Generic Faultloads Based on Software Faults for Dependability Benchmarking

João Durães
ISEC/CISUC - Polytechnic Institute of Coimbra
3030 Coimbra - Portugal
jduraes@isec.pt

Henrique Madeira
DEI/CISUC - University of Coimbra
3030 Coimbra – Portugal
henrique@dei.uc.pt

Abstract

The most critical component of a dependability benchmark is the faultload, as it should represent a repeatable, portable, representative, and generally accepted set of faults. These properties are essential to achieve the desired standardization level required by a dependability benchmark but, unfortunately, are very hard to achieve. This is particularly true for software faults, which surely accounts for the fact that this important class of faults has never been used in known dependability benchmark proposals. This paper proposes a new methodology for the definition of faultloads based on software faults for dependability benchmarking. Faultload properties such as repeatability, portability and scalability are also analyzed and validated through experimentation using a case study of dependability benchmarking of web-servers. We concluded that software fault-based faultloads generated using our methodology are appropriate and useful for dependability benchmarking. As our methodology is not tied to any specific software vendor or platform, it can be used to generate faultloads for the evaluation of any software product such as OLTP systems.

1. Introduction

The idea of benchmarking dependability features of computer systems or computer components has caught the attention of researchers and practitioners in recent years. After two decades of a success story in the area of performance benchmarks, the notion that measuring pure performance is not enough in many cases becomes more and more evident.

The performance benchmarking field has originated solid industry driven organizations such as the TPC (Transaction Processing Performance Council) and the SPEC (Standard Performance Evaluation Corporation), that have released many successful performance benchmarks. These performance benchmarks have contributed to improve peak performance of successive generations of computer systems, particularly in what concerns key components such as processors and graphic boards, at the hardware level, or database management systems (DBMS) and web-servers, at the software level.

Unfortunately, the seek for pure peak performance also have caused that, in many cases, the systems and configurations used to achieve the best performance are very far from the systems that are actually used in practice (this is in fact the main criticism on TPC and SPEC benchmarks). The fact that many businesses and applications require high availability, reliability, integrity, or other dependability attributes shows that it is necessary to shift the focus from measuring pure performance to the measurement of both performance and dependability. This is just the goal of the dependability benchmarks.

A dependability benchmark can then be defined as a specification of a standard procedure to assess dependability related measures of a computer system or computer component. The key aspect that distinguishes dependability benchmarking from existing dependability evaluation and validation techniques is the standard nature required by a dependability benchmark. This standardization can only be achieved through an agreement (explicit or tacit) from the computer industry and/or by the user community. However, the contribution of the research community is essential to show possible solutions for the complex technical problems posed by dependability benchmarking.

Many research works have established the ground for the proposal of dependability benchmarks. In addition to many works on experimental dependability evaluation, especially on the field of fault injection and robustness testing, the first works that have carved the concept of dependability benchmark are [1,2,3,4]. More recent research works have proposed and demonstrated fully functional dependability benchmarks [5,6,7,8,9,10]. Most of these proposals elaborate on the typical scenario used by performance benchmarks, which consists of four main elements: benchmarking setup, measures, workload, and procedures & rules, and add two new components: **dependability measures** and **faultload**.

The faultload is in fact the most critical component of a dependability benchmark, as it should represent a repeatable, portable, representative, and generally accepted set of faults. Years of research in the field of dependability, especially on topics such as fault injection and analysis of field data on fault manifestations, have shown that the definition of a set of faults with the properties mentioned above is very hard to attain.

It is not a surprise the fact that all the dependability benchmarks proposed so far include as faultload only operator faults [5, 6, 10] and hardware faults [8, 9]. Software faults have been completely absent from this research effort. Unfortunately, it is well-known that most of the computer outages are caused by residual software faults, which means that all the benchmark faultloads proposed so far simply ignore the most frequent source of problems in computer industry: the software faults.

This is precisely the goal of this paper, as we propose a methodology to define faultloads based on software faults. Our methodology builds on previous published results based on field data and uses a fault injection technique based on machine code mutations to emulate programming errors. The paper also shows how the proposed method can be used to define a faultload based on software faults for a dependability benchmark of web-servers, and actually presents the first comparative dependability benchmarking of two well-known web-servers. Properties such as repeatability, portability and scalability are also analyzed and validated through the set of experiments presented in the paper.

The remainder of this paper is organized as follows: the next section presents the proposed methodology for the definition of faultloads based on software faults. Section 3 shows how to generate a faultload using our methodology for application in case-study scenario of dependability benchmarking; using the results of the previous section we discuss the validation of the faultload properties in section 4. Section 5 concludes the paper.

2. Faultload definition approach

The success of a benchmark is measured by its acceptance by the industry and user/research communities. To be accepted, a benchmark must verify a number of key properties:

- ***Representativeness***: the workload submitted to the system under evaluation must represent typical profile for a given application domain. It is worth noting that benchmarks are specific to an application domain (e.g., transactional applications) or a given type of component (e.g., operating systems). Additionally, the faultload must represent the typical faults experienced by those systems in the field. The present work focuses on software faults.
- ***Portability***. One of the most relevant uses of benchmarks is the comparison of a set of systems of a given category (DBMS, for instance). Therefore, it is of paramount importance that the benchmark can be used in the different systems used in a given application domain.
- ***Repeatable.*** By definition a benchmark is a tool that quantifies a given property of a system. Obviously, running the benchmark twice the user must obtain the same results (at least in statistical terms). Additionally, different teams must be able to reproduce the results obtained for a given system.
- ***Feasibility:*** the concept of feasibility is twofold: the effort needed to prepare and execute the benchmark must be low enough to allow its use by a large community of users; and the time needed to execute the benchmark must not be too long.
- ***Low intrusiveness.*** The introduction of some instrumentation in the system under observation is unavoidable. However, the perturbation caused by the instrumentation must be minimal in order to keep the results meaningful (too much intrusiveness would change the system under benchmark).

Fault representativeness is particularly difficult to assure for software faults, as this class of faults is particularly complex and its categorization is very difficult. Because of this, the identification of the most relevant faults to include in a faultload is not as straightforward as in the case of other fault types.

In addition to the properties mentioned above, which are difficult to attain for a faultload based on software faults, there are additional difficulties:

- ***Emulation accuracy***: as software faults have a complex nature the emulation of this type of faults is far more difficult than the traditional bit-flip fault injection.
- ***Fault injection target identification***: injecting a software fault ultimately means that the target code is changed in some way (a software fault is a programming error). Because the objective of the dependability benchmark is to observe the behavior of a given module, it follows that this module cannot be directly subjected to injection of faults. Indeed, if we inject faults in it, we would no longer have the original module and any conclusions drawn afterwards might not apply.

Generally, the subject of the benchmark is a subsystem or component that is part of a larger system, which may include a variety of others components (software & hardware). Thus, it is necessary to clarify the difference between two concepts:

- *Benchmark Target* (BT) is the system or component meant to be characterized by the benchmark.
- *System Under Benchmark* (SUB) is the complete system needed to run the workload, which is normally larger than the BT.

Because we cannot inject faults in the benchmark target, we establish a clear separation between the *fault injection target* component (FIT) and the *benchmark target* (BT). The FIT is the component where the faults are injected (also a part of the SUB). The idea supporting this approach is that faults are injected in one component with the purpose of evaluating their impact on another component or in the overall system. This makes particular sense in a COTS scenario, where a system integrator may want to know how a given component will react to the activation of hidden faults of another module (Fig. 1).

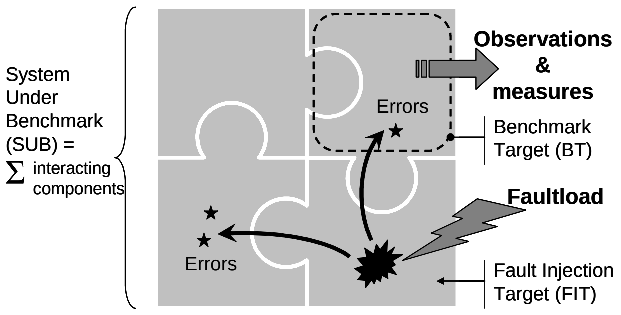
System Under Benchmark (SUB) = ∑ interacting components
Observations & measures
Errors
Benchmark Target (BT)
Faultload
Errors
Fault Injection Target (FIT)

Figure 1 – Fault injection target

The choice of an adequate FIT introduces the additional issues of fault activation rate and impact on the benchmark domain. The FIT must be chosen in a way that ensures the maximum activation rate without limiting the scope of the benchmark.

We propose a methodology for the definition faultloads based on software faults that specifically address these issues in the following manner:

- We use published results on field-data studies [11, 12] to identify a set of representative faults types and include only those types in the faultload. Particularly, in [12] the most frequent software fault types that have occurred in a large number of real programs were identified.
- We use the fault-injection technique proposed in [13] to emulate software faults. The accuracy of this technique was evaluated and proved to be acceptable in [13].
- We choose a FIT that guaranties an optimized fault activation rate and does not imposes additional restriction on the benchmark scope.

The following subsections discuss in detail the proposed methodology.

2.1. Identification of representative faults

A representative faultload is one that contains only faults that are representative of the real faults (programmer errors) that elude traditional software testing techniques and are left undiscovered in software products after shipment. The best way to estimate which faults are representative is to analyze field data on software faults discovered in programs already deployed.

The results presented in [11, 12] identify a clear trend in the software faults that usually exist in available systems: a small set of well-defined fault types is responsible for a large part of the total software faults. This set of fault types is the optimum starting point for a faultload definition. [12] also presents an extension to the Orthogonal Defect Classification (ODC) [14] fault classification scheme. The new scheme is especially useful for the emulation of software faults. Faults are classified according to the point of view of the program context in which they occur and are closely related with programming language constructs.

According to this idea, a software defect is one or more programming language constructs (statements, expressions, etc) that are either missing, wrong or in excess. This leads to the classification of faults as: Missing construct, Wrong construct, or Extraneous construct. Faults in each of these main classes are then sub-divided according to the ODC classification. This composed classification is particularly pertinent when considering fault emulation, since emulating an omission (missing construct) is substantially different from emulating an extraneous construct.

Table 1 presents the fault types selected for inclusion in our faultload. It also reproduces the statistical information regarding the representativity of the fault types according to the complete set of faults used in [12]. Faults of the extraneous construct nature were responsible for a very small portion of the total number of faults and did not justify its inclusion in the faultload. It is worth noting that this small set of simple fault types represents half of the total faults and cover four different ODC types.

2.2 Emulation technique

Once identified the most frequent types of software faults, we need a technique to emulate them accurately. The technique G-SWFIT (Generic Software Fault Injection Technique) [13] inserts directly in the target code a sequence of processor instructions that emulate the intended fault. The modifications inserted in the target code (mutation) are such that reproduce the code that would have been generated by the compiler if the software faults were in the high level source code. G-SWGIT provides good accuracy emulating software faults [13] and has already proved to be a practical technique for dependability benchmarking experiments [7]. It also has important characteristics such as independency from source code availability and portability (see [13]).

Table 1 – Representativity of the fault types included in the faultload

Fault types	Description	Fault coverage	ODC types
MVI	Missing variable initialization	2.25 %	Assignment
MVAV	Missing variable assignment using a value	2.25 %	Assignment
MVAE	Missing variable assignment using an expression	3 %	Assignment
MIA	Missing "if (*cond*)" surrounding statement(s)	4.32 %	Checking
MLAC	Missing "AND EXPR" in expression used as branch condition	7.89 %	Checking
MFC	Missing function call	8.64 %	Algorithm
MIFS	Missing "If (*cond*) { statement(s) }"	9.96 %	Algorithm
MLPC	Missing small and localized part of the algorithm	3.19 %	Algorithm
WVAV	Wrong value assigned to a value	2.44 %	Assignment
WLEC	Wrong logical expression used as branch condition	3 %	Checking
WAEP	Wrong arithmetic expression used in parameter of function call	2.25 %	Interface
WPFV	Wrong variable used in parameter of function call	1.5 %	Interface
Total faults coverage		50.69 %	

G-SWFIT is based on a two steps methodology (Fig. 2): in the first step fault locations are identified (i.e., the faultload is generated); this is done prior to the actual experimentation. In step two the faults are actually injected. The second step is usually performed during the target execution, but it can also be performed beforehand.

The identification of the location faults in step 1 is performed through the automated scanning of the target executable code. The result of this scan is a map of the target identifying the locations suitable for the emulation of specific fault types. The scanning process is guided by a library of mutation operators which is previously defined according to a given fault model. Each operator describes one specific type of fault (sometimes more than one if the fault types are similar enough) and comprises two components: a search pattern and a low-level mutation definition. The search pattern is the set of matching rules for the identification of the locations where a given fault type can be emulated. The actual fault injection in step 2 is a very simple and low intrusive task, as each fault location have been previously identified in step 1.

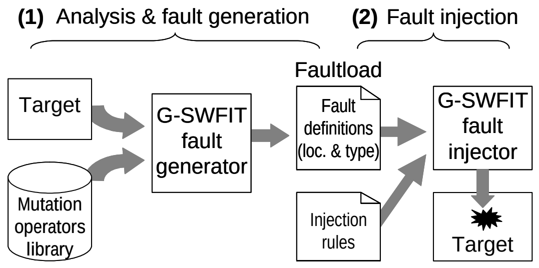

Figure 2 – G-SWFIT methodology.

The mutation operators related to the fault types of Table 1 constitute the mutation operator library used with in this work. Due to space restrictions we cannot reproduce the operators here (see [12, 15]).

2.3 Fault injection target identification

The identification of the fault injection target is a crucial step in the definition of the faultload. On one hand, faults must be injected in locations that guarantee an optimum activation rate; on the other hand, faults must not be injected in the benchmarking target itself, as seen before. Additionally, in order to ensure portability across different benchmark targets (i.e., use the same faultload in all of them), the FIT must be independent from the BT.

A strong candidate for the FIT role is the operating system (OS), especially when the BT is the application layer running on top of the OS (e.g., a database management system (DBMS), a web-server, transaction monitors, etc). Indeed, the OS verifies all the above requirements: it is indispensable of the SUB operation, it is independent from the application domain, and there is a clear frontier between the OS and the BT. The following reasons further justifies the selection of the OS as FIT:

- The use of the services of the operating system is unavoidable; if the portions of the operating system that are subjected to fault injection are adequately chosen, there will be a high fault activation rate.
- Hidden faults do exist on common available operating system. Several research works show this (e.g. [16, 2, 12]). By applying fault injection on OS, we are merely achieving the desired fault activation acceleration factor.
- By choosing the operating system as the target for fault injection any other software that runs on top of the OS can be benchmarked (because the condition that the FIT does not overlaps the BT is met). Experiments using the OS itself as BT are still possible if the target of the fault injection is carefully chosen resulting in a self-contained and clearly separated module, such as a device driver (see [7] for an example).

Although we specifically propose the operating system for the FIT role, other component of the SUB could conceivable be chosen (as long as it does not overlap with the BT itself).

2.4 Faultload fine-tuning

One of the main reasons for the use of fault injection techniques in dependability experiments is the need of an acceleration factor for the activation of faults. To achieve an optimum fault activation rate, there must be some assurance that the code subjected to faults is actually executed during the experiments. Such assurance can only be obtained if all fault locations are used. Because the OS is usually a large portion of code, using all fault locations would result in unfeasible experimentation time. Clearly, some sort of guide is necessary to identify the faults locations that have the highest probability of being activated during the execution of the dependability experiment (in a reasonable time period).

We propose a profiling phase to select the subset of the FIT code that is (most) used during the benchmark experimentation. To that effect, the SUB is exercised with the same workload that is used during actual benchmark execution. During the profiling phase, a trace of the API calls is used to identify the FIT code subset that is most used. The advantage of obtaining this FIT code subset is twofold: we have a high level of assurance that the faults injected there are activated; and, because the total number of faults injected is smaller, the time needed for experimentation is reduced.

To ensure that the FIT subset identified during the profile is representative of the FIT usage across different BT (so that faults injected there have an equivalent activation rate and the benchmark itself is fair), the profiling must be performed using several BTs of the same category (i.e., if the BT is a DMBS, the several DBMS must be used). The subset of the FIT that is selected for actual fault injection is the intersection of the results obtained with each different BT (again, of the same category). The resulting faultload is specific for a given OS and an intended domain (for example, for all the DBMS). This is not a real restriction, as existing (performance) benchmarks are already specific to a given application domain (e.g., we have TPC-C for OLTP applications, SPECWeb for web-server domain, etc.).

The methodology does not depend on the knowledge of the internals of either the operational environment (OS, platform) or the applications being observed. Thus this approach allows the generation of generic faultloads made of software faults that can be applied for dependability benchmarking in a broad range of domains. In the following section we apply this approach to define a faultload for web-server comparison regarding its behavior when executing in a faulty environment (OS).

3. Case study

In order to evaluate the proposed methodology in a realistic dependability benchmarking scenario we decided to extend the industry standard SPECWeb99 performance benchmark for web servers [17], resulting in the first dependability benchmarking experiment for web servers.

The benchmarking experiments were designed to show that the faultloads defined using the proposed methodology can really be used in future dependability benchmarks. Two different web-servers, Apache (available at www.apache.org) and Abyss (available at www.aprelium.com/abyssws) were benchmarked for comparative purposes running on top of two operating systems (MS Windows 2000/SP4 and MS Windows XP Pro/SP1), which is a realistic benchmarking scenario. Although we do not claim that these experiments constitute a formal proposal of the first dependability benchmarking for web servers, we believe that these results (which are effectively the first dependability benchmark results for web-servers) are a significant step in that direction.

The SPECWeb99 performance benchmark can be briefly described as follows [17]:

- **Benchmark setup**: SPECWeb99 uses one or more clients to submit requests to the web-server under evaluation. One of the clients (the prime client) coordinates all the others. The clients are usually executed in different machines, although that is not a requirement. Additionally, each client can be executed in different operating systems.
- **Workload**: The workload submitted to the server is representative of the average use of the common web-based services and is composed of the typical operations allowed by the HTML (GET and POST operations, both static and dynamic). The workload also reproduces common actions such as on-line registration and advertisement serving.
- **SPECWeb99 performance measures**: The measurements obtained through the SPECWeb99 client are mainly related to performance. The most relevant to our work are the following:
  - **SPEC**: This is the main SPECWeb99 metric. It measures the number of simultaneous conforming connection. SPEC defines **conforming connection** (CC%) as a connection with an average bit rate of at least 320 kbps and less than 1% of errors reported. This metric will be referred from now on as SPC.
  - **Throughput**: this is the number of operations (e.g., GETs and POSTs) per second (THR).
  - **Response time**: this is the average time in milliseconds that the operations requested by the client take to complete (RTM).

  - **Error rate**: this is the rate of errors found by the client in the requested operations (ER%).
- **Benchmark rules**: SPECWeb99 dictate very specific rules for experiment conduction. We refer the reader to [17] for more details on those rules. The most relevant at this point is that SPECWeb99 imposes that each benchmark experiment must be carried through a series of at least three batches of 1200 or more seconds each, separated by a *rampup* and a *rampdown* interval (both 300 seconds). At the beginning of each experiment there is a *warmup* period of 1200 seconds.

3.1 Experimental setup

Our experimental setup is composed of a *server machine* (Athlon XP 2600+, 512Mb) which hosts the web-server and the G-SWFIT injector, and a *client machine* (Pentium IV 2GHz, 512Mb), which runs the benchmarking client. Both machines are connected via a 100 Mbps Ethernet connection. The client machine never changes; the server machine embodies the SUB and has different instantiations according to the combination of OS and web-server used in each experiment (Fig. 3).

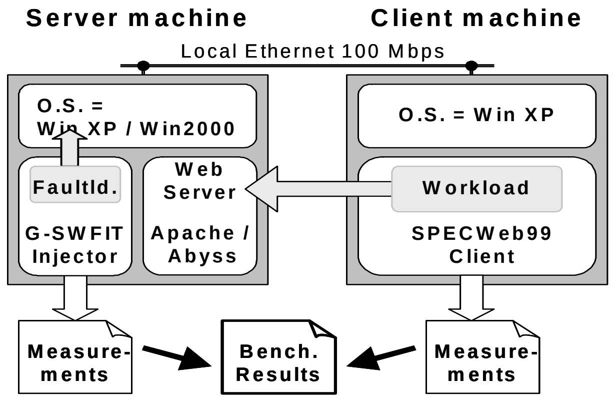

Figure 3 – Experimental setup overview.

Our G-SWFIT injector takes the faultload definition and injects each fault directly into the code of the running target. The injector is also monitors the web-server activity and, if needed, stop it and/or restart it. The G-SWFIT injector provides the following measures regarding the web-server error and availability status:

- Number of times the web-server died and did not self-restarted (**MIS**);
- Number of times the web-server had to be killed because it was not responding to HTML requests (**KNS**);
- Number of times the web client had to be killed because it was hogging the CPU and not providing service (**KCP**).

We configured the G-SWFIT fault injector to apply each software fault every 10 seconds. We reached the value of 10 seconds by observing the log files of the web-server after preliminary runs with the SPECWeb99. The average duration of the workload operations is less than a second, thus inserting each fault for a period of 10 seconds is enough time to activate the fault.

The injection of each fault may have one of the following consequences: it may be tolerated; it may simply cause performance degradation; it may cause the web-server to crash or die.

To comply with SPECWeb99 rules, we organized our experiments as a series of time slots. During each slot the web-server is exercised with the workload and the operating system is subjected to the faults defined in the faultload. Between each slot, the web-server is not exercised and no faults are injected (Fig. 4).

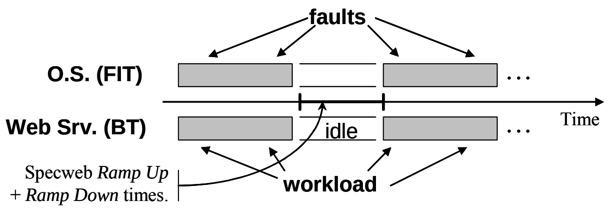

Figure 4 – Experiment structure.

3.2 Dependability benchmark metrics

The output of a benchmark is ideally a small, non-overlapping set of metrics with well-defined meanings. We propose a set of new metrics based on the readouts provided from the SPECWeb99 client and the G-SWFIT injector. These metrics provide an immediate comparative view of the web-servers:

- **Performance degradation**: This metric represent the penalty in the performance of the web-server caused by the faultload. It is composed by:
  - **SPCf**: Main SPEC measure in the presence of the faultload.
  - **THRf**: Throughput in the presence of the faultload.
  - **RTMf**: Response time in the presence of the faultload.
- **Need of administrator intervention – ADMf**: this metric gives an idea about the need of administrative intervention (human or automated) to repair the web-server. Administration intervention is needed when the web-server dies or stops providing useful service. The value for this metric can be computed as the sum of the values that represent all the situations where the web-server had to be restarted: MIS, KNS, and KCP.
- **Error rate in the presence of the faultload – ER%f**: this metric is the average percentage of errors discovered in the operations (extracted from ER%).

Table 2 – Relevant API calls.

API		Representativeness (%) of the total number of API				
Function name	Module	Apache	Abyss	Samba	Savant	**Average**
NtClose	Ntdll	2.8	1.6	2.4	0.8	**1.9**
NtCreateFile	Ntdll	0.7	0.3	0.4	0.3	**0.43**
NtOpenFile	Ntdll	1.4	0.5	1.4	0.3	**0.9**
NtProtectVirtualMemory	Ntdll	3.8	2.5	3.2	2.3	**2.95**
NtQueryVirtualMemory	Ntdll	1.9	1.1	1.6	1.1	**1.43**
NtReadFile	Ntdll	0.2	2.9	5.6	0.4	**2.28**
NtWriteFile	Ntdll	0.2	1.1	0.1	0.2	**0.4**
RtlAllocateHeap	Ntdll	14.6	17.6	4.3	17.5	**13.5**
RtlDosPathNameToNtPathName_U	Ntdll	2.3	1.2	1.9	0.8	**1.55**
RtlEnterCriticalSection	Ntdll	3.3	2.3	1.8	2.3	**2.43**
RtlFreeHeap	Ntdll	17.4	19.8	18.6	17.8	**18.4**
RtlFreeUnicodeString	Ntdll	0.3	0.8	0.9	0.6	**0.65**
RtlInitAnsiString	Ntdll	0.5	1.1	1.7	0.3	**0.9**
RtlInitUnicodeString	Ntdll	3.2	3.5	4.8	1.4	**3.23**
RtlLeaveCriticalSection	Ntdll	3.3	2.3	1.8	2.3	**2.43**
RtlUnicodeToMultiByteN	Ntdll	11.7	8.3	10.9	14.5	**11.35**
CloseHandle	Kernel32	0.9	1.1	0.9	0.2	**0.78**
GetLongPathNameW	Kernel32	0.1	0.1	0.1	0.1	**0.1**
ReadFile	Kernel32	0.2	2.9	5.6	0.1	**2.2**
SetFilePointer	Kernel32	0.2	0.2	0.1	0.1	**0.15**
WriteFile	Kernel32	0.2	1.1	0.1	0.1	**0.38**
Total call coverage		**69.2**	**72.3**	**68.2**	**63.5**	**68.34**

3.3 Faultload definition

We used an API tracing tool to discover which OS functions were the most used by each web-server. According to our methodology, the code of those API functions is the optimum target for fault injection. Because these functions belong to the operating system itself and not to the application being observed, we meet the condition that the observed application is not changed in any way. Only the API functions that were used by all the observed web-servers were eligible. Functions that were responsible for a negligible percentage of the total number of API calls were ignored. Table 2 presents the resulting set of functions. In addition to the Abyss and Apache web-servers, we also used Sambar and Savant web-servers in the profiling for the faultload fine tuning.

The most common used API call fall into the *kernel32* and *ntdll* system modules. This represents an additional advantage because, normally, all applications under Windows have these two modules mapped into its address space (thus improving the portability of the faultload). It is worth noting that although the number of selected functions is relatively small, it represents two thirds of all the calls made to the OS by the observed web-servers. It is also interesting to observe that the API usage pattern is very stable across all the four web-servers. This gives us some confidence in the assumption that other web-server will also possess a similar pattern.

We applied the G-SWFIT technique to generate the faultload. Because our work involves two operating systems, we obtained two different faultloads (Table 3).

3.4 Experimental results

Our injector has a special profile mode of operation designed to measure its performance overhead and intrusion in the overall system. While in that mode, all tasks related to an injection experiment are carried out but the injection itself does not change the target. We run a complete set of experiments with the injector in profile mode. The comparison of the results (performance and outputs) against the results observed without the injector running gives us a measure of the injector overhead and intrusiveness. Table 4 presents the results. The worst case of performance degradation is less than 2%; the average degradation is around 1%. Even in the worst case, the injector intrusiveness is not significant enough to affect the main SPECWeb figure. We conclude that the injector has a reasonably low performance penalty. No errors were signaled by the SPECWeb99 client nor registered in the log files, so we can conclude that the intrusion in the web-server operation is non-existent or very low.

Table 3 – Faultload details

	Number of faults per fault type												Total faults
	MVI	**MVAV**	**MVAE**	**MIA**	**MLAC**	**MFC**	**MIFS**	**MLPC**	**WVAV**	**WLEC**	**WAEP**	**WPFV**	
Windows 2000 SP4	149	4	129	497	147	392	200	50	33	71	11	31	1714
Windows XP SP1	192	5	117	899	253	629	471	94	59	163	11	34	2927

Table 4 – Performance degradation and intrusion evaluation.

	Windows 2000								Windows XP							
	Apache				Abyss				Apache				Abyss			
	SPC	CC%	THR	RTM	SPC	CC%	THR	RTM	SPC	CC%	THR	RTM	SPC	CC%	THR	RTM
Max. Perf.	37	100	104.2	354.2	34	100	95.9	355.5	34	100	93.9	361.2	33	100	93.7	352.5
Profile mode	37	100	103	358.1	34	100	95.3	358.1	34	100	92.9	365.5	33	100	92	359.4
Degradation (%)	0	0	1.15	1.1	0	0	0.63	0.73	0	0	1.06	1.19	0	0	1.81	1.96

The results observed with the injector running in profile mode correspond the baseline web-server behavior (i.e., the injector is considered as part of the load submitted to the server machine). The observations obtained during actual fault injection will be compared against the baseline behavior.

Each combination of web-server/operating system was tested three times (to comply with SPECWeb rules). Table 5 presents the results. We are specially interested in the average behavior variation of the web-servers relative its baseline behavior because that variation is an indicator of the web-server vulnerability (or tolerance) to faults existing in the OS.

The following observations are pertinent for the validation of the faultload properties:

- All three iterations yielded very similar results within each web-server/OS pair. This suggests that experiments using these faultloads are repeatable;
- The results allow a clear distinction between the two web-servers. This gives us confidence to the assumption that our faultloads are indeed useful for dependability experiments;
- The behavior of the two web-servers in presence of faults remains stable across both OSes. This is an important fact because it suggests that, although the faultloads are not exactly equal (but were generated through the same rules), they are equivalent in the sense that provide the same kind of BT behavior across different OSes.

Figure 5 presents a comparison of Apache and Abyss web-servers using our metrics. To allow for a clear comparison between Apache and Abyss, both web-servers appear side by side. To easily assess the impact of the faultloads over each web-server, we present both the baseline performance and the behavior with faults.

The results clearly point to the conclusion that the Apache web-server experiences less service and performance degradation than Abyss web-server when exposed to an operating system with software faults. All metrics give Apache an advantage over the Abyss: the performance relative to its normal condition (i.e., without faults) is better than in the case of Abyss; the percentage of erroneous operations lower; the number of times Apache had to be killed and/or restarted explicitly is lower than in the case of Abyss, thus requiring less administration intervention (this is consistent with the fact that Apache has a built-in mechanism that allows it to self-restart in some error situations).

It is worth noting that the relative difference between Apache and Abyss is the same for both Windows 2000 and Windows XP (see Figure 5). This leads us to conclude that the faultloads exposed an intrinsic property of the web-servers tested. This fact suggests that concrete conclusions can indeed be extracted regarding the comparison of the web-servers. Although the faultloads used for Windows 2000 and Windows XP are different (e.g., the number of fault is larger for XP), the results of the experiment are consistent. This leads to the conclusion that the methodology used for the definition of the faultloads is valid and portable across different operating systems.

Table 5 – Experimental results.

		B.T. = Apache web-server							B.T. = Abyss web-server						
		SPC	THR	RTM	ER%	MIS	KCP	KNS	SPC	THR	RTM	ER%	MIS	KCP	KNS
Win. 2000	Baseline Perf.	37	103	358.1	0	0	0	0	34	95.3	358.1	0	0	0	0
	Iteration 1	13.6	98.6	366.5	7.7	64	0	74	9.1	91.5	363.1	21.9	125	0	43
	Iteration 2	12.7	97.3	368.2	7.7	58	1	65	9.3	91.7	362.7	22.3	135	0	35
	Iteration 3	13.9	98.5	366.9	7.6	58	2	68	9	91.4	363.8	21.8	131	0	38
	Average (all iter)	13.4	98.1	367.2	7.7	60	1	69	9.1	91.5	363.2	21.9	130.3	0	38.7
Win. XP	Baseline Perf.	34	92.9	365.5	0	0	0	0	33	92	359.4	0	0	0	0
	Iteration 1	13.8	90.3	371.7	5.8	86	1	102	10	88.9	364.7	14.2	165	0	52
	Iteration 2	14.4	90.1	367.7	5.7	85	1	107	8.5	88.5	361.9	14.7	171	0	57
	Iteration 3	13.1	89.7	373.3	5.7	84	1	101	8.4	88.5	366.2	14.6	154	0	69
	Average (all iter)	13.7	90	370.8	5.7	85	1	103	8.9	88.6	364.3	14.5	163.3	0	59.3

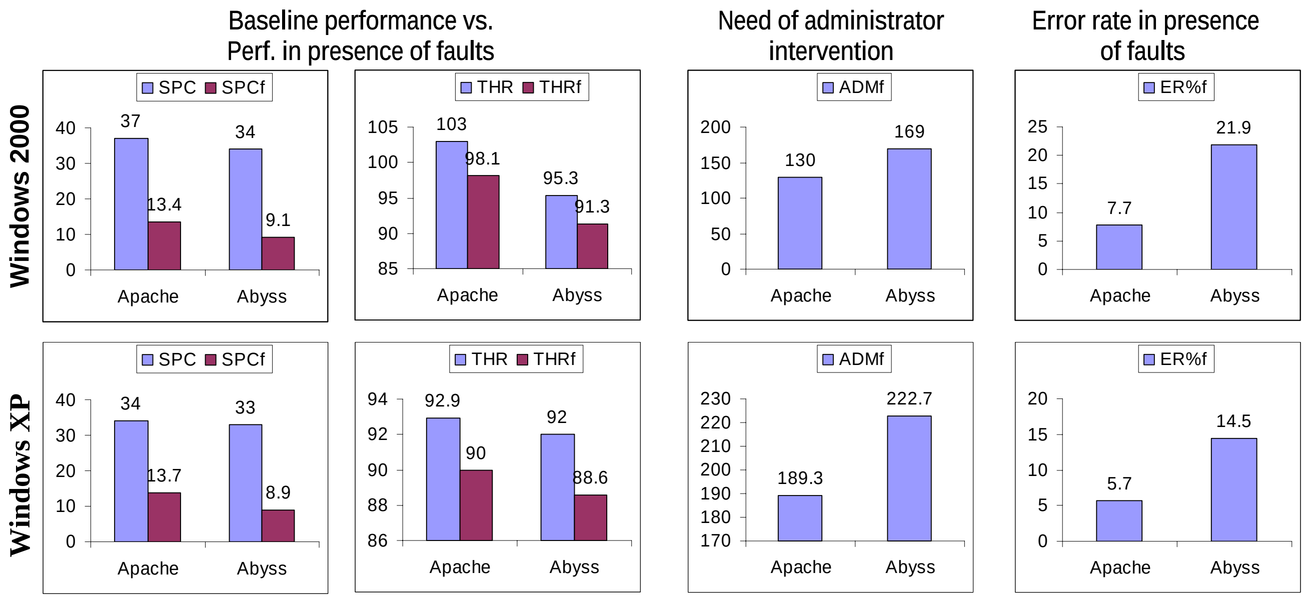

Figure 5 – Comparison of the behavior of Apache and Abyss in presence of software faults.

4. Faultload properties discussion

This section discusses the properties of the proposed faultload.

Representativeness: The faultloads used in this work are representative because they are based on representative types of software faults identified in previously published works using actual field data. Additionally, the fault locations reside in code that is representative of the most used OS API functions from the point of view of the BT.

Feasibility and benchmark execution time: In addition to showing the feasibility of the proposed approach, the case study also allows us to measure the execution time of the various steps. The time needed to generate the faultload is the time that the G-SWFIT takes to analyze the target (less than 5 minutes). The profile process used to fine tune the faultload took about 100 min. for each web-server. The duration of the experimentation is a function of the number of faults. Even with the larger faultload of the Windows XP, the experiments took about 24 hours (3 complete runs), which acceptable for dependability benchmarking.

Accuracy: As we use a technique that replicates the code that corresponds to typical programmer errors, we emulate the fault itself and not only its effects, thus providing a good accuracy. A detailed evaluation of the accuracy of the technique is provided in [13] (which concludes that this form of fault emulation has indeed a good accuracy).

Repeatability: The results of the experiments presented here show that our faultloads are repeatable: not only all iterations inside each web-sever/OS yielded similar results, but also we observed a clear behavior pattern that remains stable across the two different OSes.

Portability: The portability of the faultload is directly derived from the portability of G-SWFIT itself. In the work presented in [13] we have shown that G-SWFIT is portable: the main issue is the definition of the mutation operator library. That task is mainly dependent on the target processor architecture. In fact the programming language, the compiler, and the compiler optimization switches also have some influence on the library; however, such variations cause only the need of some additional operators in the library. Different processor architectures usually require different libraries. When porting the mutation library to other processors, all which is required is the specification of new rules and mutations. The technique itself remains the same.

Scalability: The faultload size is mainly dependent of the complexity of the FIT and not directly on the BT. In our case we have already used one of the most complex FIT available (the Windows XP) and still the experiments were reasonably feasible both in effort and time. Even using a larger BT, the faultload is not expected to grow significantly.

Non-intrusiveness: Our experiments show that the performance overhead of the faultload and the technique to deploy it is low (less than 2% in the worst case). Furthermore, no side-effects on the BT behavior were caused by the injector itself.

5. Conclusions

In this paper we presented a methodology for the definition of faultloads based on software faults for dependability research. The methodology does not depend on any specific properties of the target system or operational environment. Also it does not require any knowledge of the details of the target such as source code.

As a case study, the methodology was applied to benchmark the dependability of two different web-servers. Because we wanted to evaluate the web-server behavior, faults had to be injected in another component of the system. The operating system is a good choice when the benchmark target is in the application domain. We generated a faultload for Windows 2000 and a faultload for Windows XP. Both faultloads were fine-tuned to assure an optimum fault activation ratio. Faults were defined and injected using the G-SWFIT technique, which emulates software faults in a realistic manner.

The results of the experiments suggest that our faultloads are indeed useful to observe and compare the behavior of different applications in the presence of software faults. Our results enabled us to extract objective conclusions regarding the comparison between the two web-servers tested.

Based on the analysis of the methodology used to define the faultloads and on the results obtained with the experiments conducted, we concluded that our faultloads satisfy the key properties required for its use in dependability benchmarking contexts. Based on these experiments, a full dependability benchmark for web-servers can be defined by adding more fault models (hardware faults, operator faults, etc.) and measures.

An important characteristic of the methodology and the resulting faultloads is the fact that it is not tied to a specific type of platform. Therefore, the methodology and faultloads can be used in other experimental contexts, for example, DBMS dependability benchmarking.

References

[1] D. P. Siewiorek et al., "Developement of a Benchmark to Measure System Robustenness", Proc. of the IEEE International Fault Tolerant Computing Symposium, June 1993, pp. 88-97.

[2] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, T. Marz, "Comparing Operating Systems using Robustness Benchmarks", in Proceedings of the 16th International Symposium on Reliable Distributed Systems, SRDS-16, Durham, NC, USA, 1997.

[3] A. Mukherjee and D. P. Siewiorek, "Measuring software dependability by robustness benchmarking", IEEE Transactions on Software Engineering, Vol. 23, No. 6, June, 1997.

[4] T. Tsai and R, K. Iyer, "An Approach to Benchmarking of Fault-Tolerant Commercial Systems", Proc. of the 26th IEEE Fault Tolerant Computing Symposium, FTCS-26, Sendai, Japan, pp. 314-323, June 1996.

[5] M. Vieira and H. Madeira, "Benchmarking the Dependability of Different OLTP Systems", The International Conference on Dependable Systems and Networks, DSN-DCC2003, San Francisco, CA, June 22 - 25, 2003.

[6] Marco Vieira and Henrique Madeira, "A Dependability Benchmark for OLTP Application Environments", 29th International Conference on Very Large Data Bases, VLDB2003, Berlin, Germany, September 09-12, 2003.

[7] J. Durães, H. Madeira, "Multidimensional Characterization of the Impact of Faulty Drivers on the Operating Systems Behavior ", IEICE Transactions, Special Issue on Dependability Computing, Dec. 2003.

[8] K. Buchacker, M. Dal Cin, H.-J. Hoexer, R. Karch, V. Sieh, and O. Tschaeche, "Reproducible Dependability Benchmarking Experiments Based on Unambiguous Benchmark Setup Descriptions", The Int. Conf. on Dependable Systems and Networks, DSN-PDS03, San Francisco, CA, June 22-25, 2003.

[9] Ji J. Zhu, J. Mauro, and I. Pramanick, "Robustness Benchmarking for Hardware Maintenance Events", The International Conference on Dependable Systems and Networks, DSN2003, San Francisco, CA, June 22 - 25, 2003.

[10] A. Brown and D. Patterson, "To Err is Human", First Workshop on Evaluating and Architecting System Dependability (EASY), Joint organized with Int. Conf. on Dependable Systems and Networks, DSN-2001, Göteborg, Sweden, July, 2001

[11] J. Christmansson, Chillarege, "Generation of an Error Set that Emulates Software Faults", Proc. of the 26th IEEE Fault Tolerant Computing Symp., FTCS-26, Sendai, Japan, pp. 304-313, June 1996.

[12] J. Durães, H. Madeira, "Definition of Software Fault Emulation Operators: a Field Data Study", The Intl Conference on Dependability Systems and Networks, DSN-2003, San Francisco, CA, USA, June-2003

[13] J. Durães, H. Madeira, "Emulation of Software Faults by Educated Mutations at Machine-Code Level", Proc. of the 13th IEEE Int. Symp. on Software Reliability Engineering, ISSRE'02, Annapolis MD, USA, pp. 329-340, November 2002.

[14] R. Chillarege, "Orthogonal Defect Classification", Ch. 9 of "Handbook of Software Reliability Engineering", M. Lyu Ed., IEEE Computer Society Press, McGrow-Hill, 1995

[15] J. Durães, H. Madeira, "Definition of complete set of software fault emulation operators based on a field data Study", Technical Report DEI-005-2002, ISSN 0873-9293, Departamento de Engenharia Informática – FCTUC, 2002, http://www.dei.uc.pt/~henrique/G-SWFIT.htm.

[16] J.-C. Fabre, F. Salles, M. R. Moreno, J. Arlat, "Assessment of COTS Microkernels by Fault Injection", in Dependable Computing for Critical Applications (Proceedings of the 7th IFIP Working Conf. on Dependable Computing for Critical Applications: DCCA-7, San Jose, CA, USA, Jan 1999).

[17] "SPECWeb99 Benchmark", Standard Performance Evaluation Corporation, http://www.spec.org/web99.

Acknowledgements

Funding for this paper was provided, in part, by the Portuguese Government/European Union through R&D Unit 326/94 (CISUC) and by DBench project, IST 2000 - 25425 DBENCH.