can happen only if a higher security level transaction brings in the page to the buffer and the low-security-level transaction is given access to the page. SABRE disallows this possibility by setting a visibility level. The pages brought in by higher security level transactions are not visible to low-security-level transactions.

The third possibility, a buffer slot is known to be available when the buffer is expected to be full, can happen only if higher security transactions can free some buffer slots. This cannot happen since, by the visibility criterion, if the buffer pool appears to be full to a low-security transaction, then all buffers must be currently held by low-security transactions and such buffers cannot be replaced by high-security transactions.

The fourth possibility, a free buffer slot is unavailable when the buffer is expected to be not full, can happen only if those perceived to be empty are currently held by higher security-level transactions. However, the pin preemption facility of SABRE ensures the release of these slots.

The fifth possibility is prevented by randomizing the slot identities.

Value security

Since value security deals with the data values read by transactions, it is really determined only by the *concurrency control manager* and there is no direct role for the buffer manager in this issue. For concurrency control protocols that ensure strict one-copy global serializability, ensuring delay security automatically ensures value security. This is because the actions of higher level transactions have no effect on the serial order of lower level transactions. We consider only such concurrency control protocols in this study.

Recovery security

SABRE does not allow a high-security transaction to pre-empt the buffer slot from a low-security transaction. Therefore, a transaction restart can arise only due to *concurrency control* reasons, not out of pin pre-emption occurring at the buffer manager. Therefore, it ensures recovery security.

5.2 Real-time features

Having described the security features, we move on now to the features incorporated in SABRE to ensure good *real-time* performance.

1. Transactions of a particular clearance level are not permitted to replace the pinned or active pages of higher priority transactions belonging to the same level.

[6] We assume here that the total size of the buffer pool is public knowledge and that, for any given transaction, the occupancy of the buffer pool by transactions of that level or dominated levels is visible. Hence, the buffer pool appears full to a transaction if and only if the entire buffer pool is held by transactions of its own or dominated levels.

[7] While it is certainly possible for a lower level transaction to have replaced the page, we make the common assumption that each transaction is aware of all transaction activities at its own and lower security levels. Under this assumption, the transaction will know not to expect the page to be present if it has been replaced by transactions of its own or lower security levels.

```
if (Really Empty Slots exist) then
     slot = randomly chosen Empty Slot
else if (Dormant Slots exist) then
     /* Slot Stealing from Lower Security Levels OR
        Slot Confiscating from Higher Security Levels */
     find the lowest security level at which Dormant Slots exist
     slot = LRU (clean|dirty) among these Dormant Slots
else if (Slots of Higher Security Levels exist) then
     /* Slot Confiscating from Higher Security Levels */
     find the highest level above requester's level for which slots exist
     if (Active Slots exist) then
          find the lowest priority transaction with Active Slots
          slot = LRU (clean|dirty) among these slots
     else /* All Slots are Pinned */
          abort lowest priority transaction and release its slots
          slot = LRU (clean|dirty) among these slots
else if (Slots of Requester's Level exist) then
     if (Lower Priority Transactions with Active Slots exist) then
          find the Active Slots of lowest priority transaction
          slot = LRU (clean|dirty) among these slots
     else if (Lower Priority Transactions with Pinned Slots exist) then
          abort the lowest priority transaction and release its slots
          slot = LRU (clean|dirty) among these slots
     else /* All Slots belong to Higher Priority transactions */
          insert request in the wait queue which is
          maintained in (LEVEL, DEADLINE) priority order
else /* All Slots belong to Lower Security Level transactions */
     insert request in the wait queue which is
     maintained in (LEVEL, DEADLINE) priority order
```

Fig. 2. SABRE slot selection algorithm

2. Within each clearance level, priority-based pin pre-emption (via transaction abort) is supported.
3. An optimized "comb" slot selection algorithm is used to decide the slot in which to host a new data page.

Feature 1 helps to utilize the intra-transaction, inter-transaction and restart localities of high-priority transactions, while feature 2 ensures the absence of priority inversions. The optimized slot selection algorithm mentioned in feature 3 is described below in Sect. 5.2.1.

5.2.1 Search and slot selection policy

When a transaction requests a data page, the *visible* portion of the buffer pool (corresponding to the transaction's clearance level) is searched for the page, and if the search is successful, SABRE returns the address of the slot in which the page is present. If the page is already pinned in a conflicting mode by a higher priority transaction, then the transaction has to wait for it to be unpinned before accessing the contents. Otherwise, it can access the page immediately after gaining a pin on the page.

A search could be unsuccessful for one of two reasons: (a) because the page is really not in the buffer pool, or (b) because it is in the non-visible portion of the buffer pool. In the first case, if really empty buffer slots are available, SABRE brings the page into one of these slots and returns the address of the slot to the requesting transaction. Otherwise, an existing buffer page is chosen for replacement according to the selection algorithm described below. The victim page, after being flushed to disk if dirty, is replaced

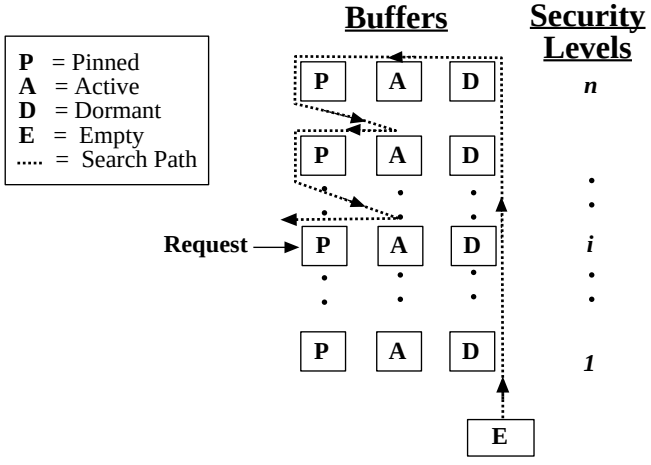

Fig. 3. COMB slot selection route

by the requested page and the associated slot address is returned to the transaction. In the second case, the requested page is made to *appear* to have been brought from disk into the buffer slot where it currently exists by "unveiling" the slot only after waiting for the equivalent disk access time.

The complete slot selection algorithm used in SABRE is shown in Fig. 2. In this algorithm, starting from really empty slots, the selection works its way up the security hierarchy looking for dormant slots and then in a "comb-like" fashion works its way downwards from the top clearance level looking for active and pinned slots until it reaches the

requester's clearance level. This route is shown pictorially in Fig. 3.[8]

SABRE's comb slot selection algorithm incorporates the security and real-time features described earlier. In addition, it includes the following optimizations designed to improve the real-time performance.

1. The search for replacement buffers is ordered based on slot category – the dormant slots are considered first and only if that is unsuccessful, are the active and pinned slots considered. This ordering is chosen to maximize the utilization of *intra-transaction* locality.
2. Given a set of candidate replacement slots (as determined by the slot selection algorithm), these candidate slots are grouped into clean and dirty subsets – only if the clean group is empty is the dirty group considered. Within each group the classical **LRU** (least recently used) policy [7] is used to choose the replacement slot. This "LRU (clean|dirty)" ordering is based on the observation that (a) it is faster to replace a clean page than a dirty page since no disk writes are incurred, and (b) the cost of writing out a disk page is amortized over a larger number of updates to the page.

5.3 Class fairness features

As mentioned in the Introduction, due to the covert-channel-free requirements, high-security transactions in secure RT-DBS are discriminated against in that they usually form a disproportionately large fraction of the killed transactions. To partially address this issue, the following optimizations are incorporated in SABRE.

Proxy disk service. The delays prescribed in security features 1 and 3 (Sect. 5.1) can be easily implemented by actually retrieving the desired page from disk into the buffer slot where it already currently exists. Obviously, this approach is wasteful. A more useful strategy would be to use the disk time intended for servicing this request to instead serve (in increasing security level order) the pending requests, if any, of higher clearance transactions for the same disk.[9] That is, the high clearance transaction clandestinely acts as a "proxy" for the low-clearance transaction.

Comb route. In the comb algorithm, the search for a replacement slot among the dormant slots begins with the *lowest* clearance level. This approach maximizes the possibility of slot stealing from lower clearance levels, resulting in increased fairness.

6 Comparative unsecure buffer management policies

To the best of our knowledge, the SABRE policy described in the previous section represents the *first* secure real-time buffer management policy. This means that there are no comparative algorithms in its class against whom its performance could be evaluated. Therefore, we have instead compared its performance with that of *unsecure* buffer management policies. These include representative choices of previously proposed policies for conventional (non-real-time and unsecure) DBMS and for real-time (unsecure) DBMS.[10] A further utility of this comparison is that it helps to quantify the performance effects of including security cognizance and real-time cognizance in the SABRE policy.

We have implemented two policies, **CONV** and **RT**, corresponding to conventional and real-time DBMS, respectively. For both these policies, the *entire* buffer pool is always visible since they are not security-cognizant. In CONV, for a successful search, the requesting transaction can access the page immediately, unless it has already been pinned in a conflicting mode – in this case the transaction has to wait for the page to be unpinned before accessing its contents. RT also follows a similar policy for successful searches, except that in the case of conflicting pins, if the pin holder is of lower priority than the requesting transaction, the pin is pre-empted by aborting the lower priority holder.

For unsuccessful searches, CONV and RT follow the slot selection algorithms shown in Figs. 4 and 5, respectively. CONV essentially implements the classical LRU approach, including also the (clean|dirty) and category-based search optimizations of SABRE, while RT implements a prioritized version of the same approach.

6.1 Baseline policies

To isolate and quantify the effects of buffer management on the system performance, we also evaluate two artificial baseline policies, **ALLHIT** and **ALLMISS**, in the simulations. ALLHIT models an ideal system where every page access results in a buffer pool "hit", while ALLMISS models the other extreme – a system where every page access results in a buffer pool "miss".

7 Simulation model

To evaluate the performance of the buffer management policies described in the previous sections, we developed a detailed simulation model of a firm-deadline RTDBS. This model is similar to that used in our previous secure RTDBS study [10], with the main difference being the addition of a buffer manager component. A summary of the key model parameters is given in Table 1.

The model has six components: a *database* that models the data and its disk layout; a *source* that generates the transaction workload; a *transaction manager* that models the execution of transactions; a *resource manager* that models the physical resources and includes a *buffer manager*; a *concurrency control (CC) manager* that controls access to shared data; and a *sink* that gathers statistics on transactions exiting the system. The model organization is shown in Fig. 6.

[8] The reason that active/pinned slots of higher security levels are considered before checking for similar slots at the requester's level is to prevent covert channels – if a page from the requester's level is taken while there are still active/pinned pages of the higher security level (which all appear empty to the requester), a covert channel may be opened. Of course, this has an adverse impact on the KillPercent of the higher security transactions.

[9] Of course, if there are no pending higher clearance requests, the disk is forced to be idle during this period.

[10] We do not compare our policies with those described in [28] because their static policies require resource reservation, while their dynamic policies are not completely free from covert channels.

```
if (Empty Slots exist) then
     slot = any Empty Slot
else if (Dormant Slots exist) then
     slot = LRU (clean|dirty) among these Dormant Slots
else if (Active Slots exist) then
     slot = LRU (clean|dirty) among these Active Slots
else  /* all slots are pinned */
     insert request in the wait queue which is
     maintained in FCFS order
```

Fig. 4. The CONV slot selection algorithm

```
if (Empty Slots exist) then
     slot = any empty slot
else if (Dormant Slots exist) then
     slot = LRU (clean|dirty) among these Dormant Slots
else if (Lower Priority Transactions with Active Slots exist)
     find the active slots of the lowest priority transaction
     slot = LRU (clean|dirty) among these slots
else if (Lower Priority Transactions exist)
     abort the lowest priority transaction and release its slots
     slot = LRU (clean|dirty) among these slots
else  /* all slots are with higher priority transactions */
     insert request in the wait queue which is
     maintained in DEADLINE priority order
```

Fig. 5. The RT slot selection algorithm

Table 1. Simulation model parameters

Parameter	Meaning	Default value
$DBSize$	Number of pages in the database	1000
$ClassLevels$	Number of classification levels	2
$ArrivalRate$	Transaction arrival rate	—
$ClearLevels$	Number of clearance levels	2
$SlackFactor$	Slack factor in deadline assignment	4.0
$TransSize$	Average transaction size	16 pages
$WriteProb$	Page write probability	0.5
$NumCPU$	Number of processors	10
$NumDisk$	Number of disks	20
$NumBuf$	Number of buffers	50
$PageCPU$	Page processing time at CPU	10 ms
$PageDisk$	Page processing time at Disk	20 ms
$MinPin$	Minimum buffer pin time	0ms
$MaxPin$	Maximum buffer pin time	100 ms
$CCReqCPU$	Concurrency control overhead	1 ms
$NumGPS$	Number of GPS (global page sets)	100
$SizeGPS$	Size of each GPS	200
$GRefCnt$	References from current GPS	500
$InterLoc$	Inter-transaction locality factor	0.14
$IntraLoc$	Intra-transaction locality factor	0.8
$LocalProb$	Local pageset probability	0.8

7.1 Database model

The database is modeled as a collection of $DBSize$ pages that are uniformly randomly distributed across all of the system disks. The database is equally partitioned into $ClassLevels$ security classification levels (for example, if the database has 1000 pages and the number of classifications is 5, pages 1 through 200 belong to level 1, pages 201 through 400 belong to level 2, and so on).

7.2 System model

The system consists of a shared-memory multiprocessor DBMS operating on disk-resident data. The physical resources of the database system consist of $NumCPU$ processors, $NumDisk$ disks, and $NumBuf$ buffers. There is a single common queue for the processors and the service discipline is pre-emptive resume, with pre-emptions being based on transaction priorities. Each of the disks has its own queue and is scheduled according to a head-of-line (HOL) policy, with the request queue being ordered by transaction priority. The $PageCPU$ and $PageDisk$ parameters capture the CPU and disk processing times per data page, respectively.

A single-level buffer consisting of $NumBuf$ identical page-sized buffer slots is modeled and the slots are allocated, replaced and synchronized according to the buffer manager in use. Each buffer slot has a dirty field, and if the dirty bit is set, the page is saved to the disk before replacement. The buffer wait queues are maintained in the order specified by the buffer manager in use. A page that is brought into the buffer pool is pinned for a uniformly distributed duration in the range $[MinPin, MaxPin]$.

In our simulator, a separate instance of the buffer manager was created for each of the buffer management policies evaluated in our experiments.

7.3 Workload model

Transactions are generated in a Poisson stream with rate $ArrivalRate$ and each transaction has an associated security clearance level and a firm completion deadline. A transaction is equally likely to belong to any of the $ClearLevels$ security clearance levels. (For simplicity, we assume in this study that the categories (e.g., Secret, Public) for data classification and transaction clearance are identical). Deadlines are assigned using the formula $D_T = A_T + SlackFactor * R_T$, where D_T, A_T and R_T are the deadline, arrival time and resource time, respectively, of transaction T. The resource time is the total service time at the resources that the transaction requires for its data processing. The $SlackFactor$ parameter is a constant that provides control over the tightness/slackness of transaction deadlines. If a transaction has not completed by its deadline, it is immediately killed (aborted and discarded from the system).

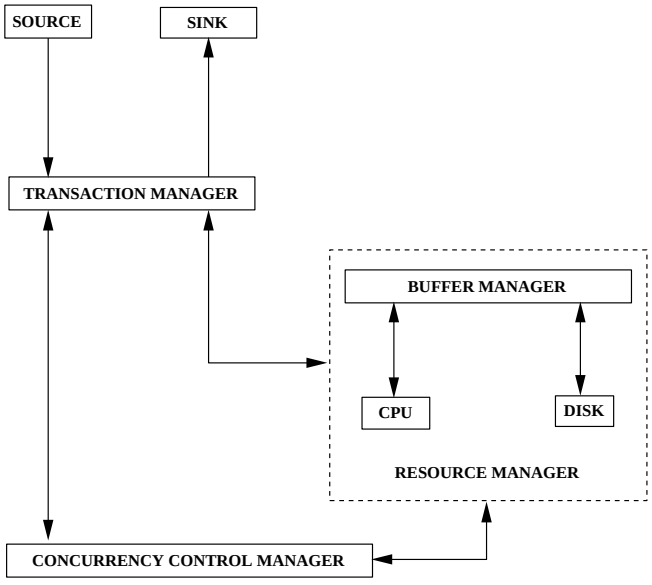

Fig. 6. Simulation model

An important point to note here is that, while the workload generator utilizes information about transaction resource requirements in assigning deadlines, *we do not assume that this information is available to the SRTDBS itself* since such knowledge is usually hard to come by in practical environments.

Each transaction consists of a sequence of page read and page write accesses. The number of pages accessed by a transaction varies uniformly between half and one-and-a-half times the value of $TransSize$. The $WriteProb$ parameter determines the probability that a transaction operation is a write. Due to security reasons, each transaction can only access data from a specific segment of the database, and page requests are generated by uniformly randomly sampling (without replacement) from the database over this segment's range. The permitted access range is determined by both the security clearance level of the transaction and the desired operation (read or write), and is according to the Bell–LaPadula specifications: a transaction cannot read (resp. write) pages that are classified higher (resp. lower) than its own clearance level. A transaction that is restarted due to a data conflict has the same clearance level, and makes the same data accesses, as its original incarnation.

7.4 Transaction execution

A transaction read or write access first involves a CC request to get access permission. The CPU overhead required to process each CC request is specified by the $CCReqCPU$ parameter. After access is provided, it is followed by either finding the page already in the buffer pool or by initiating a disk I/O to input the page. This is followed by a period of CPU usage for processing the page.

7.5 Access locality

The generation of inter-transaction and intra-transaction reference locality is handled in a manner similar to that used in [28]. In this scheme, two types of pagesets are created, *global pagesets* and *local pagesets*. The inter-transaction and intra-transaction localities are associated with the global pagesets and local pagesets, respectively.

Global pagesets are generated by sampling (with replacement) from the database using identical (truncated) normal distributions with variance $1/InterLoc^2$ – only the page location of the centerpoint of the distribution is a function of the individual global pageset. The $NumGPS$ parameter specifies the number of global pagesets generated and the size of each pageset is given by the $SizeGPS$ parameter. At the outset, one of the global pagesets is designated as the *current* global pageset and subsequent page references are generated from this pageset. After $GRefCnt$ number of references have been generated from this pageset, another pageset is uniformly randomly chosen from the remaining pagesets to become the current pageset. This mechanism is intended to model the temporally shifting behavior of inter-transaction locality seen in practice.

The pageset for a transaction T is created in the following manner. Initially a local pageset is created. This local pageset is obtained by uniformly randomly choosing pages from the current global pageset and the number of pages chosen is given by the formula $LocalPageSetSize_T = max(1, ((1 - IntraLoc) \times TransSize_T))$. After this, the transaction's pageset is created by successively choosing pages randomly from within the local pageset or from within the current global pageset. The probability that a given page access is chosen from the local pageset is specified by the $LocalProb$ parameter. This mechanism is intended to model the intra-transaction locality.

Finally, *restart locality* is modeled by ensuring that a transaction that is restarted due to a data conflict has the same clearance level and makes the same sequence of data accesses as its original incarnation.

7.6 Priority assignment

The transaction priority assignment used at all the RTDBS components, *excepting the buffer manager*, is $\text{P} = (\text{LEVEL}, \text{DEADLINE})$ to ensure non-interference in accordance with the discussion in Sect. 3.2. Among the buffer managers, SABRE also employs the $\text{P} = (\text{LEVEL}, \text{DEADLINE})$ priority assignment to ensure security. RT, however, uses $\text{P} = (\text{DEADLINE})$ since it is security-indifferent while CONV is completely priority-indifferent.

7.7 Concurrency control

The secure version [10] of the 2PL–HighPriority (2PL-HP) real-time protocol [1] is used for concurrency control and, for simplicity, only page-level locking is modeled.[11] The 2PL-HighPriority protocol ensures strict one-copy global

[11] In our earlier study of secure concurrency control [10], a validation-based protocol called S2PL-WAIT was found to provide the best real-time performance. However, since open problems remain with respect to integrating validation schemes with buffer management [15] and since the focus here is on buffer management, not concurrency control, we have for simplicity used 2PL-HP in this study.

serializability and therefore ensures value security, as discussed earlier in Sect. 5.1.2.

In the 2PL-HP scheme, the classical strict 2PL protocol [8] is modified to incorporate a priority conflict resolution scheme which ensures that high-priority transactions are not delayed by low-priority transactions. When a transaction requests a lock on a data item that is held by one or more higher priority transactions in a conflicting lock mode, the requesting transaction waits for the item to be released (the wait queue for a data item is managed in priority order). On the other hand, if the data item is held by only lower priority transactions in a conflicting lock mode, the lower priority transactions are restarted and the requesting transaction is granted the desired lock.[12] Note that 2PL-HP is inherently deadlock-free if priorities are assigned uniquely (as is usually the case in RTDBSs).

7.8 Performance metrics

The primary performance metric of our experiments is **KillPercent**, which is the percentage of input transactions that the system is *unable* to complete before their deadlines. We compute this percentage also on a *per-clearance-level* basis. KillPercent values in the range of 0 – 20% are taken to represent system performance under "normal" loads, while KillPercent values in the range of 20 – 100% represent system performance under "heavy" loads [16].[13]

An additional performance metric is **ClassFairness** which captures how evenly the killed transactions are spread across the various clearance levels. This is computed, for each class i, as the ratio $\frac{100-KillPercent(i)}{100-KillPercent}$. With this formulation, a policy is ideally fair if the fairness value is 1.0 for all classes. Fairness values greater than one and lesser than one indicate positive bias and negative bias, respectively.

7.9 Default parameter settings

The default settings used in our experiments for the workload and system parameters are listed in Table 1 and, except for the buffer parameters which are new, are similar to those used in our earlier study [10]. These settings were chosen to ensure significant locality in the data reference patterns and significant buffer contention, thus helping to bring out the performance differences between the various buffer management policies. While the absolute performance profiles of the buffer policies would, of course, change if alternative parameter settings were used, we expect that the *relative* performance of these protocols will remain qualitatively similar since the model parameters are not policy specific.

8 Experiments and results

Using the real-time database model described in the previous section, we evaluated the performance of the various buffer managers for a variety of security-classified transaction workloads and system configurations. The simulator was written using the Simscript II.5 discrete-event simulation language [22]. In this section, we present the performance results from our simulation experiments. Due to space constraints, we present results for only a set of representative experiments here – the others are available in [9]. In our discussion, only statistically significant differences are considered – all the performance graphs show mean values that have relative half-widths about the mean of less than 10% at the 90% confidence level, with each experiment having been run until at least 10,000 transactions were processed by the system.

The simulator was instrumented to generate a host of other statistical information, including resource utilizations, buffer hit ratios, transaction restarts, etc. These secondary measures help to explain the performance behavior of the buffer management policies under various workload and system conditions.

8.1 Overview of experiments

In our first experiment, we profile the performance of the buffer management policies for the default parameter settings, wherein both intra- and inter-transaction locality are present in the workload. In the subsequent set of experiments, we isolate the performance effects independently of inter-transaction locality and intra-transaction locality. While these experiments are reported for a two-security-level system, our final experiment evaluates the performance behavior in the context of a five-security-level system.

8.2 Experiment 1: inter- and intra-transaction locality

In our first experiment, we evaluate the cost of providing covert-channel-free security in the real-time environment for a system with two security levels: Secret and Public, and where there is significant inter- and intra-transaction locality. For this configuration, Fig. 7a shows the overall KillPercent behavior as a function of the (per-second) transaction arrival rate for the SABRE, CONV, RT, ALLHIT and ALLMISS buffer policies. We observe here that, as expected, the ALLHIT and ALLMISS baseline policies exhibit the best and worst performance, respectively. What is more interesting is the *vast gap* between the performance of ALLHIT and that of ALLMISS, indicating the extent to which intelligent buffer management can play a role in improving system performance and highlighting the need for well-designed buffer policies.

Among the practical protocols, we observe that the real-time performance behaviors of SABRE and RT are similar, with RT holding the edge, whereas CONV is significantly worse, especially under heavy loads. RT shows better performance than CONV because it gives preferential treatment to urgent transactions in the following ways. (1) They derive more *intra-transaction* and *restart* locality since their active pages cannot be replaced by low-priority transactions; and (2) they get "first pick" of the available slots due to the prioritized queuing for buffer slots.

[12] A new reader joins a group of lock-holding readers only if its priority is higher than that of *all* the waiting writers.

[13] A long-term operating region where the kill percentage is high is obviously unrealistic for a viable RTDBS. Exercising the system to high kill levels, however, provides valuable information on the response of the algorithms to brief periods of stress loading.

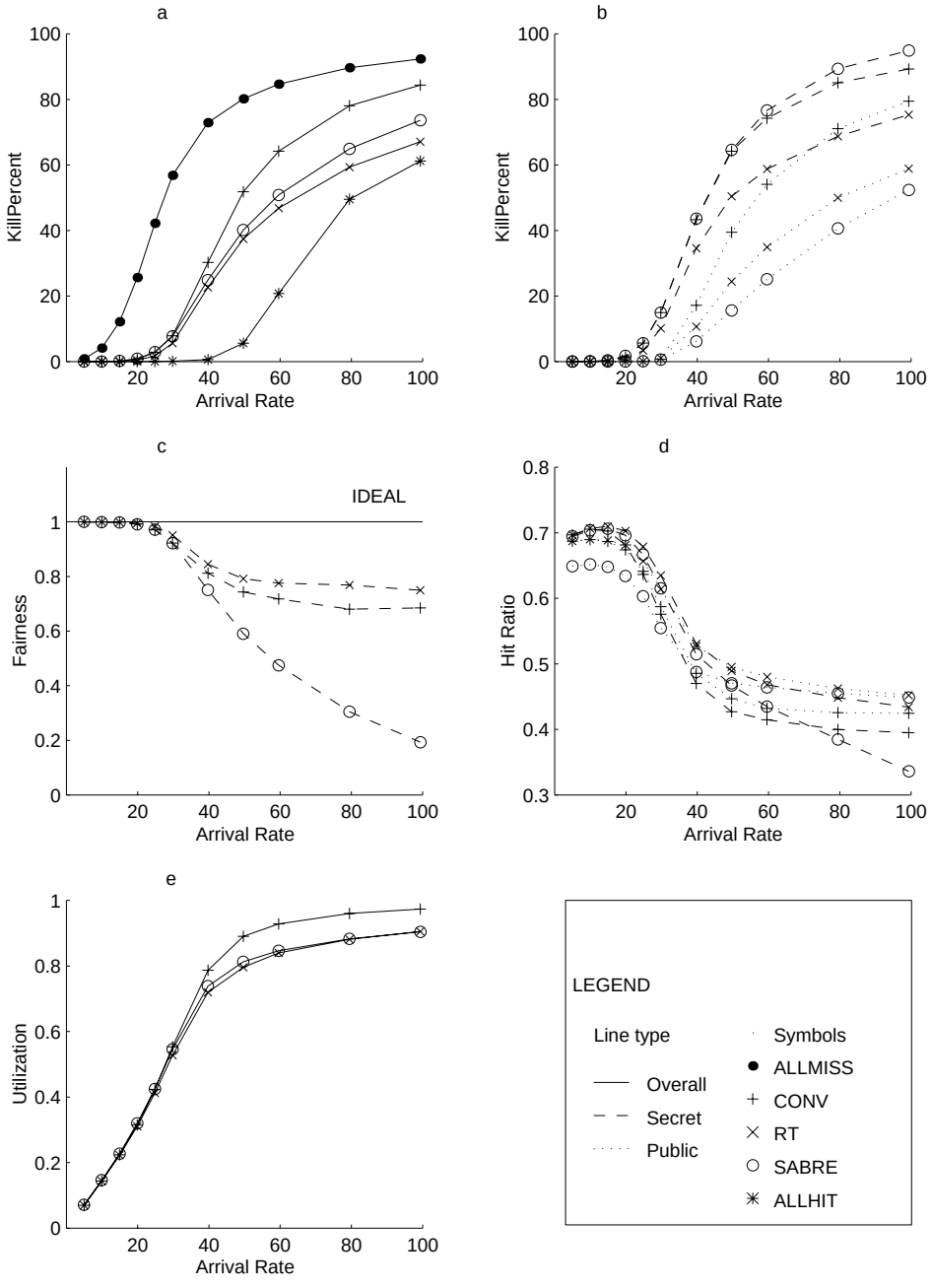

Fig. 7. Intra- and inter-transaction locality (Experiment 1): **a** Overall KillPercent, **b** level KillPercent, **c** fairness (secret), **d** buffer hit ratio, **e** buffer utilization

The reason that SABRE performs slightly worse than RT is that, since priorities are decided primarily based on clearance levels, it is still possible for a slack-deadline public transaction to restart a tight-deadline secret transaction, resulting in the secret transaction missing its deadline. This effect is highlighted in Fig. 7b, which shows the *level-wise* kill percentages – with SABRE, secret transactions (dashed lines) form a much larger proportion of the killed transactions. This is further quantified in Fig. 7c, which captures the ClassFairness metric for the secret transaction class, and clearly demonstrates that SABRE is extremely unfair to secret transactions, especially under heavy loads.

It may appear surprising in Figs. 7b and 7c that RT and CONV, although not security cognizant, are still somewhat unfair to secret transactions. This behavior is not due to the buffer policies themselves, but is a *side effect* of the rest of the system being secure and therefore discriminating against secret transactions.

In Fig. 7d, we show the *buffer hit ratio*, that is the average probability of finding a requested page in the visible portion of the buffer pool, for the various protocols on a level-wise basis. The first point to note is that in all the protocols there is a *crossover* between the secret and public hit ratios – at normal loads secret is better whereas under heavy loads public is better. This is explained as follows: in CONV and in RT, the normal load hit ratios should have been similar since they are not security cognizant, but again there is a side effect arising out of the rest of the system being secure – more secret transactions are restarted for *concurrency control* reasons than public transactions and therefore there is more *restart locality* derived for secret transactions. Under heavy loads, however, this effect is overtaken by the public

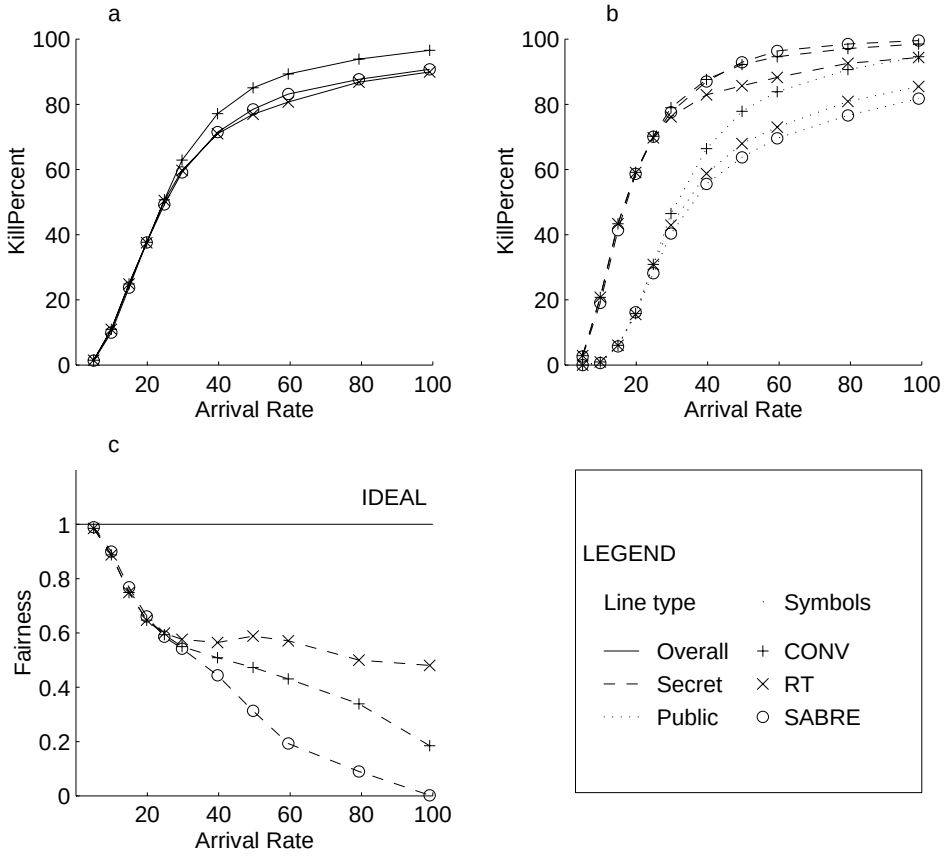

Fig. 8. Inter-transaction locality (Experiment 2): **a** Overall KillPercent, **b** level KillPercent, **c** fairness (secret)

transactions hogging most of the buffer slots and thereby making the secret transactions hot spot mostly absent from the buffer pool.

In SABRE, the secret transactions do better at normal loads, not only because of the restart locality mentioned above, but also due to two additional factors. (1) The whole buffer pool is visible to secret transactions; and (2) the "slot stealing" feature. Under heavy loads, however, *even more* of the buffer slots are occupied by public transactions as compared to CONV and RT, since secret transactions cannot replace the active pages of public transactions, and therefore the secret hit ratio is markedly worse. Another interesting observation with respect to SABRE is that, although its design prevented the use of much of the dormant page locality (as described in Sect. 5), yet its overall hit ratio is not materially worse than that of CONV or RT. This is because, except under very light loads, most of the pages in the buffer pool will be active since they represent the hot spot and therefore *active page locality predominates*.

Finally, Fig. 7e shows the *buffer utilization*, that is, the overall average number of pinned buffer slots, and it is clear here that CONV utilizes the buffers more than RT and SABRE. However, this does not result in better real-time performance because CONV is deadline-indifferent, whereas RT and SABRE selectively give the pinned slots to tight-deadline transactions.

In summary, the results of this experiment highlight the benefits of using deadline-cognizant buffer management policies. Further, it shows that SABRE provides security with only a modest drop in real-time performance. The non-interference requirement, however, causes SABRE to be rather unfair to high-clearance transactions, especially under heavy loads.

8.3 Experiment 2: inter-transaction locality

The goal of our next experiment is to study, in isolation, the impact of *inter*-transaction locality on the system performance. All parameters remain the same as those used in the previous experiment, except that the the $IntraLoc$ parameter is set to 0.0 to eliminate intra-transaction locality. The performance characteristics obtained for this environment are shown in Figs. 8a–c.

In Fig. 8a, the overall kill percentages of all the buffer managers are given. We see here that at normal loads all of them perform almost identically, but at higher loads, CONV performs the worst, whereas RT provides the best performance and SABRE performs almost as well as RT. The miss percentage values in this experiment are significantly *higher*, for all policies, than their respective kill percentage values in Experiment 1. This is due to two reasons. First, the lowered buffer hit ratio. Second, the global skew in data accesses results in hot spots. These hot spots *increase* data conflicts and, consequently, increase the number of deadline misses.

The level-wise split-up of kill percentage values is given in Fig. 8b. The secret kill percentage for RT is less than that of CONV due to its priority-cognizant, but security-indifferent characteristic. Turning our attention to the public miss percentages, CONV shows the worst kill percentage at

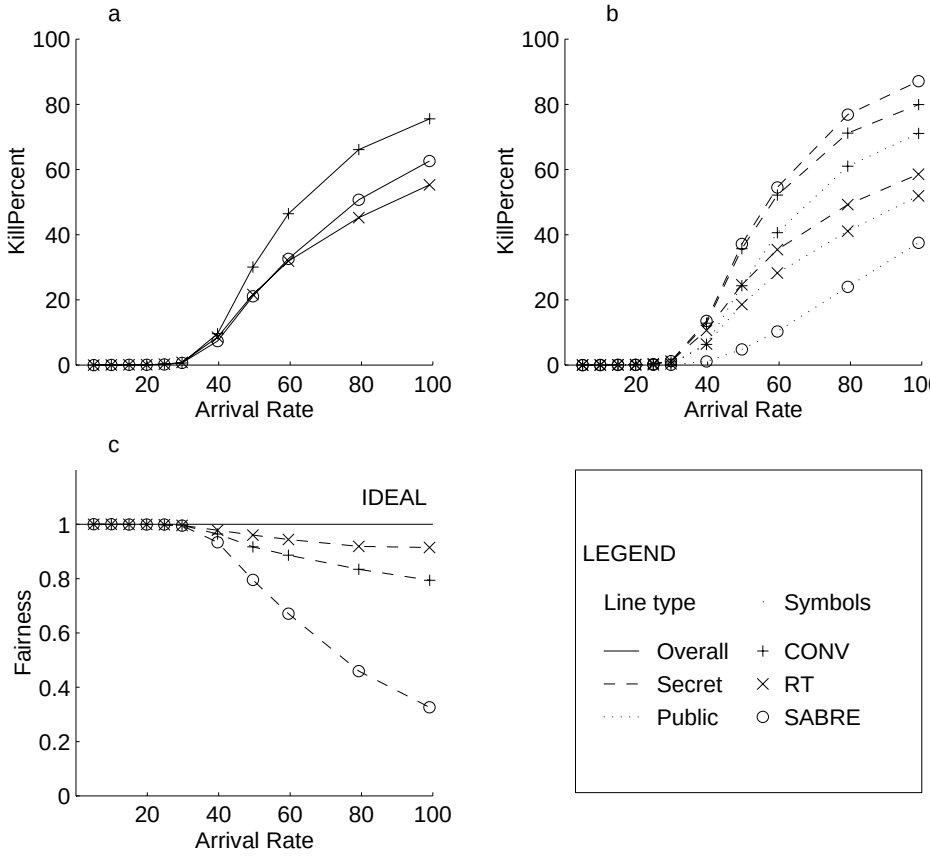

Fig. 9. Intra-transaction locality (Experiment 3): **a** Overall KillPercent, **b** level KillPercent, **c** fairness (secret)

heavy loads, whereas the RT public kill percentage is lower due to its priority-cognizance. Public transactions in SABRE have significantly better performance than their secret counterparts. This is again because the public transactions do not "feel" the secret transactions in the buffer, due to the non-interference feature of SABRE.

The fairness characteristics shown in Fig. 8c indicate that SABRE is highly unfair at heavy loads. This happens because most of the buffer slots will be made either active or pinned by the public transactions, thereby leaving little room for secret transactions in the buffer.

To conclude, the inter-transaction locality in the workload gives rise to increased restarts and thereby reduced real-time performance. Though the real-time features of RT and SABRE make them marginally better than CONV in terms of kill percentage performance, the improvement is relatively minor.

8.4 Experiment 3: intra-transaction locality

This experiment was conducted to study, in isolation, the effect of *intra*-transaction locality on the system performance. All parameter values are the same as those of Experiment 1, except that the $InterLoc$ parameter is set to 0.0 to eliminate inter-transaction locality. The results for this environment are shown in Fig. 9a-c.

We first note that the overall kill percentages (Fig. 9a) are considerably *reduced* for all the policies, compared to their values in Experiment 1. This may seem surprising given the reduced locality modeled in this experiment, but arises because the intra-locality increases the hit ratio *without* attracting the penalty of data conflicts (unlike inter-transaction locality).

Another interesting point to note here is that CONV performs much worse than RT and SABRE, unlike in Experiment 2, where there was only a marginal difference in real-time performance. This is due to the fact that the deadline-cognizance of SABRE and RT is *more effective under intra-transaction locality*. This is because they retain the active/pinned pages of the urgent transactions in the buffer, and therefore these pages are usually available when re-referred. This helps the urgent transactions to complete before their deadlines, thereby resulting in performance improvement. At heavy loads, however, SABRE's performance becomes worse than RT. This is because the security-cognizance of SABRE gains prominence over its real-time cognizance in this region.

Similar to earlier experiments, we see that with SABRE the secret transactions lose heavily to the public transactions at heavy loads as is evident from the level-wise kill percentages shown in Fig. 9b and from the fairness characteristics shown in Fig. 9c. Again, this is due to SABRE's enforcement of covert-channel security constraints.

In summary, with intra-transaction locality, the real-time features of RT and SABRE deliver significant performance benefits. Howeve, SABRE continues to be unfair as in the previous experiments.

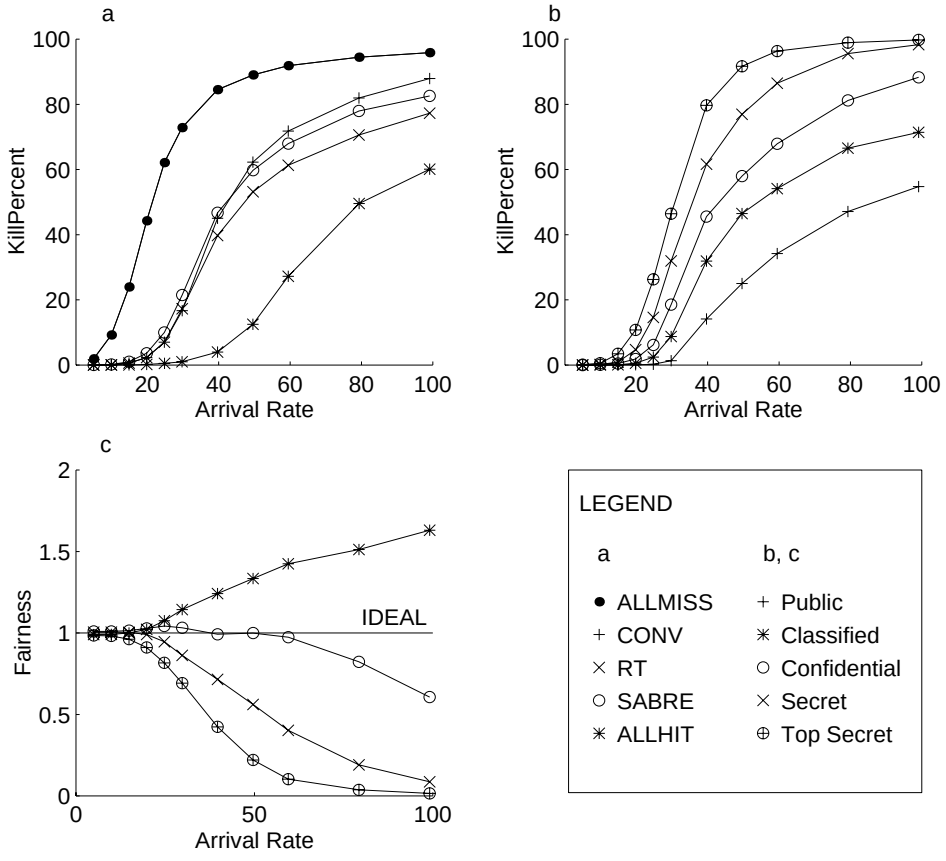

Fig. 10.a–c. Five security levels (Experiment 4): **a** Overall KillPercent, **b** level KillPercent, **c** fairness (SABRE)

8.5 Experiment 4: five security levels

The previous experiments modeled a two-security-level system. In our final experiment of this section, we study the performance behavior of the buffer managers in a system with *five* security levels (TopSecret, Secret, Confidential, Classified and Public) for both data classification and transaction clearance. All the remaining system and workload parameters are the same as those of Experiment 1.

Before presenting the results, it is important to note that in a five-level system, priority is much more *level-based* than *deadline-based*. Therefore, we might expect that the real-time features of SABRE will have correspondingly reduced effect, resulting in significantly degraded real-time performance.

The results of this experiment are shown in Fig. 10a–c. With respect to the two artificial baseline policies, ALLHIT and ALLMISS, we note again that there is a vast gap in their performance. This highlights the fact that there is a good potential for a sophisticated buffer management policy to improve the real-time performance.

Moving on to the SABRE policy, we see that at normal loads, contrary to our concerns mentioned above, it performs *only slightly worse* than both CONV and RT. This indicates that even in a system with many levels, the normal load performance of SABRE can be almost as good as that of a pure real-time policy. It is only in the heavy load region that the overall KillPercent for SABRE becomes somewhat more than that of RT but is at the same time better than CONV.

The level-wise kill percentages for SABRE are shown in Fig. 10b. These graphs clearly show the extent to which the kill percentages are skewed among the various transaction security levels, with topsecret transactions having the most number of missed deadlines and public transactions having the least.

The corresponding fairness factors for the top four security levels (Top Secret, Secret, Confidential and Classified) are shown in Fig. 10c. These figures clearly show that as the loading factor increases, progressively more and more security classes become discriminated against by the lowest security class (Public).

In summary, in a five-level system SABRE performs slightly worse than both CONV and RT at normal loads. Under heavy loads, however, SABRE performs better than CONV and worse than RT. Another observation is that the security requirements make SABRE unfair to the higher security classes, especially under heavy loads.

8.6 Other experiments

Our other experiments, described in [9], explored the sensitivity of the above results to various workload and system parameters, including the number of security levels, the buffer pool size, the locality factors, etc. The relative performance behaviors of the policies in these other experiments remained qualitatively similar to those seen here.

```
Initial condition: Admit[Public] = 1.0, KillPercent[Public] = 0.0,
                                        KillPercent[Secret] = 0.0
LOOP:  if KillPercent[Secret] > 5.0 then
           FairFactor = KillPercent[Public] / KillPercent[Secret]
           if (FairFactor < 0.95) then
                Admit[Public] = Admit[Public] * 0.95
           else if (FairFactor > 1.05) then
                Admit[Public] = min(Admit[Public] * 1.05, 1.0)
       else
           Admit[Public] = 1.0
       sleep T seconds
       measure KillPercent[Public] and KillPercent[Secret]
       goto statement LOOP
```

Fig. 11. The GUARD admission control policy (two-level)

9 Achieving class fairness

In the previous section, we quantitatively showed that the SABRE policy provides good real-time performance without violating security requirements. However, a marked *lack of fairness* was observed with respect to the performance of the high-security classes. We address the fairness issue in this section.

Policies for ensuring fairness can be categorized into *static* and *dynamic* categories. Static mechanisms such as resource reservation can provide fairness while still maintaining full security. However, they may considerably degrade the *real-time* performance due to resource wastage arising out of their inability to readjust themselves to varying workload conditions.

Dynamic mechanisms, on the other hand, have the ability to adapt to workload variations. However, as observed in [18], providing fairness in a workload-adaptive manner without incurring covert channels appears to be *fundamentally impossible*, since the dynamic fairness-inducing mechanism can *itself become the medium of information compromise*. For example, consider the following situation. A conspiring secret process submits a workload such that the secret class performance degrades. The fairness mechanism subsequently tries to improve the secret class performance by allocating more resources for the secret class. A collaborating public transaction could now feel the reduction in the availability of system resources and thereby sense the presence of the secret process in the system. Therefore, this mechanism could be exploited for covert signaling.

In summary, for full-secure applications, unfairness can only be mitigated to an extent by a judicious choice of buffer management policy, but not completely eliminated. Therefore, we move on to considering in the remainder of this section the more tractable problem of whether it is possible to dynamically provide fairness while guaranteeing *Orange security* (i.e., covert-channel bandwidth of less than one bit per second). In particular, we evaluate whether integrating the **GUARD** transaction admission control policy proposed in [11] with the SABRE buffer manager, a combination we will hereafter refer to as **OSABRE**, can achieve this goal. The GUARD policy decides, for a newly arriving transaction, whether the transaction is permitted to enter the RTDBS or is shut out permanently – in the latter case, the transaction is guaranteed to miss its deadline and is killed at deadline expiry.

In the remainder of this section, we first describe the GUARD policy and then profile OSABRE's performance.

9.1 The GUARD admission control policy

For ease of exposition, we will assume for now that there are only two security levels, *Secret* and *Public*. Later, in Sect. 9.3, we present the extension of the GUARD design to an arbitrary number of security levels.

For the two-level environment, the GUARD admission control policy is shown in Fig. 11. The basic idea in the policy is that, based on the imbalance in the transaction kill percentages of the secret and public classes, the admission of public transactions into the system is *periodically* controlled every T seconds (the setting of T is discussed below). The $FairFactor$ variable, which is the ratio of the kill percentages for the public and secret classes, captures the degree of unfairness during the last observation period. Ideally the FairFactor should be 1.0 and so, if there is a significant imbalance ($FairFactor < 0.95$ or $FairFactor > 1.05$), what is done is to decrease or increase the admit probability of the public transactions accordingly.[14] The increase or decrease margin has been set to a nominal 5%. The hope is that this mechanism will eventually result in the multiprogramming level of the public transactions reaching a value that ensures that the secret transactions are not unduly harmed. Finally, to ensure that the admission control becomes operative only when the secret transaction class is experiencing sustained missed deadlines, a threshold kill percentage of 5% is included.

One danger associated with admission control policies is that, although providing fairness, they may cause unnecessary increases in the kill percentages of *both the public and secret transactions* as compared to the corresponding values *without* admission control. Such "pseudo-fairness" is obviously not useful. However, as will be borne out in the experimental results described later in this section, the GUARD policy does not suffer from this shortcoming – it evenly redistributes the "pain" without really increasing its overall magnitude.

9.2 Guaranteeing orange security

In the GUARD policy, covert channels can arise due to the admission control mechanism. For example, a corrupt secret user can, by modulating the outcomes (increase, decrease, or constant) of the Admit[Public] computation, signal information to collaborating public users. Corresponding to

[14] No restrictions are placed on the admission of secret transactions.

```
Initial condition: Admit[1:N-1] = 1.0, KillPercent[1:N] = 0.0
LOOP:  for i = 1 to N-1
         if OverallKillPercent > 5.0 then
              FairFactor =  KillPercent[i] / OverallKillPercent
              if (FairFactor < 0.95) then
                  Admit[i] = Admit[i] * 0.95
              else if (FairFactor > 1.05) then
                  Admit[i] = min(Admit[i] * 1.05, 1.0)
          else
              Admit[i] = 1.0

       sleep T seconds
       measure KillPercent[1:N]
       goto statement LOOP
```

Fig. 12. The general GUARD admission control policy

each computation, at most $log_2 3 = 1.6$ bits of information can be transmitted (based on information-theoretic considerations [26]), and since the computation is made once every T seconds, the maximum channel bandwidth is $1.6/T$ bits per second. By setting this equal to 1 bit per second, the condition for being orange-secure, we get $T_{min} = 1.6$ s to be the minimum re-computation period.

9.3 Extension to N security levels

The GUARD policy described above assumed a two-security-level system. It can be easily extended to the generic N-security-level scenario, as shown in Fig. 12. Note that the channel capacity in the general case is $log_2 3^{N-1} = 1.6(N-1)$ bits and therefore $T_{min} = (N-1) * 1.6$ s.

9.4 GUARD implementation

The GUARD implementation comprises of a *sensory mechanism* and a *regulatory mechanism*. The sensory mechanism operates once every S seconds and calculates the kill percentage of each security level over the last observation period. These observations are continuously made available to the regulatory mechanism, which operates once every T seconds. The regulatory mechanism computes a *geometrically weighted* combination of the sensed values received in the last T seconds[15] and uses this value to determine the desired multi-programming level for each of the security levels. The sensory mechanism operates once every S seconds, while the regulatory mechanism is scheduled once every T seconds. The T value is set such that orange security is guaranteed as per the discussion in Sect. 9.2, while S is set such that a reasonable number of samples are taken within each regulatory period – for example, the heuristic followed in our experiments is to set $S = T/16$.

Transactions belonging to the highest security level are always given admission. For other transactions, however, based on the admit probability currently associated with the security level of the transaction (as determined by the Admit array of Fig. 12), the new transaction is either allowed to enter the system or is shut out permanently (and eventually killed when its deadline expires).

[15] This is similar to the system load computation method used in the Unix operating system [23].

9.5 Class fairness experiments

We conducted experiments to evaluate the performance of the GUARD admission policy and present a representative set of results here. In the following experiments, we compare two systems: (1) **FSABRE**: A full-secure SRTDBS equipped with secure 2PL-HP CC manager and SABRE buffer manager; (2) **OSABRE**: An orange-secure SRTDBS that is equipped with secure 2PL-HP CC manager, SABRE buffer manager, and the GUARD admission controller. Note that the FSABRE system was already evaluated in Sect. 8 – we reproduce its results here for comparison purposes.

9.5.1 Experiment 5: fairness with two-security levels

In our first experiment, we evaluated the performance of FSABRE and OSABRE for a two-security-level system. For this experiment, the GUARD parameters, T and S, were set to $1.6\,s$ and $0.1\,s$, respectively.

The results of this experiment are shown in Figs. 13a–c, which capture the overall kill percent, the level-wise kill percent, and the class fairness, respectively. In these figures, we see that OSABRE achieves *close to ideal fairness*. Further, at normal loads, OSABRE causes only *a small increase of the overall kill percentage* with respect to FSABRE, whereas at heavy loads, OSABRE *actually does slightly better*. At normal loads, due to the inherent lag involved in the feedback process, OSABRE sometimes tends to be over-conservative, preventing public transactions from entering even when strictly not necessary, and thereby increasing the public kill percentage. In contrast, under heavy loads, being conservative does less harm than being too liberal, and therefore its performance shows minor improvement over that of FSABRE.

In summary, the results of this experiment show that OSABRE achieves its fairness very efficiently and that the admission control component functions as intended in its design.

9.5.2 Experiment 6: fairness with five security levels

In our next experiment, we compare the FSABRE and OSABRE systems for a *five*-security-level environment similar to that evaluated in Experiment 4. The results of this

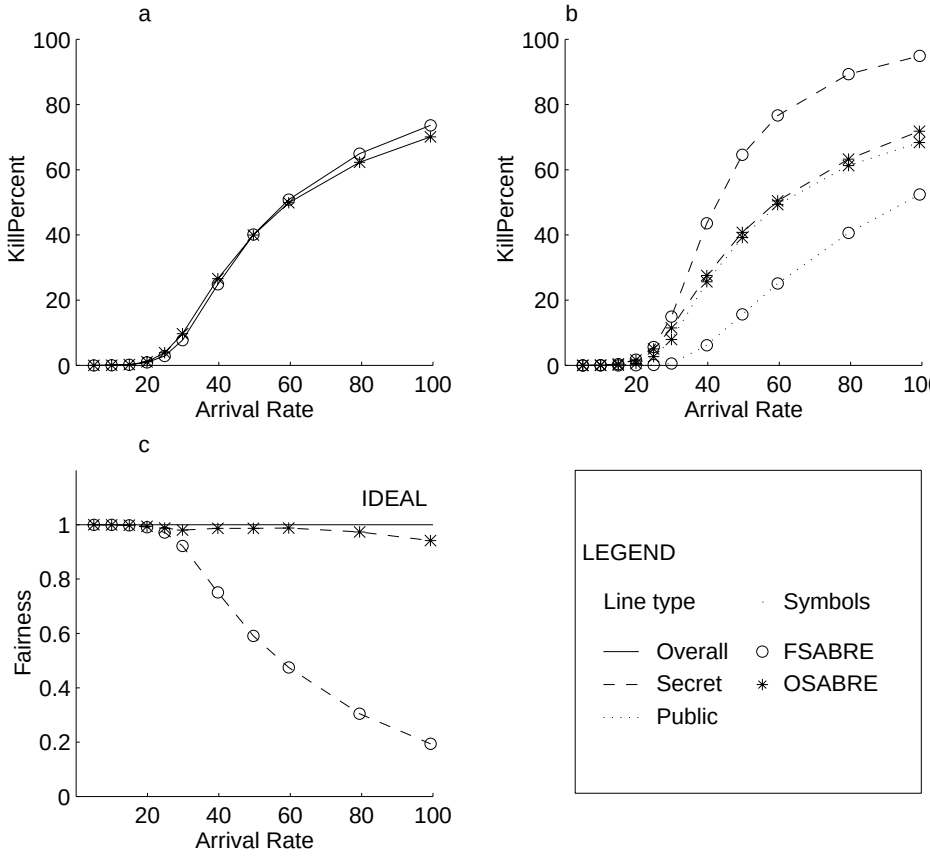

Fig. 13.a–c. KillPercent fairness of OSABRE (Experiment 5): **a** overall KillPercent, **b** level KillPercent, **c** fairness (Secret)

experiment are shown in Fig. 14a-c. Note that for this experiment, the GUARD parameters, T and S, are set to $6.4\,s$ and $0.4\,s$, respectively (as per the discussion in Sect. 9.3).

In Fig. 14a, which shows the overall KillPercent characteristics, we see that the results are similar to those seen in the previous two-security-level experiment: OSABRE performs marginally worse than FSABRE at normal loads, whereas under heavy loads, it performs slightly better. In Fig. 14b and 14c, the fairness characteristics of the FSABRE and OSABRE systems are shown. As is evident from Fig. 14c, OSABRE exhibits reasonably close to ideal fairness characteristics throughout the loading range in this many-security-level environment as well.

10 Conclusions

In this paper, we have quantitatively investigated, for the first time, the performance implications of the choice of buffer manager in both full-secure and orange-secure firm-deadline RTDBS. This is a followup to our earlier work on secure real-time concurrency control [10]. Making real-time buffer managers secure is complicated due to the multiplicity of covert-channel mediums and buffer components, and due to the inherent difficulties of simultaneously achieving the goals of full security, minimal KillPercent and complete ClassFairness. Our new SABRE policy addresses these challenges by (1) making buffer pool visibility a function of the clearance level, (2) not pre-allocating buffer slots to security levels, (3) sacrificing dormant page locality to permit unrestricted slot stealing by higher clearance transactions, (4) supporting pin-pre-emption, (5) implementing a novel Comb slot selection policy, and (6) incorporating optimizations such as proxy disk service. The SABRE policy was proved to guarantee all requirements of an interference-free scheduler, namely, delay, value and recovery security.

Using a detailed simulation model of a firm-deadline RTDBS, the real-time performance of SABRE was evaluated against the CONV and RT unsecure conventional and real-time buffer managers. Our experiments showed that (a) it is essential to include deadline-cognizance in the buffer manager for good real-time performance, thereby supporting the conclusions of [19], (b) SABRE efficiently provides security in that its real-time performance is not significantly worse than that of RT, especially for applications that have only a small number of security levels, and (c) SABRE's sacrifice of dormant page locality has little impact since active page locality is predominant.

SABRE's main drawback of bias against higher clearance transactions was addressed by the OSABRE system, which incorporated the GUARD transaction admission control mechanism. The bandwidth of the covert channel introduced by this mechanism was bounded by appropriately setting the feedback period. An orange-secure version (bandwidth less than 1 bit per second) of OSABRE was found to provide close to ideal fairness at little cost to the overall real-time performance.

In closing, we suggest that SRTDBS designers may find the **SABRE** buffer manager, either alone or in conjunction with the GUARD admission policy, to be an attractive choice

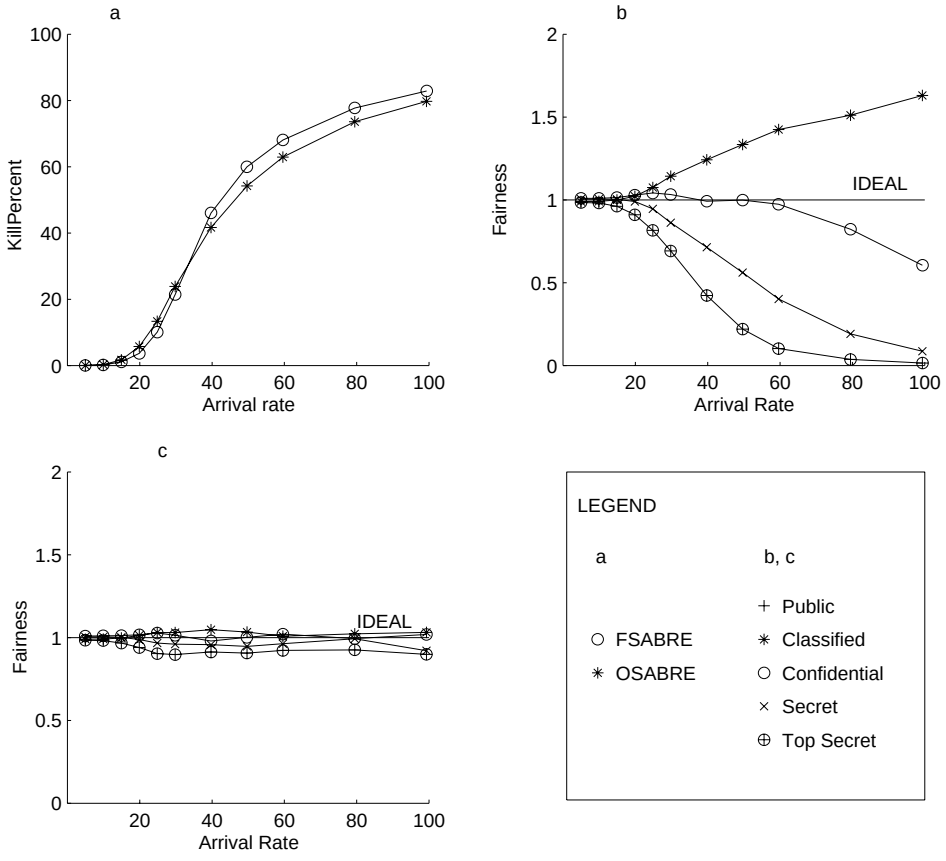

Fig. 14. Fairness with five security levels (Experiment 6): **a** Overall KillPercent, **b** fairness – FSABRE, **c** fairness – OSABRE

for achieving high-performance in secure real-time transaction processing.

10.1 Future work

In this paper, we have taken the first step towards addressing the secure real-time buffer management issue. A limitation of our work is that we have only considered environments wherein the security levels are *fully ordered* – in general, however, security hierarchies may be *partially ordered*, typically in the form of lattices. It is therefore imperative to extend the SABRE buffer manager to this more general framework. Further, apart from multilevel secrecy, additional security requirements such as preventing denial-of-service attacks [4] should also be supported.

Acknowledgements. This work was supported in part by research grants from the Dept. of Science and Technology and the Dept. of Bio-technology, Govt. of India.

References

1. Abbott R, Garcia-Molina H (1992) Scheduling Real-time Transactions: A Performance Evaluation. ACM Trans Database Syst 17(3): 513- 560
2. Abrams M, Jajodia S, Podell H (1995) Information Security. IEEE Computer Society Press, Piscataway, N.J.
3. Carey MJ, Jauhari R, Livny M (1989) Priority in DBMS Resource Scheduling. In: Yuan L (ed) Proc. of 15th VLDB Conf, August 1989, Amsterdam, The Netherlands, pp 397-410
4. Castano S, Fugini M, Martella G, Samarati P (1995) Database Security. Addison-Wesley, Reading, Mass.
5. Chou H, DeWitt D (1985) An Evaluation of Buffer Management Strategies for Relational Database Systems. In: Pirotte A, Vassiliou Y (eds) Proc. of 11th VLDB Conf, August 1985, Stockholm, Sweden, pp 127-141
6. Department of Defense Computer Security Center (1985) Department of Defense Trusted Computer Systems Evaluation Criteria. 5200.28-STD. Department of Defense, Washington, D.C., USA
7. Effelsberg W, Harder T (1984) Principles of Database Buffer Management. ACM Trans Database Syst 9(4): 560-595
8. Eswaran K, et al. (1976) The Notions of Consistency and Predicate Locks in a Database System. Commun ACM 19(11): 624-633
9. George B (1998) Secure Real-Time Transaction Processing. Ph.D. Thesis. Indian Institute of Science, Bangalore, India
10. George B, Haritsa J (1997) Secure Transaction Processing in Real-Time Database Systems. In: Peckham J (ed) Proc. of ACM SIGMOD Conf, May 1997, Tucson, USA, pp 462-473
11. George B, Haritsa J (2000) Secure Concurrency Control in Firm Real-Time Database Systems. J Distrib Parallel Databases (Special Issue on Security) 8(1): in press
12. Goguen J, Meseguer J (1982) Security Policy and Security Models. In: Proc. of IEEE Symposium on Security and Privacy, 1982, Berkeley, USA, pp 11-20
13. Gray J, Reuter A (1993) Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo, Calif.
14. Greenberg I, et al. (1993) The Secure Alpha Study (Final Summary Report). Tech. Report ELIN A012. SRI International, Menlo Park, Calif., USA
15. Harder T (1984) Observations on Optimistic Concurrency Control Schemes. Inf Syst 9(2): 111-120
16. Haritsa J, Carey M, Livny M (1992) Data Access Scheduling in Firm Real-Time Database Systems. Journal 4(4): 203-241
17. Huang J (1991) Real-Time Transaction Processing: Design, implementation, and performance evaluation. Ph.D. Thesis. University of Massachusetts, Amherst, Mass.

18. Keefe T, Tsai W, Srivastava J (1990) Multilevel Secure Database Concurrency Control. In: Proc. of IEEE Data Engineering Conf, February 1990, Los Angeles, USA, pp 337-344
19. Kim W, Srivastava J (1991) Buffer Management and Disk Scheduling for Real-Time Relational Database Systems. In: Sadanandan P, Vijayraman T (eds) Proc. of 3rd COMAD Conf, December 1991, Bombay, India
20. LaPadula L, Bell D (1976) Secure computer systems: Unified Exposition and Multics Interpretation. The Mitre Corp., Bedford, USA
21. Lampson W (1973) A Note on the Confinement Problem. Commun ACM 16(10): 613-615
22. Law AM, Larney CS (1984) Introduction to Simulation Using SIMSCRIPT 11.5. CACI Products Company, La Jolla, Calif.
23. Leffler S, McKusick M, Karels M, Quarterman J (1989) The Design and Implementation of 4.3 BSD UNIX Operating System. Addison-Wesley, Reading, Mass.
24. Liu C, Layland J (1973) Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. J ACM 20(1): 46-61
25. Sha L, Rajkumar R, Lehoczky J (1987) Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Tech. Rep. CMU-CS-87-181. Carnegie Mellon University, New York, N.Y.
26. Shannon C (1948) A Mathematical Theory of Communications. Bell Syst Tech J 27(4): 623-656
27. Ulusoy O (1994) Research Issues in Real-Time Database Systems. Tech. Report BU-CEIS-94-32. Dept. of Computer Engineering and Information Science, Bilkent University, Turkey
28. Warner A, Li Q, Keefe T, Pat S (1996) The Impact of Multilevel Security on Database Buffer Management. In: Proc. of European Symposium on Research in Computer Security, 1996, Rome, Italy, Springer Verlag LNCS 1146 pp 266-289
29. Wedekind H, Zoerntlein G (1986) Prefetching in Real-Time Database Applications. In: Zaniolo C (ed) Proc. of ACM SIGMOD Conf, June 1986, Washington D.C., USA, pp 215-226