A Framework for Database Audit and Control Flow Checking for a Wireless Telephone Network Controller

S. Bagchi, Y. Liu, Z. Kalbarczyk, R. Iyer
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL 61801
E-mail: [bagchi, yliu, kalbar, iyer]@crhc.uiuc.edu

Y. Levendel, L. Votta
Corporate Software Technology Center
Motorola Inc.
1303 East Algonquin Rd., Schaumburg, IL 60196
E-mail: I.Levendel@motorola.com,
votta@cig.mot.com

Abstract

The paper presents the design and implementation of a dependability framework for the call-processing environment in a digital mobile telephone network controller. The target environment includes a database subsystem containing configuration parameters and resource usage status, and call-processing clients for managing individual calls. This environment presents some unique challenges, since a malfunctioning client can write erroneous data to the database, and errors in the database can impact the call-processing clients. The paper describes an integrated dependability framework for the controller. The framework contains a data audit subsystem to maintain the structural and semantic integrity of the database and a preemptive control flow checking technique, PECOS, to protect the clients. Evaluation of the dependability-enhanced system by software-implemented error injection is performed. The evaluation shows that for control flow errors in the client, the combination of PECOS and data audit eliminates fail-silence violations, reduces the incidence of client crashes from 52% to 19%, and eliminates client hangs. For database injections, data audit detects 85% of the errors and reduces the incidence of escaped errors from 63% to 13%. Evaluation of combined use of data and control checking (while error injection targets the database and the client) shows coverage of 80% and indicates data flow errors as a key reason for error escapes.

Index terms: dependability framework, data audit, control flow checking, fault injection, wireless call-processing environment.

1 Introduction

Application availability can be compromised because of errors that affect application execution flow or corrupt data used during the execution. In this paper, we present the dependability design and implementation of a dependable framework for the call-processing environment of a digital mobile telephone network controller whose availability is dependant on both control and data integrity. The call-processing environment includes a database subsystem containing configuration parameters and resource usage status and call-processing clients for setting up, managing, and tearing down individual calls. Both subsystems are vulnerable to errors, which can manifest as system outages affecting many customers. Consequently, we need efficient mechanisms to provide detection and recovery from both control and data errors.

Towards this we investigate and evaluate the efficiency (in terms of coverage) of data audits and control flow checking in protecting, respectively, the database and the call-processing clients of the mobile network controller. The controller is an example of a data-driven embedded system running on a robust hardware platform typical of call-processing environments, e.g., employing hardware redundancy and providing fail-over capabilities.

In this configuration, the database subsystem is an important software component. The database contains both static system configuration parameters and dynamic system state information and provides basic services such as read, write, and search operations to the system and application processes. As the data stored in the database is subject to corruption due to a variety of hardware and software errors, database audit plays an important role in maintaining data integrity to effectively support overall system function. Another important source of system misbehavior is an error that affects the application control flow, potentially leading to the corruption of the database, application crash, or fail-silence violations[1]. Such control flow errors have been demonstrated to account for between 33% [OHL92, BAG00] and 77% [SCH87] of all errors depending upon application characteristics and the error model. Preemptive control flow checking can achieve high detection coverage in capturing control flow errors (i.e., errors that cause a divergence from the sequence of program counter values seen during the error-free execution) of the application before the application terminates abnormally. In this paper, we propose a control-flow signature generation and preemptive checking technique called *PECOS* (PreEmptive COntrol Signatures) and apply it to detect errors in control flow of the call-processing application threads.

The integrated call-processing environment, with both data and client protection, is implemented using the common adaptive framework, which allows extensible database audits combined with recovery. Specifically, we provide an API and an architecture to encapsulate database error detection and recovery.

---

[1] A fail-silent application process either works correctly, or stops functioning (i.e., becomes silent) if an internal failure occurs [BRA96]. A violation of this premise is termed a fail-silence violation. In distributed applications, fail-silence violations can have potentially catastrophic effects by causing fault propagation.

The main contributions of the paper can be summarized as follows:

1. Design and implementation of a generic, extendable framework for providing data audits and control flow checking to applications. In this framework, new detection and recovery techniques can be integrated into the system with minimum or no changes to the application.
2. An experimental evaluation of the proposed framework being used to provide integrated control and data error detection and recovery in a real wireless call-processing environment. The software-based, error-injection evaluation quantifies:
    - the combined coverage provided by data audit and control flow checking in protecting the call-processing environment and
    - the chances of error propagation between the database and the client, and vice-versa.

The evaluation of the environment with both control and data protection shows the following: (1) for corruptions to the database subsystem (a) data audit detects 85% of the errors and (b) the incidence of escaped errors are reduced from 63% to 13%; (2) for errors injected randomly to the call-processing clients (a) in the absence of any detection in the client, 8% of injected errors propagate to the database, (b) there are no (or negligible) fail-silence violations and no client hangs, and (c) the incidence of client process crash are reduced from 52% to 19%.

2 Related Work

Data Audits. In the telecommunications industry, the term *data audit* typically refers to a broad range of somewhat custom and *ad hoc* techniques for detecting and recovering from errors in a switching environment. Data-specific techniques deeply embedded in the application are generally believed to provide significant improvement in availability, although little or no actual assessment of these techniques has been available until the present publication.

Application-layer self-checking maintenance software was successfully deployed to achieve high availability in systems such as the AT&T 5ESS® switch [HAU85]. The techniques employed by its fault-tolerant software include in-line defensive checks, data audits, process activity and resource checks, and modular, hierarchical error recovery. In-line defensive checks compare data relationships and check for specific data values at various points to detect software faults as early as possible. Data audits are designed to detect errors in data and data structures. Process and resource checks ensure that a proper distribution of system resources is provided among system software. The hierarchical error recovery strategy aims to restore system operation by making localized repairs whenever possible and escalate to more global actions only if necessary.

Data audit is an application-layer solution for detecting and correcting errors in system data. A related approach is robust data structures, which aim to improve software fault tolerance by detecting and correcting errors in stored data structures that contain carefully deployed redundancy. Taylor [BLA80, TAY80(a)(b),

TAY85] presents some examples of robust storage structures and their practical implementation, as well as the underlying theory and some empirical results on the cost-effectiveness. The concept of data structure robustness can be applied to the data organization in data-centric systems to support detection of corruptions in the database structure.

Commercial off-the-shelf database systems from Oracle [ORACLE], Sybase [SYBASE], and more recently, the in-memory database from TimesTen [TIMESTEN], all include utilities to perform consistency checks of database integrity. These utility programs (e.g., OdBit from Oracle, and DBCC from Sybase) are typically invoked on the command line interface by database administrators to perform physical and logical data structure integrity checks. For example, in relational databases such as Oracle the core database engine supports a set of rules for identifying relations/dependencies between tables or records in the database. These rules can be used for detecting structural and semantic errors in the database by performing referential integrity checking or structural checking [COS00]. However, the lack of a fault-tolerant infrastructure that ties the database error detection and recovery elements together is a major limitation in the existing systems, especially when continuous availability and integrity of the database are required. While in our work we leverage the existing experience, we also contribute by:

- Improving the efficiency of the proposed detection techniques by (1) using new triggering for initiating audit, e.g., prioritizing audit where on-line system monitoring is used to direct error checking to heavily used database regions or database sections with high error rate, and (2) adding redundancy (without modifying the original database structure) to facilitate better diagnosis, e.g., identify the misbehaving database client, and protection against resource leaks, i.e., detection of orphan records and removing them from the database.
- Combining the error detection and the following recovery actions so the two are transparent to the database clients.
- Providing a unified, flexible framework for creating new audit techniques and invoking them to check the database system with little impact on the database structure and minimum interference with database clients.

It should be noted that some of the recovery actions supported by the audit subsystem depend on domain-specific knowledge about the application using the database, in this case the call-processing clients. For example, freeing a record as part of recovery is considered tolerable because, in our environment, it implies dropping one active call. For a generic database, e.g., Oracle, such recovery action may not be possible in the absence of domain knowledge about the application using the database. In this study, the data audit subsystem has been applied to a fairly large database in a commercial environment, and its effectiveness substantiated through a detailed evaluation.

Control flow monitoring. The field of control-flow checking has evolved over a period of about 20 years. The first paper, by Yau and Chen [YAU80], outlined the general control flow checking scheme using the program specification language PDL.

Hardware Watchdog Schemes. Mahmood [MAH88] presents a survey of the various techniques in hardware for detecting control flow errors. Many schemes for checking control flow in hardware have been proposed [NAM82, SCH86, MAD91, WIL90, MIC91, UPA94, MIR95]. The basic scheme is to divide the application program into blocks. Each block has a single entry and a single exit point. A golden (or reference) signature is associated with the block that represents an encoding of the correct execution sequence of instructions in the block. The selected signature is then calculated by a watchdog processor at runtime. The watchdog validates the application program at the end of a block by comparing the runtime and the golden signatures of the block. Among the hardware schemes, two broad classes of techniques have been defined that access the pre-computed signature in two different ways. Embedded Signature Monitoring (ESM) embeds the signature in the application program itself [WIL90, SCH86], while Autonomous Signature Monitoring (ASM) stores the signature in memory dedicated to the watchdog processor [MIC91]. Upadhyaya *et al.* [UPA94] explore an approach in which no reference signatures need be stored, and the runtime signature is any *m-out-of-n* code word.

Software Schemes. The software techniques partition the application into blocks, either in the assembly language or in the high-level language, and insert appropriate instrumentation at the beginning and/or end of the blocks. The checking code is inserted in the instruction stream, eliminating the need for a hardware watchdog processor. Representative software-based control flow monitoring schemes are Block Signature Self Checking (BSSC) [MIR92], Control Checking with Assertions (CCA) [KAN96], and Enhanced Control Checking with Assertions (ECCA) [ALK99].

The key problems with existing hardware and software control flow checking techniques are (1) none of the schemes are preemptive in nature and detect an erroneous control flow after executing instructions from the incorrect path. Consequently the system often crashes before any checking is triggered; (2) the existing schemes cannot handle control flow structure determined at runtime, e.g., calls to dynamic libraries or accessing functions through virtual function tables. The identified problems with existing techniques served as a reference point in designing PECOS.

3 Overview of the Target System Software and Database Architecture

The target system is a small-scale digital wireless telephone network controller that integrates many functions in standard wireless telephone network components (including call switching, packet routing, and mobility management). This section gives an overview of the key software components of the controller.

3.1 Software Components

The key components of application software running on the controller are call processing (sets and terminates clients calls) and database (supports information about system resources)[2].

3.1.1 Call-Processing Client

Call processing is the key component that provides customer-visible functionality. This process dynamically creates a call-processing thread to handle activities associated with a voice/data connection. Specifically, the thread is responsible for subscriber authentication, hardware resource allocation and deallocation, mobility management, and supporting basic and supplementary telephony features. The failure of the call-processing task alone could render the entire system unavailable from a user's point of view. It is therefore necessary to monitor its status during system operation. Since call processing requires database access, we refer to the application as a database client.

3.1.2 Database Subsystem

The database subsystem is a critical component as it contains data (static system configuration data and runtime data that indicate resource usage and process activities) necessary to support the operation of application processes, including call processing. As most of the controller operations require database access, any data error or data structure corruption in the database could have a major impact on system performance and availability.

Organization. To satisfy the real time constraints, the entire database is loaded from disk into memory at startup time and resides completely in memory during system operation. The memory region that contains the database is contiguous and is shared among all processes that require database access. To remove the possibility of memory leak and ensure continuous system operation, the space for all tables in the database is pre-allocated, and no dynamic memory allocation is used in the database. Therefore, during normal operation the size of the memory-based database stays constant.

The database consists of various tables with a pre-defined size that occupy the memory space one after another. Although the entire space is statically allocated, some tables are dynamic in nature while others are static. Static data in tables usually refer to system configuration (e.g., the number of CPUs in the system) and stay constant during operation, whereas dynamic data is often updated, e.g., on every incoming call. Note that each table usually contains a mixture of static and dynamic data.

The database subsystem exports an API for other processes to access the database. Each API primitive performs the requested operation such as write a record or move a record. Some API functions along with brief explanations are shown in Table 1.

---

[2] Other application components include system maintenance, network management, and system monitor.

DBinit	Initialize client connection to the database
DBclose	Close client connection to the database
DBread_rec	Read a record (row) in a table
DBread_fld	Read a field in a table record
DBwrite_rec	Write a record (row) in a table
DBwrite_fld	Write a field in a table record
DBmove	Move a record to another logical group

Table 1: Examples of Database API

3.2 Errors in Database

The database is subject to corruption from a number of sources: human (operator) error, software bugs or faults, and hardware faults. Static system configuration information is typically entered by operators through a user interface. Human error, such as misunderstanding of procedure or interface functions, could result in inconsistent data or data structures. Software bugs uncaught during testing phase could also cause program to write incorrect data or write into incorrect memory locations. Run-time software failure is another cause for data errors. For example, if a process that is in the middle of updating a set of database records crashes, the set of data would likely be in an inconsistent state. Memory hardware or environmental factors can also trigger memory mutilation, although basic hardware mechanisms such as ECC mask some of these errors.

The effects from these types of error appear in several ways. The most serious consequence occurs if errors are introduced into the system catalog (metadata) of the database. The system catalog contains information used by the database API to access records and consists of several database tables that are referenced on each database operation. Errors in the system catalog can cause all database operations to fail, thus bringing down the whole controller. Reduced system availability is another effect from data error. If any data indicating resource status is corrupted, the particular resource in question could falsely appear to be busy to the rest of the controller ("resource leak"), leading to a reduced system capacity. If this type of error is allowed to accumulate, system availability will be reduced. Other errors may have a local effect only. For example, a damaged call record related to a particular connection will likely bring down that connection prematurely without affecting other active connections. It is also possible for errors to have no effect on the system if the errors occur at memory locations that are not used or if they are overwritten.

To improve system availability, it is important not only to prevent errors from occurring, but also to detect and recover from errors quickly. Simplification and standardization help reduce operator and programmer errors, but the possibility of errors due to runtime software or hardware anomaly remains. While exact figures are not publicly available, data in telecommunication industry has repeatedly shown that not performing database audits significantly reduces telephone switching system availability [LEV99].

4 Audit Subsystem Architecture

The overall design of the database audit process and its interactions with other system components are shown in Figure 1. The right half of the figure shows the database, a client process, and the database API (*DB*

API) that is used by client processes to access database. A communication channel (the *message queue*) is added between the database API and the audit process to transmit events from client activities. The audit process consists of a dedicated thread (the main thread) acting as the interface to other components. The function of the main thread is to translate information from external entities (e.g., the database client) into audit messages via the *audit framework API*. The main thread allows other processes to communicate with the audit process using standard inter-process communication mechanisms. The proposed audit framework provides audit functionality and consists of the top-layer shell (the *audit interface*) and the individual elements that implement specific audit triggering, error detection and recovery techniques. The elements depicted in Figure 1 include heartbeat (*HB*), progress indicator (*Prog. Ind.*), audit (*Audit Elem.*) which encapsulates a set of error detection and recovery techniques, periodic audit (*Per. audit*) and event triggered audit (*EvTrig audit*) which support, respectively, periodic and event triggered invocation of different audit techniques (provided by the audit element).

To reduce contention with database clients, the audit elements access the database directly instead of through the database API. Bypassing the locking and access control mechanisms managed by the API reduces performance penalty, but it requires the audit process to detect database access conflicts between clients and itself to ensure the validity of audit results.

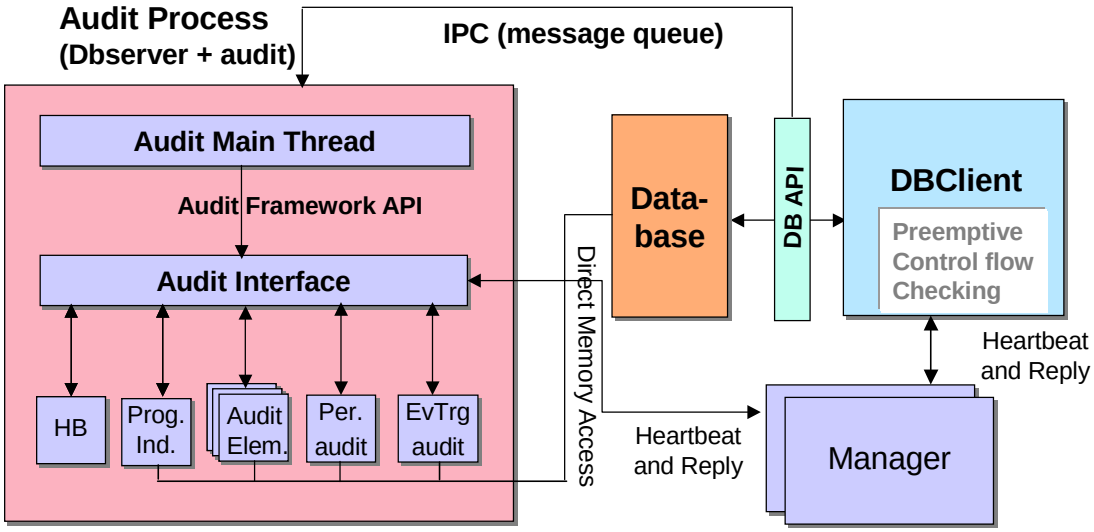

Figure 1: Target System with Embedded Audit and Control Flow Checking

The manager (running in a redundant configuration, e.g., duplication) is responsible for overseeing the overall state of the environment. The manager starts the audit process and monitors it by heartbeats as shown in Figure 1. If the audit process fails, the manager restarts it on the same or another node.

The proposed framework provides high modularity and transparency allowing for easy extensibility of the audit subsystem. New error detection and recovery techniques can be implemented, encapsulated in new elements, and added to the system. A new element to be incorporated into the system needs to define and

communicate to the audit main thread a set of messages that it (the element) accepts (i.e., is capable of processing). From that point on, the audit main thread and the audit interface will handle the proper communication. The different audit elements can be quite independent of each other, which allows for easy customizability of the audit subsystem.

4.1 The Heartbeat Element

The heartbeat element and the manager together implement a heartbeat protocol that allows the manger to detect audit process failures. Periodically, the manager process sends a heartbeat message to the heartbeat element in the audit process and waits for a reply. If the entire audit process has crashed or hung, or if there is scheduling anomaly on the controller system that prevents the audit process from running, the manger times out and restarts the audit process.

4.2 The Progress Indicator Element

The progress indicator element is used to detect deadlock in the controller database to ensure uninterrupted system operation. The database API maintains and manipulates locks transparently to the client processes to synchronize their access to the database. If a client process terminates prematurely without fully committing its transaction, the locks left behind by this process would prevent other client processes from accessing the database (or portions of it). The database API is a passive entity and is not capable of detecting and resolving deadlocks, so it is important to have deadlock detection as part of the audit process.

Detection. A standard POSIX IPC message queue is added between the database API and the audit process (see Figure 1). The database API is modified to send a message to the audit process whenever any API function is called. The message contains the client process ID information and the database location being accessed. These messages are used to increment a counter in the progress indicator element as they indicate ongoing database activity. If the audit process receives no message, leaving the counter value unchanged for an extended period of time, the progress indicator element will time out and trigger recovery.

Recovery. The progress indicator element terminates the client process holding the lock for greater than a predetermined threshold duration, thereby releasing the lock. While the threshold for a client to hold a lock is typically small (e.g., 100 milliseconds in the current implementation), the progress indicator timeout value is much larger (e.g., 100 seconds in the current implementation) in order to reduce runtime overhead.

4.3 The Audit Elements

Specific audit techniques are implemented as separate audit elements (audit element in Figure 1) in the audit process. The invocation of the audit elements can be either by a periodic trigger, or by an event trigger. The periodic trigger is based on a fixed time period. The event trigger is provided by some specific database operations, e.g., database write in the current implementation. As mentioned above, the database API is modified to send a message to the audit process after each database update. The periodic audit element uses as

its basis the periodic heartbeat query discussed earlier as trigger to perform the following specific audits: static data integrity check, dynamic data range check, structural audit, and referential integrity audit. The audit subsystem can be configured to use different events to trigger the audit elements, e.g., when the system enters a critically low available resource state. If there is an intervening update to a record being accessed by an audit element while the audit is in progress, the result of the audit is invalidated and the procedure is run later.

4.3.1 Static and Dynamic Data Check

Detection and recovery. These checks are similar to those used in commercial databases except for optimization specific to mobile controller databases. System configuration parameters and some database metadata (e.g., table ID/size/offset) are static during runtime. The audit element detects corruption in static data region by computing a golden checksum of all static data at startup and comparing it with a periodically computed checksum (32-bit Cyclic Redundancy Code). The standard recovery for static data corruption is to reload the affected portion from permanent storage.

To make audit on dynamic data possible in the target database, the range of allowable values for database fields are stored in the database system catalog. This information allows the audit program to do a range check on the dynamic fields in the database. Clearly, this technique cannot detect all corruptions in dynamic data, as the exact ranges are unknown to the audit program. The semantic audit, discussed later detects data errors with higher certainty. If the audit detects an error, the field is reset to its default value, which is also specified in the system catalog. In addition, if the table where the error occurred is dynamic, the record is freed as a preemptive measure to stop error propagation.

4.3.2 Structural Check

The structure of the database in the controller system is established by header fields that precede the data portion in every record of each table. These header fields contain record identifiers and indexes of logically adjacent records.

Detection. The structural audit element calculates the offset of each record header from the beginning of the database based on record sizes stored in system tables (all record sizes are fixed and known). The database structure (in particular, the alignment of each record and table within the database), is checked by comparing all header fields at computed offsets with expected values.

Recovery. A single error in record identifier is correctable because the correct record ID can be inferred from the offset within the database. However, multiple consecutive corruptions in header fields is considered to

be a strong indication that tables or records within the database may be misaligned, and the entire database is then reloaded from the disk to recover from the structural damage[3].

4.3.3 Semantic Referential Integrity Check

To verify data semantic, the audit process performs semantic referential integrity checking which traces logical relationships among records in different tables. *Referential integrity* requires that the corresponding primary and foreign key attributes in related records match. Corruption of key attributes leads to "lost" records, as some records participating in semantic relationships "disappear" without being properly updated. We refer to this phenomenon as *resource leak*, which is similar to memory leak found in many software systems. Since database records are limited resource themselves, they must be freed properly from semantic relationships to ensure their continuous availability.

Detection. Tables are considered as sets, and records as elements of these sets, then the semantic constraints of the application can usually be expressed as 1-to-1 or 1-to-N correspondences among these sets. These correspondences logically form a chain among each set of related records, or they may be turned into a closed loop, thereby making it 1-detectable, by connecting the head and tail with an extra correspondence. Referential integrity audit follows semantic linkages among related records in different tables to verify the consistency of these logical loops and detect invalid data impossible to find when records are examined independent of each other. Commercial databases provide some support for referential integrity checking. However, since no timestamps or process IDs for the last access of a record are maintained, automatic recovery action cannot be taken. Also, since the above recovery action requires specific knowledge of the application semantics, it is not reasonable in a generic database package.

As an example, consider the data structure established when servicing a voice connection. A process (or thread) must be spawned to manage the connection; the process needs to allocate hardware resources and record information related to the call (e.g., the IDs of the parties involved) into the connection table. Thus, a new record needs to be written into each of the following three tables:

Process Table (Process ID, Name, Connection ID, Status, …),
Connection Table (Connection ID, Channel ID, Caller ID, …),
Resource Table (Channel ID, Process ID, Status …)[4]

Clearly, the three new records form a semantic loop because the process record refers to the connection record via the "Connection ID" attribute, the connection record refers to the resource record via the "Channel ID" field, and the resource table closes the loop by pointing back to the process record via "Process ID." Thus,

---

[3] The use of doubly linked list as the data structure for logical groups within the database can allow single pointer corruption to be detected and corrected using robust data structure techniques (e.g., traversing the list of table records in both directions and making proper pointer adjustments) [SET85]. These techniques are not implemented, as they require changing the database structure (creating a doubly link list) and impose unacceptable database downtime from a client perspective (locking of the linked lists).

[4] Underlined attributes are the *keys* of their respective table.

the audit program can follow these dependency loops for each active record in each of the three tables and detect violations of semantic constraints.

Recovery. The recovery actions include *freeing "zombie" records* and *preemptively terminating the processes/threads* that are using these records. Preemptive process termination is desirable because it keeps system resources available even though an active connection may be dropped. The termination is made possible by modifying the database API to maintain, along with each database record, the ID of the client process that last accessed the record. The redundant data structure associated with the database records also includes the time of last access and counters that maintain database access frequencies.

4.4 Optimization Using Runtime Statistics

The audit techniques presented in the previous section all use rules determined off-line, such as the lower/upper bounds of a dynamic field, and the order in which database tables are audited. While these static rules allow the audit process to detect many errors, they do not take the actual runtime system behavior into account and therefore are not adaptive. This section presents two audit optimization strategies that utilize statistics collected during system operation: (a) prioritized audit triggering and (b) selective monitoring of attributes.

4.4.1 Prioritized Audit Triggering

The goal of prioritizing the audits is to detect errors without requiring more time or system resources. It is based on the assumption that database objects have different importance and that their importance varies during system operation. In this work the following criteria are used to determine the importance of database objects:

The access frequency of database tables. The tables that contain data that are more frequently updated are more liable to be corrupted due to software misbehavior and are more likely to cause error propagation to processes that use the data. Hence, the tables are arranged based on their access frequencies and the ones with higher access frequency are checked more often.

The nature of the database object. For example, the database system catalog is the most important because it is referenced on every database access and its corruption makes the entire database useless.

The number of errors detected in each table. This is based on the assumption of temporal locality of data errors; i.e., the area where more errors occurred in the recent past is likely to contain more errors in the near future. The audit process can make better utilization of system resources by focusing on the trouble spots to achieve higher error detection ratio and lower detection latency.

The periodic audit element discussed earlier checks all database tables in a predetermined order every time, regardless of how frequently each table is referenced or how the detected data errors are distributed. To support prioritized audits, information on access frequency and error history are collected at runtime by modifying the database read/write API. In addition, the audit process maintains an error log for each table, which contains the

number of errors detected in the last audit cycle. These pieces of information are combined to derive a weighted measure of importance that is used to rank all the database tables and to direct the audit to more critical sections of the database.

4.4.2 Selective Monitoring of Attributes

The selective monitoring of certain table attributes (fields) is motivated by the lack of good static audit rules in some cases. For example, although the database catalog provides the facility to record the lower and upper bounds of each table attribute, not all ranges are specified. This is due to the difficulty in characterizing certain attributes whose values cannot be predicted with relative certainty in advance. By monitoring the values of such attributes at runtime, adaptive rules can be generated for use by the audit process to detect data errors.

A related research area is using dynamic techniques to discover program invariants, as presented in [ERN99]. The idea is to infer invariant variables in the program during test runs and compare them against values captured at runtime. Examples of the possible invariants to be tested include range limits over a single numeric variable, linear relationship among two or three numeric variables, and functions among multiple variables.

In the selective monitoring of database attributes, a slightly different approach is taken. Instead of trying to verify a set of predetermined hypotheses, the data trace collected at runtime is used to derive possible invariants to be used by the audit process for error detection. These possible invariants are not completely verified, as they are inferred from only the data traces collected so far. Any abnormality detected with these derived invariants needs to be further checked by other means.

To derive the set of correct values of an attribute in the database, the audit program periodically examines the values of that attribute in all active records of the relevant table. An average number of occurrences is computed across all attribute values. A threshold value for number of occurrences is then computed using a certain fraction of the average. Any value that has appeared less frequently than the threshold is marked as suspect, and further actions, such as semantic audit, are triggered to make a final decision.

5 Evaluation of the Audit Subsystem

To assess the performance overhead and effectiveness of different audit techniques, the following experiments are conducted.

- *Audit Effectiveness with Emulated Call-Processing Client.* This experiment uses emulated call-processing client as the workload and measures the effectiveness of the audits in preventing database errors from corrupting the client processes/threads.
- *Overhead in Database API.* This experiment measures the performance overhead introduced into the database API functions that were modified to communicate with the audit process and to maintain the support data structure.

5.1 Audit Effectiveness

To demonstrate the usefulness of the audit process in protecting database clients from errors in database, a simple call-processing client was used. The client provides the basic call-processing service of setting up and tearing down a call without additional features such as call waiting or paging. The program uses multiple threads to concurrently handle incoming calls. The steps followed in each call-processing thread include authentication, resource allocation, and other phases in a typical call setup, as shown in Figure 2.

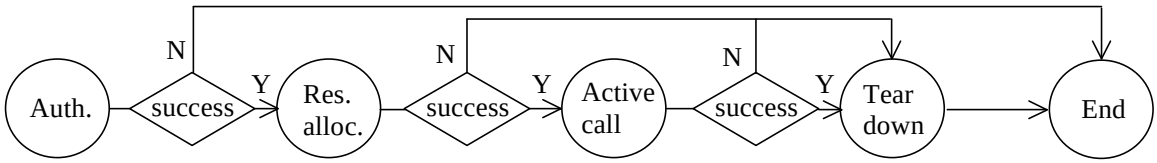

Figure 2: Call-Processing Phases Emulated in the Client Program

A run of the experiment lasted 2000 seconds, during which approximately 1000 calls were processed by the multi-threaded client program depicted in Figure 2. Random bit errors were inserted into the database at various rates, and the number of errors that escaped from the audits as well as the average call setup time were recorded. The goal of the experiment was to determine the reduction in the number of escaped errors (i.e., errors that impact the database client program) when audits are used, as well as the performance overhead seen by the client program. Table 2 shows the parameters used in the experiments.

Table 3 shows the results of running the call-processing client program with and without database audit at a fixed error rate of 1error every 20 seconds, using data from 30 runs of the experiment. Without database audits, 1884 out of the 3000 injected errors (63%) affected the application process, compared with only 402 (13%) when audits were running. The number of errors having no immediate effect, i.e., latent errors, is also greatly reduced (from 1116 to 55) when audits are running, because the entire database is checked for errors periodically. The timing of audit invocation with respect to error occurrence and the accuracy of constraints (e.g., range limits) were the two major factors in determining the number of escaped errors. Because the audits are invoked periodically in the experiment, every piece of data that is corrupted and then used by the client application between two consecutive audit invocations leads to an escaped error. In addition, the audit does not always have enforceable rules to detect all errors. Due to the processing time required by the audits, the average call setup time in the client process changes from 160 milliseconds to 270 milliseconds, i.e., a 69% increase, as measured on a Sun UltraSPARC-2 system.

Duration of each run	2000 seconds
Number of client threads	16
Call duration	20 ~ 30 seconds
Average call inter-arrival time	10 seconds
Error inter-arrival time	2,4,6,8,10,12,14,16,18,20 seconds
Interval of periodic audit	10 seconds

Table 2: Experiment Parameters

Total number of injected errors = 3000	Without Audits	With Audits
Number of errors escaped from audits and affecting application	1884 (63%)	402 (13%)
Number of errors caught by audits	N/A	2543 (85%)
Other (number of errors escaped from audits but having no effect on application)	1116 (37%)	55 (2%)
Average call setup time (msec)	160	270

Table 3: Comparison of Running Client Process with and without Audits using a 20-second Fault/Error Inter-Arrival Time

Table 4 shows a more detailed breakdown of the data presented in the last column of Table 3. The errors are classified based on their types, and errors that are not induced by the random bit-flip process (i.e., process failure and deadlock) are not shown. The results show that structural audit and static data audit are both very effective in detecting and removing errors and that both achieve 100% coverage. Dynamic data audit, on the other hand, has a lower coverage and is able to detect and remove a total of 79% (45% + 34%) of all errors in dynamic fields through range check and referential integrity check. Fourteen percent of errors escaped because the erroneous data had been used by the application process before the audit could detect them. More frequent invocation of audit is needed to reduce the number of error that escaped due to timing. Four percent of the errors escaped detection because of the lack of enforceable rules available to dynamic data audit. Suitable constraints need to be added to the database to improve audit coverage in this category.

To further investigate the relationships among audit coverage, audit invocation frequency, and error rate, the inter-arrival time of random errors is varied from 2 seconds to 20 seconds. All other parameters remain the same as shown in Table 2. Figure 3 shows that the number of escaped errors increases as the fault/error inter-arrival time decreases (error rate increases). More importantly, as the fault inter-arrival time drops below 10 seconds (the audit period used in the experiments), the increase in the number of escaped errors speeds up as the audits start to become overwhelmed by the burst of errors. The percentage of escaped errors in all injected errors remains relatively constant and ranges from 8% to 14%. In fact, as the error rate increases above the audit frequency, the percentage of escaped errors increases slowly. These results show that the database audit is useful in removing data errors and preventing error propagation under different error rates and that it does not break Dow, even when the error rate is high. The curve shows a gradual change instead of an abrupt jump as the error inter-arrival time crosses the audit period; this indicates continuity in the performance of audit.

Error types	Structural		Static Data		Dynamic Data				
					Detected		Escaped		No Effect
	Detected	Escaped	Detected	Escaped	By Range Check	By Semantic Check	Due to timing	Due to lack of rule	
Number of errors	194	0	623	0	975	751	313	89	55
Percentage	100%	0%	100%	0%	45%	34%	14%	4%	3%

Table 4: Breakdown of Inserted and Detected Errors

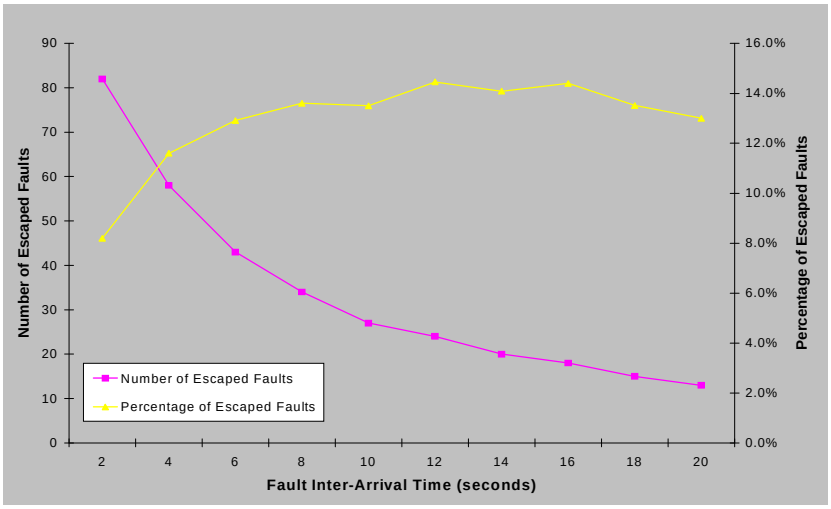

Figure 3: Number of Escaped Errors under Different Error Rates

In the region for fault/error inter-arrival time less than four seconds (in Figure 3), there is a seeming contradiction: the number of escapes increases, but the percentage of escapes decreases. This happens because, although the number of errors in the database at these error rates is high, escapes as a percentage are relatively low. This is not an acceptable situation, however, because each failure resulting from the escapes may need to be recovered, resulting in significant overhead, i.e., lower availability.

The results also show that even though the audits are effective in error detection and recovery, there is no guarantee that they can remove all data errors and in 13% of cases the error is propagated to the client despite the audits. Therefore, it is important to build checking at the database client to prevent failures of the entire call-processing application due to database corruptions.

5.2 Overhead in Database API

Figure 4 shows the average running times of the database API functions that have been modified to support various audit functionalities (see Table 1). The results are obtained by executing each function in its original and modified form 200 times on a Sun UltraSPARC-2 workstation running Solaris 2.7. The lighter portion of each bar indicates the average execution time of the original version, and the darker portion shows the time overhead introduced by the modifications made to each function. The average running times of individual functions are on the order of tens or hundreds of microseconds, and the overhead ranges from 6.5% (for DBinit) to 45% (for DBwrite_rec). In the latter case, the major contributor to the overhead is the time necessary to notify the audit process that a database client performed a write operation. Recall that this notification is used to initiate event-triggered audit. Using periodic audit eliminates this source of overhead.

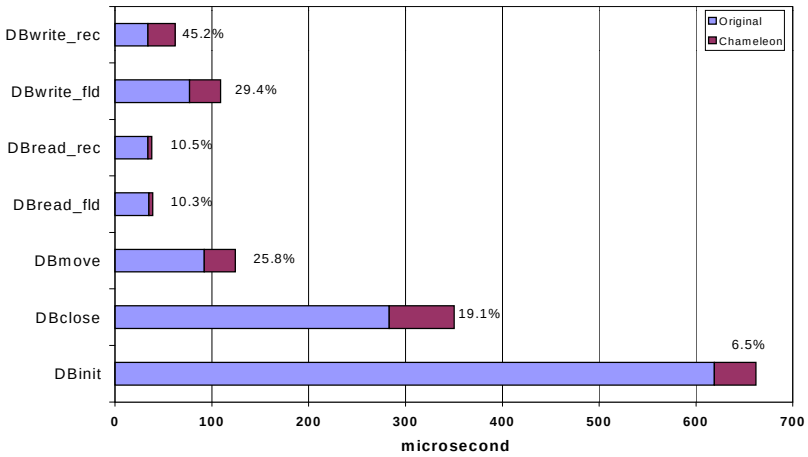

Figure 4: Run-Time Overhead of Modified Database API

5.3 Assessment of Prioritized Audit Triggering

The purposed of the experiment was to determine whether prioritized audits are equally or more effective than a full audit. Toward this end, an experiment was set up using six database tables, each with its own access frequency, to emulate a varying usage rate by a call-processing client. Assessment of the selective monitoring technique is left out owing to space constraints (the reader can find full description in [LIU00]). The goal is to compare the number of escaped errors and average detection latency between unprioritized audit and prioritized audit. An escaped error is a piece of erroneous data that is used by an application process before the audit program can detect it. In unprioritized audit, the six tables are checked in a fixed order with the same frequency regardless how each table is used by the application.

In prioritized audit, the tables that are used more often are checked more frequently. Two error models are used in the tests. The first model assumes that all memory locations have the same probability of error, i.e., uniform error distribution, whereas the second model assumes that the number of errors in a table is proportional to its access frequency. These two error models are motivated by transient hardware errors and environment-triggered errors, which tend to be random, and software bugs and runtime anomaly, which are typically related to system activities. The detailed emulation parameters are summarized in Table 5. The relative sizes and access frequency ratios are from actual controller database measurements.

Number of tables	6
Relative size ratio	7 : 18 : 1 : 125 : 8 : 4
Access frequency ratio	6 : 5 : 4 : 3 : 2 : 1
Number of application threads	16
Database access frequency for each application thread	20 operations per second (average)
Audit frequency	1 table every 5 seconds
Error injection rates (exponential distribution)	$1 / \lambda$ = 1, 2, 4 seconds

Table 5: Test Parameters

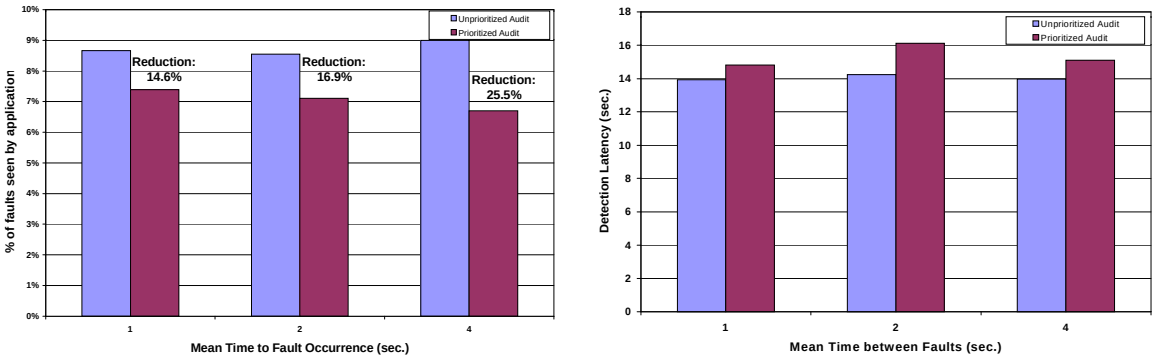

Figure 5: (a) Proportion of Escaped Errors (b) Error Detection Latency for Uniform Error Distribution

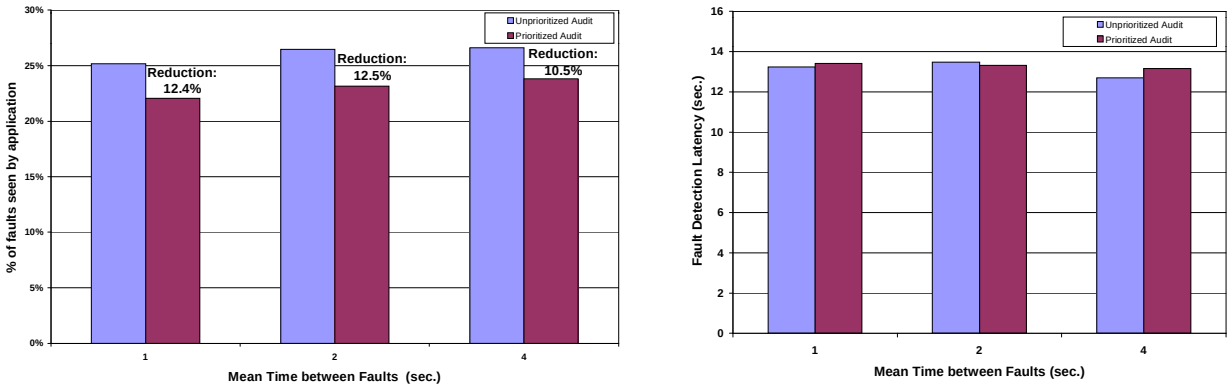

Figure 6: (a) Proportion of Escaped Errors (b) Error Detection Latency for Proportional Error Distribution

Figures 5(a) and 5(b) show the effectiveness of prioritizing audits under both uniform and proportional error distributions. Figure 5(a) shows the difference in the percentage of escaped errors for three different error rates under the uniform error distribution model. The relative height of the bars shows that the reduction in the number of escaped errors ranges from 14% to 25% when prioritized audit is used. This is a significant reduction, since each error that escapes the detection of audit could ultimately contribute to system downtime, i.e., the call-processing application crashes, losing many calls. This chart confirms that errors in heavily used tables are more likely to manifest themselves and that it is worthwhile to check them more frequently. Furthermore, the improvement becomes greater as the mean time between errors increases from one second to four seconds. This suggests that prioritized audit is more effective under lower error rates. Figure 5(b) shows the average error detection latency for uniform error distribution. As the chart indicates, the use of prioritized audits actually results in slightly higher detection latency under the three error rates. This is not surprising because when the errors are distributed evenly, focusing on a small subset of memory area will inevitably delay the detection of errors in other areas, thus increasing the average latency. However, the slightly higher average detection latency is due mostly to errors that are unlikely to manifest themselves. Therefore, prioritized audit is

still an effective optimization because it reduces the number of errors that can escape and impact application processes.

Figure 6 shows the differences in the proportion of escaped errors and detection latency under the proportional error model, which assumes more errors in more heavily used tables. Compared with the uniform error distribution model, the absolute number of escaped errors is much higher (around 25% of all injected errors), but the relative reduction in escaped errors due to the use of prioritized audit is still noticeable (around 12%). Furthermore, there is almost no difference in average error detection latency, which means prioritized audit can detect more errors in a small subset of the tables, so that the overall average latency does not increase.

The preceding results show that 13% of errors in the database propagate to the client, i.e., the client accesses and subsequently operates on incorrect data. This can make it impossible to set up the new incoming call, or the call-processing application may crash loosing all calls in progress. In this scenario, hardware/software-based recovery might be required to restore system operation. The question is: to what extent does client misbehavior cause corruption to the database, potentially leading to the failure scenario described above? The next section attempts to answer this question.

6 Joint Assessment of Data Audit and Control Flow Checking Applied to the Call-Processing Environment

In this section, we first introduce a new control flow checking technique to preemptively (i.e., before an application crash) detect control flow errors that impact the application. We then provide details of error injection based evaluation of (1) the impact of control flow errors in the client on the database (or more generally on the call-processing environment) and (2) the joint effectiveness of preemptive control flow checking (designed to cope with control flow errors) and data audits (designed to protect the database) in preventing error propagation and resulting in uncontrolled application crashes and hangs.

6.1 Preemptive Checking Principle

To protect the control flow of the application, we propose the Preemptive Control Signatures (PECOS) technique [BAG00]. PECOS monitors the runtime control path taken by an application and compares this with the set of expected control paths to validate the application behavior. The scheme can handle situations in which the control paths are either statically or dynamically (i.e., at runtime) determined. The application is decomposed into blocks at the assembly level, and a group of PECOS instructions called Assertion Blocks are embedded in the instruction stream of the application to be monitored. Normally, each block is a basic block in the traditional compiler sense of a branch-free interval, and each basic block is terminated by a Control Flow Instruction (CFI), which is used as a trigger for inserting PECOS Assertion Blocks. The Assertion Block contains (1) the set of valid target addresses (or address offsets) the application may jump to, which are determined either at compile time or at runtime, and (2) code to determine the runtime target address. The determination of the runtime target address and its comparison against the valid addresses is done *before* the

jump to the target address is made, i.e., preemptively. In case of an error, the Assertion Block raises a *divide-by-zero* exception, which is handled by the PECOS signal handler. The signal handler checks whether the problem was caused by a control flow error[5], and if so takes a recovery action, e.g., terminates the malfunctioning thread of execution.

6.1.1 PECOS Instrumentation

To automate the instrumentation, we developed a PECOS parser, which embeds assertions into the application assembly code. The current parser is implemented for the SPARC architecture. At compile time, the PECOS tool instruments the application assembly code with Assertion Blocks placed at the end of each basic block. Note that the Assertion Block itself does not introduce additional CFIs. Since we are trying to protect a CFI, it defeats the purpose to have the Assertion Block insert any further CFIs. At runtime, the task of the Assertion Block can be broken down into two subtasks:

1. Determine the runtime target address of the CFI (referred to as X_{out} in the following discussion). We consider the following situations: (a) the target address of CFI is static, i.e., it is a constant embedded in the instruction stream, (b) the target address is determined by runtime calculation, and (c) the target is the address of dynamic library call.
2. Compare the runtime target address with the valid target addresses determined by a compile-time analysis or runtime computation. In general, the number of valid target addresses can be one (jump), two (branch), or many (calls or returns). For two valid target addresses, *X1* and *X2*, the resulting control decision implemented by the Assertion Block is shown in Figure 7. The comparison is designed so that an impending illegal control flow will cause a divide-by-zero exception in the calculation of the variable ID, indicating an error.

1. Determine the runtime target address [= X_{out}]
2. Extract the list of valid target addresses [= {X1,X2}]
3. Calculate ID := X_{out} * 1/P,
   where, P = ![(X_{out}-X1) * (X_{out}-X2)]

Figure 7: High-level Control Decision in the Assertion Block

6.1.2 Error Injection Setup

To evaluate the effectiveness of PECOS preemptive control flow checking in protecting the database client, a set of error injection experiments is performed[6]. The system configuration is analogous to the one presented in Figure 1 with PECOS embedded into the client code. We define four sets of error injection campaigns: (1) without PECOS, without Audit, (2) without PECOS, with Audit, (3) with PECOS, without Audit, and (4) with

[5] The PECOS signal handler examines the PC (program counter) from which the signal was raised, and if it corresponds to a PECOS Assertion Block, concludes that a control flow error raised the signal.

[6] NFTAPE, a software-implemented fault/error injection tool, is used for conducting the error injection campaigns [STO00].

PECOS, with Audit. For each campaign, the call-processing client is the target of error injection. Table 6 provides brief descriptions of the error models used in the error injection campaigns.

ADDIF	Address line error resulting in execution of a different instruction taken from the instruction stream
DATAIF	Data line error when an opcode is being fetched
DATAOF	Data line error when an operand is being fetched
DATAInF (RAND)	Data line error when an instruction (whether opcode or operand) is being fetched

Table 6: Error Models

These error models are based on the extensive experiments of Abraham *et al.* [KAN95], adding random memory errors (DATAInF). For one set of campaigns, injections are done in the entire text segment of the application. For the second set, injections are performed only to the control flow instructions in the text segment. Thus, a total of 4 * 4 * 2 = 32 error injection campaigns are defined. For each campaign, 200 runs are performed, giving a total of 32 * 200 = 6,400 runs for the call-processing workload.

A breakpoint is the trigger for the error injection, and in each run only one error is injected. However, interestingly, a single error injection may affect multiple client threads. Since call processing is a multi-threaded application, if an error is injected into even a single instruction, it is possible that another thread may execute the same erroneous instruction. In the error injection methodology followed, once a thread reaches a breakpoint, the error is injected, the thread is made to execute the erroneous instruction, and then the error is removed. But, in the time interval between reaching the breakpoint and restoring the correct instruction, other thread(s) may come and execute the erroneous instruction. Therefore, cases of multiple errors being activated are observed.

6.1.3 Error Classification

The classification of outcomes from error injection experiments is given in Table 7. A run is considered to be in the "Error Activated but Not Manifested" category if the erroneous instruction is executed, none of the detection schemes flags an error, the comparison of the database records by the client prior to termination of the thread (step (5) in Figure 8) does not detect a mismatch, and the client prints the message indicating it completed successfully. If neither PECOS detection nor system detection flags an error, the final comparison of golden and runtime records by the client does not detect a mismatch, but the client does NOT print the message that it completed successfully, it is taken to be a case of "Application Hang." If the comparison by the client of the records in the database and the records that it wrote to the database detects a mismatch, this is taken as a case of "Fail-silence Violation." The rationale behind this is that an error in the client process has resulted in the writing of a corrupt record to the database. Since this database is shared among all the call-processing threads, the writing of a corrupt record to the database can lead to error propagation.

Category	Description
Error Not Activated	The erroneous instruction is not reached in the control flow of the application. These runs are discarded from further analysis.
Error Activated but Not Manifested	The erroneous instruction is executed, but it does not manifest as an error, i.e., the call-processing application exhibits correct behavior.
PECOS Detection	PECOS Assertion Blocks detect the error prior to any other detection technique or any other result.
Audit Detection	One of the audit mechanisms detects an error in the database
System Detection	The operating system detects the error by raising a signal (e.g., SIGBUS) and as a result the database client crashes.
Client Hang	The client gets into a deadlock or livelock and does not make any progress.
Fail-silence Violation	The client writes an incorrect data to the database; this is a major source of error propagation.

Table 7: Notation for the Results Categories of Error Injection Runs

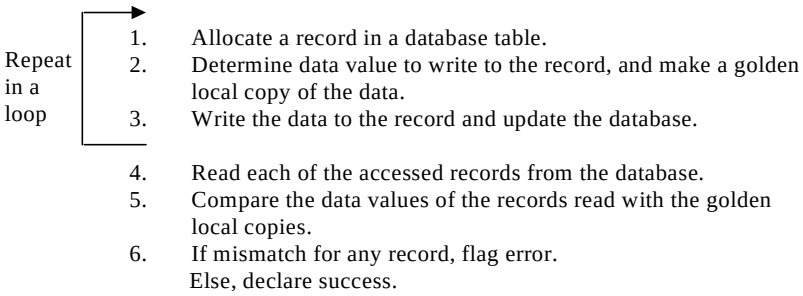

Figure 8: Main Control Structure of Database Client

6.1.4 Results and Discussion

This section presents results (Table 8 and Table 9) and discussion of our findings from error injection experiments conducted to evaluate the join effectiveness of data checking and control flow checking applied to the call-processing environment. The values in the two tables are percentages of runs resulting in each possible outcome (listed in the first column) followed by 95% confidence intervals (in parentheses)[7]. The confidence intervals are calculated assuming a binomial distribution. The last row gives the number of runs used in the error injection campaigns[8].

Table 8 and Table 9 present, respectively, a summary of results from the directed injection to control flow instructions and a summary of results from random injection to the instruction stream. The key results from the error injection campaigns are:

- Combined use of audits and PECOS is effective in reducing the proportion of system detection[9] (66% to 39% for random injections, and 52% to 19% for directed injections). Preventing the application crash allows for graceful termination of offending thread(s) (very vital recovery for our target application)

[7] Note that confidence intervals are not provided (as they are not meaningful) for the outcome categories with small number of observations. For these outcome categories, the two tables give a row number of observed cases.

[8] The number of runs per error injection campaign varies (i.e., it is not always 800) because we ignore experiments for which the application, because of an error, does not even start to execute and consequently we cannot classify these cases a operation system detection or program hang.

[9] Recall that system detection is equivalent to a call-processing application crash.

without affecting other, currently processed calls and thus to guarantees high availability of the overall system.

Category	Without PECOS Without Audit	Without PECOS With Audit	With PECOS Without Audit	With PECOS With Audit
Errors Not Activated	58%	53%	50%	50%
Errors Activated but Not Manifested	46% (40, 51) [%]	53% (48, 59) [%]	2% (1, 4) [%]	4% (2, 6) [%]
PECOS Detection	N/A	N/A	83% (79, 87) [%]	77% (73, 81) [%]
Audit Detection	N/A	8% (6, 12) [%]	N/A	1
System Detection	52% (47, 58) [%]	37% (32, 42) [%]	14% (11, 18) [%]	19% (15, 23) [%]
Client Hang	6	4	0	0
Fail-silence violation	1	1	1	0
Total Number of Injected Errors	777	738	800	787

Table 8: Cumulative Results from Directed Injection to Control Flow Instructions

Category	Without PECOS Without Audit	Without PECOS With Audit	With PECOS Without Audit	With PECOS With Audit
Errors Not Activated	64%	73%	52%	53%
Errors Activated but Not Manifested	28% (23, 34) [%]	26% (20, 32) [%]	12% (9, 15) [%]	7% (5, 10) [%]
PECOS Detection	N/A	N/A	45% (40, 50) [%]	49% (44, 54) [%]
Audit Detection	N/A	7% (4, 12) [%]	N/A	2% (1, 4) [%]
System Detection	66% (60, 71) [%]	61% (54, 68) [%]	41% (36, 46) [%]	39% (34, 44) [%]
Client Hang	4	5	2	2
Fail-silence violation	5% (3, 8) [%]	3% (1, 7) [%]	2% (1, 4) [%]	2% (1, 4) [%]
Total Number of Injected Errors	745	774	800	800

Table 9: Cumulative Results from Random Injection to the Instruction Stream

- Although the number of observed client hangs is small, the results indicate that audits and PECOS are efficient in detecting these failures. Usually mechanisms for detecting hangs use a timeout to determine whether a process/thread is still operational (or makes a progress). Arbitrary/static assumptions on what should be a timeout value do not work well in highly dynamic environments, such as call processing. More complex (difficult to implement) dynamic schemes capable of on-line adjusting of the timeout (based on current system activity) are required. Our techniques allows for quick (low latency) detection of client hangs and consequently enable fast recovery.
- The proportion of fail-silence violations is quite negligible (one case out of 328 activated errors; second column in Table 8). This indicates that in a majority of the cases, direct corruption of control flow instructions in the client does not manifest in corruption of the database. This is because of the structure of the client where there is a little scope for an incorrect control flow to cause the client to write a data different from the local data value, which is the only way a fail-silence violation to occur in the experimental setup.

The situation is different when we perform random injection to the client instruction stream. The results in Table 9 indicate larger number of fail-silence violations, 13 cases (5%) out of 267 activated errors, (second column in Table 9). Random injections can impact any instruction (not necessarily a control flow instruction) in the instruction stream of the client, and there is a higher chance that the client will write incorrect data to the database. The proposed data and control checking mechanisms successfully reduce the fail-silence violations to about 2% (seven cases out of 372 activated errors, last column in Table 9).

The above results allow us to estimate the proportion of error propagation from the client to the database while the client has an error. For injections to the application *without PECOS with audit* (see Table 9), it is seen that for random injections to the text segment of the call-processing client, the data audits pick up 7% of errors. Considering the efficiency of audits of detecting actual database errors to be 85% (see Table 3), the proportion of the error propagation is 7*(100/85) = 8%. This indicates that error propagation due to client vulnerability can lead to database corruption (i.e., fail-silence violation). As we have just discussed, our detection techniques can significantly improve fail-silence coverage.

In the previous discussion, we considered the impact of errors injected separately into the client and the database. An important question to ask now is: if errors impact both the client and the database, which of the proposed techniques (data audits or PECOS) is more valuable. Intuitively, the answer will depend on the relative proportion of the errors occurring in the database and in the client. Table 10 summarizes (a) the detection coverage obtained when the injection targets are either only the database or only the client (we use figures from Table 3 and Table 9) and (b) the estimated detection coverage when injections target simultaneously the database and the client. Coverage is given by *{100%- (System Detection + Fail Silence Violation + Hang)%}.*

In the latter case, we assume (based on the relative size of database memory image and the client text segment size) that 75% of errors are directly injected to the database and 25% to the client. We also assume that effectiveness of PECOS and audits remain approximately equal with the error rate (one error every 20-30 seconds) used in the conducted experiments. The results show that audits and PECOS jointly achieve 80% error detection coverage. It can also be observed that audits and PECOS alone achieve coverage of 73% and 42%, respectively (for assumed error mix). There is not much overlap, in terms of error types covered by audits and PECOS, and consequently we need both data and control checking to guarantee high detection coverage. The key reason for error escapes is corruption to the client, which does not manifest as a control flow error or database error but nonetheless leads to the client crash. These errors in the internal data of the client cannot be captured by the audit techniques. Consequently, to improve the overall coverage, it is important to have mechanisms for detecting errors in the data flow of the client.

Error Target	Without PECOS Without Audit	Without PECOS With Audit	With PECOS Without Audit	With PECOS With Audit
Client	28%	33% (26+7)	57% (12+45)	58% (7+49+2)
Database	37%	87% (85 +2)	37%	87%
Client + Database (25%-75% error mix)	35% (0.25*28 + 0.75 * 37)	73%	42%	80%

Table 10: System-wide Coverage for Database or Client Errors

7 Conclusions

The paper presents the design, implementation, and evaluation of a framework for providing data and control flow error detection and recovery in a wireless call-processing environment. In the proposed design, the data audit (including static and dynamic data check, structural check, and semantic referential integrity check) is responsible for detecting and correcting errors in the database of the target system. To protect the call-processing clients from control flow errors, a preemptive control flow checking technique, PECOS, is applied. The effectiveness of these techniques is evaluated through detailed error injection experiments. Several types of errors (control errors, address and data line errors, etc.) and several error rates are used for the system evaluation. An estimate of the system-wide coverage for various combinations of the two available classes of techniques (data audits and PECOS) is also provided.

The results show that the data audit subsystem is successful in detecting about 85% of the errors injected to the database. The incidences of coverage miss arise due to audits not running frequently enough or to suitable constraints on field values not being available in the database catalog. In spite of the audit subsystem, more than 13% of the database corruptions propagated and affected the clients. PECOS was shown to substantially reduce the incidence of process crashes and hangs for control flow injections to the call-processing clients. Even with PECOS present, about 8% of the client errors resulted in incorrect data being written to the database. A study of the combined effect of PECOS and data audits on the overall system coverage showed that data audits were more effective than PECOS for database and hybrid errors, while PECOS performed better for control errors to the clients.

Acknowledgments

This work was supported in part by JPL REE program and in part by Motorola Inc.

References

[ALK99] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, “Design and Evaluation of System-Level Checks for on-line Control Flow Error Detection,” IEEE Transactions on Parallel and Distributed Systems, Vol. 10, No. 6, pp. 627-641, June 1999.

[BAG00] S. Bagchi, “Hierarchical Error Detection in a SIFT Environment,” Ph.D. Thesis, Advisor: R. K. Iyer, University of Illinois at Urbana-Champaign, December 2000.

[BLA80] J. P. Black, D. J. Taylor, D. E. Morgan. An Introduction to Robust Data Structures. *Digest of Papers*, The 10th International Symposium on Fault-Tolerant Computing, Kyoto, Japan, 1980. pp. 110-112.

[BRA96] F.V. Brasiliero, et al., "Implementing Fail-Sailent Nodes for Distributed Systems," IEEE Trans. on Computers, vol.45, pp. 1226-1238, 1996.

[COS00] D. Costa, T. Rilho, H. Madeira, "Joint Evaluation of Performance and Robustness of a COTS DBMS through Fault-Injection," in *Proc. Int. Conference on Dependable Systems and Networks*, 2000, pp. 251-260.

[ERN99] M. Ernst, J. Cockrell, W. Griswold, D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proc. of the 21st International Conference on Software Engineering.* 1999. pp. 213-224.

[HAU85] G. Haugk, F. M. Lax, R. D. Royer, J.R. Williams. The 5ESS Switching System: Maintenance Capabilities. *AT&T Technical Journal*, vol. 64, no. 6, 1985. pp. 1385-1416.

[KAN95] G. Kanawati, N. Kanawati, J. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," IEEE Trans. on Computers, pp. 248-260, February 1995.

[KAN96] G.A. Kanawati, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, "Evaluation of Integrated System-Level Checks for on-line Error Detection," Proc. IEEE Int'l Symp. Parallel and Distributed Systems, pp. 292-301, Sept. 1996.

[LEV99] H. Levendel. Private communication. May 1999.

[LIU00] Y. Liu, "Database Error Detection and Recovery in a Wireless Network Controller," M.S. Thesis (Advisor: R. K. Iyer), Center for Reliable and High-Performance Computing, Univ. of Illinois, December 2000.

[MAD91] H. Madeira, J. G. Silva, "On-Line Signature Learning and Checking," Proc. 2nd IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-2), pp. 170-177, Feb. 1991.

[MAH88] A. Mahmood, E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors—A Survey," IEEE Trans. on Computers, Vol. 37, No. 2, pp. 160-174, Feb. 1988.

[MIC91] T. Michel, R. Leveugle, G. Saucier, "A New Approach to Control Flow Checking without Program Modification," Proc. 21st International Symp. on Fault-Tolerant Computing (FTCS-21), pp. 334-341, 1991.

[MIR92] G. Miremadi, J. Karlsson, U. Gunneflo, J. Torin, "Two Software Techniques for On-Line Error Detection," Proc. 22nd International Symp. on Fault-Tolerant Computing (FTCS-22), pp. 328-335, July, 1992.

[MIR95] G. Miremadi, et al., "Use of Time and Address Signatures for Control Flow Checking," Proc. IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-5), pp. 113-124, 1995.

[NAM82] M. Namjoo, "Techniques for Concurrent Testing of VLSI Processor Operation," Digest 1982 Intl. Test Conf., pp. 461-468, November 1982.

[OHL92] J. Ohlsson, M. Rimen, U. Gunneflo, "A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog," Proc. Int'l Symp. on Fault-Tolerant Computing (FTCS-22), pp. 316-325, 1991.

[SCH86] M. Schuette, J.Shen, D. Siewiorek, Y.Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes," *Proc. 16th Annual Int'l Symp. on Fault-Tolerant Computing (FTCS-16)*, pp. 138-143, 1986.

[SCH87] M.A. Schuette, J.P. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," IEEE Transactions on Computers, Vol. C-36, No. 3, pp. 264-276, March 1987.

[SET85] S. C. Seth, R. Muralidhar. Analysis and Design of Robust Data Structures. *IEEE*, 1985. pp. 14-19.

[STO00] D.T. Stott, B. Floering, Z. Kalbarczyk, R.K. Iyer, "Dependability Assessment in Distributed Systems with Lightweight Fault Injectors in NFTAPE," Proc. IEEE Int'l Computer Performance and Dependability Symp. (IPDS'2K), pp.91-100, March 2000.

[TAY80a] D. J. Taylor, D. E. Morgan, J. P. Black. Redundancy in Data Structures: Improving Software Fault Tolerance. *IEEE Transaction on Software Engineering*, vol. 6, no. 6, November 1980. pp. 585-594.

[TAY80b] D. J. Taylor, D. E. Morgan, J. P. Black. Redundancy in Data Structures: Some Theoretical Results. *IEEE Transaction on Software Engineering*, vol. 6, no. 6, November 1980. pp. 595-602.

[TAY85] D. J. Taylor, J. P. Black. Guidelines for Storage Structure Error Correction. *Digest of Papers*, The 15th Annual International Symposium on Fault-Tolerant Computing, Ann Arbor, Michigan, June 19-21, 1985. pp. 20-22.

[UPA94] Shambhu Upadhyaya, Bina Ramamurthy, “Concurrent Process Monitoring with No Reference Signatures,” IEEE Transactions on Computers, vol. 43 no. 4, pp. 475-480, April 1994.

[YAU80] S.S. Yau, F-Ch. Chen, “An Approach to Concurrent Control Flow Checking,” IEEE Trans. on Software Engineering, Vol. SE-6, No. 2, pp. 126-137, 1980.

[WIL90] K. Wilken, J.P. Shen, “Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Errors,” IEEE Transactions on Computer Aided Design, pp. 629-641, June 1990.

[SYBASE] Sybase Adaptive Server System Administration Guide, Chapter 25 http://manuals.sybase.com:80/onlinebooks/group-as/asg1200e/asesag

[TIMESTEN] TimesTen 4.0 Datasheet, http://www.timesten.com/products/ttdatasheet.html

[ORACLE] Oracle8 Server Utilities, Release 8.0, Chapter 10, http://pundit.bus.utexas.edu/oradocs/server803/A54652_01/toc.ht