Individual Diploma Thesis

Speeding up the process of generating stress tests

ANDREAS FRANGOS

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2022

UNIVERSITY OF CYPRUS
DEPARTMENT OF COMPUTER SCIENCE

Speeding up the process of generating stress tests

ANDREAS FRANGOS

Supervising Professor
Yanos Sazeidis

The Individual Diploma Thesis was submitted for partial fulfillment of the requirements for obtaining the Informatics degree of the Department of Informatics of the University of Cyprus

May 2022

Thanks

I would like to thank myself for giving my best in this Thesis research, for the hours and hard work I put in. Also, I would like to thank Georgia Antoniou for helping me throughout the whole research, always by my side to discuss ideas and to guide me. Furthermore, I would like to thank Mr. Yanos Sazeidis for pushing me into giving my best self, making the research exciting and always giving me new objectives. Moreover, I would like to thank my friend Marios Tsokkos for being always by my side giving me motivation and energy to continue, supporting me in my worst times and uplifting me in my life. And lastly I would like to thank the gym, for keeping my mind off things, helping me focus into my work and letting tough thoughts go away.

Abstract

Stress-tests - viruses that target specific microarchitectural characteristics of a processor are of tremendous importance as they not only determine how reliable a system is, but also affect characteristics of a system such as the power consumption, performance, stability etc. Today, the procedure of actually generating these viruses is time consuming (both manual and automatic) as different hardware has different characteristics and thus requires different viruses. Apart from that, the design space of the problem is big because the instruction set architecture and the possible combinations of instructions are big. In this work we investigate how we could speed up the generation time of a virus using GeST[3] an automatic framework for generating CPU stress tests.

The main focus of this work are power viruses (which are small programs stressing the CPU in terms of power consumption) on the i5-2400 Sandy Bridge Intel processor. Firstly, we generate a strong power virus for i5-2400 using GeST. Then we use the best virus which we call the reference virus along with the intermediate viruses that GeST generates to extract the parameters(such as instruction mix, IPC) that affect the power virus most. We validate the correlation of the parameters with the power consumption of the virus using Euclidean distance and interpolation. After we finalize the parameters that affect the power virus, we modify GeST so that it generates power viruses based on the Euclidean distance we created in the previous step instead of actually executing the viruses at each intermediate step. We manage to speed up the generation time of GeST for i5-2400 processor creating a virus that matches the performance of the reference power virus.

Contents

Chapter 1

1 Introduction

---

1. Motivation)
1.2 Goal
1.3 Research Questions
1.4 Contributions
1.5 Structure

1.1 Motivation

We are living in a time of technological revolution. Smartphones and laptops have become an essential part of our everyday live. This is mainly due to the characteristics of these devices, like the variety of functionalities, the stability, the cost, and the performance. To maintain these characteristics for each generation, manufacturers have developed a phase in their life cycle called reliability testing. During this phase processors are tested in extreme conditions with the help of stress-test programs. This is mandatory because it validates the design and the characteristics of the CPU.

There are two methods that one can use to generate stress tests, manual and automatic. For the manual approach, the programmer must write and test the stress-tests whereas for the automatic approach, the programmer just specifies the optimization metric and the automatic tool (usually based on machine learning optimization search algorithms like Genetic Algorithms and Gradient Descent) generates the stress test. For both methods the procedure of generating the stress-test is time consuming because the

programmer or the tool must compile and run it in each step of the way. There is a need to optimize this procedure, not only because is time consuming but also because it is being executed multiple times for each CPU generation (stress tests are based on the microarchitecture of the CPU and cannot be reused) and each of its characteristics (power consumption, voltage noise, IPC).

1.2 Goal

The goal is to develop a method that generates stress tests, without compiling and executing them. The idea is to have a reference virus represented by features like for example number of scalar instructions. Then instead of compiling and executing the intermediate viruses one can just calculate the Euclidean distance of an intermediate virus from the reference one in order to drive the optimization search. Calculating the Euclidean distance is much faster than compiling and executing the virus.

1.3 Research Questions

Research questions answered in this thesis:

1. Is the Quickgen method faster than the already existing ones? My research is indeed faster, taking only 1 minute to finish, in contrast to 5 hours using already existing methods.,
2. Can I do something better? A key of the proposed method is the availability of a reference virus, in this search I found out if it is possible to achieve a search that can produce a program with the same characteristics as the once set for reference.
3. Can I easily produce a reference program? There are several ideas that can make the production of the reference program easy, and I am going to discuss the ideas in future chapters.

1.4 Contributions

In this research I found the features and their values that affect the performance of a program, this is called the instruction mix of the program, and consists of the type of

instructions (add, vmulpd, load, store etc) and the number of those instructions in the program. Also, I found a direct correlation of those features and their values with the power consumption of a program using the Euclidean distance formula. Furthermore, I manage to produce power viruses more than 300 times faster than the original methods that compile run and uses power consumption metrics to generate power viruses.

1.5 Structure

The rest of the thesis is organized as follows: Chapter 2 and 3 describe the necessary background, specifically what is a machine, a processor, an ISA, necessary metrics like power consumption and instructions per cycle (IPC), statistical metrics and GeST. Chapter 4 describes the idea of the proposed method to speed up stress-test generation time, it specifically describes the QuickGen method. Chapter 5 presents the experimental setup of my experiments. Chapter 6 shows the results of the experimental evaluation of the proposed method and finally Chapter 7 presents the related work, future work and conclusions.

Chapter 2

Background Info

2.1 Machine & Processor & ISA

2.2 Instructions Per Cycle

2.4 Power Consumption

2.5 Statistical Metrics

2.1 Machine & Processor & ISA

This research is dedicated to finding a program that targets a machine, so a very basic factor in my research is the machine itself, and its components. It's very important to know everything about the components of the machine because every pc has its own characteristics and the program that archives maximum power consumption varies from machine to machine. So, to be able to achieve a powerful program you need to understand the machine you are using. The machine that I use is the HP compaq 8200 elite cmt pc, this pc has the Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz for processor, it is also named Sandy Bridge. This processor consists of 4 cores that do not support hyperthreading, the base frequency of the processor is 3.10 GHz, it also supports max turbo frequency at 3.40 GHz. Am only going to use the base frequency for the whole research, and all the searches will be run in two instances, this means that only 2 of the 4 cores will be used for the experiments. The processor has 6MB of cache and bus speed of 5 GT/s. Going to the microarchitecture of the processor we can observe that the L1 instruction cache can accept 16 bytes/cycle, also there is a 4-way decode, that means it can only get 4 instructions at a time. Moreover, this microarchitecture has 6 execution units, this means that it can execute 6 commands at a time, the execution ports go as follows. The first unit supports integer arithmetic logic unit, integer division, integer vector arithmetic logic unit, integer vector multiply, floating point multiply, floating point division and vector shuffle. The second unit supports arithmetic logic

unit, integer multiply, integer vector arithmetic logic unit and floating point add. The third unit supports integer arithmetic logic unit, vector shuffle, integer vector arithmetic logic unit and branch. The fourth unit supports address generation unit and load data, the same goes for the fifth, and lastly the sixth unit supports store data. With the above data we can observe that there are 3 units that support computation and 3 units that support memory access. From the 3 units that support computation all of them support integer and integer vector ALUs, only the two units support floating point computation and only one supports branches. Analysing the 3 units that support memory access, there are two units for load and AGU and only one that supports store of data.[7]

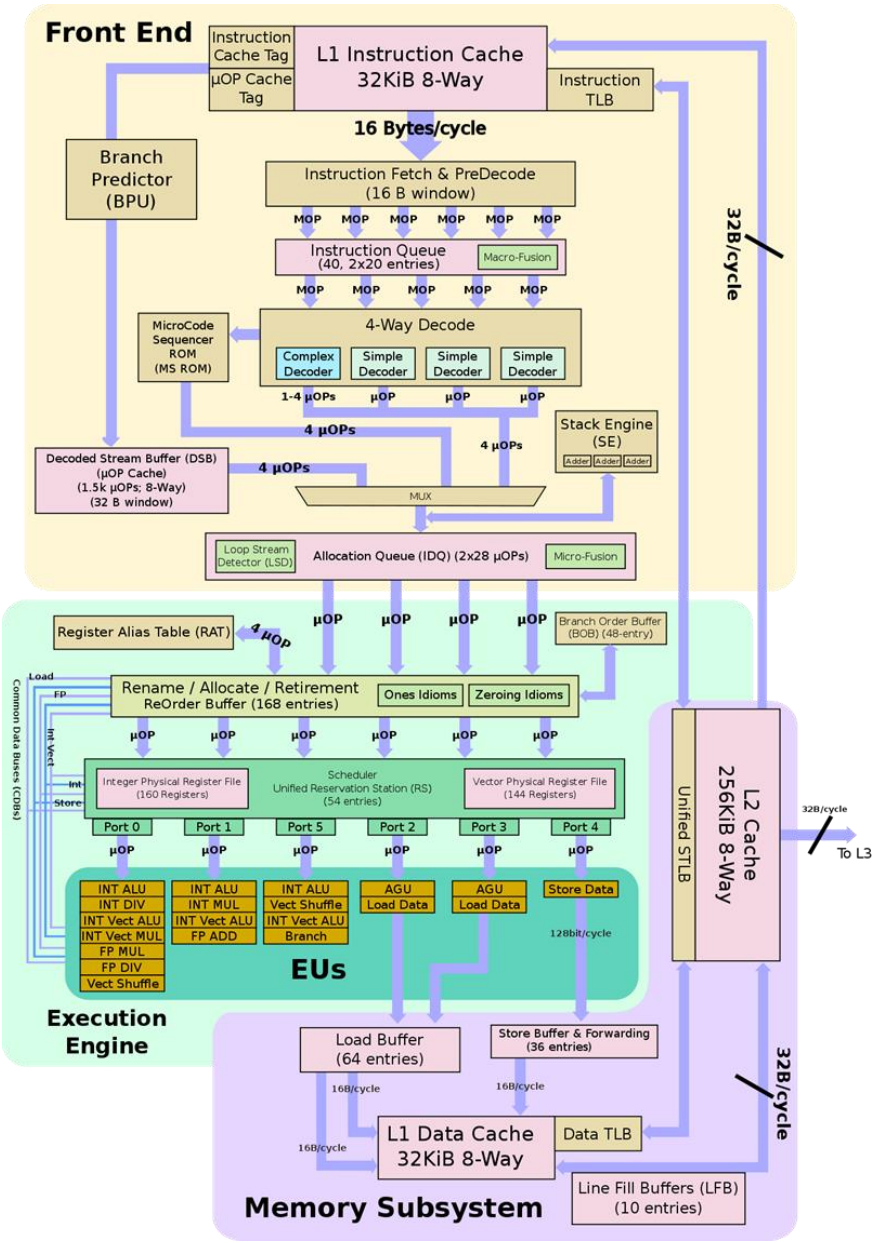


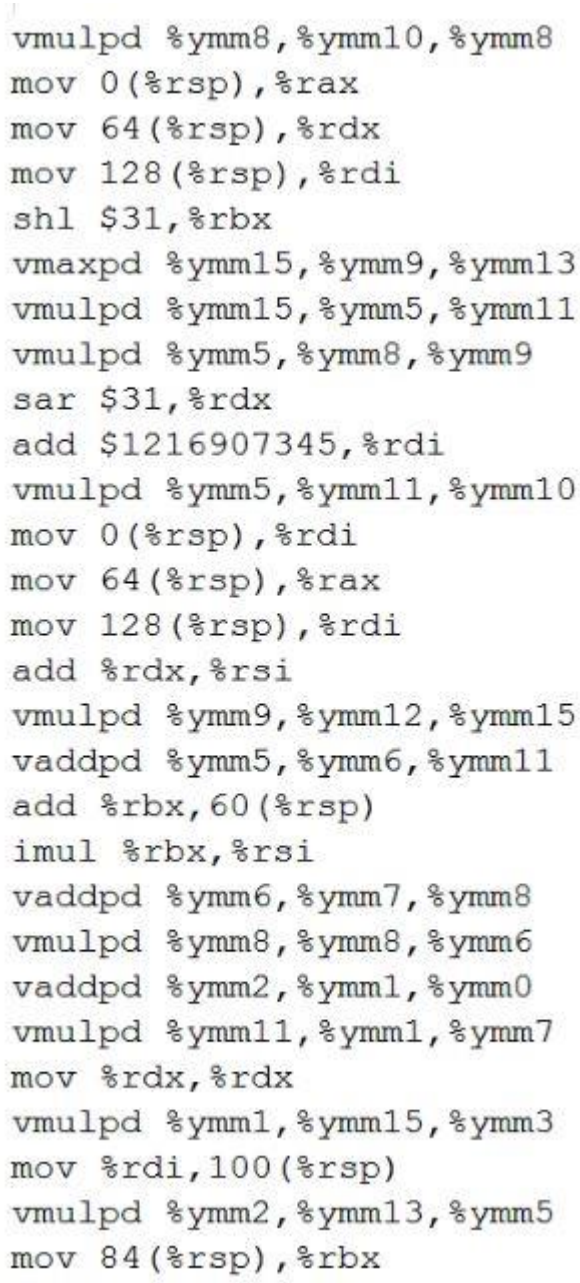
Figure 2.1.1 Microarchitecture of Sandy-Bridge processor

Figure 2.1.1 supports x86 instruction set architecture. ISA, it defines how the processor processes and executes various instructions issued by the operating system (OS) and software programs. The key characteristics of x86 architecture are, this architecture provides a logical framework for the processor to execute instructions, also allows programs and instructions to run on any processor included in the Intel 8086 family. Furthermore, this architecture provides methods for utilizing and managing the hardware components of the CPU. The x86 architecture primarily handles program functionality and provides services such as memory addressing, software and hardware interrupt handling, data types, registers, and input / output management. [11]

In this research I will be studying the x86 architecture, more specifically the instructions of this architecture.

- Scalar instruction performs computation on one number or set of data at a time.
- Vector instruction, multiple operations of the same type are performed on several datasets at once during the execution of a single processor instruction.
- Load instruction is used to move the data or memory address in memory to a register.
- Store instruction is used to move the value in the register to memory.
- Combination of instruction types.

Dependencies are very interesting because they introduce latencies in a program. The basic hazards are Read after Write (RAW), occurs when instruction B tries to read data before instruction A writes it. Write after Write (WAW), hazard occurs when instruction A tries to write output before instruction B writes it. Write after Read (WAR), occurs when instruction B tries to write data before instruction A reads it. WAW and WAR hazards are easy to overcome by implementing register renaming by software or hardware. RAW hazards are very crucial because the way to overcome them are not that easy, only by applying out-of-order execution.



```
vmulpd %ymm8,%ymm10,%ymm8
mov 0(%rsp),%rax
mov 64(%rsp),%rdx
mov 128(%rsp),%rdi
shl $31,%rbx
vmaxpd %ymm15,%ymm9,%ymm13
vmulpd %ymm15,%ymm5,%ymm11
vmulpd %ymm5,%ymm8,%ymm9
sar $31,%rdx
add $1216907345,%rdi
vmulpd %ymm5,%ymm11,%ymm10
mov 0(%rsp),%rdi
mov 64(%rsp),%rax
mov 128(%rsp),%rdi
add %rdx,%rsi
vmulpd %ymm9,%ymm12,%ymm15
vaddpd %ymm5,%ymm6,%ymm11
add %rbx,60(%rsp)
imul %rbx,%rsi
vaddpd %ymm6,%ymm7,%ymm8
vmulpd %ymm8,%ymm8,%ymm6
vaddpd %ymm2,%ymm1,%ymm0
vmulpd %ymm11,%ymm1,%ymm7
mov %rdx,%rdx
vmulpd %ymm1,%ymm15,%ymm3
mov %rdi,100(%rsp)
vmulpd %ymm2,%ymm13,%ymm5
mov 84(%rsp),%rbx
```

Figure 2.1.1 Example of code of x86 architecture

Figure 2.1.1 shows an example of a code written in x86 ISA. This can help us identify the types of instruction with visual representation. In this code we have the following examples of instructions:

Vector instructions: vmulpd, vaddpd and vmaxpd

Scalar instructions: add, sar, shl, mov

Load instructions: mov 84(%rsp), %rbx

Store instructions: mov %rdi, 100(%rsp)

Combination of Scalar and store: add %rbx, 60(%rsp)

2.2 Instructions Per Cycle

Instructions Per Cycle (IPC) is an aspect that is investigated in this research, this term is correlated on CPU performance, more specifically is the average number of instructions executed in a single clock cycle.

2.4 Power Consumption

Power consumption is the amount of energy per unit time, the measurement unit is Watts. Every machine has an idle power consumption which is the power consumption that the machine consumes when it is not being used by any program. The power consumption changes as soon as the idle state changes and a process starts running on the CPU. The power consumption grows based on how powerful the program running on the CPU is. Also, the power consumption grows when multiple programs run in multiple cores of the CPU. So, the power consumption grows based on how many programs and how powerful are the programs running. The greater the power consumption the greater the temperature of the system.

C-states or Processor idle sleep states are states when the CPU has lower or turned off specific functions. Each processor supports its own C-states where multiple parts of the CPU are turned off. Most of the time, the higher the C-states the more parts of the CPU are turned off, this reduces power consumption. Furthermore, processors have C-states that are hidden from the operating system.

P-states or power performance states are states that increase the frequency and the voltage that the CPU runs with purpose to reduce the power consumption. Each processor has its own number of P-states. [5]

2.5 Euclidean Distance and Statistical Methods

Statistical methods are used to analyse raw data and provide different ways to assess the robustness of research outputs. The methods are mathematical formulas, models, and techniques. The main methods that I used in this research are finding the average value. It is a simple method and common, you add all the data and then divide them with the total amount of data. Also, I used the method of interpolation which is a method where known values are used to estimate an unknown value. The formula of interpolation is projected below.

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

Furthermore, one other method I used in this research was the calculation of error, this was a great tool to find conclusions and it played a big part in the validation. But the most crucial method used was Euclidean distance, this method was the most important tool for my thesis progression and for the search method I found. Euclidean distance is a statistical method where you set a list of reference values and compare them with the corresponding values of other lists of values you find the deviation between the two lists. The formula of Euclidean distance is projected below.

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(y_i - x_i)^2}$$

Where x and y are vectors, and y values are subtracted from x values then squared and the result is squared rooted.

Chapter 3

Virus programs & GeST

3.1 Viruses & Stress-Tests

Computer virus, is a program that most of the times has a malicious purpose, is also a program that can replicate itself and be spread across networks to other computers. Those types of programs can serve several purposes, it always depends on what type of virus it is. But in general viruses have the power to disrupt data, cause vital damage to the operational system, causing data loss and data leakage. They achieve that using an executable file that most of the time is very difficult to trace because the file is hiding behind other processes running. (“What are Computer Viruses? | Definition & Types of Computer Viruses”). The major types of viruses are macro-viruses, which are programs that have built in scripts hiding in common applications like Microsoft Excel or Microsoft Word; it replicates itself when someone opens a file infected by macro-virus. Some other type of virus is File-infecting virus, which is parasitic and commonly infects in executable programs (.exe or .com), can overwrite host files and can damage hard drive’s formatting process. One other example of virus is Browser Hijacker, this virus redirects you to some malicious websites that you do not want to visit, sometimes it even changes the homepage of the browser. Also, there are web scripting viruses that change the code on popular websites and insert links that can install other malicious

programs, furthermore they can steal cookies and use the information they can post under your name in the website. Moreover, there are viruses that are commonly found in USBs and external hard drives, that when injected on the computer they load into the memory, taking control of the computer, those viruses are named Boot Sector Viruses. There is also a form of intelligent virus that is called Polymorphic virus, it basically changes code every time an infected file is performed to trick the anti-virus system. Lastly, some of the most dangerous types of viruses are Multipartite viruses, because they are very infectious, taking over memory, files, and boot sectors which makes it difficult to limit. [9]

3.1.2 Stress-tests

Stress-testing is essential in all systems, computers, networks, programs and devices have to be stress-tested. Stress-testing basically places systems under extreme conditions and observes if it maintains a certain level of effectiveness. Some of the most common stress-tests are running several resource-thorough programs at the same time in a single system, attempting to breach a system, flooding a server with spam emails, sending numerous requests at the same time to access a website and attempting to infect a system with viruses. This test is testing the system and progressively getting worse, until the system is unable to maintain a certain level of performance or if the system fails to perform. The systems are tested on every aspect individually, this happens because the vulnerability is easily detected. Stress-test is a time-consuming process that is mandatory for creating a functioning system. It proves that the system can work effectively under normal circumstances and also that the system maintains limited functionality even when a large part of it has been compromised.[10]

3.2 Power Virus

The power virus is a form of virus commonly used in stress tests, to test how resilient the system is under extreme conditions. This type of virus is causing maximum CPU power consumption, by executing a machine code-based program. This maximum CPU power consumption causes overheating of the system, and if the system has no

protection with the help of cooling equipment it can cause corruption of data, fault calculations and even system crashes. This may sometimes cause permanent damage to the system. The uses of power virus as I said before are for stress tests, but they can also be used for malicious purposes. Those types of viruses are suitable for benchmarking, integration tests, and thermal tests. [4]

3.3 What is GeST

GeST is an automatic framework for generating stress-tests, this framework is written in Python 3 and takes as inputs xml files for configuration parameters. This tool has 5 basic parts, those parts are the inputs, the outputs, the generic algorithm engine, the measurement, and the evaluation of the measurement. The most important part of GeST is the generic algorithm engine, this GA engine is responsible for selecting the fittest program, the program that has the best qualifications based on the measurement, the engine exchanges genes and produces mutations during the search. The GA engine is inspired by natural evolution. The process goes with generating an initial seed population of assembly instruction programs. The initial generation of programs can be randomly generated or a past generation from previous runs can be used. Next step is the measurement of everyone, in step each individual is compiled and executed, and the metric of choice is run, and the result is placed on the fitness value of each program. Continuing with the procedure, is the creation of the next generation. The fittest programs of the previous generation are selected and set as parents for the creation of the next generation, the instructions of the parents are placed in a crossover process that is basically splitting and resampling the programs together to produce children based on the parent programs , there is also a process named mutation that place a random instruction or operant in a child, this mutation takes place randomly with the help of the mutation rate, that defines how often a mutation can occur. This mutation rate is set by the user before the beginning of the search. The figure below shows how a new generation is created. This procedure is performed until the desired size of the population is reached. The figure illustrates the procedure with simple viruses, so it can be easily understood. Also, the mutation comes and changes either an instruction and replaces it for a different instruction or it changes an operand of an instruction and replaces it with a different operand. When the new generation is created, the same

procedure takes place with evaluating everyone, and finding the fittest individuals in the new generation. After that we have the Inputs, the inputs are reflected by xml files. The main xml files are the configuration file, in this file there are some inputs that must be specified before the search such as the population size, the number of loop instructions, the mutation rate, the type of crossover operation, if elitism (Best individual promoted in the next generation) will be promoted, and the number of parent programs. Furthermore, in this file the instructions and operands used in the GA search will be defined, and numerous other parameters, together with, the directory wherein the results can be stored and the names of the size and health lessons to be used by the GA search. Moving to the measurement and fitness evaluating, there is a measurement file written in python, that includes the process of measurement. This file includes the procedure on how to compile and run the metric of the measurement and then sets a parameter called fitness with the value of the metric, then the value is evaluated by a function, the individuals with the best fitness are then placed as parents for the next generation. And lastly, we have the output of the framework, this section is all the output files that the search generated, all individuals are saved as txt files, with title the generation number, the id of the individual, and the array of measurements. Furthermore, the tool provides a binary file that can be loaded to Python script to get advanced results, like statistical analysis about the fitness value of the fittest individual per generation and instruction mix breakdown of fittest individual per generation. [3]

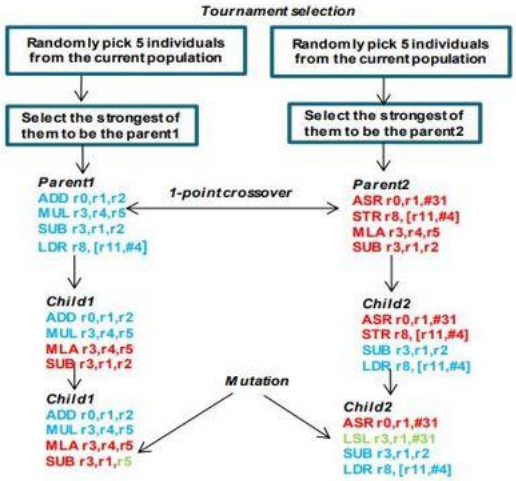


Figure 3.3.1 Crossover and Mutation process

Figure 3.3.1 demonstrates the procedure of crossover and the procedure of mutation. Cross over tis illustrated by the split of parent programs, crossover the instruction

creating new child programs (blue and red colour) and the mutation process with green colour that randomly changes instruction or register in the children's programs.

3.4 Search time

The time that this tool takes to finish a search is a very important aspect. The time is reliant on the measuring process for the fitness value. This measuring process consists of the generation of the program, the compilation of the program, and the process to gather the metrics for the program, all this is multiplied with the total amount of programs generated in the search.
By analysing the GeST's paper, we have 70 generations with a runtime of 7 hours, this concludes that we have 5 gathering of metrics and 2 seconds for the generation and compilation of the program.

3.5 Power Program Search for Stress programs

In this research I used the GeST tool to generate power viruses. I used this tool because it provides for the easy generation of power viruses. It was a very friendly framework. After reading the manual, I customized the configuration file placing in all the necessary places the details that I wanted for my search. The parameters were as follows: population size was 50, loop size was 50, mutation rate for instructions was 0.02, mutation rate for registers was 0.04, mutation rate for register initialization was 0.02, crossover was 1, elitism was true, and the tournament size (how many individuals will be selected) was 5. For measurement I used MeasurementLikwidPower.py, this python file compiles and measures the power consumption of each individual and sets the measurement on the fitness parameter of each virus. With this measurement the search will progress based on how great the power consumption is, giving as individuals with great power consumption. This is very useful in this research because I am investigating power viruses. Having this automatization made all the process of producing great power viruses easy. After the initial run a secondary validation search took place that simply validated the values of power consumption of the first search. After the validation I ended up with 50 generations of 50 power viruses each generation,

a total of 2500 viruses. Each virus had in the title the generation number, the individual id and the power consumption.

Generation of program	Compile	Run	Gathering metrics
0.018 seconds	2 seconds	5 seconds	0.018 seconds

Table 3.5.1 times of processes in normal search

Table3.5.1 shows the steps and their times that a single individual needs when the search is using the common methods to produce a powerful program. With these times, a search needs about 5-7 hours to produce 50-70 generations of 50 individuals each generation.

Chapter 4

Quickgen Method

4.1 Idea

Having in mind that we already know the reference virus, ideas on how to produce the reference virus will be introduced in the future work chapter later in this paper. The Quickgen method is the method that I discovered for producing simple, easy and quickly power viruses. This idea came from the time that a search needed to produce a powerful power virus, due to the demand to compile and gather the power consumption value of each individual. This procedure takes about 0.038 seconds for each virus, multiplying the total number of individuals we come up with a time-consuming procedure. So, my method does not compile either run metric on the individual, it uses the statistical method of Euclidean distance

Generation of program	Gathering metrics
0.036 seconds	0.036 seconds

Table 4.1.1 times of processes using Quickgen Euclidean distance search

Table 4.1.1 shows the steps and their times that an individual needs when the search is using the Quickgen method, each individual needs 0.038 seconds to be generated,

multiplying tis with 50-70 generations of 50 individuals each generation, we have total time of 1.5 – 2 minutes. After all the individuals are generated, I suggest compiling and gather power consumption metrics from 0 to 0.05 Euclidean distance range. Was observed by many attempts that the range that gives the best power consumption was the above. This has a cost; it will take from half to an hour to run the individuals. Using Quickgen method several searches can be done.

Using the features of the powerful program, I placed them into the Euclidean distance formula, using as reference values in the formula the characteristics of the most powerful virus in my search. To find the reference virus features there was a trial-and-error procedure where I was trying many combinations of features and using a chart that included a correlation with Euclidean distance and power consumption, I was looking for the combination of features that gave me the most obvious correlation of the two. Then I used normalized values of each characteristic due to the fact the size of each virus was not fixed. The normalized values were created in two ways, the first way was used for the number of individual type of instructions, i normalize the values by dividing the total amount of the individual type of instruction (eg. scalar instructions) with the total amount of instructions. The second way is used for normalizing the total amount of instructions, and its projected by the following formula: Abs(1-(Total_under_study/ Total_reference)), Abs= absolute value, Total_under_study = total instructions of the individual under study, Total_refernce = total instructions of the reference program. Using the normalized methods, I implemented the normalized characteristics in the Euclidean distance. The characteristics i found out that are more suitable for my search was the following:

- Number of scalar instructions (normalized)
- Number of load or store instructions (normalized)
- Number of scalar load or store instructions (normalized)
- Number of vmultpd instructions (normalized)
- Number of vaddpd instructions (normalized)
- Number of vmaxpd instructions (normalized)
- Number of vsubpd instructions (normalized)
- Number of vxorpd instructions (normalized)
- Normalized total amount of instructions

My method because it uses Euclidean distance needs to have a reference virus to get the characteristics, because of that we skip on how to find the reference virus and we assume that we already have the reference virus.

```
add $858993420,%rsi
vaddpd %ymm8,%ymm7,%ymm6
add %rdi,28(%rsp)
vmulpd %ymm9,%ymm6,%ymm8
vmulpd %ymm1,%ymm8,%ymm6
cmp %rsi,%rdi
vaddpd %ymm8,%ymm10,%ymm4
add $1503238485,%rdi
vmulpd %ymm7,%ymm2,%ymm11
mov 12(%rsp),%rbx
add %rsi,%rdi
mov 192(%rsp),%rdi
mov 256(%rsp),%rdx
mov 320(%rsp),%rdx
add $1646404055,%rdi
mov 192(%rsp),%rdi
mov 256(%rsp),%rdi
mov 320(%rsp),%rdi
mov %rax,92(%rsp)
vmaxpd %ymm5,%ymm10,%ymm6
mov 112(%rsp),%rdx
imul %rdx,%rdx
vaddpd %ymm15,%ymm13,%ymm11
vmulpd %ymm15,%ymm7,%ymm15
vmaxpd %ymm0,%ymm4,%ymm4
vmulpd %ymm15,%ymm3,%ymm12
vaddpd %ymm10,%ymm3,%ymm4
mov 192(%rsp),%rbx
mov 256(%rsp),%rdx
mov 320(%rsp),%rdx
mov 384(%rsp),%rdx
mov 448(%rsp),%rdx
mov 512(%rsp),%rbx
vmulpd %ymm2,%ymm3,%ymm12
vaddpd %ymm3,%ymm3,%ymm9
vmulpd %ymm1,%ymm5,%ymm10
vaddpd %ymm3,%ymm8,%ymm7
vmulpd %ymm13,%ymm10,%ymm7
```

```
add $501079495,%rsi
mov %rdx,96(%rsp)
vmulpd %ymm3,%ymm6,%ymm8
vmulpd %ymm1,%ymm8,%ymm0
cmp %rsi,%rsi
mov 0(%rsp),%rdi
mov 64(%rsp),%rdx
mov 128(%rsp),%rax
add $2004317980,%rdi
vmulpd %ymm8,%ymm13,%ymm11
add %rdi,%rdx
vmaxpd %ymm1,%ymm13,%ymm14
mov 0(%rsp),%rax
mov 64(%rsp),%rdi
mov 128(%rsp),%rdx
vsubpd %ymm1,%ymm2,%ymm15
mov 120(%rsp),%rax
vaddpd %ymm7,%ymm2,%ymm2
vmulpd %ymm15,%ymm8,%ymm14
vmulpd %ymm12,%ymm4,%ymm0
vmaxpd %ymm14,%ymm8,%ymm15
add %rdx,104(%rsp)
vmulpd %ymm12,%ymm0,%ymm6
vsubpd %ymm5,%ymm0,%ymm13
```

Figure 4.1.2 example code of an individual

Figure 4.1.2 shows an example of an individual's code. This code has 23 load or store instructions, 10 scalar instructions, 2 scalar load or store instructions, 14 vmulpd instructions, 7 vaddpd instructions, 4 vmaxpd instructions, 2 vsubpd instructions, 0 vxorpd instructions. And a total of 62 instructions. And the reference virus has the following feature values 20 load or store instructions, 9 scalar instructions, 2 scalar load or store instructions, 14 vmulpd instructions, 9 vaddpd instructions, 4 vmaxpd instructions, 2 vsubpd instructions, 0 vxorpd instructions and total of 60 instructions. The normalized values of the Figure 4.1.2 are the following:

scalar instructions	10/62
load or store instructions	23/62
scalar load or store instructions	2/62
vmultpd instructions	14/62
vaddpd instructions	7/62
vmaxpd instructions	4/62
vsubpd instructions	2/62
vxorpd instructions	0/62
Normalized total amount of instructions	Abs(1-(62/ 60))

Table 4.1.3 Features of code on Figure 4.1.2

The reference virus has the following normalized values:

scalar instructions	9/60
load or store instructions	20/60
scalar load or store instructions	2/60
vmultpd instructions	14/60
vaddpd instructions	9/60
vmaxpd instructions	4/60
vsubpd instructions	2/60
vxorpd instructions	0/60
Normalized total amount of instructions	Abs(1-(60/ 60))

Table 4.1.4 Features of Reference program

Implementing the Table 4.1.3 with Table 4.1.4 in the Euclidean:

(10/62 - 9/60)2+(23/62-20/60)2+ (2/62-2/60)2+ (14/62-14/60)2 +

(7/62+9/60)2+(4/62-4/60)2+(2/62-2/60)2+(0-0)2+(Abs(1-(62/ 60)))2

The Euclidean distance of the above equation is 0.0617

4.2 How it is integrated in my search

Since GeST tool was an easy tool I used to generate the power viruses on the first search, and because I was already familiar with it, I used it to make a search with my method. The fact that GeST is producing all kinds of stress-tests and accepts as measurements files written in python that are written specifically for the method of measurement, it made the job of integration of my method fairly easy. I written a python file that reads each individual's code and collects the amount of, scalar, load or store, scalar load or store, vmulpd, vaddpd, vmaxpd, vsubpd, vxorpd and the total amount of instructions. Using the above measurements the code calculates the normalized values of each measurement, and then with the help of the reference virus characteristics the Euclidean distance is calculated and placed on the fitness of each individual. Using this method the goal is for the individuals to achieve the reference virus characteristics, this can be possible because the GeST algorithm is sorting individuals with the Euclidean distance, the closer the characteristics are with those of the reference virus, fittest the individual. The zero value means the individual has all the characteristics that the reference virus has.

4.3 Possible Ways to Use QuickGen Method

4.3.1 Only Euclidean Distance

There are a couple of ways to implement my method, the first way is only by using the Euclidean distance. This way is trying to create a virus that is similar with the reference virus, with only using the Euclidean distance. In the configure file I fill all the necessary features of the search as with the values i already described in the above subchapter. For the measurement, the MeasurementEuclidiandistance.py file is used. This file contains the code responsible for calculating the Euclidean distance of the individual and placing the value on the fitness parameter so it can be evaluated by the GeST tool. After the search, I validate all the viruses and see if the Euclidean distance projected in the title of each individual matches the Euclidean distance I produce. After the validation process i measure the power consumption of all viruses using a script, this step is made to find if the search produced a powerful enough virus.

4.3.2 Hybrid

The other way of implementing the method into GeST, is by introducing a Hybrid method. This hybrid method generates viruses for the N generation using Euclidean Distance and for the other M generations we use power consumption. There are several ways of implementing the method:

- First Euclidean distance then power consumption
- Random
- First power consumption then Euclidean distance

In the configure file I fill all the necessary features of the search as with the values i already described in the above subchapter. For the measurement, the MeasurementEuclidianLikwid.py file is used. In this file there are 3 parameters you have to fill during at the beginning of the run, how many generations will be run by Euclidean distance, how many generations will be run by power consumption and which of the 3 ways will be used for this search (First Euclidean distance then power consumption, Random, First power consumption then Euclidean distance). Every one of the 3 ways has a purpose. The First Euclidean distance then power consumption, has the idea of first running the Euclidean distance with a reference virus in mind and getting the programs really close to the reference virus, and then with the help of power consumption metric to be able to finish off the search. This is basically getting the programs that are already near an already existing powerful virus and sort them by their actual power consumption producing the best power virus possible. The First power consumption, then Euclidean distance, has a general idea that calculating the power consumption firstly for some generations, the viruses will be to a point that they are already getting into shape for what makes them powerful. Before the search continues with the euclidean distance we can get the characteristic out of the best virus that the power consumption process produced and integrate them as the characteristics of the reference virus. The Random process has the ability to make unpredictable searches, the idea behind random is that it can produce a very powerful virus due to the random procedure. After the end of the search, it will be a validation process to be sure that the results of the Hybrid process can be trusted, the validation process will take process on both Euclidean distance and power consumption.

Chapter 5

Experimental Methodology

5.1 Hardware Configuration

5.2 GeST and viruses

5.3 IPC, Power Measurements and Validation

5.4 Instruction mix

5.5 Validation effort

5.1 Hardware Configuration

This research is based on a single machine, the machine is a HP compaq 8200 elite cmt pc, this pc has the Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz processor. This processor consists of 4 cores, with base frequency 3.10 GHz and TDP of 85 Watts. In this research I stabilize the frequency using the cpupower tool, this tool enables the operating system to scale the CPU frequency up or down to save power or improve performance. Scaling can be done automatically in response to system load, adjust itself in response to ACPI events, or be manually changed by user space programs. ("CPU frequency scaling - ArchWiki"). The processor has 6MB of cache and bus speed of 5 GT/s. Going to the microarchitecture of the processor we can observe that the L1 instruction cache can accept 16 bytes/cycle, also there is a 4-way decode. Moreover, this microarchitecture has 6 execution units. For all experiments I disable c states and p states to remove variations. Also mention that the generated viruses only stress the CPU, I did not configure other parts of the system like DRAM in terms of frequency.

5.2: GeST and viruses

As I already described, I used GeST tool for all my searches. The configurations were for all the searches the same; the only parameter changing was the file that is measuring the fitness of each virus. There were 3 main measuring files, the first file is measuring the fitness of each virus by compiling and running the virus and gathering the result of the power consumption. The second file is measuring fitness by calculating the Euclidean distance of each virus with the help of a reference virus (a virus that we already know achieves maximum power consumption). The third file measures the fitness with a hybrid method, we generate viruses for the N generation using Euclidean Distance and for the other M generations we use power consumption. And with 3 different orders to choose from, first Euclidean distance then power consumption, random, first power consumption and then Euclidean distance. The measurements of power consumption are done by running the virus for 5 seconds and then gathering the power consumption of the computer. In the Figure 3.5.1 we can see the table of the procedures and their times that a normal GeST search is discarded. In the Figure 4.1.1 we have the table of the procedures and their times that Quickgen method is using for the search. In the two tables we can see that Quickgen lacks the compile and run procedures, which are the most time-consuming parts.

5.3: IPC, Power Measurements

This research is based on a lot of metrics related to the CPU; these metrics are important to understand what impact a program has to the processor. The IPC is a metric that the compilation is mandatory because it shows how fast the processor runs the program. The method that I used for calculating the IPC was, compiling the program, and using perf tool which is a command used as a primary interface to the Linux kernel performance monitoring capabilities and can record CPU performance counters and trace points [2]. Power consumption was the basic metric to determine if a program was performing good, because this research is about creating power viruses. I used power consumption in various situations in my research, the method I used to measure it was to compile the program, and then I measured the value by running likwid-powermeter which is a tool to print power and clocking information on Intel CPUs.

5.4 Instruction mix

Based on the previous details, I based this research in a program that is producing maximum power to the system. This program is broken down into characteristics to find out what components are producing the power consumption. During this process I found out that the instruction mix is the key for producing a powerful program. To figure out the instruction mix of each program I wrote scripts in python that break down each virus. The first script gets the total number of instructions, the second script get the total amount of scalar, load or store, scalar load or store and vector instruction of each virus, and the third script brakes down the vector instructions to total of vmulpd, vaddpd, vmaxpd, vsubpd and vxorpd instructions. Breaking down the program was a very effective way to figure out what makes a powerful individual and set the right characteristics for my reference program.

5.5 Validation effort

Validation was one of the most crucial parts of my research. I put a lot of effort and time validating the methods used to gather values and metrics. The main validation was made in the methods used to gather power consumption metrics and IPC metrics, because in some instances when I was using a script that automatically collects power consumption, or IPC I observed some outliers, and when I investigate the values, I run them manually I got different results. This caught my attention, and I placed a serious amount of time validating the results again and again, and reading some works done on the topic [13]. During this process I found out that during all automatic data gathering is normal to see some alternations on the data. What I did to lower this alternation was to run the script that gather power consumption and IPC 5 times, for the 5 values find the minimum and maximum value and get rid of them, and for the 3 remaining values calculate the average value, the value of this process gives as an accurate measurement.

Chapter 6

Results

6.1 Extraction of parameters for Euclidean Distance
6.2 Optimizing Search based on power vs Optimizing Search with Euclidean distance
6.3 Results of Hybrid method

6.1 Extraction of parameters for Euclidean Distance

One of the fundamental stages of my thesis work was to find the characteristics of a powerful program. The idea was to break apart the virus on different characteristics and investigate them. The first characteristic I investigated was the size in bytes of every individual, for comparison I made a graph with x-axis the size in bytes and in y-axis the power consumption of each individual in watts.

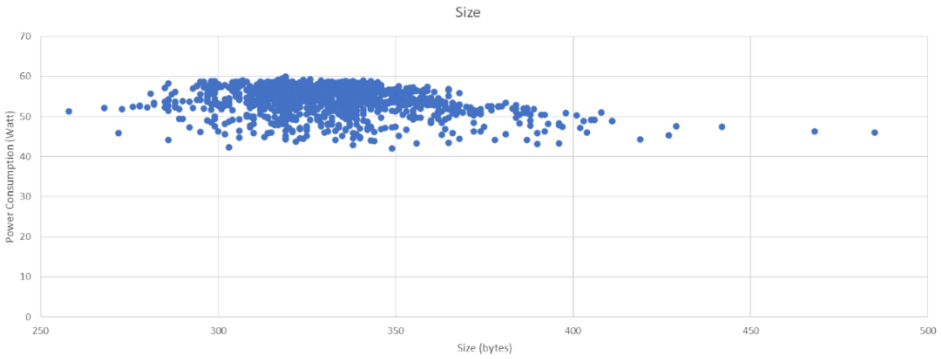


Figure 6.1.1 Graph of Power consumption – size (Bytes) of an individual

Figure 6.1.1 we observe that the area we observe in a powerful program is from 290 - 340 bytes. But in this area, we can see that there are a lot of programs that have very

low power consumption as well. This is because a powerful program does not depend only on the size.

Moving on, the next characteristic that investigated was the IPC of each individual, that was made to get an idea on what is going on during the execution and if the IPC has a correlation with the power consumption of each individual. Also, IPC shows how good the instructions are combined together to create a run that is taking advantage of all the parts of the microarchitecture.

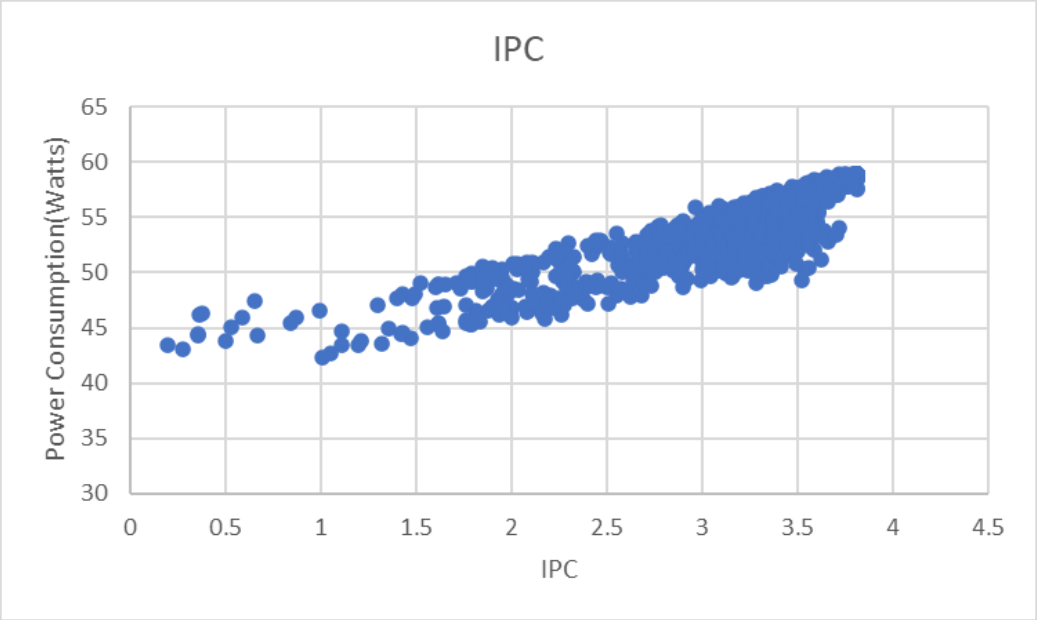


Figure 6.1.2 Graph of Power consumption – IPC

Figure 6.1.2 we can clearly see that IPC has a correlation with power consumption. The higher the IPC the higher the power consumption. This observation is very crucial for the understanding of a powerful individual.

Using the above remark, I proceeded on breaking the individuals apart and investigating their instruction mix. Firstly, I divided the individual into four instructions, vector, scalar, load or store, and scalar load or store instructions.

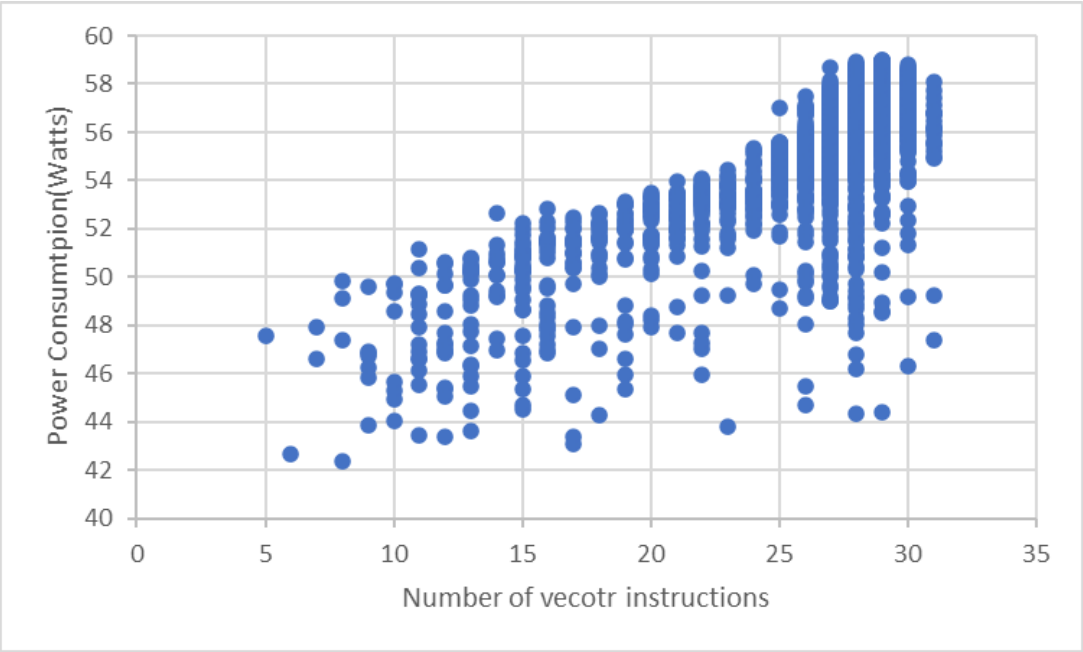


Figure 6.1.3 Graph of Power consumption – Number of vector instructions

Figure 6.1.3 represents the number of vector instructions in an individual program as in proportion the power consumption that produces. Observing the figure we can see the power consumption peaking on 29 vector instructions. Due to the fact that the program depends on other commands to be able to produce high power , we observe a wide range of values, but because we are looking for the strongest program we can ignore it and dedicate ourselves to the highest power consumption.

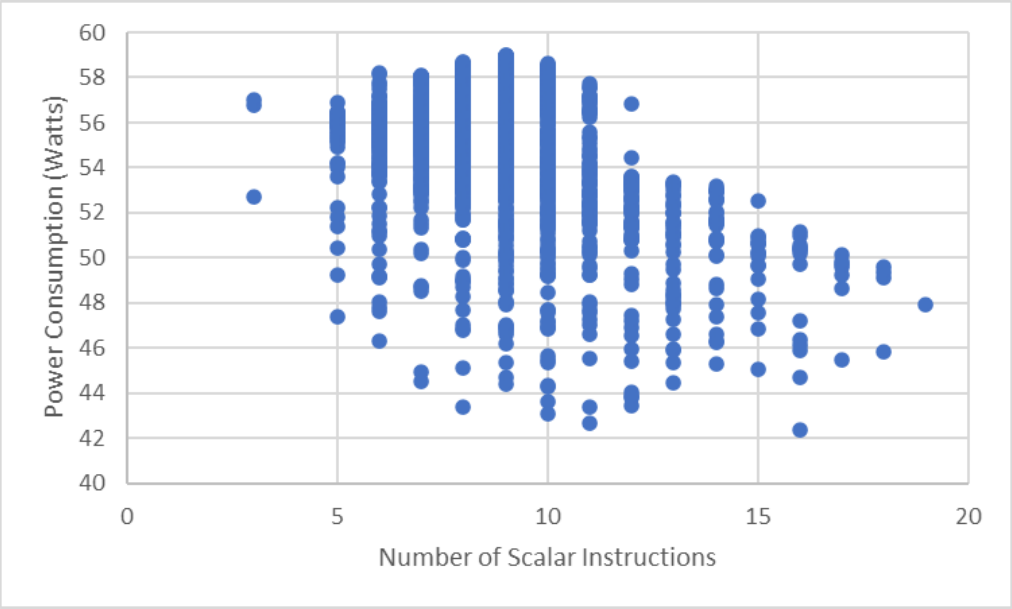


Figure 6.1.4 Graph of Power consumption – Number of scalar instructions

Figure 6.1.4 represents the number of scalar instructions in an individual program as in proportion the power consumption that produces. Observing the figure we can see the power consumption peaking on 9 scalar instructions. Also we observe that as the number of scalar instructions exceeds 10 the power consumption decreases.

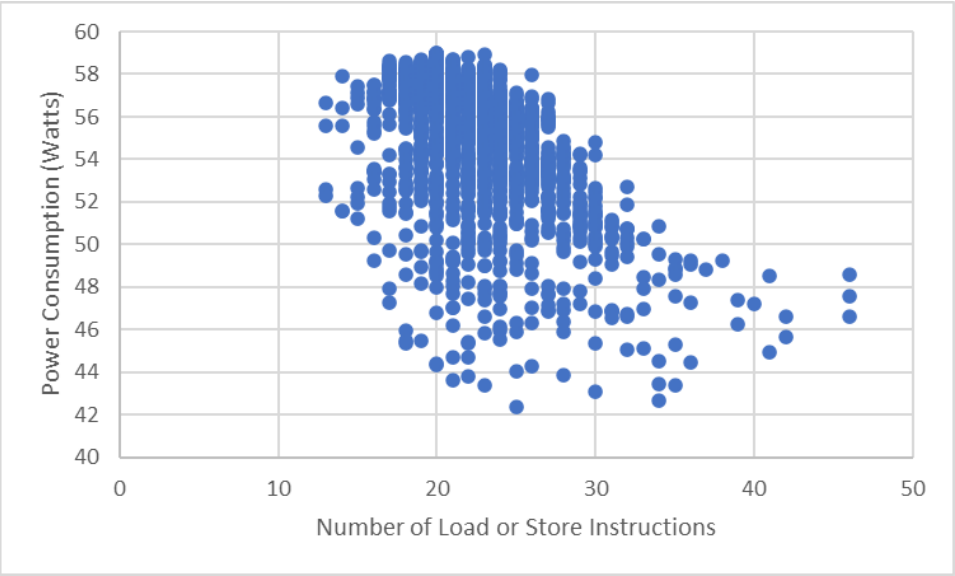


Figure 6.1.5 Graph of Power consumption – Number of Load or Store Instructions

Figure 6.1.5 represents the number of load or store instructions in an individual program as in proportion the power consumption that produces. Observing the figure we can see the power consumption peaking on 20 load or store instructions. When the number of load or store instructions gets too big the power consumption decreases, this is due to the fact that there are not enough processing ports for load or store instructions to process so many instructions and stalls appear.

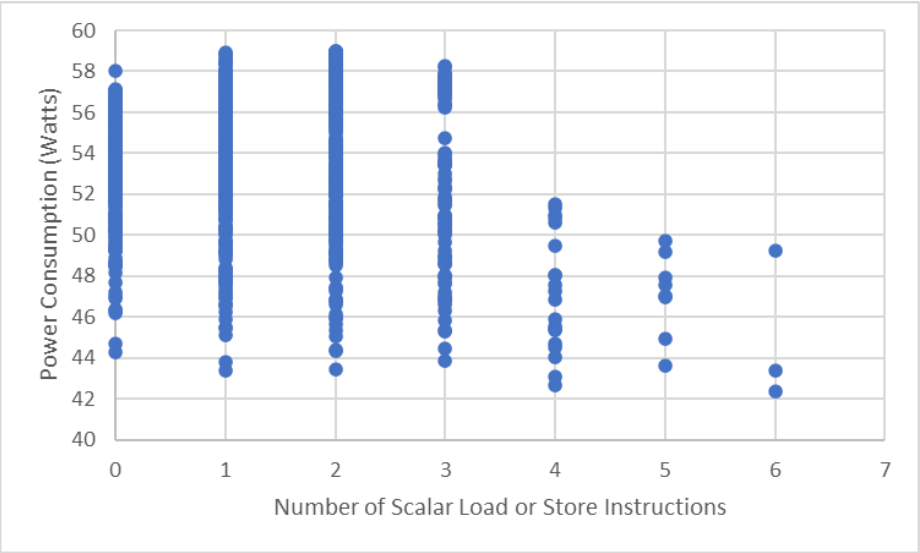


Figure 6.1.6 Graph of Power consumption – Number of Scalar load or store instructions

Figure 6.1.6 represents the number of scalar load or store instructions in an individual program as in proportion the power consumption that produces. Observing the figure we can see the power consumption peaking on 2 scalar load or store instructions. When the number of load or store instructions gets over 3 the power consumption decreases sharply.

The next step is to break the vector instructions further down into more specific instructions, the GeST framework has 5 vector instructions in the instruction list. Those instructions are vmulpd, vaddpd, vsubpd, vmaxpd, vxorpd. Considering the list, I made a script that counts the individual vector instructions of each program. This will help evaluate further what makes a powerful virus.

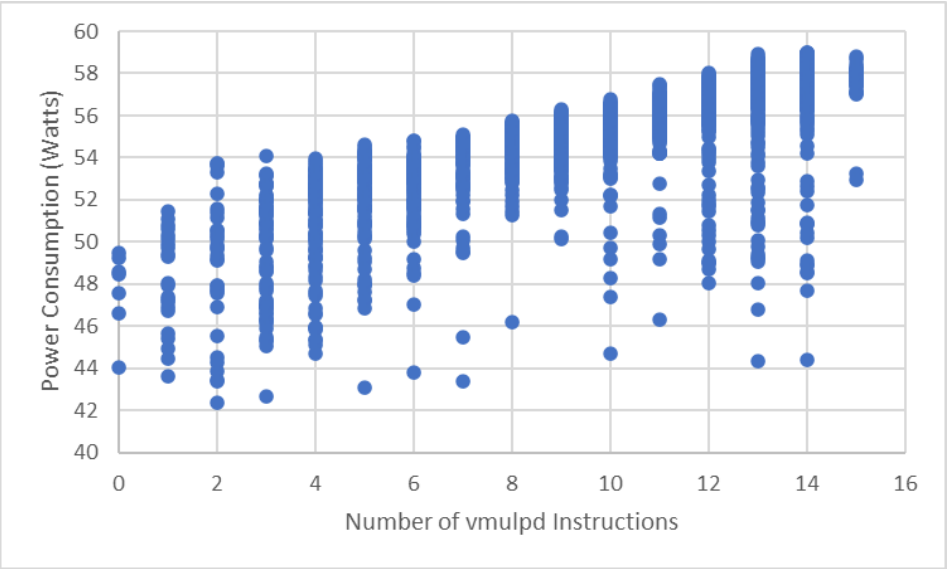


Figure 6.1.7 Graph of Power consumption – Number of vmulpd

Figure 6.1.7 represents the number of vmulpd instructions in an individual program as in proportion the power consumption that produces. Observing the figure we can see the power consumption peaking at 14 vmulpd. We can observe from the figure that vmulpd is an essential instruction to produce a powerful program.

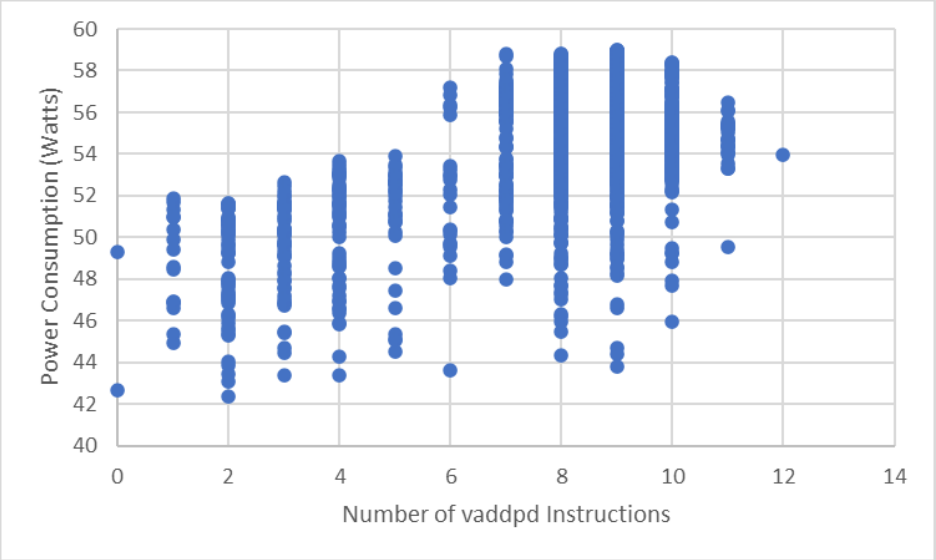


Figure 6.1.8 Graph of Power consumption – Number of vaddpd

Figure 6.1.8 represents the number of vaddpd instructions in an individual program as in proportion the power consumption that produces. Observing the figure we can see the power consumption peaking at 9 vaddpd. We can observe that between 7 to 9 vaddpd

instructions is the right amount of instructions to have a powerful program, when the number goes below 7 or above 9 the power consumption decreases.

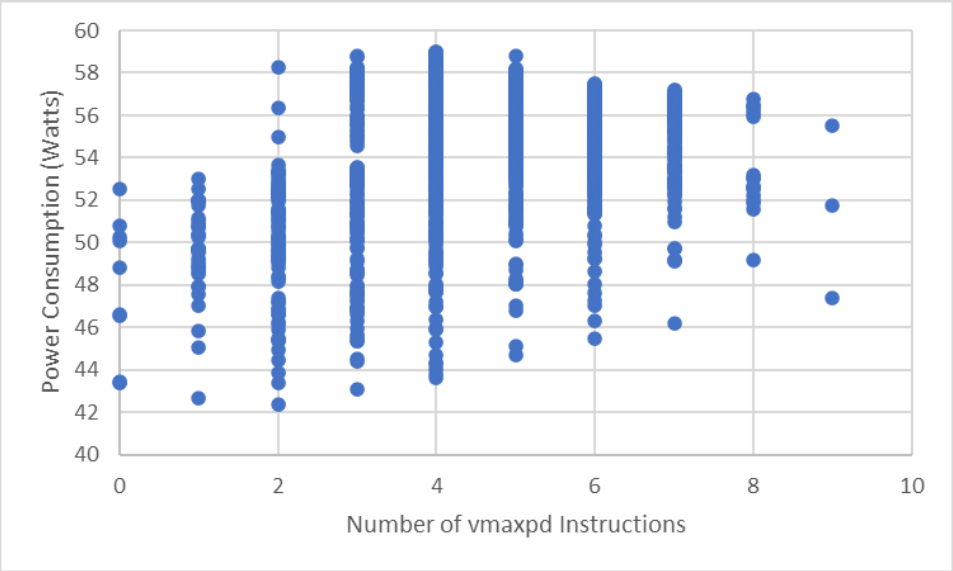


Figure 6.1.9 Graph of Power consumption – Number of vmaxpd

Figure 6.1.9 represents the number of vmaxpd instructions in an individual program as in proportion to the power consumption that produces. Observing the figure we can see 3 to 5 vmaxpd instructions are producing a powerful virus, with 4 instructions giving the best results.

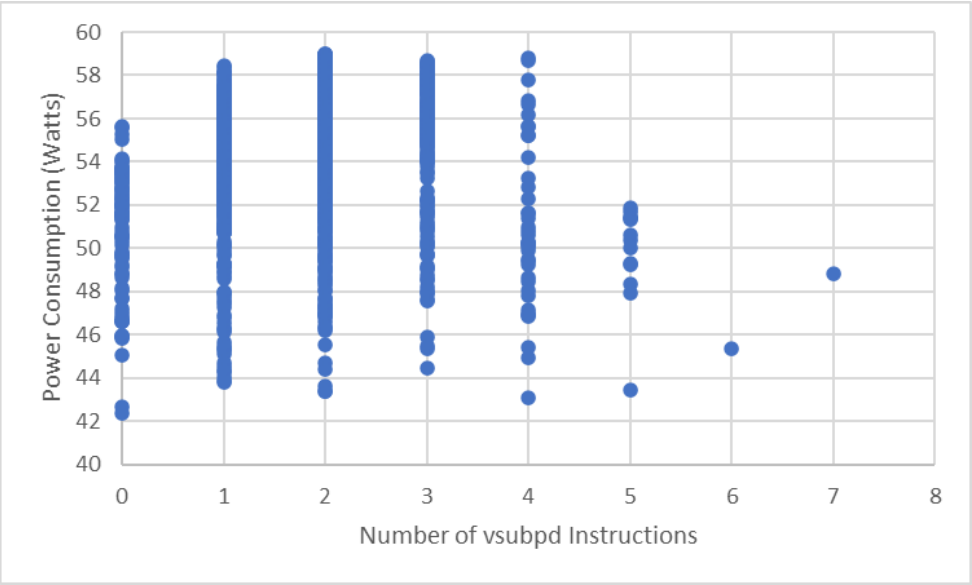


Figure 6.1.10 Graph of Power consumption – Number of vsubpd

Figure 6.1.10 represents the number of vsubpd instructions in an individual program as in proportion to the power consumption that produces. Observing the figure we can see 1 to 4 vsubpd instructions are producing powerful viruses, with 2 instructions being the optimal amount.

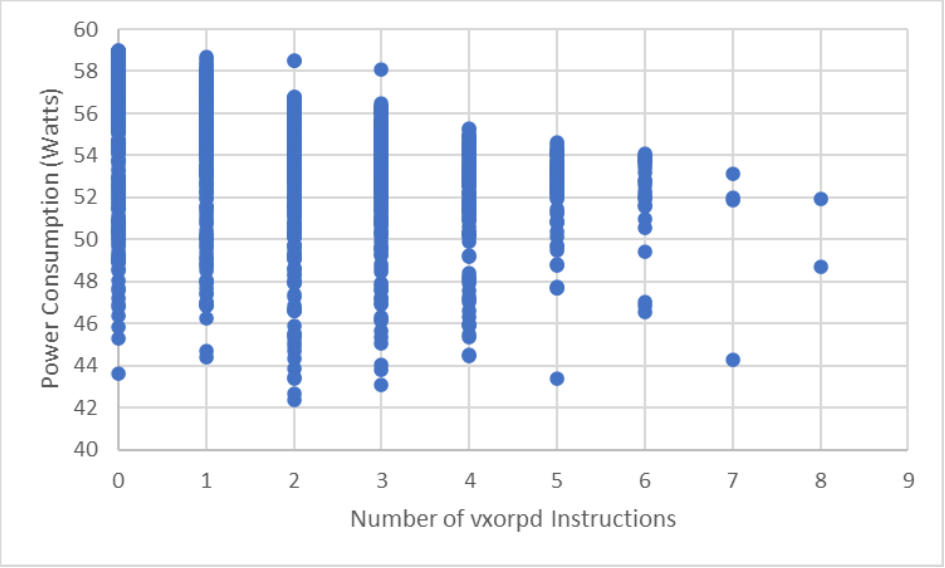


Figure 6.1.11 Graph of Power consumption – Number of vxorpd

Figure 6.1.11 represents the number of vxorpd instructions in an individual program as in proportion to the power consumption that produces. Observing the figure we can

clearly see the less of vxorpd instructions the better. As the amount of vxor instruction gets bigger the power consumption decreases.

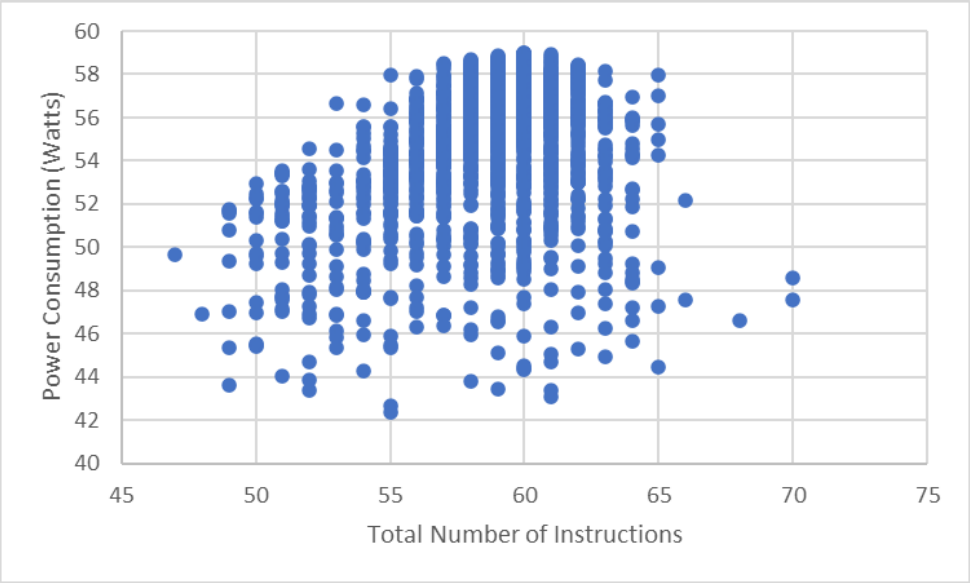


Figure 6.1.12 Graph of Power consumption – Total number of instructions

Figure 6.1.12 represents the total amount of instructions in correlation with the power consumption of each program. From the Figure 6.1.12 we can observe that from the range of 57 to 62 total amount of instructions we have programs with great power consumption. The peak of can be found at 60 instructions.

Analysing all the above figures, we extracted all the necessary characteristics that make up a powerful program. Based on the characteristics, a reference virus was built, this reference virus has the following characteristics:

scalar instructions	9
load or store instructions	20
scalar load or store instructions	2
vmultpd instructions	14
vaddpd instructions	9
vmaxpd instructions	4
vsubpd instructions	2
vxorpd instructions	0
Normalized total amount of instructions	60

Table 6.1.1 Features of the reference program

This reference virus has the exact number of instructions that the program with maximum power consumption. So, we based this values and features as the reference program that we will be using to create our Quickgen method.

6.2 Optimizing Search based on power vs Optimizing Search with Euclidean distance

Using the above characteristics of the reference program, I continued by normalizing all the values of the reference program, and then doing the same process for the rest of the programs. The normalization process took place because of the variable size of programs, so normalizing and implementing the values as percentages would give us better results during the Euclidean distance. The method of normalization is already described in chapter 5.1. The instructions were normalized by dividing the total amount of the individual type of instruction with the total amount of instructions. The total amount of instructions, and its projected by the following formula: Abs(1-(Total_under_study/ Total_reference)), Abs= absolute value, Total_under_study = total instructions of the individual under study, Total_refernce = total instructions of the reference program. The normalized values are then implemented into the formula of Euclidean distance, and the value of the Euclidean distance is calculated. The formula of the Euclidean distance is the following

Now having all the necessary tools to start the search that uses Euclidean Distance for the fitness metric. After the search I ended up with the following results.

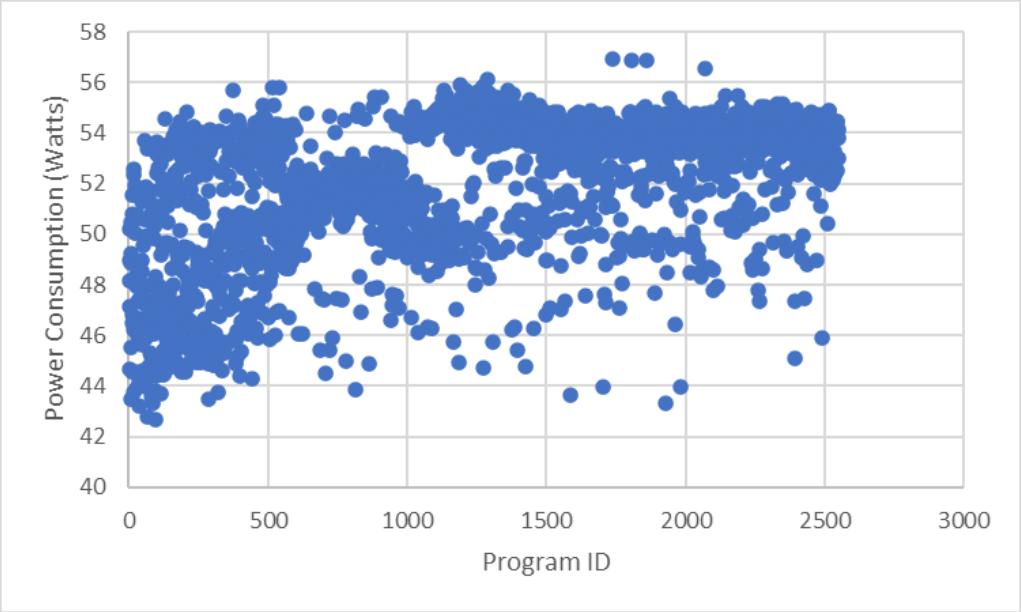


Figure 6.2.1 Graph of Power consumption – Individual ID of Quickgen search

Figure 6.2.1 we can see the power consumption that the Euclidean Distance search produced using the Quickgen method. The max power consumption stands at 56.94 Watts with ID 1737, with Euclidean distance of 0.02357 and a list of characteristics:

scalar instructions	9
load or store instructions	20
scalar load or store instructions	2
vmultpd instructions	14
vaddpd instructions	9
vmaxpd instructions	4
vsubpd instructions	2
vxorpd instructions	0
Normalized total amount of instructions	60

Table 6.2.1 Features of the best virus of Quickgen method using only Euclidean Distance

As we saw earlier in the 6.1 chapter a lot of viruses for the first search had high power consumption even if the values were not exactly the same as the program we set as

reference. We saw each instruction had an effective range where each instruction was scoring significant high-power consumption. This program has 2 main differences with the reference virus. The differences are the total of vaddpd instructions in this case a total of 8 instead of 9, and the presence of vxorpd instructions instead of no vxorpd instruction in the reference program. Both references are found within the range where we had powerful programs. Figure 6.1.8 and Figure 6.1.11 .

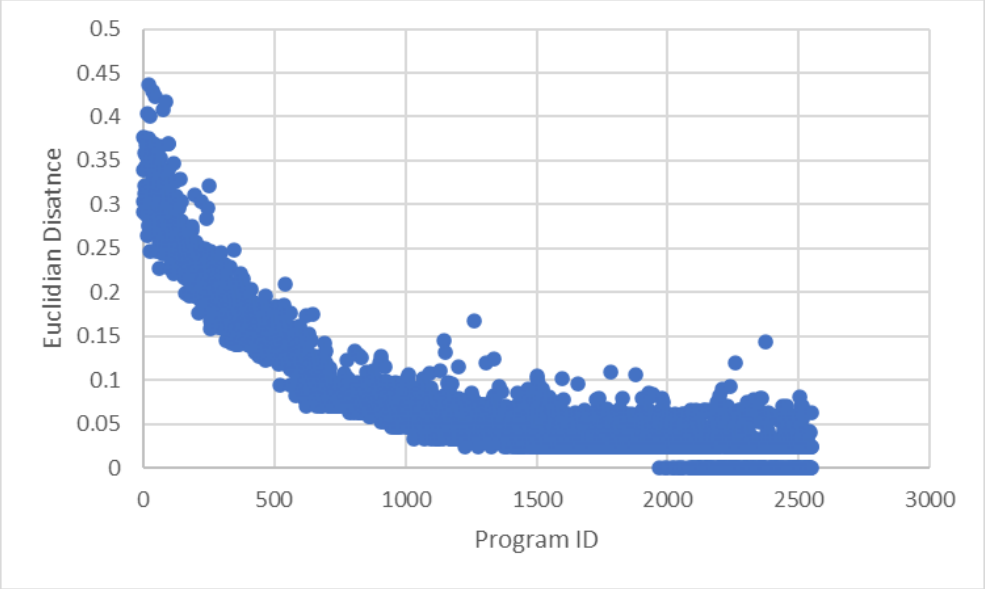


Figure 6.2.2 Graph of Euclidean Distance – Individual ID of Quickgen search

Figure 6.2.2 showcases the Euclidean distance in comparison with the Program ID using the Quickgen method, so we can clearly see the Euclidean distance decreasing. From this we can see that the fitness evaluation is working nicely, and it distributes the values correctly, seeing progress clearly.

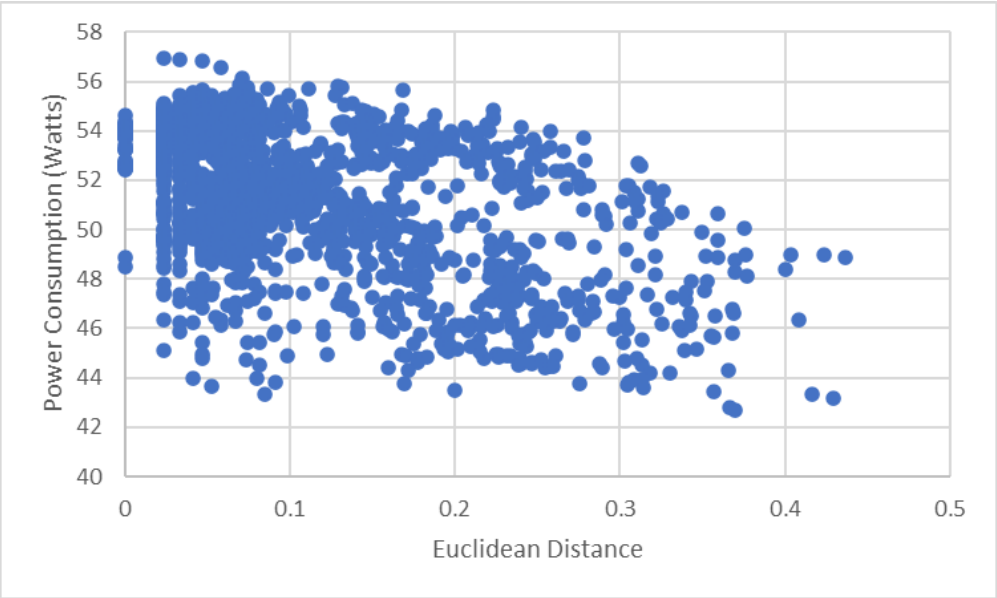


Figure 6.2.3 Graph of Power consumption – Euclidean Distance of Quickgen search

The Figure 6.2.3 illustrates the power consumption in comparison of the Euclidean Distance using the Quickgen method. We observe that the pattern of progression is not very visible. There are a lot of programs with low euclidean producing low power consumption, this happens because the reference program might have some features that are not included in the features that I used during the search. Some features that can make a significant change a produce a more powerful program are, the RAW dependencies, to break the scalar instruction into individual instructions like we did for vector commands (eg. add and mul instructions to be counted separately). We also observe from this figure that the program with maximum power consumption does not have 0 Euclidean distance, also the program with 0 Euclidean distance has a maximum power of 54.61 Watts. This is also explained by the above.

What would I suggest as the author of the Quickgen method? Since the process takes only 1.5 minutes to finish, then get the programs with Euclidean distance from 0 to 0.05, compile, run them and then gather the metric of power consumption. Then by sorting the power consumption, get the program with the maximum result.

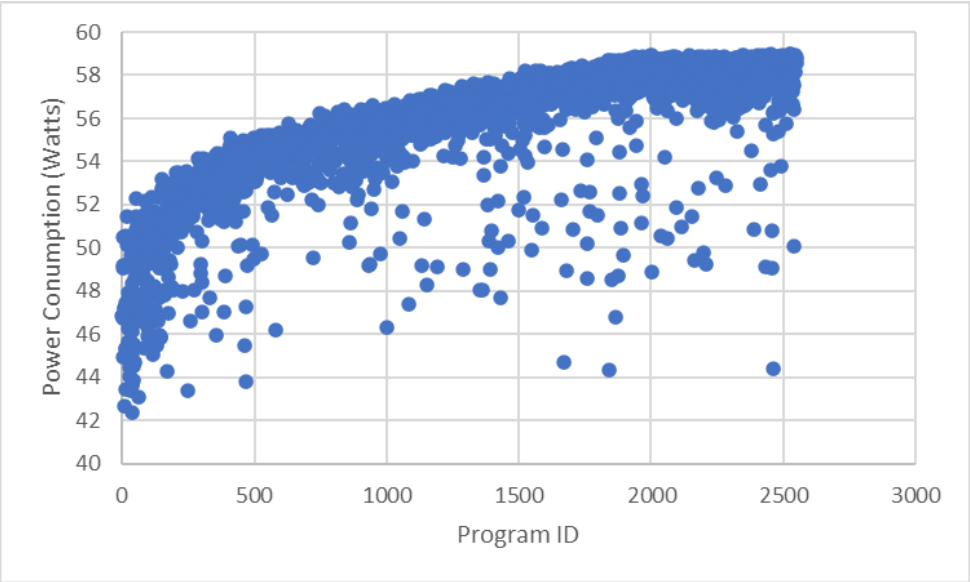


Figure 6.2.4 Graph of Power consumption – Individual ID of Normal Search

Figure 6.2.4 showcases the Power consumption in comparison with the Program ID using the normal method, so we can clearly see the power consumption has an ascending course. From this we can see that the program with ID 2475 has the maximum power consumption of 58.94 Watts.

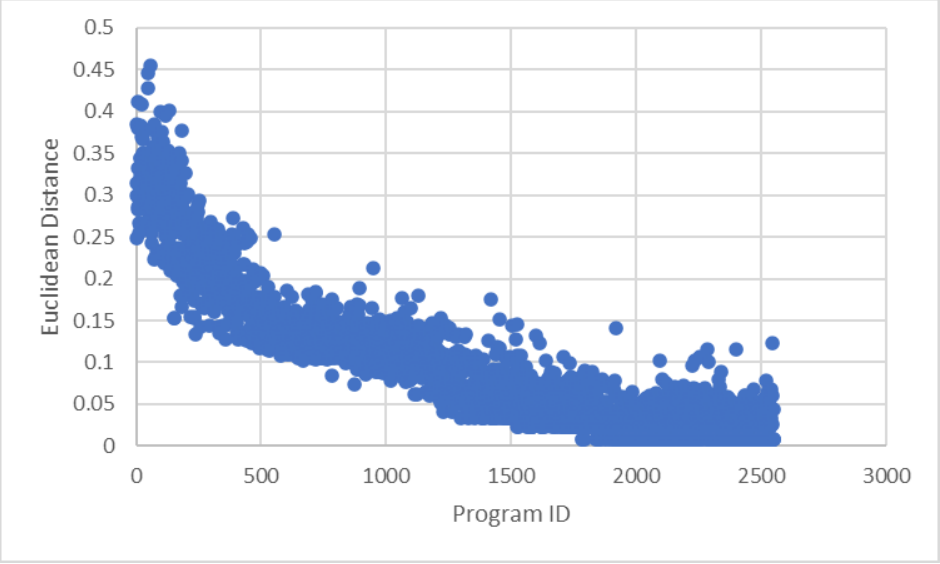


Figure 6.2.5 Graph of Euclidean Distance – Individual ID of Normal Search

The Figure 6.2.5 showcases the Euclidean distance in comparison with the Program ID using the normal method. The normal method does not use Euclidean Distance to

evaluate the fitness but in the above figure we see that the Euclidean distance is decreasing, this shows us that the Euclidean distance we are using is indeed correlated on how the program progresses in the normal method. The reference program feature values that were used to calculate the above euclidean distance, are the following:

scalar instructions	9
load or store instructions	20
scalar load or store instructions	2
vmultpd instructions	14
vaddpd instructions	8
vmaxpd instructions	4
vsubpd instructions	2
vxorpd instructions	1
Normalized total amount of instructions	60

Table 6.2.2

Table 6.2.2 the Features of the best virus using Normal search

All the numbers are then normalized with the method discussed in chapter 4.1. After the normalization process the values are implemented into the Euclidean distance formula, an example also illustrated in chapter 4.1, and the results of the formula are representing the Euclidean distance of the program under study from the reference program

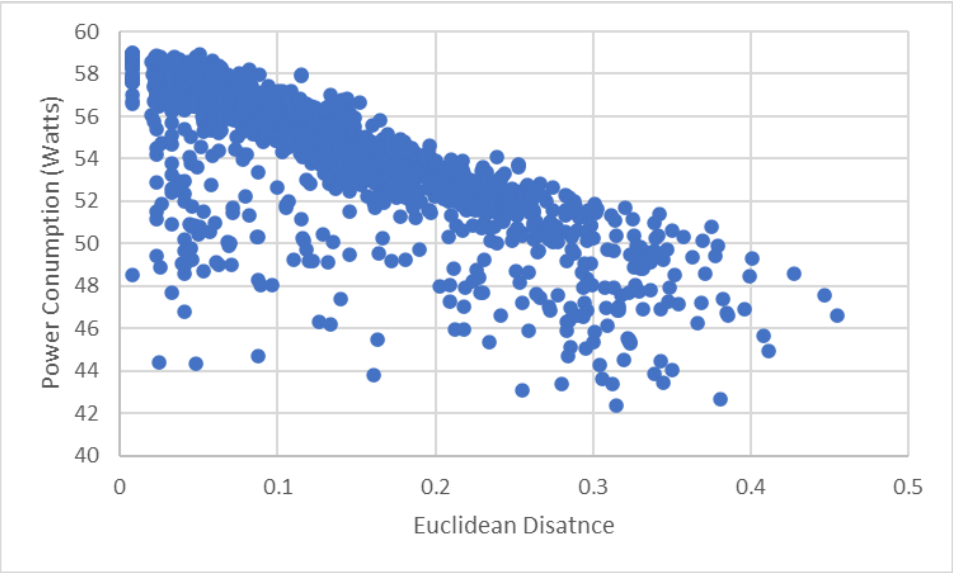


Figure 6.2.6 Graph of Power consumption – Euclidean Distance of Normal Search

The Figure 6.2.6 illustrates the power consumption in comparison of the Euclidean Distance using the normal method. We observe a visible pattern of progression but there are some values that are out of this pattern. The visible pattern shows us that there is a correlation between power consumption and the Euclidean distance we used. The features of the reference virus are discussed again in chapter 4.1 with also the normalization process.

Comparing Figure 6.2.3 with Figure 6.2.6, we come to the conclusion that the normal method is achieving a greater result than the Quickgen method. The reason behind this might not be a direct fault on the Quickgen method, because this method is achieving the optimal result which is the 0 Euclidean distance. The fault is what features are used to create the powerful virus. So, if all the features that compose a powerful program have been found the Quickgen method can achieve such a result using the Euclidean distance. The best result of the normal method was 58.94 Watts in comparison with the Quickgen method that produced the maximum result of 56.94. Comparing the results, the normal method produced a virus that is consuming 2 Watts more than the Quickgen method, but the Quickgen method is 1/360 of the time faster than the normal version.

Note: After evaluating the reference program of the normal search, and the programs with 0 Euclidean distance, I concluded that scalar instruction must be broken further

down into individual instructions like I did with vector instructions. So, I created a new reference program that looks like this

add instructions	7
cmp instructions	1
imul instructions	1
load or store instructions	20
scalar load or store instructions	2
vmultpd instructions	14
vaddpd instructions	8
vmaxpd instructions	4
vsubpd instructions	2
vxorpd instructions	1
Normalized total amount of instructions	60

Table 6.2.7 shows the new features of the used for the new search of Quickgen method

After building the new reference program I implemented it into GeST and I completed a search using the new features and their values. This search produced the following graph

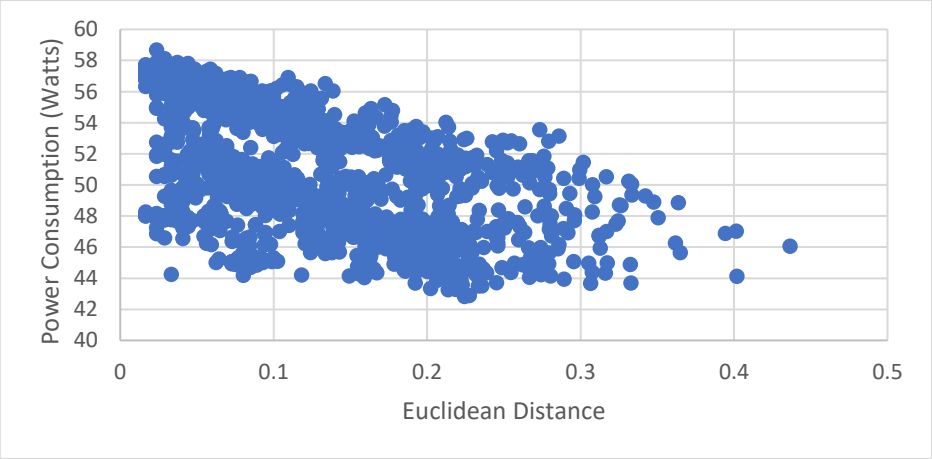


Figure 6.2.8 Graph of Power consumption – Euclidean Distance of Quickgen Search with new features

The Figure 6.2.8 illustrates the graph between Euclidean distance and Power consumption. We can clearly see that using the new Euclidean we have better results from those in 6.2.3. The most powerful program has a power consumption of 58.67

Euclidean distance of 0.02357. This gives us a program that in power competes with the reference virus, they have only 0.27 Watts difference. So this concludes that using the right features and the right values we can produce a powerful program in only 1.5 minutes.

6.3 Results of Hybrid method

In this chapter I will showcase the results of Quickgen Hybrid method, in this hybrid method I used the Euclidean distance for the first 40 generations and then for the last 10 generations I used normal search, that includes generation of the individual, compile the program, and run and gather the power consumption of the program. The idea behind this hybrid method is to produce with the Euclidean a program very similar to the reference program that we set, because the individuals would be very close to a powerful program, the power consumption will have an easier job producing a powerful individual.

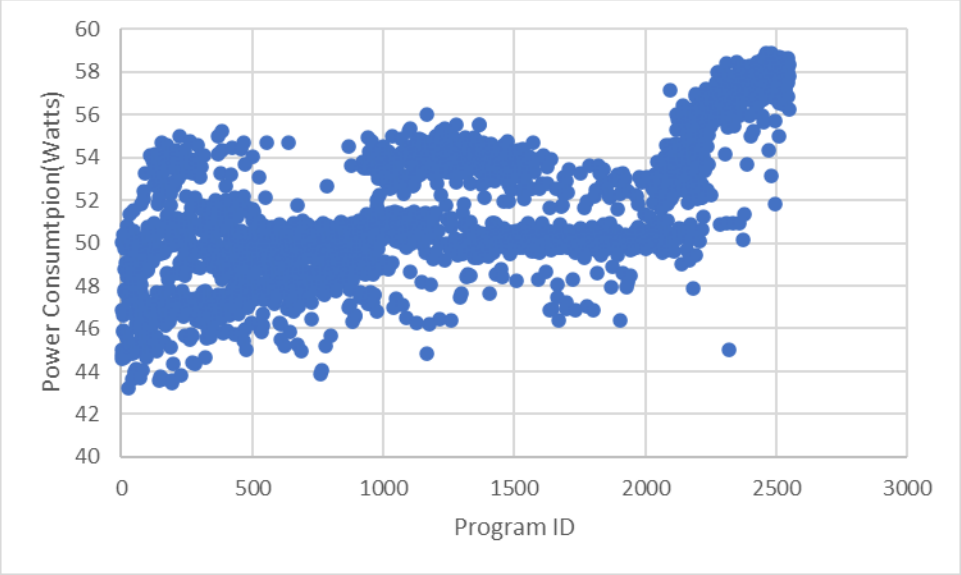


Figure 6.3.1 Graph of Power consumption – Individual ID of Quickgen Hybrid Search

Figure 6.3.1 showcases the Power consumption in comparison with the Program ID using the Quickgen Hybrid method. In the above figure we observe that the power consumption for the first 40 generation has a lot of alternations, when the power consumption is set, we clearly see a sudden upward trend. The Quickgen Hybrid

method produced a powerful individual with the id of 2411 and with power consumption 58.91 Watts.

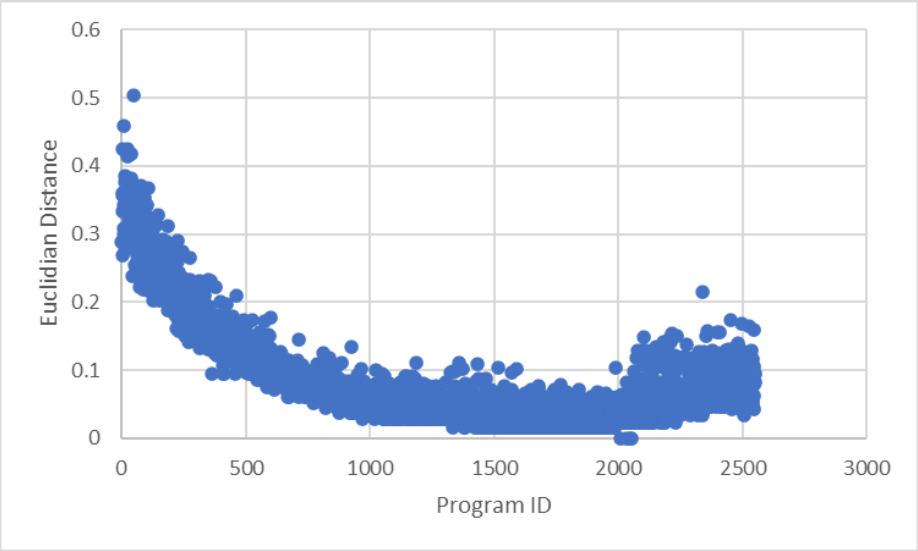


Figure 6.3.2 Graph of Euclidean distance – Individual ID of Quickgen Hybrid Search

The Figure 6.3.2 showcases the Euclidean distance in comparison with the Program ID using the Quickgen Hybrid method. We observe that the Euclidean distance has a steady downward trend until the 2000th individual, and then until the end of the search we see an upward trend, this shows that the Euclidean distance after the implementation of the power consumption as the main metric at the search is not followed.

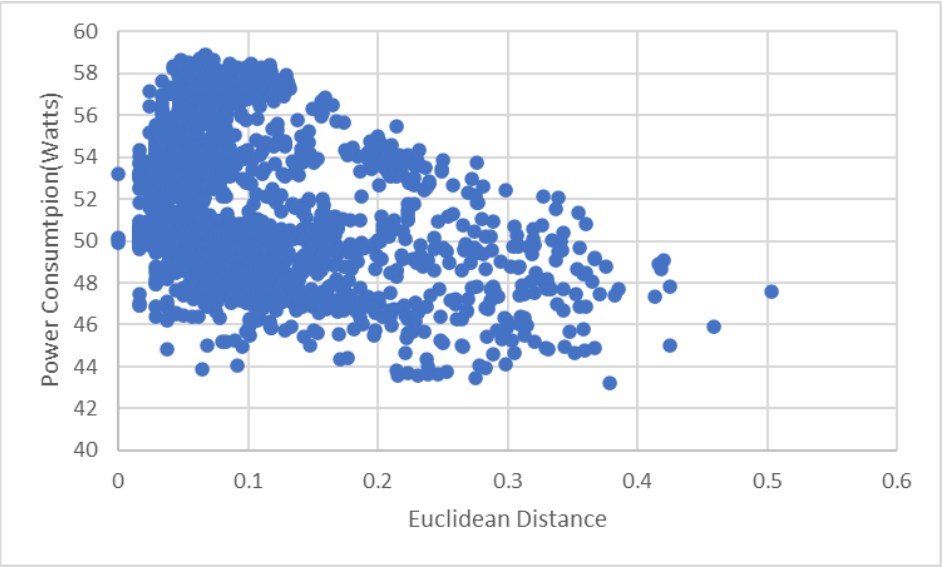


Figure 6.3.3 Graph of Power consumption –Euclidean distance of Quickgen Hybrid Search

The Figure 6.3.3 illustrates the power consumption in comparison of the Euclidean Distance using the Quickgen Hybrid method. We observe a scatter diagram with progression. We also observe that the best values of power consumption are placed within the Euclidean distance of 0.04 and 0.1.

Comparing Figure 6.3.3 with Figure 6.2.6, we conclude that the normal method is producing similar results with Quickgen Hybrid method. The normal method produced a program achieving 58.94 Watts and similarly the Quickgen Hybrid method produced a program with 58.91 Watts. The normal method achieved this with aproximently 5 hours of computation time, in contrast Quickgen Hybrid method achieved the result in only one hour of computation time. Using the Hybrid method we produce the same result with only 1/5 of the time.

Chapter 7

Related Work

Related Work

This research had as reference some related work. The major work that I based a lot of my research on is GeST an automatic framework for generating stress-test, I already descripted in detail all the aspects of GeST in chapter 3 [3]. Some other closely related work is System-level Max Power (SYMPO) – A systematic Approach for Escalating System-level Power Consumption using Synthetic Benchmarks [12]. SYMPO is a framework to automatically generate system level max-power viruses for a given machine configuration. This framework uses x86 architecture as well for creating reasonably good power viruses for any given microarchitecture within a few hours.

Chapter 8

Future Work

8.1 Reference program

8.1.1 Using Quickgen Hybrid Method

8.1.2 Using Microarchitecture

8.2 Features of reference virus

8.3 Conclusion

8.1 Reference program

As I already mentioned in chapter 4, we are assuming that we already know the reference program, the features that makes the program powerful and their values. But we should investigate methods to find the reference program and the features before or during the search without investing too much time in this process. I have some ideas on how this can be done.

8.1.1 Using Quickgen Hybrid method

The idea behind using the Quickgen Hybrid method is to use the advantage that this method gives us to be able to run power consumption metrics during the search. Knowing the features that the reference virus needs to be powerful we can begin the search using power consumption metrics and then after some generations a basic pattern on what it takes to make a powerful virus would form. Analysing this pattern, we can get the values of the features and continue the search using Euclidean distance with the feature values we just got from analysing the pattern.

8.1.2 Using Microarchitecture

The features and their values that a reference program should have to be able to produce maximum power consumption can be calculated using the microarchitecture of the CPU that we are planning to run the search on. By looking on how many instructions can be fetched during a single cycle, how many functional units are there, and what those functional units support (load or store, Vector ALU, scalar ALU, etc.). Analysing the microarchitectural components we end up with the features and values of the reference program. This is a tactic that is not very conventional because the user must have an extensive understanding on microarchitecture and how to analyse it.

8.2 Features of reference virus

The features I used to compose the reference virus are not producing the best program, as we saw in chapter 6.2, we should analyse the reference program and find some features that mange to be left out, some ideas are the RAW dependencies and the position of the instructions. Analysing those futures would be a key factor to be able to produce easier the reference program we discussed on chapter 8.1. In the case of creating the reference virus with the help of Hybrid Quickgen method it would make the examination of the pattern clearer. Also, this makes the search of reference program via Microarchitecture easier since all components that makes a program powerful would be found and corelated in a single microarchitecture and then it would be simpler to achieve that with other microarchitectures.

Conclusion 8.3

Optimizing the generation time of stress-tests is key to speed up the reliability testing phase of a processor. To achieve this, we propose a method that eliminates the compilation and execution phases required when generating power viruses automatically. Specifically, our analysis showed that the Euclidean distance with the appropriate parameters, provides a good estimation on the performance difference of a power virus compared to a reference point (in our analysis we used as a reference point the power viruses generated by the default methods of GeST). Based on this observation we

incorporated the Euclidean distance of our analysis in GeST as an optimization metric. Our experimental evaluation showed that using the Euclidean distance with the right parameters as an optimization metric can significantly speed up the generation time of a power virus without affecting its performance.

As far as we know there is no other publicly available work that uses the Euclidean distance as an optimization metric in an automatic stress test generation tool. For future work, we want to examine ways that will allow us to extract the values of the reference point automatically.

Bibliography

1. “CPU frequency scaling - ArchWiki.” Arch Wiki, 21 April 2022, https://wiki.archlinux.org/title/CPU_frequency_scaling. Accessed 12 May 2022.
2. “Getting started with the perf command.” IBM, https://www.ibm.com/docs/en/linux-on-systems?topic=performance-getting-started-perf-command. Accessed 8 May 2022.
3. Hadjilambrou, Zacharias. “GeST: An Automatic Framework For Generating CPU Stress-Tests 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 max.” 2019, p. 20. IEEE.
4. “Power virus.” HandWiki, 28 September 2021, https://handwiki.org/wiki/Power_virus. Accessed 9 May 2022.
5. “P-states and C-States.” Microsoft Docs, 31 May 2018, https://docs.microsoft.com/en-us/previous-versions/windows/desktop/xperf/p-states-and-c-states. Accessed 12 May 2022.
6. Rexford, Jennifer. “Figure 4: Selected parts of a modern x86 architecture which are...” ResearchGate, https://www.researchgate.net/figure/Selected-parts-of-a-modern-x86-architecture-which-are-relevant-to-NoHype_fig3_278688751. Accessed 8 May 2022.
7. “Sandy Bridge (client) - Microarchitectures - Intel.” WikiChip, 29 December 2020, https://en.wikichip.org/wiki/intel/microarchitectures/sandy_bridge_(client). Accessed 8 May 2022.
8. “What are Computer Viruses? | Definition & Types of Computer Viruses.” Fortinet, https://www.fortinet.com/resources/cyberglossary/computer-virus. Accessed 9 May 2022.
9. “What Are The Different Types Of Computer Viruses? | Uniserve IT.” Uniserve IT Solutions, https://uniserveit.com/blog/what-are-the-different-types-of-computer-viruses. Accessed 9 May 2022.
10. “What is stress testing? - Definition from WhatIs.com.” TechTarget, https://www.techtarget.com/searchsoftwarequality/definition/stress-testing. Accessed 9 May 2022.

11. “What is x86 Architecture? - Definition from Techopedia.” Techopedia, 24 October 2012, https://www.techopedia.com/definition/5334/x86-architecture. Accessed 8 May 2022.

12. Karthik Ganesan. “System-level Max Power (SYMPO) - a systematic approach for escalating system-level power consumption using synthetic benchmarks” 2010, p 10, IEEE

13. M. Weaver, V., 2007. Can Hardware Performance Counters be Trusted?. https://ieeexplore.ieee.org/document/4636099