Coverage-Based Reduction of Test Execution Time: Lessons from a Very Large Industrial Project

Thomas Bach*
Institute of Computer Science
Heidelberg University
69120 Heidelberg, Germany
thomas.bach@stud.uni-heidelberg.de

Artur Andrzejak
Institute of Computer Science
Heidelberg University
69120 Heidelberg, Germany
artur.andrzejak@informatik.uni-heidelberg.de

Ralf Pannemans
SAP SE
Dietmar-Hopp-Allee 16
69190 Walldorf, Germany
ralf.pannemans@sap.com

***Abstract*—There exist several coverage-based approaches to reduce time and resource costs of test execution. While these methods are well-investigated and evaluated for smaller to medium-size projects, we faced several challenges in applying them in the context of a very large industrial software project, namely SAP HANA. These issues include: varying effectiveness of algorithms for test case selection/prioritization, large amounts of shared (non-specific) coverage between different tests, high redundancy of coverage data, and randomness of test results (i.e. flaky tests), as well as of the coverage data (e.g. due to concurrency issues).**

We address these issues by several approaches. First, our study shows that compared to standard algorithms, so-called overlap-aware solvers can achieve up to 50 % higher code coverage in a fixed time budget, significantly increasing the effectiveness of test case prioritization and selection. We also detected in our project high redundancy of line coverage data (up to 97 %), providing opportunities for data size reduction. Finally, we show that removal of coverage shared by tests can significantly increase test specificity.

Our analysis and approaches can help to narrow the gap between research and practice in context of coverage-based testing approaches, especially in case of very large software projects.

I. Introduction

Testing software frequently and testing changes early are the key principles of continuous integration, a quality assurance (QA) process widely adopted in today's software projects. An assumption made here is that execution of relevant tests is ideally a lightweight task, with only moderate waiting time imposed on developers and low usage of computational resources. In large software projects, this is frequently not the case. The reasons can be manifold: large share of time-consuming integration tests, scale of the project, or particularly high requirements on software quality. For example, in the considered industrial project SAP HANA the cumulative execution time of all tests is in the range of 250 hours. Even with a dedicated cluster to execute tests, additional measures are necessary to reduce the waiting time for test results to acceptable levels.

Problems of this kind have been addressed by researchers and practitioners alike: 'Regression testing can be expensive, and the need for cost-effective techniques has helped it emerge as one of the most extensively researched areas in testing over the past two decades' [1]. Among different options, three approaches proved to be effective: test case prioritization, test case selection, and test suite reduction. These techniques filter or reorder test cases to optimize some criterion. We consider *maximizing code coverage* as the optimization objective, understanding it as a proxy for thoroughness of testing [2]. Thus, we use e.g. test case selection to identify test cases which cumulatively maximize code coverage for some fixed test time budget.

Although these methods are extensively researched, we have faced multiple challenges for applying it for SAP HANA. The first issue is the choice of algorithms for test case prioritization and selection. We found out that so-called *overlap-aware* algorithms can achieve significantly better results than traditional ones. Such algorithms consider the size of coverage *overlap* (intersection) between tests subjected to selection or prioritization. This can yield up to 50 % larger code coverage within a time budget.

Another class of issues is related to code coverage data. We found here several phenomena rarely addressed in prior research. One of them is that in system tests a large share of coverage can come from 'shared' functionality such as startup code or internal libraries. This makes it harder to distinguish tests based on coverage, and poses problems for e.g. change-centric testing. A further discovery is high redundancy of (line) coverage data. This means that e.g. either all or none of lines in a group of lines are covered by a test. Such redundancy offers opportunity to significantly reduce the size of coverage data. Finally, we have also observed and studied randomness in terms of test results (i.e. flaky tests [3]) as well as in terms of covered code.

In addition to analysing these challenges, we also propose and evaluate solution approaches in several cases. In particular, the contributions of this paper are as follows:

- We identify and discuss several challenges for efficient testing and code coverage analysis in a large-scale industrial project and confront them with results of previous work (Section III).
- We compare algorithms for test case selection and prioritization, and show that for SAP HANA an overlap-aware approach provides significantly better

*This work was conducted within a dissertation work at Heidelberg University in collaboration with SAP SE.

results than traditional algorithms. This complements and confirms previous studies on small projects (Section IV-B and Section V-B1).
- We investigate the redundancy of line coverage in SAP HANA and find redundancy levels up to 97 % (Section IV-C). This allows for scalable and simpler analysis and visualization of coverage data after a straightforward compression (Section V-B2).
- We discover significant amounts of shared coverage (i.e. code regions covered by many tests) and analyze them. We also design approaches to mitigate this problem (Section IV-D), and show that our method increases the specificity of coverage data (Section V-B3).
- We propose methods for reduction of random coverage, i.e. randomly covered lines (Section IV-E).

II. Testing of SAP HANA

We introduce SAP HANA, the application in focus of our study and provide relevant details about its testing environment and testing practices. Due to confidentiality reasons, we cannot disclose all project details.

A. SAP HANA and its Scale

The observations, empirical data, and evaluation presented in this work are based on the testing practices for SAP HANA [4], a high-performance, parallel inmemory database management system developed by SAP. SAP HANA is a very large software project with several millions lines of source code, mainly written in C and C++. The code basis combines and integrates several subprojects with a lifetime of more than 10 years.

Since many customers use SAP HANA in mission-critical scenarios, quality assurance of this product is of paramount importance. This requirement is ensured via extensive software testing practices in all development and release stages. To illustrate, there exist over 1000 test suites with cumulatively more than 130 000 tests. If executed sequentially, the total runtime of these tests (even with optimized builds) would amount to roughly 252 hours or 10.50 days (as of November 2016). In practice, tests are executed in parallel on a large cluster of servers. Actual runtimes depend on cluster size, cluster load factor and test configuration.

B. Test Organization

There are two relevant hierarchies in HANA's test environment: the test code hierarchy and the test execution (or deployment) hierarchy.

The test code is organized in *test suites*, each containing between 1 to 19 472 *tests* - 'atomic' fragments of test code.

A test suite is typically a Python file embedded in a custom testrunner framework. It can contain different types of tests, mainly: system tests and unit tests. A regression test is typically implemented as one or more SQL queries. Such a query is executed on a prepared database instance, the result is checked for correctness. Contrary to this, unit tests call directly code fragments (e.g. a method) using a test framework, and check the results.

The test deployment hierarchy determines which test suites are executed together (for the same build). One way of grouping test suites is a *test profile*. It can correspond to a project component (source module), or it can organize test suites by other criteria.

Furthermore, within a release cycle, there are multiple barriers (for milestones, for releases, for performance) with larger groups of tests. In addition, developers can execute individual tests or test suites according to their own requirements.

C. Code Coverage Data

The QA team of SAP HANA uses *DynamoRIO drcov* to regularly collect code coverage data for the test suites. Specifically, the largest bulk of the collected data is *line coverage*, i.e. information whether each particular source code line was executed ('hit') or not during a test run. Currently, coverage data of all tests in a test suite is merged and stored for each test suite. It would be surely interesting to use finer grained granularities than line coverage (i.e. on branch or statement or instruction level), but previous internal studies showed that collecting coverage data with more fine-grained granularity than line coverage significantly increases test runtimes and size of coverage data. Therefore, line coverage is used as a suitable compromise between accuracy and resource usage.

Even for line coverage, the cumulative runtime of tests with enabled instrumentation for collecting coverage data is about 662 hours or 27.50 days. A typical coverage run still needs 1 days to 2 days if executed in parallel on the test cluster. Moreover, each such coverage run generates about 130 GB of code coverage data.

III. Gap between Research and Practice

The size and complexity of SAP HANA and its testing environment exceeds by several orders of magnitude many previous studies related to test time reduction (see Section VII). As a consequence, we had to face several issues which gained less attention in academic research. This gave rise to several research challenges discussed below, namely: effective algorithms for reducing testing time (Section III-A), reliability of test results (Section III-D), specificity of coverage data (Section III-C), as well as its size (Section III-B) and quality (Section III-E).

A. Effective Test Suite Prioritization and Selection

The scale of the considered software project calls for reducing the amount of (computational) resources involved in the execution of tests, and for shortening the execution time of the test portfolio. Among several methods to achieve these goals, the most generic one is *test suite reduction.* It attempts to transform the set of tests to some subset or different set of tests with identical or similar efficiency in fulfilling the original testing goals. The testing goal is typically the detection of (all) test failures.

Given the high effort of this method (which sometimes requires rewriting test code or at least changing their hierarchy), we focus on two techniques which are easier to implement. The first is *test case prioritization* (*TCP*): TCP reorders a sequence of tests to meet some (related) testing objectives faster. In our case the objective is increased code coverage with each additional test suite. With increased coverage in shorter time, we try to reduce the time until a possible failure is found, thus shortening time until a developer gets a first feedback.

The second considered technique is *test case selection* (*TCS*). TCS selects a subset of all tests that cumulatively optimize some criterion given a constraint. In this case, our objective is to maximize code coverage by the selected test suites for a given time budget. The corresponding combinatorial problem is the *Budgeted Maximum Coverage Problem*. An alternative objective is to find a subset of test suites with minimum runtime which still achieve full code coverage (modeled by the *Weighted Set Cover Problem*).

Test selection and prioritization have been extensively researched in a wide range of variants, and many heuristic algorithms have been proposed and evaluated (Section VII). In context of our project, we implemented and compared several established techniques for both TCP and TCS and evaluated TCP in detail. As it turned out, the highest impact of the effectiveness (e.g. time reduction) of the approach was the fact whether it is *overlap-aware*, or not.

In our context this means that an *overlap-aware* algorithm considers during its optimization a possible *coverage overlap* (i.e. size of coverage intersections) between the coverage data of multiple test runs. E.g. if test A hits 100 lines (set L_A), test B covers 90 lines (set L_B), and test C covers 50 lines (set L_C), a non-overlap-aware algorithm might propose to run tests A and B first (prioritization), or only tests A and B (selection). However, if e.g. $L_B \subseteq L_A$ (i.e. test B does not provide additional coverage in respect to test A), and $L_A \cap L_C = \emptyset$, the more efficient choice of tests is A and C (since tests A and C cover together 150 LOC, but tests A and B together only 100 LOC). The latter choice requires overlap-aware algorithms as these consider the sizes of intersections between L_A, L_B, and L_C. Overlap-aware algorithms are also known in literature as additional statement coverage algorithms, or additional greedy heuristics, among others (Section VII).

Despite of the impact of this property, existing evaluations of overlap-aware algorithms were conducted on comparatively small projects. Our study shows (Section V) that in case of our target project using an overlap-aware algorithms has a very significant effect of the amount of potential time savings. Of course, this high impact is related to the significant amount of coverage overlap for the test suites of SAP HANA, which can be less pronounced in other projects. We argue that to be on the safe side only overlap-aware algorithm should be used for coverage-based TCP and TCS optimization problems.

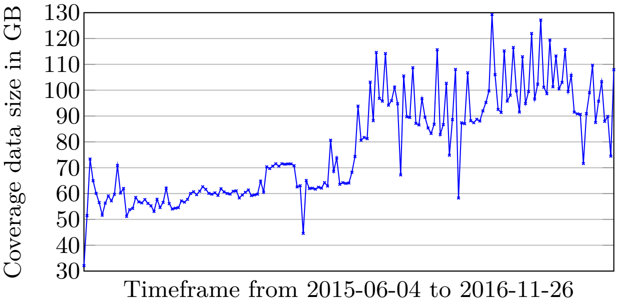

Figure 1. Growth of data size for coverage runs over one year (coverage for all test suites). Test changes lead to spikes.

B. Size of Coverage Data

The second problem less pronounced in research literature is size of the coverage data and scalability of its storage and processing. In case of SAP HANA, the cumulative size of raw coverage data for one coverage run has currently reached approx. 130 GB (for more than 1 000 test suites). The evolution of this size over one year is shown in Figure 1. For the purpose of more advanced analysis, e.g. discovery of trends and comparative studies, several months worth of data is stored. This yields currently 14 TB of data.

Apart from the (still manageable) storage requirements, processing of such data requires higher computational capacities. Processing the results of a single coverage run can require a matrix with more than 10^9 entries (with observed density of 9 % to 12 %). Processing and inspecting such matrices require more sophisticated, scalable software.

To reduce the computational requirements of data analysis and facilitate using simpler tools, we investigated methods to reduce the size of the data by exploiting the inherent redundancies. As result, data size could be significantly reduced, as discussed in Section V-B2.

C. Shared Coverage

Integration tests typically execute frequently-used program code to perform application startup, tear-down, and a set of core application functions (e.g. parsing of SQL, query execution). Furthermore, both system tests and unit tests call code from many project-internal libraries, e.g. memory management subsystem or string manipulation. This gives rise to a concept of *shared (functionality) coverage*. We define it informally as line coverage information corresponding to frequently executed code including startup/teardown, core functionality, and common application routines. Technically, the shared coverage can be identified as a set of source lines covered by at least k tests, with parameter k depending on the project (see Section V).

In general, shared coverage blurs the differences between tests in terms of their coverage, creating several detrimental effects. First, it clearly reduces the specificity of the coverage information, making it more difficult for developers to decide which tests address which parts of code. Analogously, this creates barriers to utilize techniques such as change-centric testing. In context of test clustering, shared coverage can pose a challenge for common distance metrics (e.g. Jaccard's similarity metric) as large shared coverage might dominate more subtle coverage differences. Another problem caused by shared coverage is the increased size of coverage data.

In context of SAP HANA we observed a significant degree of shared coverage: approximately 20 % of the covered lines can be found in 80 % of all test suites. Consequently, identification and removal of shared coverage is an important processing step to improve coverage-based characterization of tests.

By removing shared coverage and increasing the specificity, we are able to improve the understanding of relationship between test and parts of the code that are targeted by the test. Removing shared coverage is highly beneficial for further methods like test prioritization, clustering, or the analysis of test quality. Furthermore, such processing can significantly reduce the size of coverage data. We present approaches for filtering shared coverage in Section IV-D, and evaluate them in Section V.

D. Flaky Tests

Flaky tests are tests which show different results (pass/-fail) in multiple runs under the same conditions (inputs, local environment etc.) [3]. The reasons for such behaviour are diverse, and include issues with the test environment (e.g. file servers), performance impact of other applications, 'junk' data created by previous tests, and randomness due to concurrency issues in the application or the operating system. Another reason could be defects inducing memory leaks. However, such issues are quite unlikely due to elaborated memory management mechanisms of SAP HANA, and we have not observed those.

Empirical data from Google[1] indicates that up to 16 % of all tests have flaky behaviour in large real world projects. In case of SAP HANA, this number is lower, but still we experience a non-negligible number of flaky tests.

Flaky tests create a threat to validity of results for approaches exploiting historical test results, and to a certain degree also for research based on coverage data. The majority of papers in this domain assume perfectly stable test conditions, and repeatable, deterministic test results. This might not be the case for projects above a certain size, which calls for evaluation of such approaches for very large projects.

To eliminate the impact of flaky tests on results in this work, we run a test up to four times if a previous run fails, and we keep the coverage of failed test runs. It is worth mentioning that a previous project in context of SAP HANA has attempted to classify test results as correct or erroneous utilizing techniques from machine learning (specifically, scalable SVMs), with good results. We do not describe it here due to space limits.

E. Random Coverage

We understand by *random coverage* the phenomenon that a line of code is sometimes executed, and sometimes not executed (and thus covered or not) in multiple runs of the same test yielding the same test result. Thus, we see here randomness at the level of coverage data, not at the level of test results (as in the case of flaky tests). The effect of random coverage was already observed in other work, e.g. [5]. Apart from obtaining less reliable coverage data, random coverage creates issues for test distance and similarity metrics, and in general for all techniques which depend on accurate and sound coverage information. We have detected multiple sources of random coverage: random functions, time/date creation, multi-threaded execution, memory handling, scheduling, file system interactions, and errors of coverage tools.

For some test suites in SAP HANA, we found differences of covered lines for two identical coverage runs from 50 to several hundred lines. This is only a tiny fraction of the total number of lines covered by a test suite which typically ranges (after shared coverage removal) from 10 000 to 200 000. However, if line coverage of two such test suites *differ* by e.g. only 100 LOC, the random coverage becomes critical, as we do not know which fraction of this difference can be attributed to randomness. We present an approach to address this problem in Section IV-E.

IV. Approach

We describe our approaches for selected problems from Section III: algorithms for effective test case selection and prioritization (Section IV-A), analysis and removal of the redundancy of line coverage data (Section IV-C), filtering of shared coverage data (Section IV-C), and elimination of random coverage (Section III-E).

A. Notation

To simplify notation and the description of algorithms, we introduce the notions and symbols for summation (*add* or $+$) and subtraction (*sub* or $-$) of coverage data. Essentially, these binary operations implement the union and difference, respectively, over the sets of lines covered by two tests. A more formal definition reads as follows. Let a, b be two coverage files, i.e. sets of lines covered by a test, possibly spanning multiple source files. Then $a + b$ is the set of lines included in a or in b (i.e. $a \cup b$), and $a - b$ is the set of lines included only in a but not in b (i.e. $a \setminus \mathrm{b}$).

B. Effective Test Case Selection and Prioritization

Test case selection. We focus first on the issue of reducing cumulative time required for test runs and consider two alternative problem formulations: (i) Given a fixed time budget for test runtime we attempt to find a subset of tests with maximum coverage, and (ii) we try to find a subset of tests with minimal runtime which cumulatively achieve the best possible coverage (understood as cumulative coverage by all available tests).

[1] https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

Table I
Example coverage data. x denotes that a test hits a line

	t_1 (16 s)	t_2 (15 s)	t_3 (15 s)
L1	x	x	
L2	x	x	
L3	x		x
L4		x	x
L5	x		x

More formally, let R be the set of all available coverage files $c_1, \ldots, c_n$ (with c_i corresponding to a test i). For $P \subseteq R$ let $time(P)$ denote cumulative time to execute all tests specified by P. Furthermore, let T be a time budget (threshold), and assume that the sums are using the add $(+)$ operation according to Section IV-A. Then the Test Case Selection (TCS) problem can be expressed in either one of the following dual formulations:

TCS1 Maximize $|\sum_{c\in P} c|$ with $time(P) \leq T$ and $P \subseteq R$,
TCS2 Minimize $time(P)$ with $\sum_{c\in R} c = \sum_{c\in P} c$, $P \subseteq R$.

In both cases, we optimize over the set $P \subseteq R$ which directly translates to finding an optimal set of tests. As an example, we use Table I. For TCS1 the subset with maximal coverage for a time budget of 15 s is $\{t_2\}$, for a time budget of 20 s it is $\{t_1\}$, and it is $\{t_2, t_3\}$ for a time budget of 30 s. For TCS2, the subset with maximum achievable coverage and minimal time is $\{t_2, t_3\}$.

TCS1 can be formulated as a variation of the general 0/1 knapsack problem ([6]):

$$\text{maximize} \quad \left|\sum_{j=1}^{n} p_j * x_j\right| \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq T \tag{2}$$

where:

$$\begin{aligned}
&x_j \in \{0,1\}, \quad j = 1, \ldots, n\\
&T : \text{budget on weights (i.e. time budget)}\\
&M : \{1, \ldots, n\} \ \text{set of items (i.e. tests)}\\
&p_j : \text{"profit" of item } j \in M \quad \text{(i.e. coverage file)}\\
&w_j : \text{weight of item } j \in M \quad \text{(i.e. execution time).}
\end{aligned}$$

In Equation (1) we use (with $P = \{p_1, \ldots p_k\}$):

$$\begin{aligned}
&p_i + p_j : \text{as defined in Section IV-A}\\
&p_i * x_i : \emptyset \ \text{if} \ x_i = 0, \ \ p_i \ \text{otherwise} \ (x_i \in \{0,1\}).
\end{aligned}$$

Note that the variable x_j indicates whether a test j was selected ($x_j = 1$) or not ($x_j = 0$). In Equation (1), we can see that TCS1 is similar to the 0/1 knapsack problem, but the major difference is the overlap of coverage elements p_j.

Problem TCS1 is also known as the *Budgeted Maximum Coverage Problem* (BMCP) [7]. Khuller et al. propose a modified greedy heuristic to solve this problem time-efficiently: at each step, select the next item with the best coverage over runtime ratio and with suitable size, add it to the solution; after each step, remove the covered lines in the current item from all other items.

We also consider a parallel variant of the greedy algorithm from Khuller et al. For a parallelization factor of p, we use p buckets, each bucket with a time budget of T/p. We distribute the next selected item over these p buckets. Again, we choose a greedy selection strategy: always choose the bucket with the largest free place.

Problem TCS2 is known as the *Weighted Set Cover Problem* (WSCP). To solve this problem, we can modify the greedy algorithm outlined above by adding a pre- and postphase, and dropping the time budget. During the prephase, we find mandatory (or must-have) tests, i.e. the *only* tests covering certain source lines. During the postphase, we order all tests by the efficiency, i.e. ratio of number of covered lines over time, check for each test t_n if the test is already included in $\sum_{i=1}^{n-1} t_i$ and continue only with tests which are not already included. However, we did not evaluate this approach due to space limits.

Note that both test case selection strategies are not safe, i.e. they can omit test cases that would otherwise have detected faults. This must be considered during a risk analysis of the costs and benefits. More information about safe selection strategies can be found e.g. in [8].

Test case prioritization. We can use the solution set from TCS as input for the Test Case Prioritization (TCP) problem. This requires test reordering so that tests with the best ratio of code coverage to runtime run first. This is also called *time-aware test suite prioritization* [9], [10].

Note that in the general case we can only find a best order if the exact amount of time (or tests) is fixed. We illustrate this by the example from Table I. We run t_1 with 4 s/line first. This is the best solution for a solution set with only one element, because t_2 and t_3 have 5 s/line. The next test to add to the existing solution set must be either t_2 or t_3 which gives us a total runtime of 31 s for 5 lines. But we have done overall better if we choose $\{t_2, t_3\}$ with a total runtime of 30 s for 5 lines. This example shows that we cannot find the best solution for all cases (without a fixed time budget or a fixed amount of tests).

C. Removal of Redundancy for Line Coverage Data

We analyse the redundancy of coverage for our target project following the concept of compact coverage section from [11]. The main idea is to first identify sets (or groups) of lines which are either *all covered* or *all not covered* by each test case. We can then compact the coverage data by recording for each such line group whether it was hit by a test or not, and optionally saving the location of the original lines per group.

One of the reasons for existence of such groups are code blocks (i.e. sequences of consecutive code lines) without branches. The following listing shows such a block:

Table II
Result of redundancy removal for data in Table I

	t_1	t_2	t_3
$L1, L2 \rightarrow L12$	2	2	
$L3, L5 \rightarrow L35$	2		2
$L4 \rightarrow L4$		1	1

```
int example_function(int a, int b) {
  int c = a + b;
  int d = a - b;
  return c*d;
}
```

We can compact the coverage data of the above block (five lines) to one entry with the weight of five. For some scenarios, even this weight is not needed, and so we can just remove four lines without any further implications.

With appropriate data structures, the algorithm for identifying such groups of lines is straightforward: we search for all sets of lines with identical behaviour for all tests. Table II shows the results of this technique applied to the example from Table I.

We highlight two differences from [11]. First, the algorithmic description in [11] has a small flaw in line 7. The delete operation will invalidate the index, which could easily be fixed by a temporary variable. Second, the algorithm has a quadratic worst case runtime for the number of lines. For our case example with several million lines, this would not be feasible. This can be changed by using a HashMap to store and count the line columns to an amortized linear runtime for the number of lines.

D. Filtering Shared Coverage Data

To filter shared coverage, we have to identify and then remove it. Since the latter step is a technical detail using the sub operation from Section IV-A we focus only on identification of shared coverage. We have investigated the following three different approaches.

1) Shared Coverage Removal based on Testcount: Our main idea is based on the fact that shared coverage lines are hit by more tests than other lines. We can thus proceed in two steps: (i) compute for each covered line the testcount, i.e. number of tests that hits this line; (ii) mark (or immediately remove from coverage data) all lines with a higher testcount than a given threshold.

For small threshold values, e.g. 1, the algorithm removes all shared coverage lines. For high threshold values, this will remove only lines which are covered by nearly all tests. Both extremes are not optimal. An example for removing too much would be two tests that cover two branches of an if statement in a function. We cannot prevent that they both cover the part before the if statement, but a removal is not desired. If the threshold value is too high, we will not remove anything. For instance, a test which expects a crash or exception would not execute all shared parts. Therefore we have to identify a reasonable threshold. We find this threshold by analysing the distribution of the different testcount values versus the amount of lines with the given testcount in Section V-B3.

2) Shared Coverage Removal based on a Baseline Test: Our main idea in this variant is the assumption that shared coverage occurs because part of the code has to be executed for all or most inputs. This gives rise to creating a test which executes only such code.

We implemented it as a test with only the most fundamental functionality. In our case (SAP HANA is a database system) this is a test with simple create, insert, select, delete, and drop statements. The coverage collected for this test is the baseline and can be removed from all other coverage files. As an alternative for creating the baseline test manually, it is also possible to find a suitable candidate automatically with a combination of the previous approach based on testcounts.

3) Shared Coverage Removal based on Directories: A typical software project has a hierarchical directory structure based on rules or conventions how to organize related source files. In our case (which we believe is common in larger software projects) the common and auxiliary functionality (e.g. memory or string handling) is within its own directory. Therefore, we can remove all coverage for source files in this directory. We do not provide an evaluation for this approach, because the effect is obvious and the approach only solves a subset of the problem.

E. Random Coverage

Randomness in coverage data is a threat for the soundness of further analysis. The best approach would be to remove all source of randomness indicated in Section III-E, which is obviously not feasible in a large project.

In our current approach, we rerun a test multiple times to harmonize the different sets of lines hit created by randomness. We can either create the union of coverage data for all runs (which contains all randomly covered lines) or create the intersection for all runs (which does not contain any randomly covered line). Manual evaluation of this approach for single test suites showed reasonable results. However, we skipped an evaluation on a larger scale due to the constraints on running time and resource costs.

V. Evaluation

A. Research questions

We investigated the following research questions in context of the large-scale software project SAP HANA:

RQ1 Does an overlap-aware heuristic solver for a *Budgeted Maximum Coverage Problem* (TCS1) produce better results than non-overlap-aware solvers?

RQ2 How high is the redundancy of coverage data (in terms of source code lines with equal behaviour) for all tests, and does it change over time?

RQ3 How does removal of shared coverage improve the specificity of code coverage data?

Table III
Required time budgets for BMCP to reach different percentages of total coverage (cases span one year)

Coverage run	Greedy variant	Req. time budget (hours) for		
		90 %	99 %	100 %
2015-11-15	standard	74	137	137
	overlap-aware	19	57	123
2016-05-19	standard	110	182	191
	overlap-aware	25	70	173
2016-10-25	standard	108	193	219
	overlap-aware	24	72	196

All algorithms from Section IV and the evaluation approaches have been implemented as proprietary code in Python and Java.

B. Answers to research questions

1) Overlap-Aware Algorithm: To answer RQ1, we compare the effectiveness of the overlap-aware greedy (OAG) approach outlined in Section IV-B against a standard greedy (SG) implementation.

We apply all algorithms on the same coverage data with different time budgets (increased in 1h steps). For each time budget, we get a solution in terms of sum of lines hit – the higher the sum, the better. All algorithms run in less than 10 seconds, therefore we do not report these times.

Figure 2 shows the results for one coverage run, while Table III exhibits the results of different coverage runs over one year. OAG converges to a high coverage and reaches the maximum significantly faster than SG. The evaluation shows that SG has similar behaviour to a random shuffle and in some cases it is even worse than a random guess. OAG is significantly better, with a factor of 1.40 up to 1.50 for a time budget of up to 30 hours.

A comparison between OAG and SG for Test Case Prioritization leads to nearly identical results. We have also evaluated the impact of parallelization as described in Section IV-B with parallelization factors ranging from $p = 1$ up to $p = 50$. The relative savings are almost identical, the difference are less than 0.01 %. Therefore, we omit the results for space reasons.

2) Analysis and Removal of Line Coverage Redundancy: To answer RQ2, we apply the algorithm described in Section IV-C for different coverage runs. For each coverage run, we calculate the redundancy, which is the amount of redundant lines over total lines. Table IV shows that at most 110000 'line groups' are sufficient to represent the complete coverage with about 3000000 lines.

3) Shared Coverage Removal: We answer RQ3 in 3 steps:

(i) We identify shared coverage with the following two approaches from Section IV-D: **testcount removal**, and **baseline removal**.
(ii) We remove the shared coverage.
(iii) We analyse test specificity before and after removal.

Step (i). For the approach based on **testcount removal**, we need a threshold for removing lines with a high testcount. To find the threshold, we analyse the distribution for testcounts per source code lines, see Figure 3. We select the threshold DB_{80} at 80 % which corresponds to 238 test suites, i.e., every line hit by more than 238 test suites is removed. We select two additional thresholds for our evaluation below and above DB_{80}: DA_{60} and DC_{90}. To automate the threshold selection, it is possible to utilize the slope of the distribution curve.

Table IV
Coverage redundancy for different coverage runs

Coverage run	Lines hit	Line groups	Redundancy
2015-11-15	2 901 575	79 741	97.25
2016-05-19	3 172 337	93 162	97.06
2016-08-04	3 371 109	97 368	97.11
2016-10-25	3 510 727	104 764	97.02
2016-10-27	3 501 611	104 355	97.02
2016-10-29	3 422 442	107 402	96.86
2016-11-01	3 421 780	104 837	96.94
2016-11-03	3 399 853	104 638	96.92
2016-11-05	3 424 585	109 338	96.81
2016-11-07	3 413 424	105 235	96.92
2016-11-10	3 405 657	105 361	96.91
2016-11-12	3 391 712	108 754	96.79
2016-11-15	3 436 853	106 030	96.91

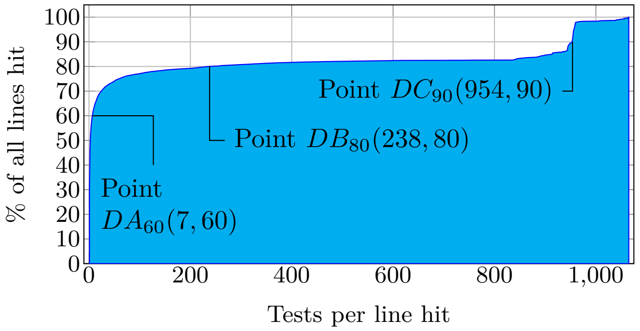

Figure 3. Exemplary distribution plot for testcount versus the relative amount of covered lines with at most testcount value of tests. E.g. 80 % of all lines hit are covered by ≤ 238 test suites and 31 % of all lines hit are covered by ≤ 1 test suite

For the **baseline removal** approach, we manually chose the baseline test suite in consultation with SAP engineers.

Step (ii) After the identification step, the removal step uses the *sub* operation introduced in Section IV-A, either with the baseline test suite or with an artificial coverage file made of all lines with a testcount above the current threshold. Table V shows the size reduction after the *sub* operation. The baseline removal reduced the coverage size to 36.01 % of the original coverage.

Step (iii) After the removal, we measure the test specificity before and after applying our approaches. We evaluate the specificity by counting the number of lines hit in source code located in different (component-specific) subdirectories and asking SAP engineers to verify whether the directory relates to the test intent or not.

We select 20 test suites randomly and compute the top five coverage directories for each test suite. These are the

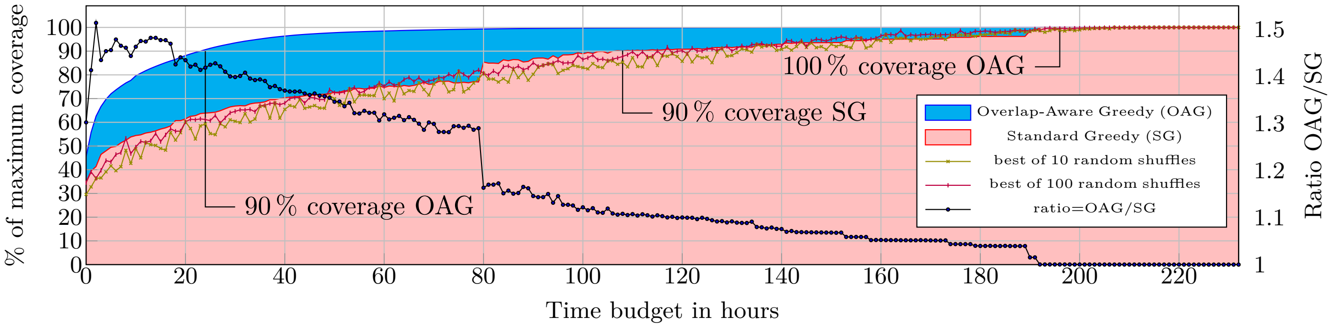

Figure 2. Exemplary comparison between different algorithms for maximum budgeted coverage problem. Higher is better

Table V
Relative size of coverage data after removal of lines with high testcounts. E.g. the percentage of lines which are hit by ≤ 238 test suites is 80 %. Removal of lines with testcount > 238 reduces the data to 15.06 % of the original size

Threshold	Testcount	Size left
0 %	0	0.00 %
30 %	0	0.00 %
60 %	7	1.93 %
70 %	25	4.64 %
80 %	238	15.06 %
90 %	954	58.44 %
99 %	1046	95.13 %
100 %	1065	100 %

Table VI
Evaluation of test specificity changes for shared coverage removal. Score depends on manual evaluation (see Section V), higher is better. Correct directories shows how many directories are correctly identified

	Case	Correct dir.	Score	Lines hit
DA_{60}	original	1/10 (7 wrong)	$-28-25=-53$	2 737 700
	removed	8/10 (0 wrong)	$5+10=15$	37 848
DB_{80}	original	3/10 (7 wrong)	$-35-26=-61$	3 811 208
	removed	8/10 (1 wrong)	$-16+9=-7$	292 087
DC_{90}	original	0 (10 wrong)	$-46-46=-92$	3 038 125
	removed	1/10 (9 wrong)	$-30-20=-50$	1 178 414

directories with the highest number of lines hit for the source files within the directory. Under the assumption that the directory layout of the filesystem represents a coupling between source files and modules, we should see an increase in specificity for the top directories.

We can represent the 20 test suites and five directories for each test suite by a 20×5 matrix. We created this matrix for six configurations: for each threshold DA_{60}, DB_{80}, and DC_{90} (see Figure 3), we have one version with the original data and one after the removal of lines with high testcount.

We showed these six matrices to test engineers from SAP and asked them to select 10 test suites where they can judge which directories are specific to this test and which ones are not. They should highlight all specific directories within the top 5 directories in green, the unspecific directories in red and the rest in yellow.

After this manual task, we attribute the following scores to the classifications: A green field gets +1, a red field −1, and a yellow field gets 0. In addition, we add a top score for the first n correct top directories: If the top n directories are green, we add additional n points; for red, we substitute n points. For instance: (g,g,r,y,y) gets $(2-1)+2 = 3$ points; (y,r,r,r,g) gets $(1-3)+0 = -2$ points.

Table VI shows the results of our evaluation. The shared coverage removal improves the specificity of the coverage. The highest increase in test specificity was possible with the highest testcount removal. This is expected since the highest specificity would be reached if we only keep lines which are hit by exactly one test. But as discussed in Section IV-D, this is not desired. This choice would remove all covered lines with multiple tests, but some of them are still interesting for further analysis. In summary, the 80 % threshold provides a good increase in test specificity but still guarantees multiple coverage for most code parts.

We have evaluated the baseline removal approach in the same way. After the baseline removal, we got 4/10 (6 wrong) correct top directories, which corresponds to a score of $-24-6=-30$, and 728 794/3 811 208 lines hit. In comparison to the DB_{80} testcount result, the test specificity is thus lower.

VI. Threats to Validity

We discuss here possible threats to validity of our work.

A. Test Suites Granularity

Our coverage data is based on test suites. A test suite includes a set of tests. The coverage contains the whole test suite, we can not extract a single test. This complicates similarity analysis and investigation of program flow, but does not affect the approaches in this paper. In practice, even single test cases are not always very specific if they are complex or include several checks in one test case.

B. Flaky Tests

As explained in Section III-D, flaky tests influence all analysis based on test success. In this work, we do not use the test success. But a flaky test could have only partial coverage due to early failure. This is mitigated in two ways. We rerun a failed test multiple times, and we decide in advance if we use failed tests runs as an input or not.

C. Safeness of TCS

Our test case selection strategies are not safe. I.e. they can omit test cases that would otherwise have detected faults. This must be considered during a risk analysis of the costs and benefits. We plan future work on this.

D. Random Coverage

As explained in Section III-E, we observed random coverage. The amount is very low compared to the total coverage size. Therefore, there is a marginal impact on our results. We evaluated multiple reruns to reduce the randomness, but in practice this significantly increases the resource costs and is currently not worth the effort.

E. Relation between Coverage and Bug Finding Ability

We did not evaluate for SAP HANA whether high coverage for a test is correlated with a high ability to find bugs. If there is no correlation (as indicated e.g. in [12], but debated in [13], but other results e.g. in [14]), this would affect the impact of our work. However, even in this case, the results are still useful for techniques on coverage data. We plan to evaluate this aspect in the future.

F. A Database is a Special Environment

By design, every database run uses a large shared part of the database stack. This might make it difficult to generalize our results to other areas. We argue that this shared design applies in fact to a wide field of software branches. Software following the MVC pattern has a large shared part, e.g. GUI software. Also a single entry point is quite common, especially in parser or I/O software.

G. Tests are not independent

Zhang et al. showed that the test independence assumptions does not always hold [15]. We also observed this for tests in a test suite. This does not affect our work, because we use coverage on a test suite level. Test suites are guaranteed to be independent, because each test suite runs completely separated from all other test suites.

VII. Related Work

Resource problems in software testing have received a lot of research attention, see e.g. a survey [8] of previous work on Test Case Selection (TCS) and Prioritization (TCP), and a more general overview in [1]. We limit our discussion to the focus of this paper.

Overlap-aware Algorithms for TCS and TCP. Previous research already investigated the difference between overlap-aware greedy heuristics and non-overlap-aware algorithms for TCS and TCP (or the underlying problem formulations BMCP and WSCP, see Section IV-B). The size of evaluation examples range from 374 to 300 000 lines of code (LOC), see Table VII. In contrast to previous papers, the application used for our evaluation is very large, and has industrial/commercial character. In numbers, it has several million lines of code and over 1 000 test suites with more than 130 000 tests, which is up to several magnitudes larger than previous studies.

Table VII
Related work comparing overlap-aware vs. non-overlap-aware solvers for TCS or TCP. Column 'Term' indicates which term is used for 'overlap-aware'

Work	Size	Term
[16]	5 classes to 22 classes	overlap-aware
[17]	53 testcases to 209 testcases	additional
[18]	374 LOC to 11 148 LOC	additional
[10]	500 LOC to 9 564 LOC	additional
[19]	2 kLOC to 80 kLOC	additional
[20]	7 kLOC to 80 kLOC	feedback technique
[21]	7.50 kLOC to 300 kLOC	additional
Our work	> 3.50 MLOC	overlap-aware

In addition to an overlap-aware greedy algorithm, there are also other heuristics proposed, e.g. evolutionary algorithms and other metaheuristic optimization algorithms [18], [8]. In our case, an overlap-aware greedy produced good enough results in a very fast runtime (less than 10 seconds for all configurations).

Work [10] claims that time-aware TCP makes no significant difference in terms of fault detection compared to standard TCP. In contrast to this results, [9] and [20] conclude that time aware-prioritization improves prioritization techniques. We are not sure if the results for small projects can be generalized to large projects. For time-aware prioritization on our project, feedback from developers indicates to an improvement, but the results are not rigorously checked.

Redundancy in line coverage. An algorithm to filter redundancy in line coverage is proposed in [11], but without stating data on redundancy. To our best knowledge, this work is the first to study and report redundancy figures for a large industrial project.

Filtering Shared Coverage: To our best knowledge, this specific problem is not reported or studied so far. We assume that this phenomenon becomes visible only in larger software projects.

VIII. Conclusions

We described several gaps between current research in regression testing and practical application in large scale projects. We have analysed an overlap-aware greedy approach for test case selection and prioritization, and showed that the overlap-aware variant produces significantly better results for test case selection compared to a standard greedy for our large scale project, up to a factor of 1.50. Our test case selection and prioritization work is already used by SAP for manual analysis with positive feedback from SAP developers. The integration in automatic CI is planned for future integration cycles. An evaluation of the impact is a future research task.

We also studied redundancy of line coverage data and found that up to 97 % of all covered lines do not have a unique behaviour and are redundant. We also addressed the problem of shared coverage, i.e. parts in the software which are executed by most tests. We introduced approaches to reduce them and evaluated the improvement in test specificity after the removal of shared parts.

Gaps in Section III indicate possible future work. We are particularly interested in approaches for identifying and handling flaky tests. An extension of our evaluation could compare other overlap-aware heuristics for large scale applications in terms of solution quality and runtime.

References

[1] A. Orso and G. Rothermel, "Software Testing: A Research Travelogue (2000–2014)," in *Proceedings of the on Future of Software Engineering*, ser. FOSE 2014. New York, NY, USA: ACM, 2014, pp. 117–132. [Online]. Available: http://doi.acm.org/10.1145/2593882.2593885 (Cited from: I and VII)

[2] R. Gopinath, C. Jensen, and A. Groce, "Code Coverage for Suite Evaluation by Developers," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 72–82. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568278 (Cited from: I)

[3] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An Empirical Analysis of Flaky Tests," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 643–653. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635920 (Cited from: I and III-D)

[4] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: Data management for modern business applications," *SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2094114.2094126 (Cited from: II-A)

[5] P. Marinescu, P. Hosek, and C. Cadar, "Covrig: A Framework for the Analysis of Code, Test, and Coverage Evolution in Real Software," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 93–104. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610419 (Cited from: III-E)

[6] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. [Online]. Available: http://link.springer.com/10.1007/978-3-540-24777-7 (Cited from: IV-B)

[7] S. Khuller, A. Moss, and J. S. Naor, "The Budgeted Maximum Coverage Problem," *Inf. Process. Lett.*, vol. 70, no. 1, pp. 39–45, Apr. 1999. [Online]. Available: http://dx.doi.org/10.1016/S0020-0190(99)00031-9 (Cited from: IV-B)

[8] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1002/stv.430 (Cited from: IV-B, VII, and VII)

[9] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "TimeAware Test Suite Prioritization," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1146238.1146240 (Cited from: IV-B and VII)

[10] D. You, Z. Chen, B. Xu, B. Luo, and C. Zhang, "An Empirical Study on the Effectiveness of Time-aware Test Case Prioritization Techniques," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, pp. 1451–1456. [Online]. Available: http://doi.acm.org/10.1145/1982185.1982497 (Cited from: IV-B and VII)

[11] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical Evaluation of Pareto Efficient Multi-objective Regression Test Case Prioritisation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 234–245. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771788 (Cited from: IV-C, IV-C, and VII)

[12] L. Inozemtseva and R. Holmes, "Coverage is Not Strongly Correlated with Test Suite Effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 435–445. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568271 (Cited from: VI-E)

[13] I. Ahmed, R. Gopinath, C. Brindescu, A. Groce, and C. Jensen, "Can testedness be effectively measured?" in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 547–558. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950324 (Cited from: VI-E)

[14] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, 2015, pp. 560–564. [Online]. Available: http://dx.doi.org/10.1109/SANER.2015.7081877 (Cited from: VI-E)

[15] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, San Jose, CA, USA, July 23–25, 2014, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610404 (Cited from: VI-G)

[16] S. Alspaugh, K. R. Walcott, M. Belanich, G. M. Kapfhammer, and M. L. Soffa, "Efficient Time-aware Prioritization with Knapsack Solvers," in *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, ser. WEASELTech '07. New York, NY, USA: ACM, 2007, pp. 13–18. [Online]. Available: http://doi.acm.org/10.1145/1353673.1353676 (Cited from: VII)

[17] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware Test-case Prioritization Using Integer Linear Programming," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 213–224. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572297 (Cited from: VII)

[18] Z. Li, M. Harman, and R. M. Hierons, "Search Algorithms for Regression Test Case Prioritization," *IEEE Trans. Softw. Eng.*, vol. 33, no. 4, pp. 225–237, Apr. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.38 (Cited from: VII)

[19] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the Gap Between the Total and Additional Test-case Prioritization Strategies," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 192–201. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486814 (Cited from: VII)

[20] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "An Empirical Study of the Effect of Time Constraints on the Cost-benefits of Regression Testing," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 71–82. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453113 (Cited from: VII)

[21] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182, Feb. 2002. [Online]. Available: http://dx.doi.org/10.1109/32.988497 (Cited from: VII)