Distributed and Parallel Databases, 5, 275–320 (1997)
© 1997 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Secure Concurrency Control in Firm Real-Time Database Systems*

BINTO GEORGE JAYANT R. HARITSA {binto, haritsa}@dsl.serc.iisc.ernet.in
Database Systems Lab, Indian Institute of Science, Bangalore- 560012, India

Received Change this; Revised Change this

Editor: Vijay Atluri

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. For real-time database systems supporting applications with firm deadlines, we investigate here the performance implications, in terms of killed transactions, of guaranteeing multilevel secrecy. In particular, we focus on the concurrency control (CC) aspects of this issue.

Our main contributions are the following: First, we identify which among the previously proposed real-time CC protocols are capable of providing covert-channel-free security. Second, using a detailed simulation model, we profile the real-time performance of a representative set of these secure CC protocols for a variety of security-classified workloads and system configurations. Our experiments show that a prioritized optimistic CC protocol, OPT-WAIT, provides the best overall performance. Third, we propose and evaluate a novel "dual-CC" approach that allows the real-time database system to simultaneously use different CC mechanisms for guaranteeing security and for improving real-time performance. By appropriately choosing these different mechanisms, concurrency control protocols that provide even better performance than OPT-WAIT are designed. Finally, we propose and evaluate GUARD, an adaptive admission-control policy designed to provide fairness with respect to the distribution of killed transactions across security levels. Our experiments show that GUARD efficiently provides close to ideal fairness for real-time applications that can tolerate covert channel bandwidths of upto one bit per second.

Keywords: Real-Time Database, Covert Channels, Concurrency Control, Firm Deadlines, Fairness, Performance Evaluation

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

* A partial and preliminary version of the results presented here appeared earlier in *Secure Transaction Processing in Firm Real-Time Database Systems*, Proc. of ACM SIGMOD Intl. Conf. on Management of Data, Tucson, Arizona, USA, May 1997.

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-specified time period after submission (in the Flash Auction at *http://www.firstauction.com*, bids that arrive more than *five minutes* 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.

In this paper, we focus 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)**. Our study is conducted in the context of real-time applications with "firm-deadlines" [19]. 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.

1.1. Security Mechanisms

For a DBMS to be considered secure, a number of requirements have been identified in the literature [7]. Among these, **secrecy**, that is, the prevention of unauthorized

knowledge of secret data, is an especially important requirement for 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.

Security violations in an 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 [24] 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 [23]. 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. Note that, from the system perspective, nothing "obviously illegal" has been done in this process by the conspiring transactions.

Covert channels that use the DBMS's *physical resources* such as the CPU or the disk as the medium for passing on information can be tackled by introducing "noise" in the form of dummy transactions that make use of these resources (similar to the "pump" scheme proposed in [21]). However, this approach is impractical for covert channels that use *data* as the medium (for example, presence or absence of a lock on a pre-determined data item). This is because, unlike physical resources which are typically few in number, the number of data items is usually enormous, especially in a DBMS. In fact, in heavily loaded systems, noise at the physical resources may be generated "for free", but this will probably never be the case for data since it is trivial to insert an additional data item that is of relevance only to the conspiring transactions. Therefore, *explicitly making data access covert-channel-free is more vital than doing the same for resource access.*

Covert channels based on data can be prevented by ensuring that low security transactions do not "see" high security transactions – this notion is formalized in [15] 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 a **concurrency control (CC)** mechanism that supports the non-interference feature. In this paper, we investigate the design and performance implications of secure CC in the context of a firm-deadline RTDBS.

1.2. Design Challenges

A variety of challenging problems arise when we attempt to integrate security into the RTDBS framework, as described below:

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 [16]. 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 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. Although a variety of real-time CC protocols (e.g. [1, 4, 19, 27]) have been proposed during the last decade, not all of them may be amenable to supporting the non-interference requirement.

3. Apart from the noise approach mentioned earlier, covert channels can also be eliminated by *static* resource allocation policies, wherein each transaction security class has a set of pre-assigned resources. This approach is taken, for example, in *replicated secure architectures* wherein a multi-level secure DBMS is built by "stacking" single-level database systems. As pointed out in [6], the drawback of such strategies, however, is that they may result in an inefficient and inflexible use of resources.

   Yet another possibility is to use "snapshot" protocols (e.g. [5]), wherein two versions of data items are provided – for data of a given security level, higher security transactions access the older version whereas transactions of the same level access the current version. A problem with this approach, however, is that higher security transactions, especially long-lived ones, have to base their results on *arbitrarily stale* data [6]. In fact, a performance study in [10] shows instances where the highest security level transactions read data items that have been subsequently overwritten by *fifteen* other transactions.

   The above approaches are *especially* problematic in the real-time context since they may result in: (a) A large increase in the number of killed transactions due to poor resource utilization; or (b) Incorrect results since real-time applications typically respond to current situations and therefore should not utilize stale data. Due to these reasons, the challenge in the RTDBS domain is to design "dynamic" and "one-copy" policies that are demonstrably secure.

4. A major problem arising out of the preferential treatment of low security transactions is that of "fairness" – a disproportionately large fraction of the high security transactions miss their deadlines and are killed.

   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* [22].

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.

In summary, for all of the above reasons, making a real-time CC manager implementation to be secure involves significant design complexity.

1.3. Contributions

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

1. We identify which among the previously proposed real-time CC protocols are capable of providing protection against covert channels, that is, which protocols can support the concept of non-interference.

2. Using a detailed simulation model of a firm-deadline RTDBS, we profile the real-time performance of a representative set of secure CC protocols. Our simulations consider a variety of security-classified transaction workloads and system configurations. To isolate and quantify the performance effects of supporting covert channel security, we also evaluate the performance of the CC protocols in the context of a baseline system that prevents *direct* unauthorized access, but not covert channels (that is, it only supports the Bell–LaPadula restrictions). Our experimental results indicate that a prioritized optimistic CC protocol, **OPT-WAIT** [19], provides the best overall performance.

3. We evaluate the effectiveness of a novel *dual-CC* approach to secure transaction concurrency control wherein different CC mechanisms are *simultaneously* used for guaranteeing security and for improving real-time performance, respectively. In particular, we investigate the performance of various combinations of CC mechanisms for resolving inter-security-level and intra-security-level data conflicts. Our results show that one of these combinations, **S2PL-WAIT**, which is composed of the S2PL [33] and OPT-WAIT algorithms, performs even better than OPT-WAIT.

4. We present **GUARD**, an adaptive admission-control policy designed to achieve fairness in terms of 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" [8], which defines security standards. Experimental evaluation of GUARD shows it to provide close to ideal fairness with very little deterioration in the overall real-time performance.

To the best of our knowledge, the above results represent the *first* detailed study of real-time database security in the firm-deadline context.

1.4. Organization

The remainder of this paper is organized as follows: The security model employed in our study is described in Section 2. Issues related to incorporating security in real-time CC protocols are discussed in Section 3 and a representative set of secure CC protocols are described in Section 4. Our new dual approach to secure real-time CC is presented in Section 5. The performance model is described in Section 6, and the results of the simulation experiments are highlighted in Section 7. Fairness issues are addressed in Section 8. Related work on real-time database security is reviewed in Section 9. Finally, in Section 10, we present the conclusions of our study.

2. Security Model

As mentioned in the Introduction, our focus in this paper is on ensuring that secret information is not made available to unauthorized users. Most secure DBMS attempt to achieve this goal by incorporating access control mechanisms based on the well-known Bell–LaPadula model [24]. 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 Figure 1), and thereby prevent *direct* unauthorized access to secure data. They are not sufficient, however, to protect from covert channels [23], as described in the Introduction.

There are two flavors of covert channels: *storage channels* and *timing channels* [2]. Storage channels include all mediums that allow the (direct or indirect) writing of a storage location by a high security transaction and the subsequent access of this storage location by a low security transaction – the failure or success of this access, or the value obtained from the access, can be modulated to signal secret information. Timing channels, on the other hand, include all mediums that allow a high security clearance transaction to signal information to a low security clearance transaction by modulating its own use of system resources in such a way that the change in response time observed by the low security clearance transaction provides information. In this paper, we focus our attention exclusively on timing channels, since storage channels can be handled by incorporating additional

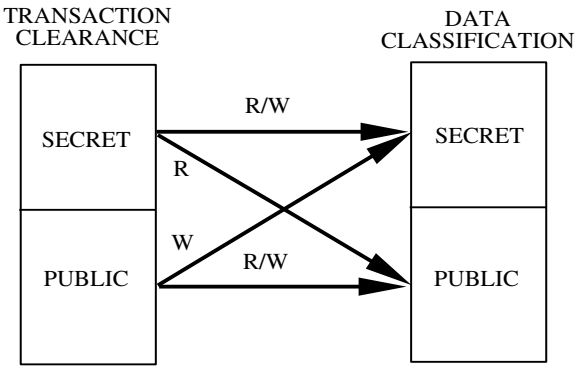

Figure 1. **Bell-LaPadula access restrictions**

access restrictions [14]. For tackling (timing) covert channels, we use the *non-interference* formalism described in [15] wherein low security transactions do not "see" high security transactions.

Finally, to facilitate understanding of the performance results, we only consider environments wherein the security levels are *fully-ordered* – in general, however, security hierarchies may be partially ordered, typically in the form of lattices.

2.1. 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 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" [8], 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 consider the design of CC protocols initially for full-secure real-time applications, and then in Section 8, we extend our scope to orange-secure applications.

3. Integrating Security and Concurrency Control

In this section, we discuss the issues involved in integrating security and concurrency control mechanisms in an RTDBS.

3.1. 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 P = (LEVEL, 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 zero upwards, with zero 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 at a lower clearance have higher priority than all transactions at a higher clearance, 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 assignment [28], wherein transactions with earlier deadlines have higher priority than transactions with later deadlines. For this choice, the priority vector would be P = (LEVEL, DEADLINE) – this priority assignment is used in most of our experimental evaluations.

3.2. Supporting Non-Interference

In conjunction with the above priority assignment policy, it would seem at first glance that, in principle, any real-time CC protocol could be used in an SRTDBS and that the actual choice of protocol would be based only on the relative performance of these protocols. However, *not all the previously proposed real-time CC algorithms are amenable to supporting security requirements*. For example, consider the 2PL Wait Promote algorithm proposed in [1]: This protocol, which is based on 2PL, incorporates a *priority inheritance* mechanism [31] wherein, whenever a requester blocks behind a lower-priority lock holder, the lock holder's priority is promoted to that of the requester. In other words, the lock holder *inherits* the priority of the lock requester. The basic idea here is to reduce the blocking time of high priority transactions by increasing the priority of conflicting low priority lock holders (these low priority transactions now execute faster and therefore release their locks earlier).

The Wait Promote approach is *not* suitable for SRTDBS. This is because it permits the blocking of high priority transactions by low priority transactions which violates the requirement of *non-interference* between the transactions of different security levels.

To generalize the above observation, a real-time CC protocol that permits, to even a limited extent, high priority transactions to be adversely affected by low priority transactions, a phenomenon known as *priority inversion* in the real-time literature [31], cannot be used in a secure RTDBS. Apart from Wait Promote, other examples of real-time CC algorithms that fall into this category include 2PL-CR [1], 2PL-OS/BI [4] and WAIT-50 [19].

4. Secure CC Protocols

In this section, we describe the CC protocols that are evaluated in our study. These include two well-known real-time protocols: 2PL-HP [1], which is based on locking, and OPT-WAIT [19], which is based on optimistic concurrency control. Both these protocols are completely free from priority inversion and can therefore be used to resolve conflicts in an SRTDBS.

Apart from the above real-time algorithms, we also consider S2PL [33], a recently proposed secure locking-based protocol that does not include any real-time-specific features. We include it here, however, for the following reasons: First, it serves as a baseline against which to compare the real-time CC algorithms. Second, we use it in one of the "dual-CC" protocols evaluated later in this study. Finally, it has been used in some of the previous research work on SRTDBS (these are discussed in Section 9).

We have also evaluated OPT-SACRIFICE [19], another real-time optimistic CC protocol, in our performance study. However, since it performed worse than OPT-WAIT in all of our experiments, we do not include it here – the results are available in [14]. We have also not included SOPT, a secure optimistic protocol proposed in [10, 34], since its basic structure is similar to OPT-SACRIFICE and, in addition, it lacks real-time characteristics as all priorities are based purely on transaction security levels.

In the remainder of this section, we describe the 2PL-HP, OPT-WAIT and S2PL protocols.

4.1. 2PL High Priority

The 2PL High Priority (2PL-HP) scheme [1] modifies the classical strict two-phase locking protocol (2PL) [12] by incorporating a priority conflict resolution scheme which ensures that high priority transactions are not delayed by low priority transactions. In 2PL-HP, 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.[2] Note that 2PL-HP is inherently deadlock-free if priorities are assigned uniquely (as is usually the case in RTDBS).

4.2. OPT-WAIT

The OPT-WAIT algorithm [19] modifies the classical forward (or broadcast) optimistic CC protocol (OPT) [30] by incorporating a *priority wait* mechanism. Here, a transaction that reaches validation and finds higher priority transactions in its conflict set is "put on the shelf", that is, it is made to wait and not allowed to commit immediately. This gives the higher priority transactions a chance to make their deadlines first. While a transaction is waiting on the shelf, it is possible that it may be restarted due to the commit of one of the conflicting higher priority transactions. If at any time during its shelf period, the waiting transaction finds no higher priority transactions remaining in its conflict set, it is committed, restarting in the process the lower priority transactions (if any) in its conflict set.

4.3. S2PL

A secure locking-based protocol called Secure 2PL (S2PL) was recently proposed in [33]. The basic principle behind Secure 2PL is to try to simulate the execution of conventional 2PL *without* blocking the actions of low security transactions by high security clearance transactions. This is accomplished by providing a new lock type called *virtual lock*, which is used by low security transactions that develop conflicts with high security transactions. The actions corresponding to setting of virtual locks are implemented on *private versions* of the data item (similar to optimistic CC). When the conflicting high security transaction commits and releases the data item, the virtual lock of the low security transaction is upgraded to a real lock and the operation is performed on the original data item. To complete this scheme, an additional lock type called *dependent virtual lock* is required apart from maintaining, for each executing transaction T_i, lists of the active transactions that precede or follow T_i in the serialization order. The complete details are given in [33].

In our implementation of S2PL, we have had to make some modifications since the algorithm (as described in [33]) does not eliminate interference under all circumstances – the details of the security loopholes and our fixes for these loopholes are given in Appendix B.

5. The Dual-CC Approach to Secure Concurrency Control

In this section, we move on to discussing our new *dual-CC* approach to secure real-time concurrency control. Our design is based on the observation that in the secure environment there are *two* categories of conflicts: *inter-level* and *intra-level*. Inter-level conflicts are data conflicts between transactions belonging to different security clearance levels whereas intra-level conflicts are data conflicts between transactions of the same level. The important point to note here is that *only inter-level conflicts can result in security violations*, not intra-level conflicts. This opens up the possibility of using *different* CC strategies to resolve the different types of conflicts. In particular, we can think of constructing mechanisms such that inter-level conflicts

are resolved in a secure manner while intra-level conflicts are resolved in a timely manner. For example, S2PL could be used for inter-level conflicts while OPT-WAIT could be used to resolve intra-level conflicts.[3]

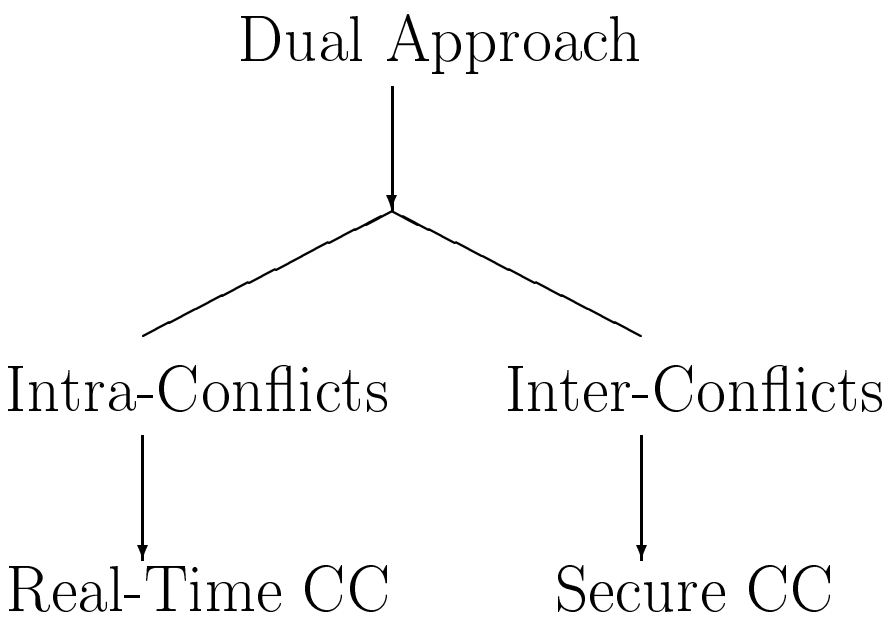

Figure 2. **Dual approach**

The advantage of the dual-CC approach, pictorially shown in Figure 2, is that the RTDBS can maximize the real-time performance, by appropriate choice of intra-level CC protocol, without sacrificing security. This is in marked contrast to the *tradeoff* approach suggested in [10] which requires the application to *compromise* on security in order to achieve enhanced real-time performance (a more detailed assessment of the tradeoff approach is presented in Section 9). Another advantage of the dual-CC approach is that the separation of security and timeliness concerns makes it possible to use even *unsecure* real-time CC algorithms (e.g., Wait-Promote, WAIT-50) for resolving intra-level conflicts! *The dual-CC approach therefore empowers the use, even in the secure RTDBS domain, of the rich set of real-time CC algorithms developed during the last decade.*

5.1. Ensuring Serializability

At first glance, it may appear that concurrently using multiple CC mechanisms could result in violation of the transaction serializability requirement. This could

happen, for example, if the serial orders enforced by the individual mechanisms were to be different. A detailed study of a generalized version of this problem is presented in [37], wherein the transaction workload consists of a mix of transaction classes and the objective is to allow each transaction class to utilize its preferred CC mechanism. They propose a database system architecture wherein intra-class conflicts are handled by the class's preferred CC manager while inter-class conflicts are handled by a new software module called the Master Concurrency Controller (MCC) that interfaces between the transaction manager and the multiple CC managers. The MCC itself implements a complete concurrency control mechanism and ensures a single global serialization order in the entire database system by using a *Global Ordering Scheme* [37].

For our study, we adapt the above architecture to the secure real-time environment. In our implementation, each security level has a Local Concurrency Controller (LCC) that resolves intra-level data conflicts. The LCCs can use *any* high performance real-time CC protocol without risking security violations. The Master Concurrency Control mechanism, however, is chosen to be a secure protocol and is responsible for handling all inter-level data conflicts. Figure 3 shows an example of this implementation for a two security level system. The proof of correctness and other details of our adaptation of the MCC architecture are given in Appendix A.

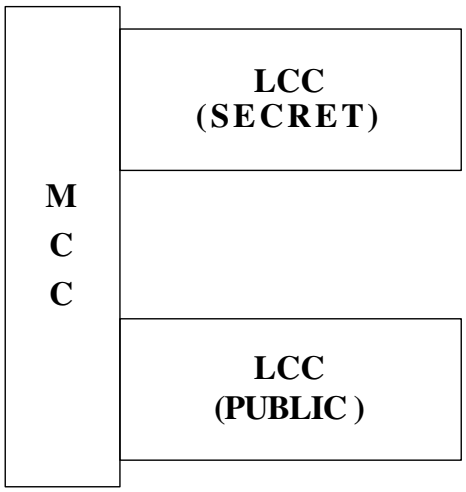

Figure 3. **MCC architecture**

5.2. Dual-CC Protocols

We have developed and evaluated three protocols, based on the dual-CC approach: WAIT-HP, HP-WAIT and S2PL-WAIT. These protocols are described below:

Table 1. Simulation Model Parameters

Parameter	Meaning	Default Value
$DBSize$	Number of pages in the database	1000
$ClassLevels$	Number of Classification Levels	2 (Secret, Public)
$NumCPUs$	Number of processors	10
$NumDisks$	Number of disks	20
$PageCPU$	CPU time for processing a data page	10ms
$PageDisk$	Disk service time for a data page	20ms
$CCReqCPU$	Concurrency Control Overhead	1ms
$ArrivalRate$	Transaction arrival rate (per second)	0 – 100
$ClearLevels$	Number of Clearance Levels	2 (Secret, Public)
$SlackFactor$	Slack Factor in Deadline Assignment	4.0
$TransSize$	Average transaction size (in pages)	16
$WriteProb$	Page write probability	0.5

WAIT-HP: Inter-level data conflicts are resolved by OPT-WAIT and intra-level conflicts are resolved by 2PL-HP.

HP-WAIT: This protocol is the "mirror" of WAIT-HP – inter-level data conflicts are resolved by 2PL-HP while intra-level conflicts are resolved by OPT-WAIT.

S2PL-WAIT: Inter-level data conflicts are resolved by S2PL and intra-level conflicts are resolved by OPT-WAIT. The scheme operates as follows: After the intra-level conflicts are resolved by OPT-WAIT, the transaction enters the "virtual commit" state, and subsequently the inter-level data conflicts are resolved by S2PL. After all conflicts are resolved, a transaction "really" commits. Recall that S2PL is *not* a restart-oriented algorithm and allows a high security transaction to continue regardless of the virtual commit of a low security conflicting transaction.

6. Simulation Model

In the previous sections, we presented a variety of secure CC protocols. To evaluate the real-time performance of these algorithms, we developed a detailed simulation model of a firm-deadline RTDBS, similar to that described in [19]. A summary of the key model parameters is given in Table 1.

The model, shown in Figure 4, has six components: a *Database* that models the data and its layout; a *Source* that generates the transaction workload; a *Transaction Manager* that models the execution of transactions; a *Resource Manager* that models the hardware resources; a *Concurrency Control (CC) Manager* that controls access to shared data; and a *Sink* that gathers statistics on transactions exiting the system. A separate instance of the CC Manager is created for each of the CC protocols evaluated in our experiments.

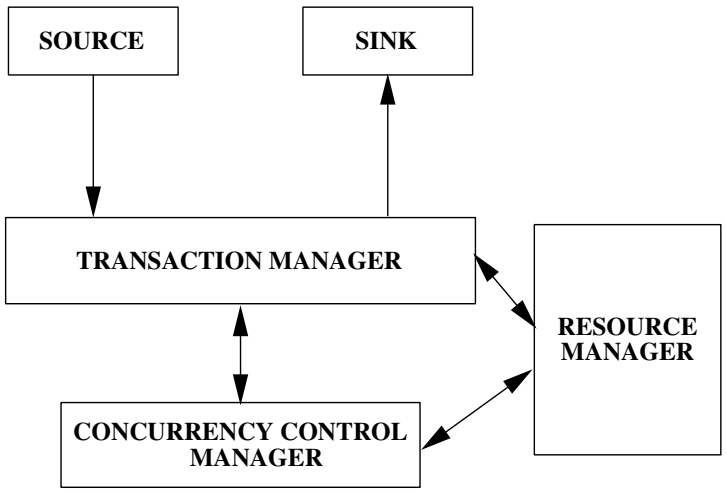

Figure 4. **Simulation Model**

6.1. 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).

6.2. System Model

The system consists of a shared-memory multiprocessor DBMS operating on disk-resident data (for simplicity, we assume that all data is accessed from disk and buffer pool considerations are therefore ignored). The physical resources of the database system consist of $NumCPUs$ processors and $NumDisks$ disks. There is a single common queue for the CPUs and the service discipline is Pre-emptive Resume, with preemptions 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. For concurrency control requests, the associated CPU processing overhead is specified by the $CCReqCPU$ parameter.

6.3. 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 dead-

line. 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).

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 sequence of data accesses as its original incarnation.

6.4. Transaction Execution

A transaction read access involves a CC request to get access permission, followed by a disk I/O to read the page, followed by a period of CPU usage for processing the page. Write requests are handled similarly except for their disk I/O – their disk activity is deferred until the transaction has committed. We assume that the RTDBS has sufficient buffer space to allow the retention of updates until commit time.

6.5. Priority Assignment

In general, the transaction priority assignment used at all the RTDBS components is P = (LEVEL, DEADLINE), based on the discussion in Section 3.1.[4] The only exception is for the DIRECT environment, described below.

6.5.1. The DIRECT Environment To help isolate and understand the performance cost that occurs due to having to eliminate covert channels, we have also simulated the performance achievable in the *absence* of covert channel security requirements. That is, the performance achievable if only Bell–LaPadula conditions had to be satisfied. For this scenario, a priority assignment of (P = DEADLINE) is used. In the following experiments, we will refer to the performance achievable under this scenario as **DIRECT** since the Bell–LaPadula conditions prevent direct unauthorized access to secret data.

6.6. 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 to 20 percent are taken to represent system performance under "normal" loads, while KillPercent values in the range of 20 percent to 100 percent represent system performance under "heavy" loads [19].[5]

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 protocol 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.

6.7. Default Parameter Settings

The default settings used in our experiments for the workload and system parameters are listed in Table 1. These settings were chosen to ensure significant data contention and resource contention in the system, thus helping to bring out the performance differences between the various CC protocols. They are also consistent with the values used in earlier RTDBS studies, facilitating comparison with their results. While the absolute performance profiles of the CC protocols would, of course, change if alternative parameter settings are used, we expect that the *relative* performance will remain qualitatively similar since the model parameters are not protocol-specific.

7. Experiments and Results

Using the real-time database model described in the previous section, we evaluated the performance of the CC protocols presented in Sections 4 and 5 for a variety of security-classified transaction workloads and system configurations. The simulator was written using the Simscript II.5 discrete event simulation language [25]. 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 [14]. 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 percent at the 90 percent confidence level, with each experiment having been run until at least 10000 transactions were processed by the system.

The simulator was instrumented to generate several other statistics, including resource utilizations, number of restarts, etc. These secondary measures help to explain the performance behavior of the CC protocols under various workload and system conditions.

7.1. Experiment 1: Resource and Data Contention

The settings of the workload parameters and system parameters for our first experiment are the default values listed in Table 1, resulting in a system with two security levels: Secret and Public, and significant levels of both resource and data contention. For this system, Figures 5a and 5b show the KillPercent behavior as a function of the overall transaction arrival rate.[6] In Figure 5a, the overall kill percentages of the fully secure algorithms and their DIRECT (Bell–LaPadula) counterparts is profiled. We see here that at normal loads the performance of the secure algorithms is worse than that of their DIRECT counterparts. In contrast, under heavy loads the performance of the secure algorithms is actually better than that of the DIRECT algorithms. The reason for this is that while the Earliest Deadline priority assignment is excellent for a set of tasks that *can* be completed before their deadlines, it becomes progressively worse as the task set overloads its capacity [20]. In this situation, the secure protocols feature of grouping the transactions into prioritized levels means that Earliest Deadline is operational within *smaller* sets of transactions, leading to improved performance at higher loads. In summary, although elimination of covert channels results in performance degradation at normal loads, it reduces the kill percentage under heavy loads.

Focusing on the secure real-time algorithms, we observe first in Figure 5a that the performance of 2PL-HP is significantly worse than that of OPT-WAIT. In fact, 2PL-HP's performance is no better than that of S2PL which, as mentioned in Section 4, is a non-real-time protocol ! The poor performance of 2PL-HP is primarily because of its "wasted restarts" problem, which was its main drawback in unsecure real-time CC also [19]: A transaction may be restarted by a higher priority transaction that later misses its deadline. This means that the restart did not result in the higher priority transaction meeting its deadline. In addition, it may cause the lower priority transaction to miss its deadline as well, apart from wasting the resources invested in the transaction prior to its restart.

The effect of the wasted restarts problem is *magnified* in the secure domain for the following reason: In unsecure real-time CC, a transaction that is close to its deadline would usually not be restarted since it would have high priority. However, in the secure model, where the transaction *level* is also a factor in the priority assignment, Secret transactions that are close to their deadlines may still be restarted due to data conflict with a Public transaction.

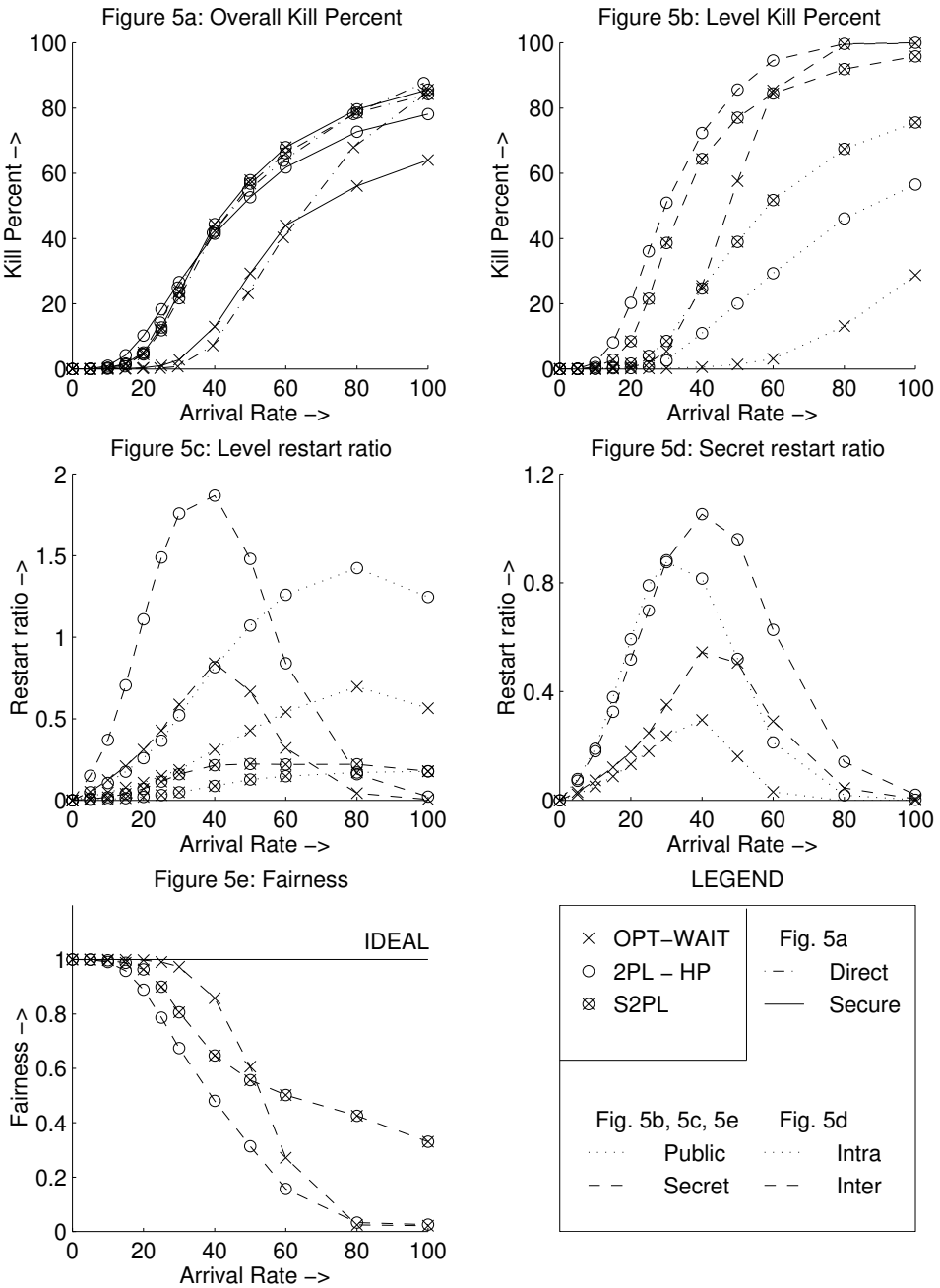

Figure 5. **Resource and Data Contention**

Moving on to OPT-WAIT, we find that it provides the *best* performance across the entire loading range. Further, there is a change of performance behavior in the secure environment in that the gap between its performance and that of 2PL-HP is *more* than that observed for unsecure real-time CC [19]. This is due to the access pattern restrictions of the Bell-LaPadula model: The definition of conflict in forward optimistic CC is that a conflict exists between the validating transaction V and an executing transaction E if and only if the intersection of the write set of V and the current read set of E is non-empty. For the LaPadula model, where *blind writes* are permitted due to the "read-below, write-above" paradigm, optimistic algorithms will correctly conclude that there is no conflict between items that are in the intersection of the write set of V and the write set of E but not in the read set of E. In fact, it is easy to see that *a validating Secret transaction will never have conflicts with executing Public transactions* in this model. Note that for 2PL-HP, however, blind-writes can unnecessarily result in write-write conflicts and cause either blocking or restarts, thereby further deteriorating its performance.

Finally, considering S2PL, we find that it manages to perform on par with 2PL-HP in spite of not being deadline cognizant. This is due to its "optimistic-like" feature of *virtual commit*, which considerably reduces the amount of blocking associated with 2PL. This phenomenon is similar to that seen in [19], wherein a conventional (non-real-time) optimistic protocol performed better than locking-based real-time protocols.

In Figure 5b, we present the kill percentages of the various CC protocols on a *per-security-level* basis. This graph clearly shows how the high-security Secret transaction class (dashed lines) suffers much more than the Public transaction class (dotted lines) to satisfy the goal of avoiding covert channels. Figure 5c provides statistics about the corresponding breakup of the "restarts ratio" (the average number of restarts of a transaction) on a level basis. We see here that Secret transactions are restarted much more often than Public transactions under normal loads. Under heavy loads the number of restarts decrease for Secret transactions since resource contention, rather than data contention, becomes the more dominant reason for these transactions missing their deadlines.

In Figure 5d, we present a different view of the transaction restarts picture. Here, the restarts of Secret transactions are categorized into those caused by Public transactions (i.e., *inter-level* restarts) and those caused by Secret transactions (i.e., *intra-level* restarts). Note that this breakup is meaningful only for Secret transactions since all restarts are intra-level for Public transactions. The graph clearly shows that Secret transactions suffer more from inter-level conflicts (dashed lines) than from intra-level conflicts (dotted lines) over most of the loading range.

Finally, in Figure 5e, we plot the *fairness factor* of each CC protocol for the Secret transaction class. We observe that at light loads when virtually *all* transactions make their deadlines, all the CC protocols are (trivially) fair. As the loading increases, however, the protocols become increasingly unfair since they selectively miss the deadlines of Secret transactions to accommodate the Public transactions. With regard to the relative fairness of the secure real-time algorithms, the graph clearly shows that OPT-WAIT provides much better fairness than 2PL-HP, and

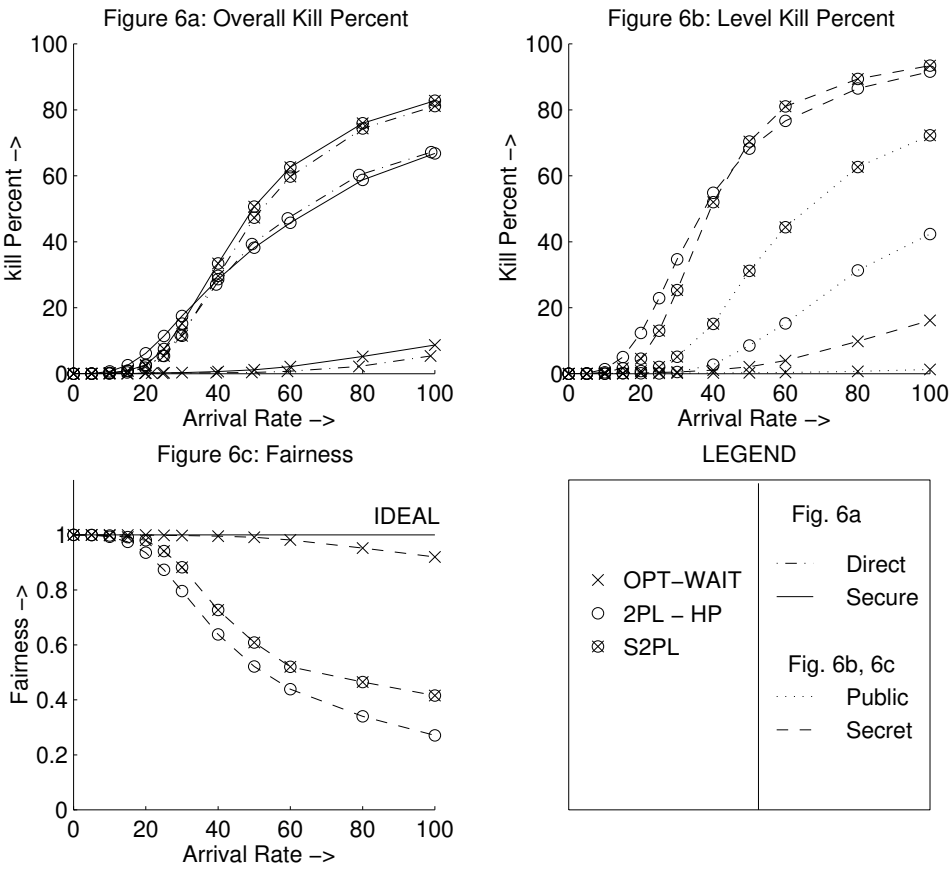

Figure 6. **Pure Data Contention**

that OPT-WAIT is the best overall. It may seem in Figure 5e that, at high loads, S2PL is more fair than OPT-WAIT. Note, however, that this is really a "pseudo-fairness" since it arises out of S2PL, due to its non-real-time nature, *missing* a large fraction of the Public transaction deadlines, rather than out of *meeting* the deadlines of more Secret transactions.

In summary, for the workload and system configuration considered in this experiment, OPT-WAIT provides the lowest kill percentage, both on an overall basis and on a per-level basis, *and* the best overall fairness.

7.2. Experiment 2: Pure Data Contention

The goal of our next experiment was to isolate the influence of data contention on the performance of the CC protocols. For this experiment, therefore, the resources were made "infinite", that is, there is no queueing for these resources [3].[7]

The performance results for this system configuration are presented in Figures 6a through 6c. We observe in these figures that the differences in the relative performance of the various protocols increases as compared to those seen in the previous experiment. The overall (Figure 6a) as well as the per-level (Figure 6b) kill percentages of OPT-WAIT are considerably better than those of 2PL-HP. Here, OPT-WAIT does better than 2PL-HP for two reasons: First, the basic wasted restarts problem of 2PL-HP occurs here too and is magnified due to the higher level of data contention. Second, the blocking component of 2PL-HP reduces the number of transactions that are making progress in their execution. This blocking causes an increase in the average number of transactions in the system, thus generating more conflicts and a greater number of restarts. With OPT-WAIT, however, transactions are never blocked for data access.

Finally, in Figure 6c, which profiles the fairness factors of the protocols, OPT-WAIT is almost ideally fair over most of the loading range since it misses very few deadlines overall. In contrast, 2PL-HP and S2PL are noticeably unfair with increasing loading levels.

In the next set of experiments in this section, we do not present S2PL's performance since it performed significantly worse than OPT-WAIT in all of these experiments. We will, however, return later to S2PL as part of a dual-CC protocol in Experiment 7, which evaluates the dual-CC approach.

7.3. Experiment 3: The Effect of Write Probability

This experiment was conducted to profile the effect of the write probability on the real-time performance for three representative transaction arrival rates: LOW (arrival rate = 15), MEDIUM (arrival rate = 50) and HIGH (arrival rate = 80). For each of these arrival rates, the write probability was varied from 0.0 to 1.0 in steps of 0.1 and the resulting performance was observed.

Figure 7 shows the KillPercent values for MEDIUM loads (we omit the HIGH and LOW performance graphs for brevity as the results are qualitatively similar). The locking protocol 2PL-HP shows poor performance at all write probability levels as compared to the OPT-WAIT protocol and this difference is magnified at high write probability levels. For OPT-WAIT, since the inter-level writes are blind writes (due to Bell–LaPadula conditions), write-write conflicts do not appear in the conflict set as explained in Experiment 1. Therefore, at high write probabilities, when most of the operations are writes rather than reads, the data conflicts in fact *reduce* in frequency. This benefit is not realized by 2PL-HP, however, as discussed in Experiment 1.

7.4. Experiment 4: Restricted Write Model

In many secure systems, the Bell–LaPadula model of "read below, write above" is further restricted to allow only "read below", that is, blind writes by low security transactions to high security data are disallowed. We conducted experiments to evaluate the performance behavior of the CC protocols under this *Restricted*

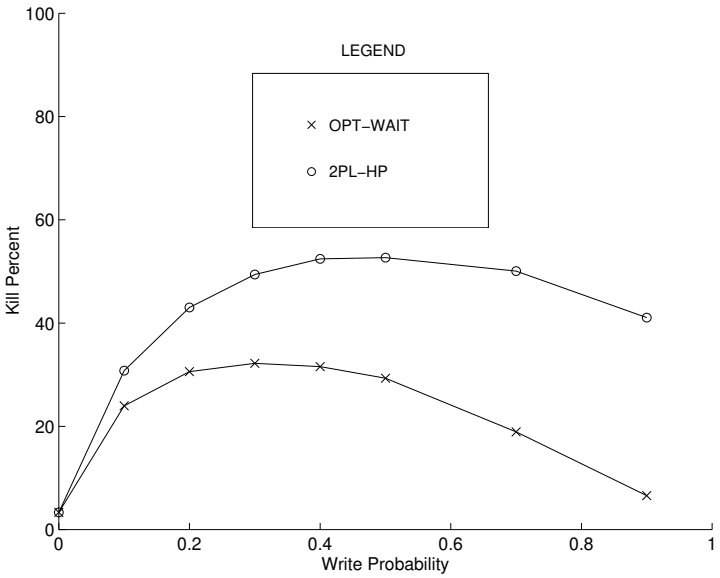

Figure 7. **Effect of Write Probability (MEDIUM arrival rate)**

Write model and a sample result is shown in Figures 8a through 8c for the same environment as that modeled by Experiment 1. In these figures, we observe that the KillPercent behavior (Figures 8a, 8b) is similar to that of Experiment 1. A noticeable difference, however, is that the Public kill percentages are now higher for all the protocols whereas the Secret kill percentages are lower, especially under light loads. The reasons for this behavior are as follows:

- The increase in missed Public deadlines is due to the increased intra-level data conflicts arising out of the confinement of Public writes to the Public partition of the database. This is further confirmed in Figure 8c, where we observe that the Public restart ratios – for all the protocols – are considerably higher as compared to their corresponding values in Experiment 1.

- In contrast, the Secret performance improves, especially under light loads, because its inter-level data conflicts are reduced since inter-level writes are disallowed. At high loads, however, the improvement becomes negligible because the physical resources are mainly hogged by the Public transactions and therefore Secret transactions miss their deadlines due to *resource contention*, rather than data contention.

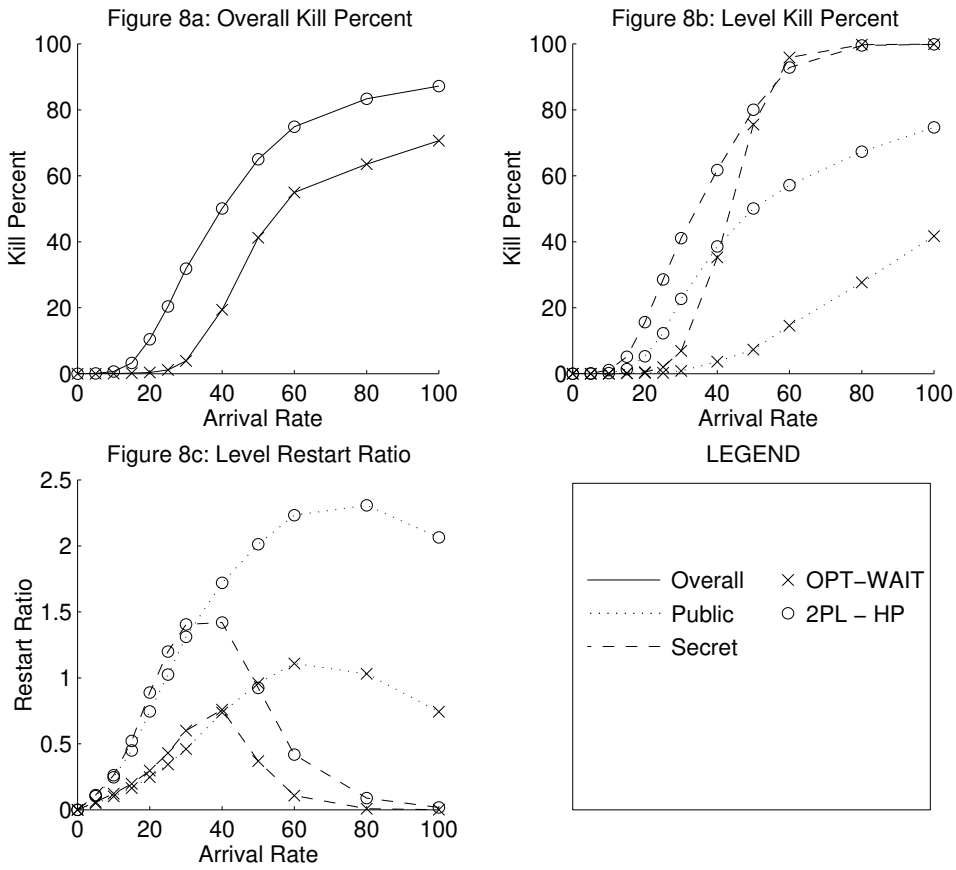

Figure 8. **Restricted Write Model**

7.5. Experiment 5: SRTDBS versus MULTI-CLASS RTDBS

There have been quite a few studies of RTDBS systems (e.g., [9]) where the input is composed of a set of disjoint prioritized transaction classes. An SRTDBS is similar to such a generic multi-class RTDBS with the security levels in the SRTDBS corresponding to the transaction classes in the multi-class environment. A major difference, however, is that the SRTDBS enforces the Bell–LaPadula conditions for data access, whereas the *entire database* is usually accessible to all classes in the multi-class environment. We conducted an experiment to evaluate the relative behavior of the SRTDBS and MULTI-CLASS systems for the environment modeled in Experiment 1. The priority assignment for the MULTI-CLASS system is the same as that of the SRTDBS, namely, P = (LEVEL, DEADLINE).

The results of this experiment are shown in Figure 9. We see here that, qualitatively, the SRTDBS performance is very similar to the MULTI-CLASS perfor-

mance. Interestingly, however, the quantitative performance of the SRTDBS is virtually identical to that of MULTI-CLASS for the Public transactions and *visibly better* with regard to the Secret transactions. The reason for this behavior is as follows:

- For the Public transactions, the conflict level in the SRTDBS case is proportional to $\frac{R * W/2}{D/2}$, where R and $W/2$ are the proportion of Public reads and writes to the Public partition, and $D/2$ is the Public partition size. Note that Public transactions do not have conflicts in the Secret partition due to the writes being blind writes.

  In the MULTI-CLASS environment, the above computation works out to $\frac{R * W}{D}$ since all Public reads and writes are over the entire database.

  Since the two conflict levels are the same, the performance of the Public transactions does not alter noticeably between the SRTDBS and MULTI-CLASS systems.

- For the Secret transactions, Secret writes conflict only among themselves in the SRTDBS environment. In the MULTI-CLASS environment, however, the Secret writes conflict with the Public reads and writes in the Public partition, resulting in increased conflict levels. This results in the performance of the SRTDBS system being superior to that of the MULTI-CLASS system.

7.6. Experiment 6: Five Security Levels

The previous experiments modeled a two-security-level system. We now investigate how the performance behavior scales with the number of security levels by modeling a system with *five* security levels: TopSecret, Secret, Confidential, Classified and Public. The results for this experiment are shown in Figures 10a through 10e.

In Figure 10a, we see that there is a greater difference between the performance of the secure algorithms and their DIRECT counterparts at normal loads, as compared to the equivalent two-level experiment (Experiment 1). This is because in a five-level system, priority is much more level-based than deadline-based, thereby denying Earliest Deadline its ability to complete most transactions in a feasible set under normal loads. Under heavy loads, however, the smaller sizes of the transaction sets in each level results in Earliest Deadline performing well for the low security transactions. (This feature of Earliest Deadline was used in the Adaptive Earliest Deadline scheduling algorithm described in [20] where transactions are split into prioritized groups with the size of the highest priority group set equal to an estimate of the maximum number of transactions that could be successfully completed by Earliest Deadline.)

In Figures 10b and 10c, we plot the KillPercent on a per-security-level basis for OPT-WAIT and 2PL-HP, respectively. These graphs clearly show the extent to which the kill percentages are skewed among the various transaction security levels,

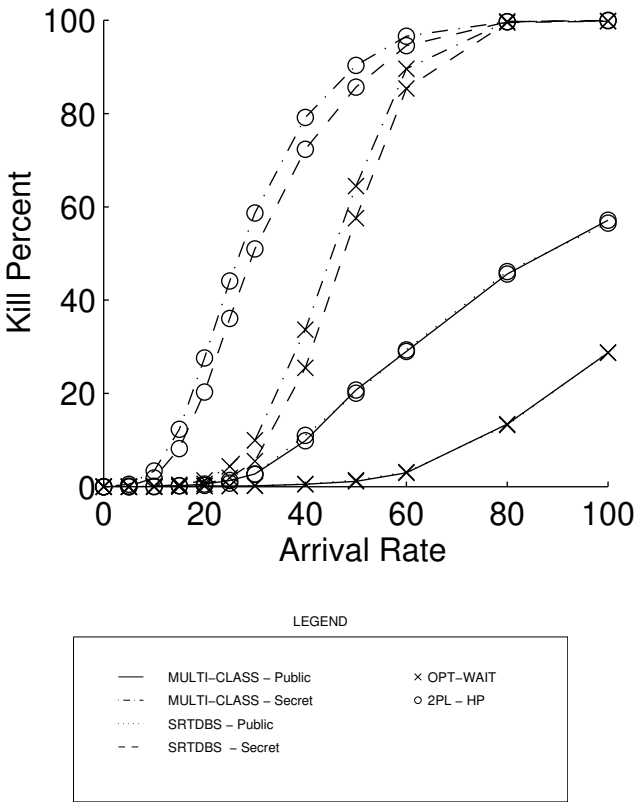

Figure 9. **SRTDBS versus MULTI-CLASS**

with the Top Secret class having the most number of killed transactions deadlines and the Public class having the least. The graphs also show that OPT-WAIT's performance is better than that of 2PL-HP for *every* transaction security level.

In Figures 10d and 10e, the fairness factors for the top four security levels (Top Secret, Secret, Confidential and Classified) are plotted on a per-security-level basis for OPT-WAIT and 2PL-HP, respectively. 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). We also find that OPT-WAIT's performance is more fair than that of 2PL-HP for *every* transaction security level.

In summary, just as in the two-security-level experiment, we find that OPT-WAIT provides the lowest kill percentage, both on an overall basis and on a per-level basis, and the maximum fairness (this observation regarding OPT-WAIT is true also with regard to the S2PL protocol whose results were not presented here).

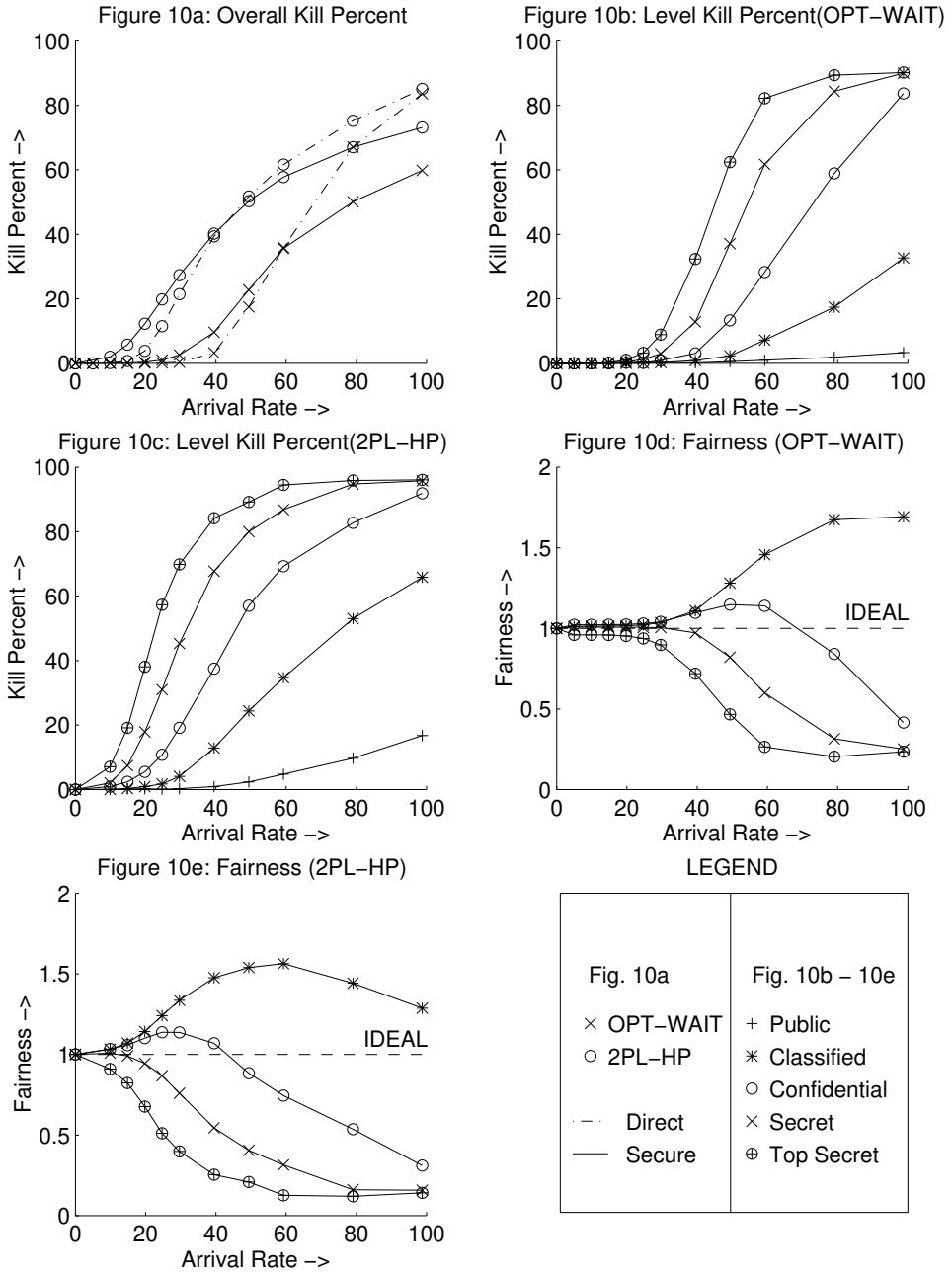

Figure 10. **Five Security Levels**

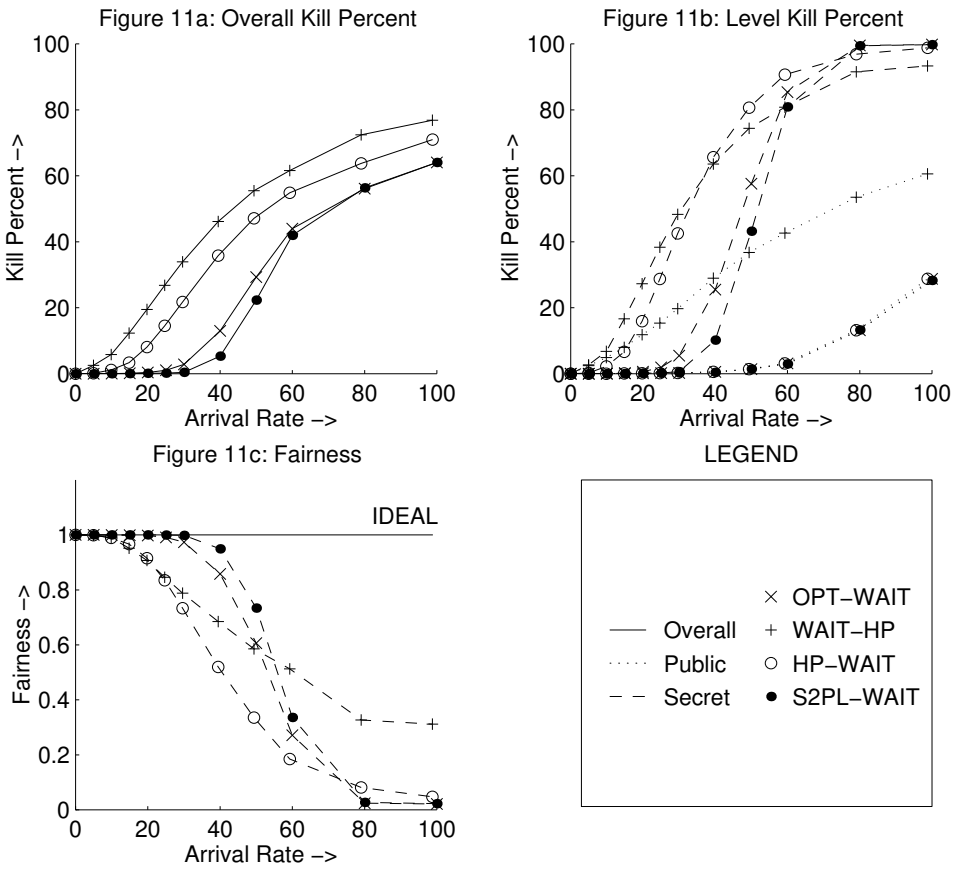

Figure 11. **Dual-CC Approach**

7.7. Experiment 7: Dual-CC Approach

In the final experiment of this section, we evaluate the performance of our new *dual-CC* approach to secure real-time CC where inter-level conflicts are handled by one protocol while intra-level conflicts are handled by a different protocol. In Figures 11a through 11c we show the performance of the three dual protocols (WAIT-HP, HP-WAIT and S2PL-WAIT) for the baseline environment. For the sake of comparison, the performance of a pure OPT-WAIT protocol is also shown in these graphs.

Focusing our attention on the WAIT-HP and HP-WAIT dual-CC protocols, we first observe in Figure 11a, which compares the overall kill percentages of the protocols, that the performance of both these approaches is considerably *worse* than that of the pure OPT-WAIT protocol. The reason that OPT-WAIT remains the best among them is that 2PL-HP is a wasteful algorithm, as seen in the previous

experiments, and therefore "dilutes" the effect of OPT-WAIT in both the dual-CC protocols.

We also observe in Figure 11a that the performance of WAIT-HP is worse than that of HP-WAIT throughout the loading range. The reason for this is that, for the workload modeled in this experiment, the number of intra-level conflicts are significantly more than the number of inter-level conflicts. Therefore, the algorithm which is used to handle intra-class conflicts has more effect on the overall kill percentage than the protocol used to handle inter-class conflicts. In WAIT-HP, it is 2PL-HP which handles intra-class conflicts and this results in worse performance than that of HP-WAIT, which uses OPT-WAIT to handle this category of conflicts.

The kill percentages of the protocols on a per-security-level basis is provided in Figure 11b and the fairness factors are shown in Figure 11c. An interesting feature in these graphs is that at high loads, the fairness of WAIT-HP is *greater* than that of OPT-WAIT – this is the first time in all the experiments discussed so far that a real-time protocol has improved on OPT-WAIT's fairness performance. The reason that this happens is the following: In WAIT-HP, due to 2PL-HP being used for intra-class conflicts, many of the Public transactions are so busy "fighting" each other that they don't ever reach the end of their execution, which is when the OPT-WAIT policy of checking for inter-class conflicts comes into play. Therefore, Secret transactions suffer much less restarts from the Public transactions. This is clearly seen in Figure 11b where the kill percentage of the Public transactions for WAIT-HP is quite high as compared to the corresponding numbers for the other protocols.

Moving on to the S2PL-WAIT dual-CC protocol, we find that, unlike the other two dual protocols, it performs better than OPT-WAIT, especially at lower loading levels. For example, at an arrival rate of 40 transactions per second, S2PL-WAIT more than halves the kill percentage suffered by OPT-WAIT (Figure 11a). The reason that this combination works well is that Secure 2PL handles inter-class conflicts *without* resorting to restarts unlike in WAIT-HP or HP-WAIT. This means that Secret transactions suffer much less in this environment (as confirmed in Figures 11b and 11c). At the same time, using OPT-WAIT for handling intra-class conflicts helps to derive the inherent good real-time performance associated with this protocol. At high loads, S2PL-WAIT performs almost identically to OPT-WAIT because, in this region, the primary reason for transactions missing their deadlines is resource contention, rather than data contention – therefore, the virtual commit feature of S2PL rarely provides the intended benefits.

In summary, this experiment shows that by carefully choosing the right combination of protocols in the dual-CC approach, we can design CC algorithms that provide even better kill percent and fairness performance than OPT-WAIT. This highlights the power and flexibility that is provided by the dual-CC approach. In fact, it may be possible to develop dual protocols that perform *even better than S2PL-WAIT* by appropriately choosing the constituent protocols.

7.8. Other Experiments

We conducted several other experiments to explore various regions of the workload space. In particular, we evaluated the sensitivity of the results to the database size, number of security levels, deadline slack factor, data access patterns, etc. The complete details and results of these experiments are available in [14]. Our general observation was that the relative performance behaviors of the protocols in these other experiments remained qualitatively similar to those seen in the experiments described here. That is, OPT-WAIT performed the best among the individual protocols, while S2PL-WAIT provided the best overall performance with respect to both the individual protocols and the dual-CC protocols.

8. Class Fairness

In the previous section, we showed that by proper choice of concurrency control mechanism, it is possible to improve the real-time performance without violating security. However, even with the S2PL-WAIT protocol, which provided the best overall real-time performance, a marked *lack of fairness* was observed with respect to the high security classes, especially for finite resource environments. 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 [22], 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 CC protocol, 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 present **GUARD**, an adaptive admission-control policy that attempts to achieve this goal.

```
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] = Admit[Public] * 1.05
       else
           Admit[Public] = 1.0
       sleep T seconds
       measure KillPercent[Public] and KillPercent[Secret]
       goto statement LOOP
```

Figure 12. **The GUARD Admission Control Policy (Two Level)**

8.1. 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 Section 8.2, we extend the GUARD design to an arbitrary number of security levels.

For the two-level environment, the GUARD admission control policy is shown in Figure 12. 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.[8] The increase or decrease margin has been set to a nominal 5 percent. 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 percent 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" had also been earlier observed with the S2PL CC protocol in Experiment 1 of Section 7. 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.

```
Initial condition: Admit[1:N] = 1.0, KillPercent[1:N] = 0.0
LOOP:  for i = 1 to N-1
         do
         if OverallKillPercent > 5.0 then
              do
              FairFactor =  KillPercent[i] / OverallKillPercent
              if (FairFactor < 0.95) then
                  Admit[i] = Admit[i] * 0.95
              else if (FairFactor > 1.05) then
                  Admit[i] = Admit[i] * 1.05
              done
          else
              Admit[i] = 1.0
         done
       sleep T seconds
       measure KillPercent[1:N]
       goto statement LOOP
```

Figure 13. **The General GUARD Admission Control Policy**

8.1.1. 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 each computation, $log_2 3 = 1.6$ bits of information can be transmitted (based on information-theoretic considerations [32]) and since the computation is made once every T seconds, the total 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 $T_{min} = 1.6$ seconds to be the minimum re-computation period.

8.2. 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 below in Figure 13. Note that the channel capacity in the general case is $log_2 3^{N-1} = 1.6(N-1)$ bits and therefore $T_{min} = (N-1) * 1.6$ seconds.

8.3. 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[9] 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 Section 8.1.1, 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 $S = T/16$.

8.4. 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:

FULL: A full secure SRTDBS equipped with the S2PL-WAIT CC manager

ORANGE: An orange secure SRTDBS that is equipped with the S2PL-WAIT CC manager and the GUARD admission controller.

Note that the FULL system was already evaluated in Section 7 – we reproduce its results here for the purpose of comparison. Further, we show only the results for the S2PL-WAIT CC manager here since, as per the evaluation presented in Section 7, it performed the best among all the CC protocols that we considered. The results for the other CC protocols, which were qualitatively similar, are available in [14].

8.4.1. Experiment 8: Two Security Levels In our first experiment, we evaluate the performance of FULL and ORANGE for a two security-level enviroment. For this experiment, the GUARD policy parameters, S and T, are set to $0.1s$ and $1.6s$, respectively.

The results of this experiment are shown in Figures 14a–c, which capture the overall kill percent, the level-wise kill percent, and the class fairness, respectively. In these figures we see that the ORANGE system achieves *close to ideal fairness*. Further, at low loads, ORANGE has only *a small increase of the overall kill percentage* with respect to FULL, whereas at heavy loads, it *actually does slightly better*. This is explained as follows: At low loads, due to the inherent lag involved in the feedback process, the GUARD policy 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 FULL.

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

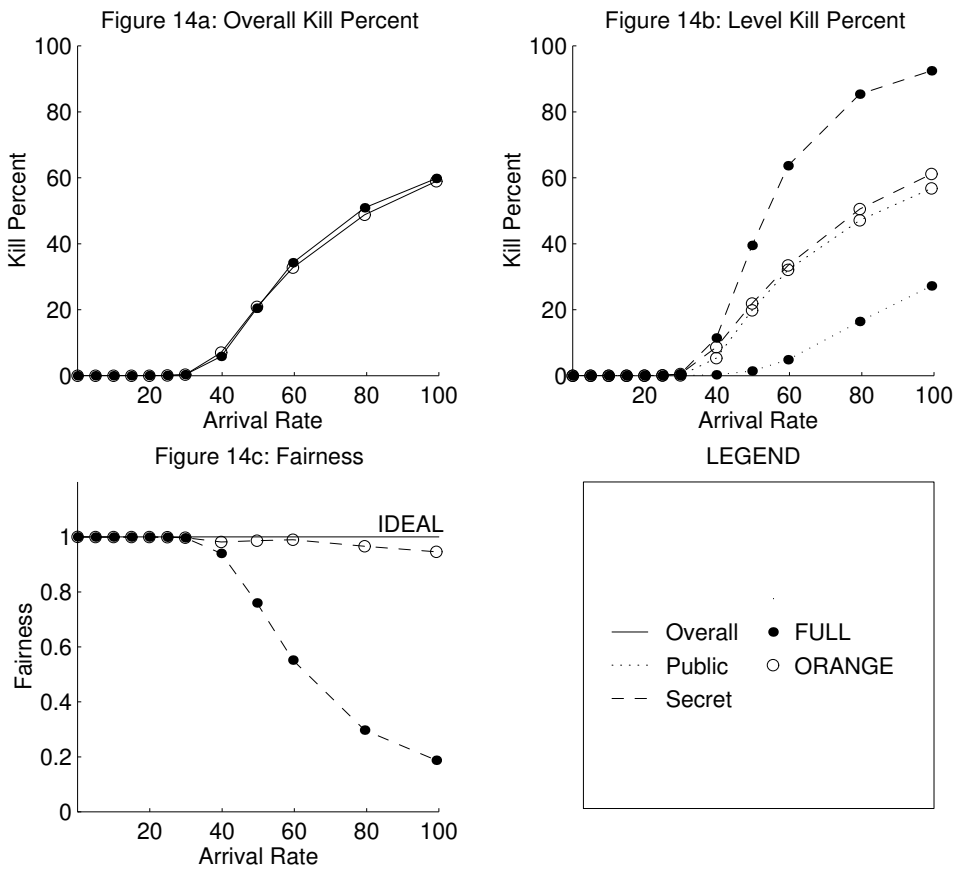

Figure 14. **Two Security Levels**

8.4.2. Experiment 9: Five Security Levels In this experiment, we compare the FULL and ORANGE systems for a five-security-level environment. The results of this experiment are shown in Figures 15a-c. We see in these figures that the ORANGE system provides, at only a marginal performance cost, reasonably close to ideal fairness characteristics throughout the loading range in this many-security-level environment as well.

9. Related Work

The design of secure CC protocols in the context of *conventional* DBMS has been investigated by several research groups (see [38, 6] for surveys). In comparison, little attention has been given to developing secure CC protocols for *real-time* database systems. The only work that we are aware of in this area is a series of papers by Son et al [11, 29, 34, 35, 36]. In particular, a concurrency control protocol

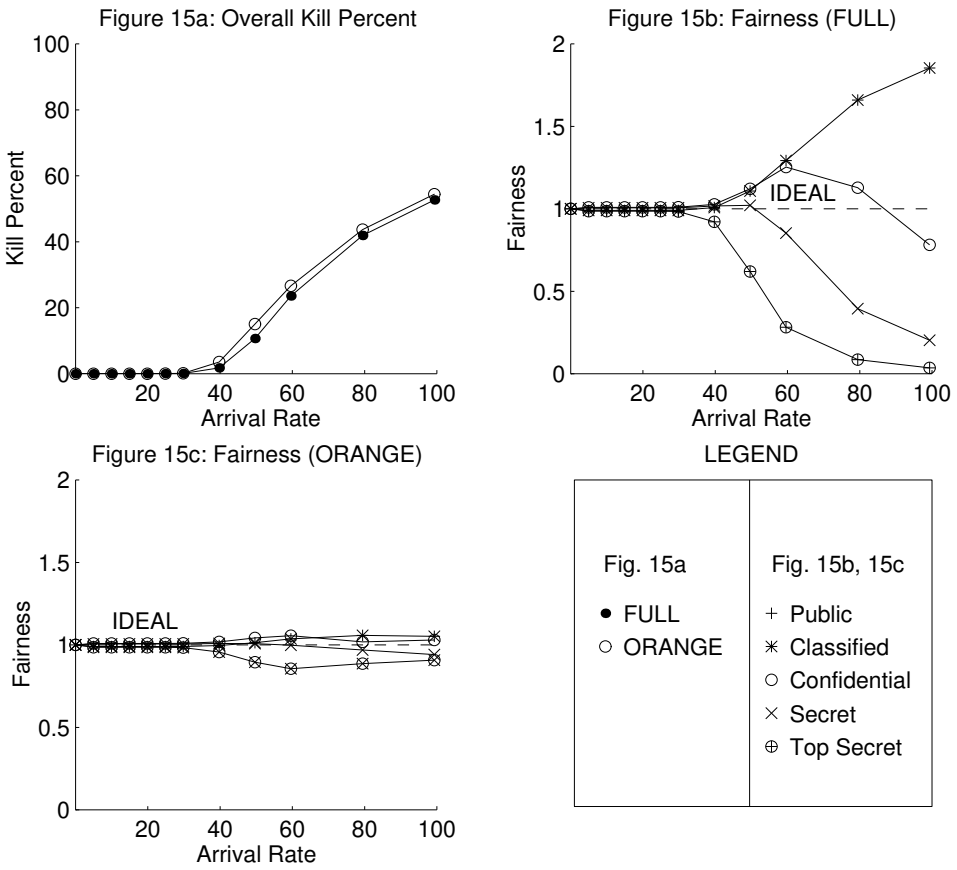

Figure 15. **Five Security Levels**

called Adaptive 2PL (A2PL) that attempts to *balance* the dual requirements of security and timeliness was presented in [11, 34, 35]. In their scheme, transactions *dynamically* choose between an unsecure version of 2PL-HP and the secure S2PL described in Section 4. The goal of the A2PL protocol is to *tradeoff* security for real-time performance with the tradeoff depending on the state of the system and the application's requirements.[10] In contrast, in our work, we have assumed that full (or orange) security is a *fundamental* requirement and that it is not permissible to improve the real-time performance at the cost of security.

There are also a few design and implementation difficulties associated with the A2PL protocol:

1. In A2PL, the decision to use 2PL-HP or S2PL is a function of the system state and the application requirements. The priority assignments in these protocols are independent and may therefore work at cross purposes. For example, con-

sider the case where a tight-deadline Secret transaction requests a data item held by a slack-deadline Public transaction. Assume that the A2PL protocol chooses the real-time option to resolve this conflict and therefore grants the lock to the requester aborting the Public holder. Assume that another slack-deadline Public transaction now makes a conflicting request on the same data item. If the A2PL protocol decides to follow the secure option to resolve this conflict, the Secret holder will have to be restarted. This means that a transaction that was earlier assisted based on deadline considerations now gets hindered on security considerations, negating the priority assignment objective.

2. Since the A2PL protocol takes only data conflicts into account, it cannot control the conflicts at other resources. Therefore, here again, a transaction that was assisted to make its deadline at the data may be hindered at the physical resources. In contrast, our GUARD policy applies a system-wide control that does not depend on the location of the performance bottleneck.

3. For applications where full security is a fundamental requirement, the A2PL protocol reduces to plain S2PL, which does not have any real-time features.

In [29], a concurrency control protocol that ensures *both* security and timeliness is proposed. For this scheme, however, the RTDBS is required to maintain *two* copies of each data item. Further, transactions are required to obtain *all* their data locks *before* starting execution (i.e., conservative locking). These requirements limit the applicability of the protocol. In our work, we have considered more general database environments where all data items are single-copy and transactions acquire data locks dynamically.

Another feature of their work is that it is primarily addressed towards "soft-deadline" applications, that is, real-time applications in which there is value to completing tasks even *after* their deadlines have expired. In contrast, we have concentrated on firm-deadline applications. The type of deadline has a significant impact on both the performance evaluation model and on the interpretation of the results, as observed earlier for (unsecure) real-time transaction concurrency control [1, 19].

10. Conclusions

In this paper, we have quantitatively investigated the performance implications of maintaining covert-channel-free security in a firm-deadline real-time database system. Unlike previous studies, which used a *tradeoff* approach between security and timeliness, we have considered full (or orange) security to be a *correctness* requirement. In comparison, the number of killed transactions is a *performance* issue. Therefore, our study investigates the problem of how to minimize the number of missed transaction deadlines *without* compromising the desired level of security. To the best of our knowledge, this is the first detailed study of real-time database security in the firm-deadline context.

We first identified that, in order to satisfy the requirement of non-interference, only those real-time concurrency control protocols that are free from priority inversion can be used in a secure RTDBS. This requirement ruled out several previously proposed real-time CC protocols, including algorithms such as 2PL Wait Promote [1] and WAIT-50 [19].

Then, using a detailed simulation model of a firm-deadline RTDBS, we studied the relative performance of the secure versions of the 2PL-HP and OPT-WAIT real-time concurrency control algorithms; a non-real-time secure algorithm, S2PL, was also included in the evaluation suite. The performance of these algorithms was also evaluated for a baseline system where only direct unauthorized access, but not covert channels, is prevented.

Our experiments showed that, under normal loads, the overall kill percent of the secure system is worse than that of the direct system, whereas under heavy loads, it is the other way around. Within the secure system, the performance of high-security transactions was significantly worse than that of the low-security transactions. Among the secure concurrency control protocols, OPT-WAIT performed best in minimizing the kill percentages on both an overall basis and on a per-level basis. Moreover, it exhibited the maximum degree of fairness. These results show that OPT-WAIT, which provided excellent performance in traditional (unsecure) real-time concurrency control [19], continues to perform well even in the secure real-time domain.

We also proposed a novel dual-CC approach to secure concurrency control wherein different concurrency control algorithms are used to resolve inter-level conflicts and intra-level conflicts. A global serialization order was ensured, in spite of having multiple CC algorithms operating simultaneously, by using the MCC system architecture described in [37]. The dual-CC combinations of HP-WAIT and WAIT-HP were implemented and evaluated – they both generally perform worse than pure OPT-WAIT. However, the dual combination of S2PL-WAIT performs better than OPT-WAIT, especially at lower kill percent levels. This is because S2PL is a non-restart-oriented algorithm unlike both OPT-WAIT and 2PL-HP, and therefore ensures reduction of the harm done to high-security transactions.

Another advantage of the dual-CC approach, not exploited here, is that the separation of security and timeliness concerns makes it possible to use even *unsecure* real-time CC algorithms (e.g., Wait-Promote, WAIT-50) for resolving intra-level conflicts. The dual-CC approach therefore empowers the use, even in the secure RTDBS domain, of the rich set of real-time CC algorithms developed during the last decade.

A fundamental problem associated with the elimination of covert channels is that the system becomes biased against the high security transactions. To address this problem, we introduced the GUARD admission control policy which implemented a simple feedback-based transaction admission control mechanism. The bandwidth of the covert channel introduced by this mechanism was bounded by appropriately setting the feedback period. Experimental evaluation of an orange-secure version of GUARD indicated that it provides close to ideal fairness at little cost to the overall real-time performance.

In closing, we suggest that SRTDBS designers may find the **S2PL-WAIT** CC protocol and the **GUARD** admission control policy to be attractive choices for achieving high-performance secure real-time transaction processing.

10.1. Future Work

Our goal in this paper was to take a first step in providing effective transaction processing solutions for real-time database applications with security requirements. We hope that it will lead to further activity in this area, especially with regard to extending our solutions, which were derived for a specific security and real-time framework, to more general environments. Important issues in this regard include supporting partially-ordered security frameworks such as lattices, evaluating performance profiles of snapshot-based concurrency control algorithms, assessing the tradeoffs involved in replicated architectures, incorporating security requirements such as data integrity and availability, and handling real-time applications with soft deadlines. In addition, solutions for other transaction processing components including buffer management and transaction recovery need to be designed.

Acknowledgements This work was supported in part by a research grant from the Dept. of Science and Technology, Govt. of India.

Notes

1. Or equivalently, the percentage of missed transaction deadlines.
2. A new reader joins a group of lock-holding readers only if its priority is higher than that of *all* the waiting writers.
3. A similar, but unrelated, dual-protocol strategy has been used earlier for the purpose of *task scheduling* in the Secure Alpha project [16].
4. The non-interference method is used for eliminating covert channels at the physical resources also.
5. 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 (as in our experiments), however, provides valuable information on the response of the algorithms to brief periods of stress loading.
6. The physical explanation for why the KillPercent graphs show a non-linear "S-shaped" behavior have been identified early on in the RTDBS literature, including queueing-theory-based analyses in [17, 18].
7. While abundant resources are usually not to be expected in conventional DBMS, they may be more common in RTDBS environments since many real-time systems are sized to handle transient heavy loading. This directly relates to the application domain of RTDBSs, where functionality, rather than cost, is often the driving consideration.
8. No restrictions are placed on the admission of Secret transactions.
9. This is similar to the system load computation method used in the Unix operating system [26].
10. The tradeoff approach, and alternative schemes to implement the tradeoff, have also been considered in the Secure Alpha project [16], which investigated the interactions between security and timeliness in the context of a distributed real-time operating system.

References

1. R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation", *ACM Trans. on Database Systems*, September 1992.
2. M. Abrams, S. Jajodia and H. Podell, *Information Security*, IEEE Computer Society Press, 1995.
3. R. Agrawal, M. Carey and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications", *ACM Trans. on Database Systems*, 12(4), December 1987.
4. D. Agrawal, A. El Abbadi and R. Jeffers, "Using Delayed Commitment in Locking Protocols for Real-Time Databases", *Proc. of ACM SIGMOD Conf.*, June 1992.
5. P. Amman, F. Jaeckle and S. Jajodia, "A Two-Snapshot Algorithm for Concurrency Control in Secure Multilevel Databases", *Proc. of IEEE Symp. on Security and Privacy*, 1992.
6. V. Atluri et al, "Multilevel Secure Transaction Processing: Status and Prospects", *Database Security, X: Status and Prospects*, Chapman & Hall, 1997, eds. Pierangela Samarati and Ravi Sandhu.
7. S. Castano, M. Fugini, G. Martella and P. Samarati, *Database Security*, Addison-Wesley, 1995.
8. "DOD Trusted Computer System Evaluation Criteria", *Department of Defense Standard, DoD 5200.28-STD*, December 1985.
9. A. Datta et al, "Multiclass Transaction Scheduling and Overload Management in Firm Real-time Database Systems", *Information Systems*, 21(1), March 1996.
10. R. David, "Secure Concurrency Control", *Master's Thesis*, Univ. of Virginia, May 1993.
11. R. David, S. Son and R. Mukkamala, "Supporting Security Requirements in Multilevel Real-Time Databases", *Proc. of IEEE Symp. on Security and Privacy*, May 1995.
12. K. Eswaran et al, "The Notions of Consistency and Predicate Locks in a Database System", *Comm. of ACM*, November 1976.
13. D. Georgakopoulous, M. Rusinkiewicz and A. Sheth, "On Serializability of Multidatabase Transactions through Forced Local Conflicts", *Proc. of 7th IEEE Intl. Conf. on Data Engineering*, 1991.
14. B. George, "Secure Real-Time Transaction Processing", *PhD Thesis*, Indian Institute of Science, December 1998.
15. J. Goguen and J. Meseguer, "Security Policy and Security Models", *Proc. of IEEE Symp. on Security and Privacy*, 1982.
16. I. Greenberg et al, "The Secure Alpha Study (Final Summary Report)", *Tech. Report ELIN A012*, SRI International, June 1993.
17. J. Haritsa, "Performance Analysis of Real-Time Database Systems", *Proc. of 10th IEEE Intl. Conf. on Data Engineering*, February 1994.
18. J. Haritsa, M. Carey and M. Livny, "On Being Optimistic about Real-Time Constraints", *Proc. of 9th ACM Symp. on Principles of Database Systems*, April 1990.
19. J. Haritsa, M. Carey and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems", *Intl. Journal of Real-Time Systems*, 4(3), 1992.
20. J. Haritsa, M. Livny and M. Carey, "Earliest Deadline Scheduling for Real-Time Database Systems", *Proc. of 12th IEEE Real-Time Systems Symp.*, December 1991.
21. M. Kang and I. Moskowitz, "A Pump for Rapid, Reliable, Secure Communication", *Proc. 1st ACM Conf. on Computer and Communications Security*, November 1994.
22. T. Keefe, W. Tsai and J. Srivastava, "Multilevel Secure Database Concurrency Control", *Proc. of 6th IEEE Intl. Conf. on Data Engineering*, February 1990.
23. W. Lampson, "A Note on the Confinement Problem", *Comm. of ACM*, October 1973.
24. L. LaPadula and D. Bell, "Secure Computer Systems: Unified Exposition and Multics Interpretation", *The Mitre Corp.*, March 1976.
25. A. M. Law and C. S. Larney, *Introduction to Simulation Using SIMSCRIPT II.5*, CACI Products company, La Jolla, Calif., 1984.
26. S. J. Leffler, M. K. JcKusick, M. T. Karels and J. S. Quarterman, *The Design and Implementation of 4.3 BSD UNIX Operating System*, Addison-Wesley, 1989.
27. Y. Lin and S. Son, "Concurrency Control in Real-Time Database Systems by Dynamic Adjustment of Serialization Order", *Proc. of 11th IEEE Real-Time Systems Symp.*, December 1990.

28. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, 20(1), 1973.
29. R. Mukkamala and S. Son, "A Secure Concurrency Control Protocol for Real-Time Databases", *Proc. of Annual IFIP WG 11.3 Conference of Database Security*, August 1995.
30. J. Robinson, "Design of Concurrency Control Protocols for Transaction Processing Systems", *PhD Thesis*, Computer Sciences Dept., Carnegie Mellon University, 1982.
31. L. Sha, R. Rajkumar and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *Tech. Rep. CMU-CS-87-181*, Depts. of CS, ECE and Statistics, Carnegie Mellon University, 1987.
32. C. Shannon, "A Mathematical Theory of Communications", *Bell Syst. Tech. J.*, October 1948.
33. S. Son and R. David, "Design and Analysis of a Secure Two-Phase Locking Protocol", *Proc. of Intl. Computer Software and Applications Conf.*, November 1994.
34. S. Son, R. David and B. Thuraisingham, "An Adaptive Policy for Improved Timeliness in Secure Database Systems", *Proc. of Annual IFIP WG 11.3 Conference of Database Security*, August 1995.
35. S. Son, R. David and B. Thuraisingham, "Improving Timeliness in Real-Time Secure Database Systems", *SIGMOD Record*, Special Issue on Real-Time Database Systems, March 1996.
36. S. Son and B. Thuraisingham, "Towards a Multilevel Secure Database Management System for Real-Time Applications", *Proc. of IEEE Workshop on Real-Time Applications*, May 1993.
37. S. Thomas, S. Seshadri and J. Haritsa, "Integrating Standard Transactions in Real-Time Database Systems", *Information Systems*, 21(1), March 1996.
38. B. Thuraisingham and H. Ko, "Concurrency Control in Trusted Database Management Systems: A Survey", *SIGMOD Record*, 22(4), December 1993.

Appendix

A. Master Concurrency Control

We present here the *Global Ordering Scheme* [37] used to ensure a single global serialization order in CC protocols based on the dual-CC approach described in Section 5. In this scheme, the Local CC managers (LCCs) are required to keep track of the local serialization order of transactions and to communicate this to the master concurrency controller (MCC) whenever requested. The LCCs communicate the local serialization order using the notion of serialization numbers which is defined shortly. The MCC then ensures that a single global serial order is maintained for the entire DBMS.

Definition 1. A **serialization number sequence** for a set of transactions, with respect to a schedule S, is a sequence of numbers such that for any pair of transactions T_i and T_j, if m and n indicate the serialization numbers of T_i and T_j respectively, then the following holds: $m < n \Rightarrow \exists$ a serial schedule conflict equivalent to S, in which T_i finishes before T_j.

The local serialization order of transactions at a LCC is communicated to the MCC by augmenting the interface presented by the LCCs to the MCC in the following fashion: Each LCC provides a function called Prepare_to_Commit. This function, when invoked with a transaction identifier, responds with an indication of whether or not the transaction can be committed. If it can be committed, the *serialization number* of the transaction with respect to the global schedule restricted to transactions of that class alone is also returned by the LCC of that class. The serialization numbers returned by a LCC are distinct and are never reused by a LCC, irrespective of the outcome of a transaction.

The MCC invokes the function Prepare_to_Commit when it gets the Commit_Trans call from the Transaction Manager. The returned serialization number is then checked for validity (explained shortly) and the transaction is committed or aborted based on the outcome of this test. We will need a few definitions before outlining the validity test.

Definition 2. Let S be a schedule of a set of transactions T produced by a DBMS employing precisely one CC policy, say M. A **serialization function, ser,** for M is a function that maps every transaction in T to one of its operations such that the following holds: S is conflict equivalent to S', where S' is the serial schedule in which T_i occurs before T_j if and only if $ser(T_i)$ occurs before $ser(T_j)$ in S.

All CC algorithms either inherently have serialization functions, or they can be introduced externally [13]. In particular, for locking protocols and optimistic algorithms, the function that maps every transaction to its commit operation is a serialization function, whereas for timestamp algorithms, the function is associated with the begin operations of transactions.

Definition 3. The **serialization time stamp** (STS) of T_i is the time at which $ser(T_i)$ for the CC policy at MCC is received by the MCC.

The MCC uses the STS to define the serialization order at the MCC while the LCCs communicate their local order through serialization numbers returned by the Prepare_to_Commit call. The MCC now ensures that the serialization orders at the MCC and the LCCs agree by maintaining the following data structures: The MCC maintains a *serialization time stamp* (STS) for each transaction. In addition to STS, the MCC also maintains for each transaction a *low water mark* (LWM) and a *high water mark* (HWM). The LWM and HWM of T_i determines the range in which its serialization number, as returned by the LCC, can lie. When a transaction is initiated, its LWM and HWM are initialized to 0 and ∞ respectively. They are updated as follows whenever a transaction T_i commits:

- For all T_j such that T_j belongs to the same class as T_i and T_j is after T_i in MCC's serial order (i.e., $STS(T_i) < STS(T_j)$) and T_j conflicts with T_i, set $LWM(T_j) = \max$ ($LWM(T_j)$, serialization number of T_i).
- For all T_j such that T_j belongs to the same class as T_i and T_j is prior (in MCC's serial order) to T_i, (i.e., $STS(T_i) > STS(T_j)$) set $HWM(T_j) = \min$ ($HWM(T_j)$, serialization number of T_i).

We now describe the test used for checking the validity of the serialization number returned in response to a Prepare_to_Commit call. Based on the result of the test, the transaction is committed or aborted.

Commit Test : The MCC allows a transaction to be committed only if its serialization number, as returned by the LCC, lies between its LWM and HWM.

We will now prove that the commit test ensures that a single global order is maintained.

THEOREM 1 *If there exists an edge $T_i \rightarrow T_j$ in the precedence graph corresponding to a schedule produced by any combination of CC algorithms at MCC and the slave LCCs, then $STS(T_i) < STS(T_j)$.*

Proof: We will assume $STS(T_i) > STS(T_j)$ and derive a contradiction.

Case 1 : T_i and T_j belong to the same class.
Case 1.1 : T_i commits before T_j
Since $STS(T_i) > STS(T_j)$, T_j is prior (in MCC's serial order) T_i and hence $HWM(T_j)$ is set to some value which is smaller than or equal to the serialization number of T_i when T_i commits. Since $T_i \rightarrow T_j$, the serialization number of T_j returned by the LCC will be greater than that of T_i. So it exceeds $HWM(T_j)$ and hence the MCC will not allow it to commit.
Case 1.2 : T_j commits before T_i
We know that $LWM(T_i)$ will be set to some value which is greater than or equal to the serialization number of T_j when T_j commits because by our assumption T_i

is after (in MCC's serial order) T_j and T_i conflicts with T_j. But $T_i \rightarrow T_j$ and hence the serialization number of T_i returned by the LCC is smaller than that of T_j. It is less than $LWM(T_i)$ and MCC will not allow it to commit.

Case 2 : T_i and T_j belong to different classes.

By the definition of STS, if $T_i \rightarrow T_j$, then $STS(T_i) < STS(T_j)$ because inter-class conflicts are regulated by the CC algorithm at the MCC, which will produce serializable schedules only.

■

B. Security Loopholes in the S2PL Protocol

We describe here the Secure 2PL (S2PL) algorithm proposed in [33, 10], and the security loopholes associated with this algorithm.

B.1. The S2PL Protocol Description

We have reproduced below their protocol exactly as it was originally proposed except that, for clarity, we have modified the description at some places. The following notation is used in the description: $queue[x]$ is a queue of operations, each of which could be holding a virtual lock $VpLock$, a dependent virtual lock $DVpLock$, a real lock $pLock$, or waiting to set a lock $pWait$. An operation p could be a read r or a write w and an action is represented in the form $p_i[x]$. The $queue$ is not a strict FIFO queue, since elements can be inserted in the middle, but it can be considered a priority queue ordered by the $before$ relation between transactions. $SL(T_i)$ is the security access class of the transaction T_i.

When a request for lock p is submitted to the S2PL CC protocol, action is taken as per the following algorithm:

$Case1$ $(p = read) \wedge \exists(wLock(T_i, x) \vee VwLock(T_i, x) \vee DVwLock(T_i, x))$
    Read value of x *written by* $w_i[x]$;

$Case2$ $((p = write) \wedge \exists rLock(T_i, x))$
    Upgrade $rLock(T_i, x)$ *to* $wLock(T_i, x)$;

$Case3$ $\exists j((qLock(T_j, x) \vee VqLock(T_j, x) \vee DVqLock(T_j, x)$
        $\vee Wait(T_j, x)) \wedge (j \neq i) \wedge ((p = write) \vee (q = write)))$
    $pos := length(queue[x])$;
    while $(pos > 0)$
        $T_j \longleftarrow$ *transaction at* $queue[x].pos$;

```
if (operation (queue[x].pos) conflicts with p)
        if (T_i ∈ before(T_j))
                ∀T_3((T_3 ∈ before(T_j)) ∧ (T_3 ∈ after(T_i))
                    ∧ (SL(T_3) > SL(T_i)) ∧ (SL(T_3) > SL(T_j)))
                    abort T_3; /* value security violation */
                if (lock(queue[x].pos) is a real lock) /* deadlock */
                    call victim selection routine;
                    if (victim = T_i) exit; /* exit from routine */
                    else continue; /* to start of while loop */
        else /* if (T_i ∉ before(T_j)) */
                if (SL(T_i) = SL(T_j))
                    if (T_j has not committed)
                                Insert(queue, pWait(T_i, x), pos);
                    else
                                if (p = read) read w_j[x];
                                Insert(queue[x], pLock(T_i, x), pos);
                elseif ((p = write) ∧ (∃z, VwLock(T_i, z)))
                    Insert(queue, DVwLock(T_i, x), pos);
                                        /* Insert after pos */
                    dependent(w_i[z]) := w_i[x];
                else if (SL(T_i) < SL(T_j))
                    Insert(queue, VwLock(T_i, x), pos);
                else
                    Insert(queue, pWait(T_i, x), pos);
                before(T_i) := before(T_i) ∪ T_j ∪ before(T_j);
                after(T_j) := after(T_j) ∪ T_i ∪ after(T_i);
                exit; /* exit from routine */
        else
                pos := pos − 1;
endwhile /* end of while loop */
if (pos = 0) Insert(queue[x], pLock(T_i, x), pos);
```

In [10], a proof is given to show that the above algorithm satisfies all the conditions for a secure interference-free scheduler [22], namely, Value Security, Delay Security and Recovery Security. These conditions are described below:

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.

Delay Security: A system is Delay Secure if the delay experienced by a subject is not affected by actions of subjects having 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.

In the remainder of this section, we analyze the S2PL protocol and point out the circumstances under which it *fails* to ensure complete security. For ease of understanding, we use transaction execution schedules. A schedule is a temporal log of the operations performed by the system for the transactions concurrently executing in the system. A transaction with identifier i and security clearance level CL is represented in the schedule by $T_i(CL)$. Corresponding to every transaction appearing in the schedule, there is a list of actions of the form $p_i[x]$ where p is an action the transaction T_i performs on the data item x. An action p can appear in the schedule in two forms, r or w, which stands for read and write, respectively. For example, the read performed on a data item x by transaction T_i is represented as $r_i[x]$. Similarly, a write performed on the data item x by transaction T_i is represented as $w_i[x]$. The actions in the schedule occur in left to right order. If some actions appear in the same vertical column, it means they can occur simultaneously (i.e., there is no ordering defined among those operations).

B.2. Recovery Security Loophole

An operational sequence by which value security in S2PL could be violated and the method to prevent such violations is explained in [10], and is reproduced below:.

Transactions	Activity					
$T_1(Secret)$	$r_1[y]$	$r_1[x]$	.....			c1
$T_2(Confidential)$					$r_2[z]$	c2
$T_3(Classified)$	$w_3[x]$			$w_3[z]$	c3	
$T_4(Public)$		$w_4[y]$	$w_4[z]$		c4	

In the above schedule, value security is violated as follows: The Secret transaction T_1 secures a read lock on data item y. Subsequently, the Public transaction T_4 submits a write request on y. Since the T_1 already holds a lock on y, T_1 is set *before* T_4 in the serialization order. Similarly T_3 is set *before* T_1 in the serialization order because T_3 is already holding a write lock on data item x when T_1 requests x. When T_3 requests a write lock on data item z T_4 already has a dependent virtual lock on it. Since T_1 is *before* T_4 and T_3 is *before* T_1, the transaction T_3 is allowed to overtake T_4 in $queue[z]$. The value of z read by the Confidential transaction T_2 will be the value written by T_4. Had T_1 not been present in the system, T_2 would have read the value written by T_3. Since it is possible for a higher security level transaction to alter the value read by a lower security level transaction, if the above schedule is allowed to execute, value security may be violated.

The way in which S2PL deals with the above kind of value security violations is to abort **all** transactions *before* T_4 and *after* T_3 whose security level is higher than T_3 and T_4. So, in the above example T_1 is aborted and the value security is maintained. However, this scheme of aborts can itself lead to *recovery security* violations, as described below.

Transactions	Activity								
$T_0(TopSecret)$	$r_0[a]$	$r_0[c]$	..						c0
$T_1(Secret)$		$r_1[b]$		$w_1[a]$				$r_2[d]$	c1
$T_2(Confidential)$								$r_2[d]$	c2
$T_3(Classified)$	$w_3[c]$						$w_3[d]$	c3	
$T_4(Public)$				$w_4[b]$	$w_4[d]$	c4			

Consider the above schedule. Here when T_4 requests for a write lock on b, T_1 already holds a read lock on the same item. Therefore, T_1 is set to *before* T_4. When T_1 requests a write lock on data item a, T_0 is already holding a read lock on it, so T_0 is set to *before* T_1 in the serialization order. When T_0 requests for a lock on data item c, T_3 is already holding a lock on it and therefore, T_3 is set *before* T_0 in the serialization order. Later T_4 performs a dependent virtual write on d and commits. Now when T_3 requests a write lock on d, T_3 is allowed to supersede T_4 since the former comes earlier in the serialization order. This would not have happened if either T_0 were absent or T_1 were absent. According to the S2PL protocol, T_0 and T_1 are both aborted to ensure value security. But then this opens another covert channel as follows:

Imagine the case where T_0 wants to send some information to T_1, it could collaborate with T_1, T_2, T_3, and T_4 and create the above schedule. Faced with this schedule, S2PL promptly detects value security violation and aborts *both* T_1 and T_0. Since T_1 is also aborted along with T_0, it is a security violation since had T_0 been absent, T_1 would not have aborted.

B.3. Delay Security Loophole

An operational sequence by which delay security in S2PL could be violated is presented below:

Transactions	Activity				
$T_1(Secret)$	$r_1[a]$				$r_1[c]$
$T_2(Confidential)$		$w_2[a]$		$w_2[b]$	
$T_3(Classified)$			$w_3[b]$		

In the above schedule, T_1 gets a read lock on data item a and later T_2 requests for a write lock on a. Since the Secret transaction T_1 already holds a read lock on data item a, a virtual lock (non-blocking) is granted for T_2. Now the Classified transaction T_3 obtains a write lock on data item b. Later, when T_2 requests a lock on data item b, since it already has secured a virtual lock (for data item a), it is granted a dependent virtual lock (again, non-blocking). Suppose T_1 was not present in the system, $w_2[b]$ of transaction T_2 would have blocked until T_3 releases its lock on data item b. Because of the presence of T_1, T_2 performs $w_2[b]$ quicker than otherwise. This is a delay security violation since T_2 completes *earlier* due to the presence of a higher security transaction.

B.4. Non-serial Schedules

Apart from the above security-related problems, S2PL may also result in *non-serial* schedules, as explained below:

Transactions	Activity											
$T_1(Secret)$						$r_1[c]$						
$T_2(Confidential)$				$r_2[b]$				$w_2[c]$				$w_2[d]$
$T_3(Classified)$	$r_3[a]$				$w_3[b]$							
$T_4(Public)$			$w_4[a]$							$w_4[d]$		

In the above schedule, when T_4 requests a write lock on the data item a, T_3 has already obtained a read lock on a, so T_3 is set *before* T_4 in the serialization order and T_3 is added to the *beforelist* of T_4. Also, T_4 is added to the *afterlist* of T_3. For similar reasons, T_3 is added to the *afterlist* of T_2 and T_2 is added to the *beforelist* of T_3. Then T_2 obtains a dependent virtual lock on data item d after which T_4 requests for a write lock on d. Now, when T_4 makes the request for d, T_2 is **not** in the *beforelist* of T_4 and therefore it is set after T_2. This is incorrect because T_2 is before T_4 in the serialization order (in fact, the *afterlist* of T_2 contains T_4). Therefore, there could be a serializability violation in the S2PL protocol.

B.5. Fixes for the S2PL Loopholes

To address the above problems, we have implemented the following fixes in our implementation of the S2PL protocol:

1. When the conditions for a potential value security violation are satisfied because of the actions of a set of higher security level transactions, the highest security level transaction in the set is aborted. This step is repeated until the conditions for value security violation are completely removed (or the set becomes empty).

2. A dependent virtual write operation on a data item is made to wait if a conflicting (real or virtual) lock is held on the data item by another transaction of the same or lower security level.

3. Instead of every transaction having its own beforelist and afterlist, a common serialization graph is maintained for determining the serialization order.