Towards Identifying the Best Variables for Failure Prediction using Injection of Realistic Software Faults

Ivano Irrera  
CISUC, Department of Informatics Engineering  
University of Coimbra  
Coimbra, Portugal  
ivano@dei.uc.pt

João Durães  
CISUC, Department of Informatics Engineering  
University of Coimbra  
jduras@dei.uc.pt

Marco Vieira  
CISUC, Department of Informatics Engineering  
University of Coimbra  
mvieira@dei.uc.pt

Henrique Madeira  
CISUC, Department of Informatics Engineering  
University of Coimbra  
henrique@dei.uc.pt

Abstract— Predicting failures at runtime is one of the most promising techniques to increase the availability of computer systems. However, failure prediction algorithms are still far from providing satisfactory results. In particular, the identification of the variables that show symptoms of incoming failures is a difficult problem. In this paper we propose an approach for identifying the most adequate variables for failure prediction. Realistic software faults are injected to accelerate the occurrence of system failures and thus generate a large amount of failure related data that is used to select, among hundreds of system variables, a small set that exhibits a clear correlation with failures. The proposed approach was experimentally evaluated using two configurations based on Windows XP. Results show that the proposed approach is quite effective and easy to use and that the injection of software faults is a powerful tool for improving the state of the art on failure prediction.

Keywords— Failure prediction, fault injection, software faults, availability

I. INTRODUCTION

Computer software has grown in complexity up to a point where its behavior is in part unpredictable. In fact, existing studies point software faults as the major cause of computer failures (e.g., [13][15]), and given the huge complexity of today’s software the weight of software faults tends to increase.

Several techniques are nowadays used to try to avoid failures, ranging from preventive software development processes and techniques (e.g., formal verification, testing) to proactive runtime measures (e.g., prediction, redundancy). However, although preventive development measures try to reduce faults, it is well known that deploying fault-free complex systems is an unachievable goal. Failure prediction is a powerful technique to decrease system failures rate by, for example, applying proactive measures such as failover to minimize or even eliminate downtime. Preliminary estimates show that five minutes in advance failure prediction can improve system availability by an order of magnitude [19].

Failure prediction algorithms can be classified in different classes, depending on the type of input data used [18], for example failure inter-arrival times’ analysis, error logs auditing, and failure tracking. A promising way to predict failures is to monitor system parameters, trying to identify symptoms of failures likely to occur in the near future [18]. In practice, system parameters (e.g., number of processes running, amount of memory allocated, mutexes) are continuously monitored at regular intervals of time, and models are applied to try to predict incoming failures, within a certain timeframe.

Failure prediction algorithms face a non-trivial problem: the selection of the system parameters to monitor. Although in a complex system there are hundreds or even thousands of variables that can be collected, some of those variables are more relevant for prediction than others. In fact, previous research shows that, even in a complex system, the set of variables that really matter is typically quite small [9]. The problem is that a large amount of field failure data is needed for identifying the variables that show the best symptoms and, because failures are rare events, obtaining that data typically takes a very long time, being even impossible in many scenarios [8].

In this paper we propose an approach for selecting the most adequate variables for failure prediction. The main idea is to inject realistic software faults to exercise the system and increase the probability of occurrence of realistic software-related failures (e.g., incorrect results, system crash, system hang). A large amount of failure-related data can thus be generated in short time. This data is automatically analyzed to identify the system variables that show symptoms of failures. These variables are then ranked based on their correlation with the observed failures, using the F-Measure proposed by van Rijsbergen [20].

Two configurations based on Windows XP and running real-world applications were used to demonstrate the effectiveness of the proposed approach. In our experiments we were able to collect data for a large number of system variables in a very short time, and to easily identify (and rank) the ones that actually showed a correlation with failures.

The outline of the paper is as follows. Section II presents background and related work. Section III describes the proposed methodology. Section IV presents the experimental evaluation and Section V discusses the results. Section VI concludes the paper.

II. BACKGROUND AND RELATED WORK

Online failure prediction is a particular class of failure prediction algorithms, whose goal is to identify, at runtime, if a
failure is likely to occur in the near future [18]. The output of online failure prediction can be either a decision that a failure is imminent, or some continuous measure that portrays how failure prone is the current system state. There are three key steps involved in failure prediction [9]: observation (selecting and capturing data), reasoning (interpreting data, and predict future system states), and reaction (using proactive recovery and failure avoidance schemes). In this paper we are particularly interested in the identification of the data that should be captured (i.e., which system variables should be monitored) during the observation phase.

In most of the failure prediction methods available today, there is a typical correlation between quality of prediction and the data used [18]. This way, selecting the most adequate variables is of utmost importance for the success of failure prediction. However, because obtaining failure related data can take a long time (as failures are rare events), in many cases the variables being monitored do not take into account the real operational profile of the system, thus not being the most indicated ones [8].

There are several methods for solving the problem of variable selection, which is a specific form of feature detection. As presented in [12], there are two main approaches for feature detection: the filter and the wrapper approach. In the prior (filtering) the feature selector filters the irrelevant attributes and is independent of any specific learning algorithm. In the later (wrapping), the most important features are filtered based on the algorithm and taking into account the specificities of the underlying learning mechanism. As a development of the wrapper approach, Hoffman [10] and Liu and Setonio [16] proposed the Probabilistic Wrapper Approach. This is a generic method that tries to conciliate the best of filtering and wrapping by playing with variables combination. Nevertheless, these approaches applied to Failure Prediction still suffer from the problem of requiring a large set of data to analyze, and of being deductive approaches.

Hoffmann et al [8] used real data from a telecommunication platform to compare two failure prediction techniques. Thirty gigabytes of data were collected during 53 days, over a four-month period. Ninety-two variables were monitored with a sample rate of one value per minute, resulting in 120960 readings per day. This is obviously a huge amount of time to collect data, which makes it impossible in many scenarios. Additionally, in [9] Hoffmann and Malek showed that only 2 out of the 92 variables monitored were indeed useful for failure prediction.

The approach proposed in this paper is based on the injection of realistic software faults to accelerate the occurrence of failures, and thus facilitate the process of data gathering. The best variables for failure prediction are then selected based on the automatic analysis of that data. By reproducing the chain of cause-effect by injecting software faults we can identify symptoms and select variables using an inductive approach (this approach can be considered a specific form of feature reduction).

Failures are deviations of the system from the expected behavior, and are provoked by errors in the system state. The adjudged or hypothesized cause of an error is a fault [2]. To produce failures in a realistic way we need to inject software faults, according to a realistic and representative model. Fault injection is the deliberate introduction of faults into a system, in such way that emulates real faults. It has become an attractive approach to validate specific fault handling mechanisms and to assess the impact of faults in actual systems, allowing the estimation of fault-tolerant system measures such as fault coverage and error latency [3][11]. The injection of software faults can thus be used to assess the impact of residual bugs on a system. In fact, it is widely accepted that the consequences of the realistic faults injected into a system are quite representative of the effect of residual software faults in the same system.

The problem of the injection of realistic software faults was first addressed by [4][5]. Several other works were conducted for studying the nature of software faults (e.g., [3][7]). G-SWFIT (Generic Software Fault Injection Technique, [7]) is a technique for the injection of realistic software faults, that make use of a library of machine-code level structures (or patterns) and possible software faults that, once introduced in such programming structures, can emulate specific classes of high-level software faults. The high-level software faults that are described in the G-SWFIT library are based on an extensive field study aimed to identify the list of bugs that can reasonably be expected to occur frequently [7]. Thus, it is possible to inject realistic software faults in a running system, according to a fault distribution that takes into account the most done errors during software development. G-SWFIT is the technique used in our work and is further explained in Section III.C.

The template is used to format your paper and style the text. All margins, column widths, line spaces, and text fonts are prescribed; please do not alter them. You may note peculiarities that arise from this style; however, we urge you to use it. It is designed to be as aesthetically appealing as possible, for no other reason than to concentrate the attention of the reader on the content of your paper.

III. VARIABLES SELECTION APPROACH

The proposed approach is essentially based on a set of experiments that include running a workload, injecting software faults, and collecting and analyzing data. The setup includes a monitored system (system for which we want to identify the best failure prediction variables) and a driver system (system in charge of controlling the execution of the experiments). These two systems should be accommodated in different machines (to isolate the effects of experimental control from the monitored system) connected using a dedicated network (to avoid interference from external network traffic). The approach includes five phases:

- **Preparation phase**: setup the experimental environment, which includes the definition of the workflow, the installation and configuration of the fault injection and monitoring infrastructure, and the identification of a (large) set of system parameters to be monitored.

- **Profiling phase**: the workload is run several times (each run is named a golden run) in the absence of artificial faults
in order to collect data for later building a profile of the typical behavior of each parameter being monitored.

- **Fault injection phase**: the workload is run several times, and during each run realistic software faults are injected (each run is named fault injection run) with the goal of generating real failure-related data. It is expected that some of the faults injected eventually lead to errors, and thus to the observation of failures.

- **Symptoms identification phase**: analysis of the data collected in order to build a model of the typical behavior of each parameter and to identify the parameters that show potential symptoms by deviating from that model.

- **Variables ranking phase**: the potential symptoms are correlated with the observed failures and used to rank the system variables. The variables that show the highest rate of valid symptoms (characterized using the F-Measure [20], which represents the harmonic mean of precision and recall) are the most adequate to be used for failure prediction.

A. Preparation phase

A workload to exercise the system needs to be defined. This workload should accurately represent the operational profile of the system being monitored as it may determine the best parameters for failure prediction.

Setting up the experimental environment includes the installation/configuration of the fault injection and monitoring tools: the prior is responsible for performing the injection of realistic software faults (e.g., a tool implementing the G-SWFIT technique [7]) and the latter for collecting data on system parameters.

There are several tools available for computer systems monitoring (e.g., LogMan for Windows XP and SAR for Linux) that allow gathering data on a huge number of parameters. However, monitoring a large number of variables may become impracticable due to the extremely huge amount of data generated and to the potential intrusion in the system being monitored. Additionally, there are parameters that are obviously useless for failure prediction (e.g., discrete parameters that show a constant value independently of the system state). This way, a preliminary filtering of the parameters to be monitored is needed.

Filtering the parameters to monitor (or exclude from monitoring) is a step that deserves some attention, as it is essential not to exclude parameters with a high potential for failure prediction. On the other hand, monitoring useless parameters adds complexity and overhead to the monitoring process and to the variables identification phase. However, in case of doubt regarding the usefulness of a given parameter, it should be considered for monitoring. In other words, only the parameters obviously useless for monitoring should be excluded in this first filtering.

Our proposal is to select the set of variables to be monitored (P) based on the levels of the classical computer abstraction stack model (see Figure 1). For example, in our experiments we selected the Operating System and levels below. This way, the set of parameters monitored included those that describe the state of the operating system resources, state of the processes running, availability and usage of network related resources, and terminal and disk I/O activity.

B. Profiling phase

The goal of this phase is to collect data to build a set of models that reflect the operational profile of each parameter when running the workload. The idea is to monitor each parameter several times under almost the same load conditions, such that a profile can be obtained. During each golden run (G_i) data are collect for each parameter (P_j) at regular intervals of time (t) over a given observation period (T). At the end of all golden runs there are (T/t)*G data points for each parameter, where G is the total number of golden runs.

The number of golden runs, the observation period and the monitoring rate can have a strong impact on the model being built. The number of golden runs determines the number of data points for each observation instant t, where a reduced number of data points restrict the observed variation of the system parameters. The observation period should be long enough to guarantee that the data gathered in each run captures accurately the system behavior over time (e.g., too short observation periods might lead to missing important events). Finally, the monitoring rate establishes the time between observations. A large observation interval reduces the number of data points collected and provides a limited view of the variation of the system state over time. On the other hand, a small observation interval increases the amount of data that has to be collected and analyzed, the noise associated to these data,

<table>
<thead>
<tr>
<th>Computer abstraction stack model</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
</tr>
<tr>
<td>Application</td>
</tr>
<tr>
<td>Middleware</td>
</tr>
<tr>
<td>Operating System</td>
</tr>
<tr>
<td>Hardware</td>
</tr>
</tbody>
</table>

Figure 2. Computer abstraction model

Figure 1. Execution profiles in Profiling phase (a) and Fault Injection Phase (b).
and the intrusiveness of the monitoring infrastructure.

As shown in Figure 2 (a), the monitored system must be explicitly restarted at the beginning of each golden run (the goal is to keep the configuration of the target system as uniform as possible among different runs). Afterwards the monitoring tool is started and the workload executed. During the workload execution the monitoring tool gathers data on the parameters being monitored and sends that data to the driver machine. After concluding the workload execution the monitoring tool is stopped. Note that, the observation period starts after the ramp up time ($T_a$), which represents the time needed for the system to reach a steady state.

C. Fault Injection phase

In this phase fault injection is applied to increase the probability of the occurrence of errors and, consequently, the rate of system failures. The key idea is that some of the software faults injected eventually lead to deviations in the variables behavior, thus showing symptoms of failures. Similarly to the Profiling phase, during each fault injection run ($F_i$) data is collect on each parameter ($P_i$) being monitored. Parameters are monitored at regular intervals ($t$) for a given period of time ($T$) and the total number of data points for each parameter is ($T/t)*F$. In addition to the data on system parameters, in this phase we need to collect information on the failure modes. A key difficulty is that the concept of failure is broad and the failure modes depend obviously on the system characteristics and the service it provides. This way, a set of system failure modes has to be defined. Adapting failure mode definitions available in the literature can facilitate this task. We identified in the CRASH scale, proposed by Koopman and DeVale [14], a potential candidate. It distinguishes the following failure modes in the context of operating systems robustness: Catastrophic (OS becomes corrupted or the machine crashes or reboots), Restart (application hangs and must be terminated by force), Abort (abnormal termination of the application), Silent (no error is indicated by the OS on an operation that cannot be performed), and Hindering (the error code returned is not correct). As shown in Figure 2 (b), the system state must be explicitly restored at the beginning of each fault injection run (in order to avoid the accumulation of effects between different fault injection experiments). Afterwards, the monitoring tool is started and the workload executed. Software faults are injected after the system has achieved a steady state (instant $T_a$). During the run, the monitoring tool gathers data on the parameters using the same sampling rate as in the Golden phase and sends that data to the driver system. If there is no system crash or hang during the workload execution then, at the end of the run the monitoring tool is stopped. Based on the failure modes definition a set of tests is conducted to check whether the faults led to a failure.

The Generic Software Fault Injection Technique (GSWFIT) [7] tool is used to inject realistic software faults based on a set of fault injection operators that reproduce directly in the target executable code the instruction sequences that represent the most common types of high-level software faults. These fault injection operators resulted from a field study that analyzed and classified more than 500 real software faults discovered in several programs, identifying the most common (the “top-N”) types of software faults [7]. Table 1 shows the 12 most frequent types of faults classified (see details and other classes of faults in [7]). The locations where the injections are performed are selected by the analysis of the target code (done by the G-SWFIT tool), which allows the identification of the places where a given fault type could indeed realistically exist, and avoids using locations where faults of the intended type could not exist. For example, the MIFS fault type in Table 1 can only be injected in target code locations that represent an IF structure. The analysis of the target code is based on the knowledge of how the high-level constructs are translated into low-level instruction sequences [7].

It is important to emphasize that the representativeness of the faults injected ensures that the experiments are representative of real situations (i.e., faults injected emulate faults that are likely to exist in the target code). In practice, injecting faults that are representative of residual faults is a necessary condition to reproduce realistic failures [7][17].

D. Symptoms identification phase

The first step towards the identification of potential symptoms is to build a profile (or model) for each variable $M(P_i)$ representing its typical behavior in the presence of the workload. Several models can be considered (see [20] for examples). In our work we used a simple, but effective approach based on the definition of boundaries (see Figure 3). In practice, it consists of taking the maximum and the minimum values monitored at each instant $t_i$ over all the Golden Runs. This data is used to define two curves representing the upper and lower bounds of values for the parameter. As these boundaries are defined based on a limited

<table>
<thead>
<tr>
<th>Fault types</th>
<th>Description</th>
<th>% of total observed</th>
<th>ODC classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIFS</td>
<td>Missing &quot;if (cond) { statement(s) }&quot;</td>
<td>9.96 %</td>
<td>Algorithm</td>
</tr>
<tr>
<td>MFC</td>
<td>Missing function call</td>
<td>8.64 %</td>
<td>Algorithm</td>
</tr>
<tr>
<td>MLAC</td>
<td>Missing &quot;AND EXPR&quot; in expression used as branch condition</td>
<td>7.89 %</td>
<td>Checking</td>
</tr>
<tr>
<td>MIA</td>
<td>Missing &quot;if (cond)&quot; surrounding statement(s)</td>
<td>4.32 %</td>
<td>Checking</td>
</tr>
<tr>
<td>MLPF</td>
<td>Missing small and localized part of the algorithm</td>
<td>3.19 %</td>
<td>Algorithm</td>
</tr>
<tr>
<td>MVE</td>
<td>Missing variable assignment using an expression</td>
<td>3.00 %</td>
<td>Assignment</td>
</tr>
<tr>
<td>WLEC</td>
<td>Wrong logical expression used as branch condition</td>
<td>3.00 %</td>
<td>Checking</td>
</tr>
<tr>
<td>WVAV</td>
<td>Wrong value assigned to a value</td>
<td>2.44 %</td>
<td>Assignment</td>
</tr>
<tr>
<td>MVI</td>
<td>Missing variable initialization</td>
<td>2.25 %</td>
<td>Assignment</td>
</tr>
<tr>
<td>MVAV</td>
<td>Missing variable assignment using a value</td>
<td>2.25 %</td>
<td>Assignment</td>
</tr>
<tr>
<td>WAEP</td>
<td>Wrong arithmetic expression used in function call parameter</td>
<td>2.25 %</td>
<td>Interface</td>
</tr>
<tr>
<td>WPFV</td>
<td>Wrong variable used in parameter of function call</td>
<td>1.50 %</td>
<td>Interface</td>
</tr>
</tbody>
</table>

**Total faults coverage** | 50.69 %
number of observations (that might not cover all possible parameter values) we need to define a tolerance, in percentage, to add (in the case of the upper bound) or remove from the bounds, thus to tolerate small measurement errors. This value depends on the system being monitored and should be defined based on experimental data observation.

The second step involves the identification of potential symptoms of failures by comparing the data from the fault injection runs to the models. In our case, for each parameter we look for values overrunning the model boundaries. In each fault injection run, a surface metric is used to measure how long a variable showed a deviating behavior, and how much was that deviation. In practice, a parameter shows a potential symptom if the area between its values over time and the upper or lower bound of the model is greater than a given threshold $S$ (specific for each variable). Note that in this phase we do not take into account the observed failures (i.e., we perform the analysis for all fault injection runs). This way, the deviations identified represent potential symptoms that need to be later correlated with those failures (see Section III.V).

The proposed model is summarized in the Equations (1), (2) and (3). To measure the deviation of a variable from its profile we need to define a surface metric, taking the distance between the overrunning points and the overrun boundary (the grey area in Figure 3). A parameter $p_i$ shows a symptom if:

$$\int_{x}^{y} |p_i(t) - \text{bound}(M(p_i(t))|dt \geq S$$

where bound(.) can be the upper or the lower boundary of the model, and $S$ is the threshold for that parameter. The interval $[x,y]$ is the interval in which the parameter shows overrunning values:

$$[x,y] = T_0 \cup T_1$$

where:

$$T_0 = \{p_i(t) \geq \text{MAX}(M(P_i(t)))\}$$

$$T_1 = \{p_i(t) \leq \text{MIN}(M(P_i(t)))\}$$

As said, a threshold allows us to identify a symptom when a fault injected activates. As the impact of the activation can be different on each parameter, a different threshold for each parameter should be defined. Thus, the threshold $S_i$ relative to the parameter $p_i$ is defined considering $p_i$ as a predictor, and maximizing its predicting power identified by its F-Measure:

$$S_i = \max_{S_{\min}, S_{\max}} F - \text{measure}(p_i)$$

The threshold is obviously very important as it influences quantities like the number of false positives (i.e., number of times the surface metric exceeds the threshold during fault injection runs in which no failures where observed) and the coverage (i.e., number of times the surface metric exceeds the threshold during fault injection runs in which failures were observed). The threshold is also needed because small noisy deviations are expectable. It mitigates the cases where values go slightly out of the typical bounds (or go out of the bounds for a very short time frame).

As an example, Figure 3 presents the profile of the system parameter Semaphores and an illustration of an identified symptom. The two bold lines represent the upper and lower bounds of the model, the black line represents the actual values coming from a Fault Injection Run (in which a hang failure was observed), and the gray area represents the surface identifying the potential symptom. In this example, the software fault is injected at $t=68$ seconds after starting the workload execution ($t=0$), and parameter values start overrunning the bounds at instant $t=135$ seconds.

It is important to emphasize that we are aware of the simplicity of the symptoms identification method used. In practice, we follow a pessimistic approach that could potentially harm the effectiveness of the overall methodology. However, if a simple symptom identification method allows identifying best variables, then it means that the proposed approach has a high practical value and can be easily improved by using a more elaborated symptom identification method.

E. Variable ranking phase

In this final phase the potential symptoms identified in the previous phase are correlated with the observed failures and used to rank the system variables. In practice, from the potential symptoms observed for each variable we need to calculate the following:

- **True positives**: number of fault injection runs in which the variable showed a symptom and a failure was observed.
- **True negatives**: number of fault injection runs in which the variable did not show any symptom and no failure was observed.
- **False positives**: number of fault injection runs in which the variable showed a symptom but no failure was observed.
- **False negatives**: number of fault injection runs in which the variable did not show any symptom but a failure was observed.

The variables that show the highest rate of valid symptoms are the most adequate to be used for failure prediction. To characterize the correlation of variables with the observed failures we propose the use of the F-Measure, introduced by
van Rijsbergen [20]. This measure represents the harmonic mean of the two popular measures used for evaluating quality of prediction, which are:

- **Precision**: a ratio of correctly predicted failures to the number of all predicted failures. In our context it can be represented as follows:

\[
\text{precision} = \frac{TP}{TP + FP}
\]  

(5)

- **Recall**: a ratio of correctly predicted failures to the number of true failures. In our context it can be represented as follows:

\[
\text{recall} = \frac{TP}{TP + FN}
\]  

(6)

- Assuming an equal weight for precision and recall [20], the F-Measure formula is:

\[
F - \text{measure} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}
\]  

(7)

The higher the predictive power of the variable, the higher is the F-Measure (which obviously ranges from 0 to 1). For example, consider a set of 100 fault injection runs that lead to system failures. A parameter that achieves a precision of 0.7 generates correct failure warnings (referring to true failures) with a probability of 0.7. A recall of 0.8 expresses that 80% of all true failures could be predicted and 20% are missed. In this case the F-measure would be approximately 0.7466. After calculating the F-Measure for each parameter, a ranking can be established. Obviously, different ranks can also be obtained using the individual precisions and recall measures.

IV. EXPERIMENTAL SETUP

To demonstrate the effectiveness of the proposed approach we have conducted two sets of experiments, each one using a different workload. As shown in Figure 4, the experimental setup consisted of two key elements: a monitored system, on which the faults were injected, and a driver system, for controlling the experiments and collect, archive, and analyze monitored data. Both the monitored system and the driver system consisted of a machine with a Pentium IV HT 3Ghz processor, 2GB of RAM, and a 200GB SATA hard disk, running Windows XP (SP3) Operating System. The two machines are connected via a Fast Ethernet network.

Two different **workloads** have been considered in our experiments in order to assess if different operational profiles may lead to different failure prediction symptoms. These two workloads stress the system in different ways. The first (WKL#1) is a light workload that is based on the 7-Zip application [1], which compresses a 4GB file using the ZIP algorithm, with the low compression option. The second (WKL#2) is a heavier workload based on the COSBI OpenSourceMark benchmark suite [6], consisting of a set of computation and input/output intensive tests, which include: compression algorithms, hardware testing, disk accesses, memory accesses, etc.

A tool that implements the G-SWFIT technique [7] is used for the injection of software faults in OS code. This tool runs on the monitored system and is controlled by the driver system (i.e., the driver system sends commands to the fault injection tool identifying the faults to be injected and the corresponding triggers). Different faults (characterized by the fault type, fault location, and fault injection instant) are injected in the Operating System. As the OS code is extremely large, specific portions were previously selected as prime candidates for fault injection. These comprise the code of the dynamic libraries (including the kernel32.dll and ntdll.dll system modules) used by the system process svchost.exe (Generic Host Process for Win32 Services). An important aspect is that the fault injection tool is run in both the golden phase (but with no fault injection) and in the fault injection phase. This assures that the impact of the fault injection tool in the system operational profile is similar in both phases.

For **data monitoring** we used the LogMan tool, a Windows application provided by Microsoft. This tool allows collecting thousands of parameters with a maximum sample rate of 1 value per second, which was the rate used in our experiments. However, as mentioned before, there are parameters that are useless for failure prediction. This way, we performed an initial manual filtering of the parameters that LogMan allows collecting and excluded a large set of them. The resulting set included 387 parameters that describe the state of the operating system resources, the state of the processes running, the availability and usage of network related resources, and information on terminal and disk I/O activity. It is important to emphasize that we did put great care on this step and tried not to exclude potentially good parameters (in case of doubt, we considered the parameter for monitoring).

V. RESULTS AND DISCUSSION

Table 2 presents the overall characterization of the experiments. A total of 1100 golden runs and 1143 fault

\begin{table}[h]
\centering
\begin{tabular}{ |c|c|c|c|c|c|c| }
\hline
\textbf{Workload} & \textbf{Golden Runs} & \textbf{Fault Injection Runs} & \textbf{Total %} & \textbf{Incorrect Results %} & \textbf{System Crash %} & \textbf{System Hang %} \\
\hline
WKL#1 & 500 & 500 & 22.20% & 0 & 4% & 18.20% \\
WKL#2 & 600 & 643 & 15.24% & 1.87% & 3.58% & 10.58% \\
\hline
\end{tabular}
\caption{Overall characterization of the experiments}
\end{table}
injection runs were conducted. The duration of each run was of 600 seconds, leading to an experimental campaign of 16 days (around 1,345,000 data points). In each fault injection run, software faults were injected approximately 70 seconds after starting the execution of the workload (value defined based on the analysis of the ramp up time of the tested configurations). Following the recommendations from [7], the number of faults injected ranged from 1 to 5. Failures were observed in a subset of the fault injection runs (111 for the configuration using WKL#1 and 98 for configuration using WKL#2). This is an expected result as in some cases the injected faults might not be activated (e.g., if the code where faults are injected is not executed) or, even if they are activated, the errors may not end up leading to failures observable during the experiments duration.

As Table 2 shows, we observed the failure modes:

- **Incorrect result**: the workload execution completes but the results are incorrect.
- **System crash**: OS becomes corrupted or the machine crashes or reboots.
- **System hang**: application or OS hangs and must be terminated by force.

The most predominant failure mode observed was the system hang and the least frequent was the generation of incorrect results. This shows that in most failure situations the faults injected leaded the OS to block and that the propagation of errors to the application level was minimum.

Figure 5 shows the F-Measure for the 387 parameters monitored in both configurations: only a small number of parameters present an F-Measure greater than zero (77 and 109 for the configurations running WKL#1 and WKL#2, respectively), which suggests that most of the monitored parameters are not useful for supporting failure prediction. In fact, only a few parameters present an F-Measure higher than 50% (17 in WKL#1 and 12 in WKL#2). This shows that the predictive power of the parameters monitored is quite limited.

Table 3 presents the Top-10 parameters (ranked using the F-Measure) for both configurations. The tolerance used for the bounds of the models representing the typical behavior of each parameter was of 10% (tuned based on the analysis of the experimental results). As we can see, the top-parameters are not the same for both system configurations (WKL#1 and WKL#2). However, there are five parameters that show up in both cases, with quite similar F-Measure (rows in gray). Although this confirms that the predictive value of the parameters might be influenced by the operational profile of the system, it also suggests that there is a small set of parameters whose predictive power is quite independent of the configuration.

To better understand the parameters correlation with failures, Figure 6 shows the detailed results for the Top-10 parameters for each configuration. As shown, precision (which characterizes the tendency for showing false alarms) is quite high for WKL#1 (always above 90%), but significantly lower for WKL#2 (less than 90% in six cases). On the other hand, recall is quite low in both configurations, which suggests that...
there were a large number of cases in which variables were not able to, individually, show symptoms of incoming failures.

VI. CONCLUSIONS AND FUTURE WORK

This paper presented an approach for the identification of the best variables for failure prediction. Software faults are injected to facilitate the generation of large amounts of failure related data that is used to select a small set of variables that exhibits a clear correlation with failures. The F-Measure, that represents the harmonic mean of the two most popular measures used for evaluating a quality of prediction (precision and recall), is used to characterize the variables.

A Windows based system running two different workloads was used to experimentally evaluate the approach. Results show that our approach can be used to easily identify the parameters that show some correlation to failures. In fact, we were able to establish a rank of the parameters using the F-Measure. We could also observe that different operational profiles (i.e., different workloads) may influence the predictive value of some variables (actually, in the two experimental scenarios considered the top ranked variables are not exactly the same). In summary, the injection of realistic software faults seems to be a powerful tool for improving the state of the art on failure prediction. Future work includes studying the possibility of composing parameters to achieve a higher prediction capability, and the applicability of our methodology in the context of dynamic systems. Other ideas include evaluating the figures of merit of prediction algorithms and accelerating the learning/training phase of prediction algorithms.

TABLE III. TOP-10 PARAMETERS, ACCORDING TO F-MEASURE (IN PERCENTAGE)

<table>
<thead>
<tr>
<th>Workload #1</th>
<th>Parameter</th>
<th>Parameter Name</th>
<th>F-Measure</th>
<th>Workload #2</th>
<th>Parameter</th>
<th>Parameter Name</th>
<th>F-Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>Pools Nonpaged Bytes</td>
<td>69.76</td>
<td>203</td>
<td>Events</td>
<td>69.28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>203</td>
<td>Events</td>
<td>68.23</td>
<td>237</td>
<td>Handle Count</td>
<td>66.66</td>
<td></td>
<td></td>
</tr>
<tr>
<td>237</td>
<td>Handle Count</td>
<td>67.85</td>
<td>204</td>
<td>Mutexes</td>
<td>66.66</td>
<td></td>
<td></td>
</tr>
<tr>
<td>251</td>
<td>Pools Paged Bytes</td>
<td>67.83</td>
<td>250</td>
<td>Pool Non-paged bytes</td>
<td>62.27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>Virtual Bytes Peak</td>
<td>67.45</td>
<td>205</td>
<td>Processes</td>
<td>55.47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>255</td>
<td>Virtual Bytes</td>
<td>66.66</td>
<td>149</td>
<td>Memory Avail. Bytes</td>
<td>52.68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>204</td>
<td>Mutexes</td>
<td>66.26</td>
<td>150</td>
<td>Memory Available KB</td>
<td>52.68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>208</td>
<td>Threads (Objects)</td>
<td>65.86</td>
<td>207</td>
<td>Semaphores</td>
<td>52.48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>254</td>
<td>Thread Count</td>
<td>65.86</td>
<td>254</td>
<td>Thread Count</td>
<td>51.89</td>
<td></td>
<td></td>
</tr>
<tr>
<td>361</td>
<td>Threads (System)</td>
<td>65.86</td>
<td>151</td>
<td>Memory Available MB</td>
<td>51.89</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


is not clear HERE that our approach wants to demonstrate that fault injection is a good idea to improve variable selection for failure prediction

manca riferimento al mio primo paper!!!!!!!!!!!!!