The final version of this paper appears in 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)

GeST: An Automatic Framework For Generating CPU Stress-Tests

Zacharias Hadjilambrou
University of Cyprus
zhadji01@cs.ucy.ac.cy

Shidhartha Das
Arm
Shidhartha.Das@arm.com

Paul N Whatmough
Arm / Harvard University
Paul.whatmough@arm.com

David Bull
Arm
dbull@arm.com

Yiannakis Sazeides
University of Cyprus
yanos@cs.ucy.ac.cy

Abstract*—This work presents GeST (Generator for Stress-Tests): a framework for automatically generating CPU stress-tests. The framework is based on genetic algorithm search and can be used to maximize different target CPU metrics such as power, temperature, instructions executed per cycle and dI/dt voltage noise. We demonstrate the generality and effectiveness of the framework by generating various workloads that stress the CPU power, thermal and voltage margins more than both conventional benchmarks and manually written stress-tests. The key framework strengths are its extensibility and flexibility. The user can specify custom measurement and fitness functions as well as the CPU instructions that will be used in the genetic algorithm search. The paper demonstrates the framework prowess by using it with simple and complex fitness functions to generate stress-tests: a) for various platform types ranging from low-power mobile ARM CPUs to high-power x86 CPUs and b) with different measurement instruments such as oscilloscopes and software accessible performance counters and sensors.

Keywords: tool, framework, stress-test, virus, power, temperature, dI/dt, genetic algorithms

I. Introduction

Stress-tests that maximize micro-architectural activity, heat-dissipation, power-consumption and voltage-noise are useful for numerous reasons that include among other: a) benchmarking, b) testing system stability, c) margining production systems, d) detecting performance bottlenecks or weaknesses in the CPU power delivery network (PDN), and e) testing the efficacy of energy-efficiency techniques such as voltage-noise mitigation mechanisms [1][2][3][4][5][6][13]. Manually crafting such stress-tests is a time consuming and tedious procedure. Consequently, previous work has proposed automated frameworks for generating stress-tests [1][2][3][4][5][6][7][8]. The basis of most of these approaches is genetic algorithm search (GA) [9].

Despite the previous work on GA based stress-test generation frameworks, to the best of our knowledge, there is no publicly available tool for researchers and practitioners for automatic stress-test generation. This work presents GeST (Generator for Stress-Tests): a GA based framework that researchers and designers can use for automatic stress-test generation. GeST has already been used for industrial purposes and in several research publications [22][23][24][25].

GeST, given a user-specified set of assembly instructions and operands, attempts to find the instruction mix, order and operands that maximize a target metric. GeST is extensible as it offers an easy interface to build upon. A user can define the instructions, which the optimization search uses, by only changing input configuration parameters. This renders the framework compatible with any ISA. Moreover, an experimenter can script custom measurement procedures and custom fitness functions (the function that drives the GA optimization) in a plug-and-play fashion using the template measurement and fitness software classes provided in the framework. The user defined measurement scripts and fitness functions are easy to integrate in the framework by simply changing the configuration parameters without performing any change in the framework's core source code. We demonstrate the power of the framework's extensibility and flexibility by: a) generating stress-tests that maximize different target metrics such as power, temperature, and dI/dt voltage-noise, b) using the framework with various measurement procedures and optimization metrics such as software accessible counters (e.g. performance counters) and external instruments (such as oscilloscopes), c) generating stress-tests on mobile ARM and server-grade ARM and x86 CPUs, d) generating stress-tests on bare-metal and OS execution environments, and e) using both simple as well as complex multi-objective fitness functions.

The rest of the paper is organized as follows: Section II provides background discussion on stress-tests, Section III presents GeST, Section IV discusses the experimental setup, Section V demonstrates the capability of GeST to generate stress-tests that maximize power consumption, temperature and IPC. In Section VI, we highlight the framework's capability to generate dI/dt voltage noise stress-tests. Section VII discusses previously proposed GA frameworks. Finally, we present concluding remarks in Section VIII.

II. Background On Stress-Tests

For the purposes of this discussion, we classify stress-tests into three categories: a) stress-tests that maximize specific micro-architectural (uArch) metrics, such as memory bandwidth, IPC and cache-misses, b) stress-tests that maximize power consumption and temperature, commonly referred to as "power-viruses", and c) stress-tests that maximize voltage noise, also known as "voltage-noise viruses" or "dI/dt-viruses". This work shows that GeST can successfully generate stress-tests for all three categories. In particular, we use the framework to generate stress-tests that maximize CPU IPC, power, temperature and dI/dt voltage-noise. While this work focuses on the CPU there is nothing fundamental that prevents using GeST for other processor components as well, for instance the last level cache (LLC) or an integrated accelerator. A brief discussion on each of the three stress-tests categories follows.

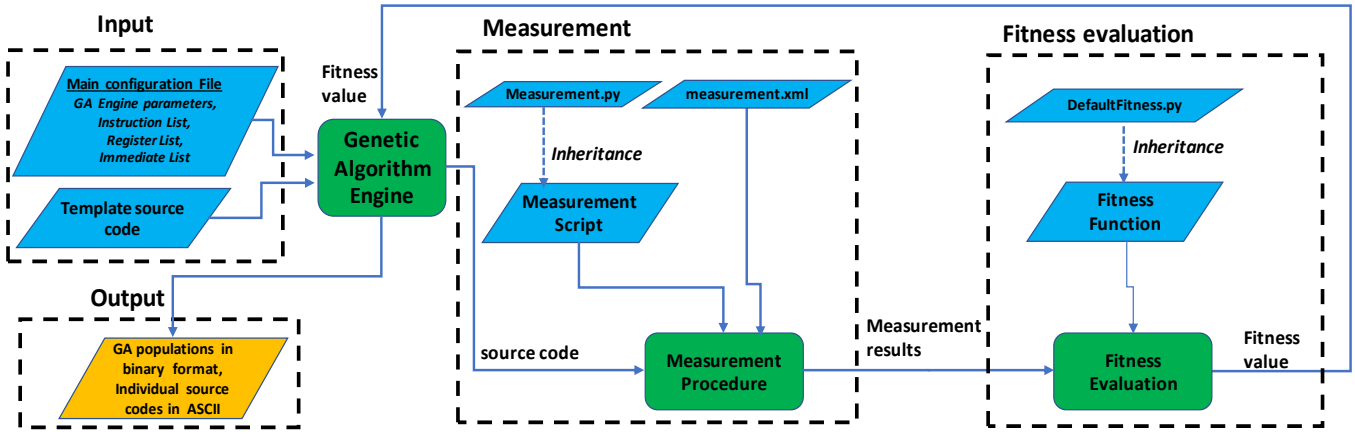

Figure 1. Framework overview.

Stress-tests for maximizing specific uArch metrics are mainly useful for performance benchmarking purposes. The AIDA test suite [15] is a good example of benchmarking stress-test software that is commonly used to test desktop and mobile system's performance. This suite includes various benchmarks to test the performance of specific CPU units (e.g. floating-point unit) and specific functions (e.g. hashing). It also includes memory latency and read/write bandwidth tests as well as specific test benchmarks for GPUs and disks. Besides performance testing, previous work has proposed using stress-tests that target specific CPU parts (ALU, FPU, L1D, L1I, L2 and L3 caches) to characterize the CPU minimum operation voltage (V_{MIN}) [14] and to generate power-models and an energy-per-instruction (EPI) profile [8].

Power-viruses maximize both sustained power consumption and heat-dissipation [5]. They are useful for characterizing a system's power and thermal margins as well as the IR drop [10][11]. In addition, they can check thermal stability, in particular, of overclocked systems (set to run at a higher than nominal voltage and frequency). Power-viruses usually maximize the micro-architectural activity by issuing many instructions per cycle [4]. Prime95 [16] is a well-known test program that maximizes power consumption and it is often used to check the stability of over-clocked CPUs.

Voltage-noise viruses attempt to maximize CPU voltage fluctuations [1][2] and they have different characteristics from power-viruses. Rather than keeping a sustained high current (I) consumption, dI/dt stress-tests attempt to cause sudden transition from very low to very high current consumption. Abrupt current increase causes the voltage to drop low. Periodic current surges that match the CPU's PDN 1st order resonance-frequency maximize the CPU voltage droops and overshoots [1][2][12]. Since low voltage operation may lead to malfunctioning [1][12], dI/dt viruses are very effective timing-error stability-tests. Voltage-noise viruses typically cause higher voltage drop than power-viruses because the dI/dt component dominates over the IR drop. The lowest voltage at which a dI/dt virus runs correctly can provide a good indication of where to set the operating voltage of the CPU (for a given operating frequency).

Typically, a dI/dt virus is a loop of assembly instructions fine-tuned to cause current variations at a rate equal to the PDN's 1st order resonance-frequency. To develop dI/dt viruses high bandwidth voltage measurements are required to measure the maximum voltage droop or the maximum peak-to-peak voltage swing. This is achieved either through external oscilloscope connected to on-package voltage sense points [1] or internal on-chip voltage sensors [12].

III. GeST Framework Description

GeST is written in Python 3 and takes as inputs xml files that define configuration parameters. The framework high-level overview is shown in Figure 1. The framework can be broken down into 5 major parts: the inputs, the outputs, the GA engine, the measurement component and the fitness evaluation function. Next, we describe in detail each of these components.

A. Genetic Algorithm (GA) Engine

The GA engine is the heart of the GeST framework and coordinates its execution. GAs optimize a target metric by applying operators inspired by natural evolution such as selection of fittest individual for breeding, exchange of genes (crossover) and mutation. Previous work has shown that GAs can generate workloads that stress the system worse or comparably to manually written stress-tests with little human guidance within few hours [1][3][4][5][6]. Our findings clearly confirm the GA suitability and effectiveness for stress-test generation. A typical GA flow is shown in Figure 2. A short description of each GA step follows:

- **Seed Population:** The first step is to create an initial seed population (generation). The population is a set of assembly instruction sequences. In GA terminology, each sequence of assembly instructions represents an **individual** of the population. The seed population can be either a new random initial population or a population from a previous GA run. In the case of a random initial population the individuals are randomly

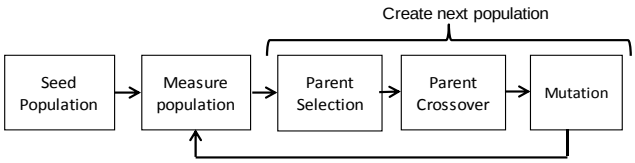

Figure 2. A typical GA flow.

generated based on the user-specified instructions, operands and loop-size.

- **Measure Individuals:** The second step involves compiling each individual, executing the resulting binary, measuring the metrics of interest during the binary execution and assigning a fitness value to the individual. In GeST the user defines the measurement procedure and fitness function as shown in Figure 1.
- **Creating next generation:** The algorithm creates a new population after all individuals are measured. The new population is created by selecting the fittest individuals as parents (e.g. the ones that scored the highest average power), exchanging instructions between the two parents (crossover) and performing mutation. A mutation operation converts an instruction or an instruction-operand (such as a register) into another, with a conversion probability, referred to as the "mutation rate". For instance, if the mutation rate is equal to 2%, then each instruction has a 2% probability to be mutated.

Figure 3 demonstrates, with the help of an example, how we generate a new population by applying tournament selection, one-point crossover and mutation operators. The procedure demonstrated in the figure is performed repeatedly until the desired population size is reached. Note that for this example each individual consists of only four instructions. First, we randomly pick five individuals from the population and select from them as "parent1" the fittest individual. The same procedure is applied to select "parent2". Then a random point in the instruction stream is selected for the crossover between the two parents. In the example the crossover point is the 2nd instruction. This means that the "child1" will inherit the first half from the "parent1" and the second half from "parent2", while "child2" will inherit the first part from "parent2" and the second part from "parent1". Finally, the example demonstrates the mutation operator. Mutation can be performed for a whole instruction i.e. the whole instruction is randomly transformed to a new instruction, or an operand of the instruction i.e. an operand is transformed to another operand. For "child1" the r2 register of the SUB instruction transforms to r5, while for "child2" the STR instruction transforms to LSL and the LSL operands are randomly generated.

Table I shows the GA related configuration parameters and their values that we empirically found to work well in our experiments. A key observation from our work is that relatively few instructions are sufficient to stress the CPU. Loop lengths of 50 instructions prove sufficient to cause large power consumption or high IPC. Voltage noise optimization is more sensitive to loop-length because the dI/dt noise is highly

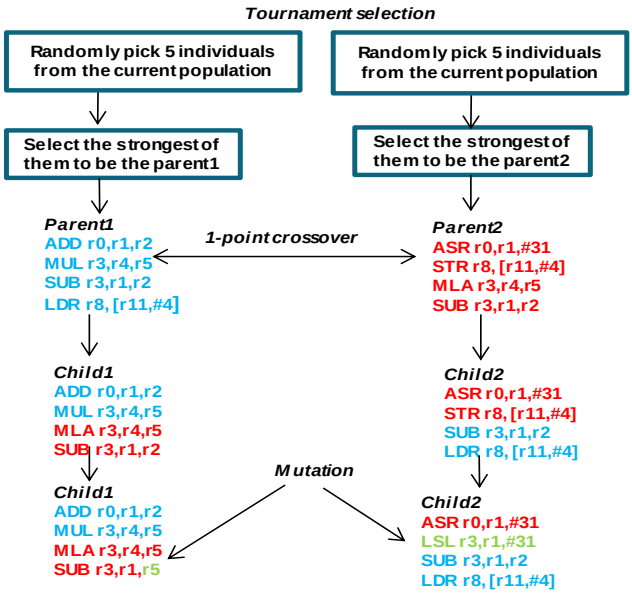

Figure 3. Demonstration of GA operators.

Table I. GA parameters.

Parameter	Default Values
population_size	50
Individual Size (number of loop instructions)	15-50
mutation_rate	0.02 - 0.08
crossover_operator	one point crossover
elitism (Best individual promoted to next generation)	TRUE
parent_selection_method	Tournament Selection
tournament_size	5

related to the PDN resonance frequency. A rule of thumb that is found to work well for dI/dt noise is to have the loop instruction length equal to IPC x clock_frequency / resonance_frequency (similar to what authors used in [3]). The IPC should be roughly equal to MAX_THEORETICAL_IPC / 2 (the rationale behind this is that dI/dt should contain low and fast activity phases hence the IPC should be somewhere in the middle). In our experience, the aforementioned equation typically results in loop lengths of 15 to 50 instructions. Another recommendation, supported from experimental findings, is that mutation rate should be low enough so that only one or at-most two loop instructions are mutated at a time. Higher mutation rate might impede the GA convergence. So, if the target is one mutated instruction, then for loop lengths of 50 instructions we need 2% mutation rate, for 15 instructions we need 8%. Finally, we have found that optimization search converges faster if children preserve some of the instruction order found in their parents (this is especially true for maximum power and maximum dI/dt search) [8]. Hence, to accelerate the GA convergence we prefer one-point crossover that does a better job in preserving the instruction-order of strong individuals compared to uniform-crossover (another well-known crossover operator), where each instruction has an equal probability to be swapped among the parents.

```
<operand                          <operand
 id="mem_result"                    id="mem_address_register"
 values="x2 x3 x4"                  values="x10"
 type="register" >                  type="register“>
</operand>                        </operand>

                                  <instruction
<operand                            name="LDR"
 id=“immediate_value"               num_of_operands="3
 min="0"                            operand1=“mem_result"
 max="256"                          operand2="mem_address_register"
 stride="8"                         operand3=“immediate_value“
 type=“immediate"                   "format="LDR op1,[op2,#op3]"
>                                    type="mem"
  </operand>                      > </instruction>
```

Figure 4. Example of an instruction definition and its necessary operands.

B. GeST Inputs

The GeST inputs consists of the main configuration file and the template source code. We describe in detail the format and use of these files.

1) Main Configuration File

The main configuration file is a xml file that specifies: a) the GA engine related input parameters shown in Table I, b) the instructions and operands used in the GA search, and c) various other parameters, such as, the directory where the results will be saved and the names of the measurement and fitness classes to be used by the GA search. GA engine related configuration parameters (individual size, mutation rate etc.) are explained in the previous subsection. The following discussion focuses on how to specify the instructions and operands used by the GA optimization search.

The registers, immediate values and instructions used by the optimization are defined in the main configuration file. Figure 4 shows an example of how a user can define an instruction and its required operands. The instruction in the example is the ARM ISA LDR (load from memory). The first required parameter is the instruction name, it is used to identify the instruction and must be unique. The second parameter is the number of instruction operands. LDR has 3 operands: a) the register where the result will be written, b) the register that holds the base memory address, and c) an immediate value that holds the memory offset. These operands must be separately defined in the same configuration file (also shown in the figure). The operand definition is described in detail in the next paragraph. The instruction definition links to the operand definitions through the operand ids. In our example, the third to fifth instruction-definition parameters define the operand ids which are “mem_result”, “mem_address_register” and “immediate_value”. If the instruction definition contains an undefined operand id, the framework will terminate the execution. In addition, if the instruction definition contains incompatible to the ISA specification operands, then generated instructions sequences that contain this instruction will fail to compile. Continuing with the instruction definition parameters, the “format” parameter specifies the instruction format. It prescribes to the framework how the instruction must be printed in the generated output source code. The op1, op2 and op3 keywords in the format specification will be replaced by the corresponding operands. Finally, an instruction type is specified, that is useful for various reasons. For example, it allows analyzing the instruction breakdown of the generated stress-tests in terms of integer, float, SIMD, memory and branch instructions. It is worth noting that through the same instruction specification interface the experimenter can specify both individual-instructions as well as whole instructions sequences that will be atomically included in the GA optimization search.

Regarding operand definitions, both register operands and immediate operands require their potential values to be specified. For register type operands the values are specified through the “values” parameter that accepts space separated register names. In Figure 4, the user has specified that the LDR result register can be anyone of the x2, x3 or x4 registers. Regarding immediate operands, the potential values of an immediate are expressed through maximum, minimum and stride parameters. In the example the user allows the immediate value to take 33 different values, from 0 to 256 in strides of 8 i.e. 0,8,16,24…256. Essentially, in this example there are 99 possible ways the GA can use the LDR instruction (3 registers for memory result x 1 memory address register x 33 immediate values). The GA randomly generates any one of the 99 possible forms when generating the initial random population and when performing the mutation operation. As the search is progressing, the GA will converge to the instruction variation that maximizes the target metric. If none of the evaluated instruction’s possible variations helps to maximize the fitness value, then the instruction will likely stop appearing in the GA generated source codes. For instance, consider a long-latency instruction like integer division (DIV) used in an IPC maximization search. After, few generations the DIV instruction will most probably be eliminated from the individuals because it does not contribute in generating fit populations.

An operand definition, if desired, can be common for multiple instructions. For instance, the “mem_address_register” and “immediate_value” can be used by other memory instructions that the user may want to define, such as for the ARM ISA instructions LDP, STR, STP. The instruction and operand specification interface can serve one more purpose: as the means to force or explicitly avoid instruction dependencies. For instance, if optimizing for maximum instructions per cycle (IPC) it may be undesirable to have short-latency integer instructions depending on memory loads. Thereby, to avoid integer instructions depending on memory loads the user can specify two disjoint sets of integer register operands, one for memory destinations and one for source operands of all other integer operations.

2) Template source code

The GA uses the instruction and operand definitions to generate individuals during the optimization search. These individuals are printed inside a template source code file (the location of the template file is provided in the main configuration file by the experimenter) that will be eventually compiled in a binary. The template source file must contain an empty loop body that is filled with the GA generated

individuals. To indicate where the individual will be printed, the string "#loop_code" must be written within the empty loop body. Before compiling an individual, the framework removes the "#loop_code" string and prints the instruction sequence starting from the indicated line. Within the template file the user can also specify some fixed code that can be part of the loop body across all individuals e.g. add NOP instructions for padding. The template source file may also include user specified initialization code that contains register and memory initialization. We find that register values have considerable effect on power consumption, so they must be initialized judiciously. For this work, we have use checkerboard patterns (e.g. 0xAAAAAAAA) since they increase bit switching that helps in maximizing power or dI/dt voltage-noise.

It is worth mentioning that while this work performs GA searches at assembly programming level, the instruction definition interface and the template source file can be also used to perform optimization at a higher-level language (e.g. at a C code level).

C. Measurement and Fitness Evaluation

Each source code is compiled to a binary and measured on the target machine (the GA framework if needed can run on a separate workstation). This procedure typically involves transferring the source file to the target machine, compiling the binary on the machine, running the binary, measuring the metric of interest through a measurement instrument (such as multimeter, oscilloscope or software accessible counters) while the binary is running, and, finally, stopping the binary execution and calculating the fitness value based on the measurements. An abstract Python class, refer to as the "Measurement.py", provides the template for scripting such measurement procedures. Moreover, the class contains various utility functions that can be useful for scripting these procedures. For instance, the class contains functions for communicating through ssh with the target machine such as copying files over scp and executing any arbitrary command. To create a custom measurement script the user must inherit the Measurement.py class and overwrite the "init" and "measure" functions. The "init" should contain specific to the measurement procedure parameter initializations (e.g. number of active CPU cores, number of measurement samples to take etc.) and the "measure" function defines the actual measurement procedure. The specific measurement parameters initialized in "init" function should be defined in a xml configuration file (not in the main configuration file). Both the measurement class name and its corresponding configuration file should be specified in the main configuration file. The framework utilizes the Python language capability to dynamically load a class. This means that the user defined class is dynamically loaded by only specifying the class name in the input configuration file. No other change in the source code is required.

Eventually, a fitness value will be given on the generated individual based on the measurement results. This is needed so that the GA can rank the individuals and pick the fittest ones that satisfy the most the optimization goal(s). An individual can have many measurements associated with it, e.g. maximum voltage droop and average power consumption. The framework offers a default fitness class "DefaultFitness.py" that simply uses the first measurement (in the list order) as the fitness function. More complicated fitness functions might be desired, for instance, maximize voltage droop while keeping average power low. The framework offers the user the ability to define such functions by writing a custom class that inherits from "DefaultFitness.py" and overrides the "getFitness" function. Similarly, to the measurement scripts, to use the custom fitness class the user must specify the fitness class name in the main configuration file.

D. Output

The framework's output is the source code of all individuals. Each source code is saved in a different file. The name of the file includes: the population number, individual id and an array of measurements. For example, for the individual with id number 10 that belongs to population number 1 and with measured average and peak power of 1.3W and 1.33W respectively the file name would look like this 1_10_1.30_1.33.txt. By default, the first measurement is the fitness value, this naming convention facilitates the quick retrieval of the fittest individual using basic UNIX commands.

Moreover, each GA population is saved in a separate binary file. This binary file contains the source code, the id, the parent ids and the measurement values of each individual. These binary files can be loaded in a Python script for advanced result post-processing. As part of the framework release, there is a Python script that reads the populations in binary format and extracts statistics such as the fitness value of the fittest individual per generation and instruction mix breakdown of fittest individual per generation. Furthermore, the binary population files can be used as seed population for a new GA search (by default a new GA search starts with a randomly generated population). In such case, the user must specify the location of the seed population file in the main configuration file.

Additionally, in the output directory of each GA run the following are saved for record-keeping: the GA source code, the configuration files and the template individual source file used for the run.

IV. Experimental Setup

We evaluate GeST on 4 different CPUs shown in Table II. We generate power viruses for ARM Cortex-A15 and Cortex-A7 running on a bare-metal environment (without OS). The chips are hosted on a CoreTile Versatile Express evaluation board [17]. The board offers external measurement points that allow measuring CPU power, current and voltage. We hook an ARM energy-probe [20] on the measurement points to read the power. Next, we generate a power virus and an IPC virus for the Ampere Computing X-Gene2 ARM-based server CPU [18]. This server offers temperature sensor readings accessible through the i2c interface [21]. We use the i2c interface to generate the power virus by optimizing towards maximum temperature. The IPC virus is generated by monitoring the IPC from the perf Linux utility [19]. On the same system we demonstrate the GeST ability to optimize complex fitness

Table II. Experimental details.

CPU	# of Cores	Board	Environment	Stress-test developed	Measurement Instrument
ARM Cortex-A15	2	CoreTile Versatile Express	Bare Metal	power-virus	ARM energy probe
ARM Cortex-A7	3	CoreTile Versatile Express	Bare Metal	power-virus	ARM energy probe
Ampere X-Gene 2	8	Validation Board	Centos 7.2	power-virus and IPC virus	i2c temperature sensor readings, performance counters
AMD Athlon II X4 645	4	Asus M5A78L LE	Windows 8.1	dI/dt virus	External Oscilloscope hooked on voltage-sense points

functions (multi-objective) by generating a virus that targets both high temperature and instruction stream simplicity (fewer unique instructions). Lastly, we generate a dI/dt voltage noise virus on an AMD desktop CPU hosted on an Asus M5A78L LE motherboard. This motherboard offers high-bandwidth voltage measurements through external power-pads. We hook an active differential probe on the power-pads and measure the peak-to-peak voltage-noise on an external Agilent MSO9254A oscilloscope.

GA searches are performed on a single core. GeST can do multi-core optimizations by launching multiple workload instances but optimizing on single core has the advantage of less measurement variability which helps the GA optimization to converge faster. This is especially true when runs are conducted within an OS environment. Despite the GA search performed on a single core, a virus is tested by running it on all cores. All results reported in this paper are measured with all cores active with each core running a separate virus instance. The viruses developed in this work do not make use of shared resources (e.g. LLC). Hence, the generated viruses scale well with multi-core execution because running multiple virus instances is not causing performance interference. The other workloads are also executed on all cores. For single-thread benchmarks we execute multiple instances and for multi-thread benchmarks (e.g. NAS, Parsec) we execute one instance with multiple threads.

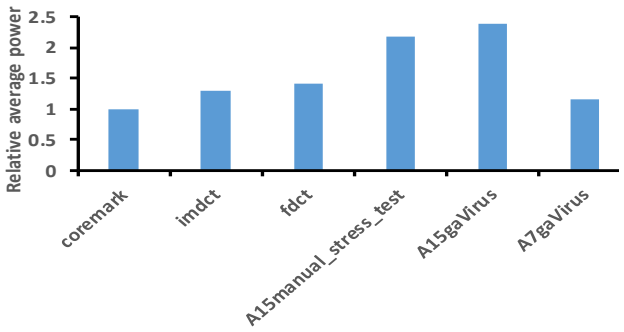

Figure 5. Cortex-A15 power results.

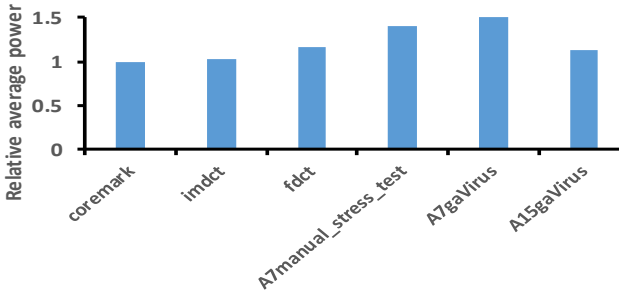

Figure 6. Cortex-A7 power results.

We are aware of a previous work [6] that evaluates a GA framework for power-virus generation on simulated multi-cores and reports significant increase in power-consumption when virus threads access shared memory. This increase in power consumption is attributed to the high engagement of network-on-chip, which in the simulated systems has a large contribution in total power consumption (for some runs more than 33% of the total power). In all CPUs we tested, we have successfully generated effective power/thermal stress-tests that exceed the fitness of the worst-case workload or manually-written stress-test by at least 10% without using shared memory. With that said, memory instructions that access shared memory can be added to the GeST optimization. The user must provide a template file that initializes shared-memory and launches multiple workload threads (in case the shared memory is defined in kernel then multiple process instances instead of threads should also work). Moreover, the user must define in the main configuration file the instructions that access the shared-memory. This important extension is beyond the scope of this work.

Regarding framework execution time, the GeST runtime is defined by the following factors: a) time to measure each individual, b) for how many generations the optimization is performed, and c) how many individuals are measured per generation (population size). In our experience GeST produces stress-tests that exceed significantly conventional workloads after 70-100 generations. Given 50 individuals per population and 5 seconds per measurement (which is typical for power optimization) the runtime is approximately 7 hours.

In the framework release we include measurement scripts and fitness functions that can be used for power, IPC, dI/dt noise and instruction-stream simplicity optimization for x86 and ARM ISA.

V. Power Viruses Generation

We develop a power-virus for Cortex-A15 and Cortex-A7 in a bare-metal environment. The measurement function for this optimization executes each GA generated binary for few seconds and takes multiple power readings during the binary execution. The fitness function calculates the average value of all power samples. Fittest individuals are considered the ones with the highest average power.

The relative (normalized to coremark benchmark) power-results for Cortex-A15 and Cortex-A7 are shown in Figure 5

Table III. Instruction breakdown of Cortex-A15 and Cortex-A7 power viruses.

GA virus	Short Latency Int	Long Latency Int	Float/ SIMD	Mem	Branch	Total Loop Instructions
Cortex-A15	4	5	22	18	1	50
Cortex-A7	8	6	16	10	10	50

and Figure 6 respectively. First, it is worth noting that the GA generated stress-test both on Cortex-A15 and Cortex-A7 cause the highest power consumption and surpass the manually written stress-tests (A15manual_stress_test, A7manual_stress_test) as well as conventional bare-metal workloads (coremark, imdct, fdct). This emphasizes the GA's ability to generate worst-case pathological scenarios that are hard for humans to produce. The other interesting observation is that Cortex-A7 GA virus is not a good stress-test for Cortex-A15 and Cortex-A15 virus is not a good stress-test for Cortex-A7. Different CPU designs require different stress-tests to maximize their CPU power consumption. The need for different stress-tests for dissimilar micro-architecture is also evident by the differences in the instruction mix between the Cortex-A15 and Cortex-A7 GA-power-viruses depicted in Table III. The breakdown is shown in terms of short latency (1 cycle) integer instructions (e.g. ADD, SUB), multi-cycle instructions (e.g. MUL), float or SIMD instructions, memory instruction and branch instructions. Both stress-tests consist of a loop of 50 instructions. The table shows that to raise the Cortex-A7 power consumption it is important to add a lot of branch instructions (10 instructions out of 50 are branches) while for Cortex-A15 only one branch is used. Also, Cortex-A7 virus prefers slightly shorter latency integer instructions as compared to Cortex-A15 virus. A common observation for both viruses is that floating point/SIMD instructions are dominant.

Next, we test GeST on the X-Gene2 ARM-based server CPU. We generate a virus that maximizes chip temperature (and hence the power) using chip temperature sensor feedback. We compare the temperature of the virus (denoted as powerVirus) with various benchmarks (Parsec and NAS suite) and a virus that maximizes IPC (denoted as IPCvirus) generated with the GA using perf Linux utility. Figure 7 shows the relative (normalized to bodytrack benchmark) chip temperature. The power virus outperforms all other workloads by reaching the highest chip temperature.

The IPC virus also raises the chip temperature very high (but lower than power virus). IPC virus is expected to cause high temperature because it causes very high CPU activity. It is interesting to understand what characteristics make the power virus cause higher temperatures. Table IV provides a comparison of the IPC and the power viruses. The IPC virus achieves 12% higher IPC but also 12% lower power consumption than the power virus. As expected, the IPC virus does not contain any long latency integer instruction. Also, the IPC virus makes moderate use of memory operations. On the other hand, the power virus contains a few long-latency instructions and uses a lot of memory operations. Perhaps the modest use of long-latency instruction helps to increase the power consumption and temperature (which is the goal of the virus) by keeping active the issue queue and the dependency tracking logic. Also, the more frequent engagement of the memory subsystem results in a higher power consumption. While the use of long-latency operations and many memory instructions increases the temperature, it also reduces the IPC. This highlights an interesting tradeoff that the GA is capable to make to maximize the temperature. This analysis clearly shows that the highest IPC does not automatically convert to highest power consumption and temperature. A recipe for the highest power consumption and temperature (at least for the X-Gene2) seems to be a combination of high IPC (not the highest) with heavy use of memory instructions and modest use of long-latency operations.

A. Complex Fitness Functions

We demonstrate the GeST capability to optimize a complex fitness function by generating a power-virus that achieves both high temperature and simplicity in terms of using less unique instructions (unique opcodes). Simplicity of the generated power-viruses is desired for various reasons such as for ease in isolating inefficiencies in initial chip samples, like hotspots, and instructions that are power-intensive. To optimize for both high-temperature and simplicity, we use the GeST interface for scripting custom fitness functions (presented in Section III.C). We use Equation 1 for calculating the individual's fitness. The fitness can take values from 0 to 1 and the equation has two parts, with both parts contributing equally to the fitness value. The first part rewards high temperature. The contribution of the temperature part must be bounded to a 0-1 value range

$$F = (M\_T - I\_T) / (MAX\_T - I\_T) * 0.5 + (T\_I - U\_I) / T\_I * 0.5$$

Fitness (F), M_T (measured temperature), I_T (idle temperature), MAX_T (max temperature), T_I (total instructions), U_I (unique instructions)

Equation 1. Complex fitness function rewarding high temperature and instruction simplicity.

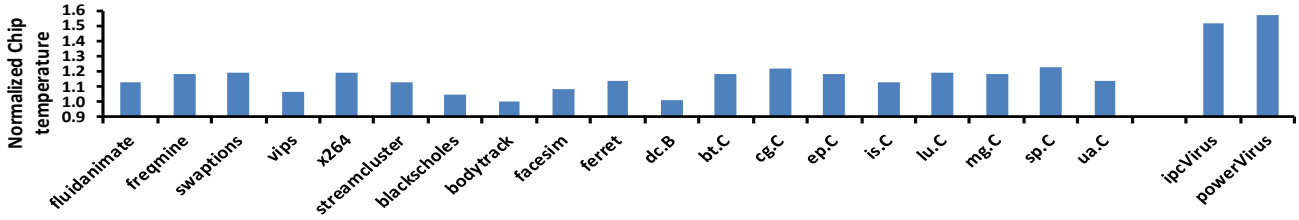

Figure 7. X-Gene2 chip temperature results.

Table IV. Power virus, simple power virus and IPC virus comparison.

GA virus	ShortInt	LongInt	Float/SIMD	Mem	Branch	Relative IPC	Relative Plug Power (W)	Relative Chip Temp.	# of Unique Instructions
powerVirus	22	5	9	12	2	1	1	1	21
powerVirusSimple	16	7	13	11	3	0.94	0.99	1	13
IPCvirus	26	0	15	6	3	1.12	0.88	0.94	13

(temperature score), hence, we normalize the measured temperature with the maximum possible temperature. The maximum temperature can be obtained either from a previous GA run or from specifications e.g. TJMAX. An issue with the temperature score is that even during idle operation the temperature is not negligible because of ambient temperature. Thereby, we must subtract the idle temperature to avoid overestimation of the temperature score. The second part of the equation is the simplicity which rewards having less unique instructions. It is also bounded between a 0-1 value range. Assuming individuals of 50 instructions, an individual with 25 unique instructions would be assigned a simplicity score of 0.5 whereas an individual with 15 unique instructions would be assigned a simplicity score of 0.7 (without taking in account the 0.5 weight factor).

We run the GA with the complex fitness function for the same number of populations as the GA that generated the power virus. The characteristics of the fittest individual (powerVirusSimple) are shown in Table IV. This virus has very similar characteristics with the original power virus. Specifically, we observe the same characteristics we discussed in the previous paragraphs such as fairly high IPC, significant use of memory and modest use of long latency integer instructions. However, there is also a difference, the new power virus prefers to spend more instruction slots for floating point and long latency instructions at the expense of the short latency instructions. This has an impact on the IPC which is 6% lower compared to the original power virus but this doesn't affect its temperature and power consumption. The simple power virus achieves virtually the same power and the same temperature as the original power virus. The complex fitness optimization is considered successful as the simple power-virus achieves the same stress-level as the original power-virus while using only 13 unique instructions instead of 21.

VI. Voltage Noise Virus Generation

This section demonstrates the capability of GeST to generate voltage noise viruses and consequently stability-tests. For this study we use an AMD Athlon II X4 645 CPU hosted on an Asus M5A78L LE motherboard. This motherboard offers high bandwidth voltage sense points that can be used to monitor voltage noise. This is achieved by connecting an oscilloscope to the sense points through an active differential probe. The GA generates the dI/dt virus by optimizing towards maximum peak-to-peak voltage. The framework runs each GA generated binary for a few seconds. During the binary execution the minimum and maximum voltage observed on the oscilloscope are recorded. The binaries that achieve the highest difference between maximum and minimum recorded voltages are considered the fittest.

Figure 8 shows the max-min voltage noise caused by various workloads compared to the GA generated virus. The GA dI/dt virus clearly outperforms the other workloads including well known stability-tests such as Prime95 and AMD's own stability test. Since the dI/dt virus causes the highest voltage-noise it should stress the system's stability better than the other workloads. A good stability-test must have high $V_{MIN.}$ To characterize the V_{MIN} of a workload we run the workload multiple times and each time we lower the operating voltage in steps of 12.5mV. We keep the CPU frequency stable at the nominal value of 3.1GHz. The highest voltage at which a workload executes correctly (without corruption, error messages, crashes) is the workload's $V_{MIN.}$ Figure 9 shows the V_{MIN} of the various workloads we tested on the AMD CPU. The dI/dt virus is the best stability-test because it causes instability at a higher voltage, even higher than the commonly used AMD stability test and Prime95.

Our results show that workloads designed to draw very high power are not suitable stability-tests as they are not designed to drop the voltage very low and induce timing errors. Prime95 is a workload known to raise the CPU power consumption very high. Such workloads are a good choice for characterizing thermal stability and making sure that the temperature will not exceed a critical threshold during normal operation. But they are inadequate for characterizing the

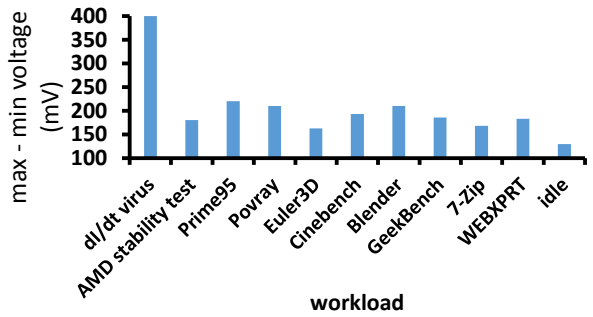

Figure 8. Voltage-noise results on AMD Athlon CPU.

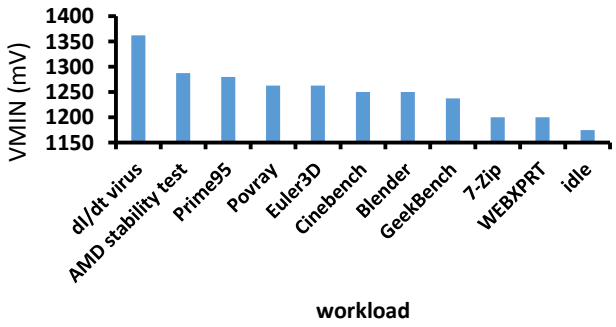

Figure 9. V_{MIN} results on AMD Athlon CPU.

Table V. Comparison of related work on GA frameworks.

Framework	OptimizationType	Optimization-Language	Evaluated-On	Metrics Evaluated	Component Stressed	References
AUDIT	Instruction-Level	x86 ISA	Real-Hardware / Simulator	dI/dt	CPU	[1][3]
MAMPO	Abstract-Workload	SPARC ISA	Simulator	power	CPU+DRAM	[7],[6]
Joshi et al.	Abstract-Workload	Alpha ISA	Simulator	power	CPU	[4]
Powermark	Abstract-Workload	C	Real-Hardware	power	Full-System	[5]
GeST	Instruction-Level	ARM,x86	Real-Hardware	dI/dt,power	CPU	this work

susceptibility to timing errors.

VII. Related Work

This section discusses and compares qualitatively GeST with previous GA frameworks for stress-test generation. Table V provides an overview of the state-of-the-art GA stress-test generation frameworks.

We consider the pairs of works [1][3], and [6][7] as each representing the same framework. Particularly, the work in [3] has evaluated a dI/dt GA framework on a simulated environment and subsequently on real multi-core hardware [1]. Similarly, the work in [7] generates GA power viruses for single-core CPUs and a latter extension on multi-core CPUs [6]. In a different line of work, Joshi et al. [4] evaluated a power-virus GA framework on an Alpha ISA single-core simulator and Polfliet et al. [5] evaluated a power-virus GA framework on real-hardware using x86 multi-cores.

As shown in Table V, there are two dominant approaches in designing GA frameworks for stress-test generation: a) based on an abstract-workload model and b) based on instruction-level primitives (usually assembly instructions). In the abstract-model frameworks the individual is a vector of workload related parameters such as instruction-mix, register-dependency distance, memory-stride profile, branch transition rates etc. The GA operators are performed on this abstract workload profile. A workload generator stochastically generates the assembly (or higher-level language) code based on the values of the abstract model parameters. On the other hand, for the instruction-level optimizations the individual is the actual source code of the virus. The GA performs the optimization directly on the source-code and has full-control on the instruction-mix, instruction-order and instructions' operands. GeST as presented in Section III utilizes the instruction-level optimization approach.

An advantage of the abstract workload model is that it reduces the design space. A disadvantage of the abstract model is that it fails in optimizing the instruction order and the instruction opcodes simply because these parameters are out of GA control. Previous work [8] reports that instruction-order can make up to 17% difference in power for the same activity factor and instruction-mix.

Moreover, knobs typically found in abstract-workload frameworks that allow fine tuning memory accesses and branch behavior, through parameters such as memory stride and branch transition rate, seem not so relevant, at least, for high power and dI/dt workloads. As reported in previous work [4][7] and confirmed in this work, power-viruses are characterized by high IPC, very predictable branches and extremely high L1 hit rates. These characteristics can easily be achieved with instruction-level optimization. Regarding dI/dt optimization, all previous work utilized instruction-level optimizations [1][2][3]. This is the case since dI/dt optimization is very sensitive to the workload frequency that must match the PDN resonance frequency. For such optimization search, instruction-order is more important than disruptive events such as cache-misses and branch-misprediction that cause non-determinism and limit the capacity to control the workload frequency [2].

Another design choice of a GA framework is the optimization language. Most frameworks prefer generating assembly code except [5] that prefers a high-level language like C. The advantage of using higher level language is that it makes the framework versatile to the hardware platform of interest. Using a higher-level language makes sense in conjunction with an abstract workload model. For instruction-level optimization this is not so practical because it prevents GA to directly optimize the instruction type mix and order (the final instruction order and types depend on the compiler). For GeST we prefer the assembly instruction level optimization. The versatility of GeST that allows its use with any hardware platform stems from providing an interface to the experimenter to specify the instructions that will be used in the optimization. Thereby, this allows the experimenter to use GeST to customize and optimize for any ISA.

Finally, another important GA framework aspect is the component it targets. Most works justifiably target the CPU as it is generally accepted that CPU is the most active and power-hungry component. In [5] authors generated full-system stress-tests that also stressed the network-interface-card and hard-disk. This is achieved by adding a thread that sends network packets and a thread that performs disk reads, the invocation frequency of these threads is a parameter of the abstract-workload-profile. GeST is as an instruction-level optimization framework that primarily targets CPU, but it is also applicable to any other component that can be stressed through a stream of instructions. For instance, with GeST is possible to stress LLC or DRAM by instructing the framework to optimize towards cache-misses and providing in the input file load/store instruction definitions with various strides, base memory registers and various min-max immediate values. We are currently investigating such extensions.

To conclude this discussion, to the best of our knowledge none of the other previously presented GA frameworks is publicly available. Also, we believe that GeST is the first work that targets user-friendliness, extensibility, flexibility and re-usability. GeST achieves these features by providing a clean interface to experimenters for: a) scripting their own measurement procedures, b) writing custom fitness functions,

and c) specifying instructions and operands that will be used in the GA search.

VIII. Conclusion

This work proposes GeST, a framework for automatic stress-test generation based on GA. While GA based automatic frameworks are not a novel concept, to the best of our knowledge there is no publicly available framework that researchers and practitioners can use. The framework presented in this paper has successfully been demonstrated in industrial platforms and has been used for various research publications [22][23][24][25]. The framework codebase is available in [26].

The key strengths of the framework are its flexibility and extensibility as it provides an easy interface to the experimenter that can be used for building upon the framework. We demonstrate the flexibility and the effectiveness of the framework by generating, among other, power and dI/dt stress-tests (viruses) on various CPUs with simple and complex fitness functions. The generated viruses stress the system more than conventional workloads and manually written stress-tests.

While this paper demonstrates GeST on real hardware, there is no fundamental restriction that prevents the framework from being used for pre-silicon stress-test generation in conjunction with accurate power, temperature, performance and voltage-noise models/simulators.

Acknowledgment

This work is funded by the H2020 Framework Program of the European Union through the UniServer Project (Grant Agreement 688540) – http://www.uniserver2020.eu. Part of this work has been conducted during an internship of the first author at Arm Research.

References

[1] Kim, Youngtaek, Lizy Kurian John, Sanjay Pant, Srilatha Manne, Michael Schulte, William Lloyd Bircher, and Madhu Saravana Sibi Govindan. "AUDIT: Stress testing the automatic way." In Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on, pp. 212-223. IEEE, 2012.

[2] Bertran, Ramon, Alper Buyuktosunoglu, Pradip Bose, Timothy J. Slegel, Gerard Salem, Sean Carey, Richard F. Rizzolo, and Thomas Strach. "Voltage noise in multi-core processors: Empirical characterization and optimization opportunities." In Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on, pp. 368-380. IEEE, 2014.

[3] Kim, Youngtaek, and Lizy Kurian John. "Automated di/dt stressmark generation for microprocessor power delivery networks." In Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design, pp. 253-258. IEEE Press, 2011.

[4] Joshi, Ajay M., Lieven Eeckhout, Lizy K. John, and Ciji Isen. "Automated microprocessor stressmark generation." In High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on, pp. 229-239. IEEE, 2008.

[5] Polfliet, Stijn, Frederick Ryckbosch, and Lieven Eeckhout. "Automated full-system power characterization." IEEE Micro 31.3 (2011): 46-59.

[6] Ganesan, Karthik, and Lizy K. John. "MAximum Multicore POwer (MAMPO): an automatic multithreaded synthetic power virus generation framework for multicore systems." Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011.

[7] Ganesan, Karthik, Jungho Jo, W. Lloyd Bircher, Dimitris Kaseridis, Zhibin Yu, and Lizy K. John. "System-level Max Power (SYMPO)-A systematic approach for escalating system-level power consumption using synthetic benchmarks." In Parallel Architectures and Compilation Techniques (PACT), 2010 19th International Conference on, pp. 19-28. IEEE, 2010.

[8] Bertran, R., Buyuktosunoglu, A., Gupta, M. S., Gonzalez, M., & Bose, P. (2012, December). Systematic energy characterization of cmp/smt processor systems via automated micro-benchmarks. In Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on (pp. 199-211). IEEE.

[9] Mitchell, Melanie. An introduction to genetic algorithms. MIT press, 1998.

[10] Zu, Yazhou, Charles R. Lefurgy, Jingwen Leng, Matthew Halpern, Michael S. Floyd, and Vijay Janapa Reddi. "Adaptive guardband scheduling to improve system-level efficiency of the POWER7+." In Proceedings of the 48th International Symposium on Microarchitecture, pp. 308-321. ACM, 2015.

[11] Das, Shidhartha, Paul Whatmough, and David Bull. "Modeling and characterization of the system-level Power Delivery Network for a dual-core ARM Cortex-A57 cluster in 28nm CMOS." Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on. IEEE, 2015.

[12] Whatmough, Paul N., Shidhartha Das, Zacharias Hadjilambrou, and David M. Bull. "14.6 An all-digital power-delivery monitor for analysis of a 28nm dual-core ARM Cortex-A57 cluster." In Solid-State Circuits Conference-(ISSCC), 2015 IEEE International, pp. 1-3. IEEE, 2015.

[13] Grenat, Aaron, Sanjay Pant, Ravinder Rachala, and Samuel Naffziger. "5.6 adaptive clocking system for improved power efficiency in a 28nm x86-64 microprocessor." In Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International, pp. 106-107. IEEE, 2014.

[14] Papadimitriou, G., Chatzidimitriou, A., Kaliorakis, M., Vastakis, Y., & Gizopoulos, D. (2018, April). Micro-Viruses for Fast System-Level Voltage Margins Characterization in Multicore CPUs. In Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on (pp. 54-63). IEEE.

[15] https://www.aida64.com/

[16] https://www.mersenne.org/

[17] http://infocenter.arm.com/help/topic/com.arm.doc.ddi0503i/DDI0503I_v2p_ca15_a7_tc2_trm.pdf

[18] https://www.hotchips.org/wp-content/uploads/hc_archives/hc26/HC26-11-day1-epub/HC26.11-4-ARM-Servers-epub/HC26.11.430-X-Gene-Singh-AppMicro-HotChips-2014-v5.pdf

[19] https://perf.wiki.kernel.org/index.php/Tutorial

[20] https://developer.arm.com/products/software-development-tools/ds-5-development-studio/streamline/arm-energy-probe

[21] https://linux.die.net/man/8/i2cget

[22] Whatmough, Paul N., Shidhartha Das, Zacharias Hadjilambrou, and David M. Bull. "Power integrity analysis of a 28 nm dual-core arm cortex-a57 cluster using an all-digital power delivery monitor." IEEE Journal of Solid-State Circuits 52, no. 6 (2017): 1643-1654.

[23] Hadjilambrou, Zacharias, Shidhartha Das, Marco A. Antoniades, and Yiannakis Sazeides. "Leveraging CPU Electromagnetic Emanations for Voltage Noise Characterization." In 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 573-585. IEEE, 2018.

[24] Hadjilambrou, Z., Das, S., Antoniades, M. A., & Sazeides, Y. (2018). Sensing CPU voltage noise through Electromagnetic Emanations. IEEE Computer Architecture Letters, 17(1), 68-71.

[25] Tovletoglou K, Mukhanov L, Karakonstantis G, Chatzidimitriou A, Papadimitriou G, Kaliorakis M, Gizopoulos D, Hadjilambrou Z, Sazeides Y, Lampropulos A, Das S. Measuring and Exploiting Guardbands of Server-Grade ARMv8 CPU Cores and DRAMs. In 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W) 2018 Jun 25 (pp. 6-9). IEEE.

[26] https://github.com/toolsForUarch/GeST