Gashi, I., Popov, P. & Strigini, L. (2007). Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. IEEE Transactions on Dependable and Secure Computing, 4(4), 280 - 294. doi: 10.1109/TDSC.2007.70208 <http://dx.doi.org/10.1109/TDSC.2007.70208>

Fault Tolerance via Diversity for Off-The-Shelf Products: a Study with SQL Database Servers

Ilir GASHI, Peter POPOV, and Lorenzo STRIGINI, *Member, IEEE*

Abstract— If an off-the-shelf software product exhibits poor dependability due to design faults, software fault tolerance is often the only way available to users and system integrators to alleviate the problem. Thanks to low acquisition costs, even using multiple versions of software in a parallel architecture, a scheme formerly reserved for few and highly critical applications, may become viable for many applications. We have studied the potential dependability gains from these solutions for off-the-shelf database servers. We based the study on the bug reports available for four off-the-shelf SQL servers, plus later releases of two of them. We found that many of these faults cause systematic, non-crash failures, a category ignored by most studies and standard implementations of fault tolerance for databases. Our observations suggest that diverse redundancy would be effective for tolerating design faults in this category of products. Only in very few cases would demands that triggered a bug in one server cause failures in another one, and there were no coincident failures in more than two of the servers. Use of different releases of the same product would also tolerate a significant fraction of the faults. We report our results and discuss their implications, the architectural options available for exploiting them and the difficulties that they may present.

Index Terms— C.4.b Fault tolerance, C.4.f Reliability, availability, and serviceability, H.2.4.i Relational databases, D.2.17.e Error processing, design diversity, COTS software, fault records, non-crash failures, database availability, experimental results.

1 Introduction

THE use of "off-the-shelf" (OTS) – rather than custom-built – products is attractive in terms of acquisition costs and time to deployment but brings concerns about dependability and "total cost of ownership". For safety- or business-critical applications, in particular, purpose-built products would normally come with extensive documentation of good development practice and extensive verification and validation; when switching to mass-distributed OTS systems, users – system designers or end users – often find not only a lack of this documentation, but anecdotal evidence of serious failures and/or bugs that undermines trust in the product. Despite the large-scale adoption of some products, there is usually no formal statistical documentation of achieved dependability levels, from which a user could attempt to extrapolate the levels to be achieved in his/her own usage environment.

For all these reasons, when systems are built out of OTS products, fault tolerance is often the only viable way of obtaining the required system dependability [1], [2], [3]. These considerations apply not only to OTS software, but also to hardware, like microprocessors, or complete hardware-plus-software systems. In this paper we will consider "software fault tolerance" (by which we mean "fault tolerance against software faults"), focusing on a specific category of software products. Fault tolerance may take multiple forms [4], from simple error detection and recovery add-ons (e.g. wrappers) [5] to full-fledged "diverse modular redundancy" (e.g. "N-version programming": replication with diverse versions of the components) [4]. Even this latter class of solutions becomes affordable with many OTS products and has the advantage of a fairly simple architecture. The cost of procuring two or even more OTS products (some of which may be free) would still be far less than that of developing one's own product.

All these design solutions are well known from the literature. The questions, for the developers of a system using OTS components, are about the dependability gains, implementation difficulties and extra cost that they would bring for that specific system. We report here some evidence about potential gains, and briefly discuss the architectural issues that would determine feasibility and costs, for a specific category of OTS products: SQL database servers, or "database management systems" (DBMSs)[1].

This category of products offers a realistic case study of the advantages and challenges of software fault tolerance in OTS products. DBMS products are complex, mature enough for widespread adoption, and yet with many faults in each release[2]. Fault tolerance in DBMS products is a thoroughly studied subject, with standard recognized solutions, some of which are commercially available. But these solutions do not give full protection against software faults, because they assume fail-stop [8] or at least self-evident

• *I. Gashi, P. Popov and L. Strigini are with Centre for Software Reliability, City University, Northampton Square, London EC1V 0HB, UK (telephone: + 44 20 7040 {0273, 8963, 8245}, e-mail: {ec233, ptp, strigini}@csr.city.ac.uk).*

Manuscript received 10th of October 2006.

[1] Everyday terms may be ambiguous when discussing redundant and diverse architectures. We will apply these conventions: a *DBMS product* is a specific software package; a fault-tolerant database server includes one or more *channels* (each performing the database server function), each including an installation of a DBMS product (these may be the same product or different ones - different *versions*) and a *replica* of the database. Two replicas of the database will be physically different if they are in channels that use different DBMS products. They may also exhibit temporary differences due to the asynchronous operation of the channels. We follow the popular usage of the word "bug" as synonym for "software fault" or "defect".

[2] And even features that imply an accepted possibility of an incorrect behavior, albeit rare. An example of the latter is the known "write skew" [6] problem with some optimistic concurrency control architectures [7]

failures[3]: errors are detected promptly enough that the database contents are not corrupted, or that a suitable correct checkpoint can be identified and used for rollback. There is no guarantee that software faults in the OTS DBMS products themselves will satisfy this assumption. As we document here, they do not, and we know of no published statistical evidence of the frequency of violations, which one could use as evidence that the assumption is satisfied with high enough probability for a specific application of one of these OTS products.

There are many OTS SQL DBMS products, obeying (at least nominally) common standards (SQL 92 and SQL 99), which makes diverse redundancy feasible in principle. For instance, a parallel-redundant architecture using two replicas of a database, managed by two diverse DBMS products, would allow error detection via comparison of results from the two DBMS products. A fault-tolerant server capable of tolerating server software faults can be built from installations of two or more diverse DBMS products, connected by middleware that makes them appear to clients as a single database server. There are clearly problems as well: in particular, existing DBMS products have certain concurrency control and fault tolerance features that rely on lack of diversity between replicated executions for their proper and efficient operation. However, it is worth exploring the costs and benefits of solutions that accept the drawbacks of diversity in return for improved dependability. For many users, there is no practical alternative to OTS DBMS products, and performance losses may well be acceptable in return for improved assurance. In addition to tolerating faults in general, users may look at software fault tolerance as a way of guaranteeing good service during upgrades of the DBMS products, when new bugs might appear that are serious under the usage profile of their specific installation, and/or of delaying "patches" and upgrades, thus reducing the total cost of ownership of DBMS products.

As a preliminary assessment of the potential effectiveness of software fault tolerance with DBMS products, we have studied publicly available fault reports for four DBMS products (two open-source and two closed-development). We ask questions about the potential effectiveness of *design diversity* – deploying two different products. Fault reports are the only publicly available dependability evidence for these products, so our study concerns *fault diversity* among them. Complete failure logs would be much more useful as statistical evidence, but they are not available. Many vendors discourage users from reporting already known bugs; detailed failure data are rarely available even to the software vendors themselves. This scarcity of data also makes it difficult to estimate how dependable a DBMS product will be for a specific installation. But the many reports of failures of DBMS products suggest that some users need reliability improvements.

In a first study [9], we looked at the set of bugs reported for one release of each DBMS product. For each bug, we took the bug script (a sequence of SQL statements) that would trigger it and ran it on all four DBMS products (if possible), to check for coincident failures: if the bug script does not trigger failures in the other DBMS product, we take this as evidence that software fault tolerance would tolerate that fault. We found that a high number of reported faults would not be tolerated (or even detected) by existing, non-diverse fault-tolerant schemes but did not cause coincident failures in any two DBMS products, offering a way of tolerating them.

These intriguing results suggested a potential for considerable dependability gains from using diverse OTS DBMS products, but they only concerned a *specific snapshot* in the evolution of these products. We therefore ran a follow-up study with later releases of DBMS products (thus with different set of bug reports), with results that substantially confirm the previous ones. This paper reports the complete results of the two studies.

The rest of the paper is organized as follows: in Section 2, we briefly discuss the architectural issues in software fault tolerance with DBMS products – feasibility, design alternatives and performance issues – since they determine the usefulness of the empirical results we report; Section 3 presents the results of the two empirical studies of known bugs of DBMS products, including the comparisons between older and newer releases of two DBMS products; Section 4 contains a discussion of the implications of our studies; Section 5 contains a review of related work on database replication, interoperability of databases, empirical evidence on DBMS products' faults and failures and diversity with off-the-shelf components and Section 6 contains conclusions and outlines of further work.

2 Architectural Considerations

2.1 Current Solutions for DBMS Replication

Standard solutions for automatic fault tolerance in databases use the mechanisms of atomic transactions and/or checkpointing to support backward recovery, which can be followed by retry of the failed transactions. These solutions will tolerate transient faults, if detected, and if combined with replication will mask permanent faults, without service interruption.

Various data replication solutions exist [10], [11], [12], [13], [14]. In commercial DBMS products, they are often called "fail-over" solutions: following a (crash) failure of the primary DBMS product, the load is transparently taken over by a separate installation of the DBMS product holding a redundant copy of the database, at the cost of aborting the transactions affected by the crash. Multiple copies may be used. The code for fault tolerance is integrated inside the DBMS product. A recent survey [15] calls this a "white box" solution. Alternatively, replication can be managed by middleware separate from the DBMS products: "black box" solutions (fault tolerance is entirely the responsibility of the middleware), or "gray box" (the middleware exploits useful functions available from the DBMS products [16]). Our discussion here will refer to "black box" solutions: the only ones that can be built without access to OTS source code, and most convenient for studying the design issues in the

[3] By "self-evident failures" we will mean failures that a generic client of the DBMS product can detect without depending on knowledge of the specific database and its semantics. They are those failures that – as seen by the client – consist in issuing an error message to the client, spontaneously aborting a transaction, "hanging" or crashing.

use of redundancy and diversity. We will assume that fault tolerance is managed by a layer of middleware; clients see the fault-tolerant database server via this middleware layer, which co-ordinates the redundant channels.

Existing data replication solutions use sophisticated schemes for reducing the overhead involved in keeping the copies up to date. Their common weakness is their dependence on the assumption of "fail-stop" or at least "self-evident" failures. This assumption simplifies the protocols for data replication, and allows some performance optimization. For instance, in the Read Once Write All Available (ROWAA) [10] replication protocol the read statements[4] are executed by a single replica while the write statements are executed by all replicas. These fault-tolerant solutions are considered adequate by standardizing bodies [17], despite the assumption being false in principle. Some recent solutions [18] seek further optimization by executing the write statements on a single replica, which then propagates the changes to all the (available) replicas.

As we shall see, current OTS DBMS products suffer from many bugs that cause non-crash, non self-evident failures. The failures that these cause may be undetected erroneous responses to read statements, and/or incorrect writes to all the replicas of the database.

For these kinds of failure, the current data replication solutions are deficient, in the first place from the viewpoint of error detection. Two kinds of remedy are possible:

- *database-*, or *client-specific* solutions that depend on the client (an automatic process or a human operator) to run reasonableness checks on the outputs of the DBMS product and order recovery actions if it detects errors. Good error detection may be achieved by exploiting knowledge of the semantics of the data stored and the processes that update them. This knowledge may also support more efficient error recovery than simple rollback and retry. The main disadvantages are high implementation cost (especially with a workforce generally unaware of the need for fault tolerance), high run-time cost, at least for human-run checks, and the possibility of low error detection coverage if the database is – as common – the sole repository of the data[5].
- *generic* solutions that use active replication [19] for error detection, so that errors can be detected by comparing the results of redundant executions, and/or corrected, via voting or copying the results of correct executions.

2.2 Diversity

Replication will give a basis for effective fault tolerance if the multiple channels do not usually fail together on the same demand, or at least they tend not to fail with identical erroneous results. To pursue such *failure diversity*, a designer building a fault-tolerant database server can use various forms of diversity:

- simple separation of redundant executions. This is the weakest form, but it may yet tolerate some failures. It is well known that many bugs in complex, mature software products are "Heisenbugs"[6] [20], i.e., they cause apparently non-deterministic failures. When a database fails, its identical copy may not fail, even with the same sequence of inputs. Even repeating the same operations on the same copy of a database after rollback may in principle not replicate the same failure;
- *design diversity*, the typical form of parallel redundancy for fault tolerance against design faults: the multiple replicas of the database are managed by diverse DBMS products;
- *data diversity* [21]: thanks to the redundancy in the SQL language, a sequence of one or more SQL statements can be "rephrased" into a different but logically equivalent sequence to produce redundant executions, reducing the risk of a failure being repeated when the rephrased sequence is executed on the same or another replica of even the same DBMS product. Two of the present authors have reported elsewhere [22] on a set of "rephrasing rules" that would tolerate at least 60% of the bugs examined in our studies. Another possibility is varying the "hints" to the "query optimizer" of the DBMS that are included with SQL statements.
- *configuration diversity* (which can be seen as a special form of data diversity). DBMS products have many configuration parameters, affecting e.g. the amount of system resources they can use (amount of RAM and/or the "page size" used by the database), or the degree of optimization to be applied to certain operations: given the same database contents, varying these parameters between two installations can produce different implementations of the data and the operation sequences on them, and thus decrease the risk of the same bug being triggered in two installations of the same DBMS product by the same sequence of SQL statements.

These precautions can in principle be combined (for instance, data diversity can be used with diverse DBMS products), and implemented in various ways, including manual application by a human operator.

Among the above forms of diversity, design diversity appears the most likely to avoid coincident failures in redundant executions, but it may impose substantial limitations or design costs. In the first place, OTS DBMS products, even if they nominally implement the operations of the standard SQL language, in practice use different "dialects": they use different syntax for commands that are semantically the same (this problem can be solved via automatic, on-the-fly translation); more importantly, each offers extra, non-standard features, which would require either more complex translation ("rephrasing" of statements, mentioned above as a form of data diversity, can be useful to overcome problems with translation, as we have shown [22]), and/or clients to be limited to using a common subset

[4] We will use the term "statement" to refer to the SQL requests that are sent to the DBMS product. These may be read or write *data manipulation language* (DML) statements or *data definition language* (DDL) statements

[5] Simple reasonableness or "safety" checks are often available, but have limited efficacy against some failure scenarios. E.g., reasonableness checks may prevent the posting of incredibly large movements in a company's accounts, yet allow many small systematic errors, allowing large cumulative errors to build up before the problem comes to light.

[6] The name introduced by Gray [20] for bugs that are difficult to reproduce, as they only cause failures under special conditions: "strange hardware conditions (rare or transient device fault), limit conditions (out of storage, counter overflow, lost interrupt, etc.) or race conditions", "Bohrbugs" instead appear to be deterministic (the failures they cause are easy to reproduce in testing).

among the features of the diverse DBMS products. In addition, many aspects of database operation are specified in a non-deterministic fashion, making the goal of ensuring consistency among replicas difficult even with same-product replication, and more so with diverse replication.

A special case of design diversity is using successive releases of the same DBMS product. This will avoid or greatly reduce the problems due to "dialect" differences. It may be expected to tolerate fewer faults, since the successive releases will share large portions of their code, including some bugs; but it may be attractive for "smoothing out" upgrades which may otherwise cause peaks of unreliability in a database installation, due to the new faults introduced, and at the same time evaluating the new release to decide when it has reached sufficient dependability to be used alone. Similar practices have been applied for embedded and safety critical systems [23], [24].

We now discuss briefly the architectural options available in designing automated fault tolerance solutions with some form of diversity applied to OTS DBMS products. A basic "black box" replication architecture delegates the management of redundancy to a layer of middleware, as in Fig. 1, so that the multiple DBMS products appear to clients as a single server. There may be any number of channels, though typical values would be one (using "time redundancy" – repeating the execution on the single DBMS product – when needed), two or three (the minimum that allows error masking through voting). We will normally refer to systems with two replicas, unless otherwise noted.

This basic architecture can be used for various fault tolerance strategies, with different trade-offs between coverage for various types of failures, performance, ease of integration etc [25]. The most serious design issues concern ensuring replica determinism, for those replication schemes that require it. The difficulty is that each DBMS product has its own concurrency control strategy, and these are non-deterministic and may be different between products. Proprietary replication solutions deal with this problem by using knowledge of the implementation of a DBMS product. For a middleware layer managing generic OTS products, this is more difficult, especially since commercial vendors may keep these details secret. The middleware can instead artificially serialize statements in the same way on all replicas [26], [27]. There are performance costs, but these will be acceptable for many installations, though intolerable on others, depending on the amount and pattern of write transactions in a specific installation.

A separate requirement, easier to satisfy, is that any voting/comparison algorithm need to allow for "cosmetic" differences between equivalent correct results issued by different DBMS products, e.g. differences in the padding blank characters in character strings or different numbers of digits in the representations of floating point numbers. Trade-offs exist here between embedding in the algorithm more knowledge about the idiosyncrasies of each specific product, and keeping it more generic at the cost of possibly lower coverage.

2.3 Design Options for Fault Tolerance via Diverse Replication

2.3.1 Detection of Server Failures

Erroneous responses to read statements can be detected by comparing the outputs of the channels, detecting those non-self-evident failures that cause some discrepancy between these outputs.

Both design diversity and data diversity increase the chance of detection, compared to simple replication. Replica determinism is necessary, i.e., discrepancies between correct results must be rare, as they may cause correct results to be treated as erroneous, and thus a performance penalty. Self-evident failures are detected as in non-diverse servers, via the server's error messages (i.e., via the existing error detection mechanisms of the DBMS products) and time-outs.

Erroneous updates to the databases that will only cause output discrepancies in the future are also a concern. To detect them, the middleware can compare the contents of the database replicas, via the standard read commands of the DBMS products. There is a degree of freedom in how much should be compared, allowing latency/performance trade-offs. The middleware could just ask each DBMS product for the list of the records modified in each write operation, and then read and compare their contents. In principle, though, a buggy DBMS product could omit some changed records from the list it returns. So, a designer could decide to compare a superset of the data that appear to be affected, trading off time for better error detection.

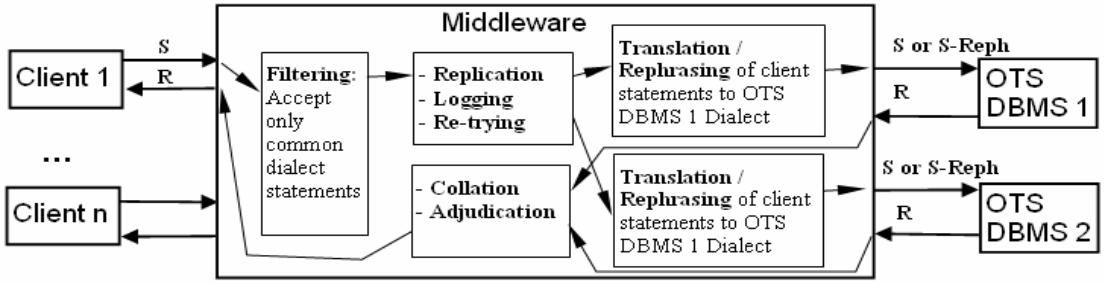

Fig. 1 - A stylized design of a fault-tolerant database server with two *channels*. Each channel includes an installation of an *OTS DBMS* product (these may be the same or different products, including different releases of the same product) and a replica of the database. The *middleware* must ensure connectivity between the *clients* and the DBMS products, some filtering of the statements sent by clients (e.g. returning error messages to the client for statements that are not supported by both the underlying OTS DBMS products), replication and concurrency control, management of fault tolerance (error detection; error containment, diagnosis and correction; state recovery), as well as *translation* of SQL statements ("S" in the figure) sent by the clients to the dialects of the respective OTS DBMS products (translation may be done in off-the-shelf add-on components). Support for "data diversity" through "*rephrasing*" may also form part of the same components which perform translation: *rephrasing* rules will produce rephrased versions – "S-reph" in the figure –of the statements sent by clients. The middleware must also *adjudicate* the results – "R" in the figure – from the OTS DBMS products and return a result to the client[s].

Another trade-off is possible between error latency and the overhead imposed by the fault-tolerant operation: error detection can be scheduled in a more or less pessimistic mode. In the most pessimistic mode, at each operation the middleware performs all its comparisons before forwarding to the client the response from the DBMS product[s]. More optimistically, it can forward most responses immediately, and run the checks in parallel with the subsequent operation of the client and DBMS products. A natural synchronization point is at transaction commit: the middleware only allows the transaction to commit if it detected no failures.

In addition, the middleware can use slack capacity for a background audit task, comparing the complete contents of the database replicas.

2.3.2 Error Containment, Diagnosis and Correction

Error containment is tightly linked to detection. For read statements, the middleware receives multiple responses for each statement sent to the diverse channels, one from each of them, and must return a single response to the client. In general, the middleware will present to the client a DBMS product failure as a correct but possibly delayed response (masking), or as a self-evident failure (crash - the behavior of a "fail-silent" fault-tolerant server; or an error message - a "self-checking" server). DBMS product failures can be masked to the clients, if the middleware can select a result that has a high enough probability of being correct:

- if more than two redundant responses are available, it can use majority voting to choose a consensus result, and to identify the failed replica which may need a recovery action to correct its state.
- with only two redundant channels, if they give different results the middleware cannot decide which one is in error. A possibility is not to offer masking, but simply a clean failure to be followed by manual diagnosis of the problem. Alternatively, additional redundant execution can be run by replaying the statements, possibly with "data diversity", i.e., rephrasing the statements [22].

Depending on how redundant executions are organized, the middleware may need to resolve rather complex scenarios, e.g., two diverse DBMS products, A and B, may give different responses upon first submission for a read statement, while upon resubmission of a rephrased statement A produces an error message and B a result matching A's previous result; but this is a standard adjudication problem [28], [29], [30] for which the design options and trade-offs are well known.

Again, the need for replica determinism is the main design issue with these schemes.

2.3.3 State Recovery

Besides selecting probably correct results, adjudication will identify probably failed channels in the fault-tolerant database server (diagnosis). This improves availability: the middleware can selectively direct recovery actions at the channel diagnosed as having failed, while letting the other channel(s) continue providing the service.

The state of a channel can be seen as composed of the state of permanent data in the database and that of volatile data in the DBMS product's variables. For erroneous states of the latter, since the middleware cannot see the internal state of each executing DBMS product, some form of "rejuvenation" [31] must be applied, e.g. stopping and restarting the DBMS product.

As for state recovery of the database contents, it can be obtained:

- via standard backward error recovery – rollback followed by retry of logged write statements –, which will sometimes be effective (failures due to Heisenbugs), at least if the failures did not violate the ACID properties in the affected transactions. "Data diversity" will extend the set of failures that can be recovered this way. To command backward error recovery, the middleware can use the standard database transaction mechanisms: aborting the failed transaction and replaying its statements may produce a correct execution. Alternatively or additionally, it can use checkpointing [32]: the middleware orders the states of the database replicas to be saved at regular intervals (by database "backup" commands: e.g., in PostgreSQL the pg_dump command). After a failure, a database replica is restored to its last checkpointed state and the middleware replays the sequence of (all or just write) statements since then (the redo log provided in some DBMS products cannot be used because it might contain erroneous writes). For finer granularity of recovery, the checkpoint-rollback mechanism can be used within transactions: this allows the handling of exceptions within transactions, and should be applied when using data diversity through "rephrasing" [22];
- additionally, diversity allows one to achieve forward recovery by essentially copying the state of a correct database replica into the failed one (similarly to [33]). Since the formats of database files differ between the DBMS products, the middleware would need to query the correct channel[s] for their database contents and command the failed channel to write them into the corresponding records in its database, similar to the solution proposed in [34]. This would be time-consuming, perhaps to be completed off-line, but a designer can use multi-level recovery, in which the first step is to correct only those records that have been found erroneous on read statements.

During any recovery phase, the fault-tolerant server would work with reduced redundancy. A two-channel fault-tolerant server would be reduced to a non-fault-tolerant configuration. Trade-offs open to the designer involve the duration of the recovery phase (it can be shortened by more efficient algorithms or by reducing the extent of the state that is checked/corrected), and the degree of conservatism applied during non-fault-tolerant operation.

3 Our Studies of Bug Reports for Off-The-Shelf DBMS Products

3.1 Generalities

We use the following terminology. The known bugs for the OTS DBMS products are documented in bug report repositories (i.e. bug databases, mailing lists etc). Each *bug report* contains a description of the bug and a *bug script* for reproducing the *failure* (the erroneous behavior that the reporter

of the bug observed). The bug script may come with indications on the database states that are preconditions for the failure (e.g., in the form of statements to issue for the database to reach one such state), plus the statements (and values for their parameters) which reproduce the *failure*. In our study we collected these bug reports and *ran* the bug scripts on installations of each of the DBMS products we used (we will use the phrase "*running a bug*" for the sake of brevity).

What constitutes an individual bug is of course not definable by any a priori rule [35, Section 2]: people characterize a bug in terms of the apparent mistakes made by the developers, of code changes required to fix it, and/or of circumstances on which the software fails. We define a "demand" as the complete circumstances (i.e. an initial state plus a series of statements) that would cause failure. A bug report does not necessarily identify the whole set of demands (the "failure region") on which the product fails and would no longer fail if the bug were corrected. When running a bug script, we usually tested all DBMS products in our study on at least one demand (the same for all) mentioned in the bug report, and listed the bug as present in all DBMS products that failed on that demand. In some cases, we also tested the DBMS products with other similar demands - variations of the statements and/or parameters specified in a bug script. We did this when a bug script did not seem to trigger a failure in the DBMS product to which it related, to check whether the bug did appear to be present, but the reporter may have been imprecise in characterizing the conditions for triggering it; and when a bug script caused failures in more than one DBMS products, to study and compare the "failure regions" identified in the two products, especially to determine their overlap and whether the DBMS products fail identically throughout them.

3.1.1 Reproducibility of Failures

As mentioned earlier, DBMS products offer features that extend the SQL standard, and these extensions differ between products. Bugs affecting these extensions literally cannot exist in a DBMS product that lacks them. We called these bugs "dialect-specific". For example, Interbase bug 217138 [36] affects the use of the UNION operator in VIEWs, which PostgreSQL 7.0 VIEWs do not offer, and thus cannot be run in PostgreSQL 7.0: it is a dialect-specific bug.

Another reproducibility issue arises when a bug script does not cause failure in the DBMS product for which the bug was reported. We called these bugs 'Unreproduced' bugs. They may be Heisenbugs [20] or bugs reported without enough detail for reproducing them. Compared to our preliminary report [9], we have been able to trigger some more previously 'Unreproduced' bugs (and thus we report updated statistics): by running variations of the incomplete bug script, as mentioned above; or thanks to more complete bug scripts posted after our collection period or to mailing lists other than the main repository for the respective DBMS product.

3.1.2 Classifications of Failures

We ran each bug first on the DBMS product for which it was reported, and then (after translating the script into the appropriate SQL dialect[s]) on the other DBMS product[s]. We classified bugs into Reproduced and Unreproduced and into dialect-specific and non-dialect-specific bugs, as explained previously, and failures into different categories that would require different fault tolerance mechanisms:

Engine Crash failures: a crash or halt of the core engine of the DBMS product.

Incorrect Result failures, which are not engine crashes but produce incorrect outputs: the results do not conform to the DBMS product's specification or to the SQL standard.

Performance-related failures. We classified as performance failures: i) failures that are so classified in bug reports; ii) failures observed by us if either the DBMS product clearly "hung" or, whatever the observed latency, the bug script generated a query plan indicating potential performance problems, e.g. with an un-utilized column "index" in a SELECT statement using that column.

Other failures: e.g. security related failures, such as incorrect privileges for database objects (tables, views etc.)

We further classified failures according to their detectability by a client of the DBMS products:

Self-Evident Failure: engine crash failures, internal failures signaled by DBMS product exceptions (error messages) or performance failures

Non-Self-Evident Failures: incorrect result failures without DBMS product exceptions, with acceptable response time.

For clients with access to at least two diverse DBMS products the failures would be:

Divergent failures: any failures where DBMS products return different results. All failures affecting only one out of two (or at most n-1 out of n) DBMS products are divergent. Even if all fail but 'differently' the failure will still be divergent.

Non-divergent failures: the ones for which two (or more) DBMS products fail with identical symptoms. For some bugs, all demands we ran caused non-divergent failures, for others only some demands did. In the tables that follow we use the labels "non-divergent – *all demands*" and "non-divergent – *some demands*" for these two cases.

All the *divergent* or *self-evident* failures are *detectable* by a client of the database server when at least two replicas of the database are available, on different DBMS products. Failures that are *non-divergent* and *non-self-evident* are *non-detectable*.

3.2 The First Study

3.2.1 Description of the Study

In our first study [9] we used four DBMS products: two commercial (Oracle 8.0.5 and Microsoft SQL Server 7, without any service packs applied) and two open-source ones (PostgreSQL Version 7.0.0 and Interbase Version 6.0). Interbase, Oracle and MSSQL were all run on the Windows 2000 Professional operating system; PostgreSQL 7.0.0 (not available for Windows) was run on RedHat Linux 6.0 (Hedwig). We use the following abbreviated identifiers (for PostgreSQL we include the release number in the identifier since we will report later on results of one of its later releases):

- *PG 7.0* - for PostgreSQL 7.0.0
- *IB* - for Interbase 6.0

- *OR* - for Oracle 8.0.5
- *MS* - for Microsoft SQL Server 7

For each of these DBMS products there is an accessible repository of reports of known bugs. We collected the IB bugs from SourceForge [37], the PG 7.0 bugs from its mailing list, [38], MS bugs from its service packs site [39] and OR bugs from the Oracle Metalink [40].

We only used bugs that caused failure of a DBMS product's core engine. Other bugs, e.g. causing failures of a client application tool, various connectivity (JDBC/ODBC etc.) or installation-specific bugs were not included in the study, because in a future fault-tolerant architecture these functions would be provided by the middleware.

For each reported bug, we attempted to run the corresponding bug script. Full details are available in [36] (and also provided as Supplement A).

3.2.2 Detailed Results

In total, we included in the study 181 bug reports: 55 for IB, 57 for PG, 51 for MS and 18 for OR. None of these bugs was reported for more than one DBMS product. Out of these 181 bugs, 70 were dialect-specific (could be run in only one of the four DBMS products); 58 could be run in all four DBMS products; 26 could be run in only two DBMS products and 27 in only three DBMS products.

Table 1 contains the results of the first study. The structure of the table is as follows. Each gray column lists the results produced when the bugs reported for a certain DBMS product were run on that DBMS product. For example, we collected 55 known IB bugs, of which, when run on our installation of IB, 8 did not cause failures (Unreproduced). The 47 bugs that caused failures are further classified in the part of the column below the double horizontal line, after the "Failure observed" row. Performance failures and engine crashes are self-evident. Incorrect Result failures and "Other" failures can be self-evident or non-self-evident, depending on whether the DBMS product gives an error message.

To the right of the gray column, three columns present the results of running the IB bugs on the other three DBMS products. For example, we can see that out of 55 IB bugs, 24 cannot be run in PG 7.0 (dialect-specific bugs). Of the other 31, which we ran in PG 7.0: 3 are classified as "Undecided Performance" meaning that the bug report indicated a "performance failure" but we could not decide, from the query plan and observed response time, whether a performance failure also occurs in PG 7.0; 27 did not cause a failure in PG 7.0; only 1 caused a failure in both IB and PG 7.0. The table shows that this particular failure was a non-self-evident incorrect result. Details about the bugs causing coincident failures were given in [9].

As for the failure types, we can see that most of the bugs for each DBMS product cause Incorrect Result failures. The percentage of non-self-evident failures is also high: they range from 44% for MS to 66% for IB. Engine crashes are less frequent: they range from 13% for MS to 21.5% for OR.

3.2.3 Implications for Fault Tolerance: Two-Version Combinations

We now look more closely at the two-version combinations of the four different DBMS products. We want first to find out how many of the coincident failures are *detectable* (i.e. *divergent* or *self-evident*) in the two-version systems. Table 2 contains a summary of the results on each of the six possible two-version combinations[7].

Only twelve coincident failures occurred (note that there were thirteen bugs that caused failures in a different DBMS product than the one for which they were reported (as detailed in Table 1); one bug (MS bug report 56775) [36], although reported for MS, did not cause failure in MS (Unreproduced) but did cause failure in PG 7.0); only four of these twelve are non-detectable. We can see that diversity allows detection of failures for at least 95% of these bugs (41

TABLE 1. STUDY 1: RESULTS OF RUNNING THE BUG SCRIPTS ON ALL FOUR DBMS PRODUCTS.
ABBREVIATIONS: IB – INTERBASE 6.0; PG 7.0 - POSTGRESQL 7.0.0; OR – ORACLE 8.0.5; MS – MICROSOFT SQL SERVER 7.

		IB	PG 7.0	OR	MS	PG 7.0	IB	OR	MS	OR	IB	MS	PG 7.0	MS	IB	OR	PG 7.0
Total bug scripts		55	55	55	55	57	57	57	57	18	18	18	18	51	51	51	51
Bug script cannot be run (Functionality Missing)		n/a	24	21	17	n/a	33	27	24	n/a	14	14	13	n/a	36	35	31
Total bug scripts run		55	31	34	38	57	24	30	33	18	4	4	5	51	15	16	20
Undecided performance		0	3	3	3	0	0	0	0	0	0	0	1	0	3	4	2
No failure observed		8	27	31	33	5	24	30	31	4	4	4	3	12	11	12	12
Failure observed		47	**1**	0	**2**	52	0	0	**2**	14	0	0	**1**	39	**1**	0	**6**
Types of failures	**Poor Performance**	3	0	0	0	0	0	0	0	1	0	0	0	6	0	0	0
	Engine Crash	7	0	0	0	11	0	0	0	3	0	0	0	5	0	0	0
	Incorrect Result – **Self-evident**	4	0	0	**1**	14	0	0	**1**	3	0	0	0	10	0	0	**6**
	Incorrect Result – **Non-self-evident**	23	**1**	0	**1**	20	0	0	**1**	7	0	0	**1**	17	**1**	0	0
	Other – **Self-evident**	2	0	0	0	2	0	0	0	0	0	0	0	1	0	0	0
	Other – **Non-self-evident**	8	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0

[7] Here we only include bugs (reported for any of the four DBMS products) that could be run on both DBMS products, i.e. we exclude dialect-specific bugs. For instance, Table 2 shows that a total of 71 bugs could be run on both IB and PG 7.0. In detail, 31 of these were reported for IB and 24 for PG; these two numbers can be deduced from Table 1. The remaining 16 were bugs of either OR or MS which could be run on both IB and PG 7.0 – these numbers are not directly deducible from Table 1 due to some bugs being dialect-specific for one DBMS product but not another; they can however be obtained from [36]).

TABLE 2. STUDY 1: SUMMARY OF RESULTS FOR THE TWO-VERSION COMBINATIONS.
ABBREVIATIONS: *S.E. – SELF-EVIDENT FAILURE*; *N.S.E. - NON-SELF-EVIDENT FAILURE.*

Pairs of DBMS Products	Total number of bug scripts run	Bugs scripts causing failure (in at least one DBMS product)	One out of two DBMS products failing		Both DBMS products failing						
			s.e.	n.s.e.	Non – Divergent				Divergent		
					All Demands		Some Demands		1 s.e. & 1 n.s.e.	2 s.e.	2 n.s.e.
					s.e.	n.s.e.	s.e.	n.s.e.			
IB + PG 7.0	71	49	22	26	0	**1**	0	0	0	0	0
IB + OR	69	32	11	21	0	0	0	0	0	0	0
IB + MS	78	43	17	23	**1**	**2**	0	0	0	0	0
PG 7.0 + OR	72	33	16	16	0	0	0	0	0	0	**1**
PG 7.0 + MS	85	48	20	21	0	**1**	0	0	**3**	**3**	0
OR + MS	80	18	11	7	0	0	0	0	0	0	0

out of 43, for the IB+MS pair). Moreover, it would support masking and forward recovery (following the self-evident failure of a single channel) for a fraction of bugs varying between 11/32 (34%) for the IB+OR pair) and 11/18 (61%) for the OR+MS pair. More details on these bugs are in [9] and [36].

3.3 The Second Study

3.3.1 Description of the Study

To repeat the study on later releases of DBMS products, we collected 92 new bug reports for the later releases of the open-source DBMS products: PostgreSQL 7.2 and Firebird 1.0 (abbreviated as PG 7.2 and FB respectively; Firebird is the open-source descendant of Interbase 6.0. The later releases of Interbase are issued as closed-development by Borland). We excluded the closed-development DBMS products as most of their bug reports lacked the bug scripts needed to trigger the faults. But we still translated the new bug scripts of bugs reported for the open-source DBMS products into the dialects of the closed-development ones, and ran them in the releases used in our first study (Oracle 8.0.5 and MSSQL 7.0). The results of the second study are shown in Table 3 (for full details see [36]). The classifications of faults and failures are as defined in Section 3.1.

Incorrect results are still the most frequent failures. Engine crashes are slightly more frequent than in the first study but still no more than 22.2%. The number of non-self-evident failures is lower than in the first study: 35% for PG 7.2 and 53% in FB. The number of bugs causing coincident failures was again low: in the second study we observed a total of 5 coincident failures. None of the bugs caused failures in more than two DBMS products. The coincident failures are detailed in Section 3.3.3.

3.3.2 Implications for Fault Tolerance: Two-Version Combinations

Table 4 shows the results of the two-version combinations of the 4 DBMS products used in the second study. None of the bugs caused non-detectable failures for all demands. Here there are some bugs that are "non-divergent" for "some demands" only. One caused non-detectable failure only for a few demands in the common failure region, but detectable failure on the others. Three bugs caused self-evident failures in both DBMS products and one caused non-self-evident failure in one and self-evident failure in the other.

So, diversity allows detection of failures for all these bugs. It would allow masking and forward recovery (following the self-evident failure of a single channel) for a fraction of bugs varying between 11/28 (39%) for the FB+MS pair and 12/20 (60%) for the PG 7.2+MS pair.

3.3.3 Common Bugs

It is interesting to describe in some more detail some of the bugs that caused coincident failures, listed in Table 5, and speculate about the probable frequency and severity of the failure observed (similar accounts for bugs in Study 1 are in our preliminary report [9]).

Arithmetic-related bugs

Firebird bug 926001 [36] causes non-self-evident failure in both FB and PG 7.2 when the DBMS product is asked to add two values of type Timestamp (a timestamp value contains both date and time information). Due to rounding errors, FB always gives a result that is 1 second less than the correct result, whereas PG 7.2 adds the dates but not the time of the second timestamp value (i.e. it treats the operation as $Timestamp_1 + Date_2$). The failure rate for this bug would be highest in applications that require high precision arithmetic computations with timestamp datatypes. On most demands the erroneous results of the two DBMS products would be different: the failure is non-divergent only for some (probably rare) demands.

FB bug 926624 [36] causes a crash in both FB and MS. The crash is due to a stack overflow from attempting to use in the column part of the SELECT statement an arithmetic expression longer than: 8000 characters in FB; 2834 characters in MS. Therefore FB fails for a smaller set of demands than MS. The expected correct behavior is for the DBMS product to process the statements, or to give an error message that warns the user of the maximum allowed expression length. The failure rate for this bug would probably be low for most installations, as SELECT statements would seldom contain such long arithmetic expressions.

Miscellaneous bugs

FB bug 910423 [36] causes failure in both FB and MS. Fig. 2 shows the demands for which they fail. The failure consists in allowing the datatype of a table column to be changed from integer to string even when the string type is specified to be shorter than needed to hold the data already stored in the column. The expected correct behavior is for the DBMS product to refuse (with an error message) to change the datatype of either any column that already contains data, or

TABLE 3. STUDY 2: RESULTS OF RUNNING THE BUG SCRIPTS OF FB AND PG ON ALL FOUR DBMS PRODUCTS.
ABBREVIATIONS: FB – FIREBIRD 1.0; PG 7.2 - POSTGRESQL 7.2; OR – ORACLE 8.0.5; MS – MICROSOFT SQL SERVER 7.

			FB	PG 7.2	OR	MS	PG 7.2	FB	OR	MS
Total bug scripts			43	43	43	43	49	49	49	49
Bug script cannot be run (Functionality Missing)			n/a	12	15	13	n/a	29	29	30
Total bug scripts run			43	31	28	30	49	20	20	19
Undecided performance			0	1	2	1	0	2	2	0
No failure observed			4	29	26	27	4	17	18	18
Failure observed			39	**1**	0	**2**	45	**1**	0	**1**
Types of failures	**Poor Performance**		4	0	0	0	5	0	0	0
	Engine Crash		6	0	0	**1**	10	0	0	**1**
	Incorrect Result	**Self-evident**	7	0	0	0	13	**1**	0	0
		Non-self-evident	20	**1**	0	**1**	15	0	0	0
	Other	**Self-evident**	1	0	0	0	1	0	0	0
		Non-self-evident	1	0	0	0	1	0	0	0

TABLE 4. STUDY 2: SUMMARY OF RESULTS FOR THE TWO-VERSION COMBINATIONS.
ABBREVIATIONS: *S.E. – SELF-EVIDENT FAILURE*; *N.S.E. - NON-SELF-EVIDENT FAILURE*.

Pairs of DBMS Products	Total number of bug scripts run	Bugs scripts causing failure (in at least one DBMS product)	One out of two DBMS products failing		Both DBMS products failing						
			s.e.	n.s.e.	Non – Divergent				Divergent		
					All Demands		Some Demands		1 s.e. & 1 n.s.e.	2 s.e.	2 n.s.e.
					s.e.	n.s.e.	s.e.	n.s.e.			
FB + PG 7.2	51	47	26	19	**1**	0	0	**1**	0	0	0
FB + OR	46	25	10	15	0	0	0	0	0	0	0
FB + MS	46	28	11	15	0	0	**1**	0	**1**	0	0
PG 7.2 + OR	47	21	13	8	0	0	0	0	0	0	0
PG 7.2 + MS	47	20	12	7	0	0	**1**	0	0	0	0

TABLE 5. BUGS THAT CAUSE COINCIDENT FAILURES

For which DBMS product was the bug reported? \ On which additional DBMS product was failure observed?	FB	PG	OR	MS
FB	N/A	**1** - (Bug ID 926001)	0	**2**– (BugIDs 910423, 926624)
PG	**1** (Bug report date 16/05/2003)	N/A	0	1 – (BugID 847)

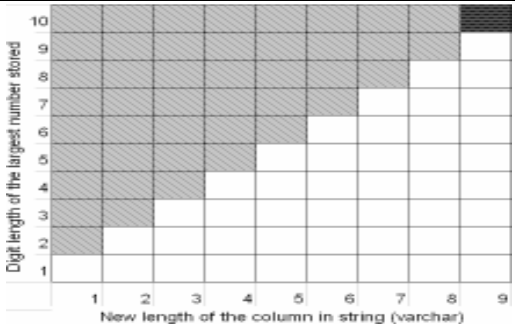

Fig. 2. FB bug 910423: demands on which MS fails (light gray shaded boxes) and demands for which both FB and MS fail (dark gray).

at least those containing data that wouldn't fit in the new length specified. If a client later tries to read the column affected, the two DBMS products react differently: FB responds with an error message (self-evident failure), while MS returns a '*' symbol. We have therefore classified the failure as divergent. As shown, MS actually fails on a superset of the demands on which FB does. It is difficult to conjecture how often applications change the datatypes of columns and hence the likely failure rates for this bug. The severity of this failure is different in the two DBMS products. FB does not lose the data stored in the column: if you just change the type again to a long enough string (>=10 in the example above) then the data can again be read. MS instead truncates the data item to the new length, so that it is irremediably lost.

PG 7.2 bug 847 [36] causes failure in both PG 7.2 and MS. PG 7.2 allows the creation of exceptions that return a message longer than 4000 characters, but then crashes if the exception is raised. The correct behavior is for a DBMS product to give an error message once its maximum length for exception messages is reached: either when the exception is defined or when attempting to raise the exception. The same problem occurs in MS, but the threshold message length is even smaller (440 characters), and thus failures would be more frequent.

The PG 7.2 bug reported on 16 May 2003 (with no ID in the PG 7.2 mailing list [36]) causes an error message in PG 7.2 and FB, although no error exists. The bug script is given below. The UPDATE statement causes the contents of the database to violate the UNIQUE CONSTRAINT (a constraint over a set of columns requiring that no two values for different rows be equal) at some intermediate stage, although the final state does not violate it:

```
CREATE TABLE TEST2 (V1   INT, V2   INT, CONSTRAINT
UQ_TEST UNIQUE (V1,V2));
INSERT INTO TEST2 VALUES (0,0);
INSERT INTO TEST2 VALUES (0,1);
INSERT INTO TEST2 VALUES (0,2);
UPDATE TEST2 SET V2=V2+2;
```

Violation of UNIQUE KEY constraint "UQ_TEST" on table "TEST2"

OR and MS correctly execute the script without error messages; PG 7.2 and FB perform the UNIQUE CONSTRAINT checks at intermediate states (in this case after

each row is updated), which causes the exception to be raised. The failure is not specific to this bug script. It can be triggered with any UNIQUE CONSTRAINT on integer, real or float datatypes affecting single or multiple columns, whenever an update is attempted that will (at an intermediate step during the execution) set a value of a row to that of an existing row in the table, although at the end of the execution of the statement no violations would be present. On every set of parameter values that we tried, either both DBMS products failed or neither did. The failure rate for this bug is expected to be relatively high in update-intensive applications if UNIQUE CONSTRAINT is used.

3.4 Newer vs. Older Releases (Open-Source DBMS Products)

We now look more closely at those DBMS products that were used in both studies (i.e. the two open-source products). We ran all the new bugs reported for the newer releases on the older releases, to check how many already existed there. The results are in the leftmost eight columns of Table 6 (full details are in [36]).

The structure of Table 6 is the same as that of Table 1 and Table 3. We can see that 33 bugs reported for FB also cause failure in the older release IB. Of the six that do not cause failures in IB, four were Unreproduced in FB. So only 2 bugs that caused failure in FB (the new release) appear to be new bugs, introduced in functionalities that used to work correctly. The reason might be that FB 1.0 was mainly a bug fix release, with no major enhancements, which probably also reduced the number of new problems that could be introduced.

The situation is different for PG 7.2, which featured many more enhancements, for example the support for OUTER JOINS in SELECT statements. We can see that 13 of the bugs reported for PG 7.2 cannot be run at all in the older release (they affect newly added functionality) and, more importantly, 17 of the other 36 bugs do not cause failures in the older release (2 of them are Unreproduced in both releases). This means that the development of the newer release introduced many bugs in functionality that used to work correctly in the old release.

We also ran the old bugs in the new releases of the DBMS products to see how many had been fixed. The results are in the rightmost eight columns of Table 6 (full details are in [36]).

More PG 7.0 bugs were fixed in PG 7.2 than the IB bugs fixed in FB. This high number of fixes, with the attendant risk of new bugs, may be one cause of the relatively many PG 7.2 bugs affecting pre-existing PG 7.0 functionalities(*cf.* the first half of Table 6).

3.4.1 Implications for Fault Tolerance: The Open-Source Two-Version Combinations

Table 7 shows the results for all the bugs, from both studies, that could be run on the various open-source combinations.

The first two rows concern the pairs of different releases of the same DBMS product. For PostgreSQL, we see that out of 93 bugs that caused failure in at least one of the releases, 7.0 or 7.2, only 35 cause failures in both; 58 bugs cause failures in only one of the releases. So, using diverse releases of the same DBMS product in a fault-tolerant configuration, as discussed in Section 2, does provide some protection against upgrade problems and can help to assure higher dependability. However there are still many bugs causing failures in both releases of the same DBMS product:
- 57 in Interbase/Firebird
- 35 in PostgreSQL.

This can be compared with the four DBMS product pairs using different DBMS products (last four rows in Table 7), where we get at most 2 bugs that cause coincident failures. This is because:
- The IB 6.0 bug 223512(2) which caused non-divergent coincident failure in IB 6.0 and PG 7.0, has been fixed in the newer releases of both DBMS products.
- The FB 1.0 bug 926001 [36], which causes coincident failure in the new releases FB 1.0 and PG 7.2, did not cause a failure in IB 6.0 and cannot be run in PG 7.0 (dialect-specific).

The main conclusion is to confirm the high level of fault diversity between these DBMS products, and thus potential advantages of a diverse redundant fault-tolerant server. Using different releases of the same DBMS product would also yield dependability gains, but these seem nowhere near as high as the gains that can be achieved by using diverse DBMS products.

4. Discussion

The results presented in Section 3 are intriguing and suggest that assembling a fault-tolerant database server from two or more of these OTS DBMS products could yield large dependability gains. But they are not definitive evidence. Apart from the sampling difficulties caused e.g. by lack of certain bug scripts, it is important to clarify to what extent our observations allow us to predict such gains. We gave a detailed discussion of the difficulties in [9]. In summary:
- the reports available concern *bugs*, not how many *failures* each caused. They do not tell us whether a bug has a large or a small effect on reliability, although the unknown faults – those that have not yet caused failures – would tend to have stochastically lower effect on reliability than those that did cause failures. A better analysis would be obtained from the actual failure reports (including failure counts), if available to the vendors. However, vendors are often wary of sharing such detailed dependability information with their customers;
- less than 100% of the failures that occur, and thus also of the bugs causing them, are reported. However, blatant failures are more likely to be reported than subtle (arguably more dangerous) failures. Therefore failure *under*reporting probably causes a bias towards *under*estimating the frequency of these subtler failures for which diversity would help;
- an organization needs to predict the dependability of its specific installation[s] of a diverse server, compared to a single DBMS product, which depends on the organization's (or each specific installation's) *usage profile*, which differs – perhaps markedly – from the aggregate profile of the user population which generated the bug reports.

How can then individual user organizations decide

TABLE 6. THE RESULTS OF RUNNING THE NEW SCRIPTS REPORTED FOR FB AND PG 7.2 ON THE OLDER RELEASES (IB AND PG 7.0 RESPECTIVELY) AND THE BUGS REPORTED FOR THE OLD RELEASES ON THE NEW ONES
ABBREVIATIONS: FB – FIREBIRD 1.0; IB – INTERBASE 6.0; PG 7.0 - POSTGRESQL 7.0.0; PG 7.2 - POSTGRESQL 7.2.

		FB	IB	PG 7.2	PG 7.0	PG 7.2	PG 7.0	FB	IB	IB	FB	PG 7.0	PG 7.2	PG 7.0	PG 7.2	IB	FB
Total bug scripts		43	43	43	43	49	49	49	49	55	55	55	55	57	57	57	57
Bug script cannot be run (Functionality Missing)		n/a	4	12	26	n/a	13	29	29	n/a	n/a	24	21	n/a	n/a	33	33
Total bug scripts run		43	39	31	17	49	36	20	20	55	55	31	34	57	57	24	24
Undecided performance		0	0	1	1	0	0	2	2	0	0	3	3	0	0	0	0
No failure observed		4	6	29	16	4	17	17	17	8	33	27	31	5	40	24	24
Failure observed		39	**33**	**1**	0	45	**19**	**1**	**1**	47	**22**	**1**	0	52	**17**	0	0
Types of failures	**Poor Performance**	4	**3**	0	0	5	**1**	0	0	3	**2**	0	0	0	0	0	0
	Engine Crash	6	**6**	0	0	10	**3**	0	0	7	**2**	0	0	11	**2**	0	0
	Incorrect Result – **Self-evident**	7	**6**	0	0	13	**8**	**1**	**1**	4	**2**	0	0	14	**6**	0	0
	Incorrect Result – **Non-self-evident**	20	**16**	**1**	0	15	**5**	0	0	23	**10**	**1**	0	20	**5**	0	0
	Other – **Self-evident**	1	**1**	0	0	1	**1**	0	0	2	**2**	0	0	2	0	0	0
	Other – **Non-self-evident**	1	**1**	0	0	1	**1**	0	0	8	**4**	0	0	5	**4**	0	0

TABLE 7. SUMMARY OF THE RESULTS OF BOTH STUDIES FOR OPEN-SOURCE TWO-VERSION COMBINATIONS
(ABBREVIATIONS: *S.E. – SELF-EVIDENT FAILURE*; *N.S.E.- NON-SELF-EVIDENT FAILURE*)

Pairs of DBMS products	Total number of bug scripts run	Bugs scripts causing failure (in at least one DBMS product)	One out of two DBMS products failing		Both DBMS products failing						
			s.e.	n.s.e.	Non – Divergent				Divergent		
					All Demands		Some Demands		1 s.e. & 1 n.s.e.	2 s.e.	2 n.s.e.
					s.e.	n.s.e.	s.e.	n.s.e.			
FB 1.0 + IB 6.0	157	84	8	19	**24**	**33**	0	0	0	0	0
PG 7.2 + PG 7.0	164	93	33	25	**20**	**15**	0	0	0	0	0
FB 1.0 + PG 7.2	127	65	33	30	**1**	0	0	**1**	0	0	0
FB 1.0 + PG 7.0	106	65	34	30	**1**	0	0	0	0	0	0
IB 6.0 + PG 7.2	127	79	37	41	**1**	0	0	0	0	0	0
IB 6.0 + PG 7.0	106	77	39	37	0	**1**	0	0	0	0	0

whether diversity is a suitable option for them, with their specific requirements and usage profiles? As usual for dependability-enhancing measures, the cost is reasonably easy to assess: costs of the DBMS products, the required middleware, difficulties with client applications that require vendor-specific features, hardware costs, run-time cost of the synchronization and consistency-enforcing mechanisms, and possibly more complex recovery after some failures. The gains in improved reliability and availability (fewer system failures and easier recovery from some failures, to be set against possible extra failures due to the added middleware), and possibly less frequent upgrades, are difficult to predict except empirically. Using ballpark figures may provide useful guidelines: there are studies that suggest that the "Total Cost of Ownership" may exceed the initial investment by more than one order of magnitude, and the cost of recovery from failures is a major part of this [41]. This uncertainty will be compounded, for many user organizations, by the lack of trustworthy estimates of their baseline reliability with respect to subtle failures: databases are used with implicit confidence that failures will be self-evident.

Despite all these uncertainties, for some users our evidence already means that a diverse server is a reasonable and relatively cheap precautionary choice, even without good predictions of its effects. These are users who have: serious concerns about dependability (e.g., high costs for interruptions of service or for undetected incorrect data being stored); applications which use mostly the core features common to multiple off-the-shelf DBMS products (recommended by practitioners to improve portability of the applications); modest throughput requirements for write statements, which make it easy to accept the synchronization delays of a fault-tolerant diverse server.

5. RELATED WORK

5.1 Fault Tolerance in Databases

Fault tolerance in databases has been thoroughly studied and successfully applied in established products. We already mentioned standard database mechanisms such as transaction rollback and retry and checkpointing, which can be used to tolerate faults due to transient conditions. These techniques can be used with or without data replication (discussed in Section 2) in the databases.

5.2 Interoperability Between Databases

Due to the incompatibilities between the SQL "dialects" of different DBMS products we emphasized the need for SQL translators in the middleware of a diverse fault-tolerant server. Similar ideas have been applied for increasing interoperability between DBMS products [42], [43]: the grammar

of a DBMS product is re-defined to make it compatible with that of another DBMS product, while keeping the core DBMS product engine unchanged.

5.3 Design Diversity

Fault tolerance through design diversity has been studied for over 30 years. The literature is vast: the interested reader can refer to survey papers about the effectiveness of design diversity [44], and about design aspects [4]. More recent results on the effectiveness of design diversity include measurements with very large populations of amateur programmers [45], and more detailed probabilistic models on how development affects the reliability of fault-tolerant software [46]. The literature points at substantial reliability gains from diversity, although it cautions on the difficulty of predicting them, since independence of failures between diverse versions should not be expected.

Our study differs from the earlier experimental studies in three main ways:

- we study large software products - DBMS products - rather than the small programs used in the earlier experiments;
- we study samples of known bug reports, not failures observed during testing;
- we study coincident failure points or regions rather than defects in source code; this is different, for instance, from the analysis by Brilliant et al [47] of the causes of coincident failures in the Knight and Leveson experiment [48].

5.4 Empirical Studies of Faults and Failures

The usefulness of diversity depends on the frequency of those failures that cannot be tolerated without it. There have been comparatively few studies.

Gray studied the TANDEM NonStop system (with non-diverse replication) [20]. Over the (unspecified) measurement period, 131 out of 132 faults were "Heisenbugs" and thus tolerated. A later study of field software failures for the Tandem Guardian90 operating system [49] found that 82 % of the reported failures were tolerated. However, the others caused failure of both non-diverse processes in a Tandem process, and thus system failure.

Other related studies concern the determinism and fail-stop properties of database failures, but, like our study, they concern faults rather than failure measurements. A study [50] examined fault reports of three applications (Apache Web server, GNOME and MySQL DBMS product). Only a small fraction of the faults (5-14%) were Heisenbugs triggered by transient conditions, that would be tolerated by simple rollback and retry. However, as the authors point out, the reason why they, like us, found few Heisenbugs, might be that people are less likely to report faults that they cannot reproduce. Using instead fault injection, the same authors also found [51] that a significant number of their injected faults (7%) violated the fail-stop model by writing incorrect data to stable storage. Although this fell to 2% when using the Postgres95 transaction mechanism, 2% is still high for applications with stringent reliability requirements.

5.5 Diversity with Off-The-Shelf Applications

Several research projects have addressed architectures supporting software fault tolerance for OTS software. Some have as their main aim intrusion tolerance, e.g.: HACQIT [52], which demonstrated diverse replication (with two OTS web servers - Microsoft's IIS and Apache) to detect failures (especially maliciously caused ones) and initiate recovery; SITAR [53], an intrusion-tolerant architecture for distributed services and especially COTS servers; the Cactus architecture [3], intended to enhance survivability of applications which support diversity among application modules; DIT [2], an intrusion-tolerant architecture using diversity at several levels (hardware platform, operating system platform, and web servers); the MAFTIA [54] project, which delivered a reference architecture and supporting mechanisms. Others target fault tolerance against mainly accidental faults, e.g.: the BASE approach [55] focuses on supporting state recovery for diverse replicas of components via a common abstract specification of a common abstract state, the initial state value and the behavior of each component; the GUARDS [56] and Chameleon [57] architectures aim at supporting multiple application-transparent fault tolerance strategies using COTS hardware and software components.

6. Conclusions

We have reported two studies of samples of bug reports for four popular off-the-shelf SQL DBMS products, plus later releases of two of them. We checked for bugs that would cause common-mode failures if the products were used in a diverse redundant (fault-tolerant) architecture: such common bugs are rare. For most bugs, failures would be detected (and may be masked) by a simple two-diverse configuration using different DBMS products. In summary:

- out of the 273 bug scripts run in both our studies, we found very few bug scripts that affected two DBMS products, and none that affected more than two;
- only five of these bug scripts caused identical, non-detectable failures in two DBMS products; of these five, one caused non-detectable failures on only a few among the demands affected.

The results of the second study, on later releases of the same products, substantially confirmed the general conclusions of the first study: the factors that make diversity useful do not seem to disappear as the DBMS products evolve.

Using successive releases of the *same* product for fault tolerance also appeared useful, although less so. We found a high level of fault diversity between successive releases of PostgreSQL: most of the old bugs had been fixed in the new release; many of the newly reported bugs did not cause failure (or could not be run at all) in the old release. This special form of design diversity is attractive for users who need the SQL "dialectal" features of a specific DBMS product, but gives less dependability benefits than using different products. With data diversity also a possibility, users have various trade-offs available between the wishes to exploit dialectal features and to get effective diversity.

These results must be taken with caution, as discussed in Section 4, and their immediate implications vary between users. Our evidence suggests that the forms of redundancy and diversity discussed here will improve the dependabil-

ity of DBMS products, perhaps dramatically. For some classes of DBMS installations, diversity could already be recommended as a prudent and cost-effective strategy. Yet, users with "ultra-high-dependability" requirements [58] would still have great difficulty achieving confidence that their requirements are satisfied. Our finding some common faults, however rare, certainly suggests caution. Such users might adopt our proposals, but still retain the database- or client-specific solutions mentioned in Section 2.1.

The topic of diversity with OTS software certainly deserves further study.

The need for middleware is an obstacle for users wishing to try out diversity in their applications. But our results provide a good business case for implementing the required middleware software.

The performance penalty due to controlling concurrency via the middleware would be a problem with write-intensive loads, but not if concurrent updates are rare [59].

Some other interesting observations include:

- there is strong evidence against the fail-stop failure assumption for DBMS products. The majority of bugs reported, for all products, led to "incorrect result" failures rather than crashes (64.5% vs 17.1% in our first study; 65.5% vs 19% in the second), despite crashes being more obvious to the user. Even though these are bug reports and not failure reports, this evidence goes against the common assumption that the majority of failures are engine crashes, and warrants more attention by users to fault-tolerant solutions, and by designers of fault-tolerant solutions to tolerating subtle and non fail-silent failures;
- it may be worthwhile for vendors to test their DBMS products using the known bug reports for other DBMS products. For example, in the first study we observed 4 MSSQL bugs that had not been reported in the MSSQL service packs (previous to our observation period). Oracle was the only DBMS product that never failed when running on it the reported bugs of the other DBMS products;

Future work that is desirable includes:

- statistical testing of the DBMS products to assess the actual reliability gains from diversity. We have run a few million queries with various loads, including ones based on the TPC-C benchmark, observing no failures (however, significant performance gains appear to be possible from using diverse servers [19], [59]). These results may not be particularly surprising, since these benchmarks use a limited set of well-exercised features of SQL servers. It would be interesting to repeat the tests with more varied test loads. However, these studies are likely to be most useful with reference to specific application environments, for which the usage profile can be approximated reasonably well;
- developing the necessary middleware components for users to be able to try out data replication with diverse servers in their own installations. Lack of these components is the main practical obstacle to the adoption and practical evaluation of these solutions. There are signs that some DBMS product vendors may also help with this problem: EnterpriseDB [42] and Fyracle [43] are Oracle-mode implementations based on PostgreSQL and Firebird DBMS engines, respectively. With these solutions the problem with SQL dialects is significantly reduced.

ACKNOWLEDGMENT

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council (EPSRC) via projects DOTS (Diversity with Off-The-Shelf components, grant GR/N23912/01) and DIRC (Interdisciplinary Research Collaboration in Dependability, grant GR/N13999/01) and by the European Union Framework Programme 6 via the ReSIST Network of Excellence (Resilience for Survivability in Information Society Technologies, contract IST-4-026764-NOE). We thank Bev Littlewood, Peter Bishop, David Wright and the anonymous TDSC reviewers (in particular the one who suggested further analyses with one of the bug reports) for their comments on earlier versions of this paper.

REFERENCES

[1] P. Popov, L. Strigini and A. Romanovsky, "Diversity for Off-The-Shelf Components", *Proc. IEEE DSN'00 - Fast Abstracts supplement*, New York, NY, USA, pp. B60-B61, 2000.

[2] A. Valdes, M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saidi, V. Stavridou and T.E. Uribe, "An Architecture for an Adaptive Intrusion-Tolerant Server," in *LNCS 2845 - Selected Papers from 10th International Workshop on Security Protocols '02*, B. Christianson, Crispo, B., Malcolm, J. A., Roe, M., Ed.: Springer, pp. 158-178, 2003..

[3] M.A. Hiltunen, R.D. Schlichting, C.A. Ugarte and G.T. Wong, "Survivability Through Customization and Adaptability: The Cactus Approach", *Proc. DARPA Information Survivability Conference & Exposition*, 2000.

[4] L. Strigini, "Fault Tolerance Against Design Faults," in *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*, H. Diab and A. Zomaya, Eds.: J. Wiley & Sons, pp. 213-241, 2005.

[5] P. Popov, L. Strigini, S. Riddle and A. Romanovsky, "Protective Wrapping of OTS Components", *Proc. 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction*, Toronto, Canada, 2001.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil and P. O'Neil, "A Critique of ANSI SQL Isolation Levels", *Proc. ACM SIGMOD Int. Conf. on Management of Data*, San Jose, CA, USA, 1995.

[7] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil and D. Shasha, "Making Snapshots Isolation Serializable", *ACM TODS*, 30(2), pp. 492-528, 2005.

[8] F. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors", *ACM TOCS*, 2(2), pp. 145-154, 1984.

[9] I. Gashi, P. Popov and L. Strigini, "Fault Diversity Among Off-The-Shelf SQL Database Servers", *Proc. IEEE DSN'04*, Florence, Italy, pp. 389-398, 2004.

[10] P.A. Bernstein, V. Hadzilacos and N. Goodman, "*Concurrency Control and Recovery in Database Systems*", Reading, Mass., Addison-Wesley, 1987.

[11] M. Weismann, F. Pedone and A. Schiper, "Database Replication Techniques: a Three Parameter Classification", *Proc. IEEE SRDS'00*, Nurnberg, Germany, pp. 206-217, 2000.

[12] F. Pedone and S. Frolund, "Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases", *Proc. IEEE SRDS'00*, Nurnberg, Germany, pp. 176-185, 2000.

[13] M. Patiño-Martinez, R. Jiménez-Peris, B. Kemme and G. Alonso, "MIDDLE-R: Consistent Database Replication at the Middleware Level", *ACM TOCS*, 23(4), pp. 375-423, 2005.

[14] Y. Lin, B. Kemme, M. Patiño-Martínez and R. Jiménez-Peris, "Middleware based Data Replication providing Snapshot Isolation", *Proc. ACM SIG-*

MOD'05, Baltimore, MD, USA, pp. 419-430, 2005.

[15] R. Jimenez-Peris and M. Patino-Martinez, "D5: Transaction Support", ADAPT Middleware Technologies for Adaptive and Composable Distributed Components Deliverable IST-2001-37126, 21 March, 2003.

[16] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso and B. Kemme, "Scalable Database Replication Middleware", *Proc. 22nd IEEE DCS'02*, Vienna, Austria, pp. 477-484, 2002.

[17] H. Sutter, "SQL/Replication Scope and Requirements Document", *ISO/IEC JTC 1/SC 32 Data Management and Interchange WG3 Database Languages*, H2-2000-568, 2000.

[18] B. Kemme and G. Alonso, "Don't be Lazy, be Consistent: Postgres-R, a New Way to Implement Database Replication", *Proc. VLDB'02*, Cairo, Egypt, 2000.

[19] I. Gashi, P. Popov, V. Stankovic and L. Strigini, "On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers," in *Architecting Dependable Systems II*, vol. 3069, *LNCS*, R. de Lemos, Gacek, C., Romanovsky, A., Ed.: Springer-Verlag, pp. 191-214, 2004.

[20] J. Gray, "Why Do Computers Stop and What Can be Done About it?" *Proc. SRDSDS'86*, Los Angeles, CA, USA, 1986.

[21] P.E. Ammann and J.C. Knight, "Data Diversity: An Approach to Software Fault Tolerance", *IEEE TC*, C-37(4), pp. 418-425, 1988.

[22] I. Gashi and P. Popov, "Rephrasing Rules for Off-The-Shelf SQL Database Servers", *Proc. IEEE EDCC-6*, Coimbra, Portugal, pp. 139-148, 2006.

[23] J.E. Cook and J.A. Dage, "Highly Reliable Upgrading of Components", *Proc. ICSE '99*, L.A., CA, USA pp. 203-212, 1999.

[24] A.T. Tai, K.S. Tso, L. Alkalai, S.N. Chau and W.H. Sanders, "Low-Cost Error Containment and Recovery for Onboard Guarded Software Upgrading and Beyond", *IEEE TC*, C-51(2), pp. 121-137, 2002.

[25] T. Anderson and P.A. Lee, "*Fault Tolerance: Principles and Practice (Dependable Computing and Fault Tolerant Systems, Vol 3)*", Springer Verlag, 1990.

[26] P. Popov, L. Strigini, A. Kostov, V. Mollov and D. Selensky, "Software Fault-Tolerance with Off-the-Shelf SQL Servers", *Proc. Springer ICCBSS'04*, Redondo Beach, CA, USA, pp. 117-126, 2004.

[27] R. Jimenez-Peris, M. Patino-Martinez and G. Alonso, "An Algorithm for Non-Intrusive, Parallel Recovery of Replicated Data and its Correctness", *Proc. IEEE SRDS'02*, Osaka, Japan, pp. 150-159, 2002.

[28] F. Di Giandomenico and L. Strigini, "Adjudicators for Diverse-Redundant Components", *Proc. IEEE SRDS'90*, Huntsville, AL, USA, pp. 114-123, 1990.

[29] D.M. Blough and G.F. Sullivan, "Voting Using Predispositions", *IEEE TR*, R-43(4), pp. 604-616, 1994.

[30] B. Parhami, "Voting: A Paradigm for Adjudication and Data Fusion in Dependable Systems," in *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*, H. B. Diab and A. Y. Zomaya, Eds., 2005.

[31] Y. Bao, X. Sun and K.S. Trivedi, "A Workload-based Analysis of Software Aging and Rejuvenation", *IEEE TR*, R-54(3), pp. 541-548, 2005.

[32] J. Gray and A. Reuter, "*Transaction Processing : Concepts and Techniques*", Morgan Kaufmann, 1993.

[33] K.S. Tso and A. Avizienis, "Community Error Recovery in N-Version Software: A Design Study with Experimentation", *Proc. IEEE FTCS-17*, Pittsburgh, PA, USA, pp. 127-133, 1987.

[34] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", *Proc. Third Symp. on Operating Systems Design and Implementation*, New Orleans, LA, USA, pp. 173-186, 1999.

[35] P. Frankl, D. Hamlet, B. Littlewood and L. Strigini, "Evaluating Testing Methods by Delivered Reliability", *IEEE TSE*, SE-24(8), pp. 586-601, 1998.

[36] I. Gashi, "Fault Diversity Among Off-The-Shelf SQL Database Servers: Complete Results From Two Studies", http://www.csr.city.ac.uk/people/ilir.gashi/DBMSBugReports/, 2006.

[37] SourceForge, "Interbase (Firebird) Bug tracker", http://sourceforge.net/tracker/?atid=109028&group_id=9028&func=browse, 2006.

[38] PostgreSQL, "PostgreSQL Bugs Mailing List Archives", http://archives.postgresql.org/pgsql-bugs/, 2006.

[39] Microsoft, "List of Bugs Fixed by SQL Server 7.0 Service Packs", http://support.microsoft.com/default.aspx?scid=kb;EN=US;313980, 2006

[40] Oracle, "Oracle Metalink", http://metalink.oracle.com/metalink/plsql/ml2_gui.startup, 2006.

[41] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kýcýman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman and N. Treuhaft, "Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques and Case Studies", UC Berkeley Computer Science, CSD-02-1175, 2002.

[42] EnterpriseDB, "EnterpriseDB", http://www.enterprisedb.com/, 2006.

[43] Janus-Software, "Fyracle", http://www.janus-software.com/, 2006.

[44] B. Littlewood, P. Popov and L. Strigini, "Modelling software design diversity - a review", *ACM Computing Surveys*, 33(2),pp. 177-208, 2001.

[45] M.J.P. van der Meulen, P.G. Bishop and M. Revilla, "An Exploration of Software Faults and Failure Behaviour in a Large Population of Programs", *Proc. IEEE ISSRE'04*, Rennes, France, pp. 101-112, 2004.

[46] P. Popov, B. Littlewood, "The Effect of Testing on the Reliability of Fault-Tolerant Software", *Proc. IEEE DSN'04*, Florence, Italy, pp. 265-274, 2004

[47] S.S. Brilliant, J.C. Knight and N.G. Leveson, "Analysis of Faults in an N-Version Software Experiment", *IEEE TSE*, SE-16(2),pp. 238-247, 1990.

[48] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming", *IEEE TSE*, SE-12(1), pp. 96-109, 1986.

[49] I. Lee and R.K. Iyer, "Software Dependability in the Tandem GUARDIAN System", *IEEE TSE*, 21(5), pp. 455-467, 1995.

[50] S. Chandra and P.M. Chen, "Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software", *Proc. IEEE DSN'00*, NY, USA, pp. 97-106, 2000.

[51] S. Chandra and P.M. Chen, "How Fail-Stop are Programs", *Proc. IEEE FTCS-28*, Munich, Germany, pp. 240-249, 1998.

[52] J. Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich and K. Levitt, "The Design and Implementation of an Intrusion Tolerant System", *Proc. IEEE DSN'02*, Washington, D.C., USA, pp. 285-292, 2002.

[53] F. Wang, F. Gong, C. Sargor, K. Goseva-Popstojanova, K. Trivedi and F. Jou, "SITAR: A Scalable Intrusion-Tolerant Architecture for Distributed Services", *Proc. 2001 IEEE Workshop on Information Assurance and Security*, West Point, NY, U.S.A, 2001.

[54] M. Dacier (Editor), "Design of an Intrusion-Tolerant Intrusion Detection System", MAFTIA deliverable D10, http://www.maftia.org/deliverables/D10.pdf, 2002

[55] M. Castro, R. Rodrigues and B. Liskov, "BASE: Using Abstraction to Improve Fault Tolerance", *ACM TOCS*, 21(3), pp. 236-269, 2003.

[56] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabejac and A. Wellings, "GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems", *IEEE TPDS*, 10(6), pp. 580-599, 1999.

[57] Z.T. Kalbarczyk, R.K. Iyer, S. Bagchi and K. Whisnant, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance", *IEEE TPDS*, 10(6), pp. 560-579, 1999.

[58] B. Littlewood and L. Strigini, "Validation of Ultra-High Dependability for Software-based Systems", *Comc. of the ACM*, 36(11), pp. 69-80, 1993.

[59] V. Stankovic and P. Popov, "Improving DBMS Performance through Diverse Redundancy", *Proc. IEEE SRDS'06*, Leeds, UK, pp. 391-400, 2006