Performance Analysis of Embedded Control Algorithms used in UAVs

Iago de Oliveira Silvestre*, Leandro Buss Becker*
*Graduate Program on Automation Systems Engineering (PGEAS)
Federal University of Santa Catarina - Florianópolis, SC, Brazil
Email: iago.silvestre@posgrad.ufsc.br, leandro.becker@ufsc.br

***Abstract*—Performance analysis of embedded systems is critical when dealing with Cyber-Physical Systems that require stability guarantees. They typically operate having to respect deadlines imposed during the design of the related control system. In a recent past performance analysis was typically done only by executing the code, and making measures, on the target embedded platform. Nowadays, code execution/measuring can also be done on simulation software, which offers greater degree of liberty for designers to configure the system for the desired tests. This paper presents results obtained from analyzing the performance of two control algorithms developed for controlling an Unmanned Aerial Vehicle (UAV) running on simulated and real embedded platforms. Such analysis is important for twofold reasons: better understand the timing behavior or the algorithms and evaluate architectural issues related with the target embedded platform. Raspberry Pi 3 Model B+ (with Cortex-A53 processor) is used as reference platform and serves as basis for creating different simulated versions for the analysis. Initial results highlighted the important role played by cache memory in the performance of the control algorithms and were able to detect a major bottleneck in one of the control algorithms that could compromise system stability.**

***Index Terms*—full-system simulation, gem5, embedded computing systems, UAV**

I. Introduction

An embedded computing platform combines one or more processors, memories, and input/output devices. They are typically used in the context of Cyber-Physical Systems (CPS), that is, the software running on the embedded platform controls the movements of the mechanical system that they are coupled with. For this reason they are commonly classified as real-time applications, as they need to respect computation deadlines to ensure the system stability [1].

A typical application of embedded control systems in the context of CPS relates to Unmanned Aerial Vehicles (UAVs). Such aircrafts can be controlled remotely by a human operator or can fly autonomously through a control program [2]. In both cases the use of an embedded control algorithm is required to ensure the flight stability [3], this is known as the low-level control of the aircraft. The study of control techniques for UAVs is one of the research areas explored by the ProVANT project, a collaborative research effort between Federal Universities of Santa Catarina and Minas Gerais.

This paper presents the analysis of two embedded control algorithms developed to control the flight stability of UAVs. Such analysis is valuable for embedded systems designers for allowing to observe the temporal performance of the algorithms and the platform architectural aspects that influence such performance. In the first case, it allows to: (i) test if the control program can be executed within its designed time window (deadline): (ii) if it does not meet the deadlines, help to detect where are the bottlenecks in the control algorithm implementation. From the perspective of the embedded platform, it provide means for the embedded architect to better select the most appropriate target platform. These analysis should allow to iterate and explore modifications in software and hardware [4] that need be to made so that deadlines are fulfilled and the system stability is not compromised [5].

The remainder parts of the paper are organized as follows. Section II describes modern means/tools to simulate embedded systems, also detailing gem5, the simulation tool used in our study. Section III describes the control algorithms that are being analyzed in the paper. Section IV shows the results obtained in our simulation studies. Section V draws some conclusions and highlights future works directions.

II. Full-System Simulators

The simulation of an embedded system can be accomplished through various means, which differ from each other in various ways but mostly on the level of abstraction used on its models that represent the function of microprocessors and other parts of the computer system [6]. The lowest level of abstraction to simulate an embedded system can be reached using hardware description languages, such as VHDL, which when well utilized can reach the highest level of accuracy when comparing to the real system [7]. This approach however is extremely time consuming and normally higher level of abstractions are used to model the embedded system.

For our project we focused on *Full-System Simulators* (FSS), they are software programs that simulate a computer system hardware and the operating system being used, so that the user program of interest executes just like in the real physical hardware.

A FSS allows the user to simulate operating systems, devices drivers, kernels, and middleware. For this reason one of its common use is to detect system failures before the hardware design phase is completed. It can also can be used to test the performance on different configurations of hardware to guide later design phases.

A. gem5 Simulation Platform

The present work makes use of the gem5 simulation platform [8]. It is a modular discrete event-driven simulator that originated from the merge of two existing simulation tools: the M5 from the University of Michigan and GEMS from the University of Wisconsin-Madison. The gem5 simulator offers various CPU and memory system models that differ on the level of accuracy. For instance, the more complex CPU models include the simulation of the CPU branch predictor and instruction pipelines [9], both common features of a modern CPU architecture.

The gem5 tool most accurate simulation mode which was utilized in this paper is called *Full System Mode* which allows to simulate a complete system with its components and operating system (see Figure 1 for an overview of its data flow).

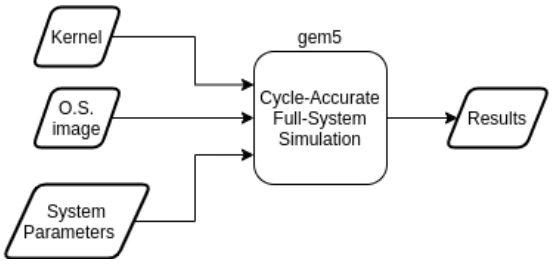

Fig. 1. Overview of gem5 Full System mode.

Gem5 also supports a high variety of ISA (Instruction Set Architecture) such as ARM, RISC-V, x86, Alpha, MIPS, Power, and SPARC, which allows among other things the study of the impact on performance by using either reduced or complex instructions set architectures [10].

Another important feature of gem5 simulation platform is M5ops, a set of opcodes that can be added to the program/code under test in order to provide very useful information. For instance, it allows the user to determine the part of the program code where the simulation tool should collect the performance data. In our case we wanted to analyze the performance of the control loops that are executed in the embedded system after the initialization, as shown in Figure 2.

```
int main()
    {
    ...
    hinfinity* control = new hinfinity();
    control->config();
    while(k<20){
    m5_reset_stats(0,0);
    out=control->execute();
    m5_dump_stats(0,0);
    k++;
    }
    return 0;
}
```

Fig. 2. M5ops being used to isolate the program portion under analysis.

The simulation analyses conducted on this paper were performed using the following gem5 models, binaries, and parameters:

- ISA: *ARM*
- CPU Model: *HPI* and *O3*
- Memory Model: *Classic*
- gem5 simulation binary: *gem5.fast*

Once a gem5 simulation is finished, it is possible to extract results by analyzing the generated stats files. These files keep a detailed log of every parameter of the hardware components used on the simulation and require some filtering by the user to extract a comprehensible overview of the test performance.

III. System Under Test

Fig. 3. VANT prototype used on this investigation.

To make the performance analysis envisioned in this investigation, we selected two C++ control programs developed within the ProVANT project. Both algorithms were designed to control the UAV prototype shown in Figure 3 and both were compiled with the floating point operations being solved by software. They differ from each other on the control technique, one uses a common LQR strategy while the other uses a mix of H_2 e $H\infty$ strategies with a feedforward control [11] to control a scenario where the UAV is carrying a load. An important characteristic is that both algorithms must be executed within a 12 millisenconds windows, otherwise the UAV may not stabilise. Table I compares source code information extracted from both control algorithms, available for download in our repository[1].

TABLE I
COMPARISON OF LQR AND H2/H∞ SOURCE CODES

	LQR	H2/H∞
Number of characters in source code	21 597	2 410 134
Compiled Binary Size	2.69 MB	19.9 MB

The difference between code characters and binary size is most likely due to the feedforward implementation on the H2/H∞ control program, with a header file of 2.3 MB and approximately 2.3 million characters.

[1] *https://github.com/iagosilvestre/ProVANT-Control-Test-gem5*

Regarding their source code structure, both share a lot in common, like the use of a C++ class structure. The Eigen library is used by both of them for solving matrix calculations and both use the Robot Operating System (ROS) libraries to package sensor data and communication info.

However, they differ in some important aspects. One difference relates with their matrix sizes, as in LQR the Expanded State Vector has 20 states and in H2/H∞ it has 24. They also distinguish on the Feedforward implementation. In LQR the disturbance is dealt as a constant value and the Feeforward can be performed as a simple addition before the output. In H2/H∞, however, since it is designed to control a scenario where the UAV has to carry an attached load, the Feedforward implementation has to calculate the disturbance with a set of complex matrix operations in every iteration of the control loop.

A. Hardware setup

For our initial test setup we chose to configure the simulation by cloning the specifications of the Raspberry Pi 3 Model B+ (see Table II), a very common ARM architecture development board. We chose this configuration in order to compare obtained results with the results of tests executed on the real hardware.

TABLE II
RASPBERRY PI 3 B+ SPECS - BASELINE SETUP

CPU	4 Core Cortex-A53 (ARMv8) 1.4Ghz
L1 I Cache	16 kB
L1 D Cache	16 kB
L2 Cache	512 kB
RAM	1GB LPDDR2 SDRAM

Regarding the Operating System image and Kernel, the tests executed in this paper used a compact aarch64 OS image file and a Linux kernel. These were chosen to be similar to the Raspberry Pi default ambient and can be found in the gem5 resources [2], however they can be built and modified freely to test different configurations, like expanding and removing the cache memory and replacing the CPU.

The simulations were conducted using a Dell G3 3590 host machine, which has a 9300h Intel quad-core CPU with a clock speed of up to 4.1 GHz, and 8 GB of 2666 MHz DDR4 RAM.

IV. OBTAINED RESULTS

A. LQR

The first control program tested was the one using the LQR strategy. Tests consisted of executing 50 LQR control loops and took around a minute to simulate on the host machine. The results from LQR execution is shown in Tables III and IV. The following observations can be highlighted from these executions:

- The first control loop took 83 μs to complete, after that the reminder 49 tested loops needed an average of 48 μs to be computed.
- After the first control loop the memory bus utilization dropped considerably, with several control loops being the best case scenario where 0% of the memory bus was used. The worst case scenario after the initialization loop incurred 0.31% of memory bus utilization. These results suggests that one of the reasons for needing more time to compute the first control loop is connected with the RAM usage, which is slower to access.
- During the first control loop 46.4% of the CPU cycles were idle, for the rest of the control loops that value dropped to an average of 8.5% with small variance. The higher amount of idle cycles during the first loop can be related with the RAM data readings mentioned before, since one of the most common events that lead to a CPU to become idle is when it is processing instructions that require data from variables that have to be fetched from RAM.

TABLE III
CPU DATA FROM LQR PROGRAM

LQR CPU Data	min	max	avg	avg*
Execution time	47 μs	82 μs	50 μs	48 μs
Idle Cycles	8.2%	45.6%	10.3%	8.4%
Cycles Per Instruction	1.598	2.762	1.677	1.621
Instructions Executed	40894	42313	41654	41650
Operations Executed (including micro ops)	49887	51378	50812	50806

*-Excluding data from the first control loop

TABLE IV
MEMORY DATA FROM LQR PROGRAM

LQR Memory Data	min	max	avg	avg*
L1 I Cache Occupancy	100%	100%	100%	100%
L1 D Cache Occupancy	100%	100%	100%	100%
L2 Cache Occupancy	99.8%	100%	100%	100%
Memory BUS Utilization	0%	10.98%	0.57%	0.02%
RAM READ	0 B	27 KB	562 B	8 B
RAM WRITE	0 B	11 KB	229 B	11 B

* Excluding data from the first control loop

The types of operations executed by the CPU are presented in Table V. More than half of the instructions are executed on the Arithmetic Logic Unit (ALU). These operations can basically consist of simple logical and arithmetic operations.

[2] *http://dist.gem5.org/dist/current/arm/aarch-system-20180409.tar.xz*

TABLE V
INSTRUCTION COUNT FROM LQR PROGRAM

Instruction Type	Total Count	Percentage
IntAlu	31744	62.46%
MemRead	9953	19.58%
MemWrite	8794	17.30%
IntMult	329	0.65%
IntDiv	0	0%

B. H2/H∞

While LQR executed very fast, the mix of H_2 and H∞ took much longer to execute. Therefore tests consisted of executing only 20 control loops and took approximately 21 minutes to execute on the host machine. An overview of the H2/H∞ execution results is shown in Tables III and VII. Follows discussions related with the obtained simulation results:

- The first control loop took 52.7 ms to compute, after that the other 19 executions had an average of 33.5 ms, well above the designed 12 ms control period.
- It showed higher RAM data transfers, with the worst case scenario being the first control loop where 25.2 MB were used. For the rest of the control loops this value dropped to an average of 12.3 MB with small variance.
- It showed a high percentage of idle CPU cycles during its entire execution, averaging at 45%. This highlights that during a big part of the program execution the CPU is waiting for data to complete its calculations.

TABLE VI
CPU DATA FROM H2/H∞ PROGRAM

H2/H∞ CPU Data	min	max	avg
Execution time	33.43 ms	52.77 ms	34.44 ms
Idle Cycles	38.53%	45.02%	44.94%
Cycles Per Instruction	3.02	3.63	3.58
Instructions Executed	12,932,425	24,512,688	13,522,197
Operations Executed (including micro ops)	16,501,948	30,322,905	17,199,833

Table VIII details the types of operations executed by the CPU. Like for LQR, these values did not change much during the 20 executed control loops.

We can see that similarly to the LQR program, the heavy majority of the CPU operations were executed on the ALU.

C. Bottleneck Analysis

With these results showing such poor performance, we went ahead and tested the same control programs on the Raspberry Pi development board to verify if the simulation data can be used to infer the performance of this control algorithm. The comparison of the execution time between the gem5 simulation and physical hardware can be seen in Figure 4 and Figure 5.

TABLE VII
MEMORY DATA FROM H2/H∞ PROGRAM

H2/H∞ Memory Data	min	max	avg
L1 I Cache Occupancy	100%	100%	100%
L1 D Cache Occupancy	100%	100%	100%
L2 Cache Occupancy	99.5%	100%	100%
Memory BUS Utilization	8.97%	11.73%	9.12%
RAM READ	12.1 MB	13.6 MB	12.2 MB
RAM WRITE	35 KB	12.0 MB	0.63 MB

TABLE VIII
INSTRUCTION COUNT FROM H2/H∞ PROGRAM

Instruction Type	Total Count	Percentage
IntAlu	13 089 591	79.20%
MemRead	1 553 578	9.39%
MemWrite	1 790 873	10.83%
IntMult	92 936	0.57%
IntDiv	109	∼0%

After the first control execution the average mismatch was of around 20%. Despite the mismatch one can note that for the H2/H∞ control simulation and physical hardware executed above the 12 ms deadline imposed by control designers, compromising the system stability. Besides that, it is also important to remind that the validation and improvement of the gem5 models is a constant work in development [12], that means that it is possible to achieve more realistic simulation results by finding and fixing the mismatch sources.

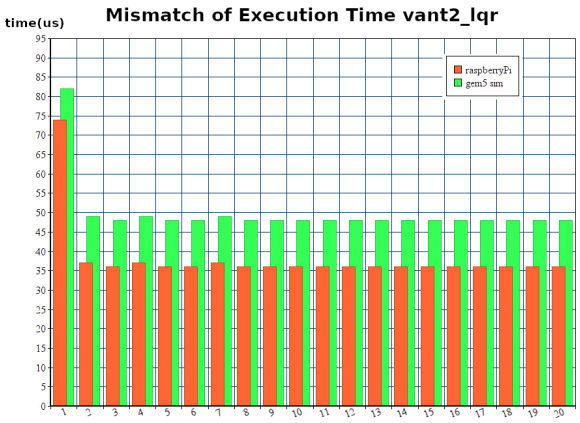

Fig. 4. Execution time of simulation and physical hardware - LQR control.

On the process of trying to find the bottleneck of this control program we noticed that one of the features that differentiates this control program from LQR is the implementation of a feedforward control with a huge 2 MB header file. To

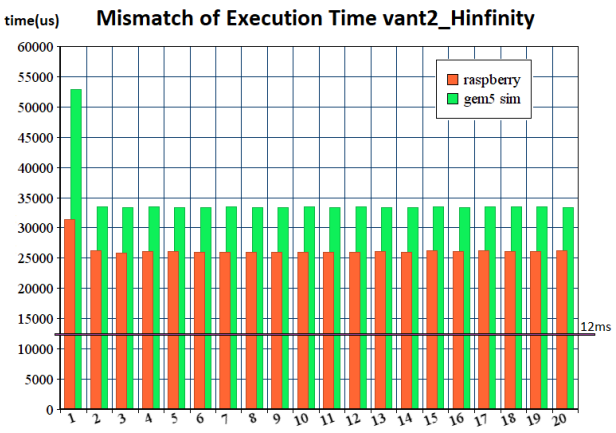

Fig. 5. Execution time of simulation and physical hardware - H2/H∞ control.

better understand the problem the feedforward function was analyzed with help from the *m5ops* library and the following observations could be measured:

- From the 33.5 ms average execution time, 33.3 ms was used on the feedforward section, evidencing that it is the bottleneck of this control strategy.
- The feedforward section was responsible for the entire RAM usage found during the control execution.
- The feedforward section is also responsible for approximately 99% of the CPU idle cycles.

D. Expanding the H2/H∞ Simulation Ambient Cache Size

An advantage of using a simulation tool like gem5 is that the simulated hardware can be easily modified. Therefore it was conducted a test where the L1 and L2 caches of the CPU were increased to 128 MB each. Although this is a very unrealistic, since most modern CPU have level 1 cache in the kB scale, the test intended to isolate the CPU operation, avoiding idle cycles due RAM interactions. Like if the data to be computed is always "ready" (near) when needed. So we executed 10 control loops with these changes and observed drastic changes in the obtained results.

- The average execution time dropped from 33.5 ms to 17.7 ms, a 52% drop in the execution time.
- As expected, the number of CPU idle cycles dropped from an average of 45% idle cycles to around 2%. This shows that the program has no longer to wait for data to be transferred from the slow RAM, as it is available in the faster data cache.

However, even with this huge cache increase, it is still evident that the feedforward control presents a bottleneck. It requires 17.6 ms to be computed, out of the 17.7 ms of the entire program. This means that either the control has to be redesigned for a larger control window if it wants to be able to run on a similar hardware to a Raspberry Pi 3 Model B+, or that the feedforward implementation should be simplified.

E. Enhancing the CPU to Cortex-A72

We also performed experiments using a faster CPU, more specifically the Cortex-A72, which is used in the recently launched Raspberry Pi 4. Its main differences in comparison with Cortex-A53 are presented in Table IX.

TABLE IX
CPU ARCHITECTURES COMPARISON

Architecture Comparison	Cortex-A53	Cortex-A72
Decode	2-wide	3-wide
Pipeline depth	8	15
Type of Execution	In-Order	Out-of-Order
L1 Cache (Instr + Data)	8–64 KB + 8–64 KB	48 KB+ 32 KB
L2 Cache	128 KB - 2 MB	0.5 - 4 MB

Since the type of execution changes, the tests on this section were conducted with the O3 cpu model. Two sets of tests were then conducted, with the first one using the Rasp. 3 B+ cache values (16kB+16kB for L1 cache and 512kB for L2) and the second set intending to use the maximum amount of cache memory available in Cortex-A72 architecture. However, due to a gem5 limitation, the cache cannot assume values that are not power of two (2^n). Therefore this second test was executed with 32kB+32kB for L1 cache and 4MB for L2. It is important to mention that CPU cores and frequency were not changed during these tests and it remained as a 1.4 GHz quad-core.

For LQR, again the tests consisted of 50 executions of the control loop. An overview of the obtained results are presented in Table X. It is possible to observe a significant improvement in the execution time in comparison with the Cortex-A53 (32% faster on average in test 1).

TABLE X
OVERVIEW OF THE LQR ON CORTEX A72

Test 1: 16 kB + 16 kB L1 & 512 kB L2			
	min	max	avg
Exec Time	32 μs (-32%)	60 μs (-27%)	34 μs (-32%)
Idle Cycles	10.0% (+22%)	38.8% (-15%)	11.3% (+10%)
Mem Bus	2.7% (++2.7%)	15.6% (+42%)	3.0% (+426%)
RAM READ	4 KB (++4KB)	32 KB (+19%)	4.4 KB (+700%)
RAM WRITE	0 B (+0%)	7 KB (-36%)	0.18 KB (-20%)
Test 2: 32 kB + 32 kB L1 & 4 MB L2			
	min	max	avg
Exec Time	27 μs (-43%)	56 μs (-32%)	29 μs (-42%)
Idle Cycles	4.0% (-51%)	36.7% (-20%)	5.3% (-48%)
Mem Bus	1.2% (++1.2%)	13.5% (+23%)	1.5% (+163%)
RAM READ	1 KB (++1KB)	31 KB (+15%)	2 KB (+264%)
RAM WRITE	0 B (+0%)	1 KB (-91%)	34 B (-85%)

For H2/H∞, again the tests consisted of 20 executions of the control loop. An overview of the obtained results is presented

in Table XI. Similarly to the LQR execution in the A72, it is possible to notice a significant improvement in the execution time in both tests (33% and 35% for tests 1 and 2 respectively).

TABLE XI
OVERVIEW OF THE H2/H∞ ON CORTEX A72

Test 1: 16 kB + 16 kB L1 & 512 kB L2			
	min	max	avg
Exec Time	22.1 ms (-34%)	44.5 ms (-16%)	23.2 ms (-33%)
Idle Cycles	34.7% (-10%)	35.2% (-22%)	35.2% (-21%)
Mem Bus	13.5% (+50%)	17.7% (+51%)	13.7% (+50%)
RAM READ	12.1 MB (+0%)	19.6 MB (+44%)	12.5 MB (+2%)
RAM WRITE	28 KB (-20%)	12.5 MB (+5%)	0.65 MB (+3%)
Test 2: 32 kB + 32 kB L1 & 4 MB L2			
	min	max	avg
Exec Time	21.3 ms (-36%)	41.2 ms (-22%)	22.4 ms (-35%)
Idle Cycles	32.7% (-15%)	34.1% (-24%)	34.0% (-24%)
Mem Bus	13.1% (+46%)	17.3% (+47%)	13.3% (+46%)
RAM READ	11.4 MB (-6%)	15.9 MB (+17%)	11.6 MB (-5%)
RAM WRITE	12.6 KB (-64%)	13.1 MB (+9%)	0.66 MB (+5%)

It is important to highlight that these set of tests should be interpreted as an indicator of the performance tendency. They should not be seen as absolute or definitive performance statement of the analyzed control algorithms. This comes from the fact that there are several parameters in the simulation tool configuration that might change to more or less some of these numbers.

V. CONCLUSIONS AND FUTURE WORKS

The gem5 simulator showed to be as a very valuable asset to test and determine programs characteristics, especially in regarding analyzing their time behavior on specific hardware configurations. This tool can be used for multiple ends but one that is very noticeable is the ability to test how the hardware configuration can impact a given program, so that designers can make the proper tuning to achieve the envisioned performance levels. It also helps designers to find bottlenecks in their programs.

A. LQR

The LQR control program showed great performance in Cortex-A53 with the Rasp. 3 B+ configuration, executing the control loop with ease inside the 12 ms control window. It also showed a relative low percentage of idle CPU cycles, which shows that the CPU is not waiting too much for data to complete its operations. This was most likely due to being a control program that had almost insignificant usage of the RAM after its first control loop. This allows to conclude that this LQR implementation was not memory-intensive for the utilized architecture and its performance was mostly led by the CPU capabilities.

B. H2/H∞

The H2/H∞ control program showed poor performance for the envisioned application needs using Cortex-A53 standard configuration (well above the 12 ms time window). With the help of gem5 we were able to easily detect that the bottleneck responsible for the poor performance was the feedforward implementation. The tests concerning the system upgrade to a Cortex-A72 CPU presented considerable execution time improvement (circa 33%), however the execution time would still be above the envisioned 12 ms time window.

We have discussed these results with the control development team and some solutions are being studied such as some matrix calculations that could have been calculated outside of the feedforward function and modifying the matrix declaration.

C. Future Works Directions

- Evaluate the impact of changing the ISA [10].
- Explore the gem5-gpu tool [13] since nowadays it is not too expensive to include GPUs on embedded platforms.
- Analyze additional other control methods used in Provant, such as MPC (Model Predictive Control) [14].
- Evaluate the results of binaries compiled with floating point hardware enabled.

REFERENCES

[1] A. Aminifar, *Analysis, Design, and Optimization of Embedded Control Systems*. Linköping University, 2016.

[2] G. V. Raffo, M. G. Ortega, and F. R. Rubio, "An integral predictive/nonlinear h∞ control structure for a quadrotor helicopter," *Automatica*, vol. 46, no. 1, pp. 29–39, jan 2010.

[3] F. Silvano, "Projeto da arquitetura de software embarcado de um veículo aereo não tripulado," Master's thesis, Federal University of Santa Catarina, 2014.

[4] R. Tashiro and M. S. Oyamada, "An environment for design space exploration using gem5-McPAT," in *2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC)*. IEEE, nov 2016.

[5] B. P. Lathi, *Linear systems and signals*. Oxford University Press, 2005, ch. 9 - Time-Domain Analysis of Discrete-Time Systems.

[6] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of GEM5 simulator system," in *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, jul 2012.

[7] V. Pedroni, *Circuit Design and Simulation with VHDL*. MIT Press Ltd, 2010.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, may 2011.

[9] M. R. Zargham, *Computer Architecture*. Prentice Hall, 1996.

[10] A. Akram, "A study on the impact of instruction set architectures on processor's performance," Master's thesis, Western Michigan University, 2017.

[11] R. Donadel, "Modeling and control of a tiltrotor unmanned aerial vehicle for path tracking," Master's thesis, Federal University of Santa Catarina, 2015.

[12] A. Akram and L. Sawalha, "Validation of the gem5 simulator for x86 architectures," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 53–58.

[13] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2015.

[14] G. M. T. Miranda, "Multi-core model predictive control strategy for a tilt-rotor uav in system-in-the-loop simulation," Master's thesis, Federal University of Minas Gerais, 2018.