Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Datateknik
2024 | LIU-IDA/LITH-EX-A--24/001--SE

Real-Time System Benchmarking with Embedded Linux and RT Linux on a Multi-Core Hardware Platform

–

Kian Hosseini

Supervisor : Kamran Hosseini
Examiner : Zebo Peng

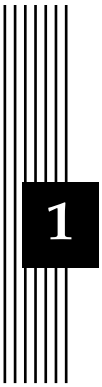
Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

To catch up with the growing trend of parallelism, this thesis work focuses on the adaption of embedded real-time systems to a multicore platform. We use the embedded system of Xilinx ZCU-102, a multicore board, as an example of an embedded system without getting deep into its architecture. First, we deal with the tasks required to be able to make an embedded system operational and discuss why they are different from those for normal computer systems. The processes it takes to make a custom operating system for the given Xilinx embedded system are examined and patching the custom operating system along with customizing it is studied. We then take a look at related work in the field of benchmarking real-time systems and embedded systems and with a good understanding of related work propose a design similar to the related work for benchmarking embedded systems. The benchmarks we use run on multiple cores and aim at challenging the Xilinx board's capabilities of running real-time tasks when the other cores on the board are occupied with performing independent tasks. We test the designed benchmarks on different conditions under two different operating systems of RT-Linux and Embedded Linux to study the differences between them. We then note how the RT-Linux would be a real upgrade for real-time systems if multicore operations are considered. The final result we have obtained is that core idling might decrease the performance of real-time tasks and RT-Linux might experience more interrupts but it is also better at recovering from interrupts.

Contents

List of Figures



1 Introduction

1.1 Motivation

In the modern day, embedded systems are being used very commonly and their number is increasing rapidly. There is a wide variety of usage for embedded systems in automotive, avionics, robotics, factory production, and medical centers. This diversity of usage has led to some embedded systems performing crucial real-time tasks. The real-time tasks are expected to have predictable and precise execution timing; therefore, the embedded system executing the real-time tasks should have a valid guarantee that its performance meets the criteria of predictability and preciseness that are required. While that factor is of consideration, the architecture of embedded systems and their designs are very complex while purposed to take up a small space. The mentioned scenario results in the embedded system cutting off corners in implementing hardware. The characteristics of sophisticated design while also having a limited amount of hardware compared to other computer systems have made benchmarking and measuring the performance of embedded systems very challenging.

The hardware of embedded systems is usually designed for a specific purpose using various architectures and patterns and these designs are not close to the patterns that we are commonly used to in the textbooks. This diversity has led to embedded systems having different characteristics in performance compared to others. Further on, many operating systems are developed for embedded systems such as VxWorks[14], Linux, RT-Linux[8], Xenomai[15], and LITMUS[11]. These operating systems are further tuned for the hardware definition that they operate on. In this thesis work, we will address the difficulties that it takes to install and operate the operating systems of Linux and RT-Linux on the embedded system of AMD Xilinx ZCU-102[2]. Further on, we will benchmark the board running the same benchmark application that is designed on the two different operating systems.

1.2 Aim

As explained in the previous section the embedded systems are diversely designed. They are often complicated and the operating systems running on them twist program execution patterns on the systems even further. In this thesis work, we take the embedded system of ZCU-102 as an example and treat it as a black-box embedded system without taking its

architecture and design into consideration. We aim to take a glance over the procedure of how difficult it is to install an operating system and how different operating systems are installed. We address how operating systems are booted on the embedded system and how programs are executed on it. We then aim at the concern over why there are few benchmarks for embedded systems and why the benchmarks are very diverse and not unified. Finally, we focus on the fact that the system has the capability of running multiple programs, we will aim to see how the system would perform in the scenario of having multiple cores operate at the same time. In that manner, we will introduce our custom benchmark which is a variation of the Taclebench benchmark[7] aimed to execute on a basis that other programs operate in parallel with it to test all cores.

1.3 Research questions

During the time of this work, we encountered many obstacles and challenges. The challenges encountered are mainly related to the three questions listed below.

1. The embedded systems have specific functionality and are usually designed for specific purposes. What are the challenges to make an embedded system operational for performing real-time applications?
2. What could be a good measure to benchmark the real-time performance of an embedded system while it has multiple applications running on different cores?
3. The RT-Preempt Linux patch aims to enhance the real-time capability of the Linux operating system. How good is the performance of RT-Preempt Linux compared to Embedded Linux in the criteria of running multicore programs?

1.4 Delimitations

The time that it took to figure out how to make the embedded system operational, and install a custom operating system on it was way greater than expected. Unfortunately, it resulted in the benchmarking phase not having enough quality as we desired. The benchmarking could be done more extensively and thoroughly, to be more precise, extract more information, and trade the process of running the program better. Nevertheless, with the perspective we had by taking the system as a black box and not delving deep, good results were obtained.

2 Background and Related Work

2.1 Background

Real-time applications are the applications that are time crucial and require to be executed on exact criteria of timing and deadlines. They usually are sequential applications doing volatile tasks that operate on the highest priority in the system. In many scenarios, embedded systems have to guarantee acceptable execution of real-time applications such as medical applications and avionics. The mentioned acceptable execution can be presented by metrics such as the number of deadlines missed, timing jitter, throughput, and analysis of response time. In this master thesis work, we focus on measuring the change in execution time and analyzing it while the system performs real-time tasks and the stress on the system is increased. The benchmarks implemented are designed to put the application in the worst possible execution pattern to provide a guarantee over the worst possible scenario.

Embedded systems encompass a wide array of devices, each tailored to unique functions and operating under specific hardware constraints. Due to speciality in use and diversity in hardware specifications, the embedded systems hardware architecture does not follow a standard pattern. Accordingly, these devices often have custom operating systems tailored for their specific architecture and needs. In this thesis work, we will focus on the embedded system of Xilinx ZCU-102 and aim to measure its multicore capabilities in running real-time sequential applications. Many custom operating systems can be used for this board such as RT-Linux[8], VxWorks[14], LITMUS[11] and Xenomai[15]. Due to a lack of time, RT-Linux and Xenomai were focused on.

The Embedded System Under Test

Unfortunately, there is a wide array of embedded systems each with its capabilities and specifications. The embedded systems often have their own custom operating systems as well. For this master thesis work, we consider the embedded system of the Xlinix ZCU-102 board developed by AMD company [2]. The board has the capability of running its operating system from an SD card, which is the capability we will use. It also has the capability of downloading the operating system from a server using TFTP boot. The system has a quad-core ARM Cortex-A53 processor. Other versions of the board also offer AMD Ryzen CPU and they have the capability of running with GPU as well but our board operates solely on the Arm CPU.

The board offers scaling of the CPU clock but during the thesis work, we kept the board clock fixed without scaling. The board has two levels of cache. The memory in use by the system is PS 4GB DDR4 64-bit SODIMM with two layers of cache.

In this thesis work the details of the board were not important to us as the board was treated as a black-box. The reason behind that is that there are so many embedded systems with different hardware specifications yet we wanted to guarantee that this research is done in general and is not fine-tuned for any type of hardware. The most important factor for us was that the system had four cores of CPU that could be utilized and we focused on making sure that during our tests all the CPU cores would be put to test.

2.2 Related Work

Although the area of benchmarking embedded systems and especially real-time benchmarking of embedded systems seems to be very diverse, this work has been heavily inspired by similar work that has been done in the past. This section will mention some of the work that has inspired this work ordered by their priority.

A Comparison of Scheduling Latency in Linux, PREEMPT RT, and LITMUS[6]

This paper does a very similar job by comparing scheduling latency of two similar Linux patches of RTPreempt Linux and LITMUS Linux. The testing is done by porting Cyclictest[10] to LITMUS[11] and also measuring scheduling overhead of scheduling using feather trace. The normal cyclic test could not be used on LITMUS so they ported cyclic test to LITMUS. They measured worst case execution time and response time. They designed a master manager program with the highest priority which then made a thread for every single core and pinned to it. Then after the execution they do a measurement phase which then they would gather the data. The reason why they did that porting of cyclic test but keeping the skeleton is that LITMUS has its own API but follows the same pattern and procedure as Embedded Linux. In the end, latencies of normal Linux, RTlinux and LITMUS were compared. The experiments were done on same hardware with 16 cores.They disabled frequency scaling, interrupts, deep sleep and optimization. The goal was to measure scheduling policy. The measuring were done in three phases, system with no background tasks and no workload, cpu bound stress and third being io bound stress. Result was that even in idle, LITMUS increases latency in average. The CPU bound tasks or stress were focused on putting cache pressure same as the work on this thesis work. The focus was mainly on L2 shared caches. The result was that under stress RTLinux could perform way better than Linux and LITMUS gave overhead which was suspected due to scheduling policy or lack of optimization. Then they do interrupt based testing. They also saw no improvement of speed up or scheduling optimization when they disabled interrupts compared to when there were interrupts. But they used interrupt benchmarks to put stress and they saw all latencies increase and RT preempt was actually suffering more than normal Linux. In the end, paper also said that we can see there are overheads but we cannot classify what type of overhead it is. So it's better to just take it as that there is a preemption or overhead and not to get too deep in it.

RT-Bench: an Extensible Benchmark Framework for the Analysis and Management of Real-Time Applications[13]

This research has done a similar work by adding capabilities of realtime benchmarking to an existing benchmark. They focused on periodic execution, memory tracing and making the benchmark portable. They then used Isolbench and SD-VBS benchmark to test their tuning of benchmark toward real-time capabilities. Their goal was to adapt already made benchmark to properties of real-time systems. The benchmark that is developed has a management program that controls the workload, scheduling, periodicity and priority of the benchmark. This

also allows a unified performance reporting and monitoring. The benchmark measures response time of tasks and measures if they have met the deadline and also utilizes performance counters that are distributed in running tasks to monitor the distribution of resources such as cpu utilization and memory access. They make the ordinary one shot benchmarks into periodic benchmarks that are suited for realtime applications which also factor out the overhead of initialization and tear-down. The benchmarks are also pinned to specific cores, with specific priorities. the memory allocation is also independent. the RT-bench has three specific components, the benchmark generator, utils, measuring tools. They ported benchmarks which then will be using same interface and using same memory, only generated using a generator. The benchmark also has a measurement segment. in this segment they did the cpu pinning, data management and cleaning along with analysis gathering using high level code of python and bash. After the measurement is done, they would plot the results using plotting tools. We did not go for such approach since we wanted our code to be compatible with bare metal. The final result was that the ARM platform would perform more predictably under stress.

On Performance of Kernel Based and Embedded Real-Time Operating System: Benchmarking and Analysis[12]

This paper focused on benchmarking Xenomai real-time patch of Linux, RTPreempt patch of Linux and eCos real-time operating system. It did a fascinating job of benchmarking in four metrics of latency, task switching, preemption time and deadlock break time. The results were that for network processing the kernel based real-time operating system (xenomai and RTPreempt) could perform better and it is also better for multi tasking. In comparison, the dedicated real-time operating system (eCos) was better in processing dedicated applications that need less context switching. The benchmark testbed of the system was custom made in C. This capability allowed the writer to test many metrics.

This is a good news for the reason that the ABB realtime systems focus on network in communication and do packet processing. The expectation of custom benchmark can focus on the field of packet trade and see where it can reach the optimality in how much packet transmittion is needed to become better operating system compared to a dedicated OS like VxWorks[14]. Using test beds that are already proven in literature is believed to give more value to the research, unfortunately, makes the work way harder and the metrics that could be obtained more limited.

Performance Comparison of VxWorks, Linux, RTAI, and Xenomai in a Hard Real-Time Application[3]

This paper did a similar job to us by comparing operating systems of normal linux, RTAI version of Linux, XENOMAI Linux, and VxWorks on a Motorola MVME5500. Their tests were focused around measuring interrupt latency, inter-process communication and rescheduling of real-time tasks. The paper indicated that in order to make operating systems of Xenomai and RTAI work on their system they had to change the operating systems and port the operating systems to their system. The tests ran on the system were designed to give the hardware 64 signal inputs and read the latency of a single output signal with oscilloscope. The timing measurements of the oscilloscope were then used to designate interrupt latency and rescheduling time. The second stage of the tests was focused on network transmission of tasks while the tasks ran on different operating systems. The board was connected to a secondary computer and outputs were changed from output signal to IP packets to be transmitted on the network. The results were that operating systems operated closely with Vxworks performing the fastest and Xenomai performing the slowest for the reason that interrupts of Xenomai are ported to the nanokernel developed for Xenomai. While running the tests, it was observed that Xenomai scheduling over-head was greater than RTAI and standard Linux

which was due to layered design of Xenomai Linux. The network transmission latency of VxWorks was greater than Xenomai and RTAI which would indicate that in scenarios where network transmission is of importance; it is better to use Linux-based real-time systems.

The mentioned work gave us a very good inspiration for how to compare the performance of operating systems. The work did a very good job by measuring the performance of their unit under test using external hardware such as an oscilloscope and another computer. Such an approach can be used to further enhance the work that is represented in this paper and expand the accuracy of the work of this paper further.

TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research[7]

The TacleBench benchmark suit is a gathering of benchmarks for embedded systems aimed to put the embedded system on a condition that execution of benchmark takes worst-case of execution in order to measure worst case execution time for the CPU. The TacleBench is implemented in C language and it does not use any other libraries to operate. It consists of many benchmarks in different criteria such as parallel, kernel, sequential, test and simulation application. We focused on the kernel set of the benchmarks for the reason that it had enough test scenarios and it covered all programming patterns. The kernel benchmark then later got adapted to parallel scenarios in order to see how the system reacts if one of the cores would run the kernel benchmark while other cores were busy with their own independent tasks.

The parallel benchmark suit was not chosen for the reason that we found it to be to small and not useful to benchmark how decision making is made when cores are stressed. Unfortunately, we found no interest in the parallel benchmark set. It would not be wise to stress the cores independently when the cores are already idle with their own parallel tasks.

Analysis and Benchmarking Performance of RealTime Patch Linux and Xenomai in Serving a Real-Time Application[4]

The work focused on comparing the performance of RT-Linux and Xenomai real-time operating system which is an operating system based on Linux. The focus of the paper was to compare the scheduling policies of the mentioned operating systems on a network-based basis. For that purpose, they implemented a real-time application using network and another computer to monitor and measure the performance of the system under test. The metrics that were aimed for comparison were process time, jitter, and throughput. They observed that Xenomai performs better when throughput is low and accuracy is of more importance and if multi-tasking is of importance RT-Linux is a better option.

This work made a good comparison of operating systems in network-based scenarios where interrupts would be more common. The final results obtained by the researchers were very close to what we obtained and the design of their benchmark was similar to ours.

How fast is fast enough? Choosing between Xenomai and Linux for real-time applications[5]

This paper compares Linux, RT-Prempt Linux and Xenomai Linux. The research was done for the reason that installing Xenomai takes a lot of effort and the researcher aims to designate if the effort justifies itself. The tests were done by using two hardware the main hardware ran the tests and another hardware gave the inputs then monitored the outputs of the under test hardware and timed it. The tests conducted both periodic and spontaneous tasks and the measurements and were designed to measure the response time of hardware under test while running the three mentioned different operating systems. The result of the research was that in all scenarios the RT preempt patch was performing better than standard Linux

in scheduling latency reduction. The research also adds that the overhead of response time added task timings by Xenomai was less than RT-Preempt.

The researchers of this work did a fantastic job and gave a very good direction to our work. The inspiration of having a hardware monitor and time another hardware has inspired our design to have a thread that manages and times another thread that is under test which will be thoroughly discussed in the methodology chapter. The test results and criteria of measurements were centered around response time and scheduling policy of the operating systems which we believe provides a piece of good information to anyone who is interested in the field as we believe it would be a good compliment to our research work.

3 Method

3.1 Preparing the Operating system of the boards.

The process started by installing Linux on the hardware. We used a variation of Yocto [9] called PetaLinux[1]. PetaLinux is developed by AMD company to build custom versions of Linux purposed for the hardware of AMD. The advantage of using PetaLinux was that the hardware configuration of the system is pre-implemented by the AMD company which made learning how to make custom operating systems for the boards easier. The embedded systems do not have a unified structure in hardware design and usually follow a customized pattern; this characteristic has led to the board having no bios or means to identify how to boot the operating system; instead, this process is created by the person creating the operating system. The custom Linux that was created for the board also came with instructions for the board on how to boot the operating system. The instructions for the board that would lead to the identification of the hardware and setting up the memory are the instruction set that are built by using PetaLinux is called the first-stage boot loader that acts as a hardware map. The first stage boot loader then starts the second stage boot loader which is similar to the bios of a normal motherboard. The second stage boot loader picks up the kernel of the operating system and all the root files that are required and starts the booting process of the system. The Linux we installed was the version 5.15 of the Embedded Linux and we used the most minimalistic Linux possible by disabling all side applications to guarantee no interference at a later stage of running the benchmarks and applications. The compiled components were mounted on an SD card and put on the board where the board would read them in read-only mode. After that, we patched the 5.15 Linux with the same RT preempt patch that was most recent designed for that specific version. We then made sure that clock scaling was disabled, and the clock was adjusted to only 1kHZ on both operating systems. The result of this process was that an identical Linux was made with the same root file system and kernel but now the scheduler of the kernel is patched by the decision-making protocols of RT-Preempt Linux. For ease of access, we proceeded by installing the C++ library on the root file system of both versions of Embedded Linux and RT Linux and provided identical first-stage boot loader and second-stage boot loader to boot up the different operating systems to guarantee all stages are the same except for the operating system kernels themselves.

There was also an attempt to install VxWorks, however, it was not successful. VxWorks can also be installed on the board by transitioning the handle from the second-stage boot

loader designed by PetaLinux to the second-stage boot loader designed by VxWorks and then booting the operating system with the same pattern. After the process, we noticed a strange pattern of booting up that resulted in the system crashing after boot so we abandoned it.

3.2 Running TacleBench on The System

We used cross-compiling using the compiler provided by ARM. The idea of cross-compile was that it would be independent of the operating system and tailored by the machine language and architecture without interference from the embedded system itself. By doing that we could take factors of compiler optimization for the operating system out of the way. The TacleBench consists of many categories of benchmarks all purposed around calculating runtime. We chose the single kernel benchmark as we felt that it contained all the patterns that a program might need such as array traversal, function calling, matrix multiplication, and complex arithmetic. We put the board on the test and measured the runtime of the kernel benchmarks by timing only the execution time of benchmarks and not considering the time it takes for memory allocation.

3.3 Implementation of Custom Benchmark

Our aim by using the embedded system is to utilize all the capabilities the system has to offer, and measure if using all of them can align with our requirement of running real-time systems. The factor we focused on was that the system has four cores that it operates on and it is better to design a multicore benchmark for the board that puts all the cores to the test. With that in mind, we had two scenarios we could design our benchmark around; a scenario would be that it would run tasks in parallel. The other would be that we would divide a task into parallel subtasks and run them on each core; The first option would be a better option since most real-time operations are sequential and dividing them would not be beneficial to us. On the other hand, the parallel benchmark suit of TacleBench did not pick the interest that was in our goal. Instead, the kernel benchmark of TacleBench was chosen that would engulf all criteria and patterns that any software program would have.

The custom benchmark designed for the board was heavily inspired by Cyclic Test[6] and RT-Bench[13]. The design of the benchmark is implemented in the way that it would contain a manager thread that would monitor and create the sub-threads of the benchmark, a benchmark thread, and two stress threads. The focus of the benchmark was to utilize all cores and the management thread guarantees that. Compared to the original benchmark, we adapted some needed changes in initialization or running in order to tune the benchmarks for running them multiple times and make sure all the memory accesses and allocations are independent. The process of benchmarking starts by pinning the management thread on a core and running it with topmost priority. The management then gives the choice to the user to run any of the TacleBench kernel benchmarks that are implemented. The benchmarks of table bench are pinned to another core where they would run deterministically and sequentially while the management would monitor memory association, initialization, and memory deletion after the stress process ends. After the control is given to the benchmarking thread to run on a core, it would be pinned to that specific core with topmost priority, and the start of its execution time will be timestamped. When the execution is over, the end of execution would be timestamped by the benchmark thread then the results and memory allocation would be handed back over to the management thread to record the final results and execution time along with deallocation of memory. The other segment that is managed by the management thread is the stress thread segment which consists of two threads. The user can choose between having no stress which would run normal TacleBench or running one or two stress threads in parallel. One stress is pinned to one core or two stresses each pinned to a core. The purpose of stress is to put stress onto the cache, memory transfer bus, and memory of the system. The stresses

```
void *example(void *data)
{

  allocate memory
  allocate/initilize clock
  double *return_pointer = (double *)data;
  double res = 0;
  for (int i = 0; i < iteration_count; i++)
  {
    start_time = clock();
    //timing window
    initilize benchmark();
    run benchmark();
    int result = get return of benchmark();
    //end of timinig window
    end_time = clock();
    report()
    write result on global array()
  }
  *return_pointer=dif;
  return data;
}
```

Figure 3.1: An Example of Benchmark Thread

are programmed in a way that it constantly accesses a wide local array that is bigger than the cache line which would cause cache misses which would congest shared cache and memory bus with cache misses. It would then force memory to retrieve new information hence the whole memory unit would be under constant stress. Every thread whether being a benchmark thread, manager, or stress thread has its own memory allocation, therefore we could guarantee that there would never be any racing condition. Note that the memory that is used by the stresses is local memory to every thread allocated in the heap so we could guarantee no deadlock or race condition.

3.4 Overall Theory and Guarantee

So far we have focused on porting TacleBench to a scenario that runs sequentially as before but now other programs (which here are taken by the extreme condition and represented by the stress) are present on the other cores. Both operating systems run on the same hardware and no configuration of hardware is changed. The operating systems themselves have the same clocks, the same root file system, and no side applications. The clock scaling of the operating systems is disabled and they are both made the same way. The benchmark implemented for the test has been constructed by using cross-platform and using compiler of ARM to guarantee there would be no optimization purposed for the operating system and all fields are equal.

4 Results

4.1 Running the Benchmark on RT-Preempt

After designing the benchmark, implementing it then compiling it for the system without considering the operating system. We proceeded to carefully prepare an equal condition for the system to measure the performance of RT-Linux and Embedded Linux on the ZCU-102 board. We first prepared the system for running the benchmark and checked the status of the system. The system was connected to a computer using the USART gate which we used to boot the system and communicate with it. The operating systems and the benchmark were on an SD card used by the system to boot and read the benchmark. After the system was operational we checked the Linux version for each set of benchmarks. The already running processes on every thread were monitored and we saw minimal applications running. We used the TOP command to monitor running processes and we saw that zero percent of our desired cores were being used. The only task allocation of our targeted core was cycling between the communication application that we used for mentoring the program with our main computer and the root system command. The applications that were already running were suspected to be the main applications of the system so they were not touched. After monitoring the state of the application, the custom benchmark was then loaded up and the priority of it was changed to maximum priority. We paused the custom benchmark and used the "ps" command to track the ID of our custom benchmark changed its priority to maximum priority then resumed the benchmark. Unfortunately due to having only one interface of monitoring and communication with the system, we could not constantly monitor the state of allocated CPU cores as the benchmark was running for reason that the communication interface was being occupied by the benchmark and the only option to check the CPU allocation would have been achievable by pausing the benchmark.

Each benchmark was run one thousand times and experiments were done repeatedly. Since the adaption of Tacle-bench and the benchmark itself were both deterministic benchmarks, running them repeatedly would not result in different sets of results. While running the processes on conditions of having no stress, one stress pinned to a core and finally having two stress pinned to each core. We saw that the run time of the benchmark was the same and on some occasions, a spike in the runtime would be sighted.

The execution run time analysis of the benchmark on normal conditions without any stress is as below. All the recordings are in milliseconds.

Run Time Analysis of the Benchmark While Having No Stress					
Benchmark Name	Average	Median	Variance	Maximum	Category
bubble sort	304.223	302.0	19.336	314	2
binary search	4.599	5	0.6983	15	1
bitonic search	23.903	24.0	2.8204	37	1
complex	6.554	6.0	1.0581	18	1
count negative	38.296	38.0	4.4127	51	1
crc	10.057	10.0	4.1278	64	1
factorial	3.444	3.0	0.6735	15	1
filter bank	13098.85	13098.0	13.995	13124	3
fir2dim	11.277	11.0	1.4176	24	1
irr	4.516	4.0	0.5803	15	1
insert sort	4.696	5.0	0.8664	17	1
jfdctinit	10.324	10.0	1.1681	23	1
ims	392.509	389.0	22.418	402	2
ludcmp	12.457	12.0	1.7378	24	1
matrix	34.457	34.0	3.6157	47	1
minver	8.747	9.0	0.6996	20	1
prime	3.712	4.0	0.6677	16	1
recursion	6.41	6.0	0.8927	19	1
st	293.797	291.0	20.500	305	2

We ran each benchmark one thousand times and during that time we saw run time to be nearly consistent without any fluctuations, we could observe that the variance of run times in all benchmark cases are very low for all the benchmarks. The benchmarks could be classified into three categories relatively to their runtime. The first category is the benchmark cases of having a low run time that is less than 30 milliseconds, run time of around 300 milliseconds, and a runtime of above one thousand milliseconds. Category two consists of benchmarks that heavily depend on memory and category three is unique to a string filtering and parsing application known as filterbank that heavily depends on memory. It makes sense for the reason that we expect memory retrieval to be the most time-consuming process here. Other applications of category one also use memory but the memory access is so bounded that all memory can be cached or memory accessing is optimized like quick sort algorithm and insert sort.

4.2 Run Time Analysis of the Benchmark While Having One Stress on Read While Operating on RT-Preempt

We then proceeded by putting one stress on another core that was reading from a memory array having a size bigger than a cache line but with no race condition or memory interference with the benchmark or management threads. The timing analysis results are as below.

Run Time Analysis of the Benchmark While Having One Stress on Read					
Benchmark Name	Average	Median	Variance	Maximum	Category
bubble sort	633.991	302.0	227818.797	1330	2
binary search	10.599	4.0	6181.41961	1024	1
bitonic search	47.084	24.0	23276.4253	1050	1
complex	15.625	6.0	9218.27464	1027	1
count negative	80.609	38.0	41663.4836	1062	1
crc	20.113	10.0	10198.1383	1033	1
factorial	5.514	3.0	2061.17898	1019	1
filter bank	26240.608	26144.0	786277.752	27946	3
fir2dim	26.453	11.0	15328.5543	1036	1
irr	7.583	4.0	3094.48960	1026	1
insert sort	9.765	5.0	5148.78255	1028	1
jfdctinit	18.469	10.0	8193.51655	1028	1
ims	774.495	389.0	243286.742	1416	2
ludcmp	24.735	12.0	12285.3941	1033	1
matrix	60.711	34.0	26138.4098	1052	1
minver	12.808	9.0	4111.91505	1028	1
prime	5.898	4.0	2065.70930	1021	1
recursion	9.522	6.0	3096.54005	1030	1
st	575.618	291.0	208577.759	1321	2

Overall the pattern of execution time did not change much. Normally the run time was the same, however, on some occasions, there were sudden jumps in execution time where the execution time would increase to be more than triple the execution time of the unstressed scenario and sometimes greater. We saw that there were few cases of high execution time on the benchmarks of category one which had low run time. The benchmarks of category two had spikes around every three runs and the benchmarks of category three changed execution time drastically while also experiencing spikes. As can be seen in the table, the median remained the same for all benchmarks except for the "filter bank" benchmark, the average increased slightly which indicates that only a few cases were changed but the variance changed drastically which indicated that the impacts of spikes were drastic which indeed they were. Our guess was that due to heavy stress on the board, if the board was running to meet expected deadlines, only one core being stressed would be enough to make sure some occasions the application that would expectedly run normally would surely miss the deadlines.

4.3 Run Time Analysis of the Benchmark While Having Two Stress on Read While Operating on RT-Preempt

After finishing the benchmarking of system by having three cores occupied with a management thread, a benchmark thread and a stress thread each on different core, we proceeded to add another stress thread on the fourth core to have all cores occupied. Both stresses had their own independent memory which were bigger than cache line accessing them constantly to put stress on memory transfer unit of the system. The results of the experiments are given as follows.

Run Time Analysis of the Benchmark While Having Two Stress on Read					
Benchmark Name	Average	Median	Variance	Maximum	Category
bubble sort	920.355	302.0	867607.6266	2341	2
binary search	17.941	5.0	25404.07959	2033	1
bitonic search	82.506	24.0	115114.3022	2057	1
complex	16.811	7.0	20396.47575	2040	1
count negative	114.867	38.0	149313.5008	2069	1
crc	26.427	10.0	32381.83850	2039	1
factorial	9.645	4.0	12125.64461	2021	1
filter bank	39541.709	39209.0	3042647.167	42244	3
fir2dim	35.641	11.0	48350.78089	2036	1
irr	14.805	5.0	20335.82279	2040	1
insert sort	8.921	5.0	8069.738497	2017	1
jfdctinit	28.652	10.0	36417.99889	2036	1
ims	1146.781	391.0	957574.2913	2427	2
ludcmp	26.756	12.0	28452.50296	2039	1
matrix	89.074	34.0	107393.7462	2069	1
minver	27.222	9.0	36437.99270	2039	1
prime	16.143	4.0	24279.60415	2029	1
recursion	14.69	7.0	16143.23113	2029	1
st	872.688	292.0	838161.7103	2351	2

After adding the second stress on read, we saw that the impact of the spikes doubled and we could see the records of maximum increasing in double their value, the number of spikes were not follow a particular pattern but overall their number either remained the same or increased by only a few amount. The impact of stress on the categories of the benchmark was exactly the same as the time when there was only one stress on the benchmark and overall if the spikes were excluded from the runtimes, the other behaviors were identical.

4.4 Run Time Analysis of the Benchmark While Having One Stress on Write While Operating on RT-Preempt

We then converted the process of stress threads to writing on a random access array instead of reading from it. The same as before, we set one thread for stressing one core which was for writing a random access array, one thread for running the benchmark, and one thread for managing the benchmark thread and the stress thread. Each thread was pinned to a core resulting in three cores being occupied. The results of the experiments are given as follows.

Run Time Analysis of the Benchmark While Having One Stress on Write					
Benchmark Name	Average	Median	Variance	Maximum	Category
bubble sort	609.835	302.0	216925.9477	1498	2
binary search	4.57	4.0	1.294394394	16	1
bitonic search	49.1	24.0	25067.94194	1052	1
complex	10.633	6.0	4053.928239	1017	1
count negative	74.527	38.0	35708.93820	1068	1
crc	18.1	10.0	8203.163163	1039	1
factorial	7.573	3.0	4116.589260	1024	1
filter bank	26179.763	26114.0	725491.1800	28131	3
fir2dim	25.523	11.0	14191.46693	1033	1
irr	7.652	4.0	3060.913809	1018	1
insert sort	9.843	5.0	5145.285636	1025	1
jfdctinit	24.652	10.0	14218.53142	1034	1
ims	777.249	389.0	241689.9889	1423	2
ludcmp	21.665	12.0	9193.115890	1032	1
matrix	66.772	34.0	31888.16017	1061	1
minver	19.98	9.0	11145.38498	1025	1
prime	10.886	4.0	7120.787791	1024	1
recursion	8.461	6.0	2056.472951	1023	1
st	577.573	291.0	207678.7514	1333	2

We saw that the results were exactly identical to the results of benchmarking when the stress was on reading from the array. We believe this similarity was expected due to the reason that memory accessing of cache and memory bus is not secluded to writing or reading.

4.5 Run Time Analysis of the Benchmark While Having Two Stress on Write While Operating on RT-Preempt

In the next step, we added another stress on the fourth core which was on writing the memory, and we saw that the results were identical as if the stresses were on reading from memory. The results of the experiments are given as follows.

Run Time Analysis of the Benchmark While Having Two Stress on Write					
Benchmark Name	Average	Median	Variance	Maximum	Category
bubble sort	895.218	302.0	847908.0605	2427	2
binary search	10.932	5.0	12260.61198	2032	1
bitonic search	76.606	24.0	103769.1879	2065	1
complex	12.942	7.0	12241.85048	2042	1
count negative	111.207	38.0	142299.8580	2075	1
crc	22.568	10.0	24448.80418	2039	1
factorial	11.857	4.0	16339.34789	2036	1
filter bank	39557.545	39240.0	3032434.394	41729	3
fir2dim	35.855	11.0	48768.22219	2052	1
irr	12.888	5.0	16193.26672	2029	1
insert sort	7.011	5.0	4143.490369	2040	1
jfdctinit	28.999	10.0	36623.34033	2059	1
ims	1192.304	390.0	979007.9114	2433	2
ludcmp	33.232	13.0	40707.53571	2052	1
matrix	96.369	34.0	120258.161	2075	1
minver	25.257	9.0	32554.24119	2043	1
prime	4.116	4.0	2.703247247	18	1
recursion	16.839	6.0	20308.54762	2031	1
st	867.372	292.0	833386.6702	2336	2

The results we observed were identical to the time the stress was on reading from memory. The spikes doubled on impact and delays were doubled, the number of spikes either the same or increased in numbers.

4.6 Run Time Analysis of the Benchmark While Having No Stress on Embedded Linux

After running all the benchmarks on the RT-Linux, we changed the operating system to Embedded Linux and repeated the process again. We followed the same procedure as before making sure that both benchmarks were in identical condition. After that, we ran the same benchmark code again. Due to the program being cross-compiled for the machine architecture we could guarantee that the same application was running for both operating systems. The runtime analysis of the benchmark, while there is no stress on the core, is as below.

Embedded Linux Run Time Analysis of the Benchmark While Having No Stress					
Benchmark Name	Average	Median	Variance	Maximum	Category
bubble sort	300.76	300.0	3.63603603	310	2
binary search	3.388	3.0	0.31376976	12	1
bitonic search	22.672	23.0	0.68109709	35	1
complex	5.342	5.0	0.30333933	14	1
count negative	36.855	37.0	0.76674174	47	1
crc	8.901	9.0	3.41060960	63	1
factorial	2.455	2.0	0.25022522	4	1
filter bank	13010.42	13009.0	6.62822822	13026	3
fir2dim	10.144	10.0	0.58584984	24	1
irr	3.45	3.0	0.30380380	10	1
insert sort	3.56	4.0	0.41081081	14	1
jfdctinit	9.13	9.0	0.26336336	18	1
ims	388.242	388.0	4.89232832	399	2
ludcmp	11.292	11.0	0.58331931	21	1
matrix	33.122	33.0	0.94406006	43	1
minver	7.792	8.0	0.37511111	17	1
prime	2.741	3.0	0.20612512	6	1
recursion	5.327	5.0	0.47454554	15	1
st	290.441	290.0	4.34286186	303	2

We saw that the runtime pattern follows a consistent pattern with no fluctuations or spikes, the same as when the benchmark was running on RT-Preempt without any interruptions. The only difference when compared to RT-Preempt was that overall the runtime was faster, which we believe must be due to scheduling policy differences between RT-Preempt Linux and Embedded Linux.

4.7 Run Time Analysis of the Benchmark While Having One Stress on Read While Operating on Embedded Linux

We then proceeded by putting one stress on a core and then running the benchmark again. The stress would read from an array having a size bigger than a cache line to put stress on the memory unit. The timing analysis results are as below.

Embedded Linux Run Time Analysis of the Benchmark While Having One Stress On Read					
Benchmark Name	Average	Median	Variance	Maximum	Category
bubble sort	589.834	300.0	1080869.96240	4343	2
binary search	3.399	3.0	0.50630530530	9	1
bitonic search	42.69	23.0	80463.5494494	4047	1
complex	9.523	5.0	16181.3267977	4028	1
count negative	73.153	37.0	144410.682273	4070	1
crc	21.072	9.0	48395.8166326	4038	1
factorial	2.587	3.0	0.63106206206	8	1
filter bank	26048.226	25051.0	3711691.38230	29966	3
fir2dim	18.351	10.0	32292.2120110	4032	1
irr	7.596	4.0	16148.3511351	4022	1
insert sort	7.702	4.0	16155.6748708	4023	1
jfdctinit	13.262	9.0	16135.0744304	4026	1
ims	785.818	388.0	1443066.48135	4425	2
ludcmp	11.422	11.0	1.13505105105	20	1
matrix	66.277	33.0	129347.774045	4070	1
minver	16.032	8.0	32241.5064824	4026	1
prime	6.913	3.0	16177.9914224	4025	1
recursion	17.546	5.0	48461.5914754	4030	1
st	579.711	290.0	1081312.67215	4328	2

We saw that spikes were once again present though their number was almost the same as when the benchmark was operating with stress in RT-Linux. It could be observed that category one had only a few spikes in a thousand iterations and in some cases, none; category two was more impacted by spikes that could not be ignored category three changed runtime entirely; these were the same events as we encountered previously while running the benchmark on RT-operating system. The median for the benchmarks remained the same which indicated that the majority of the benchmarks were running the same. The maximum indicates the impact of the spikes which were four times more impactful than spikes of RT-Preempt. Compared to RT-Preempt, the frequency of the spikes was less as well. Another event we noticed was that in some cases benchmark spikes could not be seen. These were the benchmarks for which their runtime was very low. However, this event happened randomly and when we restarted the benchmark on different times, they might experience a spike and some other benchmarks might not.

4.8 Run Time Analysis of the Benchmark While Having Two Stress on Read While Operating on Embedded Linux

We then proceeded by adding another stress on the system resulting in all the cores being occupied. The first core was having management pinned to it with top priority, the second core had the benchmark thread pinned to it and the third and fourth cores had stress threads pinned to them which constantly read from their independent memory without race condition. The timing analysis of benchmarking while having two stress is as below.

Embedded Linux Run Time Analysis of the Benchmark While Having Two Stress on read					
Benchmark Name	Average	Median	Variance	Maximum	Category
bubble sort	880.085	301.0	4304072.232007	8337	2
binary search	35.653	4.0	256553.2198108	8032	1
bitonic search	54.822	23.0	256767.2856016	8053	1
complex	13.598	5.0	64553.14754354	8040	1
count negative	117.29	37.0	637877.8997997	8077	1
crc	33.14	9.0	192542.8452452	8030	1
factorial	2.611	3.0	0.696375375375	8	1
filter bank	39049.461	37050.0	14842069.99847	46857	3
fir2dim	26.474	10.0	128804.3416656	8047	1
irr	7.673	4.0	16075.49356456	4013	1
insert sort	11.86	4.0	64259.68608608	8020	1
jfdctinit	41.473	9.0	256485.0102812	8033	1
ims	1134.827	389.0	5432806.619690	8444	2
ludcmp	19.525	11.0	64393.82319819	8036	1
matrix	105.493	33.0	574267.8457967	8061	1
minver	32.243	8.0	192374.7947457	8027	1
prime	2.955	3.0	1.286261261261	27	1
recursion	21.637	5.0	128638.1834144	8031	1
st	870.285	291.0	4306696.085860	8339	2

After running the test we could observe the same pattern as having two stress on the core while operating on the RT-Linux. The spikes became more frequent and their impact doubled as well. compared to RT-Linux, the number of spikes was less but the impact was more significant.

4.9 Run Time Analysis of the Benchmark While Having One Stress on Write While Operating on Embedded Linux

We then repeated the process by converting the stress threads pinned to cores to writing on a memory array instead of writing. The timing analysis results are as below.

Embedded Linux Run Time Analysis of the Benchmark While Having One Stress on Write					
Benchmark Name	Average	Median	Variance	Maximum	Category
bubble sort	602.645	300.0	1127028.0690440	4362	2
binary search	3.568	3.0	1.1285045045045	16	1
bitonic search	46.977	23.0	97074.142613613	4057	1
complex	17.75	5.0	48563.643143143	4036	1
count negative	69.329	37.0	128958.77153053	4073	1
crc	9.224	9.0	7.0468708708708	66	1
factorial	2.679	2.0	2.0900490490490	13	1
filter bank	26120.504	25084.0	3891807.0590430	29414	3
fir2dim	10.405	10.0	3.3423173173173	33	1
irr	11.651	3.0	32321.060259259	4034	1
insert sort	7.845	4.0	16220.753728728	4031	1
jfdctinit	17.455	9.0	32446.426401401	4041	1
ims	734.853	388.0	1278727.2386296	4477	2
ludcmp	19.637	11.0	32686.876107107	4067	1
matrix	61.539	33.0	113173.86634534	4078	1
minver	16.306	8.0	32484.114478478	4046	1
prime	3.029	3.0	3.5477067067067	41	1
recursion	17.769	5.0	48620.273912912	4040	1
st	584.702	290.0	1101718.1653613	4357	2

We observed the same as operating on RT-Preempt Linux, the Embedded Linux also performed identically, hence the result of having stress on write was treated the same as being on read. The final results were identical to the test of having one stress on read as if we just ran another instance of tests on the board.

4.10 Run Time Analysis of the Benchmark While Having Two Stress on Write While Operating on Embedded Linux

After occupying three cores out of four cores that the system offers us, we proceeded with the final stage of the testing by occupying all four cores with one manager thread pinned to the first core, one benchmark thread, and two stress cores writing on an array each pinned to a core. The timing analysis results are as below.

Embedded Linux Run Time Analysis of the Benchmark While Having Two Stress on Write					
Benchmark Name	Average	Median	Variance	Maximum	Category
bubble sort	863.284	301.0	4207162.8201641	8377	2
binary search	11.791	4.0	64550.882201201	8038	1
bitonic search	79.315	23.0	449820.34211711	8067	1
complex	5.84	6.0	3.8082082082082	21	1
count negative	101.629	37.0	514135.49285185	8122	1
crc	57.643	9.0	385829.64719819	8062	1
factorial	2.928	3.0	2.8957117117117	16	1
filter bank	39140.824	37091.0	15332098.803827	48654	3
fir2dim	34.838	10.0	194263.56732332	8082	1
irr	3.903	4.0	2.9905815815815	19	1
insert sort	12.155	4.0	64740.541516516	8050	1
jfdctinit	25.814	9.0	129796.95235635	8094	1
ims	1143.833	388.0	5504597.0902012	8468	2
ludcmp	35.947	11.0	194309.41961061	8089	1
matrix	65.79	33.0	257833.73763763	8075	1
minver	32.617	8.0	193504.29460560	8052	1
prime	11.173	3.0	64496.938009009	8034	1
recursion	13.849	6.0	64695.509708708	8049	1
st	853.784	291.0	4213284.9102542	8379	2

We saw that results were once again were identical to having four cores occupied but the stress threads being on memory read. Out of completion and making sure no stone is unturned we have done the writing benchmarks to complete as well. Nevertheless, it seems that the system always has the same behavior whether stress on memory is on writing or reading from memory.

4.11 Illustrations Based on The Analyzed Data

After analyzing the data, for better understanding, we have used line graphs to illustrate the different sections of experiments that have similarities to have a better understanding of the system when conditions are similar.

System under no stress

A graph presentation of the system under no stress is as below. The system performs very consistently, with very low to no fluctuation and variance, as shown in the tables before. The red line presents the "complex" arithmetic benchmark which is a representative example of the category one benchmark, the green line presents "bubble sort" which is a representative example of category two and the blue line is the "filter bank" benchmark which represents category three. The graph's horizontal axis shows the system's behavior in different iterations and the vertical axis presents milliseconds required to finish the execution time. The behavior of the system under no stress is shown in Figure 4.1.

Behavior of system under one stress

The spikes introduced to the system are shown in Figure 4.2. The benchmarks of category one shown by the red line graph experienced the least amount of spikes and in the majority of iterations, their behavior remained the same. The benchmarks of category two experienced spikes around once every three iterations and the benchmarks of category three experienced an increase in runtime along with spikes. For ease of visibility, the iterations presented are 200 iterations instead of a total of 1,000 iterations.

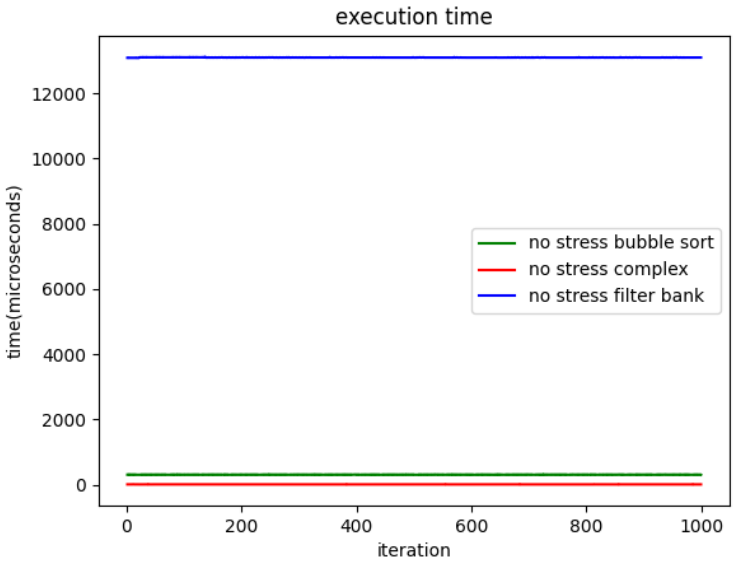


Figure 4.1: Execution time of the benchmarks under no stress while operating on RT-Linux.

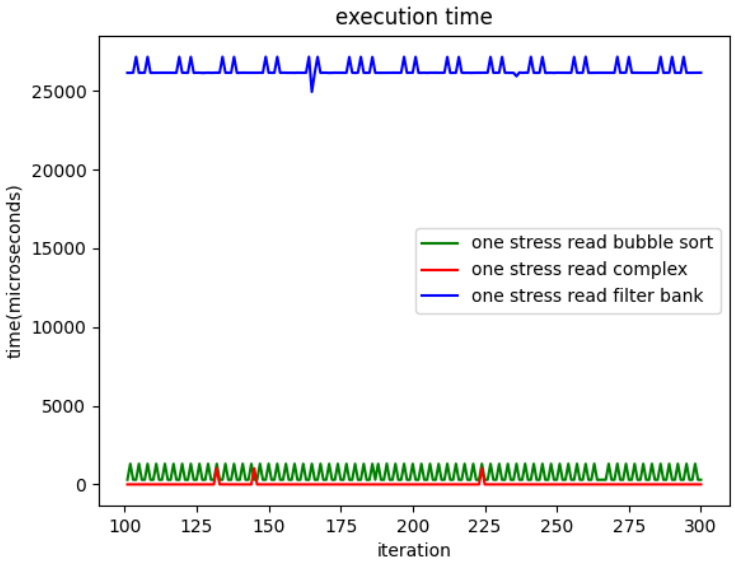


Figure 4.2: Execution time of the three categories under one stress of read while operating on RT-Linux.

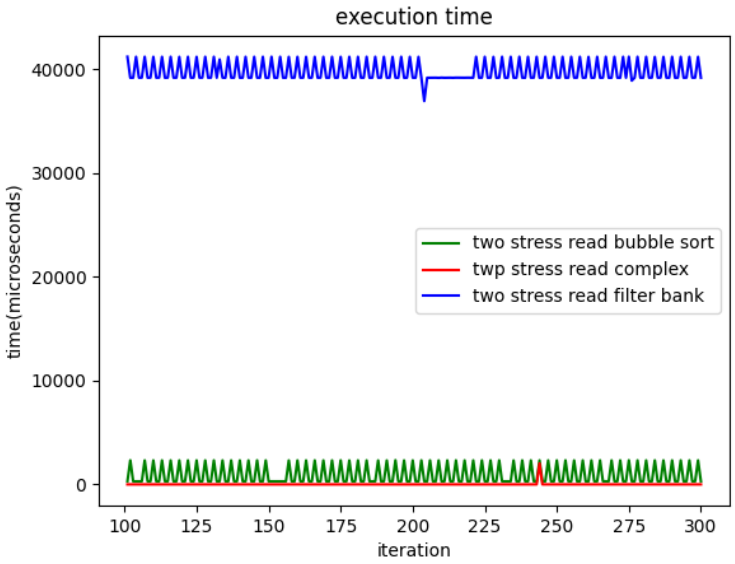


Figure 4.3: Execution time of the three categories under two stress of read while operating on RT-Linux.

Behavior of system under two stress

Figure 4.3 presents the behavior of the system while two cores are under stress. The number of spikes for benchmarks of categories one and three have remained the same while the number of stress for benchmarks of category three has increased drastically. For all benchmarks, the impact of stress has doubled as well.

Comparison of category one under various stress conditions

The category one benchmarks are the benchmarks with a low run time that usually do not need much memory accessing or all memory that needs to be accessed is cached. In Figure 4.4, we see the impact of spikes on the system when it is under stress in the one thousand iterations of benchmarking. The green line presents the behavior of the benchmark under normal conditions which is a consistent line. The red line graph shows the behavior of the benchmark with one stress. As it can be seen, the system has few spikes and besides that, the behavior of the system is identical to the condition that it did not have stress. The blue line shows the behavior of the benchmark when two cores are stressed, as it can be seen the impact of spikes has doubled. There is no prediction over the number of spikes and when they happen so we do not have an explanation for why the pattern of spikes or their number is shown as presented in the figure.

Comparison of category two under various stress conditions

In Figure 4.5 we see the comparison of category two under various stress conditions. For ease of visualisation, the impact is presented in the number of only fifty iterations. We see that the impact of the spikes is doubled when the stress is doubled, the normal run time without stress is the same and the number of spikes is the same as well with some timing change that happens randomly. We do not have any explanation for why the timing change is presented or why it happens.

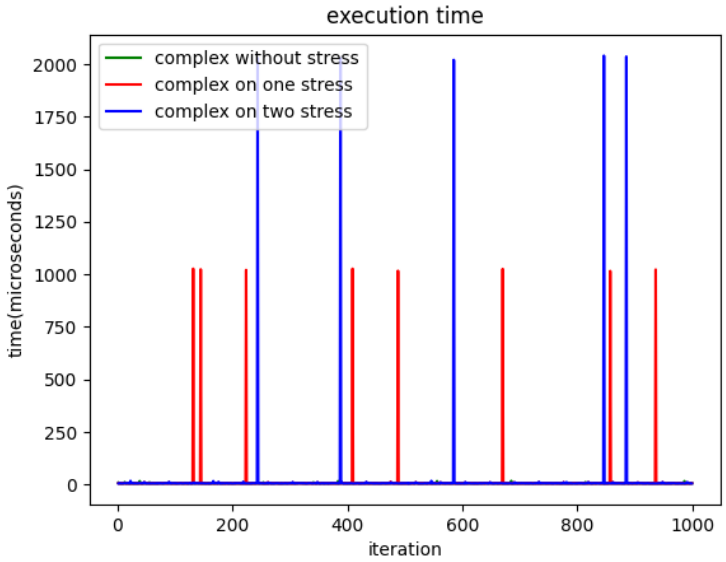


Figure 4.4: Execution time comparison of category one while operating on RT-Linux.

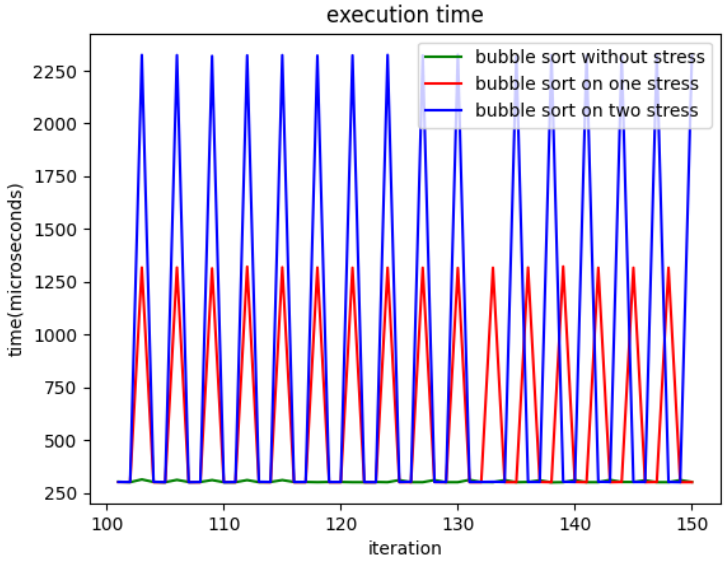


Figure 4.5: Execution time comparison of category two while operating on RT-Linux.

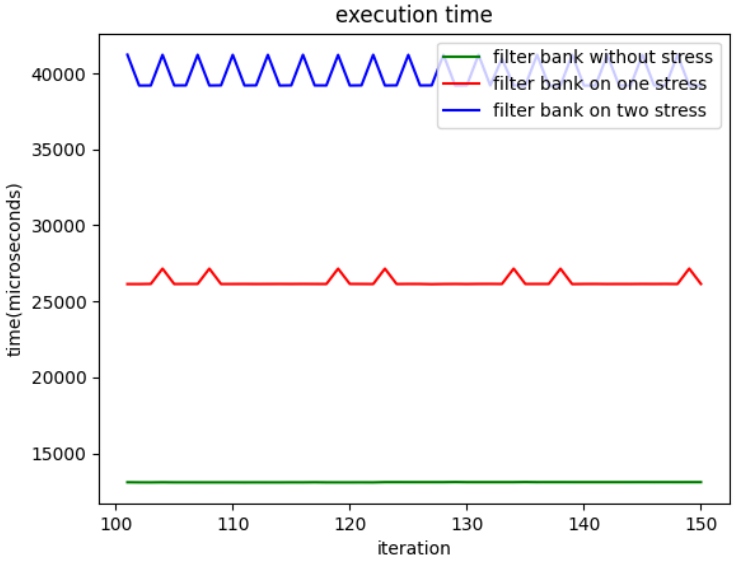


Figure 4.6: Execution time comparison of category three while operating on RT-Linux.

Comparison of category three under various stress conditions

In Figure 4.6 we see the comparison of category three under various stress conditions. the visualization is made under fifty iterations so that spikes are visible. We see that execution time changes drastically when stress is introduced and some spikes happen which also increase in number when stress is increased. We do not know when the spikes happen yet we know they are present.

Graphs of system under writing stress

As we discussed earlier, the system behaved exactly similarly when stress was on writing on memory instead of reading for memory. Nevertheless, it's better to present figures of our discovery. Figures 4.7 and 4.8 sequentially demonstrate the behavior of the system under one stress thread and two stress threads. As we can see the figures are very similar to Figures 4.1 and 4.2.

Graphs of the system while operating on Embedded Linux

Figure 4.9 shows the performance of the system while there is no stress on the system. As it can be seen the system does not have any fluctuations or spikes. Figure 4.10 shows the performance of the system while one stress thread is active, the spikes compared to Figure 4.3 are less frequent but more impactful. Figure 4.11 shows the performance of the system while having two stress threads running on two different cores, we see that the behavior of the system is similar to Figure 4.4 with the difference that now impacts the spikes but their frequency is less which is the same behavior pointed out frequently on this master thesis report. Figure 4.12 shows the performance of Category One while Embedded Linux is running Figure 4.13 is Category Two along with figure 4.14 is of Category Three. For the reason that the same explanation is valid for the graphs as we discussed before, we do not explain the same encounters as we explained for the previous figure. Figure 4.15 and Figure 4.16 showcase the performance of the system when stress threads are written on memory array instead of reading from it.

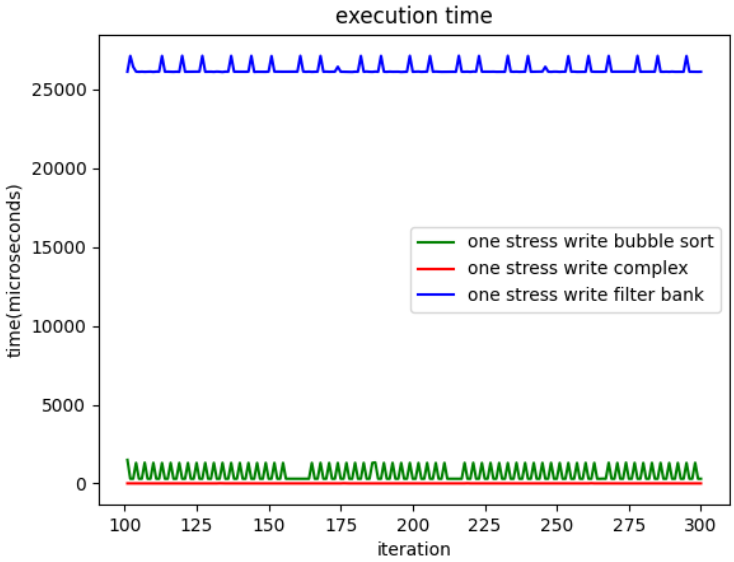


Figure 4.7: Execution time of the three categories under one stress of write while operating on RT-Linux.

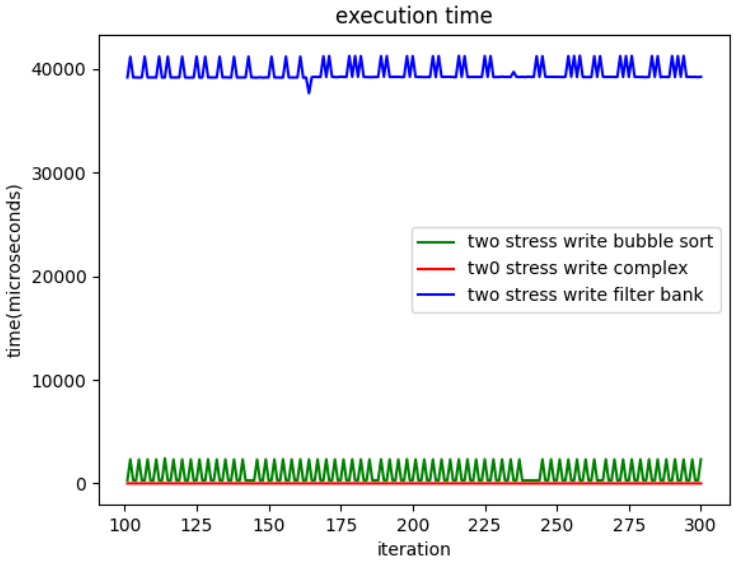


Figure 4.8: Execution time of the three categories under two stress of write while operating on RT-Linux.

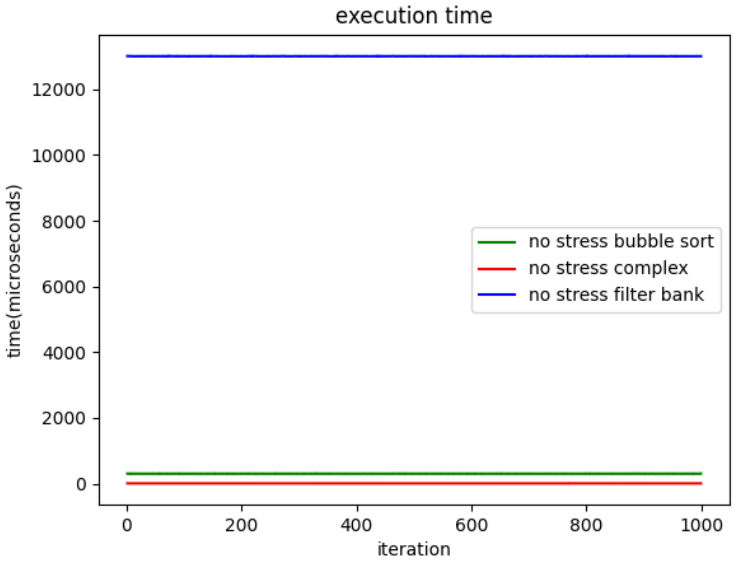


Figure 4.9: Execution time of the benchmarks under no stress while operating on Embedded Linux.

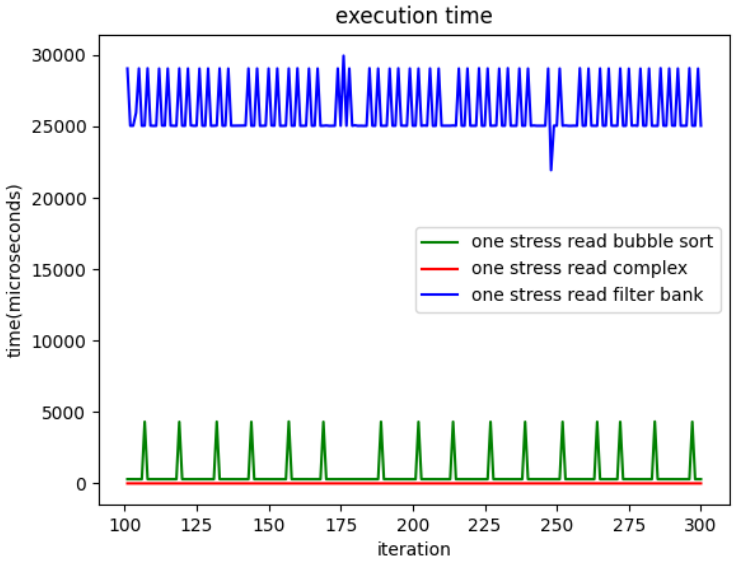


Figure 4.10: Execution time of the three categories under one stress of read while operating on Embedded Linux.

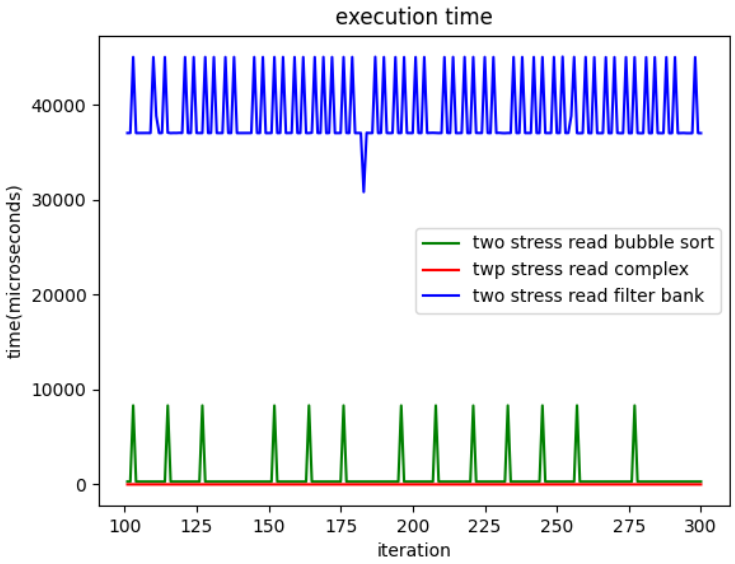


Figure 4.11: Execution time of the three categories under two stress of read while operating on Embedded Linux.

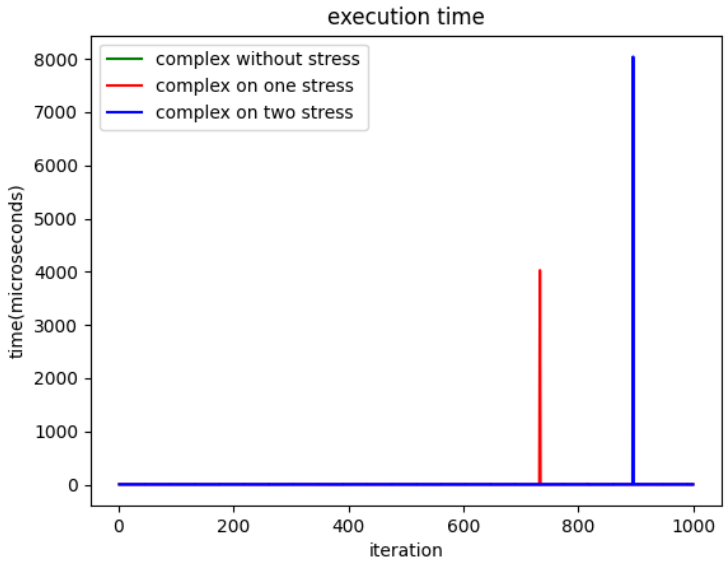


Figure 4.12: Execution time comparison of category one while operating on Embedded Linux.

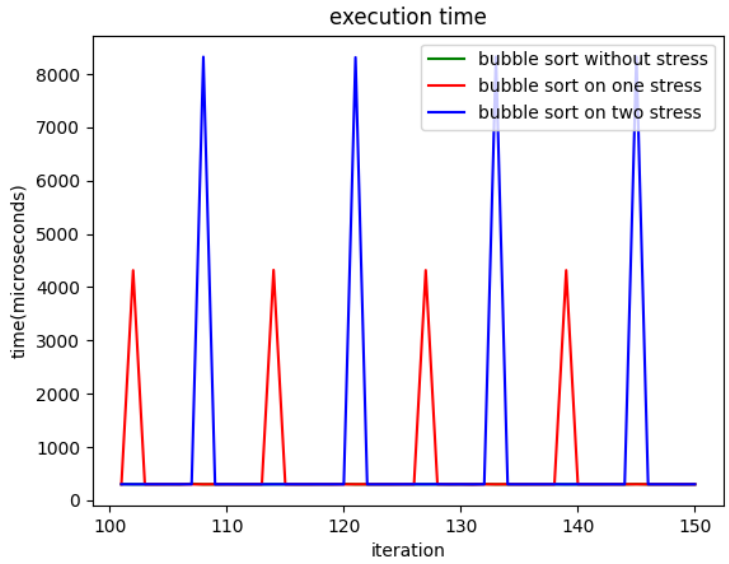


Figure 4.13: Execution time comparison of category two while operating on Embedded Linux.

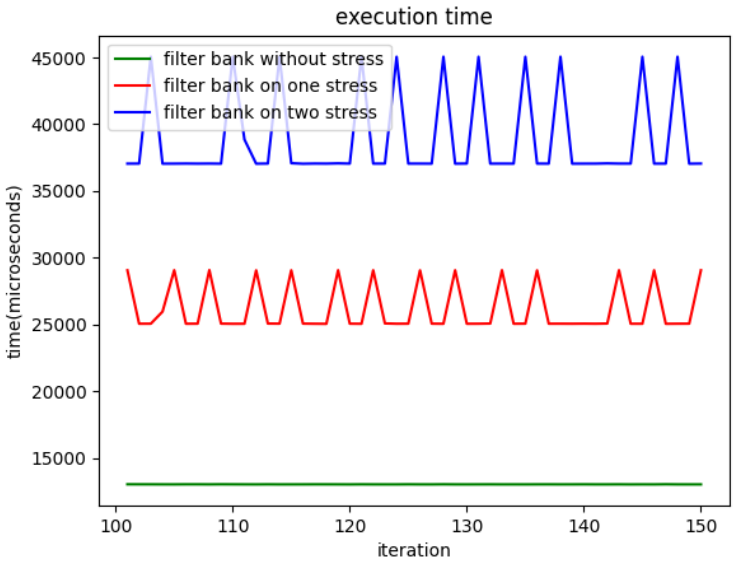


Figure 4.14: Execution time comparison of category three while operating on Embedded Linux.

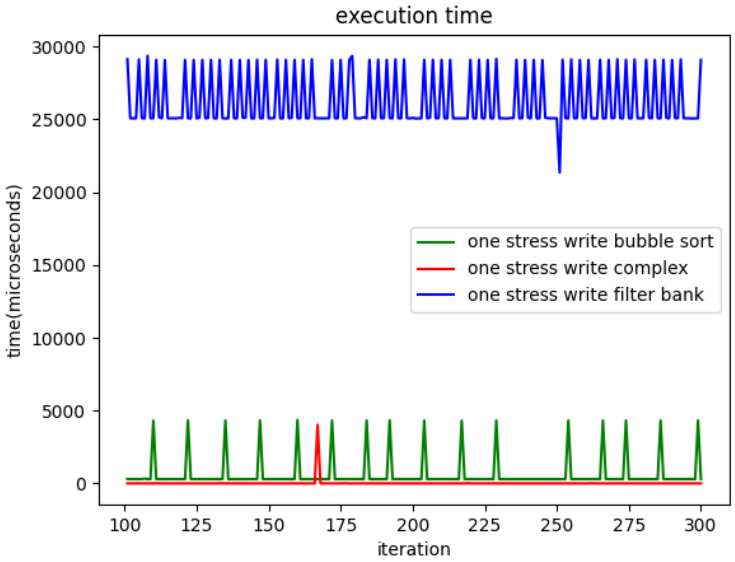


Figure 4.15: Execution time of the three categories under one stress of write while operating on Embedded Linux.

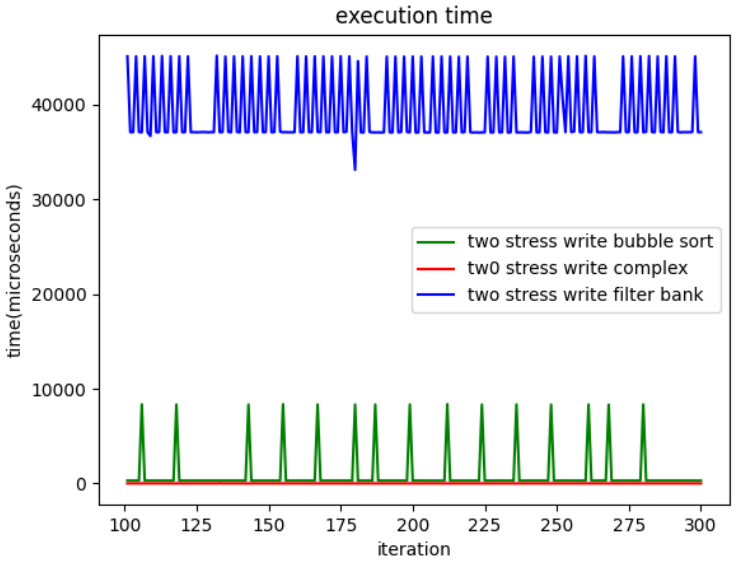


Figure 4.16: Execution time of the three categories under two stress of write while operating on Embedded Linux.

5 Discussion

5.1 Results

After running the benchmarks and analyzing the execution time of different stresses, we saw that while there was no stress running on the board the performance of the system was very stable with very small to no variance. All the threads were running on maximum priority with no interruption and that could be reflected on the smooth run time. The performance of the system while running the two different operating systems of Linux and RT-Linux were nearly identical while Linux was a bit faster in all scenarios, but this speedup was near to not being noticeable. When a stress was in effect on the side cores of the system while we could guarantee that there was no race condition or interference on data access, it could be observed that normal execution time was still the same but on some occasions, there were spikes that introduced a great deal of delay in the execution time on a few iterations. It should be emphasized that the threads that were running on the system were the same and no side threads or any means to interrupt were added to the system.

The spikes were more common on the benchmarks that were more memory dependent such as bubble sort or the benchmark to calculate standard deviation; we categorized these benchmarks in category two. The benchmarks that were not memory-dependent were also experiencing very few spikes which was uncommon. This meant that even if a program is not memory dependent and is coded efficiently, it might experience runtime delay due to inefficiency or stress on memory. It was observed that if the benchmark is heavily dependent on memory accessing different memory locations and the memory gets used frequently, the stress on the memory would hinder the performance of the benchmark drastically rather than only a few iterations experiencing spikes, which is pointed out by our observations from category three. When the system undergone more stress, the number of spikes in most occasions remained the same, some increased, and only a few decreased. The pattern of spikes and how often they would happen was not consistent or predictable. What was constant was that the impact of doubling the stress resulted in impact of spikes being higher to being nearly double as well.

In the next step, we compared the performance of RT-Linux and Embedded Linux, it could be seen that the number of spikes of Embedded Linux was less than RT-Linux but the impact of the spikes was more on the execution time. Since the system was in exact same scenario

and only the operating systems were different we could say that this change of pattern was due to a change of decision-making in scheduling that is implemented in RT-Linux.

It should be pointed out that we also compared the performance of the system while changing the stress from reading to writing and we saw that there was no change to the behavior of the system when the stress type changed.

5.2 Method

The purpose of the experiments done in this master thesis work was to see how normal programming patterns would perform in a scenario where multiple independent programs run on a multicore embedded system. Unfortunately, we could not find a relation between Linux priorities and the priority of Pthreads of C. Even though the highest priority was given to both the benchmark application and threads of the benchmark, we could not guarantee that the threads could win the competition of obtaining priority compared to other system threads of Linux when they operate on a different priority basis. We could guarantee that there were no side applications, and no other application was running other than our benchmark application. Yet the operating system itself also competes with applications to obtain CPU cores to run.

Due to lack of time, a few of the benchmarks of the Taclebench had to be skipped on for the reason that porting them would take too much time, however, since we could observe that most of the benchmarks follow the same pattern, we could deduce that porting more benchmark would not result in different conclusions.

Compared to the related work done in the field, we saw that it was very hard to make the system operational and install programs on it. This made us unable to use benchmarks like Cyclictest and RT-Bench on the board. Nevertheless, the related work gave direction to the implementation of our benchmarks. The related work did a better job of delving deep into the architecture of the embedded system while we took the system as a black box and focused more on running the benchmark applications.

5.3 The work in a Wider Context

Testing the benchmarks on VxWorks was in our plan, it would be very good to test it on other operating systems. The rest of the Taclebench could be imported though we doubt anything new would be obtained. More benchmarks could be done by running the benchmarks on other cores instead of stress running on them. Monitoring measures could also be implemented to understand the behavior of hardware while the benchmark is running; this would allow a more in-depth insight into why the spikes would happen while the benchmark is running.

Furthermore, we suspect that the issue sighted in the research here could also be seen in normal computers even though to a lesser extent.

6 Conclusion and Future Work

6.1 Conclusion

This thesis work was purposed to see how embedded systems perform real-time tasks on a multicore platform while the execution is multi-threaded and the tasks operating on the multiple cores are independent. We did not want to make the tests complicated and took the approach of making sure that all the threads were independent of one another. The results that we obtained were very fascinating. Having the advantage of running parallel programs or using multiple cores of the embedded system comes at a risk as well. The risk that we discovered was that even running tasks on multiple cores might hinder the performance of our crucial tasks even if the tasks are independent. We expect that this observation happened due to the sophisticated architecture of embedded systems.

Having the opportunity to run multiple tasks is a great advantage, yet we observed that the best way to utilize the advantage is to program applications that are designed to run in parallel or be very mindful of the multicore architecture of the system. The reason is that idle cores or cores that are not utilized properly might hinder the performance even if not intended to be running other tasks due to the scheduler of the operating system aiming to maximize throughput.

While doing the work, we observed that installing operating systems and making embedded systems functional due to their less common use compared to standard computers and customized hardware is way harder than normal. That also made our capability in benchmarking very limited.

We also took the opportunity to compare Real-Time Linux and standard Linux. We could observe that Real-Time Linux is more adapted to parallelism and interruption between the tasks can be recovered more smoothly. However, for the same reason, interruptions might happen more often than standard Embedded Linux, and the RT patch of Linux might not be a good solution for vital real-time applications if the multicore nature of the embedded system gets ignored.

6.2 Recall of the Research Questions

After analyzing the results and experiments and taking notes on the tasks completed benchmarking the embedded system under test, it is good to go back to the research questions we asked at the start of the thesis work and answer them.

1. The embedded systems have specific functionality and are usually designed for specific purposes. What are the challenges to make an embedded system operational for performing real-time applications?

   Embedded systems usually have their own specific architecture and making them operate requires a custom operating system that is built for the architecture and hardware of the system, this makes every embedded system unique, and treating them is not as standardized as commonly used computers. Furthermore, making an embedded system capable of running various applications, especially real-time applications requires the installation of additional libraries and software which would make operating on them more challenging. In our case, making a custom version of Linux operating for our embedded system was very hard and we tried to install a Python interpreter environment and VxWorks operating system on the board. Unfortunately, due to the complexity, we could not achieve our goal of installing Python on our operating system or a custom VxWorks for the board.

2. What could be a good measure to benchmark the real-time performance of an embedded system while it has multiple applications running on different cores?

   While there are many metrics to measure the real-time performance of a system, in the criteria of multicore operation we saw that measuring the execution time of running the real-time task while the system is not running processes on multiple cores and in another scenario utilizing all cores would be a good measurement and comparison reference. Our goal was to discover how the behavior of the system would change if the same system would transit from running on a single core to multicore without having any conflicts or race conditions. Other metrics such as jitter calculation and response time analysis can also be used which relative works have touched upon them.

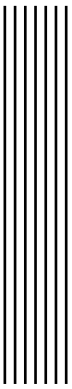
3. The RT-Preempt Linux patch aims to enhance the real-time capability of the Linux operating system. How good is the performance of RT-Preempt Linux compared to Embedded Linux in the criteria of running multicore programs?

   According to our analysis, we observed that in the criteria of one core running real-time task and other cores being occupied with their own independent tasks the RT-Preempt Linux might experience more spikes due to congestion of memory unit yet it can also recover from them more easily. On the other hand, the Embedded Linux would run more smoothly with less possibility of spikes. However if the spikes would happen the Embedded Linux would have a harder time recovering from them.

6.3 Future Work

In this thesis work, we discovered the change of behavior of an embedded system when the system operates on multiple cores instead of a single core. We saw how congestion of memory would affect the execution time of a real-time application even though it is not getting interrupted. However, we could not get deep into what exactly would happen on the memory side and what would cause such hindrance. Further on, if the discovered problem belongs to the operating system and decision making of the operating system or the problem is more fundamental and it's due to the architecture. It was originally planned to test the custom benchmark implemented on the VxWorks operating system as well to see if the VxWorks

operating system would perform with the same behavior as the two operating systems that we tested.



Bibliography

[1] AMD. *Peta-Linux*. 2023. URL: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html (visited on 12/05/2023).

[2] *AMD ZCU*. 2023. URL: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html#specifications (visited on 12/05/2023).

[3] Antonio Barbalace, Adriano Luchetta, Gabriele Manduchi, Michele Moro, Anton Soppelsa, and Cesare Taliercio. "Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application". In: *IEEE Transactions on Nuclear Science* 55.1 (2008), pp. 435–439.

[4] Antonio Barbalace, Adriano Luchetta, Gabriele Manduchi, Michele Moro, Anton Soppelsa, and Cesare Taliercio. "Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application". In: *IEEE Transactions on Nuclear Science* 55.1 (2008), pp. 435–439.

[5] Jeremy H Brown and Brad Martin. "How fast is fast enough? Choosing between Xenomai and Linux for real-time applications". In: *proc. of the 12th Real-Time Linux Workshop (RTLWS'12)*. 2010, pp. 1–17.

[6] Felipe Cerqueira and Björn B. Brandenburg. "A Comparison of Scheduling Latency in Linux, PREEMPT-RT, and LITMUS RT". In: 2013. URL: https://api.semanticscholar.org/CorpusID:14096981.

[7] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. "TACLeBench: A benchmark collection to support worst-case execution time research". In: *16th International Workshop on Worst-Case Execution Time Analysis*. 2016.

[8] Linux Foundation. *preempt RT-Linux*. 2016. URL: https://wiki.linuxfoundation.org/realtime/start (visited on 12/05/2023).

[9] Linux foundation. *yoctoproject*. 2023. URL: https://www.yoctoproject.org/about/project-overview/ (visited on 12/05/2023).

[10] Thomas Gleixner. *cylictest*. 2023. URL: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start (visited on 12/05/2023).

[11] Litmus. *The LITMUS RT project*. 1999. URL: http://www.litmus-rt.org. (visited on 12/05/2023).

[12] Mastura D Marieska, Paul G Hariyanto, M Firda Fauzan, Achmad Imam Kistijantoro, and Afwarman Manaf. "On performance of kernel based and embedded real-time operating system: Benchmarking and analysis". In: *2011 International Conference on Advanced Computer Science and Information Systems*. IEEE. 2011, pp. 401–406.

[13] Mattia Nicolella, Shahin Roozkhosh, Denis Hoornaert, Andrea Bastoni, and Renato Mancuso. "Rt-bench: An extensible benchmark framework for the analysis and management of real-time applications". In: *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. 2022, pp. 184–195.

[14] Microsoft Windriver. *VxWorks*. 1987. URL: https://www.windriver.com/resource/vxworks-product-overview#:~:text=VxWorks%C2%AE%20is%20the%20industry's,a%20modern%20approach%20to%20development. (visited on 12/05/2023).

[15] *Xenomai*. 2010. URL: https://xenomai.org/ (visited on 12/05/2023).