Insert your title here

Do you have a subtitle?
If so, write it here

Nauber Gois · Pedro Porfírio · André Coelho

Abstract Insert your abstract here. Include keywords, PACS and mathematical subject classification numbers as needed.

1 Introduction

Load, performance, and stress testing are typically done to locate bottlenecks in a system, to support a performance-tuning effort, and to collect other performance-related indicators to help stakeholders get informed about the quality of the application being tested [1] [2].

Typically, the most common kind of performance testing for Internet applications is load testing. Application load can be assessed in a variety of ways [3]:

– Concurrency. Concurrency testing seeks to validate the performance of an application with a given number of concurrent interactive users [3].
– Stress. Stress testing seeks to validate the performance of an application when certain aspects of the application are stretched to their maximum limits. This can include maximum number of users, and can also include maximizing table values and data values [3].
– Throughput. Throughput testing seeks to validate the number of transactions to be processed by an application during a given period of time. For example, one type of throughput test might be to attempt to process 100,000 transactions in one hour [3].

Nauber Gois, Pedro Porfírio, André Coelho
Av. Washington Soares, 1321 - Edson Queiroz, Fortaleza - CE, 60811-905
Tel.: +123-45-678910
Fax: +123-45-678910
E-mail: naubergois@gmail.com
E-mail: porfirio@unifor.br
E-mail: acoelho@unifor.br

The performance testing aims at verifying a specified system performance. This kind of test is executed by simulating hundreds of simultaneous users or more over a defined time interval [4]. The purpose of this assessment is to demonstrate that the system reaches its performance objectives [1]. Term often used interchangeably with "stress" and "load" testing. Ideally "performance" testing is defined in requirements documentation or QA or Test Plans [5].

In a load testing, the system is evaluated at predefined load levels [4]. The aim of this test is to determine whether the system can reach its performance targets for availability, concurrency, throughput, and response time. Load testing is the closest to real application use [6]. A typical load test can last from several hours to a few days, during which system behavior data like execution logs and various metrics are collected [7].

Stress testing investigates the behavior of the system under conditions that overload its resources. The stress testing verifies the system behavior against heavy workloads [1] [5], which are executed to evaluate a system beyond its limits, validate system response in activity peaks, and verify whether the system is able to recover from these conditions. It differs from other kinds of testing in that the system is executed on or beyond its breakpoints, forcing the application or the supporting infrastructure to fail [4] [6].

This chapter surveys the state of the art literature in stress testing research. The thesis extends the survey presented by Jiang et al. to the Stress Testing context [8]. This survey will be useful for stress testing practitioners and software engineering researchers with interests in testing and analyzing software systems. We proposed the following four research questions:

- How is a proper stress designed?
- How is a stress test executed and automated?
- What are the main problems found by stress tests?
- How are the stress tests results analysed?

The population in this study is the domain of software testing. Intervention includes application of stress test techniques to test different types of non-functional properties. We used the following search terms:

- Load Testing: load test, Load Testing
- Stress Testing: stress test, stress testing
- Performance Testing: performance tests
- Test tools: jmeter, load runner, performance tester

1.1 Stress Test Process

Contrary to functional testing, which has clear testing objectives, Stress testing objectives are not clear in the early development stages and are often defined later on a case-by-case basis. The Fig. 7 shows a common Load, Performance and Stress test process [8].
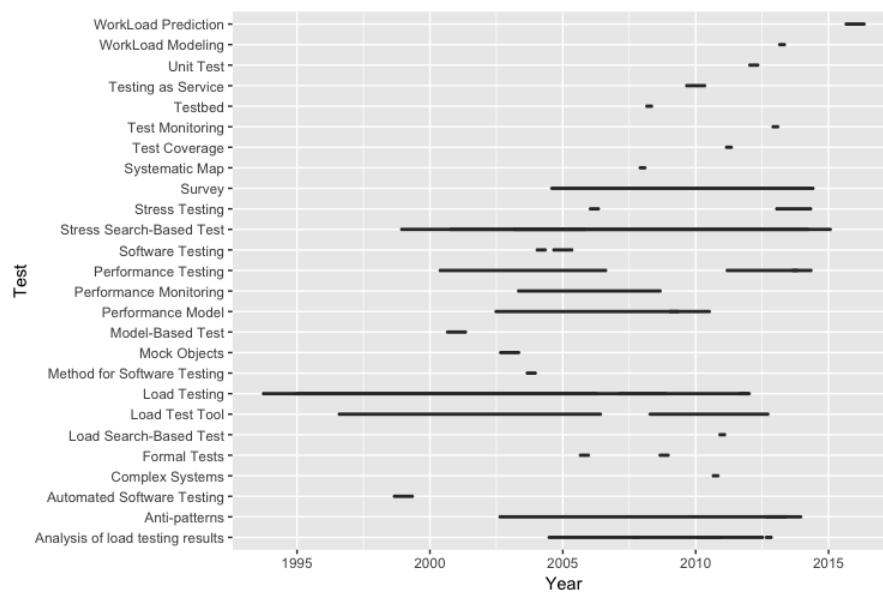
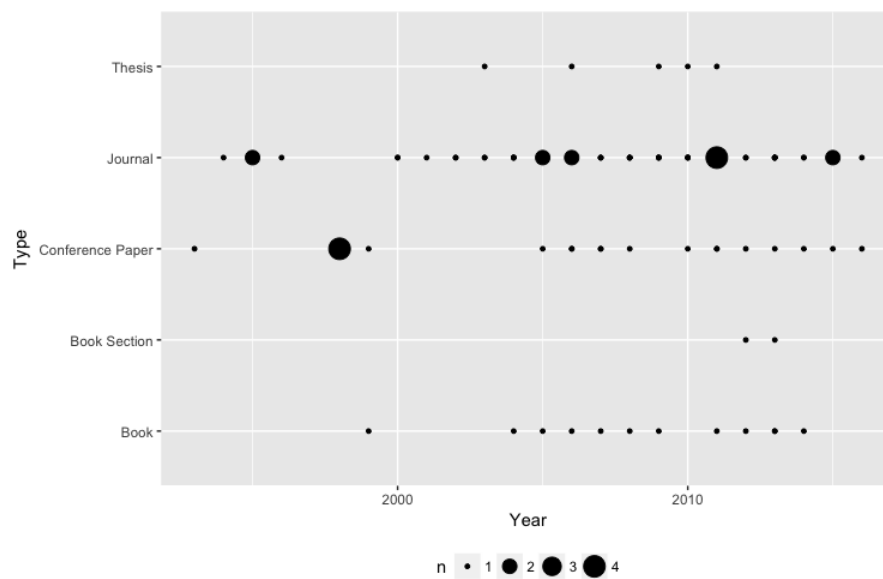Fig. 1: Bubble chart for results of search with the keyword 'Load Testing'

Fig. 2: Bubble chart for results of search with the keyword 'Load Testing'

The goal of the load design phase is to devise a load, which can uncover non-functional problems. Once the load is defined, the system under test executes the

Fig. 3: Bubble chart for results of search with the keyword 'Load Testing'

Fig. 4: Word cloud for results of search with the keyword 'Load Testing'

load and the system behavior under load is recorded. Load testing practitioners then analyze the system behavior to detect problems [8].

Once a proper load is designed, a load test is executed. The load test execution phase consists of the following three main aspects: (1) Setup, which includes system deployment and test execution setup; (2) Load Generation and Termination, which consists of generating the load; and (3) Test Monitoring and Data Collection, which includes recording the system behavior during execution[8].

The core activities in conducting an usual Load, Performance and Stress tests are [9]:
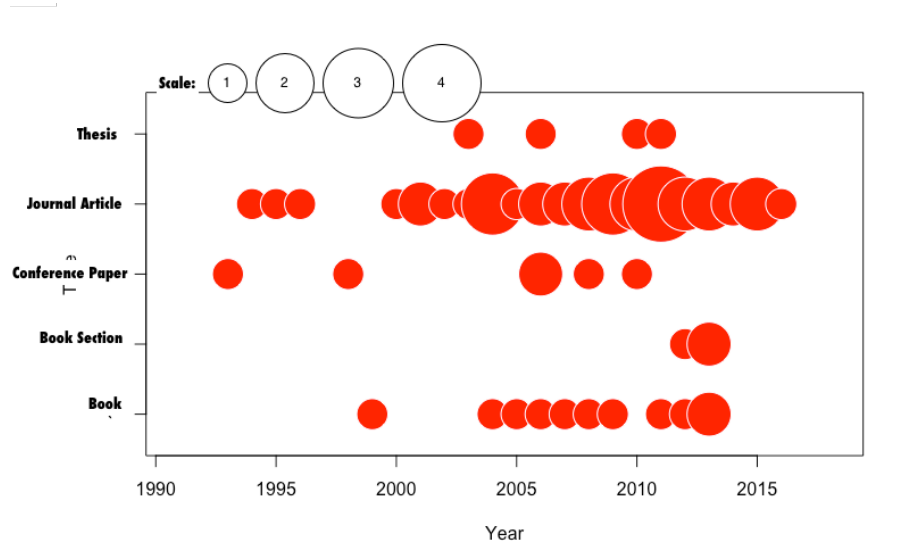
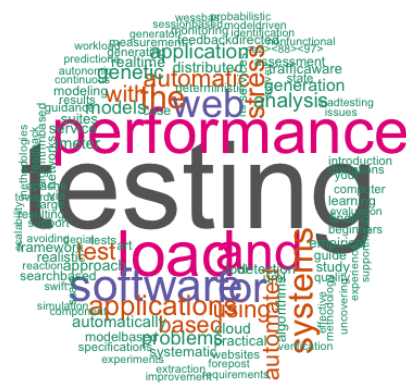Fig. 5: Bubble chart for results of search with the keyword 'Stress Testing'

Fig. 6: Word cloud for results of search with the keyword 'Stress Testing'

– Identify the test environment: identify test and production environments and knowing the hardware, software, and network configurations helps derive an effective test plan and identify testing challenges from the outset.
– Identify acceptance criteria: identify the response time, throughput, and resource utilization goals and constraints.
– Plan and design tests:identify the test scenarios.In the context of testing, a scenario is a sequence of steps in an application. It can represent a use case or a business function such as searching a product catalog, adding an item to a shopping cart, or placing an order [2]. This task includes a description of the speed,
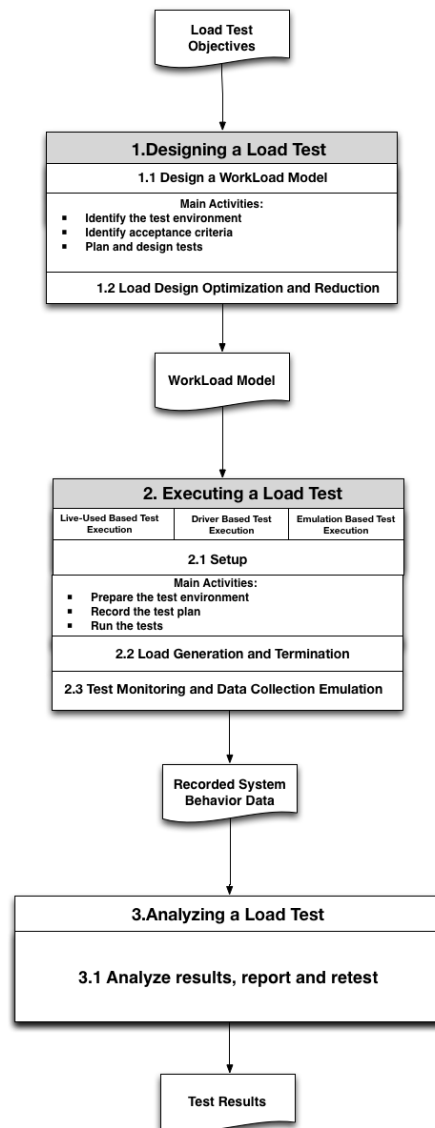
Fig. 7: Load, Performance and Stress Test Process [8][9]

availability, data volume throughput rate, response time, and recovery time of various functions, stress, and so on. This serves as a basis for understanding the level of performance and stress testing that may be required to each test scenario [5].

– Prepare the test environment: configure the test environment, tools, and resources necessary to conduct the planned test scenarios.

– Record the test plan: record the planned test scenarios using a testing tool.

– Run the tests: Once recorded, execute the test plans under light load and verify the correctness of the test scripts and output results.
– Analyze results, report, and retest: examine the results of each successive run and identify areas of bottleneck that need addressing.

2 Research Question 1:How is a proper stress designed?

The design of a stress test depends intrinsically on the load model applied to the software under test. Based on the objectives, there are two general schools of thought for designing a proper load to achieve such objectives [7]:

– Designing Realistic Loads (Workload Descriptive).
– Designing Fault-Inducing Loads (Workload Generative).

In Designing Realistic Loads, the main goal of testing is to ensure that the system can function correctly once. Designing Fault-Inducing Loads aims to design loads, which are likely to cause functional or non-functional problems [7].

Stress testing projects should start with the development of a model for user workload that an application receives. This should take into consideration various performance aspects of the application and the infrastructure that a given workload will impact. A workload is a key component of such a model [6].

The term workload represents the size of the demand that will be imposed on the application under test in an execution. The metric used for measure a workload is dependent on the application domain, such as the length of the video in a transcoding application for multimedia files or the size of the input files in a file compression application [10] [6] [11].

Workload is also defined by the load distribution between the identified transactions at a given time. Workload helps researchers study the system behavior identified in several load models. A workload model can be designed to verify the predictability, repeatability, and scalability of a system [10] [6].

Workload modeling is the attempt to create a simple and generic model that can then be used to generate synthetic workloads. The goal is typically to be able to create workloads that can be used in performance evaluation studies. Sometimes, the synthetic workload is supposed to be similar to those that occur in practice in real systems [10] [6].

There are two kinds of workload models: descriptive and generative. The main difference between the two is that descriptive models just try to mimic the phenomena observed in the workload, whereas generative models try to emulate the process that generated the workload in the first place [4].

In descriptive models, one finds different levels of abstraction on the one hand and different levels of fidelity to the original data on the other hand. The most strictly faithful models try to mimic the data directly using the statistical distribution of the data. The most common strategy used in descriptive modeling is to create a statistical model of an observed workload (Fig. 8). This model is applied to all the workload attributes, e.g., computation, memory usage, I/O behavior, communication, etc. [4]. Fig. 8 shows a simplified workflow of a descriptive model. The workflow has six

phases. In the first phase, the user uses the system in the production environment. In the second phase, the tester collects the user's data, such as logs, clicks, and preferences, from the system. The third phase consists in developing a model designed to emulate the user's behavior. The fourth phase is made up of the execution of the test, emulation of the user's behavior, and log gathering.
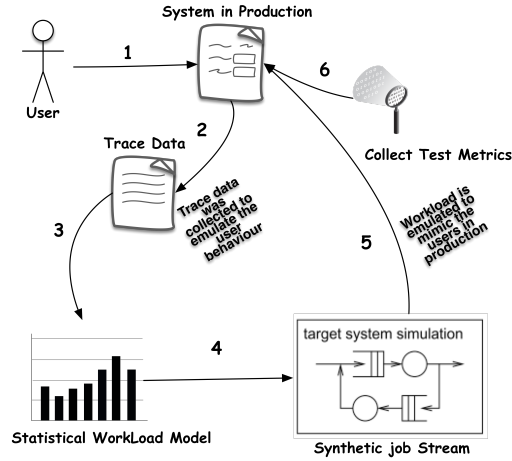

Fig. 8: Workload modeling based on statistical data [4]

Generative models are indirect in the sense that they do not model the statistical distributions. Instead, they describe how users will behave when they generate the workload. An important benefit of the generative approach is that it facilitates manipulations of the workload. It is often desirable to be able to change the workload conditions as part of the evaluation. Descriptive models do not offer any option regarding how to do so. With the generative models, however, we can modify the workload-generation process to fit the desired conditions [4]. The difference between the workflows of the descriptive and the generative models is that user behavior is not collected from logs, but simulated from a model that can receive feedback from the test execution (Fig. 9).

Both load model have their advantages and disadvantages. In general, loads resulting from realistic-load based design techniques (Descriptive models) can be used to detect both functional and non-functional problems. However, the test durations are usually longer and the test analysis is more difficult. Loads resulting from fault-inducing load design techniques (Generative models) take less time to uncover potential functional and non-functional problems, the resulting loads usually only cover a small portion of the testing objectives [8]. The presented research work uses a generative model.

There are several approaches to design generative or descriptive workloads:
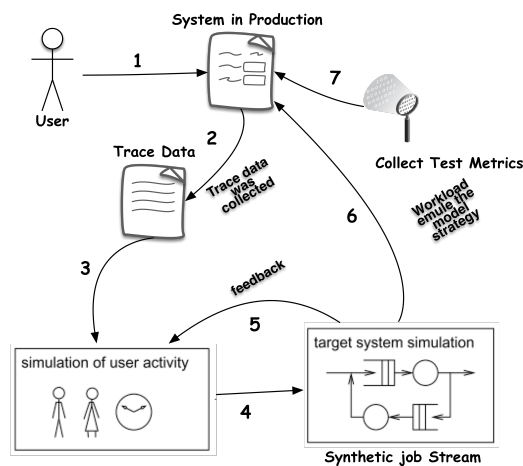
Fig. 9: Workload modeling based on the generative model [4]

– Model-based Stress testing: a usage model is proposed to simulate users' behaviors.
– Feedback-Directed Learning Software Testing: is an adaptive, feedback-directed learning testing system that learns rules from system execution [12] [13].
– Search-based Stress testing.

2.1 Model-based Stress Testing

Model-based testing is an application of models to represent the desired behavior of a System Under Test or to represent testing strategies in a test. Some research aproaches proposes models to simulate or generate realistic loads. Model-based testing (MBT) is a variant of testing that relies on explicit behaviour models that encode the intended behaviours of a system under test. Test cases are generated from one of these models or their combination [14] [15].

The model paradgim is what paradigm and notation are used to describe the model. There are many different modelling notations that have been used for modelling the behaviour of systems for test generation purposes [14] [16].

– State-Based (or Pre/Post) Notations. These model a system as a collection of variables, which represent a snapshot of the internal state of the system, plus some operations that modify those variables. Each operation is usually defined by a precondition and a postcondition, or the postcondition may be written as explicit code that updates the state [14].
– Transition-based Notations. These focus on describing the transitions between different states of the system. Typically, they are graphical node-and-arc notations, like finite state machines (FSMs). Examples of transition-based notations

used for MBT include FSMs themselves, statecharts, labelled transition systems and I/O automata [14].

- History-based Notations. These notations model a system by describing the allowable traces of its behaviour over time. Message-sequence charts and related formalisms are also included in this group. These are graphical and textual notations for specifying sequences of interactions between components [14].
- Functional Notations. These describe a system as a collection of mathematical functions. The functions may be first-order only, as in the case of algebraic specifications, or higher-order, as in notations like HOL [14].
- Operational Notations. These describe a system as a collection of executable processes, executing in parallel. They are particularly suited to describing distributed systems and communications protocols. Examples include process algebras such as CSP or CCS as well as Petri net notations. Slightly stretching this category, hardware description languages like VHDL or Verilog are also included in this category [14].
- Stochastic Notations. These describe a system by a probabilistic model of the events and input values and tend to be used to model environments rather than SUTs. For example, Markov chains are used to model expected usage profiles, so that the generated tests scenarios [14].
- Data-Flow Notations. These notations concentrate on the data rather than the control flow. Prominent examples are Lustre, and the block diagrams of Matlab Simulink, which are often used to model continuous systems [14].

A User Community Modeling Language (UCML) is a set of symbols that can be used to create visual system usage models and depict associated parameters [17]. The Fig. 10 shows a sample where all users realize a login into the application unde test. Once logged in, 40% of the users navigates on the aplication, 30% of the users realizes downloads. 20% of users realizes uploads and 10% of users performs deletions.
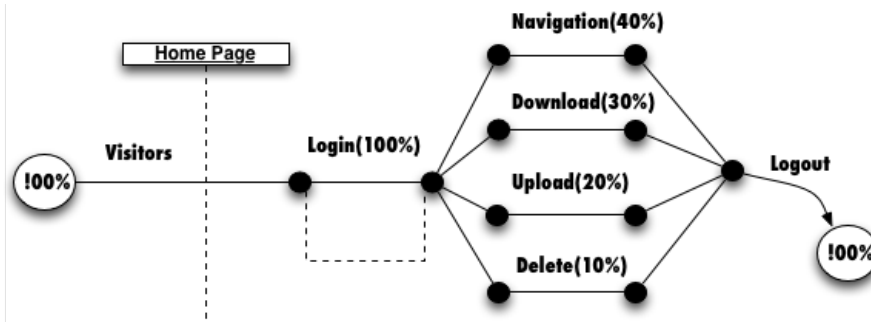

Fig. 10: User community modeling language [17]

Another technique to create workload models it is Stochastic Formcharts. The work of Draheim and Weber's Formoriented analysis is a methodology for the specification of ultra-thin client based systems. Form-oriented models describe a web application as a bipartite state machine which consists of pages, actions, and transitions

between them. Stochastic Formcharts are the combination of formoriented model and probability features. The Fig. 11 shows a sample where all users have a probability of 100% of realize a login into the application under test. Once logged in, users have a probability of 40% of navigate on the application and so on [18].
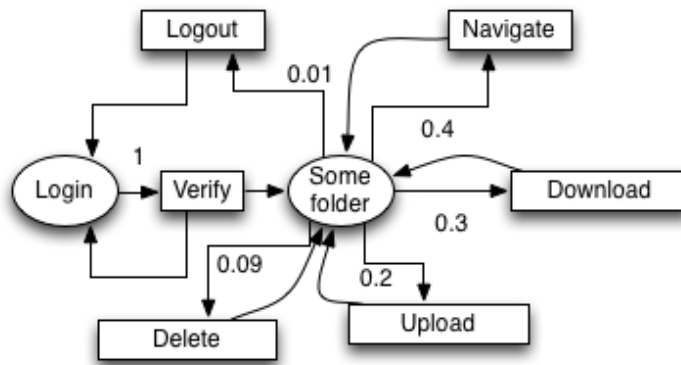

Fig. 11: Stochastic Formcharts Example [18] [17]

One way to capture the navigational pattern within a session is through the Customer Behavior Model Graph (CBMG). Figure 12 depicts an example of a CBMG showing that customers may be in several different states—Home, Browse, Search, Select, Add, and Pay—and they may transition between these states as indicated by the arcs connecting them. The numbers on the arcs represent transition probabilities. A state not explicitly represented in the figure is the Exit state [19] [8] [20].
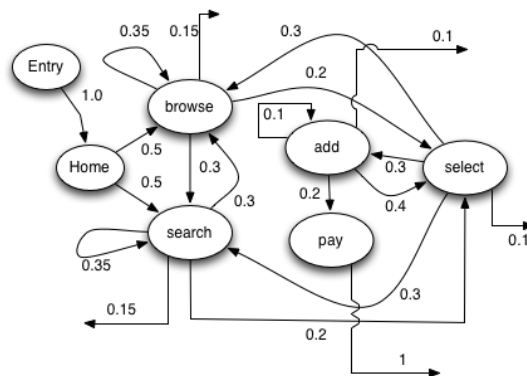

Fig. 12: Example of a Customer Behavior Model Graph (CBMG) [19] [8] [20]

Garousi et al. proposes derivate Stress Test Requirements from an UML model. The input model consists of a number of UML diagrams. Some of them are stan-

dard in mainstream development methodologies and others are needed to describe
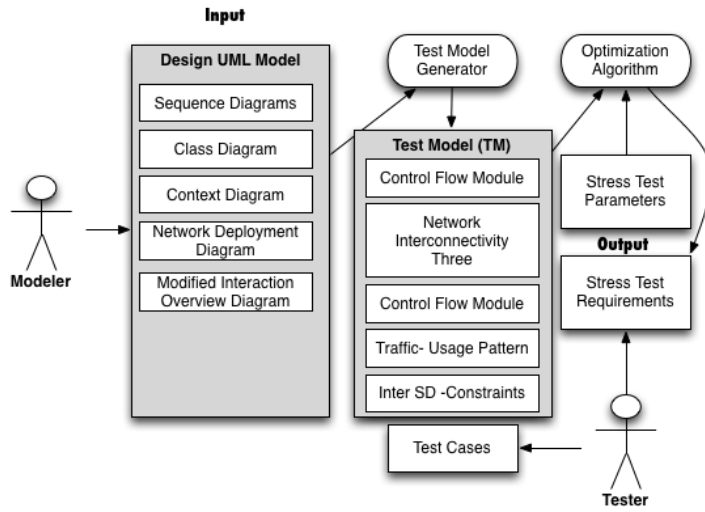the distributed architecture of the system under test (Fig. 13).

Fig. 13: Model-based stress test methodology

Vogele et al. presents an approach that aims to automate the extraction and trans-
formation of workload specifications for an model-based performance prediction of
session-based application systems. The research also presents transformations to the
common load testing tool Apache JMeter and to the Palladio Component Model [21].
The workload specification formalism (Workload Model) consists of the following
components, which are detailed below and illustrated in Fig. 14:

– An Application Model, specifying allowed sequences of service invocations and
  SUT-specific details for generating valid requests.
– A set of Behavior Models,each providing a probabilistic representation of user
  sessions in terms of invoked services.
– A Behavior Mix, specified as probabilities for the individual Behavior Models to
  occur during workload gen- eration.
– A Workload Intensity that includes a function which specifies the number of con-
  current users during the workload generation execution.

2.2 Feedback-Directed Learning Software Testing

Feedback-ORiEnted PerfOrmance Software Testing (FOREPOST) is an adaptive,
feedback-directed learning testing system that learns rules from system execution
traces and uses these learned rules to select test input data automatically to find more
performance problems in applications when compared to exploratory random perfor-
mance testing [22].
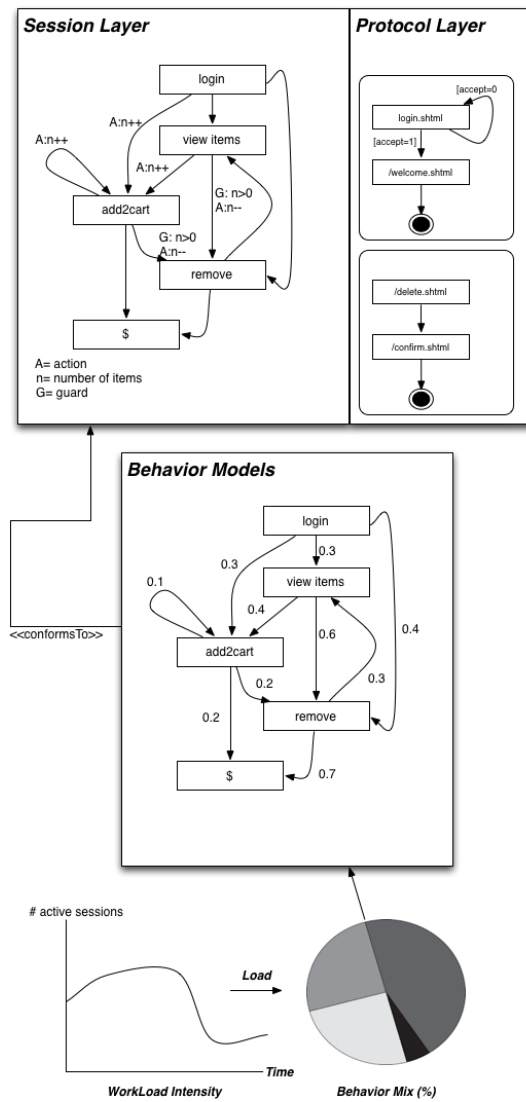
Fig. 14: Exemplary workload model

FOREPOST uses runtime monitoring for a short duration of testing together with machine learning techniques and automated test scripts to reduce large amounts of performance-related information collected during AUT runs to a small number of descriptive rules that provide insights into properties of test input data that lead to increased computational loads of applications.

The Fig. 15 presents the main workflow of FOREPOST solution. The first step, The Test Script is written by the test engineer(1). Once the test script starts, its execution traces are collected (2) by the Profiler, and these traces are forwarded to the

Execution Trace Analyzer, which produces (3) the Trace Statistics. The trace statistics is supplied (4) to Trace Clustering, which uses an ML algorithm, JRip to perform unsupervised clustering of these traces into two groups that correspond to (6) Good and (5) Bad test traces.

The user can review the results of clustering (7). These clustered traces are supplied (8) to the Learner that uses them to learn the classification model and (9) output rules. The user can review (10) these rules and mark some of them as erroneous if the user has sufficient evidence to do so. Then the rules are supplied (11) to the Test Script. Finally, the input space is partitioned into clusters that lead to good and bad test cases, to find methods that are specific to good performance test cases. This task is accomplished in parallel to computing rules, and it starts when the Trace Analyzer produces (12) the method and data statistics that is used to construct (13) two matrices (14). Once these matrices are constructed, ICA decomposes them (15) into the matrices for bad and good test cases correspondingly. Finally, the Advisor (16) determines top methods that performance testers should look at (17) to debug possible performance problems.
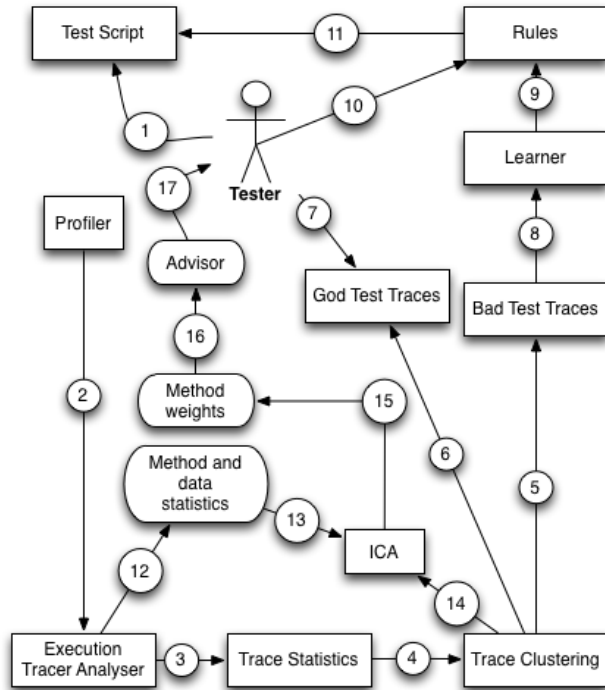

Fig. 15: The architecture and workflow of FOREPOST

2.3 Search-Based Stress Testing

Search-Based Testing is the process of automatically generating test according to a test adequacy criterion,encoded as a fitness function, using search-based optimization algorithms, which are guided by a fitness function. The role of the fitness function is to capture a test objective that, when achieved, makes a contribution to the desired test adequacy criterion [23].

Search–Based Testing uses metaheuristic algorithms to automate the generation of test inputs that meet a test adequacy criterion. Many algorithms have been considered in the past, including Simulated Annealing, Parallel Evolutionary Algorithms [24],Evolution Strategies, Estimation of Distribution Algorithms , Scatter Search , Particle Swarm Optimization , Tabu Search and the Alternating Variable Method. An advantage of meta-heuristic algorithms is that they are widely applicable to problems that are infeasible for analytic approaches. All one has to do is come up with a representation for candidate solutions and an objective function to evaluate those solution [25].

The application of metaheuristic search techniques to test case generation is a possibility which offers much benefits. Metaheuristic search techniques are high-level frameworks which utilise heuristics in order to find solutions to combinatorial problems at a reasonable computational cost. Such a problem may have been classified as NP-complete or NP-hard, or be a problem for which a polynomial time algorithm is known to exist but is not practical [26].

One of the most popular search techniques used in SBST belong to the family of Evolutionary Algorithms in what is known as Evolutionary Testing. Evolutionary Algorithms represent a class of adaptive search techniques based on natural genetics and Darwin's theory of evolution. They are characterized by an iterative procedure that works in parallel on a number of potential solutions to a problem. Figure 16 shows the cycle of an Evolutionary Algorithm when used in the context of Evolutionary Testing [25].

First, a population of possible solutions to a problem is created, usually at random. Starting with randomly generated individuals results in a spread of solutions ranging in fitness because they are scattered around the search–space. Next, each individual in the population is evaluated by calculating its fitness via a fitness function. The principle idea of an Evolutionary Algorithm is that fit individuals survive over time and form even fitter individuals in future generations. Selected individuals are then recombined via a crossover operator. After crossover, the resulting offspring individuals may be subjected to a mutation operator. The algorithm iterates until a global optimum is reached or another stopping condition is fulfilled [25].

The fitness evaluation is the most time consuming task of SBST. However, for time consuming functional testing of complex industrial systems, minimizing the number of generated individuals may also be highly desirable. This might be done using an assumption about the "potential" of individuals in order to predict which individuals are likely to contribute to any future improvement. This prediction could be achieved by using information about similar individuals that have been executed in earlier generations.
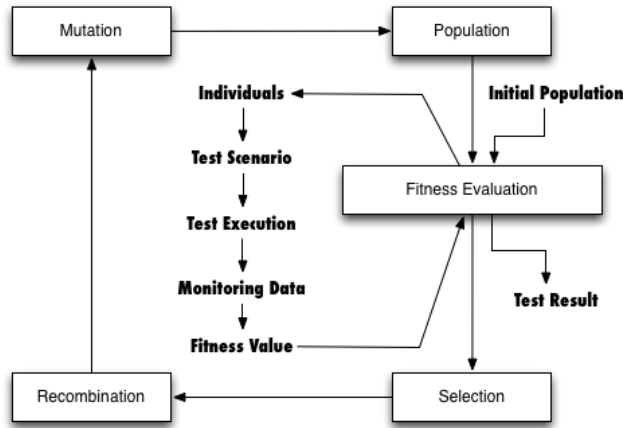
Fig. 16: Evolutionary Algorithm Search Based Test Cycle[25].

2.3.1 Non-functional Search-Based Testing

SBST has made many achievements, and demonstrated its wide applicability and increasing uptake. Nevertheless, there are pressing open problems and challenges that need more attention like to extend SBST to test non-functional properties, a topic that remains relatively under-explored, compared to structural testing. There are many kinds of non-functional search based tests [7]:

– Execution time: The application of evolutionary algorithms to find the best and worst case execution times (BCET, WCET).
– Quality of service: uses metaheuristic search techniques to search violations of service level agreements (SLAs).
– Security: apply a variety of metaheuristic search techniques to detect security vulnerabilities like detecting buffer overflows.
– Usability: concerned with construction of covering array which is a combinatorial object.
– Safety: Safety testing is an important component of the testing strategy of safety critical systems where the systems are required to meet safety constraints.

A variety of metaheuristic search techniques are found to be applicable for non-functional testing including simulated annealing, tabu search, genetic algorithms, ant colony methods, grammatical evolution, genetic programming and swarm intelligence methods. The Fig. 17 shows a comparison between the range of metaheuristcs and the type of non-functional search based test.The Data comes from Afzal et al. [7]. Afzal's work adds to some of the latest research in this area ([27] [28] [29] [30] [31] [32]).

The search for the longest execution time is regarded as a discontinuous, non-linear, optimization problem, with the input domain of the system under test as a search space [33]. The application of SBST algorithms to stress tests involves find-
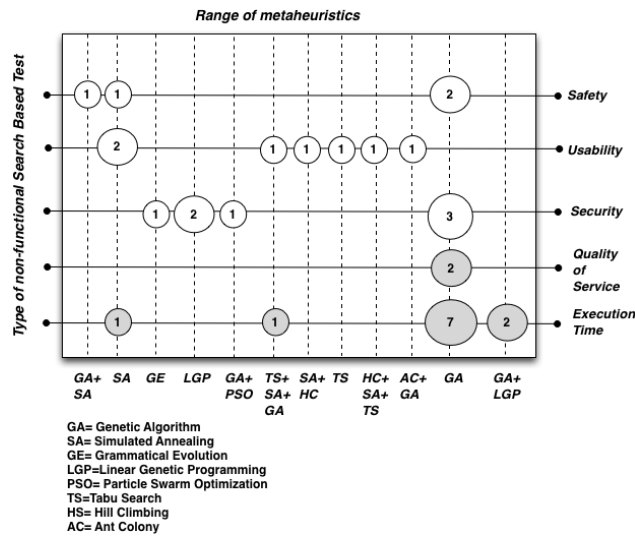
Range of metaheuristics

Fig. 17: Range of metaheuristics by Type of non-functional Search Based Test[7].

ing the best- and worst-case execution times (B/WCET) to determine whether timing constraints are fulfilled [7].

There are two measurement units normally associated with the fitness function in stress test: processor cycles and execution time. The processor cycle approach describes a fitness function in terms of processor cycles. The execution time approach involves executing the application under test and measuring the execution time [7] [34].

Processor cycles measurement is deterministic in the sense that it is independent of system load and results in the same execution times for the same set of input parameters. However, such a measurement is dependent on the compiler and optimizer used, therefore, the processor cycles differ for each platform. Execution time measurement is a non deterministic approach, there is no guarantee to get the same results for the same test inputs [7]. However, stress testing where testers have no access to the production environment should be measured by the execution time measurement [6] [7].

Table 1 shows a comparison between the research studies on load, performance, and stress tests presented by Afzal et al. [7]. Afzal's work adds to some of the latest research in this area ([27] [28] [29] [30] [31] [32]). The columns represent the type of tool used (prototype or functional tool), and the rows represent the metaheuristic approach used by each research study (genetic algorithm, Tabu search, simulated annealing, or a customized algorithm). The table also sorts the research studies by the type of fitness function used (execution time or processor cycles).

The studies can be grouped into two main groups:

- Search-Based Stress Tesing on Safety-critical systems.
- Search-Based Stress Testing on Industrial systems.

Table 1: Distribution of the research studies over the range of applied metaheuristics

	Prototypes		**Functional Tool**
	Execution Time	Processor Cycles	Execution Time
GA + SA + Tabu Search			Gois et al. 2016 [32]
GA	Alander et al.,1998 [35] Wegener et al., 1996 and 1997 [36][37] Sullivan et al., 1998 [33] Briand et al., 2005 [38] Canfora et al., 2005 [39]	Wegener and Grochtmann, 1998 [40] Mueller et al., 1998 [41] Puschner et al. [42] Wegener et al., 2000 [43] Gro et al., 2000 [44]	Di Penta, 2007 [45] Garoussi, 2006 [27] Garousi, 2008 [46] Garousi, 2010 [28]
Simulated Annealing (SA) Constraint Programming			Tracey, 1998 [47] Alesio, 2014 [30] Alesio, 2013 [29]
GA + Constraint Programming			Alesio, 2015 [31]
Customized Algorithm		Pohlheim, 1999 [48]	

2.3.2 Search-Based Stress Tesing on Safety-critical systems

Domains such as avionics, automotive and aerospace feature safety-critical systems, whose failure could result in catastrophic consequences. The importance of software in such systems is permanently increasing due to the need of a higher system flexibility. For this reason, software components of these systems are usually subject to safety certification. In this context, software safety certification has to take into account performance requirements specifying constraints on how the system should react to its environment, and how it should execute on its hardware platform [29].

Usually, embedded computer systems have to fulfil real-time requirements. A faultless function of the systems does not depend only on their logical correctness but also on their temporal correctness. Dynamic aspects like the duration of computations, the memory actually needed during program execution, orthe synchronisation of parallel processes are of major importance for the correct function of real-time systems [37] .

The concurrent nature of embedded software makes the order of external events triggering the systems tasks is often unpredictable. Such increasing software complexity renders performance analysis and testing increasingly challenging. This aspect is reflected by the fact that most existing testing approaches target system functionality rather than performance [29].

Reactive real-time systems must react to external events within time constraints. Triggered tasks must execute within deadlines. Shousha develops a methodology for the derivation of test cases that aims at maximizing the chance of critical deadline misses [49].

The main goal of Search-Based Stress testing of Safety-critical systems it is finding a combination of inputs that causes the system to delay task completion to the greast extent possible [49]. The followed approaches uses metaheuristics to discover the worst-case execution times.

Wegener et al. [36] used genetic algorithms(GA) to search for input situations that produce very long or very short execution times. The fitness function used was the execution time of an individual measured in micro seconds [36]. Alander et al. [35] performed experiments in a simulator environment to measure response time extremes of protection relay software using genetic algorithms. The fitness function used was the response time of the tested software. The results showed that GA generated more input cases with longer response times [35].

Wegener and Grochtmann performed a experimentation to compare GA with random testing. The fitness function used was duration of execution measured in processor cycles. The results showed that, with a large number of input parameters, GA obtained more extreme execution times with less or equal testing effort than random testing [37] [40].

Gro et. al. [44] presented a prediction model which can be used to predict evolutionary testability. The research confirmed that there is a relationship between the complexity of a test object and the ability of a search algorithm to produce input parameters according to B/WCET [44].

Briand et al. [38] used GA to find the sequence of arrival times of events for aperiodic tasks, which will cause the greatest delays in the execution of the target task. A prototype tool named real-time test tool (RTTT) was developed to facilitate the execution of runs of genetic algorithm. Two case studies were conducted and results illustrated that RTTT was a useful tool to stress a system under test [38].

Pohlheim and Wegener used an extension of genetic algorithms with multiple sub-populations, each using a different search strategy. The duration of execution measured in processor cycles was taken as the fitness function. The GA found longer execution times for all the given modules in comparison with systematic testing[48].

Garousi presented a stress test methodology aimed at increasing chances of discovering faults related to distributed traffic in distributed systems. The technique uses as input a specified UML 2.0 model of a system, augmented with timing information.The results indicate that the technique is significantly more effective at detecting distributed traffic-related faults when compared to standard test cases based on an operational profile [27].

Alesio, Nejati and Briand describe a approach based on Constraint Programming (CP) to automate the generation of test cases that reveal, or are likely to, task deadline misses. They evaluate it through a comparison with a state-of-the-art approach based on Genetic Algorithms (GA). In particular, wthe study compares CP and GA in five case studies for efficiency, effectiveness, and scalability. The experimental results show that, on the largest and more complex case studies, CP performs significantly better than GA. The research proposes a tool-supported, efficient and effective approach based on CP to generate stress test cases that maximize the likelihood of task deadline misses [29].

Alesio describe stress test case generation as a search problem over the space of task arrival times. The research search for worst case scenarios maximizing deadline

misses where each scenario characterizes a test case. The paper combine two strategies, GA and Constraint Programming (CP). The results show that, in comparison with GA and CP in isolation, GA+CP achieves nearly the same effectiveness as CP and the same efficiency and solution diversity as GA, thus combining the advantages of the two strategies. Alesio concludes that a combined GA+CP approach to stress testing is more likely to scale to large and complex systems [31].

2.3.3 Search-Based Stress Testing on Industrial systems

Usually, the application of Search-Based Stress Testing on non safety-critical systems deals with the generation of test cases that causes Service Level Agreements violations.

Tracey et al. [47] used simulated annealing (SA) to test four simple programs. The results of the research presented that the use of SA was more effective with larger parameter space. The authors highlighted the need of a detailed comparison of various optimization techniques to explore WCET and BCET of the of the system under test [47].

Di Penta et al. [45] used GA to create test data that violated QoS constraints causing SLA violations. The generated test data included combinations of inputs. The approach was applied to two case studies. The first case study was an audio processing workflow. The second case study, a service producing charts, applied the black-box approach with fitness calculated only on the basis of how close solutions violate QoS constraint. The genome representation is presented in Fig 18. The representation models a wsdl request to a webservice.
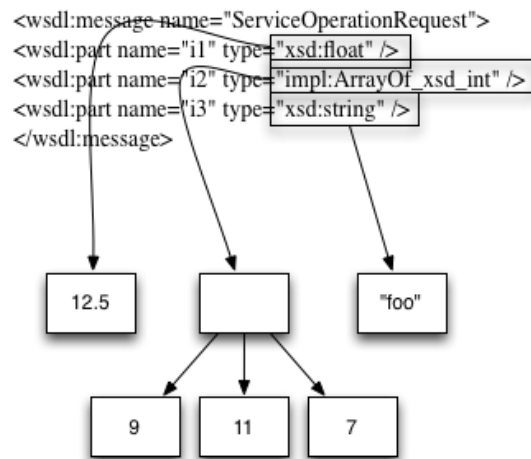

Fig. 18: Genome representation [45].

In case of audio workflow, the GA outperformed random search. For the second case study, use of black-box approach successfully violated the response time constraint, showing the violation of QoS constraints for a real service available on the Internet [45].

Gois et al. proposes an hybrid metaheuristic approach using genetic algorithms, simulated annealing, and tabu search algorithms to perform stress testing. A tool named IAdapter, a JMeter plugin used for performing search-based stress tests, was developed. Two experiments were performed to validate the solution. In the first experiment, the signed-rank Wilcoxon non- parametrical procedure was used for comparing the results. The significant level adopted was 0.05. The procedure showed that there was a significant improvement in the results with the Hybrid Metaheuristic approach. In the second experiment, the whole process of stress and performance tests, which took 3 days and about 1800 executions, was carried out without the need for monitoring by a test designer. The tool automatically selected the next scenarios to be run up to the limit of six generations previously established [32].

3 Research Question 2: How is a stress test executed and automated?

The stress test execution consists of deploy the system and steup test execution ; generating the workloads according to the configurations and terminating the load when the load test is completed and recording the system behavior. There are three general approaches of load test executions [6][8]:

- Live-User Based Executions: The test examines a system's behavior when the system is simultaneously used by many users or execute a load test by employing a group of human testers.
- Driver Based Executions: The driver based execution approach automatically generate thousands or millions of concurrent requests for a long period of time using a software tool.
- Emulation Based Executions: The emulation based load test execution approach performs the load testing on special platforms and doesn't require a fully functional system and conduct load testing.

Usually, a stress test execution it is performed with Driver Based Executions approach [9] [20] [17]. There are three categories of load drivers [8]:

- Benchmark Suite: specialized load driver, designed for one type of system. For example, LoadGen is a load driver specified used to load test the Microsoft Exchange MailServer.
- Centralized Load Drivers: refer to a single load driver, which generates the load.
- Peer-to-peer Load Drivers: refer to a set of load drivers, which collectively generate the target testing load. Peer-to-peer load drivers usually have a con- troller component, which coordinates the load generation among the peer load drivers.

3.1 Load Test Tools

A stress test need to perform hundreds or thousands of concurrent requests to the application under test. Automated tools are needed to carry out serious load, stress, and performance testing. Sometimes, there is simply no practical way to provide reliable, repeatable performance tests without using some form of automation. The aim of any automated test tool is to simplify the testing process.

Workload generators are software products based on workload models to generate request sequences similar to real requests. They are designed and implemented as versatile software tools for performing tuning or capacity planning studies. Workload generators typically have the following components [6]:

– Scripting module: Enable recording of end-user activities in different middleware protocols;
– Test management module: Allows the creation of test scenarios;
– Load injectors: Generate the load with multiple workstations or servers;
– Analysis module: Provides the ability to analyse the data collected by each test interation.

There are several tool to execution of Stress testing. In this tools, the procedure is semi-automated, whereas the execution of the tests itself is performed by a tool, the choice of scenarios to be executed as well as the decision to start new execution batteries are activities of the test designer or tester.

Normally, load test tools uses test scripts. Test scripts are written in a GUI testing framework or a backend server-directed performance tool such as JMeter. These frameworks are the basis on which performance testing is mostly done in industry. Performance test scripts imitate large numbers of users to create a significant load on the application under tests [22].

Performance testing tools, such as Rational's PerformanceStudio, allow for load testing, where the tool can be programmed to run a number of client machines simultaneously to load the client/server system and measure response time. Load testing typically involves various scenarios to analyze how the client/server system responds under various loads [50].

Comparing Web workload generators is a laborious and difficult task since they offer a large amount and diversity of features. In this section we contrast generators according to a wide set of features and capabilities, focusing on their ability to realize search-based tests or have learning capacities.

WebStone was designed by Silicon Graphics in 1996 to measure the performance of Web server software and hardware products. Nowadays, both executable and source actualized code for WebStone are available for free. The benchmark generates a Web server load by simulating multiple Web clients navigating a website. All the testing done by the benchmark is controlled by a Webmaster, which is a program that can be run on one of the client computers or on a different one [20] [51] .

TPC Benchmarkt (TPC-W) is a transactional Web benchmark defined by the Transaction Processing Performance Council that models a representative e-commerce evaluating the architecture performance on a generic profile.The models uses a remote browser emulator to generate requests to server under test. TPC-W adopts the

CBMG model to define the workloads in spite of this model only characterizing user dynamic behavior partially. The remote browser emulators are located in the client side and generate workload towards the e-commerce Web application, which is located in the server side (e-commerce server) [20] [19].
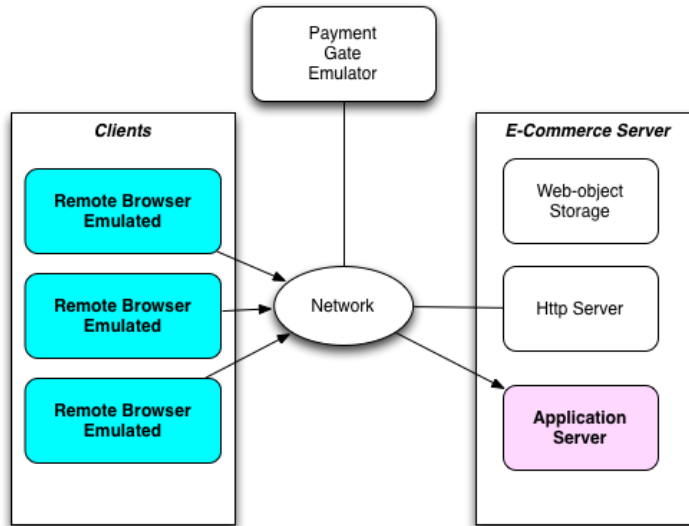

Fig. 19: TPC-W architecture [20] [19]

Open STA is an open source software developed in C++, and released under the GPL licence. OpenSTA provides a script language which permits to simulate the activity of a user. This language can describe HTTP/S scenario and all the test executions is managed in a graphical interface. The composition of the test is very simple, allowing the tester choose scripts for a test and a remote computer that will execute each test.

LoadRunner is one of the most popular industry-standard software products for functional and performance testing. It was originally developed by Mercury Interactive, but nowadays it is commercialized by Hewlett-Packard. LoadRunner supports the definition of user navigations, which are represented using a scripting language. The basic steps are recorded, creating a shell script. Next, this script is then taken off-line, and undergoes further manual steps such as data parameterization and correlations. Finally, the desired performance scripts are obtained after adding transactions and any other required logic (Fig. 20). LoadRunner scripting only permits partial reproduction of user dynamism when generating Web workload, because it cannot define either advanced interactions of users, such as parallel browsing behavior, or continuous changes in user's behaviors [20].

WebLOAD is a software tool for Web performance commercialized by RadView. It is oriented to explore the performance of critical Web applications by quantifying the utilization of the main server resources. The tool creates scenarios that try to
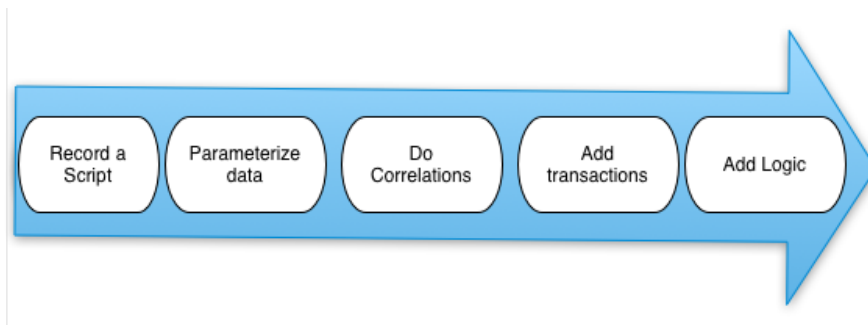
Fig. 20: Load Runner Scripting

mimic the navigations of real users. To this end, it provides facilities to record, edit and debug test scripts, which are used to define the scenarios on workload characterization. The execution environment is a console to manage test execution, whose results are analyzed in the Analytics application. Since WebLOAD is a distributed system, it is possible to deploy several load generators to reproduce the desired load. Load generators can also be used as probing clients where a single virtual user is simulated to evaluate specific statistics of a single user. These probing clients resemble the experience of a real user using the system while it is under load [20].

Apache JMeter is a free open source stress testing tool. It has a large user base and offers lots of plugins to aid testing. JMeter is a desktop application designed to test and measure the performance and functional behavior of applications. The application it's purely Java-based and is highly extensible through a provided API (Application Programming Interface). JMeter works by acting as the client of a client/server application. JMeter allows multiple concurrent users to be simulated on the application [52] [9].

JMeter has components organized in a hierarchical manner. The Test Plan is the main component in a JMeter script. A typical test plan will consist of one or more Thread Groups, logic controllers, listeners, timers, assertions, and configuration elements:

- Thread Group: Test management module responsible to simulate the users used in a test. All elements of a test plan must be under a thread group.
- Listeners: Analysis module responsible to provide access to the information gathered by JMeter about the test cases .
- Samplers: Load injectors module responsible to send requests to a server, while Logical Controllers let you customize its logic.
- Timers: allow JMeter to delay between each request.
- Assertions: test if the application under test it is returning the correct results.
- Configuration Elements: configure detais about the request protocol and test elements.

Table **??** summarizes the studied workloads generators as well as the grade (full or partial) in which they fulfill the features described bellow. None of the presented

tools uses heuristic or learning resources when choosing the scenarios to be tested or the workloads to be applied in the test.

4 Research Question 3: What are the main problems found by stress tests?

Performance problems share common symptoms and many performance problems described in the literature are defined by a particular set of root causes. Fig. 21 shows the symptoms of known performance problems [53].
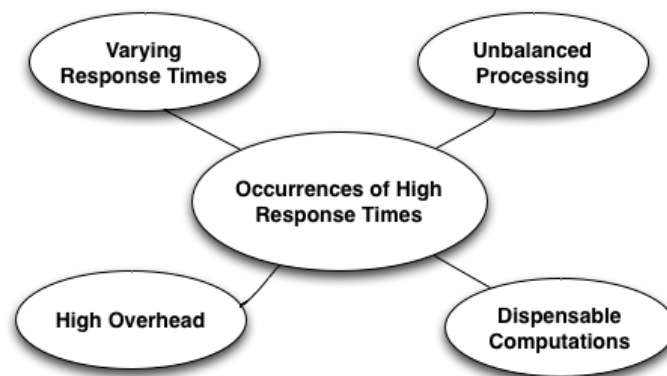
Fig. 21: Symptoms of know performance problems [53].

There are several antipatterns that details features about common performance problems. Antipatterns are conceptually similar to patterns in that they document recurring solutions to common design problems. They are known as antipatterns because their use produces negative consequences. Performance antipatterns document common performance mistakes made in software architectures or designs. These software Performance antipatterns have four primary uses: identifying problems, focusing on the right level of abstraction, effectively communicating their causes to others, and prescribing solutions [54]. The table 2 present some of the most common performance antipatterns.

Table 2: Performance antipatterns

antipattern	Derivations
Blob or The God Class	
Unbalanced-Processing	Concurrent processing Systems
	Piper and Filter Architectures
	Extensive Processing
Circuitous Treasure Hunt	
Empty Semi Trucks	
Tower of Babel	
One-Lane Bridge	
Excessive Dynamic Allocation	
Traffic Jam	
The Ramp	
More is Less	

Blob antipattern is known by various names, including the "god" class [8] and the "blob" [2]. Blob is an antipattern whose problem is on the excessive message traffic generated by a single class or component, a particular resource does the majority of the work in a software. The Blob antipattern occurs when a single class or component either performs all of the work of an application or holds all of the application's data. Either manifestation results in excessive message traffic that can degrade performance [55] [56].

A project containing a "god" class is usually has a single, complex controller class that is surrounded by simple classes that serve only as data containers. These classes typically contain only accessor operations (operations to get() and set() the data) and perform little or no computation of their own [56]. The Figures 22 and 23 describes an hypotetical system with a BLOB problem: The Fig. 22 presents a sample where the Blob class uses the features A,B,C,D,E,F and G of the hypotetical system; The Fig. 23 shows a static view where a complex software entity instance, i.e. Sd, is connected to other software instances, e.g. Sa, Sb and Sc, through many dependencies [57][53].

Unbalanced Processing it's characterises for one scenario where a specific class of requests generates a pattern of execution within the system that tends to overload a particular resource. In other words the overloaded resource will be executing a certain type of job very often, thus in practice damaging other classes of jobs that will experience very long waiting times. Unbalanced Processing occurs in three different situations. The first case that cause unbalanced processing it is when processes cannot make effective use of available processors either because processors are dedicated to other tasks or because of single-threaded code. This manifestation has available processors and we need to ensure that the software is able to use them. Fig. 24 shows a sample of the Unbalanced Processing. In The Fig. 24, four tasks are performed. The task D it is waiting for the task C conclusion that are submmited to a heavy processing situation.

The pipe and filter architectures and extensive processing antipattern represents a manifestation of the unbalanced processing antipattern. The pipe and filter architectures occurs when the throughput of the overall system is determined by the slowest filter. The Fig. 25 describes a software S with a Pipe and Filter Architectures problem:
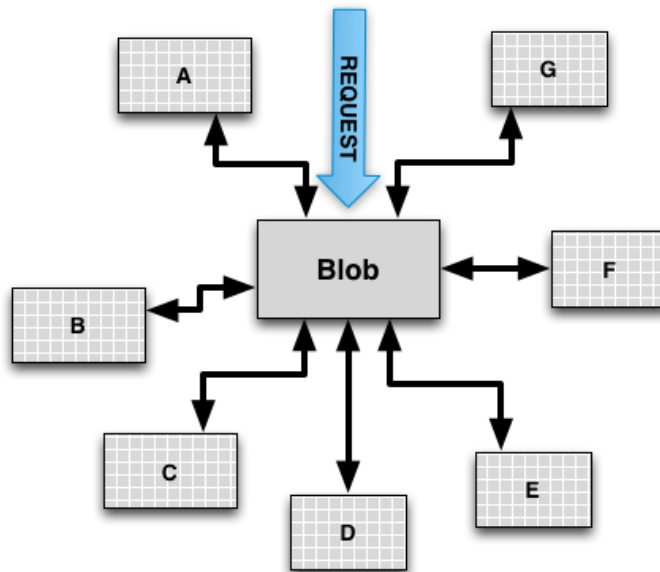
Fig. 22: The God class[53].
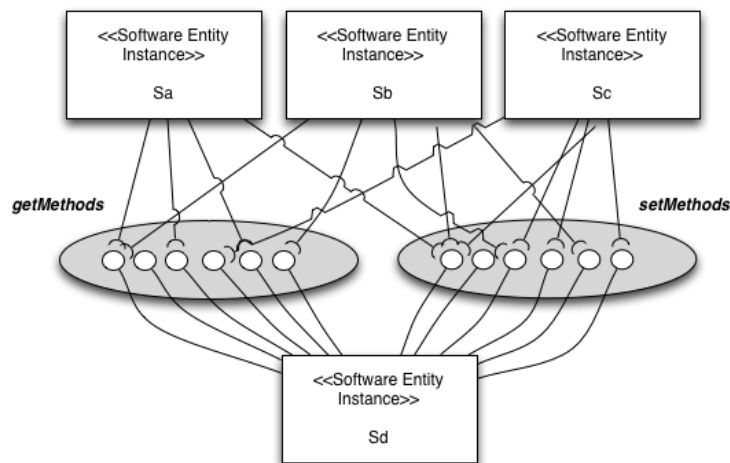
Fig. 23: The God class[57].

(a) Static View, there is a software entity instance, e.g. Sa, offering an operation (operation x) whose resource demand (computation = $compOpx, storage = $storOpx, bandwidth = $bandOpx) is quite high; (b) Dynamic View, the operation opx is invoked in a service and the throughput of the service ($Th(S)) is lower than the required one. The extensive processing occurs when a process monopolizes a processor

and prevents a set of other jobs to be executed until it finishes its computation. The Fig. 26 describes a software S with a Extensive Processing problem: (a) Static View, there is a software entity instance, e.g. Sa, offering two operations (operation x, operation y) whose resource demand is quite unbalanced, since opx has a high demand (computation = $compOpx, storage = $storOpx, bandwidth = $bandOpx), whereas opy has a low demand (computation = $compOpy, storage = $storOpy, bandwidth = $bandOpy); (b) Dynamic View, the operations opx and opy are alternatively invoked in a service and the response time of the service ($RT(S)) is larger than the required one [57].
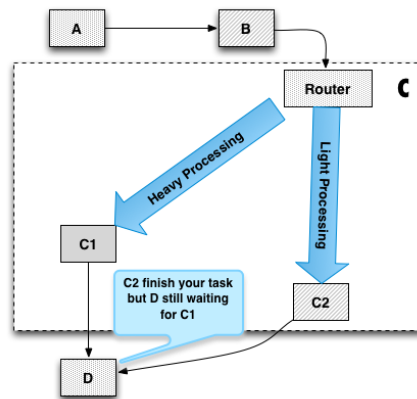
Fig. 24: Unbalanced Processing sample [53].
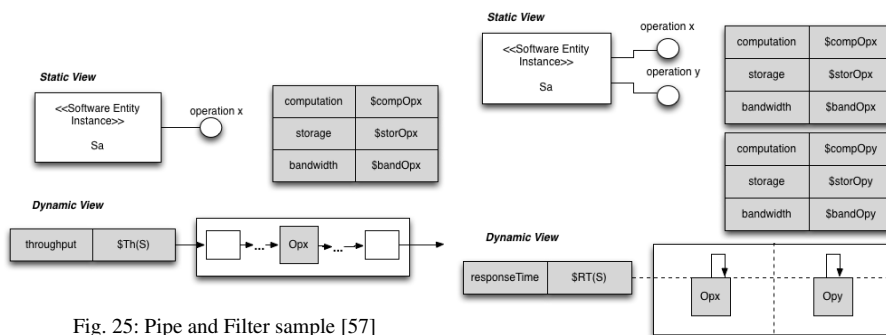
Fig. 25: Pipe and Filter sample [57]

Fig. 26: Extensive Processing sample [57].

Circuitous Treasure Hunt antipattern occurs when software retrieves data from a first componet, uses those results in a second component, retrieves data from the second component, and so on, until the last results are obtained [58] [59]. Circuitous

Treasure Hunt are typical performance antipatterns that causes unnecessarily frequent database requests. The Circuitous Treasure Hunt antipattern is a result from a bad database schema or query design. A common Circuitous Treasure Hunt design creates a data dependency between single queries. For instance, a query requires the result of a previous query as input. The longer the chain of dependencies between individual queries the more the Circuitous Treasure Hunt antipattern hurts performance [13]. The Fig. 27 shows a software S with a Circuitous Treasure Hunt problem: (a) Static View, there is a software entity instance e.g. Sa, retrieving information from the database; (b) Dynamic View, the software S generates a large number of database calls by performing several queries up to the final operation [57].
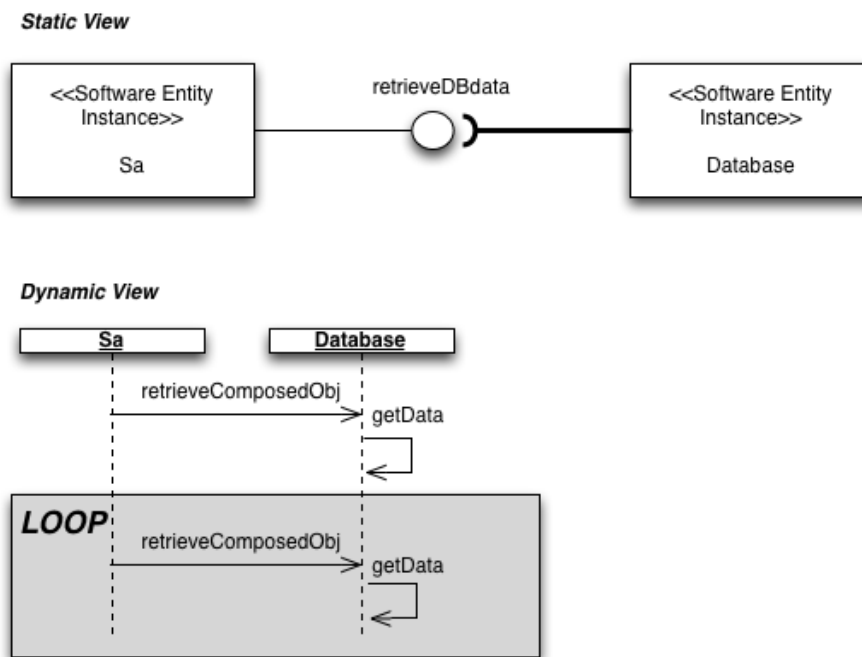
Fig. 27: Circuitous Treasure Hunt sample [57]

Empty Semi Trucks occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both [60]. There are a special case of Empty Semi Trucks that occurs when many fields in a user interface must be retrieved from a remote system. Fig. shows a software S with a Empty Semi Trucks problem: (a) Static View, there is a software entity instance, e.g. Sa, retrieving some information from several instances (Remote Software 1, . . . , Remote Software n); (b) Dynamic View, the software instance Sa generates an excessive message traffic by sending a big amount of messages with low sizes, much lower than the network bandwidth, hence the network link might have a low utilization value [57].
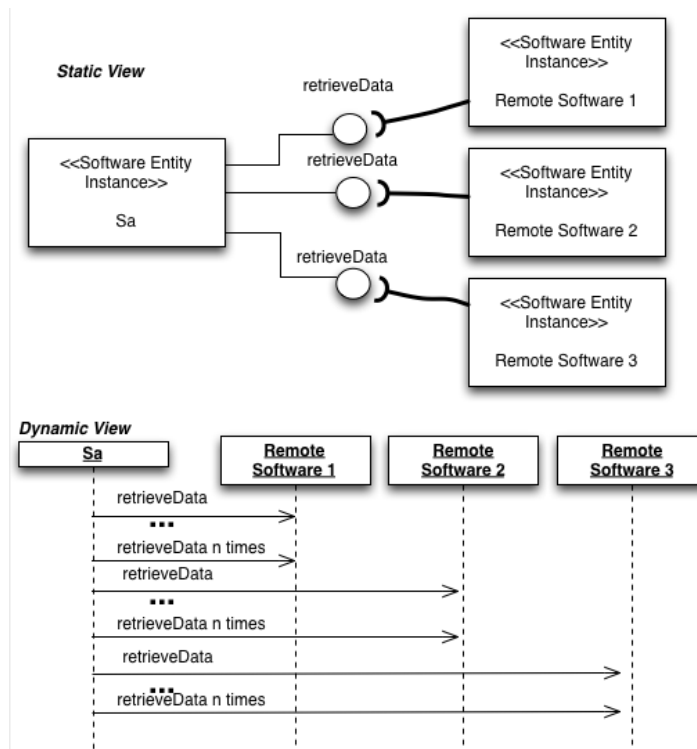
Fig. 28: Empty Semi Trucks sample [57].

The Tower of Babel antipattern most often occurs when information is translated into an exchange format, such as XML, by the sending process then parsed and translated into an internal format by the receiving process. When the translation and parsing is excessive, the system spends most of its time doing this and relatively little doing real work [59]. Fig. shows a system with a Tower of Babel problem: (a) Static View, there are some software entity instances, e.g. Sa, Sb, . . . , Sn; (b) Dynamic View, the software instances Sd performs many times the translation of format for communicating with other instances [57].

One-Lane Bridge is a antipattern that occurs when one or a few processes execute concurrently using a shared resource and other processes are waiting for use the shared resource. It frequently occurs in applications that access a database. Here, a lock ensures that only one process may update the associated portion of the database at a time. This antipatterns is common when many concurrent threads or processes are waiting for the same shared resources. These can either be passive resources (like semaphores or mutexes) or active resources (like CPU or hard disk). In the first case, we have a typical One Lane Bridge whose critical resource needs to be identified. Figure 3.10 shows a system with a One-Lane Bridge problem: (a) Static View, there is a software entity instance with a capacity of managing $poolSize threads; (b) Dy-
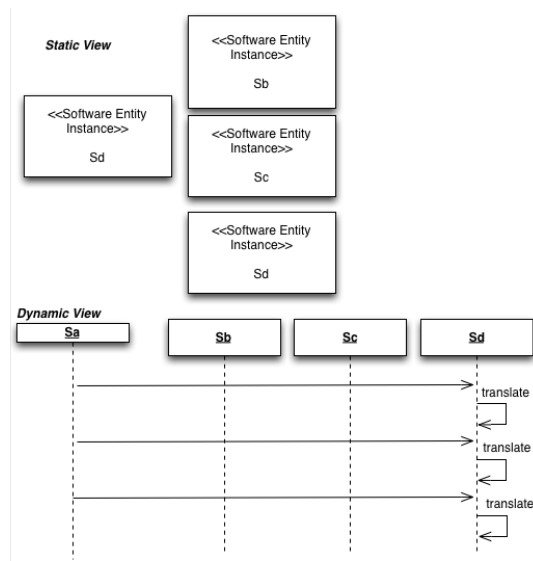
Fig. 29: Tower of Babel sample [57]
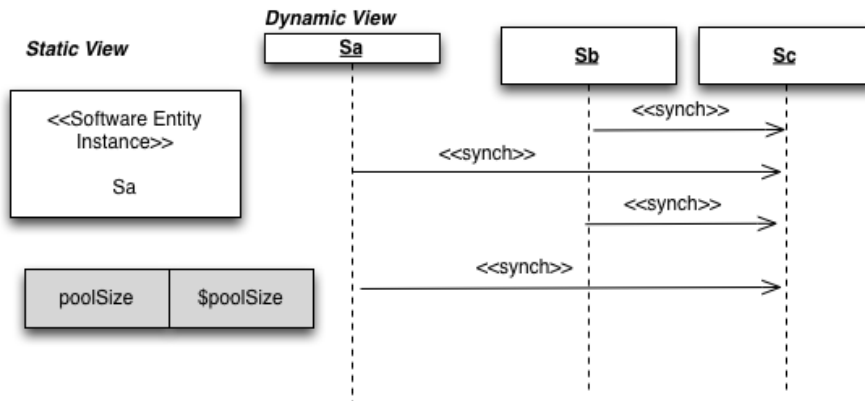
Fig. 30: One-Lane Bridge sample [57].

namic View, the software instance Sc receives an excessive number of synchronous calls in a service S and the predicted response time is higher than the required [57].

Using dynamic allocation, objects are created when they are first accessed and then destroyed when they are no longer needed. Excessive Dynamic Allocation, however, addresses frequent, unnecessary creation and destruction of objects of the same class. Dynamic allocation is expensive , an object created in memory must be allocated from the heap, and any initialization code for the object and the contained objects must be executed. When the object is no longer needed, necessary clean-up must be performed, and the reclaimed memory must be returned to the heap to avoid memory leaks [58] [59].
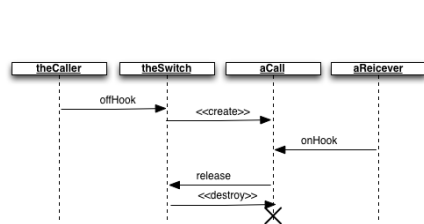
Fig. 31: Excessive Dynamic Allocation.
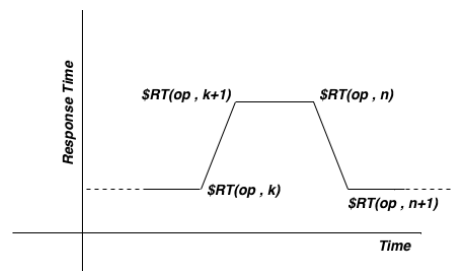
Fig. 32: Traffic Jam Response Time [57].

The Fig. 31 shows a Excessive Dynamic Allocation sample. This example is drawn from a call (an offHook event), the switch creates a Call object to manage the call. When the call is completed, the Call object is destroyed. Constructing a single Call object it is not seem as excessive. A Call is a complex object that contains several other objects that must also be created. The Excessive Dynamic Allocation occurs when a switch receive hundreds of thousands of offHook events. In a case like this, the overhead for dynamically allocating call objects adds substantial delays to the time needed to complete a call.

The Traffic Jam antipattern occurs if many concurrent threads or processes are waiting for the same active resources (like CPU or hard disk). This antipatterns produces a large backlog in jobs waiting for service. The performance impact of the Traffic Jam is the transient behavior that produces wide variability in response time. Sometimes it is fine, but at other times, it is unacceptably long. Figure 32 describes a software with a Traffic Jam problem, the monitored response time of the operation shows a wide variability in response time which persists long [57].
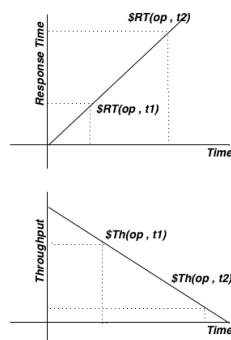
Fig. 33: The Ramp sample [57].
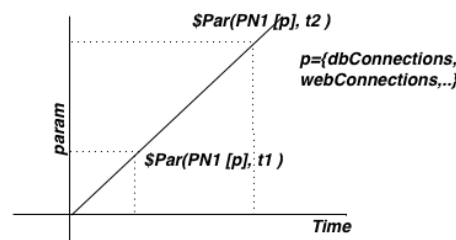
Fig. 34: More is Less sample [57].

The Ramp it is a antipattern where the processing time increases as the system is used. The Ramp can arise in several different ways. Any situation in which the amount of processing required to satisfy a request increases over time will produce the behavior. With the Ramp antipattern, the memory consumption of the application

is growing over time. The root cause is Specific Data Structures which are growing during operation or which are not properly disposed [13] [59]. Fig. 33 shows a system with The Ramp problem: (i) the monitored response time of the operation opx at time t1, i.e. $RT(opx, t1), is much lower than the monitored response time of the operation opx at time t2, i.e. $RT(opx, t2), with t1 < t2; (ii) the monitored throughput of the operation opx at time t1, i.e. $Th(opx, t1), is much larger than the monitored throughput of the operation opx at time t2, i.e. $Th(opx, t2), with t1 < t2.

More is less occurs when a system spends more time "thrashing" than accomplishing real work because there are too many processes relative to available resources. More is Less are presented when it is running too many programs overtime. This antipattern causes too much system paging and systems spend all their time servicing page faults rather than processing requests. In distributed systems, there are more causes.They include: creating too many database connections and allowing too many internet connection. Fig. 34 describes a system with a More Is Less problem: There is a processing node PN1 and the monitored runtime parameters (e.g. database connections, etc.) at time t1, i.e. $Par(PN1[p], t1), are much larger than the same parameters at time t2, i.e. $Par(PN1[p], t2), with t1 < t2.

5 Research Question 4: How are the stress tests results analysed?

The system behavior recorded during the stress test execution phase needs to be analyzed to determine if there are any load-related functional or non-functional problems [8].

There can be many formats of system behavior like resource usage data or end-to-end response time, which is recorded as response time for each individual request. These types of data need to be processed before comparing against threshold values. A proper data summarization technique is needed to describe these many data instances into one number.

There are three types of data summarization techniques proposed in the literature. Jiang et al. use response time analysis as an example to describe the proposed data summarization techniques [8]:

- Maximum values;
- Average or Medium Vales;
- Percentile-values.

Some researchers advocate that the 90-percentile response time is a better measurement than the average/medium response time, as the former accounts for most of the peaks, while eliminating the outliers [8].

6 Conclusion

Table 3: My caption

Test Design	Test Execution	Load Driver	Test Tool License	Search-Based Stress Test Application	Search-Based Test Metaheuris.	Test Results
Model-Based Testing	Live-User Based	Benchmark Suite	Comercial	Real-Time	Genetic Algorithms (GA)	Average
FOREPOST	Driver-Based	Centralized L. Drivers	Open-source	Industrial	Simulated Annealing	90-Percentile
Search-Based Testing	Emulation-Based	Peer-to-peer L. Drivers			Tabu Search	80-Percentile
					Ant Colony	70-Percentile
					Particle Swarm	Maximum
					Hill Climbing	Minimal

Table 4: My caption

Year	Type	Researches	Classification
1999	Book	[50]	Automated Software Testing
2004	Book	[3]	Method for Software Testing
2005	Book	[5]	Software Testing
2006	Book	[61]	Formal Tests
2007	Book	[2]	Load Testing
2008	Book	[52]	Load Test Tool
2009	Book	[6]	Load Testing
2011	Book	[62]	Complex Systems
2012	Book	[63]	Unit Test
2013	Book	[9]	Load Test Tool
2013	Book	[10]	WorkLoad Modeling
2014	Book	[64]	Performance Testing
2012	Book Section	[65]	Load Testing
2013	Book Section	[66]	Analysis of load testing results
1993	Conference Paper	[67]	Load Testing
1998	Conference Paper	[35]	Stress Search-Based Test
1998	Conference Paper	[42]	Stress Search-Based Test
1998	Conference Paper	[41]	Stress Search-Based Test
1998	Conference Paper	[33]	Stress Search-Based Test
1999	Conference Paper	[68]	Stress Search-Based Test
2005	Conference Paper	[39]	Stress Search-Based Test
2006	Conference Paper	[18]	Load Testing
2006	Conference Paper	[69]	Stress Testing

2007	Conference Paper	[45]	Stress Search-Based Test
2007	Conference Paper	[70]	Load Testing
2008	Conference Paper	[71]	Analysis of load testing results
2010	Conference Paper	[72]	Testing as Service
2011	Conference Paper	[73]	Performance Testing
2011	Conference Paper	[?]	Load Search-Based Test
2012	Conference Paper	[60]	Anti-patterns
2013	Conference Paper	[74]	Performance Testing
2014	Conference Paper	[75]	Performance Testing
2015	Conference Paper	[76]	Anti-patterns
2016	Conference Paper	[32]	Stress Search-Based Test
1994	Journal	[77]	Load Testing
1995	Journal	[78]	Load Testing
1995	Journal	[51]	Load Testing
1996	Journal	[79]	Load Test Tool
2000	Journal	[80]	Performance Testing
2000	Journal	[44]	Stress Search-Based Test
2001	Journal	[81]	Model-Based Test
2002	Journal	[19]	Performance Model
2002	Journal	[58]	Anti-patterns
2003	Journal	[82]	Mock Objects
2003	Journal	[83]	Performance Monitoring
2004	Journal	[84]	Analysis of load testing results
2004	Journal	[85]	Performance Testing
2004	Journal	[1]	Software Testing
2004	Journal	[26]	Survey
2005	Journal	[38]	Stress Search-Based Test
2005	Journal	[86]	Load Test Tool
2005	Journal	[48]	Stress Search-Based Test
2006	Journal	[27]	Stress Search-Based Test
2006	Journal	[27][87]	Stress Search-Based Test
2007	Journal	[88]	Performance Testing
2007	Journal	[89]	Load Test Tool
2007	Journal	[90]	Performance Model
2008	Journal	[91]	Stress Search-Based Test
2008	Journal	[92]	Systematic Map
2008	Journal	[15]	Testbed
2008	Journal	[93]	Performance Model
2009	Journal	[94]	Performance Monitoring
2009	Journal	[95]	Analysis of load testing results
2009	Journal	[7]	Survey
2009	Journal	[16]	Formal Tests
2010	Journal	[96]	Load Testing
2010	Journal	[97]	Analysis of load testing results
2010	Journal	[28]	Stress Search-Based Test
2010	Journal	[98]	Survey

2011	Journal	[99]	Analysis of load testing results
2011	Journal	[25]	Stress Search-Based Test
2011	Journal	[100]	Load Testing
2011	Journal	[101]	Load Testing
2011	Journal	[102]	Performance Model
2011	Journal	[103]	Load Testing
2011	Journal	[104]	Load Testing
2011	Journal	[105]	Test Coverage
2012	Journal	[106]	Load Testing
2012	Journal	[22]	Stress Testing
2013	Journal	[107]	Test Monitoring
2013	Journal	[108]	Analysis of load testing results
2013	Journal	[17]	Load Testing
2013	Journal	[53]	Anti-patterns
2013	Journal	[109]	Stress Testing
2013	Journal	[29]	Stress Search-Based Test
2014	Journal	[13]	Anti-patterns
2014	Journal	[30]	Stress Search-Based Test
2015	Journal	[110]	Survey
2015	Journal	[12]	Stress Testing
2015	Journal	[111]	Survey
2015	Journal	[31]	Stress Search-Based Test
2016	Journal	[21]	WorkLoad Prediction
2003	Thesis	[112]	Stress Search-Based Test
2006	Thesis	[27]	Stress Search-Based Test
2010	Thesis	[8]	Analysis of load testing results
2009	Thesis	[113]	Performance Model
2011	Thesis	[57]	Analysis of load testing results

References

1. C. Sandler, T. Badgett, T. Thomas, p. 200 (2004)
2. M. Corporation. Performance Testing Guidance for Web Applications (2007). URL http://www.amazon.com/Performance-Testing-Guidance-Web-Applications/dp/0735625700http://msdn.microsoft.com/en-us/library/bb924375.aspx
3. W.E. Perry, *Effective methods for software testing* (2004). DOI 10.1002/1521-3773(20010316)40:6<9823::AID-ANIE9823>3.3.CO;2-C. URL http://onlinelibrary.wiley.com/doi/10.1002/cbdv.200490137/abstract$\backslash$nhttp://scholar.google.com/scholar?hl=en{\&}btnG=Search{\&}q=intitle:Effective+Methods+for+Software+Testing{\#}2$\backslash$nhttp://scholar.google.com/scholar?hl=en{\&}btnG=Search{\&}q=intitle:Effective+methods+for
4. G.a. Di Lucca, A.R. Fasolino, Information and Software Technology **48**, 1172 (2006). DOI 10.1016/j.infsof.2006.06.006
5. W.E. Lewis, D. Dobbs, G. Veerapillai, *Software testing and continuous quality improvement* (2005). URL http://books.google.com/books?id=fgaBDd0TfT8C{\&}pgis=1
6. I. Molyneaux, *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*, 1st edn. ("O'Reilly Media, Inc.", 2009)
7. W. Afzal, R. Torkar, R. Feldt, Information and Software Technology **51**(6), 957 (2009). DOI 10.1016/j.infsof.2008.12.005

8. Z. Jiang, Automated analysis of load testing results. Ph.D. thesis (2010). URL http://dl.acm.org/citation.cfm?id=1831726

9. B. Erinle, *Performance Testing With JMeter 2.9* (2013)

10. D.G. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation* (Cambridge University Press, 2013)

11. M.C. Gonçalves, (2014)

12. Q. Luo, A. Nair, M. Grechanik, D. Poshyvanyk, Empirical Software Engineering pp. 1–51 (2015). DOI 10.1007/s10664-015-9413-5

13. A. Wert, M. Oehler, C. Heger, R. Farahbod, QoSA 2014 - Proceedings of the 10th International ACM SIGSOFT Conference on Quality of Software Architectures (Part of CompArch 2014) pp. 3–12 (2014). DOI 10.1145/2602576.2602579. URL http://dx.doi.org/10.1145/2602576.2602579

14. A.P. Mark Utting, B. Legeard, Software Testing Verification and Reliability **24**(8), 297 (2012). DOI 10.1002/stvr

15. Testing of Software and . . . pp. 266–282 (2008)

16. R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A.J.H. Simons, S. Vilkomir, M.R. Woodward, H. Zedan, ACM Comput. Surv. **41**(2), 1 (2009). DOI http://doi.acm.org/10.1145/1459352.1459354

17. X. Wang, B. Zhou, W. Li, Journal of the Chinese Institute of Engineers **36**(1), 74 (2013). DOI 10.1080/02533839.2012.726028. URL http://www.tandfonline.com/doi/abs/10.1080/02533839.2012.726028

18. D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, G. Weber, in *Conference on Software Maintenance and Reengineering (CSMR'06)* (2006). DOI 10.1109/CSMR.2006.43

19. D.A. Menascé, G. Mason, (June), 1 (2002)

20. P.N. Mohammad S. Obaidat, F. Zarai, *Modeling and Simulation of Computer Networks and Systems Methodologies and Applications*

21. C. Vogele, A. van Hoorn, E. Schulz, W. Hasselbring, H. Krcmar, Software and Systems Modeling (October), 1 (2016). DOI 10.1007/s10270-016-0566-5. URL "http://dx.doi.org/10.1007/s10270-016-0566-5

22. M. Grechanik, C. Fu, Q. Xie, 2012 34th International Conference on Software Engineering (ICSE) pp. 156–166 (2012). DOI 10.1109/ICSE.2012.6227197

23. M. Harman, P. McMinn, IEEE Transactions on Software Engineering **36**(2), 226 (2010). DOI 10.1109/TSE.2009.71

24. E. Alba, F. Chicano, Computers and Operations Research **35**(10), 3161 (2008). DOI 10.1016/j.cor.2007.01.016

25. A.I. Baars, K. Lakhotia, T.E.J. Vos, J. Wegener, Federated Conference on Computer Science and Information Systems (FedCSIS 2011) pp. 917–923 (2011). URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp={\&}arnumber=6078178

26. P. McMinn, R. Court, S. Testing, P. Street, Software testing, Verification and reliability **14**, 1 (2004). DOI 10.1002/stvr.294

27. V. Garousi, Traffic-aware Stress Testing of Distributed Real-Time Systems based on UML Models using Genetic Algorithms. Ph.D. thesis (2006)

28. V. Garousi, IEEE Transactions on Software Engineering **36**(6), 778 (2010). DOI 10.1109/TSE.2010.5

29. S. Di Alesio, S. Nejati, L. Briand, A. Gotlieb, IEEE Xplore pp. 158–167 (2013). DOI 10.1109/ISSRE.2013.6698915

30. S. Di Alesio, S. Nejati, L. Briand, A. Gotlieb, Principles and Practice of Constraint Programming pp. 813–830. DOI 10.1007/978-3-319-10428-7{_}58

31. S.D.I. Alesio, L.C. Briand, S. Nejati, A. Gotlieb, ACM Transactions on Software Engineering and Methodology **25**(1) (2015)

32. N. Gois, P. Porfirio, A. Coelho, T. Barbosa, in *Proceedings of the 2016 Latin American Computing Conference (CLEI)* (2016), pp. 718–728

33. M.O. Sullivan, S. Vössner, J. Wegener, D.b. Ag, pp. 1–20

34. N.J. Tracey, A search-based automated test-data generation framework for safety-critical software. Ph.D. thesis, Citeseer (2000)

35. J.T.J. Alander, T. Mantere, P. Turunen, in *Neural Nets and Genetic Algorithms* (1998)

36. J. Wegener, H. Sthamer, B.F. Jones, D.E. Eyres, Software Quality Journal **6**(2), 127 (1997). DOI 10.1023/A:1018551716639. URL http://www.springerlink.com/index/uh26067rt3516765.pdf

37. B.J. J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer, EuroSTAR'96: Proceedings of the Fourth International Conference on Software Testing Analysis and Review (1996)

38. L.C. Briand, Y. Labiche, M. Shousha, Proceedings of the 2005 conference on Genetic and evolutionary computation - GECCO '05 p. 1021 (2005). DOI 10.1145/1068009.1068183

39. G. Canfora, M.D. Penta, R. Esposito, M.L. Villani,

40. J. Wegener, M. Grochtmann, Real-Time Systems **15**(3), 275 (1998). DOI Doi10.1023/A: 1008096431840

41. F. Mueller, J. Wegener, Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No.98TB100245) (1998). DOI 10.1109/RTTAS.1998.683198

42. P. Puschner, R. Nossal, Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279) (1998). DOI 10.1109/REAL.1998.739738

43. H. Wegener, Joachim and Pitschinetz, Roman and Sthamer, Proceedings of the 1st International Workshop on Automated Program Analysis, Testing and Verification (WAPATV'00) (2000)

44. H. Gross, B.F. Jones, D.E. Eyres, Software, IEE Proceedings- **147**(2), 25 (2000). DOI 10.1049/ip-sen

45. M.D. Penta, G. Canfora, G. Esposito, in *Proceedings of the 9th annual conference on Genetic and evolutionary computation* (2007), pp. 1090–1097

46. V. Garousi, Proceedings of the 10th annual conference on Genetic and evolutionary computation - GECCO '08 p. 1743 (2008). DOI 10.1145/1389095.1389433

47. N.J. Tracey, J.a. Clark, K.C. Mander, (1998)

48. H. Pohlheim, M. Conrad, A. Griep, Analysis (724), 804 (2005). DOI 10.4271/2005-01-0750

49. M. Shousha, Performance stress testing of real-time systems using genetic algorithms. Ph.D. thesis, Carleton University Ottawa (2003)

50. E. Dustin, J. Rashka, J. Paul, *Automated Software Testing: Introduction, Management, and Performance* (1999)

51. G. Trent, M. Sake, WWW Conference'95 (1995)

52. E.H. Halili, *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites.* (2008). DOI 10.1017/CBO9781107415324.004

53. A. Wert, J. Happe, L. Happe, Proceedings - International Conference on Software Engineering (May), 552 (2013). DOI 10.1109/ICSE.2013.6606601

54. W.H. Brown, R.C. Malveau, H.W. McCormick, T.J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis* (John Wiley & Sons, Inc., 1998)

55. V. Cortellessa, L. Frittella, pp. 171–185 (2007)

56. C.U. Smith, L.G. Williams, Proceedings of the second international workshop on Software and performance - WOSP '00 pp. 127–136 (2000). DOI 10.1145/350391.350420. URL `http://portal.acm.org/citation.cfm?doid=350391.350420`

57. C. Trubiani, Language (2011)

58. C. Smith, L. Williams, Cmg-Conference- **2**, 797 (2002). URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.6968{\&}rep=rep1{\&}type=pdf`

59. C.U. Smith, L.G. Williams, Computer Measurement Group Conference pp. 717–725 (2003). URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.4517{\&}rep=rep1{\&}type=pdf`

60. D. Arcelli, V. Cortellessa, C. Trubiani, Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures (QoSA '12) pp. 33–42 (2012). DOI 10.1145/2304696.2304704. URL `http://doi.acm.org/10.1145/2304696.2304704`

61. K. Havelund, M. Núñez, G. Roşu, B. Wolff, *Formal Approaches to Software Testing and Runtime Verification* (2006). URL `http://www.ulb.tu-darmstadt.de/tocs/79304567.pdf`

62. A.J. Oliner, *Using Influence to Understand Complex Systems* (Stanford University, 2011). URL `http://books.google.com.br/books?id=bynaAuMtiSAC`

63. T. Kaczanowski, *Practical unit testing with testNG and Mockito* (2012)

64. C. Geiger, (2014). URL `ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart{\_}fi/FACH-0188/FACH-0188.pdf`

65. A. Avritzer, A.B. Bondi, in *Resilience Assessment and Evaluation of Computing Systems* (2012), pp. 305–322

66. M. Mayo, S. Spacey, Lecture Notes in Computer Science (Springer Berlin Heidelberg, 2013), pp. 158–171. URL `http://link.springer.com/chapter/10.1007/978-3-642-39742-4{\_}13`

67. A. Avritzer, B. Larson, in *ACM SIGSOFT Software Engineering Notes* (ACM, New York, NY, USA, 1993), ISSTA '93, pp. 82–88. DOI 10.1145/154183.154244

68. J. Wegener, H. Sthamer, H. Pohlheim, Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054) (1999). DOI 10.1109/REAL.1999.818852

69. S. Abu-nimeh, S. Nair, M. Marchetti, (2001)

70. M.D. Barros, J. Shiau, . . . and Networks, 2007. . . . (2007)

71. Z.M.Z. Jiang, A.E.A. Hassan, G. Hamann, P. Flora, in *IEEE International Conference on Software Maintenance, ICSM* (2008), pp. 307–316. URL `http://ieeexplore.ieee.org/xpl/login.jsp?tp={\&}arnumber=4658079{\&}url=http://ieeexplore.ieee.org/xpls/abs{\_}all.jsp?arnumber=4658079http://ieeexplore.ieee.org/xpls/abs{\_}all.jsp?arnumber=4658079`

72. L. Yu, W.T. Tsai, X. Chen, L. Liu, Y. Zhao, L. Tang, W. Zhao, in *Proceedings - 5th IEEE International Symposium on Service-Oriented System Engineering, SOSE 2010* (2010), pp. 181–188. DOI 10.1109/SOSE.2010.36

73. T. Nivas, C. Csallner, in *European Conference on Software Engineering / Foundations of Software Engineering* (2011)

74. P. Bazilinskyy, M. Brunner, Student Conference on Optimisation of Software (2013). URL `http://www0.cs.ucl.ac.uk/staff/Yuanyuan.Zhang/StuConOS2013/stuconos2013{\_}submission{\_}3.pdf`

75. A.U. Gias, K. Sakib, in *Proceedings of the 28th international conference on Software engineering*, vol. undefined (2014), vol. undefined, pp. 608–609. DOI 10.1145/2591062.2591139. URL `http://dx.doi.org/10.1145/2591062.2591139`

76. A. Alebrahim, M. Heisel, in *ACM International Conference Proceeding Series*, vol. 08-12-July (2015), vol. 08-12-July. DOI 10.1145/2855321.2855357. URL `http://dx.doi.org/10.1145/2855321.2855357`

77. A. Avritzer, E. Weyuker, . . . international symposium on Software testing . . . pp. 44–57 (1994). DOI 10.1145/186258.186507

78. A. Avritzer, E. Weyuker, Software Engineering, IEEE . . . **21**(9), 705 (1995). URL `http://ieeexplore.ieee.org/xpl/login.jsp?tp={\&}arnumber=464549{\&}url=http://ieeexplore.ieee.org/xpls/abs{\_}all.jsp?arnumber=464549http://ieeexplore.ieee.org/xpls/abs{\_}all.jsp?arnumber=464549`

79. C.S.D. Yang, L.L. Pollock, SIGSOFT Softw. Eng. Notes **21**(3), 201 (1996). DOI 10.1145/226295.226318. URL `http://dl.acm.org/citation.cfm?id=226318http://doi.acm.org/10.1145/226295.226318`

80. E. Weyuker, F. Vokolos, IEEE transactions on software engineering **26**(12), 1147 (2000). DOI 10.1109/32.888628

81. I.K. El-far, J.a. Whittaker, Encyclopedia of Software Engineering pp. 1–22 (2001). DOI 10.1002/0471028959.sof207

82. M.a. Brown, E. Tapolcsanyi, Matrix pp. 1–17 (2003)

83. M.K. Aguilera, J.C. Mogul, J.L. Wiener, P. Reynolds, A. Muthitacharoen, ACM SIGOPS Operating Systems Review **37**, 74 (2003). DOI 10.1145/1165389.945454

84. C. Dumitrescu, I. Raicu, M. Ripeanu, I. Foster, Proceedings - IEEE/ACM International Workshop on Grid Computing pp. 289–296 (2004). DOI 10.1109/GRID.2004.21

85. S. Barber, . . . Energy Conference, 2004. INTELEC 2004. 26th . . . pp. 1–13 (2004). URL `http://ieeexplore.ieee.org/xpls/abs{\_}all.jsp?arnumber=1410995`

86. S. Kiran, A. Mohapatra, R. Swamy, 2nd International Symposium on Technology Management and Emerging Technologies, ISTMET 2015 - Proceeding pp. 74–78 (2015). DOI 10.1109/ISTMET.2015.7359004

87. D.C. Ag, D. Berlin, S. Wappler, Proceedings of the 8th annual conference on Genetic and evolutionary computation pp. 1917–1924 (2006)

88. Y. Cai, J. Grundy, J. Hosking, Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07 p. 353 (2007). DOI 10.1145/1321631.1321684. URL `http://portal.acm.org/citation.cfm?doid=1321631.1321684`

89. D. Nevedrov, pp. 1–11 (2007)

90. P. Fritzsche, D. Rexachs, E. Luque, 2007 4th IEEE Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (2007). DOI 10.1109/IDAACS.2007.4488453

91. M. Bayan, J.W. Cangussu, Proceedings of the 2008 ACM symposium on Applied computing - SAC '08 p. 661 (2008). DOI 10.1145/1363686.1363847. URL `http://portal.acm.org/citation.cfm?doid=1363686.1363847`

92. W. Afzal, R. Torkar, R. Feldt, Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008) (October 2015), 488 (2008). URL http://richard.torkar.googlepages.com/a{_}systematic{_}mapping{_}study{_}on{_}non-fu.pdf

93. C. Lutteroth, G. Weber, 2008 12th International IEEE Enterprise Distributed Object Computing Conference pp. 149–158 (2008). DOI 10.1109/EDOC.2008.40. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4634766

94. M. Acharya, V. Kommineni, ASE2009 - 24th IEEE/ACM International Conference on Automated Software Engineering pp. 409–420 (2009). DOI 10.1109/ASE.2009.95

95. Z.M. Jiang, A.E. Hassan, G. Hamann, P. Flora, 2009 IEEE International Conference on Software Maintenance pp. 125–134 (2009). DOI 10.1109/ICSM.2009.5306331. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5306331

96. X. Wang, B. Zhou, W. Li, International Symposium on Parallel and Distributed Processing with Applications pp. 483–490 (2010). DOI 10.1109/ISPA.2010.24

97. H. Malik, Software Engineering, 2010 ACM/IEEE 32nd . . . **2**, 421 (2010). DOI 10.1145/1810295.1810408

98. J. White, A. Pilbeam, p. 6 (2010). DOI 10.1.1.74.371. URL http://arxiv.org/abs/1010.3233

99. C. Babbar, N. Bajpai, D. Sarmah, International Journal of Technology (2011)

100. C. Barna, M. Litoiu, H. Ghanbari, International conference on Autonomi pp. 91–100 (2011)

101. T.H.D. Nguyen, B. Adams, Z.M. Jiang, A.E. Hassan, M. Nasser, P. Flora, Proceedings - Asia-Pacific Software Engineering Conference, APSEC pp. 282–289 (2011). DOI 10.1109/APSEC.2011.59

102. Y. Shoaib, O. Das, Electronic Notes in Theoretical Computer Science **275**, 123 (2011). DOI 10.1016/j.entcs.2011.09.009. URL http://dx.doi.org/10.1016/j.entcs.2011.09.009

103. I.d.S. Santos, A. Santos, P.d.S. Neto, SEKE (2011)

104. P. Zhang, S. Elbaum, M.B. Dwyer, Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering pp. 43–52 (2011). DOI 10.1109/ASE.2011.6100093. URL http://dl.acm.org/citation.cfm?id=2190151http://dx.doi.org/10.1109/ASE.2011.6100093

105. Computer Standards and Interfaces **33**(2), 152 (2011). DOI 10.1016/j.csi.2010.06.006

106. M. Yan, H. Sun, X. Wang, X. Liu, 2012 IEEE International Conference on Cluster Computing (2), 576 (2012). DOI 10.1109/CLUSTER.2012.20. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6337826

107. M. Vasar, S.N. Srirama, M. Dumas, Proceedings of the WICSA/ECSA 2012 Companion Volume on - WICSA/ECSA '12 p. 53 (2012). DOI 10.1145/2361999.2362008. URL http://dx.doi.org/10.1145/2361999.2362008$\backslash$nhttp://dl.acm.org/citation.cfm?doid=2361999.2362008

108. H. Malik, H. Hemmati, A.E. Hassan, pp. 1012–1021 (2013)

109. C. Barna, M. Shtern, M. Smit, V. Tzerpos, M. Litoiu, **X**(February) (2013). DOI 10.1145/0000000.0000000

110. Z.M. Jiang, A. Hassan, IEEE Transactions on Software Engineering **5589**(2), 1 (2015). DOI 10.1109/TSE.2015.2445340. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7123673

111. M. Harman, Y. Jia, Y. Zhang, 8th IEEE International Conference on Software Testing, Verification and Validation (ICST) (Icst) (2015). URL http://www0.cs.ucl.ac.uk/staff/mharman/icst15.pdf

112. M. Shousha, Performance Stress Testing of Real-Time Systems Using Genetic-Algorithms. Ph.D. thesis, Carleton University (2003)

113. A. Ganapathi, Predicting and optimizing system utilization and performance via statistical machine learning. Ph.D. thesis (2009). URL http://escholarship.org/uc/item/9b92g2pz.pdfhttp://www.mendeley.com/research/predicting-optimizing-system-utilization-performance-via-statistical-machine-learning/{\%}5Cnhttp://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-181.pdf