Using *gem5* Simulator to Support Design Space Exploration Targeting ARM Architecture

Iago de Oliveira Silvestre*, Antonio Carlos Beck Filho†, 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
†Informatics Institute, Federal University of Rio Grande do Sul - Porto Alegre, RS, Brazil
Email: caco@inf.ufrgs.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 temporal constraints imposed during the design of the related control system. Until recently, performance analysis was done exclusively by executing the code on the target platform and making measures. Usually code execution/measuring can also be done on simulation software, which offers greater degree of flexibility for designers to configure the platform accordingly for the desired tests. However there usually is a common concern whether the results from such simulation software are reliable enough to be used to guide Design Space Exploration during a project development. This paper presents results obtained from analyzing the performance of control algorithms developed for controlling an Unmanned Aerial Vehicle (UAV) running on simulated and real ARM embedded platforms. Such analysis is important for mainly three reasons: better understand the timing behavior of the algorithms, evaluate architectural issues related with the embedded platform, and also determine how accurate is the simulation software. 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. Results showed that gem5 has enough timing precision when compared with results from the real hardware, so it proved to be a useful tool to support design space exploration on ARM architecture projects. An additional contribution of this work is the creation of a cloud computing environment at Google Colab, making significantly easier the use of gem5 simulator by non-experts.**

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

I. Introduction

Typical embedded computing platforms combine 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 many of them are real-time applications, as they need to respect timing constraints (e.g. deadlines) to ensure the system stability [1].

Unmanned Aerial Vehicle (UAV) control is a typical application of embedded system in the context of CPS. UAVs 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], which 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 UFSC and UFMG.

This paper presents the analyses of 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. This allows us 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 the bottlenecks are 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 analyses 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].

Simulation tools are often used during this analysis to speed up the process. However, these tools are limited in regarding the CPU architectures that they support, with some architectures, such as Intel x86, having significantly more simulation tools available than other architectures, such as ARM. Depending on the level of details in the simulation process, the simulation tool can be used to support Design Space Exploration [6]. Despite the benefits of the simulation tools, a common problem is the initial barrier to learn the specifics of the chosen simulation software and time to set up the simulation process in a reproducible and agile manner. As an attempt to make the gem5 simulation more streamlined, a cloud computing environment was developed through the use of Google Colab. Such environment is let available for the scientific community, as further detailed.

We also did some tests, both in hardware and software, regarding the usage of cache memory for the control programs. In the literature it is normally discussed the problems of using cache when trying to achieve determinism in a computer program, a couple of different approaches can be used then to try to improve program determinism and a more detailed discussion of this topic is presented in subsection V-D.

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 details the developed Google Colab implementation of the gem5 simulation process. Section V shows the results obtained in our simulation studies. Section VI 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 [7]. One of the lowest levels 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 [8]. This approach however is extremely time consuming and normally higher level of abstractions are used to model the embedded system.

The balance of enough abstraction to keep the simulation process agile with sufficient accuracy so that the results can be comparable to actual performance is going to depend on the project requirements. For our project, we focused on *Full-System Simulators* (FSS), which 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. FSS can provide up to three levels of precision: functional level, instruction level, and CPU cycles level, with the latter being the most accurate when compared to real performance and was the one chosen for the evaluation presented in this paper.

A. gem5 Simulation Platform

The present work makes use of the gem5 simulation platform [9]. 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 [10], both common features of a modern CPU architecture.

The gem5 tool provides two simulation modes. The *Full System Mode* allows to simulate a complete system with its components and operating system (see Figure 1 for an overview of its data flow). In the *Syscall Emulation Mode* the simulation is limited to the execution of user space programs and to use system services developed by the tool, trading lower accuracy for higher simulation speed. Since the Full System Mode offers higher accuracy, it was the chosen mode for the analyses conducted in this paper.

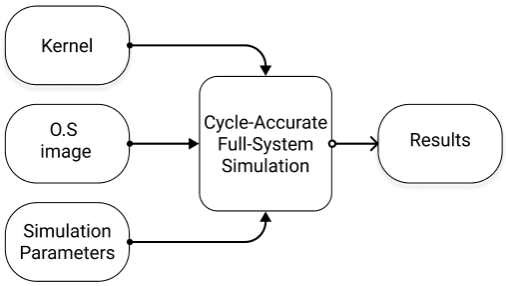

Fig. 1. Overview of gem5 Full System mode.

Gem5 supports a high variety of ISA (Instruction Set Architecture) such as ARM, RISC-V, x86, Alpha, MIPS, Power, and SPARC. This allows the study of the impact on performance by using either reduced or complex instructions set architectures [11]. Another important feature of gem5 is the facility to create and restore checkpoints along the simulation. This is important because it allows creating a checkpoint after the kernel and operating system initialization that, depending on the complexity of the adopted hardware models, can take a decent amount of time - and this checkpoint can be used to skip such initialization on further simulations. Another possibility is to use checkpoints to swap CPU models between parts of the program being tested, since this is not possible to be done during simulation runtime. Thereby one can start with a simplified CPU model and further change to a more complex one during a specific segment of the code.

Listing 1. M5ops being used to isolate the program portion under analysis

```
1  int main(){
2  ...
3  hinfinity *control = new hinfinity();
4  control->config();
5  while(k<100){
6     m5_reset_stats(0,0);
7     out=control->execute();
8     m5_dump_stats(0,0);
9     k++;
10 }
11 return 0;
12 ...
```

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 Listing 1.

Lastly, gem5 also offers a couple different simulation binaries for the users to choose based on what they need for their research purpose, as presented on Table I.

TABLE I
AVAILABLE BUILD TARGETS OF GEM5

Binary name	Optimizations	Run time debugging support	Profiling support
gem5.debug		X	
gem5.opt	X	X	
gem5.fast	X		
gem5.prof	X		X
gem5.perf	X		X

The simulation analyses conducted on this paper was 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.

Fig. 2. VANT prototype used on this investigation.

III. SYSTEM UNDER TEST

To make the performance analyses envisioned in this investigation, three C++ control programs developed within the ProVANT project were selected. They were designed to control the UAV prototype shown in Figure 2, and cover a different spectrum of control techniques with varying complexity. The first algorithm uses a common Linear–quadratic regulator(LQR) strategy, the second a mix of H_2 and H∞ strategies with a feedforward control [12] to control a scenario where the UAV is carrying a load, and finally the last one uses uses an approach called adaptive mixing control [13]. An important characteristic is that these algorithms we're designed be executed within a 12 millisenconds windows, otherwise the UAV may not stabilise. Table II compares source code information extracted from the control algorithms, which are available for download in our repository [14].

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

	LQR	H2/H∞	AdapMix
Characters in source code	21 597	2 410 134	84 530
Compiled Binary Size	2.56 MB	9.27 MB	10.9 MB

Regarding their source code structure, they share a lot in common, like the use of a C++ class structure. The Eigen library is used by all of them for solving matrix calculations and both use the Robot Operating System (ROS) libraries to package sensor data and communication info. One should notice the huge header file of 2.3 MB only present in the H2/H∞ program, needed by its Feedforward implementation.

They differ, however, in some important aspects. The first difference relates with their matrix sizes, as in LQR the Expanded State Vector has 20 states, in the adaptive mixing 18 and in H2/H∞ it has 24. This suggests more complex matrix calculations for the H2/H∞. 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. There is no feedforward action present in the adaptive mixing control algorithm.

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 III), 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. Besides, tests with modifications in the specifications presented in Table III were also conducted.

TABLE III
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 [15] , 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.

Regarding the compiling process for these algorithms, we should mention that:

- arm-linux-gnueabi-g++ was used to cross-compile the programs to the ARM architecture.
- To allow the execution of hardware floating-point instructions the compiler flag -mfloat-abi=softfp was used
- To optimize the code, the -O2 compiler flag was used. -O2 was chosen instead of -O3 since we wanted to optimize the code but maintain it small as possible and -O3 can sometimes generate more machine code.

IV. Google Colab Environment

As an attempt to make the gem5 simulation process more streamlined, a Google Colab environment was developed where you can use cloud computing to experiment with gem5. The environment can be accessed in [16]. So far it allows the user to simulate ARM architecture compiled binaries on a Full System simulation mode on a Linux based machine.

The process to use gem5 through Colab is further detailed in the environment link in the form of text, however it basically consists of:

- Installing the gem5 dependencies and the simulation tool, in this step our precompiled files are also downloaded to cut the initial setup required to run gem5
- Import your prebuilt ARM compiled binary from your Google Drive
- Choose simulation Parameters, such as CPU clock speed, number of cores and cache sizes.
- Run the simulation block
- Download the stats.txt file from the /gem5/m5out path to analyze your program simulation performance

For more familiarized gem5 users, there are options to create your own checkpoints and run the control program under test directly from the simulation terminal, these options however require the user to access the Google Colab host machine through Secure Shell protocol (SSH), besides a basic understanding of the gem5 simulation process.

We have also prepared a basic guide to use the Google Colab gem5 implementation to simulate ARM binaries in the form of video, which can be accessed in [17]

V. Results

The simulation process for each algorithm consisted of the execution of 100 control loops, then we analyzed the data from each control loop as well as the overall program performance. For the H2/H∞ and AdapMix algorithms some additional tests on the Raspberry Pi board we're made to measure gem5 precision on some CPU data such as total instructions count.

A. LQR

An important measurement for control programs is the execution time of each control loop. This is used to determine whether the control program is able to execute its algorithm inside the designed control window of time. The simulation results for the control loops executed for the LQR algorithm is presented in Table IV.

TABLE IV
CPU Data from LQR program

LQR CPU Data	min	max	avg
Execution time	1 us	29 us	1.5 us
Idle Cycles	2.04%	22.55%	2.45%
Cycles Per Instruction	1.53	2.35	1.58
Instructions Executed	1 352	17 000	1 501
Operations Executed (including micro ops)	1 829	19 060	2 003

From these results we can see that the LQR algorithm was able to be executed in quick manner with a worst case execution time of 29 microseconds on the first control loop, still well inside the designed control window of 12 milliseconds. It is also interesting to analyze Memory utilization during the control program execution, this information is present in gem5 results and is presented in table V.

TABLE V
Memory Data from LQR program

LQR Memory Data	min	max	avg
Memory BUS Utilization	0%	3.93%	0.09%
RAM READ	0 KB	3.6 KB	0.04 KB
RAM WRITE	0 KB	1 KB	0.01 KB

As we suspected we can see that LQR was a program that barely used the Memory BUS of the Raspberry Pi, with the worst case being the first control loop with reads and writes of a few kilobytes. During the rest of the control loops it was common to have zero DRAM BUS utilization, suggesting that Raspberry cache was sufficient for this control program data profile. One other thing available in gem5 results that we can analyze is the types of CPU instructions executed during the control program, this data is available in Table VI.

We can see that for the LQR algorithm there were very few Floating Point instructions executed, with the big majority being executed in the ALU (Arithmetic Logic Unit) and memory instructions. This information combined with the temporal performance can be useful for the process of choosing the CPU for a project. In the case of the Linear-Quadratic Regulator we see that floating point instructions were a small percentage of the overall instructions, this coupled with the fast performance showed in Table IV suggests that a CPU with hardware floating point is not necessary for this control implementation.

TABLE VI
INSTRUCTION COUNT FROM LQR PROGRAM

Instruction Type	Total Count	Percentage
IntAlu	107 770	50.13%
IntMult	281	0.13%
FloatAdd	3000	1.40%
FloatMult	2100	0.98%
FloatMultAcc	8400	3.91%
MemRead	55024	25.59%
MemWrite	38111	17.73%

1) Comparison to Raspberry Pi results: We also ran some tests on the Raspberry Pi board, which confirmed to us the longer execution time for the first control loop, for the first 10 control loops the execution times are shown in Figure 3.

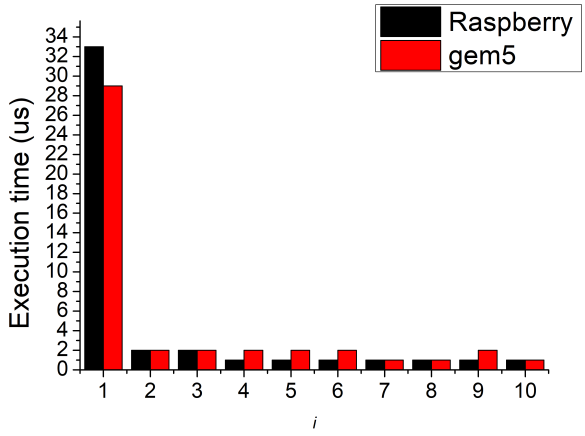

Fig. 3. Execution time of simulation and physical hardware for the LQR.

We can see that for the LQR control program, the executions times between gem5 simulation and real hardware were fairly similar. Overall, gem5 was fairly accurate for this LQR implementation we tested and provided data with sufficient precision to help us understand the control program performance.

B. H2/H∞

Similarly to the LQR results, we start with an analysis of each control loop and later an overview of the control program execution as a whole. The control loops execution times are presented in Table VII along with some other CPU data.

For this control algorithm we saw very different results, with an average execution time of around 2 milliseconds and a worst case of 6.57milliseconds for the first control loop execution, dangerously close to the 12 milliseconds designed control window when we consider that these simulation only tested the control thread performance with no other processes being performed, which will surely not be the case in an UAV

TABLE VII
CPU DATA FROM H2/H∞ PROGRAM

H2/H∞ CPU Data	min	max	avg
Execution time	2.15 ms	6.57 ms	2.25 ms
Idle Cycles	49.30%	76.6%	75.73%
Cycles Per Instruction	3.44	7.06	6.95
Instructions Executed	442 908	2 669 085	467 169
Operations Executed (including micro ops)	451 162	3 097 272	484 690

embedded system. Another worrying result is the high percentage of CPU idle cycles with an average of approximately 75%, this indicates that the CPU is most likely waiting for data to conclude its instructions for a good part of the control program execution.

We can check the simulation results for memory utilization in table VIII.

TABLE VIII
MEMORY DATA FROM H2/H∞ PROGRAM

H2/H∞ Memory Data	min	max	avg
Memory BUS Utilization	16.69%	23.10%	22.83%
RAM READ	1.84 MB	2.38 MB	1.87 MB
RAM WRITE	0.20 MB	2.08 MB	0.22 MB

Here we can see that the memory BUS utilization was high during every control loop executed, with an average of approximately 23% and reads/writes averages reaching the megabyte scale, this is worrying since RAM is significantly slower and this high usage could be causing the high amount of CPU idle cycles mentioned before.

Going to the control program overview part of the results, we can check what types of instructions were executed during it, this data is presented in Table IX.

TABLE IX
INSTRUCTION COUNT FROM H2/H∞ PROGRAM

Instruction Type	Total Count	Percentage
IntAlu	24 630 915	50.92%
IntMult	105 433	0.22%
FloatAdd	4 052 400	8.38%
FloatMult	5 532 600	11.44%
FloatMultAcc	916 400	1.89%
MemRead	9 229 763	19.08%
MemWrite	3 681 940	7.61%

Here we can see a significantly higher amount of floating point operations when compared to LQR results, most likely due to the feedforward implementation of this program, where multiple matrices are calculated every control loop. This

coupled with the relatively poor temporal results showed in Table VII suggests that hardware floating point calculations are probably necessary for this control program to perform adequately on the Raspberry Pi 3 B+ CPU.

1) Comparison to Raspberry Pi results: Similarly to the procedures for the LQR analysis, we ran some tests on the Raspberry Pi board to measure the control loops execution time mismatch between simulation and real hardware, shown in figure 4.

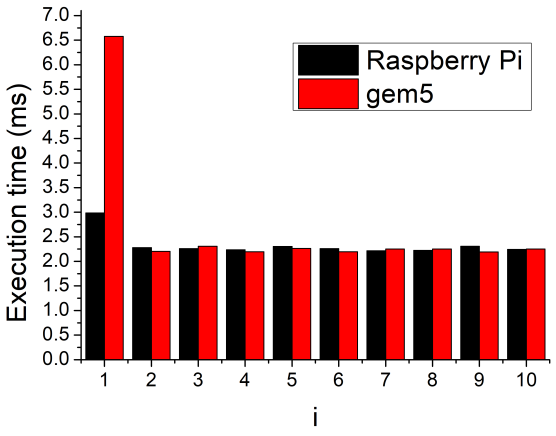

Fig. 4. Execution time of simulation and physical hardware for the H2/H∞ .

Here we can better see whether gem5 was accurate on execution time since the control program is significantly more complex than the LQR, we see that other than a big mismatch on the first execution time, gem5 was fairly accurate during the rest of the control loops for the H2/H∞ control program .

Additionally we also ran some tests on the Raspberry Pi using the perf tool available for Linux distributions and compared the results to the ones obtained in gem5 simulation, shown in Table X.

TABLE X
PERF AND GEM5 COMPARISON FOR THE H2/H∞ PROGRAM

gem5 and perf Comparison	Perf	gem5
Execution time	226.8 ms	227.4 ms (+0.2%)
CPU Cycles	316 552 247	318 526 537(+0,6%)
CPU Instructions	45 863 580	47 006 248 (+2.5%)
L1 dcache Loads	12 406 046	11 412 729 (-8%)
L1 dcache Misses	807 055	1 491 960 (+85%)
L2 cache loads	5 078 387	4 083 738 (-20%)
L2 cache Misses	3 039 040	3 084 321 (+1.5%)

We can see that perf, running on the Raspberry Pi, and gem5, running on our host machine, offered fairly similar results for program performance, with an exception of the level 1 data cache misses, which had a big mismatch of around 85%.

C. AdapMix

The last control program tested was an Adaptive Mixing algorithm, the results obtained for control loops execution is available in Table XI among some other CPU data.

TABLE XI
CPU DATA FROM ADAPTIVE MIXING PROGRAM

AdapMix CPU Data	min	max	avg
Execution time	159 us	463 us	172 us
Idle Cycles	9.42%	31.99%	9.70%
Cycles Per Instruction	1.66	2.42	1.68
Instructions Executed	134 120	266 482	135 898
Operations Executed (including micro ops)	167 519	324 880	169 654

We can see that in regards to execution time, this control algorithm was kind of a middle term between the LQR and H2/H∞. The average execution time was of 172 microseconds, well withing the 12 milliseconds designed control window and it also had a relatively small percentage of CPU idle cycles, with an average of around 10%. We can also check memory utilization in Table XII.

TABLE XII
MEMORY DATA FROM ADAPTIVE MIXING PROGRAM

AdapMix Memory Data	min	max	avg
Memory BUS Utilization	0%	10.24%	0.15%
RAM READ	0 B	84.3 KB	1.43 KB
RAM WRITE	0 B	113.2 KB	1.21 KB

As suspected, the memory BUS utilization was fairly low for the adaptive mixing algorithm, with reads and writes averages on the kilobytes scale. Similarly to the LQR control program the worst case was during the first control loop and during the rest of the control loops it was common to have zero DRAM BUS utilization, suggesting that Raspberry cache was sufficient for this control program data profile inside the basic control loop.

On a program overview analysis, we can see what types of CPU instructions were executed with the simulation results, this data is presented in Table XIII.

Here we can see that there were a few floating point instructions, but mostly it consisted of ALU and memory instructions. This data coupled with the relatively good temporal performance showed in Table XI suggests that perhaps hardware floating point calculations are not a necessity for this control program to execute well inside it's 12 milliseconds designed control window.

1) Comparison to Raspberry Pi results: Similarly to the procedures for the LQR and H2/H∞ analysis, we ran some tests on the Raspberry Pi board to measure the control loops execution time mismatch between simulation and real hardware, shown in figure 5.

TABLE XIII
INSTRUCTION COUNT FROM ADAPMIX PROGRAM

Instruction Type	Total Count	Percentage
IntAlu	8 501 492	48.58%
IntMult	146 011	0.83%
FloatAdd	39500	0.23%
FloatMult	83 800	0.48%
FloatMultAcc	1 560 600	8.92%
MemRead	4 726 114	27.01%
MemWrite	2 092 529	11.96%

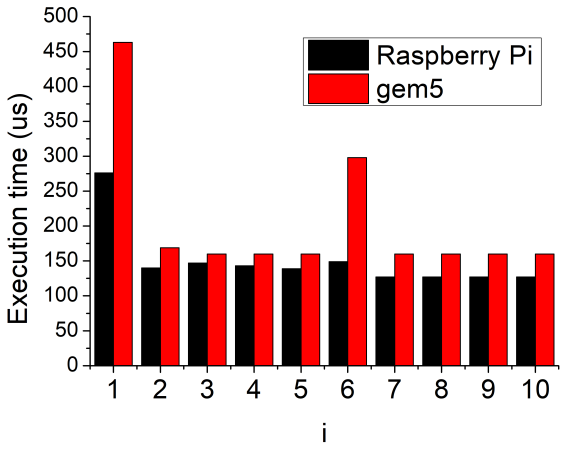

Fig. 5. Execution time of simulation and physical hardware for the AdapMix .

For this adaptive mixing algorithm we saw a higher mismatch between execution time of gem5 simulation and of the Raspberry Pi board. For the two control programs tested before only the first control loop had a big mismatch, but for this control program the first and sixth control loops had significantly high mismatches.

Additionally we also ran some tests on the Raspberry Pi using the perf tool available for Linux distributions and compared the results to the ones obtained in gem5 simulation, shown in Table XIV.

TABLE XIV
PERF AND GEM5 COMPARISON FOR THE ADAPMIX PROGRAM

gem5 and perf Comparison	Perf	gem5
Execution time	15.33 ms	17.25 ms (+12%)
CPU Cycles	21 328 246	24 158 670 (+13%)
CPU Instructions	12 152 587	14 027 697 (+15%)
L1 dcache Loads	5 785 617	6 532 073 (+13%)
L1 dcache Misses	122 528	243 220 (+98%)
L2 cache loads	464 477	284 646 (-40%)
L2 cache Misses	5 352	5 809 (+10%)

Most of the data extracted from perf running on the Raspberry Pi was similar to the ones we found in gem5 simulation, with the exclusion of level 1 data cache misses and level 2 cache loads.

D. Software Prefetching

One of the problems from the use of cache memory that is commonly discussed in literature is the fact that you can not achieve determinism in the execution of your program since there is no control in which data is stored or released from the cache during execution. However simply not using the cache memory most of the times is not an option, once you increase the complexity of the algorithm, the performance impact of the cache memory on execution time can be very easily verified.

To analyze this some tests on the gem5 simulator were made where we reduced the system cache to the minimum amount possible in gem5, which is 64 Bytes total for level 1 cache (32 Bytes for Instruction Cache and 32 Bytes for Data Cache) with no level 2 cache. We then tested the algorithms that performed most poorly (H2/H∞ and AdaptiveMixing) executing 100 control loops to verify the impact of virtually removing the cache memory, the results are shown in Table XV.

TABLE XV
GEM5 PERFORMANCE COMPARISON WITH NO CACHE*

H2/H∞		
	Default Raspberry 3 Cache	32B+32B L1
Exec Time	227.4 ms	842.13 ms
Idle Cycles	75.73%	86.41%
RAM READ	187.55 MB	666.60 MB
RAM WRITE	22.18 MB	39.54 MB
AdapMix		
Exec Time	17.25 ms	360.9 ms
Idle Cycles	9.70%	88.93%
RAM READ	0.3091 MB	258.99 MB
RAM WRITE	0.3349 MB	11.94 MB

* - 64 Bytes level 1 cache due to gem5 limitations

The results obtained from gem5 simulation showed a major impact on performance when we virtually remove the cache memory. Besides the very noticeable impact on execution time which increased 371% for the H2/H∞ and approximately 2100% for the AdaptiveMixing, we can also notice an increase to CPU idle cycles and major impact on the DRAM memory usage, particularly for the AdaptiveMixing control program.

To increase your program determinism while still using a cache memory, multiple solutions can be pursued, some of them are listed below:

- ScratchPad Memory
- Processing In Memory
- Software Prefetching

We implemented Software Prefetching in our control algorithms through a builtin prefetch function in the C++

compiler. For these tests we tried prefetching the matrices and vectors slightly before they we're going to be needed for calculations for the control algorithm. These tests showed us that the use of Software Prefetching as we implemented it did not offer impactful improvements in performance or memory usage for the control programs tested. The control programs with software prefetching alternatives are available in our repository.

VI. 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.

Regarding the Google Colab implementation, it can provide new users a fairly simpler way to execute simulations, so that they can perform these simulations in any machine, regardless the Operating System, simply using a Web-browser. It also speeds up the simulation process significantly through the use of checkpoints, although this method could also be used in a host machine in case the user knows how to create and restore it appropriately.

Given the obtained results, we can also conclude that gem5 is suitable to be used for ARM Design Space Exploration. Most of the simulation results presented good indicatives of the programs' performance and are certainly of interest during many phases of a project life, mostly during earlier stages, when it can help to adjust the hardware configuration needed to execute the embedded system code in a timely manner.

A. Future Works Directions

- Evaluate the impact of changing the ISA [11] on the performance results of the developed control programs.
- Explore the use of the gem5-gpu [18], a tool that originated from the merge of the gem5 and GPGPU-sim simulation tools. Nowadays it is not too expensive to include GPUs on embedded platforms and there are some studies on how to better use the combination of CPU and GPU to improve performance.
- Improve the study of how to find and solve program bottlenecks through the gem5 simulation tool.
- Explore other results that can be generated through the gem5 simulation, such as the CPU pipeline visualization through the use of the Konata tool.
- Explore the use of customized OS to reduce the amount of background processes and to better isolate the performance of the program-under-interest. One possibility is to remove the user space programs from the OS and create a custom disk image.
- Explore the use of gem5 to determine system scalability through simulation of other processes competing for CPU against the control programs.
- Improve the Google Colab implementation, through additional ISA precompiled gem5 binaries and different CPU models checkpoints for example.

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] 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)*, 2016, pp. 220–225.

[7] 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.

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

[9] 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.

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

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

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

[13] M. Kuipers and P. Ioannou, "Multiple model adaptive control with mixing," *IEEE TRANSACTIONS ON AUTOMATIC CONTROL*, vol. 55, no. 8, Aug 2010.

[14] I. O. Silvestre. (2021, Aug.) Provant control programs. [Online]. Available: https://github.com/iagosilvestre/ProVANT-Control-Test-gem5

[15] gem5. (2021, Aug.) gem5 linux image. [Online]. Available: http://dist.gem5.org/dist/current/arm/aarch-system-20180409.tar.xz

[16] I. O. Silvestre. (2021, Aug.) gem5 prototype in colab. [Online]. Available: https://colab.research.google.com/github/iagosilvestre/gem5Colab/blob/master/SBESC_gem5_Prototype.ipynb

[17] ——. (2021, Aug.) gem5 colab video guide. [Online]. Available: https://www.youtube.com/channel/UCuMtLsiR-Amuw9AizYQLCnw

[18] 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.