Analysis of Execution Path Variability in Kernel Space

Peter Okech
Department of Physics
University of Nairobi
Nairobi, Kenya
Email: pokech@uonbi.ac.ke

Nicholas Mc Guire
OpenTech Research EDV
Bullendorf, Austria
Email: der.herr@hofr.at

William Okelo-Odongo
School of Computing & Informatics
University of Nairobi
Nairobi, Kenya
Email: wokelo@uonbi.ac.ke

***Abstract*—Programs running in contemporary execution platforms displays variabilities in properties such as execution time and execution path in kernel context. The variability is influenced by several factors associated with hardware and operating system. We set to investigate the non-determinism or variability in execution path of programs in kernel space. We focus on the role of processor affinity, the pinning of a thread of execution on a processor. First, we executed a program and let the operating system dynamically place the program in any of the two available cores, and then recorded the kernel traces of the program. Secondly, we statically allocated one of the core to execute the program and similarly recorded the traces of its execution. We then compared the number of kernel functions in the execution traces of the two for practical similarity using statistical test of equivalence. We determined that restricting a thread of execution to a single core without changing other factors does not limit the variability of kernel execution paths. The results obtained allows us to develop and test logical systems based on replicas executing in different cores of a multicore hardware system, in place of true physical multichannel systems.**

I. Introduction

The predominant view held by computer scientists is that sequential programs are deterministic automatons. In this view, a fault-free non-concurrent program will return the same results given the same input. Further, any execution should take the same amount of time to compute its results in the given execution platform and follow the same control flow path each time. Though sequential programs are designed to be deterministic, their correct execution in contemporary hardware/software platform are not.

It has been observed that there is significant variation in the execution time of applications running on contemporary computing platforms under the same conditions. The factors that lead to this variation are either hardware or software related. Modern hardware is becoming more complex mainly due to advanced architectural features such as caches, pipelines, branch prediction, out-of-order execution, and the availability of multiple cores or processors. These features aimed at increasing the average speed of execution makes it hard to predict the timing behavior of a process. Software also influences the execution time variation of an application, for example the operating systems scheduling jitter. In [1], the authors argued that there are distinct parts of the system jitter that are associated with code-paths being executed in system level context due to inherent randomness in complex software systems running on nondeterministic hardware.

In contrast to execution time variability that has been extensively studied [2, 3], the existence of many possible different execution paths for the same program inputs has not been accorded these same attention. The variability in the execution path is more pronounced if you consider the low level instructions executed by the operating system on behalf of a user application.

From the perspective of an operating system such as GNU/Linux, a program executes either in user or kernel context. In user space, the control flow is fully determined by the program's input in the absence of explicit randomization. Thus the code path of an execution can be predicted with high level of certainty. In contrast, when a process is executing in kernel space, the path taken will depend not only on the data passed from the user space but also on the global state of the system. The global state consists of the state of hardware, the state of the operating system and that of all the other processes. The other execution programs and operating system activities alter the hardware state which the process of interest will encounter, while preemption may have an arbitrary effect on the state of the process. The effect of all these is that every execution instance of a program happens in a 'different' execution environment.

In our earlier work [4], we showed that for any non-trivial system call invoked by a program with the same user space input values, there are several possible paths that can be taken by an execution instance. Some of the paths are more frequently taken than others. The execution path variability observed arises in complex hardware/software execution environment such as a system with GNU/Linux running on contemporary super-scalar architecture.

In this paper, we report the results of our analysis of the path non-determinism of a simple program executing in the Linux kernel. There are several factors that can influence execution path variability. Specifically, for the purpose of this paper we chose to focus on one aspect, the effect of thread placement on the variability of execution path. We set out to determine whether statically binding a thread to a specific processor core as opposed to letting the operating system schedule the thread affects the randomness of its execution path. Our approach

involved tracing the path taken in kernel space by the routines of the set of system calls made by an application, using the tool Ftrace [5]. This was first done for the execution of program scheduled (allocated) to run on any of the available cores by the operating system and secondly for the program pinned to a processor core. The trace data was then transfered to a PostgreSQL database for analysis. We then compare execution runs for the two scheduling scenarios to detect any practical differences in the execution paths.

The main contributions of our work are:

- We analyze a self contained code fraction consisting of a set of system calls as a single unit. This is in contrast to the our previous work in which we analyzed single system calls in isolation.
- We have established a method that is easy to reproduce, that can be used to compare the execution paths in the kernel level of an application. We make use of a set of statistical tests that we propose as suitable for such comparisons.
- We have determined that forcing an application to execute on a specific processor core does not noticeably constrain the level of randomness with respect to execution path taken by an execution instance in kernel space.

The rest of this paper is structured as follows. Section II introduces the problem. In section III, we describe the method for gathering and analysis of data and present that result in section IV. We then provide a review of work related to ours in V . Finally, section VI provides a conclusion and gives an indication of the work we intend to do in the future.

II. Problem Description

The execution characteristic of an application program is heavily influenced by its execution environment, consisting of hardware and system software. With advances in computer architecture, computer systems are not the deterministic machines that they were once viewed as. Modern CPUs possess features such as pipelines, branch prediction, several levels of caches, out of order execution, frequency scaling, and multiple cores. These features introduce a certain level of perceived indeterminism. To take advantage of the facilities offered by hardware as well as the need to offer more services, the operating system, the software that manages the hardware are also growing in complexity. This can be seen in the growth in size of the GNU/Linux operating systems from 2 MLOC in 2001 to more than 9 MLOC in 2011 [6]. The complexity of both hardware and system software introduces a certain level of non-determinism in the execution of application programs.

The software architecture of an operating system such as the GNU/Linux can be viewed as consisting of two layers, the kernel space and the user space. Kernel space is the memory area where the kernel code is loaded and executed, and user space the memory area where user processes are loaded and executed. While the user space of a process is isolated from other processes, the kernel space though protected from access by user processes can be considered as a shared area of all processes.

In a typical user application the operation performed by the program include some computation, input and output via external devices, and making use of the services provided by the operating system. To access hardware and other system resources, the application issues a request to the operating system through the system call interface. In a given execution, the user process will invoke several system calls in its interaction with the operating system. While the kernel is servicing these request, it will execute its code on behalf of the requesting process in kernel space. After servicing the request, control is returned to the application to continue its execution in user space.

The control flow in user space is largely determined by the inputs of the application. In the absence of explicit randomization or the use of the results of non-deterministic system calls in branching decisions, the execution path at the user level is highly predictable. On the other hand, if we consider the applications execution in kernel context, we expect that due to the complexity of the hardware/software platform, there would be considerable variability both in the applications execution time and path.

The execution time are influenced by many factors [7], including the state of the processor and that of the kernel. We believe that other execution characteristics such as the execution path taken by an execution instance is also influenced by these same factors. In our investigation, we designed an experiment to determine the influence of processor affinity on the path taken by a program. For this, we executed our program without restricting the core in which the program would execute. The other was to execute the the same program, but have the program execute only on one of the core of the dual core processor available to us. The main aim was to find out if there is a practical significance (equivalence) in the paths taken by executing programs in the two scenarios. Further, we sought to find out if restricting a program to execute in a particular core limits the level of randomness of the execution path.

We are particularly interested in behavior of software systems whose main goal is to monitor and/or control physical processes. Some of these software systems can be modeled as a program having an iterative loop. In the body of the loop, inputs are first read then values computed before they are output. This read-compute-write cycle, typically with a sleep/pause of a given period, is repeated in a non-terminating loop (do while(1);). This looping construct is show in the pseudo-code below.

```
do
    read inputs
    perform computation
    write output
    sleep (n)
while (1);
```

TABLE I
EXPERIMENTAL MACHINE SPECIFICATION

HP Compaq	
Processor	Intel(R) Core 2 Duo E8400
No of Processors	2
CPU Speed	3.00 GHz
RAM	2 GB
OS	Debian 7.0, Linux kernel 3.16.1

TABLE II
EXPERIMENTAL DESIGN

Execution run	Free	Pinned
A	freeA	pinnedA
B	freeB	pinnedB
C	freeC	pinnedC
D	freeD	pinnedD
E	freeE	pinnedE

III. METHOD

In this section, we describe the investigation that we carried out. First, we describe the experimental environment and, secondly describe the steps we undertook during the experiment.

A. Environment

All the programs were executed in an Intel Core 2 Duo machine running GNU/Linux kernel version 3.16.1. The experimental set up is shown in Table I. The kernel was configured for kernel tracing using the FTrace tool, with the following configuration parameters enabled.

CONFIG_FUNCTION_GRAPH_TRACER
CONFIG_STACK_TRACER
CONFIG_DYNAMIC_FTRACE

B. Description of the Experiment

We wrote a program in C to implement the pseudo-code previously described, reading the input from a file on disk, updating the value read and then writing it back to the file. For the purpose of tracing the execution of our program and ensuring that the program terminates, we chose to have an iterative loop of 1,000 iterations. We implemented the task/loop body as a POSIX thread, with the main thread used to simply call the task thread, and terminate the application when the thread exits. The program was compiled using the gcc compiler defaults and the thread library pthread. From the user space, the set of system calls {read, write, lseek, fsync, nanosleep, rt_procmask, rt_sigaction} were invoked repeatedly by the program's task thread. These system calls and the kernel routines they called were the target of our tracing.

We executed the program in an idle system with no load other than background daemons and the operating system.

To collect data, we used trace-cmd [5], a front-end the FTrace tool to record the traces of the kernel functions called by the test program. The resulting data file is a binary file containing events (function entry and exit events) of the kernel routines that were called by the system calls invoked from user space. To aid the analysis task, we first converted the trace data file to text using trace-cmd report facility, then we pre-processed the trace file using shell scripts in order to convert the file to text delimited format suitable for exporting to a PostgreSQL database. We chose to transfer the data to a database to make it easy to perform complex queries and analysis on the data.

For the experiment, we created two experimental groups for the factor under investigation: 1) execution with no thread affinity and 2) execution with thread pinned to a processor core. We refer to the group without the scheduling affinity, i.e. the thread placement is left to the OS, as "free", and the second group in which the thread of execution is forced to run only in one of the processor core as "pinned".

We used the repeated experimental approach in order to reduce experimental error due to uncontrollable factors. We therefore performed 5 repetitions for each of the two groups, identified by the letters A, B, C, D and E. This experimental design approach resulted in a total of 10 execution runs as identified in Table II. In the experiments, each of the 10 execution runs contained a recording of 1,000 traces of function call graphs - our sample size in statistical sampling terminology.

To ensure the independence of each experimental run, we restart the operating system kernel right before loading and executing the test program, making sure that each experiment test starts in a renewed execution environment.

In order to confirm if forcing a thread to execute on one particular core of a multicore machine restricts the level of randomness of the kernel execution path, we need to compare traces of execution runs that are bound to a core against those that are placed by the operating system scheduler. We consider this task as a test on whether the random variables representing the diverse paths in our experiments are drawn from the same distribution. In other words, do the execution paths of the thread dynamically placed by the operating system and those statically bound to a processor core come from the same population or not.

We performed comparisons of the two groups first by using heuristic procedures and secondly through statistical tests. For the heuristics, we plotted several graphs to provide visual information about the differences in the count of the number of kernel routines in execution runs from the two groups. To determine how practically significant the differences noticed were, we used the two one-sided test (TOST) test of equivalence to perform the comparisons.

IV. EXPERIMENTAL RESULTS

We aimed at tracing the execution of a user application in kernel space and analyzing the varying properties of the executing program, focusing on the execution path. Our experiments produced data files containing, for each system call invoked by the user space program, the function call graphs of the kernel routines called by these system calls. We restricted our analysis to the iterative section of the program,

TABLE III
THE NUMBER OF PROCESSOR SWITCHES

Experimental Run	Count
freeA	18
freeB	7
freeC	15
freeD	5
freeE	7

implemented as a separate thread of execution distinct from the main program thread as explained in section III-B.

A. Processor Switch

Based on the POSIX thread implementation, there are two threads in the our test program, the main thread (which we refer to as the process) and the task thread (referred to as the thread) that executes the read-compute-write cycle. The scheduler allocates either of the processor cores (P0 or P1) to the process when it starts. When the thread start executing (the call to pthread_create()), it is either scheduled to run on the same CPU core as the process or assigned to the other CPU core. When the thread terminates (the call to pthread_join()), the process continue to execute on any of the cores assigned to it at that time. Table III show the number of processor switches of the thread for the first experimental group - the "free" group. The number of processor switches ranged from a low of 5 to a high of 18.

In the second experimental group with thread pinning, three of the execution runs namely pinnedA, pinnedD and pinnedE had the process allocated core P1, and so the thread continued to execute in the same processor. As for pinnedB and pinnedC, the process executes in P0 while the thread between its creation and exit was executed in core P1.

B. Length of the Execution Path

The path taken by an execution instance can be described by the count of the number of kernel functions called during the execution instance. We refer to this as the length of the execution path. The data plots showing the length of the path for execution runs freeA (lower figure) and pinnedA (upper figure) are shown in Figure 1. These figures clearly shows that there is a differences in the data collected from the two groups.

Within each of the experimental group, we were able to visual confirm that in the experimental runs there were observable differences as expected due to nondeterminism in the execution platform. These differences are depicted in Figure 2 for the "free" experimental group and Figure 3 for the "Pinned" group. For the sake of readability, we present only two runs in each figure out the pool of the 5 experimental runs.

C. Comparing the Experimental Distributions

We also created histograms from the data set. Figure 4 show the corresponding histograms of the data from the experimental runs freeA (lower figure) and pinnedA (upper figure).

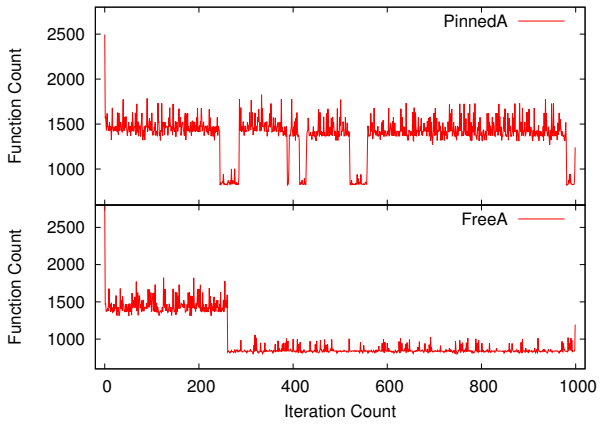

Fig. 1. Plot of Length of Execution Paths. The lower figure shows the graph of the number of functions for the dynamic processor core allocation while the upper figure the graph for static allocation

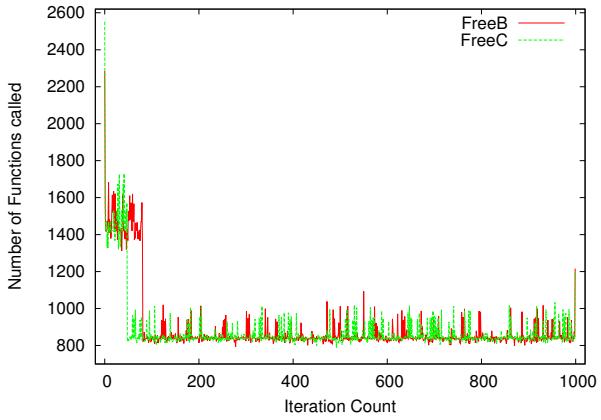

Fig. 2. Comparison of Lengths of Paths (Free) for the Execution Runs freeB and freeC

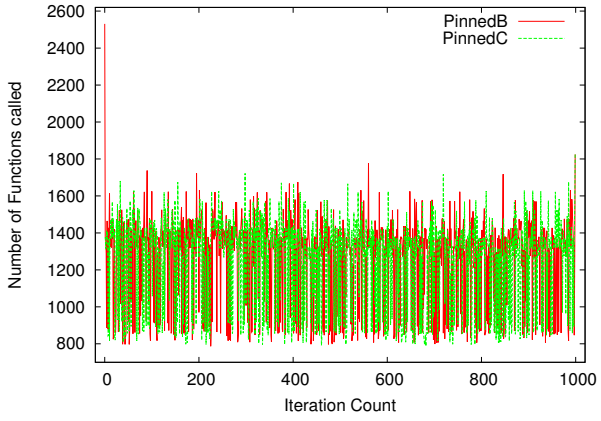

Fig. 3. Comparison of Lengths of Paths (Pinned) for the Execution Runs pinnedB and pinnedC

Further, to get an intuitive feeling of the differences in the probability density distribution of the length of the execution paths for the groups "free" and "pinned", we generated kernel density plots for the respective experimental runs. For clarity,

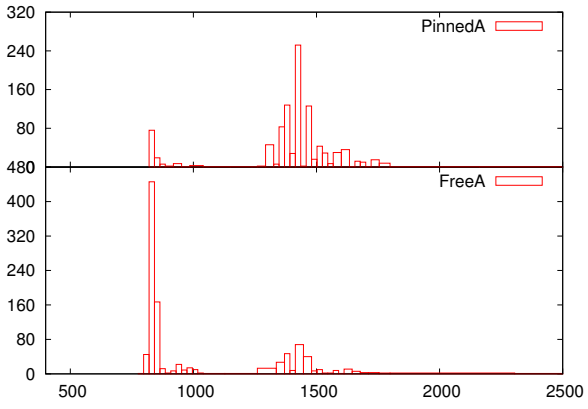

Fig. 4. The Occurrence Frequency Plot

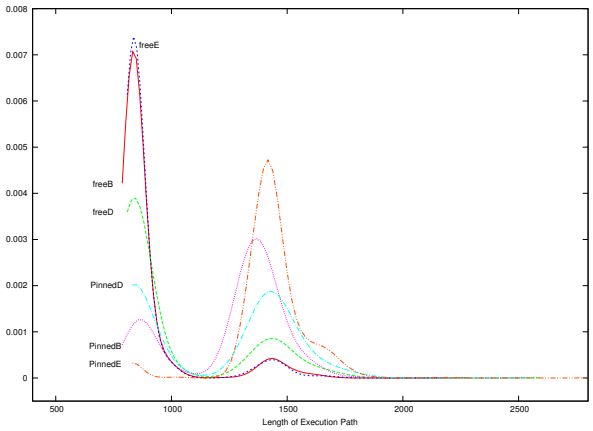

Fig. 5. The Kernel Density Plots for some of the Experimental Runs

we present distributions for some of the execution runs, given in Figure 5. The histograms of Figure 4 and the kernel density estimates of Figure 5 show distributions with having two local maxima, with the major peaks to the left and minor peaks to the right for experimental runs from the "free" group. The position of the major and minor peaks for the "pinned" group is the reverse of that of the "free" group.

To provide a different perspective on the data, we calculated the quartiles of the length of execution paths for each of the experimental runs. We then generated a graph showing the variations using a box plot. The plot is given in Figure 6. The plots show that the data from the "pinned" group are skewed to the right and are more likely to be dispersed than those that come from the "free" group.

D. Test of Equivalence

As evident from the results given in the preceding section, we were able to confirm that there exists differences in the execution path distribution of the two groups. Our next step was to use statistical tests to determine the significance of these differences. Due to the characteristics of our data set, we believe it is appropriate to use non-parametric statistical tests instead of the parametric counterparts. Based on intuition and

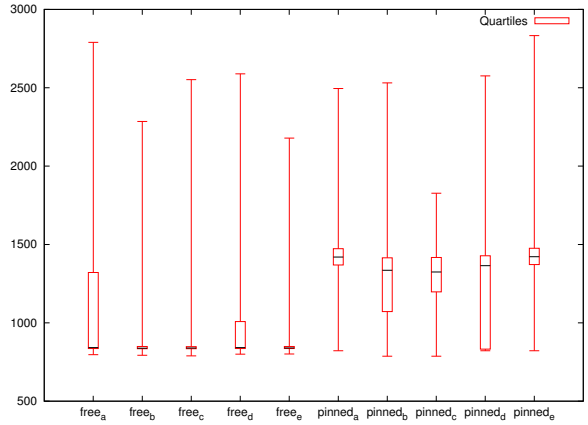

Fig. 6. The Box Plot of Execution Path Lengths for All Experimental Runs

previous work on using thread affinity to make execution time more predictable, we sought to determine if the execution path variability is reduced when a thread of execution is pinned to a processor core.

We formulated our question as a test of equivalence of the execution paths between the paths generated by program execution when the operating system is free to allocate an available processor core and the execution paths whereby the program's thread of execution is bound to a particular core. To determine the practical significance, we chose to use the confidence interval approach. We set the margin of similarity of the execution path lengths between the groups at 393 functions, that is if the length of the paths of any two samples differ by a count of less than +/-393 kernel functions, then they are for practical purposes considered similar. This value was chosen after computation based on the difference in the average length of the path, the individual and, pooled standard deviations for the first execution runs at 95% confidence level. The value falls within the 20% to 40% range if we use a rule of the thumb decision.

We performed equivalence testing of unpaired samples using the robust TOST in the statistical package R. We selected the second execution run in the "free" group, freeB (run with the lowest median), and compared it with other execution runs. Out of all the comparisons, only three of the test results are presented Table IV. The differences in the sample means of the execution runs freeB and freeA, at the 95% confidence interval is within the margins of similarity [-393, +393] so we can claim equivalence. This is expected since the two runs are from the same group. We cannot claim equivalence for the difference at the same confidence level for the execution runs pinnedC and pinnedE. It is however possible to claim statistical non-inferiority, thus dismissing the assertion that pinning the thread affinity strategy used in this experiment would result in reduced variability in the execution path of a program in kernel space.

TABLE IV
COMPARISON OF SELECTED EXECUTION RUNS AGAINST THE RUN FREEB

	freeA	pinnedC	pinnedE
Difference in Mean	-69.12	-472.55	-583.96
95% Confidence Interval			
Lower Limit	-89.098	-489.57	-588.85
Upper Limit	-49.13	-455.53	-579.07
Falls in Margin of Similarity?	Yes	No	No
Claim	Equivalent	Not Equivalent	Not Equivalent

V. RELATED WORK

In our work, we trace kernel routines/functions called by the system calls invoked from user space to identify the execution path of a user application in kernel context. Execution tracing has been applied in different scenarios, including comprehension of operating system behavior [8]. In contrast to our work, the authors in [8] focus on the task of understanding the operating system kernel for the purposes of program maintenance.

There have been several work that examine the runtime variability of applications. The authors in [2] have done an experimental study on the variation of execution times in repeated program runs and particularly the influence of operating system jitter on the variability. They compare the various samples obtained in their experiments using different test protocols which apply different statistical techniques. Similarly, the authors of [3] performed a study on the variability of execution time in multicore architectures. They evaluated the effect of thread migration strategies on execution time variation.

These two works are similar to ours in two ways. First, they all investigate the variable characteristic of an executing application. The difference is that while they focus on execution time, we look at that of the execution path as defined by the kernel routines called during execution. Secondly, they identify and investigate the factors that influence the variability, OS jitter and thread affinity strategies respectively. Currently our work do not consider the effect of the operating system but investigates the influence of thread affinity on execution path variability. In contrast to the affinity strategies described in [3] where the executing thread has exclusive access to the core (no other thread is allowed to execute on that processor core), we implement the strategy whereby the thread is restricted to only a particular core that it can share with other threads.

VI. CONCLUSION AND FUTURE WORK

In this paper, the experiments that were performed and the analysis of the data collected to determine the effect of binding a thread of execution to one of the core of a multicore processor on execution path variability in kernel space are described. The work described is achieved through recording traces of the kernel functions executed by the operating system on behalf of the user application, when 1) the operating system dynamically allocates the program to an available core and 2) the programmer statically binds the program to a particular core. To reduce the effect of errors, each experiment was repeated five times.

The analysis show, based on the sample statistics of data collected from the execution runs, that the probability distributions that describe the execution paths from the two experiments are different. The results also show that there is no ambiguity in the differences in the two experimental groups, and thus a claim of non-inferiority is valid. The conclusion derived is that restricting an application to run in one of the core of a dual core machine does not reduce the level of execution path variability.

The achieved results provides the opportunity to build logical multi-channel systems on a multi-core machine. The simple case would be a system in which two replicas of the same simple non-diverse user application each pinned to a particular core of a dual-core hardware architecture system running in parallel. Such a setup simulates a two channel system that will makes it possible to test if path non-determinism in a redundant architecture can result in runtime diversity of the replicas.

With respect to the assessment of indeterminism of kernel level execution path, there is a plan to extend this work in two main ways. First, analyze the effect of system load on path non-determinism. Secondly, investigate how changing the values of kernel parameters such as resource limits would affect the randomness of the execution path.

REFERENCES

[1] N. Mc Guire, P. Okech, and G. Schiesser, "Analysis of inherent randomness of the linux kernel," in *Proc. 11th Real-Time Linux Workshop*, 2009.

[2] P. E. Nogueira, R. Matias Jr, and E. Vicente, "An experimental study on execution time variation in computer experiments," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1529–1534.

[3] A. Mazouz, D. Barthou *et al.*, "Measuring and analysing the variations of program execution times on multicore platforms: Case study," 2010.

[4] P. Okech, N. Mc Guire, and C. Fetzer, "Utilizing inherent diversity in complex software systems," in *Proceedings of The Australian System Safety Conference (ASSC2014), Australian Computer Society, Inc.*, 2014.

[5] S. Rostedt, "The World of Ftrace," Linux Foundation Collaboration Summit, Apr. 2009.

[6] N. Palix, G. Thomas, S. Saha, C. Calves, G. Muller, and J. Lawall, "Faults in linux 2.6," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 4, 2014.

[7] P. Puschner and M. Schoeberl, "On composable system timing, task timing, and wcet analysis," in *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2008.

[8] E. Vicente, G. Dyany, R. Matias, and M. de Almeida Maia, "Improving program comprehension in operating system kernels with execution trace information." in *SEKE*, 2012, pp. 174–179.