2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

A Framework for Evaluating Software on Reduced Margins Hardware

Konstantinos Parasyris, Panos Koutsovasilis, Vassilis Vassiliadis,
Christos D. Antonopoulos, Nikolaos Bellas, Spyros Lalis
Department of Electrical and Computer Engineering
University of Thessaly
Email: {koparasy, pkoutsovasilis, vasiliad, cda, nbellas, lalis}@uth.gr

***Abstract*—To improve power efficiency, researchers are experimenting with dynamically adjusting the voltage and frequency margins of systems to just above the minimum required for reliable operation. Traditionally, manufacturers did not allow reducing these margins. Consequently, existing studies use system simulators, or software fault-injection methodologies, which are slow, inaccurate and cannot be applied on realistic workloads. However recent CPUs allow the operation outside the nominal voltage/frequency envelope. We present eXtended Margins eXperiment Manager (XM^2) which enables the evaluation of software on systems operating outside their nominal margins. It supports both bare-metal and OS-controlled execution using an API to control the fault injection procedure and provides automatic management of experimental campaigns. XM^2 requires, on average , 5.6% extra lines of code and increases the application execution time by 2.5%. To demonstrate the flexibility of XM^2, we perform three case studies: two employing bare-metal execution on a raspberry PI, and one featuring a full-fledged software stack (including OS) on an Intel Skylake Xeon processor.**

I. Introduction

As technology feature size scales and transistor variability increases, chip manufacturers resort to extra provisioning in terms of increased voltage margins and reduced operating frequency to guarantee correct CPU operation even under the worst possible combination of operating conditions. But such voltage and frequency guardbanding also leads to significant power and energy overheads.

To address this problem, researchers have been investigating a variety of undervolting and overclocking techniques for statically or dynamically shaving-off excessive margins [1], [2], [3], [4]. The effects on system reliability are typically modeled using system fault simulators, or by injecting faults (such as bit flips) at the application level during execution.

These approaches have two major drawbacks. First, system simulators are slow and have a limited capability for executing large workloads with realistic input sets. Simulation speed is important because a large number of fault injection campaigns is usually required in order to study a fault scenario and to reach conclusions with a high level of statistical confidence. Second, timing errors caused by undervolting/overclocking are difficult to model analytically, and are extremely sensitive to a large number of parameters, such as the number and distribution of paths in a CPU, the micro-architectural design of the CPU, the contents of the instruction and data streams, the manufacturing process, and even the ambient temperature, IR drops, aging, and so on. Even identical chips with the same microarchitecture, using the same technology libraries and executing identical code may exhibit different behavior [5]. Therefore, using simulation-based or software fault injection to study the effects of timing errors may be inaccurate, since it cannot capture all the complex effects that take place when the CPU is stressed beyond its nominal operating points.

Recently, a number of commercial platforms have been released, on which it is technically possible to set the operating frequency and voltage beyond nominal values. This makes it possible to evaluate the effects of reduced margins on software reliability while running the software natively at full speed [1], [6], [7], [8]. Building on the availability of such platforms, we present eXtended Margins eXperiment Manager (XM^2), a framework for conducting experiments on real hardware at non-nominal configurations. XM^2 can be used (i) to study, analyze and understand how and under what circumstances modern CPU microarchitectures fail when executing code at unsafe margins (sections VI-A and VI-B), and (ii) to evaluate energy gains when operating the CPU with reduced, but still safe margins (section VI-C). This is possible in modern CPUs as has already been shown recently for ARMv8 cores [1] (iii) to evaluate the resilience of applications – or the whole software stack – to errors, in a realistic setup.

XM^2 adopts a reusable, platform-neutral approach that can scale in simultaneous multi-board, multi-CPU and multi-process execution campaigns with or without operating system support. The framework supports user-defined methods for the collection and classification of the different execution outcomes, and can manage very large campaigns, thus relieving the user from manually initiating, controlling and monitoring the experiments. We evaluate and illustrate different use cases and the versatility of XM^2 with two completely different platform setups: a bare-metal ARM Cortex A53 processor, and a x86-64 Skylake Xeon processor running a full operating system stack.

The rest of the document is structured as follows: section II describes the requirements to enable the functionality of XM^2. Section III outlines the user interface, and section IV the design and implementation of our tool. Section V presents, evaluates and discusses the usability of XM^2. Section VI presents the case studies. Section VII gives an overview of related work. Finally, section VIII concludes the paper.

2158-3927/18/$31.00 Â©2018 IEEE
DOI 10.1109/DSN.2018.00043

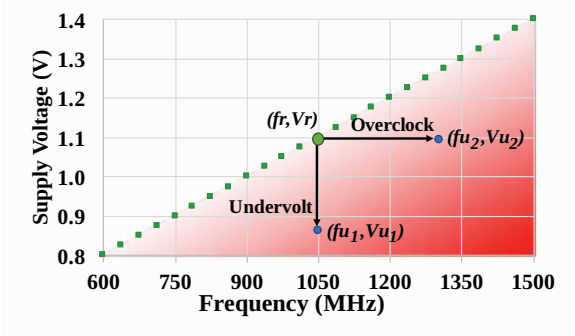

Fig. 1. The green points represent nominal (reliable) Voltage/Frequency configurations for an ARM Cortex A53. The red area below the line corresponds to the unreliable configuration space that can be exploited by XM^2.

II. Platform Requirements

In modern CPUs there are different sources of unreliability, which can be categorized according to the transient or permanent nature of the resulting errors [9]. Interestingly, transient errors may be induced in an attempt to improve energy efficiency or performance, by explicitly setting the supply voltage or the operating CPU frequency outside the manufacturer-defined working envelope. Scaling the supply voltage below and increasing CPU frequency above nominal values is referred to as undervolting and overclocking, respectively. Figure 1 illustrates the configuration space.

XM^2 facilitates the design and implementation of experimental campaigns to characterize either the hardware itself, or the resilience of software operating on top of overclocked or undervolted hardware. XM^2 can be used on top of different hardware platforms and software stack configurations. It assumes the following support from the underlying hardware and software of the platform used for the experiments:

Hardware support: The hardware must provide support for controlling and scaling the system operating point (voltage, frequency) beyond the normal working envelope. Modern Intel x86-64 CPUs offer such capabilities, starting from the Haswell family, through the programmable Fully Integrated Voltage Regulator (FIVR) [10]. Several processors based on ARM architectures offer similar functionality. The AppliedMicro X-Gene 2 [1] chip does so through the SLIMpro management processor included in the chip. The ARM Cortex A53 processor in Raspberry PI 3b boards can be set to operate in non-nominal conditions via a configuration file.

Compiler support for common function attributes: Every application using XM^2 must be linked with a thin library, used to notify external systems about the execution status of the application. It also undertakes the management of the application and supports data exchange with external systems. The library exploits the common function attributes *constructor* and *destructor* provided by the *gcc* compiler.

Connectivity: We assume that the target (tested) system can support TCP connections to other systems. These are used to orchestrate the execution campaign, to supply input data, and to collect results of the computation from the target system.

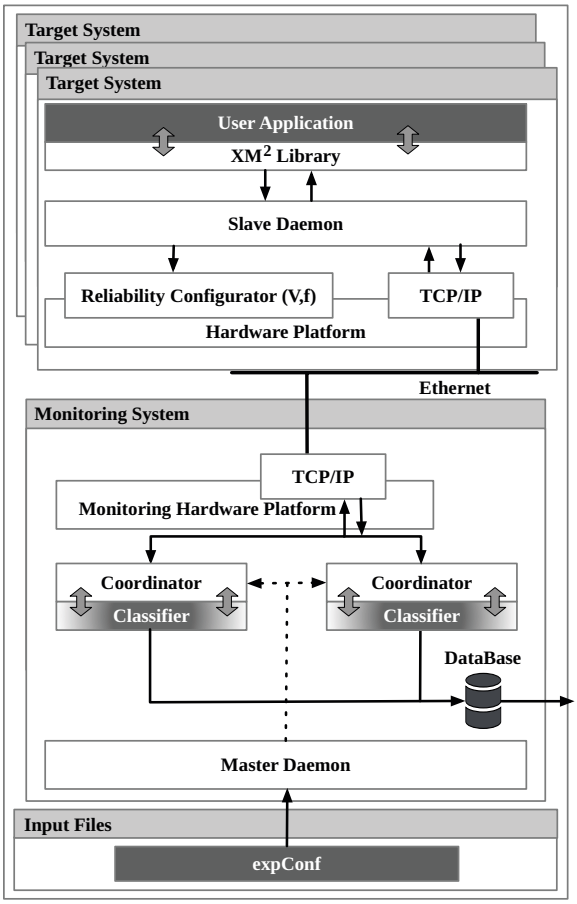

Fig. 2. System architecture of XM^2. It comprises a single *monitoring* system and multiple *target* systems. The components corresponding to dark gray boxes are supplied by the user. XM^2 includes a built-in classifier of results, however the latter can be substituted by a user-provided one.

The TCP functionality is provided either by the OS, or directly by the aforementioned library when running on bare metal.

Remote reset support: When operating at extended configurations, errors leading to full system failure are likely. Therefore, the target system needs to offer a hardware interface for a full/clean reset.

III. Tool Design and Configuration

An experimental characterization campaign on top of unreliable hardware typically involves the execution of multiple experiments under the same configuration (in terms of the underlying hardware, its voltage/frequency configuration, the input set of the application etc). After each experiment terminates, its results are checked and classified, depending on potential effects of faults. This experimental procedure continues until the number of experiments is sufficient to provide statistically significant results.

Figure 2 presents a high level overview of XM^2. It is structured in a distributed way, comprising a single *monitoring* system, and one or more *target* systems of the same hardware architecture. The monitoring system deploys a *Master*

daemon which spawns a *Coordinator* thread for each target system. Every target system spawns a *Slave* daemon, which receives commands from the *Coordinator* and orchestrates the experimental campaign locally, through the library which manages/invokes the application.

A. Configuration File

We employ a configuration file that allows the user to define an experimental campaign by using a single file called *expConf*. The user defines the following parameters:

Target Application: An absolute path to the binary file which will be executed on the target system.

Target System: The Internet Protocol (IP) or Media Access Control (MAC) addresses of the target system(s).

Input File: Input of the application to be executed on the target system(s). XM^2 supports only a single input file per application. If the application requires multiple inputs, they need to be combined into one file by the user.

Operating Configuration: The voltage and frequency settings for the reliable ($Nominal$) and unreliable ($unRel$) configuration of the target system. Applying aggressive overclocking or undervolting settings increases the frequency of errors. Notably, the user can change simultaneously both the frequency and the supply voltage in the $unRel$ configuration.

Result Classification: XM^2 comes with a default classifier, which characterizes the outcome of each experiment as: (i) *Exact:* if the result is identical to that of a nominal execution; (ii) *SDC:* if the result differs from that of a nominal execution; (iii) *Data Abort:* if the CPU raised a data abort trap due to accessing a non existent physical memory address; (iv) *Illegal Instruction:* if the CPU raised a trap because it detected an non-existent opcode; (v) *CPU Crash:* if the execution time exceeds, by far, the time of a nominal execution.

Nominal outputs for the same target may differ, for example in multi-threaded applications which use floating point arithmetic. To be flexible, XM^2 allows users to provide their own classifiers that implement a customized comparison between the golden results and the application output. For example, one can use a deviation threshold to detect erroneous results.

Termination Criteria: The user may define a custom binary which is used by the XM^2 to determine when to terminate a campaign. The XM^2 invokes the user defined binary using as input the number of experiments classified in each category. The default termination checker terminates a campaign simply when reaching a predefined number of experiments, which can be set by the user via the *expConf* file.

Nominal Experiments: The user defines the number of experiments to be performed by XM^2 in *Nominal* setting. These experiments are used to profile the execution time of the application and to obtain the error-free (golden) output files.

B. Run-time Library API

The run-time library that accompanies XM^2 needs to be linked with the target application. It offers an API that enables the application to control data exchange with the *Coordinator* node, as well as hardware switching between the *Nominal* and *unRel* states. Table I lists the primitives of the API.

TABLE I
API TO THE RUN-TIME LIBRARY OF XM^2.

void readInput(void *ptr, size_t sz,size_t nmemb):
Receives *nmemb * sz* bytes from the input file available at the *Coordinator* filesystem and stores them to the memory region pointed to by *ptr*.
void writeOutput(void *ptr,size_t sz,size_t nmemb)
Sends *nmemb * sz* bytes from the memory region pointed to by *ptr* to the *Coordinator*.
void switchToRel()
Switch to $Nominal$ state. The implementation of the function is architecture dependent. It is *blocking* – the *Slave* daemon needs to acknowledge the state switch to the *Coordinator*.
void switchToUnRel()
Switch to $unRel$ state. Similar semantics as *switchToRel*.

C. Example

Listing 1 provides an example of the *expConf* configuration file. The file initially assigns a name to the target system and specifies it using its *IP* and the *MAC* address. In this example there are two target systems (*PC_A, PC_B*). The specified *Nominal* operating point is used by XM^2 to compute error-free outputs (golden) as well as to determine the normal execution time of the application for each target system. The configuration file also includes a list of $unRel$ configurations. A separate experimental campaign will be executed for each of those configurations.

The configuration file also provides the paths to the application binary and the input file. The keyword *Monitor* indicates that these files reside in the monitoring filesystem and need to be fetched over the network. In this example, the user also specifies the *Classifier* binary, which will be used to classify the outcome of each experiment/run. The user specifies the number of experiments (10) to be performed by XM^2 in the *Nominal* configuration to obtain the golden output file. Finally, the user defines the maximum number (500) of experiments to be performed on each $unRel$ configurations.

Listing 2 outlines the modified source code of a mini-application implementing a Sobel filter. Lines 4,5,7 and 8 contain function calls to the run-time library of our tool. Finally, Listing 3 outlines a classifier which categorizes experiments as *Exact, Acceptable, SDC*. *Exact* experiments are those that produce a bit-wise exact copy of the result of the error-free execution. *Acceptable* experiments produce outputs with a *PSNR* higher than 50.0 dB in comparison with a *golden* output. All other experiments are categorized as *SDC*. Note that the classifier is not invoked if the application terminates abruptly due to a runtime error or a crash. In this case, the framework automatically classifies the experiment as *CPU Crash, Illegal Instruction*, or *Data Abort*.

IV. Flow of a Fault Injection Campaign

Figure 3 illustrates a simplified time-line of a fault injection campaign controlled by our tool. Initially, the user provides the *expConf* file to the *Master Daemon*. The daemon creates

```
{
  "TargetSystem":{
  "idName":"PC_A"
    "IP Address":"10....",
    "MAC address": "AA:BB:CC:DD:EE:FF",
    "Nominal": [1.2,1200],
    "unRel": [ [1.2, 1320],[1.2, 1330],[1.0, 1320]]}
  "TargetSystem_":{
  "idName":"PC_B"
   ....}
  "Application":{
    "Path":["pathToExecutable","Monitor"]
    "InputFiles":[["pathToInput",262144, "Monitor"]]
    "Classifier":"/path/to/psnr.exe"
    "Termination" : { "default":{500}},
    "NominalExp" : {10}}
}
```

Listing 1. An example *expConf* file using the *json* format.

```
#include <FIOrchestrator.h>
void sobel(unsigned char* in, unsigned char *out);
int main( int argc, char* argv[]){
  readInput( in, sizeof(char), SIZE);
  switchToUnRel();
  sobel(in,out);
  switchToRel();
  writeOutput( out, sizeof(char), SIZE);
  return 0
}
```

Listing 2. Source code of the application extended with calls to the run-time API.

```
#include <FIOrchestrator.h>
float PSNR(unsigned char* gld, unsigned char *tst);
int main( int argc, char* argv[]){
  float res = PSNR(gold,test);
  if ( isinf(res) & 1)
    printf("Exact inf\n");
  else if ( res > 50.0 )
    printf("Acceptable \%f\n",res);
  else
    printf("SDC \%f\n",res);
  return 0
}
```

Listing 3. Source code of a custom classifier.

a new database and spawns a *Coordinator* thread for each of the target systems. The *Coordinator* connects to the respective *Slave* daemon on the target system using the *TCP* protocol, and transfers the *Nominal* configuration, the application binary and inputs to the target system. The inputs will be used when the target application uses the *readInput* function.

The *Slave* daemon performs the *Nominal* number of experiments as specified in the *expConf* (without transitioning to the $unRel$ state). The purpose of this step is to produce error-free golden outputs as well as to profile the time required to execute the application under nominal conditions. The golden file is used by the classifier for comparison against the outputs of unreliably executed code.

At this point the actual experimental campaigns start. The *Coordinator* sends the configuration parameters of the $unRel$ state to the *Slave*. These parameters will be used for all subsequent experiments. The *Slave* then spawns the application. Any requests to read input data by the application using the XM^2

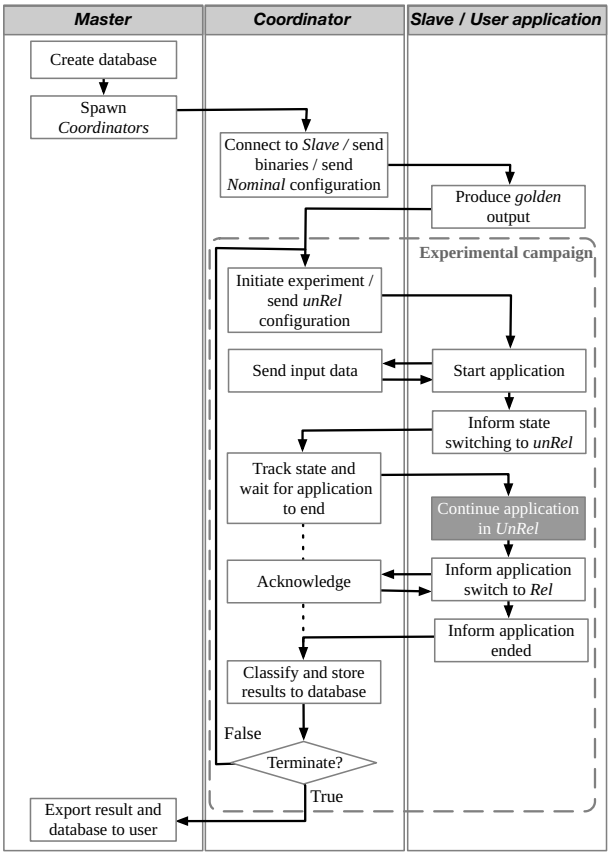

Fig. 3. Flow chart for the main steps performed by XM^2 for the basic case of an experimental campaign that does not result to crashes. The dark box is the only state where the target system is configured at an unreliable state.

API calls are handled transparently. Before a *Slave* transitions to the $unRel$ state it notifies the *Coordinator* first. At this point the *Coordinator* starts a watchdog which waits for the application to terminate. The maximum waiting time is equal to the profiled time when the code was executed reliably, increased by 10%. In case the $unRel$ frequency is lower than the frequency of the *Nominal* point, we proportionally increase the waiting time to match the maximum expected performance degradation due to frequency scaling. If the *Coordinator* does not receive any information about the application status within this period, the *Slave* is reset and the corresponding experiment is flagged as *CPU Crash*.

If the application terminates abruptly, e.g example due to executing an *Illegal Instruction*, the *Slave* informs the *Coordinator* and the experiment is classified accordingly. In case the application terminates normally, the *Slave* sends all output data, as defined by the *writeOutput* call, to the *Coordinator*.

Afterwards, the *Coordinator* invokes the classifier binary to characterize the experiment. If the experiment is not flagged as *Exact*, the *Coordinator* resets the *Slave* so that it is re-initialized to a valid state. The *Coordinator* then evaluates the termination criterion and either terminates the campaign or proceeds to deploy the next experiment. When a campaign

TABLE II
RASPBERRY 3B SPECIFICATIONS

System On Chip	Broadcom BCM2837
Instruction Set	ARMV-8
CPU	4x ARM Cortex-A53, 1.2Ghz
	In Order
	Dual Instruction Decode and execute
	6KB Conditional Predictor
	256 Entry Indirect Predictor
	NEON advanced SIMD
	8 - 64K I-Cache With Parity
	8 - 64K D-Cache With Ecc
RAM	1 GB LPDDR2 (900MHz)

terminates, XM^2 prints the statistics for the different experiment classifications, as well as the path to the output database. In the database, we store the classification of each experiment and a path to the raw output data of executions. If there are more $unRel$ configurations specified in the *expConf*, a new experimental campaign starts, otherwise the tool terminates.

V. EVALUATION

To evaluate our framework we use three *Raspberry PI 3b boards* (Table II) as target systems. We set *Nominal* configuration to $f = 600MHz$, $V_{dd} = 0.8V$. Even though XM^2 supports both undervolting and overclocking, for the evaluation we perform overclocking. We overclock the system by providing a list of $unRel$ states starting from $V = 1.2V, f_u = 1370MHz$ with intermediate steps increasing f_u by $10MHz$, up to the highest frequency state ($V = 1.2V, f_u = 1450MHz$). The termination criterion for the experimental campaign is a number of experiments equal to 2000, which provides a confidence level of 98% and an error margin of 2.5%. For the evaluation, we use the default classifier.

We use *Circle*, a C++ library supporting execution on bare metal, to evaluate the error resiliency of software under unreliable execution without any interference from the OS software stack, e.g. scheduler of Linux kernel or background OS services. *Circle* provides several C++ classes which selectively enable or use different hardware features (*MMU, Interrupt Support* etc.).

The target systems are reset whenever necessary using a small circuit per system, which employs a transistor operating as a switch to connect the respective reset pins on the *Raspberry PI 3b*. The circuits are controlled by the monitoring system through a serial interface.

We evaluate the programmers' effort to use our tools in terms of extra lines of code (LOCs) that are introduced to the source code of an application. Moreover, we quantify the communication overhead introduced by XM^2 between the *Coordinator* and the target system. We use three benchmarks: *Blackscholes [11], Inversek2j, DCT*. *Blackscholes* implements a mathematical model for a market of derivatives, *inversek2j* calculates the angles of a 2-joint arm using the kinematic equation and *DCT* is a module of the JPEG compression and decompression algorithm.

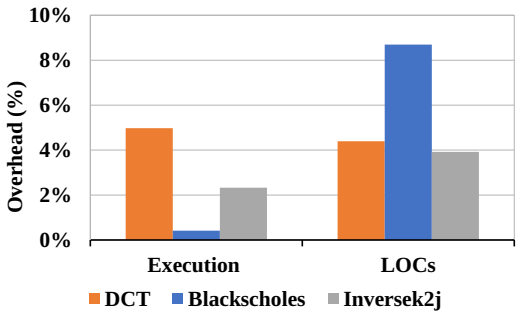

Fig. 4. Overhead of XM^2 in terms of execution time and additional lines of code (LOC) when compared to a native execution and the original version of the code respectively.

Figure 4-left presents the execution time overhead (%) due to the communication protocol and data exchange between the *Coordinator* and the target. XM^2 adds, in the worst case (*DCT*), an extra 5% of execution time compared with a native execution on the target platform under the same configuration. The execution time to compute *DCT* is not negligible, compared to the time needed to transfer the data. Consequently, this benchmark results to the highest overhead. The remaining benchmarks are mainly compute-bound. On average XM^2 introduces an execution time overhead of 2.5%.

Figure 4-right illustrates programmers' effort to prepare an application for our framework. In *Blackscholes*, the developer needs to unpack and pack the input/output data prior to transferring them, thus the volume of the new code is equal to 8.7% of the existing one. The remaining two benchmarks are small in terms of LOCs, so even small code additions produce a large overhead ($\%$). In both cases we simply replace the *fread, fwrite* functions with *readInput, writeOutput* of the XM^2 API. Moreover, we add two extra function calls to switch between states. On average, preparing applications for XM^2 requires 5.6% extra LOCs.

Figure 5 shows the experimental campaign results evaluating the reliability of the system under different overclocked configurations. *Blackscholes* uses double precision arithmetic. Due to the representation of such numbers, faults are un-

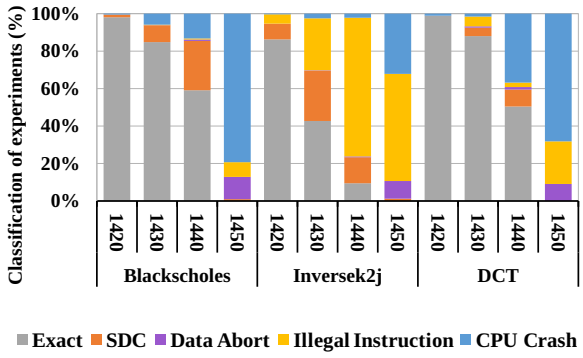

Fig. 5. Experimental results for different applications and different overclocked configurations.

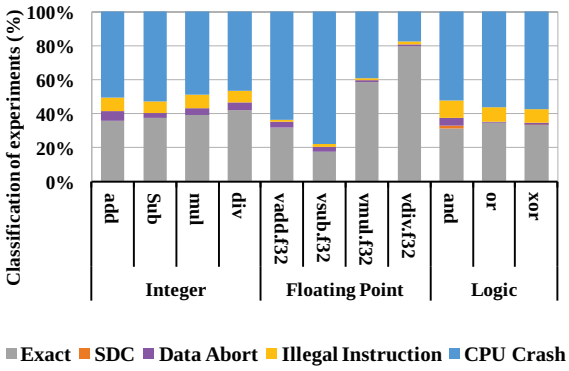

Fig. 6. Experimental results of the instruction error resiliency characterization when $V_u = 1.2V, f_u = 1450MHz$. The X-axis shows the different microkernels and the Y-axis presents the classification of the experiments according to the effects of overclocking on execution.

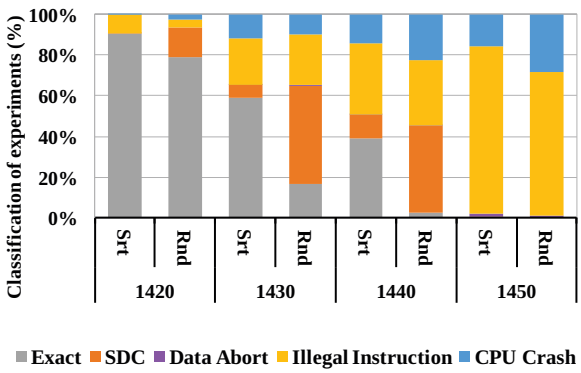

Fig. 7. Experimental results stressing the branch predictor for the two microkernels for different overclocked frequencies (f_u).

likely to be masked. Therefore, this benchmark suffers the highest percentage of *SDCs* (up to 26%) when executed on $f_u = 1440MHz$. *Inversek2j* uses primarily trigonometric functions, which heavily rely on branches. The experiments indicate that faults corrupt the computation of the target address, resulting in decoding memory that does not contain instructions. Consequently, 74% of the experiments result to *Illegal Instructions* when executed at $f_u = 1440MHz$. Finally, 36% of *DCT* experiments result in *CPU Crash* when executed at the same frequency. This benchmark employs six nested loops to iterate through the image pixels and apply the coefficient transformation. Corruptions in the control flow of these loops often results to infinite loops. Therefore, execution is terminated by the watchdog and experiments are classified as *CPU crashes*.

VI. Case Studies

In this section we demonstrate the versatility of XM^2 using three case studies. The first two studies focus on recording and analyzing the behavior of small kernel programs running on the overclocked *Raspberry PI* platform, whereas the third study focus on the behavior of these kernels running on the undervolted Skylake processor.

A. Instruction Level Error Resiliency Analysis

Initially, we employ our tool to assess the resiliency of ARM instruction when executed individually in the overclocked Cortex A53 pipeline causing minimal disruptive events such as cache misses and branch mispredictions (Listing 4). We selected a subset of instructions that perform integer (*add, sub, mul, div,*), floating (*vadd.f32, vsub.f32, vmul.f32, vdiv.f32*), and boolean (*or, and, xor*) arithmetic. The microkernels use only four registers, two as input, one for temporary storage, and one to accumulate the final result which is propagated to the *Coordinator* after the end of the execution.

```
for (){
instruction r0, r1, r2
instruction r3, r3, r0
...
}
```

Listing 4. Template of microkernels used to stress the same execution path of the Pipeline.

In our case, we evaluate the error resiliency of instructions using one input file which sets the register to the value of 1. However the user could define multiple input files and perform multiple experimental campaigns. Finally, the *Nominal* and $unRel$ configurations are the same as in the previous section.

We observe that microkernels which execute multiple times the same instruction, regardless of the instruction, produce exact results even when we increase the CPU frequency by 20% (from $f_u = 1200MHz$ to $f_u = 1440MHz$). When we overclock by an additional $10MHz$ the reliability of most kernels significantly drops as shown in Figure 6. This abrupt fall in reliability confirms previous findings that there are (V, f) settings called *Points of First Failure (PoFF)* at which circuits start to exhibit massive errors.

B. Error Resiliency of Source Code and Algorithm Transformations

In this case study, we demonstrate how XM^2 is used to identify source code transformations with a large effect on application resiliency. We focus on transformation which affect branch prediction mechanisms and the cache hierarchy.

To evaluate the vulnerability of the *branch prediction* mechanism we employ two simple kernels. The *Sorted* kernel traverses an array that contains sorted values and compares each value with the mean of the array. Depending on the outcome of each comparison a variable is increased or decreased. The branch predictor is able to predict correctly the behavior of the branches in the *Sorted* version. The second kernel, called *Random*, traverses the same array, however, as the name implies, the values are stored randomly within the array resulting to a very high misprediction rate (and subsequent pipeline flushes).

Figure 7 shows that the *Random* microkernel has higher percentages of *SDCs* across all frequencies. The two kernels have similar behavior only for extreme overclocking

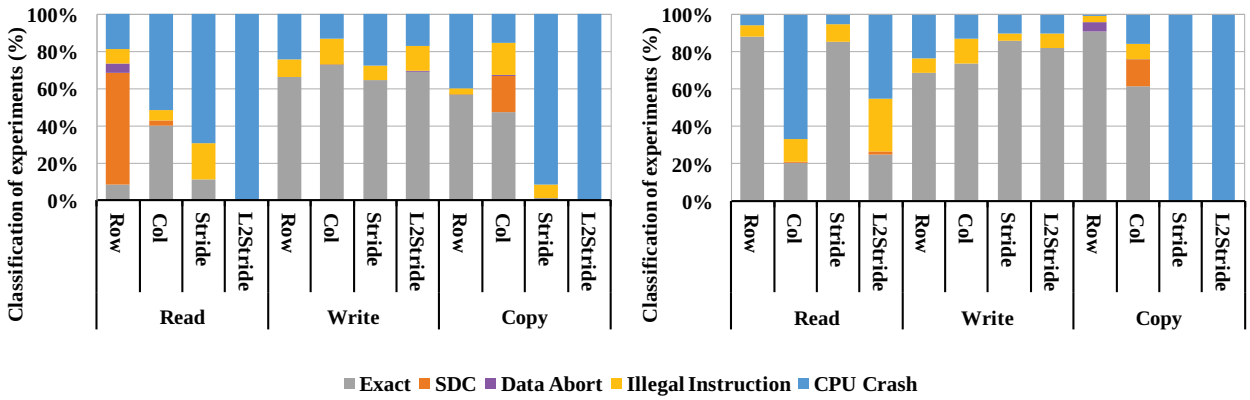

Fig. 8. Experimental results of the *Cache* microkernels of Table III for $unRel = (1.2V, 1430MHz)$, when the hardware prefetcher is enabled (left) and disabled (right). The Y axis presents the classification of the experiments according to the effects of overclocked execution.

$f_h = 1450MHz$, leading to an increased percentage of *Illegal instructions*. We believe that this is due to the high branch misprediction rate which set the Program Counter to memory addresses without valid code or to a memory wrong segment.

For the cache we create kernels which perform *read, write* or *memcopy* using different memory access patterns (Table III). Figure 8 presents results of the cache evaluation. Note that the *write* operation shows a higher degree of robustness in comparison with the rest of the operations. As the CPU automatically enables the read allocate mode during the execution of *write* microkernels, the cache is barely utilized. The *read* operations are slightly more robust than the *memcopy* operations. Actually, the *memcopy* operations usually result in *CPU Crash*, whereas the *read* operations result in *Illegal instructions*. When comparing the different access patterns, the more complex the pattern, the higher the number of experiments which terminate abnormally. In the case of the *L2 Stride*, all experiments result in *CPU Crash*.

We observe that the *Strided patterns* have high *CPU Crash* probabilities. We assume that the prefetcher may have a negative impact on reliability. To validate our assumption we programmatically disable the prefetcher and recompile the cache microkernels. Executing these binaries is trivial since the *expConf* files are the same and only the attribute describing the binary paths should change. The results of the fault injection campaign without the prefetcher are presented on the right side of Figure 8. In general, deactivating the prefetecher increases slightly the application resiliency, however in the strided patterns the results remain the same.

TABLE III
DIFFERENT MEMORY ACCESS PATTERNS USED BY THE SOURCE CODE TRANSFORMATION CASE STUDY.

Row	This pattern accesses bytes in the same order as they are stored in main memory. This is the optimal way to access the memory resulting in lowest *L1*,*L2* cache and *TLB* miss rate.
Col	This pattern accesses the first byte of each memory page, leading to the worst performance of the memory access. Each memory access causes a L1 and L2 cache miss.
Stride	This pattern iterates through the 2D array using a stride equal to the cache line (64 bytes). We disable the hardware prefetcher, so that every memory access leads to a L1 and L2 cache miss. The prefetcher would have been able to detect this strided pattern.
L2Stride	This pattern is similar to *Stride*. We disable the hardware prefetcher, so that every memory access leads to a L1 and an L2 cache hit. Similarly to *Stride*, the prefetcher would be have been able to detect this strided memory access pattern.

C. System voltage Margins characterization

This case study uses XM[2] to identify the voltage margins of four x86-64 systems that feature the Skylake Xeon E3 v5 1220 processor and run Linux Ubuntu OS 16.04LTS. We evaluate and study eight applications (from the PARSEC and SPLASH benchmark suites) and taking full advantage of the customization capabilities of our framework, we provide the appropriate custom classifiers via the *expConf* file. Our goal is to identify the maximum voltage reduction for each application without compromising the correctness of the application.

We employ a *python* script which sets the $unRel$ value of the *expConf* file and searches for the maximum undervolt degree. For each *expConf* we set the maximum number of experiments to be equal to 10. Using a custom termination criteria we terminate the campaign if an experiment does not produce exact output. In such a case the search algorithm moves to lower undervolting values, otherwise it continues to higher undervolting values.

When the execution campaign concludes, we compute the cumulative distribution function (CDF) of the average fault probability for each target system. The results are presented in Figure 9. They show that the four CPUs exhibit a different behavior when it comes to exploiting voltage margins. Note that a more gradual CDF indicates a broader range of margin opportunities depending on the workload characteristics.

VII. RELATED WORK

In CLKSCREW[12], the voltage and frequency scaling capabilities of modern processors are exploited in order to compromise system security, by injecting faults during code execution and extracting cryptographic keys from the ARM

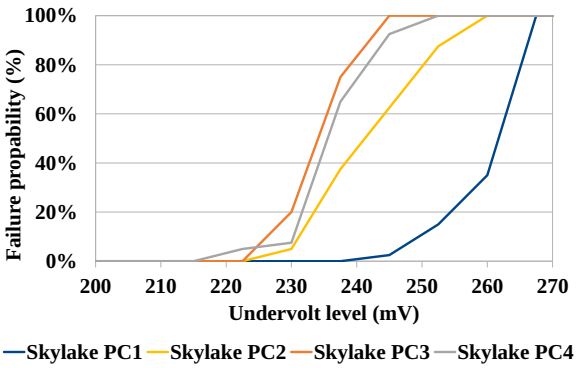

Fig. 9. Voltage margins of the four Skylake CPUs. The X axis represents the amount of undervolting with respect to nominal V_{dd}. The Y axis is the average fault probability CDF for each CPU with respect to undervolting.

TrustZone. This work focuses on the security risks raised by modern energy management techniques. RIFLE [13] and MESSALINE [14] introduce a deterministic and reproducible fault injection technique at the pin-level of a processor. FIAT [15] and FERRARI [16] implement a software-level fault injection which models complex systems with great accuracy however, ensuring that the simulated models are realistic and restraining simulation time are significant challenges. VERIFY [17], MEFISTO [18] and GemFI [19] are fault injection simulators that provide high accuracy in both the location and the timing of the fault, but they introduce significant overhead. Finally, REFINE [20] allows fault injection in the back-end of the LLVM compiler. In contrast to these works, XM^2 performs fault injection natively on the targeted system, therefore it provides native execution time and does not rely on any fault models as they manifest due to real hardware errors.

VIII. Conclusions

This paper introduces XM^2, a software framework which facilitates the experimental evaluation of the effects of voltage/frequency margins on the operation of CPU platforms. XM^2 can be used to study the behavior of software running on platforms operating – potentially unreliably – outside their nominal operational envelope. The results obtained by our tool are accurate and allow evaluation with realistic workloads and working sets. Our framework enables the setup, automatic execution and statistical processing of the results of large experimental campaigns, with minimal configuration effort, limited modifications to applications (on average 5.6% extra LOCs) and limited overhead (on average 2.5%) compared to native, reliable executions. For example, using XM^2, we studied the effects of extended margins on two widely different ISAs. We showed that on an x86-64 Skylake CPU the voltage margins are equal to 220mV, whereas on the ARM cortex A53 the frequency margins are equal to $200Mhz$.

Acknowledgment

This research has been funded from the European Community Horizon 2020 programme under grant no. 688540.

References

[1] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, D. Gizopoulos, P. Lawthers, and S. Das, "Harnessing voltage margins for energy efficiency in multicore CPUs," in *Proc. of the Annual IEEE/ACM Int. Symposium on Microarchitecture, MICRO 2017*, 2017.

[2] S. Achour and M. C. Rinard, "Approximate computation with outlier detection in topaz," in *Proc. of the 2015 International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2015.

[3] D. Blaauw, S. Kalaiselvan, K. Lai, W. Ma, S. Pant, C. Tokunaga, S. Das, and D. M. Bull, "Razor II: in situ error detection and correction for PVT and SER tolerance," in *IEEE Int. Solid-State Circuits Conference, ISSCC , Digest of Technical Papers*, 2008.

[4] K. Parasyris, V. Vassiliadis, C. D. Antonopoulos, S. Lalis, and N. Bellas, "Significance-Aware Program Execution on Unreliable Hardware," *ACM Trans. Archit. Code Optim.*, 2017.

[5] S. Das, D. Roberts, S. Lee, S. Pant, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "A self-tuning dvs processor using delay-error detection and correction," *Solid-State Circuits, IEEE Journal of*, vol. 41, no. 4, 2006.

[6] A. Bacha and R. Teodorescu, "Using ECC Feedback to Guide Voltage Speculation in Low-Voltage Processors," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[7] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi, "Safe limits on Voltage Reduction Efficiency in GPUs: A Direct Measurement Approach," in *2015 48th Annual International Symposium on Microarchitecture (MICRO)*, 2015.

[8] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, C. Magdalinos, and D. Gizopoulos, "Voltage Margins Identification on Commercial x86-64 Multicore Microprocessors," in *2017 IEEE 23rd Int, Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2017.

[9] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. J. Sorin, A. Meixner, A. Biswas, and X. Vera, "Architectures for online error detection and recovery in multicore processors," in *Design, Automation and Test in Europe, DATE 2011*, 2011.

[10] E. A. Burton, G. Schrom, F. Paillet, J. Douglas, W. J. Lambert, K. Radhakrishnan, and M. J. Hill, "FIVR—Fully Integrated Voltage Regulators on 4th Generation Intel® Core ™ SoCs," in *Applied Power Electronics Conference and Exposition (APEC), 2014 Twenty-Ninth Annual IEEE*. IEEE, 2014, pp. 432–439.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proc. of the 17th Int. Conference on Parallel Architectures and Compilation Techniques*, 2008.

[12] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[13] H. Madeira, M. Rela, F. Moreira, and J. G. Silva, "RIFLE: A general purpose pin-level fault injector," in *Proc. of the European Dependable Computing Conference (EDCC)*, 1994.

[14] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Trans. on Software Engineering*, 1990.

[15] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault Injection Experiments Using FIAT," *IEEE Trans. on Computers*, 1990.

[16] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," *IEEE Trans. Comput.*, vol. 44, 1995.

[17] V. Sieh, O. Tschche, and F. Balbach, "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions," in *Proc. of the Symposium on Fault-Tolerant Computing (FTCS)*, 1997.

[18] E. Jenn, J. Arlat, M. Rimn, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool," in *Proc. of the Symposium on Fault-Tolerant Computing (FTCS)*, 1994.

[19] K. Parasyris, G. Tziantzoulis, C. Antonopoulos, and N. Bellas, "Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP Int. Conference on*, 2014.

[20] G. Georgakoudis, I. Laguna, D. S. Nikolopoulos, and M. Schulz, "Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed," in *Proc. of the Int. Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.