

Parallel Execution of Test Runs for Database Application Systems

Florian Haftmann¹

Donald Kossmann^{1,2}

Eric Lo²

¹i-TV-V AG
D-85774 Unterföhring
{*firstname.lastname*}@i-TV-T.de

²ETH Zurich
CH-8092 Zurich
{*firstname.lastname*}@inf.ethz.ch

Abstract

In a recent paper [8], it was shown how tests for database application systems can be executed efficiently. The challenge was to control the state of the database during testing and to order the test runs in such a way that expensive *reset* operations that bring the database into the right state need to be executed as seldom as possible. This work extends that work so that test runs can be executed in parallel. The goal is to achieve linear speed-up and/or exploit the available resources as well as possible. This problem is challenging because parallel testing can involve interference between the execution of concurrent test runs.

1 Introduction

Testing is the most expensive phase of the software development cycle. Large software vendors like Microsoft spend 50 percent of their development cost on testing. As another example, SAP has currently a product release cycle of 18 months of which six months are used only to execute tests. Typically, some tests must be carried out with every check-in of new code. Larger-scale integration tests that cover the whole application must be carried out every night or at least once a week. As a result, there is a huge demand to automate and optimize testing.

In the software engineering community, a great deal of frameworks have been proposed in order to manage and implement tests. The most popular framework used in practice is JUnit [2] for testing Java applications. All that work is not directly applicable to test database applications or more generally *stateful* applications. As discussed in a recent paper [8], testing database applications has special requirements: While testing a database application, the state

of the application must be controlled because the result of a function call strongly depends on the state of the database. For example, a test that checks the reporting component of an order management application must always be executed against the same test database in order to make sure that the report shows the same orders every time this test is executed. Controlling the state of the test database is a challenging task, if many tests (possibly thousands) need to be executed and if some of these tests involve updates to the database (e.g., tests that test the insertion of a new order). The work presented in [8] devised a framework to control the state of a test database and gave strategies and heuristics to execute tests in the most efficient way. That work solved the problem for a serialized world in which only one test run is executed at a time. The purpose of this work is to extend that work so that tests can be executed concurrently on a single or several machines.

Executing test runs in parallel is obviously very important if many test runs need to be executed. The goal is to exploit the available resources as well as possible. If several machines are available, the goal is to achieve linear speed-up; that is, the running time of executing all tests decreases linearly with the number of machines. In order to achieve this speed-up, it is important to balance the load on all machines – just as in all parallel applications [5]. At the same time, however, it is also important to control the state of the test database(s) and to execute the test runs in such a way that the number of database reset operations is minimized – just as for non-parallel testing in [8]. As a result, parallel testing involves solving a two-dimensional optimization problem: (a) *partitioning*: deciding which test runs to execute on which machine; and (b) *ordering*: deciding in which order to execute the test runs on each machine. In order to solve this optimization problem, this work makes the following contributions:

- A dynamic scheduling approach is presented that allows a test run scheduler to carry out load balancing and at the same time control the state of the test database(s) and minimize the number of reset operations.
- Two alternative architectures, *Shared-Nothing* (SN) and *Shared-Database* (SDB) are presented. As in parallel query processing [16], SN scales better to a large

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005

number of machines. SN executes test runs concurrently on different machines and database instances so that the execution of concurrent test runs does not interfere. SDB executes test runs concurrently on different threads using the same database instance and, thus, the concurrent test runs interfere. SDB, however, is important if the test resources of an organization are limited to one or only a few machines because it fully exploits these resources. In practice, often a combination of both architectures works best.

- For both architectures, SN and SDB, alternative scheduling strategies are presented in order to determine which test runs are to be executed on which machine/thread and in which order.
- The trade-offs of the scheduling strategies are studied experimentally using real test cases from a supplier relationship management (SRM) application and comprehensive simulations.

The remainder of this paper is organized as follows: Section 2 gives an overview of the results presented in [8] because those results form the basis for this work. Furthermore, Section 2 extends the framework of [8], presents the dynamic scheduling approach, and describes the SN and SDB architectures for concurrent and parallel testing. Section 3 presents alternative scheduling strategies for parallel testing in the SN architecture. Section 4, in turn, describes alternative scheduling strategies for parallel testing in the SDB architecture. Section 5 contains the results of performance experiments that compare the trade-offs and effectiveness of the proposed techniques for both architectures. Section 6 gives a brief overview of related work. Section 7 describes conclusions and possible avenues for future work.

2 Database Regression Tests

This section gives a brief overview of the framework presented in [8]. After that, this framework is extended for the parallel execution of tests.

2.1 Regression Test Framework

Re-iterating the definitions of [8], testing a database application involves the following components:

Test Database \mathcal{D} : The state of an application at the beginning of each test. In general, this state can involve several database instances, network connections, message queues, etc.

Reset \mathcal{R} : An operation that brings the application back into state \mathcal{D} . This operation is potentially needed after the execution of a test that updated the database. As discussed in [8], there are alternative ways to implement \mathcal{R} . For the purpose of this work, we will use the same method as used in [8] which takes about two minutes for a 180 Megabytes test database on conventional hardware and a standard RDBMS (Section 5).

Request: The execution of a function of the application. The result of the function depends on the parameters of the function call (encapsulated in the request) and the state of the test database at the time the request is executed. A request can have side effects; i.e., change the state of the test database.

Test Run: A sequence of requests that are always executed in the same order. For instance, a test run tests a specific business process that is composed of several actions (login, view product catalog, place order, specify payment, etc.). The test run is the unit in which failures are reported. It is assumed that the test database is in state \mathcal{D} at the beginning or the execution of a test run. During the execution of a test run the state may change due to the execution of requests with side-effects.

Failed Test Run: A test run for which at least one request does not return the expected result. A failed test run indicates a bug in the application program.

Schedule: A sequence of test runs and resets. The test runs and reset operations are carried out one at a time; there is no concurrency in the framework of [8].

2.2 Centralized Scheduling Strategies

The goal of a scheduling strategy is to find a schedule of test runs and resets that minimizes the running time. At the same time, it must be guaranteed that there are no false negatives; false negatives arise, for example, if no resets are executed and a test run is executed and the test database is not in state \mathcal{D} at the beginning due to the prior execution of test runs with requests that have side-effects. A naïve approach to avoid false negatives is to carry out a reset before the execution of each test run. Unfortunately, this approach has very poor performance (testing can take weeks instead of hours), as shown in [8]. Again, re-iterating the findings of [8], the following techniques were proposed to find good schedules in a centralized setting (i.e., if the test runs are executed one at a time):

Optimistic++: The Optimistic++ strategy executes the test runs in a random order. The key idea is to apply resets lazily. If a test run fails (i.e., a request returns an unexpected result), then there are two possible explanations for this failure: (a) the application program has a bug, or (b) the test database was in a wrong state due to the earlier execution of other test runs. In order to make sure that only bugs are reported, the Optimistic++ strategy proceeds as follows in this event:

- Reset the test database (i.e., execute \mathcal{R}).
- Re-execute the test run that failed.

If the test run fails again, then the test run is reported so that engineers can look for a potential bug in the application program. If not, then the first failure was obviously due to the test database being in the wrong state. In this

case, the test run is not reported and Optimistic++ remembers the test runs that were executed before this test run and records a *conflict* in a conflict database¹. Formally, a conflict is denoted as $\langle T_i \rangle \rightarrow T$, with $\langle T_i \rangle$ a sequence of test runs and T a test run. A conflict $\langle T_i \rangle \rightarrow T$ indicates that if $\langle T_i \rangle$ is executed, then the database must be reset before T can be executed. For example, one of the test runs in $\langle T_i \rangle$ could insert a purchase order and T could be a test run that tests a report that counts all purchase orders. Based on the collected conflicts in the conflict database, the Optimistic++ strategy tries to avoid running a test run twice, in all subsequent tests. For instance, if all or a superset of the test runs in $\langle T_i \rangle$ have been executed, then Optimistic++ will automatically reset the test database (i.e., execute \mathcal{R}) before the execution of T , thereby avoiding a failure of T due to a wrong state of the test database.

The Optimistic++ strategy (and all the following strategies which extend Optimistic++) is susceptible to a phenomenon called *false positives*; i.e., a test run does not fail although it should fail. As discussed in [8], however, this phenomenon is rare in practice. Also, this risk is affordable because in return it is possible to execute a large number (thousands) of test runs which would otherwise not be possible.

Slice: Slice extends the Optimistic++ strategy. Rather than executing the test runs in a random order, the Slice heuristics use the conflict information in order to find a schedule in which as few resets as possible are necessary. For example, if test run T_1 tests the insertion of a purchase order into the database and T_2 tests a report that counts the number of purchase orders, then T_2 should be executed *before* T_1 . The conflict information is gathered in the same way as for the Optimistic++ strategy. If there is a conflict between test runs $\langle T_i \rangle$ and T , then Slice executes T *before* $\langle T_i \rangle$. At the same time, however, Slice does not change the order in which the test runs in $\langle T_i \rangle$ are executed because those test runs can be executed in that order without requiring a database reset. Such a sequence of test runs is called a *slice*.

The Slice heuristics can best be described by an example with five test runs T_1, \dots, T_5 (taken from [8]). At the beginning, no conflict information is available, so that the five test runs are executed in a random order. Let us assume that this execution results in the following schedule (\mathcal{R} denotes the database reset operation, T_i denotes the execution of test run T_i):

$$\mathcal{R} T_1 T_2 T_3 \mathcal{R} T_3 T_4 T_5 \mathcal{R} T_5$$

From this schedule, we can derive the two conflicts: $\langle T_1 T_2 \rangle \rightarrow T_3$ and $\langle T_3 T_4 \rangle \rightarrow T_5$. Correspondingly, there are three slices: $\langle T_1 T_2 \rangle$, $\langle T_3 T_4 \rangle$, and $\langle T_5 \rangle$. Based on the conflicting information in the conflict database and the collected slices, Slice executes T_3 before $\langle T_1 T_2 \rangle$ and T_5 before $\langle T_3 T_4 \rangle$ in the next iteration. In other words, Slice executes

the test runs in the following order: $T_5 T_3 T_4 T_1 T_2$. Let us assume that this execution results in the following schedule:

$$\mathcal{R} T_5 T_3 T_4 T_1 T_2 \mathcal{R} T_2$$

In addition to the already known two conflicts, the following conflict is added to the conflict database: $\langle T_5 T_3 T_4 T_1 \rangle \rightarrow T_2$. The slices after this iteration are: $\langle T_5 T_3 T_4 T_1 \rangle$ and $\langle T_2 \rangle$. As a result, the next time the test runs are executed, the Slice heuristics try the following order: $T_2 T_5 T_3 T_4 T_1$.

The Slice heuristics reorder the test runs with every iteration until reordering does not help anymore either because the schedule is perfect (no resets after the initial reset) or because of cycles in the conflict data (e.g., $\langle T_1 \rangle \rightarrow T_2$, $\langle T_2 \rangle \rightarrow T_3$, $\langle T_3 \rangle \rightarrow T_1$). The details can be found in [8].

It should be noted that the Slice heuristics are not perfect. There are situations in which the Slice heuristics produce sub-optimal schedules. So far, no polynomial algorithm has been found that finds an optimal schedule (minimum number of reset) given a set of test runs and conflicts. Although it has not been proven yet, the problem is believed to be \mathcal{NP} hard.

Graph-based heuristics: Graph-based heuristics are alternatives to the Slice heuristics. They were shown to perform overall as well as the Slice heuristics for a serial execution of test runs in [8]. The idea is to model conflicts as a directed graph in which the nodes are test runs and the edges $T_j \rightarrow T_k$ are conflicts indicating that T_j might update the test database in such a way that a reset (\mathcal{R}) is necessary with probability w in order to execute T_k after T_j ². The edges are weighted by this probability w . Based on this weighted directed graph, a graph reduction algorithm can be applied in order to find a good order to execute the test runs. The best known heuristics for the graph reduction were called *MaxWeightedDiff* in [8].

The details of graph-based heuristics and the MaxWeightedDiff strategy, in particular, are presented in [8]. The details of extending the MaxWeightedDiff strategy for parallel testing can be found in [9]. Due to space constraints, we cannot describe these details in this paper and only give a sketch of the main ideas.

2.3 Parallel Testing

As mentioned in the introduction, parallel testing is a two dimensional scheduling problem. In addition to deciding in which order to execute the test runs, a scheduling strategy must partition the test runs. Depending on the architecture, Shared-Database or Shared-Nothing (see below), a parallel execution can increase the number of resets due to interference (Shared-Database) or decrease the number of resets (Shared-Nothing) by executing test runs that are in conflict concurrently. As a result, conflict information ought to be taken into account in order to decide on which machine to

¹Readers interested on the implementation details of the conflict database are referred to [8].

²Similar to the Slice heuristics, conflicts are detected and collected along the testing in graph-based heuristics.

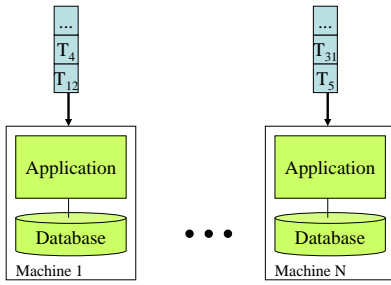


Figure 1: Shared-Nothing (SN) Architecture

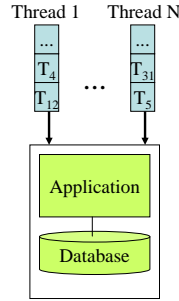


Figure 2: Shared-Database (SDB) Architecture

execute which test run. Furthermore, it is important to balance the load on all machines so that the resources are used as well as possible. Load balancing can be carried out without conflict information; load balancing should be carried out taking the current load of machines and the estimated length of test runs into account.

2.3.1 Shared-Nothing vs. Shared-Database

Figures 1 and 2 show the two scenarios considered in this work for the concurrent execution of test runs. In the Shared-Nothing architecture (SN) shown in Figure 1 there are N separate and independent installations of the application and its underlying database. In Figure 1, each of these installations is on a separate machine, but it is also possible that several installations are on a single machine or that a single installation spans several machines (e.g., the application tier could be distributed on a cluster of machines). What is important for this architecture is that the installations do not share state and, thus, do not interfere. For presentation purposes, we use the term *machine* in order to denote an installation of the application in the remainder of this paper. Furthermore, on each installation only one test run is executed at a time.

The Shared-Database architecture (SDB) is shown in Figure 2. In this architecture, there is only one installation of the application and its underlying database and test runs are executed concurrently on this instance. In Figure 2, the installation of the application and the database is on a single machine; again, however, the installation could be distributed on several machines. The important observation is that concurrent test runs interfere in this scenario because they read and update the same database.

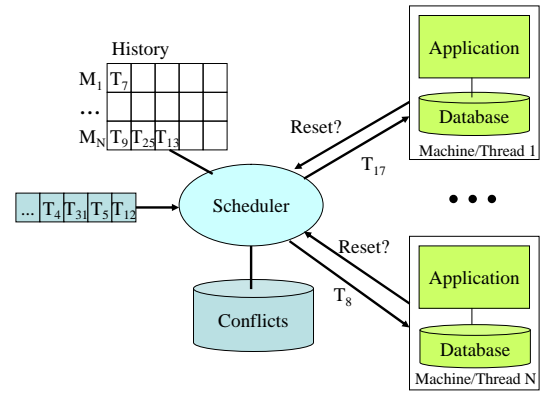


Figure 3: Scheduler for Parallel Testing

The trade-offs between the SN and SDB architectures are straightforward. Obviously, the SN architecture scales better to a large number of machines. Also, scheduling for the SN architecture is easier. On the other hand, the SN architecture wastes resources because the resources of each installation are not fully utilized by a single thread of test runs; e.g., if multi-processor machines are used. Furthermore, some organizations simply cannot afford more than one test installation: each installation requires extra system administration work and possibly software license fees are required for each installation. As a result, a combination of both architectures (multiple installations and concurrent test thread for each installation) are useful in practice, and thus, we will explore both architectures as part of this work.

2.3.2 Scheduling Parallel Tests

For both architectures (SN and SDB), we propose the model presented in Figure 3 in order to schedule test runs for concurrent execution. The scheduler has an input queue of test runs. How to order the test runs in this input queue depends on the scheduling strategy (Sections 3 and 4). At the beginning, the scheduler takes the first test run from its input queue and submits it for execution to Machine 1 in the SN architecture or to Thread 1 in the SDB architecture. (Figure 3 shows machines in the SN architecture, but the same principles apply to feeding test threads in the SDB architecture.) Furthermore, the scheduler submits the second test run to the second machine/thread and so on until all N machines/threads are busy.

When a machine (or thread), say M_i , has completed the execution of a test run, say T_k , M_i notifies the scheduler that it is ready to execute a new test run. The scheduler keeps a history of all test runs that have been executed on M_i and correspondingly places T_k into its history for M_i . Furthermore, the scheduler selects the next test run to be executed on M_i from its input queue. In most cases, the scheduler selects the first test run from its input queue, but there are occasions in which it is beneficial not to select the first test run from the queue. In SN, for example, if it is known that T_{12} and T_{17} are in conflict in Figure 3, then it might be beneficial not to execute T_{12} on M_1 after T_{17} has

been executed on M_1 and instead execute T_5 on M_1 and wait until another machine becomes available for T_{12} . In order to decide which test run to execute next, the scheduler takes the conflict database, the history, and the order in the input queue into account. Alternative policies how such optimizations are applied for SN and SDB are described in Sections 3 and 4.

When a machine/thread informs the scheduler that it has completed the execution of a test run, it also indicates whether it had to reset the database in order to execute the test run. Recall from the description of the Optimistic++ strategy in Section 2.2 that the database is reset whenever a test run fails in order to make sure that this failure is not due to the test database being in the wrong state. If a reset has been carried out by M_i in order to execute T_k , then the scheduler updates its history information and the conflict database in the following ways for the SN architecture:

- *Conflict Database:* A conflict $H_{M_i} \rightarrow T_k$ is inserted into the conflict database. Here H_{M_i} represents the sequence of test runs recorded in the history of the scheduler for M_i . This conflict follows directly from the definition of a conflict and the fact that a reset was necessary in order to execute T_k after the execution of the test runs in H_{M_i} .
- *History:* The history for M_i is flushed; i.e., $H_{M_i} := T_k$. The updates of all test runs that were executed on M_i before T_k are undone due to the reset so that these test runs need not be recorded in the history anymore.

For the SDB architecture, the conflict database and history are updated in the following way if the database has been reset:

- *Conflict Database:* A conflict $\bigcup_{i=1, \dots, N} H_{M_i} \rightarrow T_k$ is inserted into the conflict database.³
- *History:* The history for all machines is flushed; i.e., $H_{M_j} := \emptyset$ for $j = \{1, \dots, N\} \setminus \{i\}$ and $H_{M_i} := T_k$.

The rules for the SDB architecture are the same as for the SN architecture; the difference is that they affect all threads rather than just a single machine.

2.3.3 Summary

Obviously, parallel testing can significantly reduce the running time of executing a set of test runs. In the SN architecture, it can be expected that the speed-up is linear because there is no interference and if all test runs have roughly the same length (if not, bin packing must be applied to ensure load balancing). In fact, super-linear speed-up is possible because conflicting test runs can be executed on different machines so that the total number of resets is reduced. In the SDB architecture, linear speed-up is only possible for small levels of concurrency and if database resets are rare. A database reset blocks all activity. Nevertheless, if a

³An ordered union is carried out by merging the test runs according to their timestamps.

scheduling strategy makes sure that conflicting test runs are not executed concurrently, then significant speed-ups can be achieved in this architecture, too, because hardware resources (disks, multiple CPUs and co-processors) are better exploited.

In order to implement parallel testing, the framework of [8] is applicable, but needs to be extended. First, the notion of conflict needs to be refined (as discussed in the previous section). More importantly, the scheduling strategies must be extended in order to decide on which machine or in which thread to execute a test run. How to extend the scheduling strategies for the SN and SDB architectures is the subject of the next two sections.

3 Scheduling Test Runs for SN

This section presents how the centralized scheduling strategies of Section 2.2 can be extended for the Shared-Nothing architecture. As shown in Figure 3 and described in Section 2.3, there are two decisions that need to be made in order to schedule test runs for parallel execution:

- Determine the order of test runs in the scheduler’s input queue of test runs.
- Define a criterion in which situations the scheduler will *not* select the first test run of its input queue for execution.

The remainder of this section describes how the Optimistic++, Slice, and MaxWeightedDiff heuristics from [8] are extended to make these two decisions for the Shared-Nothing architecture (SN).

3.1 Parallel Optimistic++

The Optimistic++ policy takes a very simplistic parallel scheduling approach:

- The test runs are put into the scheduler’s input queue in random order.
- The scheduler always selects the first test run from its input queue when a machine becomes available.

Note, however, that the parallel Optimistic++ strategy for SN does maintain a history and conflict information as presented in Section 2.3. The parallel Optimistic++ strategy uses this information to issue a database reset before a test run is executed for the first time on a machine, if this test run is known to be in conflict with a sub-set of the test runs in the history of that machine. This way, the parallel Optimistic++ strategy avoids unnecessarily executing a test run twice as a result of the test database of that machine being in the wrong state.

3.2 Parallel Slice Heuristics

The key idea of the parallel Slice heuristics for SN is to schedule whole slices rather than individual test runs. Recall from Section 2.2 that a slice is a sequence of test runs

that can be executed without a database reset; i.e., there are no conflicts within a slice. A slice is defined as a sequence of test runs that was executed successfully between two resets. In the SN architecture, naturally, the test runs of a slice should be executed sequentially on the same machine.

Example: Again, the parallel Slice algorithm can best be described using an example. At the beginning, the parallel Slice heuristics behave exactly like the parallel Optimistic++ heuristics; that is, the order of test runs in the scheduler’s input queue is random and whenever a machine is available, the next test run is selected. Let us assume that this results in the following schedule in a scenario with two machines:

$$M_1 : \mathcal{R} T_1 T_2 T_3 \mathcal{R} T_3$$

$$M_2 : \mathcal{R} T_5 T_6 \mathcal{R} T_6 T_7 T_8$$

As a result, four slices can be identified: $\langle T_1 T_2 \rangle$, $\langle T_3 \rangle$, $\langle T_5 \rangle$, and $\langle T_6 T_7 T_8 \rangle$. Accordingly, the following conflicts are detected: $\langle T_1 T_2 \rangle \rightarrow T_3$ and $\langle T_5 \rangle \rightarrow T_6$. Using the centralized Slice heuristics (Section 2.2 and [8]), the test runs executed on M_1 and the test runs executed on M_2 are re-ordered separately. As a result, we obtain two re-orderings, one for each machine ($\langle s \rangle$ denotes the delimiters of slices):

$$O_1 : \langle s \rangle T_3 \langle s \rangle T_1 T_2 \langle s \rangle$$

$$O_2 : \langle s \rangle T_6 T_7 T_8 \langle s \rangle T_5 \langle s \rangle$$

These two (partial) orders of test runs are merged to one (total) order of all test runs which serves as the order for the input queue of the scheduler in the next iteration. This merge is carried out in a round-robin fashion. That is, the total order is:

$$\langle s \rangle T_3 \langle s \rangle T_6 T_7 T_8 \langle s \rangle T_1 T_2 \langle s \rangle T_5 \langle s \rangle$$

Given this input queue, the scheduler operates as described in Section 2.3. T_3 is executed on Machine 1 and T_6 is executed on Machine 2. If a third machine were available in this iteration of testing, then the scheduler would *not* select T_7 because T_7 should be executed on Machine 2; instead, the scheduler would select T_1 . Likewise, if there are only two machines and Machine 1 has completed the execution of T_3 , then the scheduler would select T_1 for execution on Machine 1. Let us assume that indeed only two machines are available in this iteration and that the scheduler produces the following schedules on these two machines in the next iteration:

$$M_1 : \mathcal{R} T_3 T_1 \mathcal{R} T_1 T_2$$

$$M_2 : \mathcal{R} T_6 T_7 T_8 T_5 \mathcal{R} T_5$$

As a result, there are still the same four slices as in the first iteration. Two conflicts are added to the conflict database (in addition to the two conflicts detected in the first iteration): $\langle T_3 \rangle \rightarrow T_1$ and $\langle T_6 T_7 T_8 \rangle \rightarrow T_5$. At this point, there is a cyclic conflict between $\langle T_3 \rangle$ and $\langle T_1 T_2 \rangle$ so that no re-ordering is attempted for these two traces (see [8] for details). Likewise, there is a cyclic conflict between $\langle T_6 T_7 T_8 \rangle$

and $\langle T_5 \rangle$ so that those two slices are not reordered either. As a consequence, the order in which the test runs are put into the scheduler’s input queue for the third iteration is the same as in the second iteration (after a round-robin merge).

Even though the order of test runs in the input queue did not change, the dynamic behavior of the scheduler is different due to the additional conflicts recorded in the conflict database. At the beginning, T_3 is scheduled for execution on M_1 and T_6 is scheduled for execution on M_2 . At this point, the state of the input queue is as follows (again, using $\langle s \rangle$ in order to depict beginnings and endings of slices for presentation purposes):

$$T_7 T_8 \langle s \rangle T_1 T_2 \langle s \rangle T_5 \langle s \rangle$$

When M_1 completes the execution of T_3 , then the following rules are applied in order to find the next test run to be executed on M_1 :

- T_7 and T_8 are not selected because they are part of a slice that is currently executed on another machine, M_2 .
- T_1 is not selected because there is a conflict $\langle T_3 \rangle \rightarrow T_1$ and T_3 is in the history of M_1 .
- T_2 is not selected because it is part of the same slice as T_1 , and T_1 has a conflict.
- T_5 is selected because it does not violate any of the three rules above.

As a consequence, T_5 is executed on M_1 . This is in the spirit of the Slice heuristics (keeping slices intact) and at the same time gives the biggest hope to minimize the total number of database resets.

Now what happens next when M_1 completes the execution of T_5 ? At this point, there are no more candidate test runs for M_1 left. Rather than letting M_1 go idle, the Slice heuristics simply selects the first test run from the scheduler’s input queue in this event; i.e., T_7 in this example.

In summary, there are four ideas in the design of the parallel Slice heuristics: (a) The order of test runs in the scheduler’s input queue is determined by applying the central Slice heuristics to each machine individually and then merging the partial orders (using a round-robin approach). (b) The scheduler executes all test runs of the same slice on the same machine. (c) The scheduler dynamically uses the history and conflict information in order to make sure that conflicting slices are executed on different machines as much as possible. (d) Utilizing all available machine resources has higher priority than minimizing the number of database resets. In other words, if no suitable test run is found in the input queue, the scheduler selects the first test run from the input queue, thereby causing an addition reset, rather than letting the machine go idle.

3.3 Parallel MaxWeightedDiff Heuristics

The parallel MaxWeightedDiff heuristics work as follows: The conflict graph is constructed in the same way as in

the regular MaxWeightedDiff heuristics for a serial execution [8]. Likewise, the order of test runs in the scheduler’s input queue is determined just as for the centralized MaxWeightedDiff heuristics. When the execution of a test run has completed on, say, machine M_x , then the scheduler selects the first test run from its input queue, if the cumulated weights of test runs in the history of M_x to that test run is lower than a certain threshold. Otherwise, this criterion is tested for the second test run, third test run and so, until a suitable test run is found. If no such test run exists, then the first test run is selected. As a threshold, we use 1 in all experiments reported in this paper because this setting was very good in all situations. An example and details of the algorithm are given in [9] and cannot be presented due to space constraints.

4 Scheduling Test Runs for SDB

Just as for the SN architecture, the key questions that need to be answered for the Shared-Database architecture (SDB) are in which order the test runs are inserted into the scheduler’s queue and in which situations the scheduler selects the first test run from this queue. Furthermore, there is a question of how to schedule the reset operation and how to detect failures in the presence of concurrent test runs. In all, making the right scheduling decision is more important for SDB than for SN. Due to the inference of concurrent test runs, it is important to make sure that conflicting test runs do not run concurrently in addition to making sure that conflicting test runs are not executed subsequently. The remainder of this section shows how the Optimistic++, Slice, and MaxWeightedDiff strategies can be applied to the SDB architecture. Furthermore, alternative implementations for the database reset operation are discussed.

Throughout this section, it is assumed that the multi-programming level, N , (i.e., the number of available test threads) is constant. How to dynamically adjust N based on, say, the load of the machine and its resources is one important avenue for future work.

4.1 Parallel Optimistic++

The parallel Optimistic++ strategy for SDB works in exactly the same way as the parallel Optimistic++ strategy for SN (Section 3.1). That is, the test runs are put into the scheduler’s input queue in random order and the scheduler always selects the first test run from its input queue when a test run has been completed (i.e., when a test thread becomes available). The Optimistic++ strategy for SDB and SN differ in the way that conflicts are recorded (Section 2.3) and reset operations are scheduled (Section 4.4).

4.2 Parallel Slice

In the SN architecture, one of the key ideas was to execute test runs from the same slice sequentially on the same machine. In the SDB architecture, the idea is to execute test runs from the same slice concurrently on different threads because these test runs are known not to be in conflict.

Again, the best way to describe the Slice heuristics for SDB is by the means of an example.

In this example, there are seven test runs executed concurrently in two threads on one machine. At the beginning, the test runs are executed in a random order. Let us assume that the following schedule is generated: (A database reset terminates the execution of test runs in both threads in SDB. How to schedule such resets is described in Section 4.4.)

Thread 1: T_1 T_2 \mathcal{R} T_2 T_3 \mathcal{R}
 Thread 2: T_5 T_6 T_7 T_8 T_8

In this example, T_2 is the first test run that fails because the database was in a wrong state for T_2 . At this point, it is not clear whether an update request in T_1 , T_5 or T_6 or a combination of update requests carried out as part of these test runs caused the failure of T_2 . Considering the time stamps when T_1 , T_5 , and T_6 were started, a conflict $\langle T_1 T_5 T_6 \rangle \rightarrow T_2$ is recorded. The first slice is $\langle T_1 T_5 T_6 \rangle$ because these test runs could be executed concurrently/sequentially without a failure.

The second test run that fails is T_8 . The corresponding conflict is $\langle T_2 T_7 T_3 \rangle \rightarrow T_8$. Accordingly, there are two more slices: $\langle T_2 T_7 T_3 \rangle$ and $\langle T_8 \rangle$. Based on these slices and conflicts, the order in which the test runs are inserted into the scheduler’s input queue for the next iteration of testing is as follows (again showing the slice delimiters for presentation purposes):

$$\langle s \rangle T_8 \langle s \rangle T_2 T_7 T_3 \langle s \rangle T_1 T_5 T_6 \langle s \rangle$$

These test runs are then scheduled in this order; that is, the scheduler always selects the next test run from its input queue after the execution of a test run has been completed and a test thread becomes available. There is no dynamic re-scheduling based on histories and the conflict information.

In summary, the parallel Slice heuristics for SDB work according to the following two principles. One, the ordering of test runs for the scheduler’s input queue is carried out based on slices and conflict information in exactly the same way as for the Slice heuristics in a centralized setting (Section 2.2 and [8]). The only difference is that timestamps are used to define a total order on concurrent test runs. Two, the scheduler always selects the first test run from its input queue and does not apply any more complex heuristics to dynamically reorder the test runs based on history and conflict information.

4.3 Parallel MaxWeightedDiff

The parallel MaxWeightedDiff heuristics for SDB works in a very similar way as the centralized MaxWeightedDiff heuristics. That is, the test runs are ordered in the scheduler’s input queue in the same way as described in [8] for a serial execution of test runs. Furthermore, the scheduler always selects the first test run from its input queue when a test thread becomes available. Again, an example and details of the approach can be found in [9].

4.4 Scheduling Database Resets

Another question that is specific to the SDB architecture is how to schedule a database reset when a test run fails, potentially due to a wrong state of the test database. The question is what happens to the test runs that are executed in concurrent threads? This question does not arise in the SN architecture because a reset caused by a failure of a test run on one machine does not impact the concurrent execution of test runs on other machines.

Conceptually, scheduling a reset in the SDB architecture is related to the problem of scheduling a check point for database recovery in a database engine [6]. There are several options:

- *lazy*: Concurrent test runs that have started are completed, but no new test runs are started as soon as a test run fails and a database reset becomes necessary.
- *eager*: Concurrent test runs are aborted and the database reset is carried out immediately. After the database has been reset, all test runs that have not completed must be restarted.
- *deferred*: The database reset is deferred and the failed test run is re-scheduled to be executed at the end. In other words, the first reset is carried out after every test run has been tried once. After that, a database reset is carried out and the test runs that failed in the first round are re-scheduled (using Optimistic++, Slice, or MaxWeightedDiff heuristics).

The trade-offs between these three alternatives are fairly complex and the choice for the best approach depends on the number of conflicts between test runs. We plan to explore these trade-offs as part of future work. For the purpose of this work, we chose the *lazy* strategy because it is very robust and minimizes the amount of wasted work by failed test runs.

Another related question is how to restart the execution of test runs after a database reset. Again, there are several options:

- *single-threaded*: Execute the failed test run again in isolation; i.e., without starting any other test runs concurrently. The advantage of this approach is that if the test run fails again, it is guaranteed that this failure is due to a potential bug in the application and not due to the database being in a wrong state at the time of the failure due to the execution of update requests by concurrent test runs.⁴ The disadvantage is that parallelism is lost.
- *multi-threaded*: Execute the failed test run concurrently with other test runs. Clearly, the advantage of

⁴A test run is *not* a transaction and can span several database transactions. Typically, every request or a small number of requests are implemented by the application as a single database transaction in order to achieve recoverability of long running business transactions. As a result, test runs (which represent such long running business transactions) do see updates carried out by other, concurrent test runs.

this approach is that no parallelism is lost. The disadvantage is that if the test run fails again, that test run must be executed yet another time until it is successful or fails in a single-threaded environment.

- *mix*: Many other strategies are conceivable. For instance, it is possible to restart with a lower degree of parallelism and/or in single-threaded mode after another failure.

Again, the choice between these alternatives depends on the number of conflicts and the probability that a test run fails after a restart due to other concurrent test runs. We plan to study these effects in detail as part of future work. For the purpose of this work, we chose the *single-threaded* approach because this approach is robust and minimizes the amount of wasted work due to re-executing failed test runs.

5 Performance Experiments and Results

This section presents the results of performance experiments in order to study the effectiveness of the alternative scheduling strategies for the SN and SDB architectures. The following sets of experiments were made:

- simulation with synthetic test runs in SN;
- execution of real test runs in SDB;
- simulation with synthetic test runs in SDB.

The real test runs were taken from an SRM application using the test installation of an industrial customer of that application. Carrying out experiments with several installations of the application (SN) was not possible in this environment because the customer did not have the available resources to do such experiments. The synthetic test runs were generated with the purpose to study the performance of the strategies for a large number of test runs with varying characteristics (e.g., number of conflicts).

5.1 Experimental Environment

Synthetic Test Runs: The synthetic test runs were generated using a dummy database application in order to have control over the length of the test runs (running time when executed in isolation) and the conflicts between test runs. Table 1 summarizes the characteristics of these synthetic test runs. In all experiments, 10,000 synthetic test runs were used. The length of a test run was chosen randomly in the range of 0 minutes (just one request) to three minutes (450 requests) using a uniform distribution. These settings were inspired by the real test cases (Table 2). The number of conflicts between the test runs was varied from 10,000 (low) to five millions (high) and a uniform distribution was used in order to randomly generate conflicts between test runs when the test runs were synthesized as in [8]. We also experimented with a Zipf distribution in which there was skew in the conflict distribution and some test runs were in conflict with many other test runs, but we do not show the

number of test runs	10,000
length of test runs	0 min - 3 min (avg. 1.5 min)
number of conflicts	10,000 (low) - 5 mio (high)
conflict distribution	uniform

Table 1: Synthetic Test Runs

number of test runs	61
length of test runs	0 min - 3 min (avg. 1.5 min)
number of conflicts	unknown
conflict distribution	unknown

Table 2: Real Test Runs

results here for brevity: the results with the Zipf distribution were almost identical with the results for the uniform distribution.

Real Test Runs: The real test runs were taken from a real database application (BTell by i-TV-T AG) using the test installation of one of the customers of that application (Unilever). BTell is a Java application with approximately 3000 classes and a relational database schema with approximately 500 tables. These BTell test runs were generated manually by test engineers who administrate the application for Unilever. These test runs were also used in the experiments of [8]. The characteristics of this set of test runs is given in Table 2. There are 61 test runs and the length characteristics are the same as for the synthetic test runs. The conflict distribution could not be determined because doing so would involve expensive analysis of the BTell application code.

Simulation of SN: In order to carry out experiments with the SN architecture, we used an event-based simulator that simulated the execution of the synthetic test runs and of database reset operations. A database reset operation costed two minutes and the execution times of the test runs were determined by their length (between 0 and 3 minutes, Table 1). Up to 50 machines could be simulated this way. The details of the simulator can be found in [9].

Hardware for SDB: For the experiments with real test runs in the SDB architecture, we used a machine with two 3.2 GHz Pentium 4 processors and 4 GB of main memory running Linux. The BTell application and database (IBM DB2) were installed on this machine. Executing a reset costed two minutes on this machine.

Simulation of SDB: In order to carry out experiments with synthetic test runs in the SDB architecture, we again used a simulator. The simulator modeled a cluster of machines that each run an instance of the application and a centralized database server. In other words, a scalable three-tier architecture such as that of SAP [10] is modeled. In this experiment, the hardware resources are not the bottleneck; instead, the interference of the execution of concurrent test runs is the limiting factor. Again, the details of this simulator are described in [9].

Execution of Experiments: We studied the Optimistic++, Slice, and MaxWeightedDiff heuristics as described in Sections 3 and 4. The full details of all algorithms (in particular the MaxWeightedDiff heuristics) can be found in [9]. In all experiments, the conflict database was initially empty. We ran a total of thirty iterations, thereby incrementally building up conflict information and improving the scheduling decisions. This section reports on the average running time and average number of resets of the last ten iterations.

We also measured the CPU overhead of the scheduler in order to make scheduling decisions. This overhead, however, was negligible (only a few seconds) in all experiments so that we do not report these results in this paper. We also carried out experiments that study how quickly the alternative strategies learn the relevant conflict information in order to produce good schedules. These results are presented in [9] and are omitted in this paper for brevity. The observations are almost the same for the parallel strategies as for their centralized counterparts which were studied in [8]: Slice converged very fast (within a few iterations) and MaxWeightedDiff more slowly (approximately 10 iterations, depending on the number of conflicts). Both improved significantly (50 percent speed-up) from the first to the last iteration.

5.2 Shared-Nothing Simulation

Low Conflict: Recall that the goal is to achieve linear speed-up for a large number of machines in an SN architecture. Table 3 shows the running times (in hours) and number of resets for synthetic test runs with a low number of conflicts (10,000) between the test runs. The following observations can be made:

First, all three strategies achieve a linear speed-up with a growing number of machines. The running time is almost 50 times as high if only one machine is available than if 50 machines are available. The Slice heuristics even showed a super-linear speed-up if 50 machines were available for making sure that conflicting test runs ran on different machines as much as possible. Although not shown in this experiment, this scalability would easily go beyond 50 machines up to the point at which load balancing and bin packing of test runs with different lengths actually matters or the scheduler itself becomes a bottleneck.

Second, all three strategies have roughly the same running times. They only differ in the number of resets (Slice is the best strategy in this respect). However, for the low conflict synthetic test runs, the number of resets is fairly low for all three strategies and executing resets does not impact the running time significantly. (Note that resets are executed in parallel with test runs and other resets in SN.) Only if 50 machines are available, Slice outperforms the other strategies for producing schedules with an extremely small number of resets.

High Conflict: Table 4 shows the results of the experiments carried out with the synthetic test runs with a high

Approach	1 machine		5 machines		10 machines		50 machines	
	Time	Reset	Time	Reset	Time	Reset	Time	Reset
Optimistic++	254.9	99	54.4	82	27.9	75	5.8	62
Slice	249.2	35	52.8	37	26.2	34	3.4	10
MWD	249.0	60	54.5	69	27.9	68	5.7	57

Table 3: Running Time (hours), Resets
Synthetic, Low Conflict, SN

Approach	1 thread		2 threads		5 threads	
	Time	Reset	Time	Reset	Time	Reset
Optimistic++	41	7.0	22	6.6	16	5.8
Slice	31	3.0	18	3.8	12	4.2
MWD	37	3.5	19	4.2	13	4.2

Table 5: Running Time (min), Resets
Real, SDB

number of conflicts on SN. Again, it can be observed that all three strategies scale well with an increasing number of machines. If only one machine is available, the test runs take about 15 days; with 50 machines available, the test runs can be carried out within one night. Furthermore, Slice has the lowest number of resets and thus shows the best running time in all settings; the differences in running times, however, are moderate for the same reasons as in the experiments with the low conflict test runs (Table 3).

5.3 Shared-Database, Real Test Runs

Table 5 shows the running times (in minutes) and numbers of resets of the alternative strategies for the real test runs using Unilever’s test installation of the BTell application (SDB). The goal of the SDB architecture is to exploit the resources of the available machine as well as possible. As can be seen in Table 5, increasing the multi-programming level up to five concurrent test threads gives significant reductions in the running time. The running time drops almost linearly until the machine resources are saturated which is at about five concurrent test threads. After that, increasing the multi-programming level does not result in any improvements.

The parallel Slice heuristics are the winner with regard to the number of resets and running time. In the best case (5 test threads), it outperforms Optimistic++ by 33 percent. It is also slightly better than the MaxWeightedDiff heuristics, but the margins are small.

In all, these experiments confirm that a parallel execution of test runs can be very beneficial, even if only one machine is available and the resources for testing are limited. These experiments were carried out at a typical customer and, therefore, we expect these experiments to be representative for a large class of test environments. The experiments also confirm that a good scheduling strategy (such as Slice) is important in the SDB architecture with an increasing multi-programming level because it tries to avoid that conflicting test runs are executed concurrently. We will explore this effect in more detail in the next subsection when we present simulation experiments on SDB with a higher degree of concurrency.

Approach	1 machine		5 machines		10 machines		50 machines	
	Time	Reset	Time	Reset	Time	Reset	Time	Reset
Optimistic++	358.0	1788	72.0	1787	35.9	1775	6.8	1753
Slice	306.4	867	63.9	1098	31.8	1038	6.4	1048
MWD	359.4	1792	72.1	1784	35.9	1780	7.6	1767

Table 4: Running Time (hours), Resets
Synthetic, High Conflict, SN

5.4 Shared-Database Simulation

Tables 6 and 7 show the running times (in hours) and number of resets of the alternative strategies using synthetic test runs and the simulator for the SDB architecture. Table 6 shows the results for synthetic test runs with a low number of conflicts. Table 7 shows the results for synthetic test runs with a high number of conflicts. Recall that the purpose of these experiments was to study the interference of concurrent test runs in the SDB architecture for high degrees of concurrency (up to 50 test threads).

If the number of conflicts is low (Table 6), then interference is not an issue. The number of resets stays constant, independent of the number of test threads. Correspondingly, almost linear speed-up can be achieved until the database server or the network becomes saturated. Also, since the number of conflicts is low, all three scheduling strategies show almost the same performance: Slice has the lowest number of resets, but in terms of response time, all three strategies are almost identical.

If the number of conflicts is high (Table 7), interference indeed becomes an issue in an SDB architecture. Comparing Tables 6 and 7, it can be seen that both the number of resets and the running times are much higher for the synthetic test runs with a high number of conflicts. Also, the running time with 50 test threads is twice as high as with 10 test threads, indicating that increasing the degree of concurrency can hurt performance significantly if the number of conflicts are high. Nevertheless, even for a high number of conflicts, the SDB architecture can be beneficial and significant performance improvements can be achieved as compared to a serial execution of test runs, if the number of test threads is controlled.

6 Related Work

The most relevant related work is the work described in [8]. This paper extends the framework, algorithms, and experimental results of [8] for a parallel (concurrent) execution of test runs. Furthermore, products that support a parallel test environment for stateless applications are beginning to appear on the marketplace; e.g., TestStand [1]. In the software engineering community, there has been a great deal of work in the general area of testing; e.g., white box and black box testing, analysis of the coverage of test cases, and methodologies to plan and integrate the test phase into the software development life cycle [15]. In order to speed up the execution of testing, the selective execution of test cases has gained a great deal of attention (e.g., [13]). The idea is to execute only those test cases that are potentially affected by a change in the application. Clearly, all that

Approach	1 thread		5 threads		10 threads		50 threads	
	Time	Reset	Time	Reset	Time	Reset	Time	Reset
Optimistic++	254.9	99	61.0	85	31.0	85	6.0	70
Slice	249.3	35	57.4	34	30.2	33	5.9	32
MWD	249.1	60	60.1	62	31.9	58	5.9	57

Table 6: Running Time (hours), Resets
Synthetic, Low Conflict, SDB

work is orthogonal to the work presented in this paper. In addition, the scheduling problem we tackled is also related to compiler design for multi-CPU machines or CPUs with hyper-threading (e.g., [4]), but with totally different assumptions (CPU needs to know and analyze the code) and performance trade-off consideration.

In the database community, there is only very little work on testing. The RAGS system [14] generates a large number of SQL queries in order to test a relational database system. There has also been work on the generation of test databases based on integrity constraints defined in the database schema [12, 3]. Furthermore, there has been work on quickly generating large databases with certain attribute value distributions in order to test the performance and scalability of a database system [7]. Again, all this work is orthogonal to the work presented in this paper.

7 Conclusion

This paper studied how the parallel (concurrent) execution of test runs can speed-up the execution of a potentially large number of tests. To this end, a dynamic scheduling approach was proposed which has several advantages (Section 2.3). First, it extends the framework of [8] in order to control the state of the database and apply expensive database reset operations lazily, thereby minimizing the number of times that expensive database reset operations need to be carried out. Second, it allows to make dynamic decisions in order to carry out load balancing and schedule conflicting tests in the best possible way. Based on this general approach, three scheduling strategies were devised that differ in the way that they order the test runs and make dynamic scheduling decisions for concurrent test runs.

It is pretty obvious that significant improvements with a parallel execution of test runs can be achieved if there are several machines available and separate installations of the application (and database). In fact, all three strategies that we studied showed linear speed-up for SN. In SN, testing is an embarrassingly parallel operation. If a good scheduling strategy is used (e.g., Slice), then even super-linear speed-ups are possible.

A less obvious result is that a concurrent execution of test runs can speed-up the execution of test runs on a single installation of the application and database; even on a single machine. This scenario was called SDB. This scenario is very common in practice due to limitations in the budget to administrate the test machines and buy multiple software licenses for test installations. This result is less obvious because concurrency in an SDB architecture might incur interference between the concurrent test runs. Never-

Approach	1 thread		5 threads		10 threads		50 threads	
	Time	Reset	Time	Reset	Time	Reset	Time	Reset
Optimistic++	357.9	1788	160.1	1385	157.5	1231	258.0	1425
Slice	306.4	967	120.6	793	112.1	796	259.8	1422
MWD	359.4	1792	164.6	1396	156.0	1251	204.5	1067

Table 7: Running Time (hours), Resets
Synthetic, High Conflict, SDB

theless, the gains that can be achieved by exploiting the resources of a single machine (e.g., multiple processors, disks, and co-processors) by such a multi-threaded execution are higher than the additional cost (more resets) due to interference. Using real test runs from a commercial database application and a real industrial-strength test environment, it could be shown that a speed-up of a factor of three could be achieved by executing test runs concurrently in an SDB architecture. By the means of simulation, it could be shown that only for a high degree of concurrency (50 or more test threads), the performance deteriorates due to interference. Overall, for an SDB architecture, having a good scheduling strategy (such as Slice) is more important than for the SN architecture.

The initial results obtained in this study are encouraging. Nevertheless, there is need for future work. First, it is possible to think of more sophisticated scheduling strategies (e.g., based on machine learning techniques). That way it might be possible to get even better results. Furthermore, there is room for improvement with respect to the scheduling of the reset operation in the SDB architecture (Section 4.4). Another important avenue for future work is to study a two-step scheduling approach for a combination of SN and SDB. In such a scenario, there are several installations of the software (SN), but each installation is exploited in the best possible way applying the strategies for SDB. The two-step scheduling approach has a *global* scheduler for SN and a local scheduler for SDB; both of these schedulers share the same conflict information. Furthermore, an important open question is how to dynamically control the multi-programming level (number of concurrent test threads) for SDB; we plan to adopt ideas from adaptive load control techniques to avoid lock contention thrashing in databases [11]. Finally, we still believe that the whole field of testing database applications is still in its infancy. As listed in [8], there are several aspects that nobody has ever studied; an example is testing non functional requirements such as scalability of a database application.

References

- [1] NI TestStand. <http://zone.ni.com/zone/jsp/zone.jsp>.
- [2] K. Beck and E. Gamma. Programmers love writing tests., 1998. <http://members.pingnet.ch/gamma/junit.htm>.
- [3] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An AGENDA for testing relational database applications. *Software Testing, verification and reliability*, 2004.
- [4] M. chou Chang and F. Lai. Efficient exploitation of instruction-level parallelism for superscalar processors by

- the conjugate register file scheme. *IEEE Trans. Comput.*, 45(3):278–293, 1996.
- [5] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Comm. of the ACM*, 35(6):85–98, 1992.
 - [6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
 - [7] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD Conference*, pages 243–252, 1994.
 - [8] F. Haftmann, D. Kossmann, and A. Kreutz. Efficient regression tests for database applications. In *Conference on Innovative Data Systems Research (CIDR)*, pages 95–106, 2005.
 - [9] F. Haftmann, D. Kossmann, and E. Lo. Parallel execution of test runs for database application systems. Technical report, ETH Zurich, 2005.
 - [10] A. Kemper, D. Kossmann, and F. Matthes. SAP R/3: A database application system (tutorial). In *SIGMOD Conference*, page 499, 1998.
 - [11] A. Mönkeberg and G. Weikum. Performance evaluation of an adaptive and robust load control method for the avoidance of data-contention thrashing. In *VLDB*, pages 432–443, 1992.
 - [12] A. Neufeld, G. Moerkotte, and P. C. Lockemann. Generating consistent test data for a variable set of general consistency constraints. *VLDB J.*, 2(2):173–213, 1993.
 - [13] D. S. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Trans. Softw. Eng.*, 23(3):146–156, 1997.
 - [14] D. R. Slutz. Massive stochastic testing of SQL. In *VLDB*, pages 618–622, 1998.
 - [15] I. Sommerville. *Software Engineering (5th ed.)*. Addison Wesley Longman Publishing Co., Inc., 1995.
 - [16] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.