Exploring Thread Mapping Strategies for RTEMS SMP Based Real-Time Systems

Masterarbeit

Vorgelegt von:
Ebenezer Samson
Matr.Nr.: 0389984
Studiengang: Luft- und Raumfahrt

Fakultät Verkehrs- und Maschinensysteme
Fachgebiet Raumfahrttechnik
Institut für Luft- und Raumfahrt
Marchstraße 12-14
10587 Berlin

Betreuer:
Prof. Dr.-Ing. Klaus Brieß
Mario Starke, M.Sc

05.01.2020

Declaration of Authorship

I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgement in this thesis. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged.

Place and date

Signature

Agreement on rights of utilization

The Technische Universität Berlin, represented by the Chair of Space Technology, may use the results of the thesis at hand in education and research. It receives simple (non-exclusive) rights of utilization as according to § 31 Abs. 2 Urheberrechtsgesetzt (Urhg). This right of utilization is unlimited and involves content of any kind (e.g. documentation, presentations, animations, photos, videos, equipment, parts, procedures, designs, drawings, software including source code and similar).
An eventual commercial use on part of the Technische Universität Berlin will only be carried out with approval of the author of the thesis at hand under appropriate share of earnings.

Place and date

Signature

Professor Dr.-Ing. Klaus Brieß
Head of the Chair of Space Technology

Acknowledgement

First and foremost, I would like to thank God Almighty for his grace in enabling me to complete this thesis with success.

I would like to extend my gratitude to Prof. Dr. Ing. Klaus Brieß, Dipl. Ing. Cem Avsar and the MSE team for the invaluable support and for setting up an excellent platform to achieve my career goals.

I would like to pay my special regards to Daniel Lüdtke and the OSS (On-board software for Space Systems) group at DLR (Deutsches Zentrum für Luft- und Raumfahrt), Braunschweig for providing a great working environment.

I wish to express my deepest gratitude to Jan Sommer and Zain Hammadeh for their expert guidance and support which has been invaluable in steering me throughout my thesis.

I also thank Mario Starke for his assistance in providing formal supervision and constant feedback at TU Berlin.

I wish to acknowledge the support and great love of my family. They kept me going on and this work would not have been possible without their input.

Abstract

In recent years, multi-core processor configurations have become common in general-purpose computing. Multi-core systems offer exciting opportunities such as power-efficient processing and parallel execution. But they are not the first choice when it comes to real-time computing systems. Difficulties in developing a predictable system is one of the serious challenges that prevent them from being used for real-time applications. In spite of their complexity, multi-core processors are inevitable in future real-time systems, especially the ones used in space applications. This is due to the rising demand for high processing power and power-efficient hardware in space systems. Space research organisations around the world are taking efforts to overcome the challenges posed by multi-core real-time systems. DLR is currently interested in addressing the real-time scheduling problem in multi-core systems. Because of the scope for RTEMS as a real-time operating system for multi-core systems, DLR is particularly interested in analysing the RTEMS schedulers.

Real-time task scheduling is a well-studied problem on uni-core processors for which there are mathematically proven algorithms that can guarantee schedulability of task sets. These algorithms are not applicable to multi-core systems due to the added spatial dimension(number of cores) over the already existing uni-core scheduling problem. Partitioned scheduling is one of the multi-core scheduling approaches where the the threads are statically assigned to the available cores and then each core is treated as a uni-core scheduling problem. The strategy employed in assigning the threads to the cores is an active area of research. Recent studies have highlighted the huge potential for partitioned scheduling. Most of the analysis in this field are simulation studies based on an ideal mathematical model of the scheduler algorithm. The outcome of these studies are applicable only for the ideal scenario. But, due to the implementation overhead, a practical real-time scheduler does not behave similar to the ideal model. Moreover, the hardware dependability plays a huge part in this aberration. It is difficult to model these behaviours in simulation based studies. Therefore, an experimental analysis of the RTEMS scheduler is presented in this thesis.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

As the state-of-the-art real-time systems are evolving in complexity and scale, the demand for high performance processors will continue to increase[SJPL08]. Given the power-wall limitation in uni-core processors, the chip manufacturers have embraced multi-core technologies as a way to continue the performance improvements in their products. Multi-core processors are common in many areas, but currently not within the space industry. The reason being the added complexity in analysing software behaviour has so far not outweighed the benefit of higher processing power[CHS+14]. This trend, however, is changing due to the rising demand for on-board computational power for the future space missions. Multi-core technology is becoming an attractive solution to fulfil the increasing on-board processor demands. While offering more computational power, multi-core systems consumes significantly lesser power and have a smaller thermal footprint than a uni-core processor.

In space applications, multi-core processors are needed:

- To fulfill the high reliability and throughput requirements needed for autonomous spacecrafts.
- To achieve high fault tolerance with redundant processing on human-rated spacecrafts.
- For on-board processing of high resolution imaging payloads without violating the power and thermal requirements.
- Where the ground station processing is not feasible due to the data transfer rate limitations between the spacecraft and ground, especially in deep-space missions.
- For on-board processing of high data rate instruments[kel19].

Given the need for multi-core processors in the space industry, the European Space Agency(ESA) and the National Aeronautics and Space Administration(NASA) are taking active efforts to make the multi-core systems reliable for use in space applications. ESA is actively involved in developing the Next Generation Multiprocessor(NGMP) for their future missions named LEON4. LEON4 is a quad-core SMP System-on-Chip(SoC) based on SPARC V8 architecture[SHG]. SMP is a widely used multiprocessor architecture, where all the processing cores are homogeneous and are connected to a common memory space. SMP architecture makes it possible to run a single operating system(OS) instance on the system. ESA is also looking for a reliable Real-Time Operating System(RTOS) support for LEON4. Lack of readily available and easily analysable real-time OS support for SMP configurations is another drawback for using multi-core processors [CHS+14]. RTEMS is an open source real-time operating system which is used

in many space applications[Joe19]. RTEMS supports SMP system and ESA has been recently involved in the qualification of RTEMS for LEON4[leo].

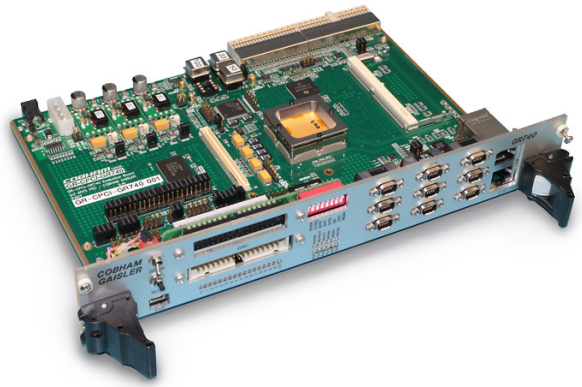


Figure 1.1: LEON4 Development Board[l4d]

Predictability is one of the essential parameters in safety-critical systems. Multi-core systems poses new challenges especially in RTOS environment due to the unpredictability involved. The core component that affects predictability is the scheduler of the RTOS. Real-time task scheduling on multi-core processors is an active field of research. One of the multi-core scheduling problem is the strategy in assigning the tasks or threads to the given cores. There is no optimal solution for this problem and mathematically it is found to be an NP-hard problem[Ull75]. Most of the research work being carried out in this field are based on mathematical scheduling algorithms. The advantage of this approach is a simplified analysis[DS14] and the outcome can be applicable to any scenario in an ideal world. But in the practical world, the mathematical algorithms do not behave exactly as expected. This effect is due to the unavoidable software overheads incurred while implementing them. If these overheads are considered in an analysis, the outcome of the study are not transferable and they are applicable only for the hardware and software used for the study. In other words, each and every hardware and software combination used for the study yields unique results. In this thesis, the real-time multi-core scheduling problem is studied on an SMP hardware using RTEMS.

1.2 Objective

To understand how the RTEMS scheduler performs on an SMP system in handling a test task set, firstly, an advanced task model that can generate a sample task set with precedence constraints is needed. Then the sample task set is scheduled using the global approach. Then some thread mapping strategies shall be used to study the partitioned scheduling approach. The objective for this thesis are summarised below:

1. Generation of a sample task set.
    a) Implementation of a task model that can generate task sets with precedence constraints.

   b) Schedulabilty of the the sample task set under the resulting six different scheduling schemes.

2. Selection of the Thread-to-Core mapping strategies
   a) This is essential to implement partitioned scheduling
   b) Reviewing of the partitioning schemes from recent studies and selection of the suitable schemes for this study.
   c) Selection and implementation of at least two mapping strategies.
3. Analyse the RTEMS global scheduling algorithms.
   a) Analyse the real-time performance of a sample task set under various Central Processing Unit(CPU) loads using global scheduling.
   b) Carrying out the analysis without modifying the kernel by using only the functions that are available for the application developers.
4. Analyse the RTEMS partitioned scheduling algorithms.
   a) Analyse the real-time performance of a sample task set under various CPU loads using partitioned scheduling.
   b) Carrying out the analysis without modifying the kernel by using only the functions that are available for the application developers.

1.3 Organization

This thesis is organised in six chapters.

Chapter 2 begins with the essential background knowledge needed for proceeding with the study. It gives an introduction to the basics of computing systems, real-time systems and real-time task models. It also gives the definitions and terminologies required to establish a common notation. The chapter concludes with an overview of the RTEMS kernel and its features.

Chapter 3 elaborates the scheduling problem in real-time systems, the uni-core scheduling algorithms and their mathematical proofs and the increased complexity in scheduling on a multi-core system and the global and partitioned scheduling approaches used in multi-core scheduling.

Chapter 4 describes the approach and explains the methodology used in carrying out this study along with the various approaches considered, the selection of suitable instruments and tools.

Chapter 5 describes the implementation part of this study. The RTEMS features that were used in implementing the sample test cases, the chosen test scenarios, the experimental setup, post-processing of the experiment data and the presentation and discussion of the results.

Chapter 6 summarizes the outcome of this thesis, the results and its significance, the problems that occurred during the time period of this study and finally gives an outlook about the future scope for continuing this work.

2 Background

2.1 Introduction to real-time systems

The phrase "Real-time Systems" represents a class of computing systems that must react within precise time constraints to events in the environment. These systems are suitable for applications where not only the correctness of the computation but also the time at which results are produced is equally important[But11]. Real-time does not mean an instantaneous response from the system. But the system guarantees a time range within which the response can be expected. This is the main differentiating factor when compared to a general-purpose computing systems. A general-purpose computing system does not provide a guarantee on the response time. Guaranteed response time is essential for many time critical applications such as the control systems used in autonomous spacecrafts. Real-time response is essential for the stability of such systems and any compromise on the response time will end in a catastrophe.

2.1.1 Embedded systems

Embedded systems are computing devices which are characterised by low power consumption and small size factor. These devices are mostly preferred to be used for real-time applications. The architecture of a modern embedded system does not differ much from a general-purpose system. One of the earliest modern embedded system used for space application is the Apollo Guidance Computer(AGC). AGC was used for the guidance, navigation and control of the Apollo spacecraft that took astronauts to the moon in the 1960s. Apart from the redundant design, the hardware architecture of the AGC was similar to any other general-purpose computing system at that time. Embedded systems are usually developed for a single purpose and they are optimized for reliability and performance[Con19]. The optimization can be usually seen in the form of reduced memory and a simpler Instruction Set Architecture(ISA). The usage of simpler ISA leads to a smaller physical size, low power consumption and superior thermal characteristics. Hardware components like Memory Management Unit(MMU), that can lead to non-deterministic behaviour, are usually eliminated as well. An example of a modern embedded hardware is the SoC design based on the popular ARM Cortex-A7 MPcore. Broadcom BCM 2836 used in Raspberry Pi 2B is one of such SoC designs, that can run GPOS like Linux and suitable for RTOS as well.

2.1.2 Embedded software architecture

Unlike hardware architecture, software for real-time systems vary significantly from the software used in general-purpose systems. In order to satisfy the timing requirements, real-time software

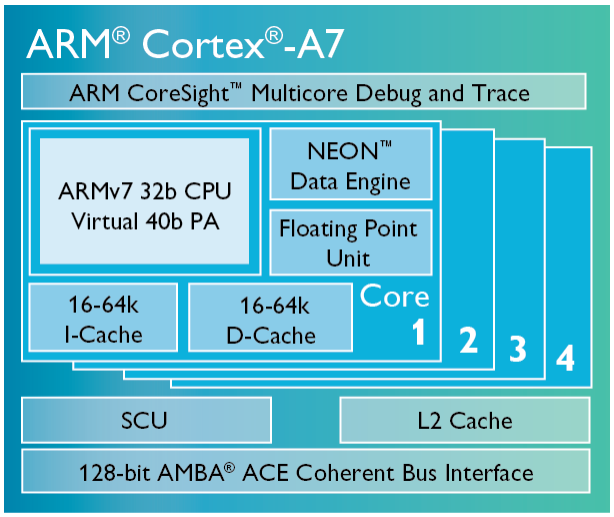


Figure 2.1: SMP Architecture[ca7]

should behave in a predictable manner. Development of a single self-sufficient program will result in a predictable and deterministic system. But this approach is only viable for simple real-time applications. As applications become more and more complex it becomes harder and time-consuming to develop using this approach. As a result, modern real-time applications are developed in a similar way as a general-purpose application is developed. Applications are developed as modules that can be integrated on an operating system(OS). The applications use the services provided by the OS and these services are developed and made available to the application developers by the Kernel developers. Module-based approach using OS offers various advantages like multi-threading, re-usability and portability. Together they contribute to less error-prone codes and faster development times. However, the operating system used for real-time applications has less overhead and predictable response times when compared with a general-purpose OS.

In the context of real-time, the term operating system is often mixed together with the term kernel. Usually kernel is the part of an OS that provides basic functionalities for the user application to execute on the hardware. Basic functionalities include services for thread management, time management, inter-thread communication and synchronization, memory management and I/O management. In many cases RTOS provide these basic facilities only, so they can also be referred as a real-time kernel. Since an application developer relies on the services provided by an OS to build the application, the developer has only a little or no control over lower-level functionalities that make up a service. These functionalities may affect the real-time performance of the application. The responsibility lies with the kernel developers to ensure that the implementation of low-level functionalities does not affect the performance of the system. There are open portability standards like POSIX which makes the application portable across different OSs. Each and every kernel is developed with different approaches so the application performance varies across different OSs.

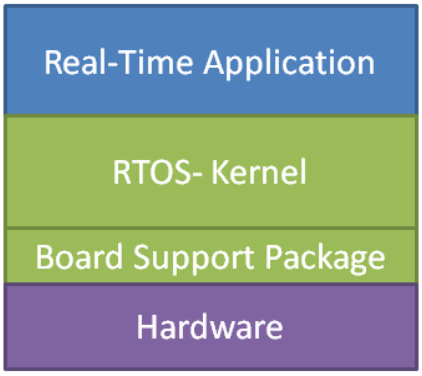


Figure 2.2: Software Architecture

2.1.3 Multi-threading

A Thread is the basic processing entity that is managed by an OS. It is essentially an abstraction offered by the operating system for programmers to have multiple execution paths(i.e. multiple threads) within the same program. Multi-threading is one of the important services needed in any modern real-time application and it is inevitable for multi-core systems. A single real-time application can be decomposed into a collection of threads that can be executed concurrently. The OS manages all the threads in the system using a scheduler. The scheduler allows the CPU time to be shared among the threads so that there is a concurrent progress in the system.

In uni-core processors, the scheduling algorithm determines which task gets assigned to the CPU at any given time. In real-time systems, each thread is assigned with a priority based on their importance. This allows for the thread with the highest priority to always get the processor. This property is essential for meeting the timing requirements of a system. In multi-core processors, with the availability of extra processing cores, it is possible for threads to be executed in parallel which eventually increases the performance of multi-threaded applications. Multi-core schedulers have an additional responsibility of the spatial allocation of threads to different cores, in addition to the temporal allocation on each core. The response time of the system depends on the efficiency of the scheduler and the scheduling policy. The scheduling algorithm has to make better use of the available resources while trying to meet the timing requirements of the given task set. Chapter 3 explains more in detail about the challenges involved in scheduling algorithms to fulfill both of these requirements.

Given a system with multiple threads, the schedulers in GPOS focus on achieving maximum throughput by focusing on fairness among the threads. While the RTOS scheduler does not focus on fairness among threads, it treats every individual thread differently and tries to fulfill the timing constraints. This calls for defining threads differently in RTOS. A thread in RTOS is usually characterised by parameters like priority and deadlines. To understand the essential terminologies needed are explained in the following section.

In addition to providing APIs for thread creation and management, RTOS must also provide other essential features to support multi-threading. In order to perform useful computation

with multiple threads, threads must communicate with each other. This calls for inter-thread communication primitives like message queues and data buffers. A real world problem may not always be parallel in nature, so the threads must work in a synchronized fashion to achieve the final output. Other important features are synchronization primitives like signals, events and semaphores for the application developers to achieve this synchronization.

2.1.4 Classification of real-time tasks

Figure 2.3: Real-Time system

It is usual for threads in real-time systems to exist forever but they are not always available to be executed on the CPU. Let us consider a typical thread in a Proportional-Integral-Derivative(PID) controller system. The PID controller uses the incoming sensor data and computes the results to be used for controlling the actuators. The thread responsible for the control law computation will not be activated unless a new sensor data is available. If the sensors have a fixed sampling rate of 1Hz then the computation thread gets activated(or ready for execution) at the frequency of 1Hz. Then the computation thread is known as a Periodic thread with a Period(P) of 1 second. Rate of a periodic thread is 1/P. The activation time of a periodic thread can be predicted by knowing its Period. If C is the execution time of the task then the Utilization(U) for a periodic task is given by,

$$U = C/P \tag{2.1}$$

It is possible to develop a deterministic and predictable system if all the threads in the system are periodic. But, usually, in any practical system, not all the threads are periodic. In case of the same PID example, if the sensors don't have a fixed sampling rate then the activation time of the computation threads will not be periodic. In this case, the computation thread can either be an aperiodic thread or a sporadic thread. If a non-periodic thread has a soft deadline, then it is known as an aperiodic thread. If it has a hard deadline then it is known as a sporadic thread. Activation times of non-periodic threads are unpredictable and the periodic formulations are no longer valid. Some assumptions are made to study and analyse a system with non-periodic threads. For aperiodic tasks, an upper time bound on activation time is assumed. For sporadic tasks, a lower time limit on the inter-arrival time is assumed.

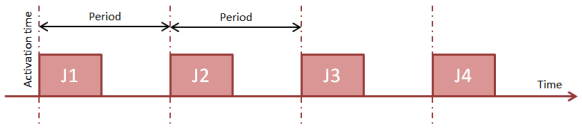


Figure 2.4: Periodic Task

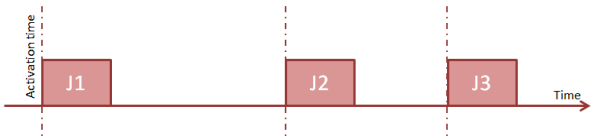


Figure 2.5: Non-Periodic Task

2.2 Terminologies of real-time tasks

A real-time task can be characterised by the following parameters:

Activation time(Release time): It is the point of time when a task(or a thread) becomes "Ready" for execution. The task can only be scheduled to the processor at or after the Activation time.

Release jitter: Due to the practical overhead, activation may not occur at the expected instance in time. But, it usually occurs within an upper or lower bound time interval known as Release jitter.

Start time: The point of time when the task starts its execution.

Finish time: The point of time when the execution of the task is completed.

Execution time: It is the time taken by the given task to complete the its execution without preemption.

Response time: Time elapsed between the release time until the task completes its execution. Execution time and response time may not be same always, since a thread might not be executing continually. By carefully imposing timing constraints on each and every thread in the system, the overall response time of the system can be improved.

Absolute Deadline(Deadline): It is the point of time by which the task has to complete its execution. It is measured in the absolute time scale.

Relative Deadline: It is the allowable response time of a given task. It is measured in relation to the activation time of the task.

Lateness: The finish time of a particular task with respect to its relative deadline. Late-

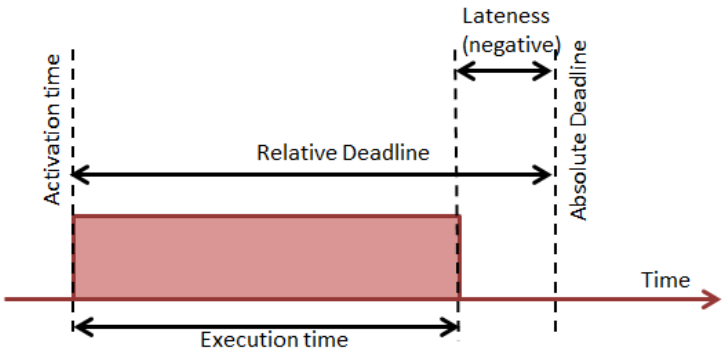


Figure 2.6: Real-time Task Terminologies (a)

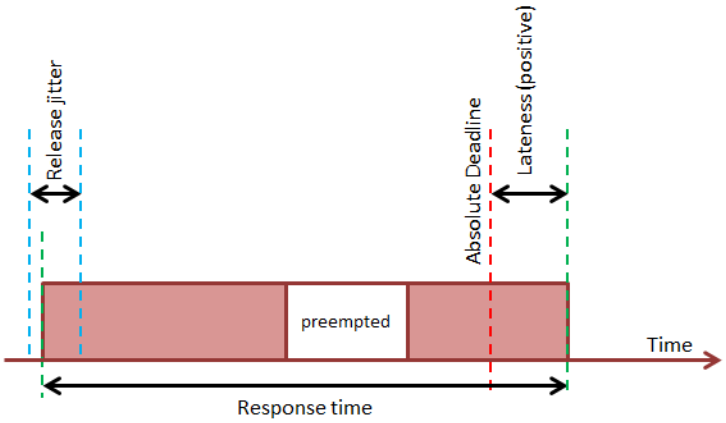


Figure 2.7: Real-time Task Terminologies (b)

ness is negative when the execution of a task is completed before its relative deadline and positive if it completes after the relative deadline.

Laxity(Slack time): The maximum time that a task can be delayed, after it has been activated, so that the task completes its execution before the deadline.

Tardiness(Exceeding time): The maximum duration that a task will remain in the "Ready" state after missing the deadline.

2.3 Types of task constraints

Real-time tasks work under three major classes of constraints[But11]

2.3.1 Timing constraints

All the real-time threads work under user-imposed timing constraints known as deadlines. The goal of the real-time scheduler is then to fulfill the timing constraints of each and every task in the system. Criticality of a deadline denotes the severity of the consequences that would arises in case of a deadline miss. Based on criticality, a deadline can be classified either as Hard or Soft deadline. Missing a hard deadline will result in a catastrophic failure of the system. In case of missing a soft deadline, the likelihood of a catastrophic failure of the system is less. Many real-time applications only work under soft deadlines.

2.3.2 Precedence constraints

In addition to the timing constraints, there are other constraints that affect the response time of a task. Many of the real world problems are not parallel in nature. This limitation creates inter-dependencies between different tasks and hence some tasks have to wait for other tasks to complete their execution before proceeding with its own execution. This type of constraint imposed on a task either by design or by the user is known as precedence constraints.

2.3.3 Resource constraints

Hardware capacity of a given system poses a limitation on the maximum tasks that can be executed at a particular instance in time. When the number of threads in the system is more than the available processors, some of the threads wait in an idle state until CPU allocation. This adds to the response time of the thread. It is the responsibility of the scheduler to effectively allocate the threads to the available hardware resources for execution. Shared memory system architectures, like SMP systems, encounter another type of resource constraint which occurs when two or more tasks need to access the same address space. In this situation, the synchronization methodology imposed on tasks by the user prevents simultaneous memory access. So the given task has to wait for the other tasks to release the shared memory before proceeding with its execution. This eventually adds up to the response time of the task.

2.4 RTEMS

Real-Time Executive for Multi-processor System(RTEMS) is an open source RTOS which is being used in many safety critical systems especially in the European space industry. RTEMS has a good space heritage. It is a part of the Electra software radio on NASA's Mars Reconnaissance Orbiter, and the ESA's Trace Gas Orbiter.

RTEMS kernel supports several features. The most important features are multi-threading, networking and file systems. It supports many host platforms and target architectures including ARM, PowerPC, Intel, Blackfin, MIPS, SPARC, RISC-V and others. Applications can be developed in C, C++ using the RTEMS Classic API or the open standard POSIX API.[SSF+09].

RTEMS provides a simple build system to build the cross tool chain for the desired target architecture which can then be used to build the kernel and user applications.

RTEMS provides a multi-threaded environment in a single address space. In POSIX terms, it is considered as a single process with multiple threads. RTEMS tries its best to hide its internal structures to the user. While exposing very little detail on its inner structure, the services are made available to the applications through various resource managers. Each resource manager is a logical set of related components to provide a particular service to the application.

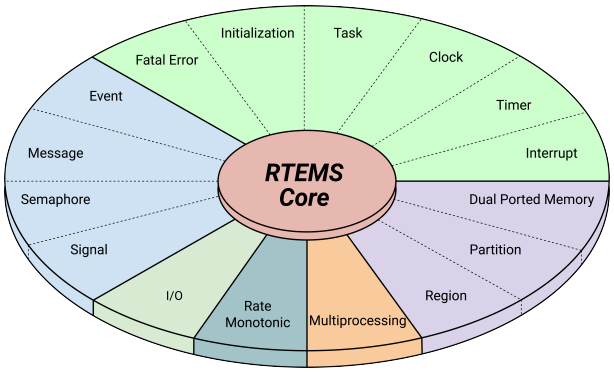


Figure 2.8: RTEMS Organization[rte]

The official RTEMS release version is 4.11 with an experimental SMP support. SMP support was improved in RTEMS 5 but it is still under development phase and there has been no official release. The development branch of RTEMS provides SMP support for ARMv7-A, PowerPC, RISC-V and SPARC architectures. In 2017 ESA offered a contract to improve the SMP support in RTEMS for NGMP(LEON4) and for the qualification of RTEMS SMP for space applications[Ver].

3 The Scheduling Problem

3.1 Scheduling of tasks

Task scheduling is the method of assigning processor to the tasks. At any point in time, each and every task in the system must be in either one of the following states.

Running: When the thread is actually being executed on the hardware.

Ready: When the thread has met all the conditions for execution and it has all the necessary data and instructions required for execution and it is waiting on the scheduler for the processor allocation.

Blocked: When the thread does not meet the requirements for execution or it does not contain the necessary data or instructions needed for the execution and it is waiting either to meet the required execution condition or to get the data or instruction from memory.

Terminated: When the thread has completed execution and it is out of the execution cycle.

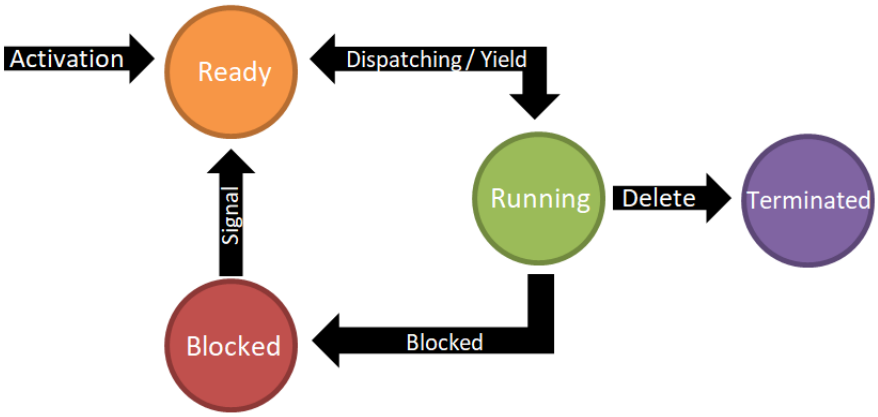


Figure 3.1: Task States

A **scheduler** can allocate the tasks to the processor only when they are in the Ready state. Some schedulers are capable of interrupting a Running thread and deallocate them from the hardware so that it can allocate the processor to another task. This process is known as preemption and the scheduler is known as the preemptive scheduler. Preemption is one of the essential feature in real-time systems. A Preemptive scheduler always ensures that the highest priority thread at any given instant gets the CPU. Some schedulers cannot interrupt a running

task by design and they are known as cooperative schedulers. In this case the scheduler waits until the task voluntarily give up the CPU using the yield system call and moves to the ready state. The task then waits for the scheduler to allocate the CPU again.

Time Triggered schedulers are activated on hardware timer interrupts. When the scheduler thread is executed, its first task is to determine whether task switching at that time instant is required or not. Task switching is usually required when a task with a higher priority task is waiting in the ready state or if the task that is being executed wants to voluntarily give up the processor. If task switching is required, then its second job is to select the next task from the list of all the "Ready" tasks at that instant. Given a list of ready tasks, the selection of the next task is based on the scheduling policy of the scheduler.

One of the important step that occurs when a new thread is about to be assigned to the processor is **context switching**. Each thread has a private memory context on which they operate. Context switching is the process of storing the context of the current thread and retrieving and loading the state of the next thread which is to be scheduled next. Dispatcher is a sub-module of the scheduler which is usually responsible for context switching i.e. for saving and restoring thread context. The time taken for deallocating a task and reallocating another task is known as **dispatch latency**. For real-time systems, dispatch latency has to be deterministic or bounded.

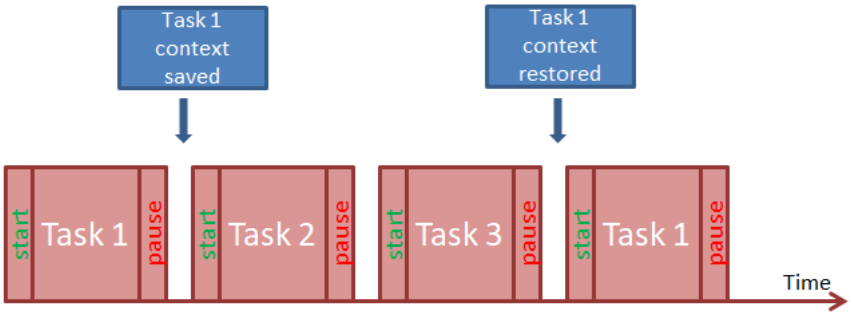


Figure 3.2: Context Switching

3.1.1 Scheduling policies

Scheduling policies for real-time systems can be either Priority driven policy or Time sharing policy. Since this thesis only focuses on schedulers with priority driven policies, time sharing policies are not discussed. The main idea behind priority driven policies is to avoid fair treatment of all tasks in the system. Tasks in the system are classified into different levels based on their importance. Tasks in the higher level of importance are assigned with a higher priority and vise versa. Priorities can be assigned to tasks either statically or dynamically and the scheduling policies are classified based on how the priorities are assigned. In this thesis we will be looking at three priority driven scheduling policies that are supported by RTEMS. These scheduling polices are mathematically proven to be optimal for uni-core processors under certain conditions.

1. **Static Priority:** Priorities of the tasks are defined at compile time.
    a) *Rate Monotonic(RM):* For a periodic task, the priorities are assigned based on the inverse of their period, P. This policy ensures that higher priority is given to tasks with shorter period and lower priority is given to tasks with longer period.
    b) *Deadline Monotonic(DM):* Tasks are assigned priorities according to their deadlines. This policy ensures that the tasks with shorter relative deadline are given higher priority.
2. **Dynamic Priority:** Priorities are computed and assigned in run-time. Priority of the task may vary with respect to the other tasks available at that instance.
    a) *Earliest Deadline First(EDF):* At every scheduling event, the task which is closest to the deadline is given the highest priority. EDF scheduling policy has a utilization bound of 1.

3.2 Important terminologies in scheduling

Schedule: A schedule is an assignment of tasks to the processor so that each task is executed until completion [But11]

Schedulability: A task set(T) is said to be schedulable with an algorithm(A) if A generates a feasible schedule.

Feasible schedule: A feasible schedule is the one in which all the tasks in the task set(T) meets their deadlines.

Feasible task set: A task set(T) is said to be feasible if there exists an algorithm that generates a feasible schedule for T.

Optimal scheduler: An Optimal scheduler fails to meet a deadline only if no other algorithm of the same type will also fail. In other words, if a task set(T) is not schedulable in the optimal scheduler then it is not schedulable under any other algorithm.

Heuristic scheduler: Heuristic scheduler generates a schedule according to a heuristic function that tries to satisfy an optimal criterion, but there is no guarantee of success.

Utilization bound: Utilization bound of a scheduling algorithm is the limit below which all task sets meet their deadline.

3.3 Real-time scheduling on multi-core systems

In uni-core processors, tasks are scheduled one after the other and context-switched frequently. The scheduler provides the temporal resolution i.e. it decides which thread gets allocated to the hardware at each time instant and if needed, when to interrupt and reallocate. The scheduler ensures thread-level concurrency in the system. The problem of finding a feasible schedule in a uni-core processor is an NP-hard problem[Ull75]. In practice, it means that the time for finding a feasible schedule grows exponentially with the number of tasks in the system. The scheduling problem in uni-core processors have been studied for over 40 years. As a result, there are mathematically proven optimal scheduling algorithms available. The proof of optimality of these algorithms holds true only under certain conditions.

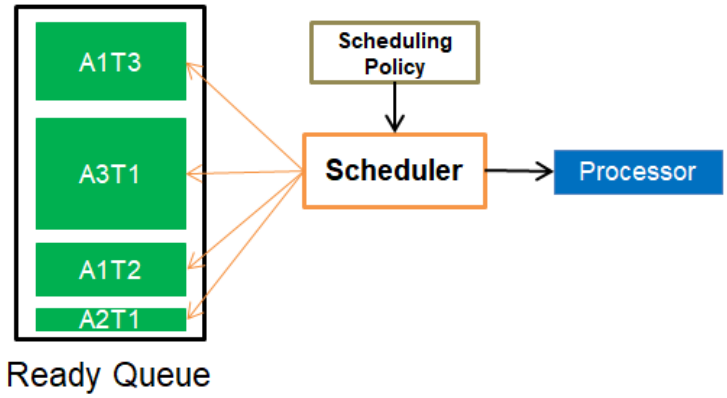


Figure 3.3: Uni-Core Scheduling

Figure 3.4: Temporal Organization

In multi-core processors, due to the availability of extra processing elements, threads can be scheduled on different cores and can be executed in parallel. The scheduler ensures thread-level parallelism in the system. However, in multi-core processors the problem of scheduling is two dimensional. The scheduler has to resolve the spatial and temporal dimensions i.e. the scheduler has to allocate threads to each core in addition to scheduling threads on each core. In this case, the problem of finding a feasible schedule is an NP-complete problem. Even though there are many recent studies on real-time multi-core scheduling, due to the nature of the problem, there will be no optimal solution. There are two different approaches to scheduling in multi-core systems: Global and Partitioned.

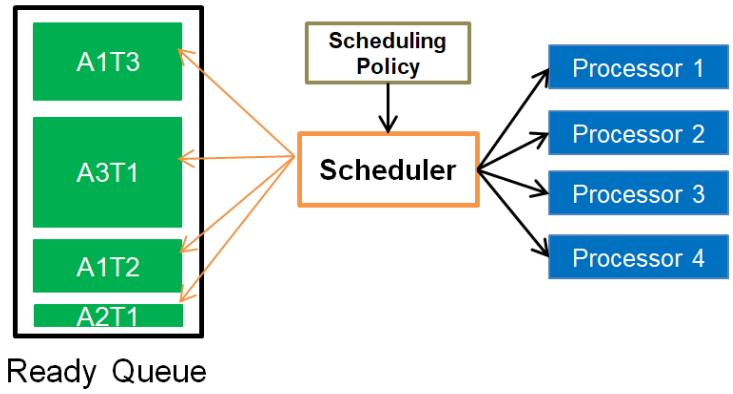


Figure 3.5: Multi-Core Scheduling

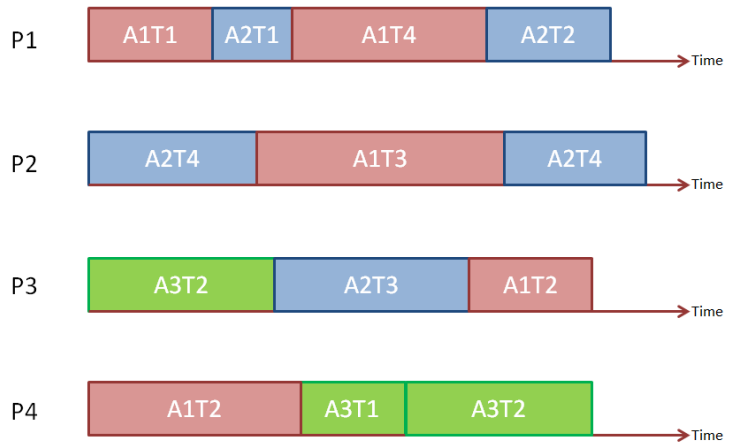


Figure 3.6: Temporal and Spatial Organization

3.3.1 Global scheduling

In the global scheduling approach, both the temporal and spatial dimensions are dealt with together. The scheduler selects a task from a single system-wide ready queue and assigns the task to an available core. The scheduler performs reallocation whenever a higher priority task is available for execution than the lowest priority task in any of the cores. The optimal uni-core scheduling policies like DM, RM and EDF are not optimal when used in this approach. But recent studies have shown that tardiness can be bound on Global-EDF algorithm if utilization bound is 0.69.

This approach commends a higher run time overhead for the scheduler. However, the advantage of this approach is a good load balancing among the cores. No core is ever idle when a task is ready to execute. So this approach is better for dynamic systems. There might be many preemptions and a lot of context switching which add up the response time of a task. But the major disadvantage of this approach is the inevitable task migration among different cores. Modern multi-core processors are organised with different levels of memory caches for faster

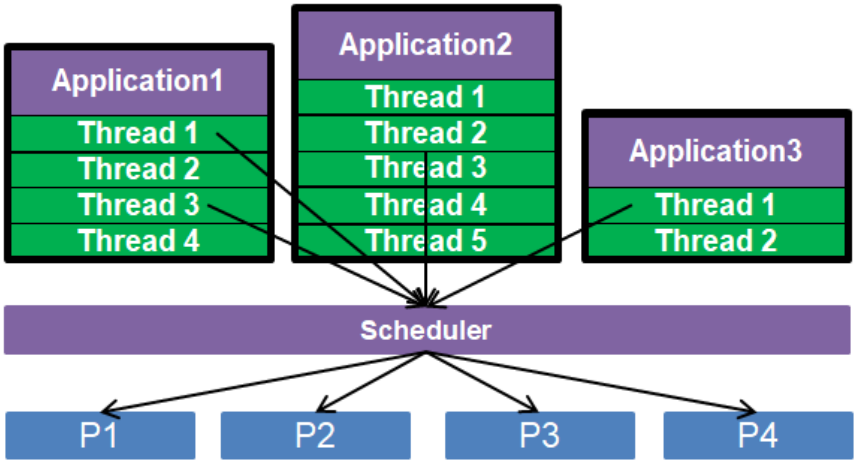


Figure 3.7: Global Scheduling Approach

execution of the instructions. Some caches are global but usually the faster caches are private to each core. If a particular task(T1) is continuously being executed on a particular core(C1), then the private cache of C1 is filled with the data needed by T1. This leads to a faster execution of T1 on C1. But when this task migrates to a new core(C2), the private cache of C2 may not have all the data needed by the T1. This leads to many cache misses until the data needed by T1 is made available in C2 cache.

3.3.2 Partitioned scheduling

In the partitioned scheduling approach, the temporal and spatial dimensions are dealt separately. This approach essentially resolves the multi-core scheduling problem into two separate problems. The first problem is the mapping of tasks to the cores. The second problem is the usual uni-processor temporal scheduling problem. With the uni-processor scheduling problem being an already well-known and understood problem, optimal scheduling algorithms like EDF, DM and RM policies can be used on each core. Mapping of tasks to the cores is a conventional bin-packing problem, which is NP-hard. Bin-Packing problem has heuristic approaches that lead towards optimal solutions. Only static task mapping is considered in this study.

The major disadvantage of this approach is that there might arise situations when a task in one of the cores is idle while a task in another core is waiting for execution. Following are the three major advantages of partitioned scheduling approach which makes them desirable.

1. Analysing the multi-core system becomes easier when using partitioned approach since this approach considers multi-core scheduling as a combination of multiple uni-processor scheduling problems.
2. The scheduling algorithm on each core can be independent of each other so different scheduling algorithms can be used on each core. This feature is desirable in certain applications. Especially, when a system contains a combination of safety critical tasks

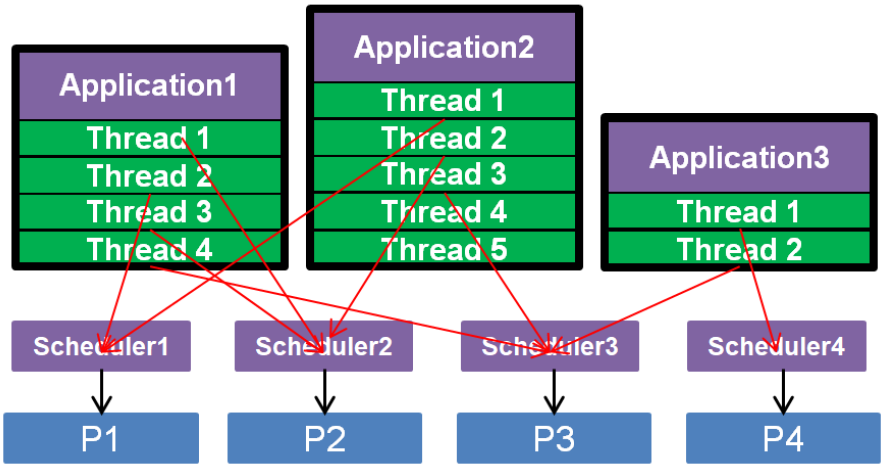


Figure 3.8: Partitioned Scheduling Approach

and general tasks, they can be scheduled in different cores and can be scheduled using different algorithms.

3. Since this approach does not have task migrations, the benefits of respecting the cache hierarchy are preserved. Scheduling using this approach is not work-conserving which is a major disadvantage.

4 Methodology

This chapter describes the approach used in order to achieve the objectives of this study. Beginning with the introduction of the requirements and followed by the flowchart(4.1) that summarises the steps involved in designing the experiment.

Table 4.1: Requirements

No.	Requirement
1	The selected processor shall have a core count of more than two
2	The development board shall be supported by RTEMS
3	RTEMS SMP shall selected processor shall be SMP support
4	The sample task set shall be generic and replicate a practical real-time application.
5	The selected sample task set shall be easier to recreate.
6	The sample task set shall be easier to implement.
7	The sample task set shall be analyzed for two different periods.
8	The test shall be conducted for five utilization of the task set ranging from 2.1 to 2.8
9	The process, starting from the task set generation until the deployment of the application on the hardware shall be automated.
10	Core utilization bound needed for partitioning shall be according to ECSS standards.
11	Kernel shall not be modified for the study.
12	Timing characteristics shall be observed outside of the system under test.
13	The duration of test shall be for at least one hyper-period.
14	The sampling time shall be less than half of the configured unit time of RTEMS.
15	Each test shall be repeated five times and the average values shall be used for the study.

4.1 Hardware selection

A hardware platform to carry out the experiments should be supported by RTEMS. Even though RTEMS supports many target platforms, the support for SMP exists only for ARMv7-A, PowerPC, RISC-V and SPARC architectures. In addition to the RTEMS support, the selected processor should also have a higher core count so that an efficient thread mapping shall be proposed. Raspberry Pi 2B board was selected because it fulfilled all these criteria. The Broadcom BCM2836 SoC used in Raspberry Pi 2B has four ARM Cortex A7 cores based on ARM v7-A ISA. Moreover, RTEMS provides the Board Support Package(BSP) for Raspberry Pi 2B. RTEMS SMP support for ARMv7-A is well tested because of the popularity of ZedBoard among

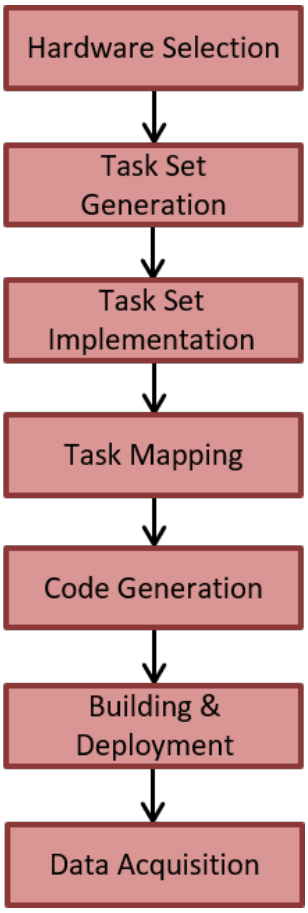


Figure 4.1: Methodology

the RTEMS community. Choosing ARM is also relevant since the qualification of RTEMS SMP is only being done for SPARC for ESA's NGMP. This study might give new insights into the RTEMS scheduler performance on ARM. Due to the popularity of ARM processors on embedded hardware, the scope for ARM SMP system is high. The silicon manufacturers have also started to offer Radiation-hardened hardware based on ARM for space applications[a2013].

4.2 Task set generation

To carry-out the schedulability analysis, a sample task set is required. All the tasks in the sample task set must be defined by deadline and period. Initially, implementation of a practical kalman filter algorithm was considered. It proved to be a tedious job to implement a practical algorithm and lacked the flexibility needed to achieve the objectives. Another limitation of using a practical algorithm is that the results are only valid for the particular application being studied. So a generic sample task set is required so that experimental results shall be applicable to any other application.

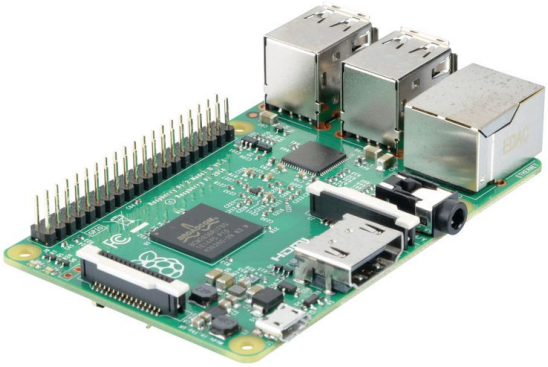


Figure 4.2: Raspberry Pi 2 Version B[ras]

In the design of real-time systems, abstract models are used to validate the non-functional properties such as the timing behavior in schedulability analysis[SEGY11]. These abstract models should express the system behavior as precisely as possible. It should also be possible to recreate the task set so that the tests are repeatable. This systematic approach makes the results more credible. Some scheduling studies, for simplicity, ignore the dependencies of tasks and consider all the tasks as independent and periodic. But in order to imitate the behaviour of a practical application, the model should also generate a task set with dependencies. The task model should be scalable.

Modeling the task set with Directed Acyclic Graph(DAG) is an approach usually used in many scheduling studies. With this approach, the task set can be considered as a collection of task graphs. Each task graph is a Directed Acyclic Graph(DAG). DAGs capture the task behaviour and produce expressive models that can be easily implemented. It can also provide dependency relations among tasks.

Task Graphs For Free(TGFF) is a popular tool among researchers in academia and research to generate DAGs[DRW98]. Fig 4.4 shows a DAG generated using TGFF. Each graph node is a task and the graph arc represents the dependency between the tasks. Tasks can be generated with different periods. Based on the depth of each graph, TGFF algorithm assigns the period and deadline for each node. TGFF includes a pseudo-random number generator[SEGY11]. The pseudo-random number accepts a seed parameter that controls the structure and other aspects of the graph. By varying the seed while holding all the other parameters constant, task set families with an arbitrary number of task sets may be generated. TGFF tool with its flexibility is a good fit for generating task sets for this study.

4.3 Task set implementation

Before proceeding with the experiment, the generated task set must be implemented as an RTEMS application. There is no single method of converting the task model into an actual implementation. Each method of implementation has its own overhead and they affect the

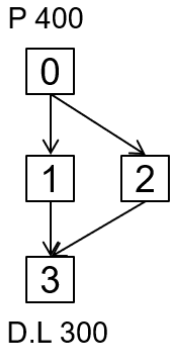


Figure 4.3: Simple DAG

results of the experiment. The implementation method used for a study may not be the best of all the possible implementations, but it must be able to sufficiently fulfill all the requirements of the study. The implementation methodology used for this analysis is documented in the following sections.

RTEMS offers two standard APIs, POSIX API and Classic RTEMS API, for developing applications. The Classic RTEMS API was chosen for implementing our sample task set since it is well documented and provides all the features needed for the analysis. In RTEMS perspective, a task is the smallest thread of execution which can compete on its own for system resources[Pro19]. Each Node of a task graph was implemented as an RTEMS task. For RTEMS classic API, the entry point for user applications is a task named "Init". In the "Init" task, all the tasks needed for the selected task set are created and then started. Each task was created in a separate C file so that they can be compiled individually. The implemented code is given in listing (5).

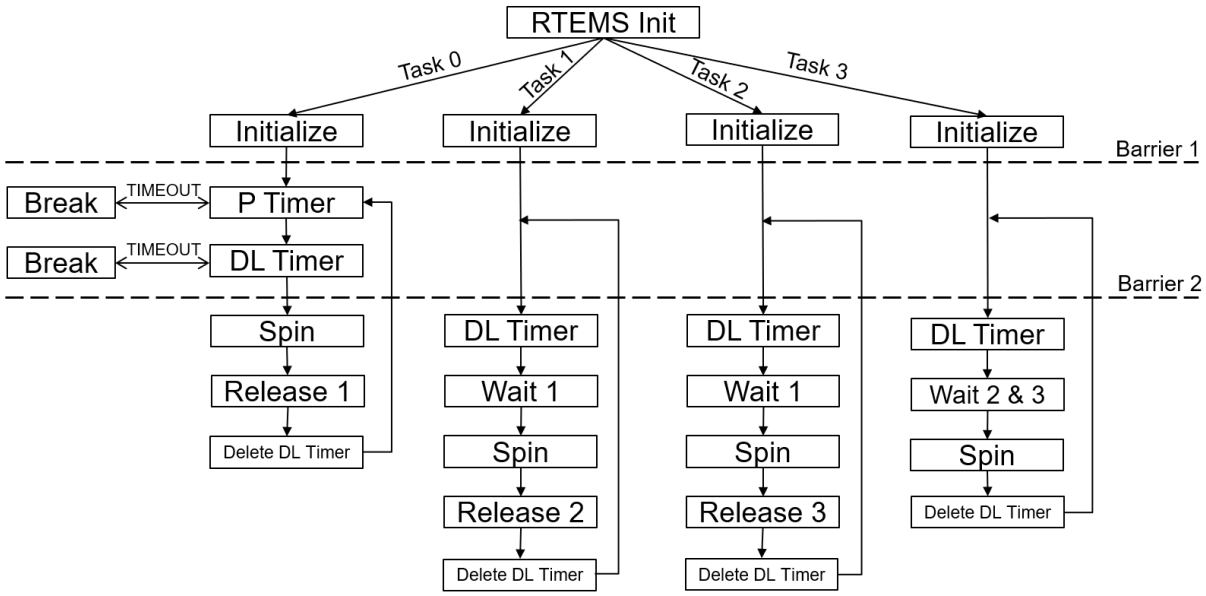


Figure 4.4: Implementation of the Simple DAG

4.3.1 Assignment of period and deadline

To schedule the task set with the Earliest Deadline First Scheduler, each task must be assigned with a deadline. In RTEMS, the period and the deadline for a task is assigned using the Rate Monotonic(RM) Manager. In addition to attribution of a task's deadline, the RM manager is responsible for the periodic activation of the task. To set a deadline for a task, a rate monotonic period object has to be created within the task. The argument needed for creating an RM period object is the task's period. Every deadline in RTEMS is treated as implicit i.e. the deadline of a task is equal to its period. This assumption introduces an extra overhead when implementing a task that has a deadline less than its period. RTEMS documentation gives an example for implementing such tasks in which two rate monotonic objects are created, one with the argument period and another one with the deadline[Pro19]. The RM object created with the deadline must be deleted after every execution of the task. This was done to avoid reactivation of the task at the deadline.

4.3.2 Generating execution time and period

As mentioned earlier, TGFF assigns deadlines to the sink-nodes i.e. nodes that does not fan out. TGFF also generates the period of all the generated graphs along with the execution time for each node. In this analysis, experiments were conducted on the same graphs with different Period and Utilization values. TGFF lacks an interface to relate execution time with the utilization of the graph. Therefore execution times that are generated by TGFF are ignored. UUniFast algorithm is used to generate the execution time of each node based on the Total utilization of a graph. This is done in two steps. Firstly, the UUnifast Algorithm breaks down the total utilization of the task set into individual utilization values for each and every task in the system. Secondly, using the period and the utilization values of each task, the execution time of individual tasks were calculated[DB09].

4.3.3 Spin loop

There is no computation needed in any of the tasks, but each task must be made to occupy the CPU for the duration of its execution time. A spin-loop/busy-wait function was implemented within each task to achieve this. RTEMS tracks time internally beginning from the system boot. Unit of time in RTEMS is a "tick". Tick duration can be configured while building the executable. For this analysis, a tick duration of 1 millisecond was selected. The execution time generated in milliseconds is converted to ticks while calling the spin function. When a task is given access to the CPU by the scheduler, it calls the spin function. Using the services offered by the RTEMS time manager, the spin function makes the task occupy the CPU until the duration of its execution time has elapsed.

4.3.4 Synchronization

To realize task dependencies for the sample task set, barrier and semaphore feature of RTEMS are used. Barriers are used when multiple tasks are required to wait until a certain synchronization point. In this implementation, barriers are used in two situations. The first is to realize the periodicity of a graph and secondly, for the initialization of all tasks. Since each graph has a single entry point, a single RM object created in the entry node controls the period of every task in the graph. All the tasks in the graph, except the entry task, waits for a signal at every period. This signal is realized using a barrier.

To implement precedence relations among the tasks, semaphores are used. Each precedence is thought of as a producer-consumer scenario and a single counting semaphore is used between two tasks, the preceding task controls the value of the semaphore. As long as the value of the semaphore is more than zero, the trailing task is allowed to execute. This strategy allows the tasks from previous periods to continue execution.

4.4 Thread mapping strategies

Partitioned scheduling approach calls for the tasks to be statically mapped to all the cores in the system. Uni-core schedulers are then used on each core to schedule the mapped tasks. Mapping tasks to cores is a bin-packing problem. For the given task set, the mapping can be done to satisfy either one of the two possible objectives. The tasks can either be mapped in a way to balance the load among all the cores or to minimize the number of cores used. From a power-saving point of view, the former approach allows the CPUs to run with less frequency while the later allows some of the cores to be disabled completely[SJPL08]. While both the approaches can be used for power-saving, there are also some hybrid strategies being studied. But only the two basic strategies are used in this study.

4.4.1 Bin-Packing heuristics

European Cooperation for Space Standardization(ECSS) standard specifies that the processor's utilization must be limited to 0.75 for uni-core processors. For multi-core processors, this can be extended proportionally to the number of cores in the processor. For the selected Raspberry Pi 2B hardware, with four CPU cores, the utilization shall be limited to a maximum utilization of 3. For a total utilization of 3, First-Fit Decreasing Utilization(FFDU) and Worst-Fit Decreasing Utilization(WFDU) heuristics were used for task mapping in this study.

In both the FFDU and WFDU heuristics, the tasks are first arranged in the decreasing order of their utilization values. The FFDU algorithm then iterates over the sorted tasks and maps them to the first available core. The mapping generated by the FFDU algorithm achieves the minimum possible cores needed for scheduling the given task set. The WFDU algorithm iterates over the sorted tasks and maps them to the first core with maximum availability. The mapping generated by the WFDU algorithm achieves balance across cores.

4.5 Code generation

While designing the experiment, a test bench, through which the entire process of task set generation to building an executable can be automated, was considered. The test bench will assist in analysing many task graphs with different parameters with ease. TGFF accepts a ".tgfopt" file that contains the configuration for generating a task set. This configuration file will be the entry point of the test bench where the users are allowed to modify the parameters to generate the desired task set. TGFF outputs two files that contain information about the generated task set. The ".eps" file contains the visual representation of the graph and it can be viewed with any postscript viewing program. The ".tgff" file has the text-based information like task name, period and precedence relations of the generated graph[DRW98]. This file will be the input of a parser program that creates a high-level data structure of the generated task set. These high-level data structures of the generated task set are either the dictionary or the list data type of the python programming language, that can be easily processed further. The code generator essentially outputs three header files:

timimg.h :

The execution time arguments for the spin function in each task are given as macros. The period or deadline parameter for creating the RM object is also given as macros. The definitions of these macros are found in the "timing.h" header file. A limited sample code generated is given in listing (1)

system.h : The first part of the system.h file contains the RTEMS configuration which are kept constant for all the tests. The second part of the file contains the scheduler configuration. Global schedulers are configured in a single line but partitioned scheduling is configured according to the following steps.

1. Selection of the scheduler algorithms
2. Initialize the scheduler data structures
3. Populate the scheduler table
4. of schedulers to the cores

A limited sample code generated is given in listing (4)

ffdu.h and wfdu.h : After the UUnifast algorithm assigns the individual utilization values, the heuristic algorithms are called. The FFDU and WFDU heuristic algorithms maps individual tasks to the cores based on the utilization limits on each core. In RTEMS, mapping of the tasks to the cores is done by calling the "rtems_task_set_schedule" method. This method must be called before starting the task with two arguments, the name of the scheduler instant defined in "system.h" and the task name. The data structure created by the parser program contains the task name. The scheduler names are fixed for every scheduler algorithm. According to the mapping generated by the ffdu and wfdu algorithms, the code generator creates the corresponding header files with "rtems_task_set_schedule" call for every task. A limited sample

code generated is given in listing (2) and listing (3)

4.6 Building and deploying the executable

To develop RTEMS application, the RTEMS kernel for Raspberry Pi 2B should be built. RTEMS offers its own variation of toolchain containing cross-compiler, linker, debuggers and other debugging tools. To make the process of building the cross-toolchain easier, RTEMS offers RTEMS Source Builder(RSB) tool. After building the RTEMS toolchain using RSB, the RTEMS kernel source code can be downloaded and built as a static library. The same toolchain can then be used to compile any RTEMS application and linked with the RTEMS library to create an executable.

The standard method of deploying the created RTEMS executable on Raspberry Pi 2B is by manually transferring them via a micro-SD card. Deployment can be automated by using a boot-loader. U-boot is an open source preliminary boot-loader used in embedded devices. U-boot extends the Raspberry Pi's capability to read executable via network using the Trivial File Transfer Protocol(TFTP). It was found that due to incompatibilities between the latest RTEMS kernel and the Raspberry Pi 2B's firmware, deploying the RTEMS executable via network did not work. So the RTEMS executables are deployed manually via a micro-SD card.

4.7 Data acquisition

For every task with a deadline, three parameters are observed.

1. Response time of the task.
2. Number of deadline misses.
3. Lateness of every execution.

RTEMS maintains an internal data structure containing the CPU usage and RM object statistics. This way of capturing the parameters internally in an SMP system demands more synchronization overhead and thereby increasing the complexity of the program. A simpler solution to capture these parameters externally was therefore used for this study. Every tasks with a deadline is assigned a general-purpose Input Output(GPIO) pin. As a part of system initialization, every GPIO pin is set to Low. In every deadline task, before calling the spin function, it sets the value of its GPIO pin to High. After returning from the spin function, the GPIO pin is set to Low. To calculate the response time of the task, the start time of every period is required. To find the start time of every period, the same methodology of toggling GPIO values is used in the entry node of every task graph. By measuring the time taken between the start time of a period and the corresponding High to Low transition of a task, the response time of the task can be deduced.

For the purpose of observing the GPIO toggling events, an eight-channel logic analyser with a maximum sampling frequency of 500KHz was used. Sigrok's open source signal analysis software suite was used on the host to record the GPIO pin values. Sigrok-cli tool can record the observed values in many formats. CSV files are easier for post-processing but consumes more storage space for the desired sampling rate and sampling period, so the .sr file format was selected. Sigrok/v2(.sr) file format is a proprietary compressed file format of the Sigrok project that can consume lesser storage space. But to avoid excess usage of storage, each of the .sr files should be extracted one at a time before post processing the data.

5 Evaluation

In this chapter the sample task set is evaluated under different scheduling algorithms and the results are presented. Selection of the task graphs for implementation and the post-processing of the experimental data is given.

5.1 Experimental setup

Listing 5.1: TGFF input

```
tg_cnt 2
task_cnt 20 5
task_degree 3 2
period_laxity 1
period_mul 1, 0.5, 2
tg_write
eps_write
vcg_write

table_label COMMUN
table_cnt 3
table_attrib price 80 20
type_attrib exec_time 50 20
trans_write
```

The task set was generated with the "simple.tgffopt" configuration file included with the TGFF tool. This configuration listing (5.1) generates five task graphs. Due to time constraints for this study, only the first two(TASK_GRAPH 0 and TASK_GRAPH 1) task graphs were selected as the sample task set for this study. TASK_GRAPH 0 has a single implicit deadline while TASK_GRAPH 1 has six deadlines that are less then its period. These two task sets were implemented as a single RTEMS application. Five different utilization values for the task set were considered. In addition to the period generated by TGFF, the sample task set with one more randomly selected period was analysed. Real-time performance of the different schedulers were analysed for each utilization and period. The following table(5.1) summarises the inputs of all the tests conducted:

The following are the RTEMS SMP schedulers:

1. Earliest Deadline First SMP Scheduler(EDF_SMP)

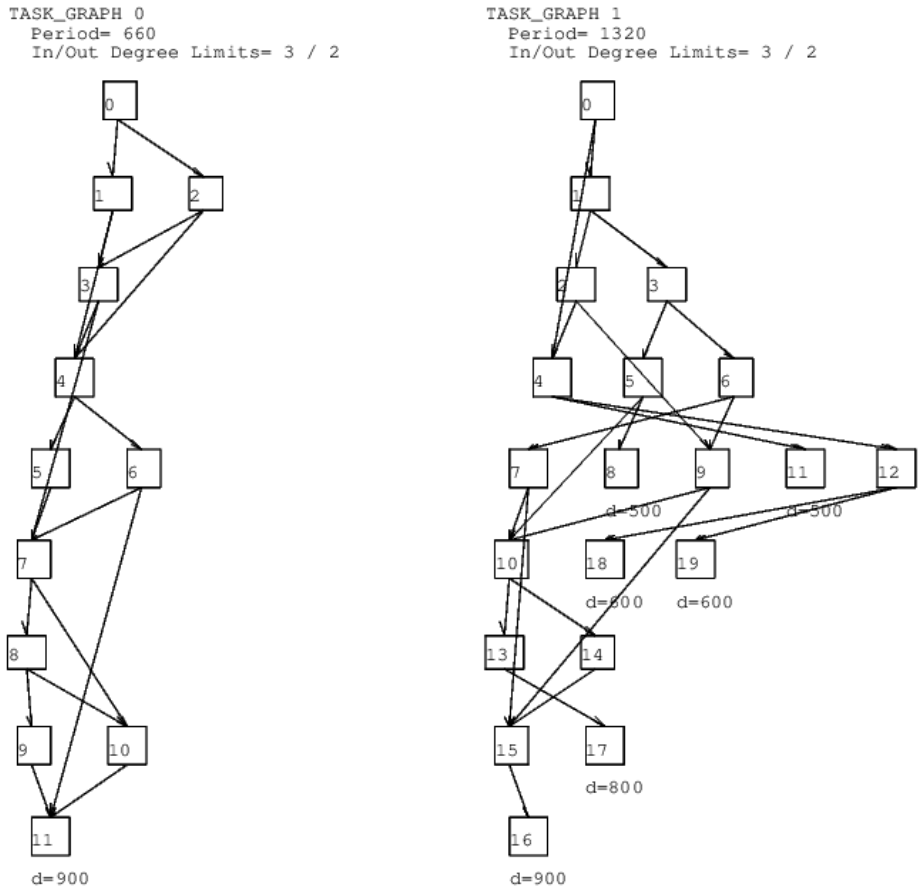


Figure 5.1: Sample Task Set

2. Deterministic Priority SMP Scheduler(DP_SMP)
3. Simple Priority SMP Scheduler(SP_SMP)
4. Arbitrary Processor Affinity Priority SMP Scheduler(AP_SMP)

SP_SMP and AP_SMP schedulers are the variations of the DP_SMP. DP_SMP is a fixed priority scheduler with one ready queue per priority level. SP_SMP is also a fixed priority scheduler but in the effort to reduce the memory footprint of the kernel, the tasks of all priority level share a single ready queue. AP_SMP is also a fixed priority scheduler with the addition of processor affinity support. Since SP_SMP and AP_SMP are just the variations of the two base schedulers, only the two base schedulers(EDF_SMP and DP_SMP) are used in this study. Both the Global Partitioned approaches are studied. For Partitioned approach, two heuristics are used for mapping the threads to cores. Experiment was conducted and the data was collected for a duration of one hyper-period. A total of sixty experiments were conducted and each experiment was repeated five times and the average values were collected.

Table 5.1: Experimental Parameters

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10
Total Utilization	2.1		2.3		2.5		2.7		2.8	
Gr.0 Utilization	1.3		1.6		1		0.9		1.4	
Graph 0 Period	900	2700	900	1800	900	3600	900	3600	900	2700
Gr.1 Utilization	0.8		0.7		1.5		1.8		1.4	
Graph 1 Period	1320	5280	1320	5280	1320	1320	1320	1320	1320	3960

5.2 Post-processing the experimental data

As mentioned in 4, due to high memory overhead, the test data were stored in ".sr" file format. It is a compressed file format and for processing, each file must be extracted, processed and deleted one after the other. The number of deadline tasks in the sample task set is seven, so seven GPIO pins were allocated to them. Since two task graphs with different periods were used, start time of each period must be captured. Due to the limitation in the number of channels in the data logger, single channel was used to capture the period information of both the graphs. The entry node of both the tasks were allowed to access the same GPIO pin. In-order to differentiate the periods of the two different task graphs, at the beginning of every period, the entry node of TASK_GRAPH 0 would generate a pulse with a pulse width of 1ms and the entry node of TASK_GRAPH 1 would generate a pulse with a pulse width of 2ms. While processing the experiment data it was found that, during certain point in time, it was hard to differentiate the period signals between the two task graphs. To solve this issue only the start time of the initial period was collected from the data and all the start times are generated from the initial period. The results of all the experiments are plotted in the section.

5.3 Results

To evaluate different schedulers, three different parameters are plotted. The core utilization after mapping the tasks using FFDU and WFDU strategies, the response time of deadline tasks and the number of deadline misses.

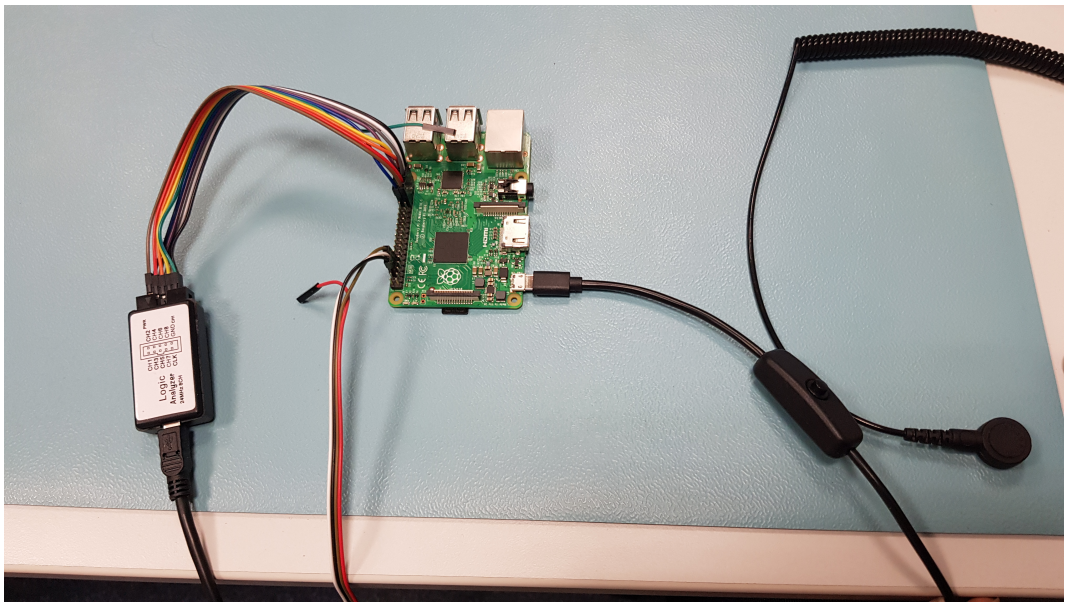
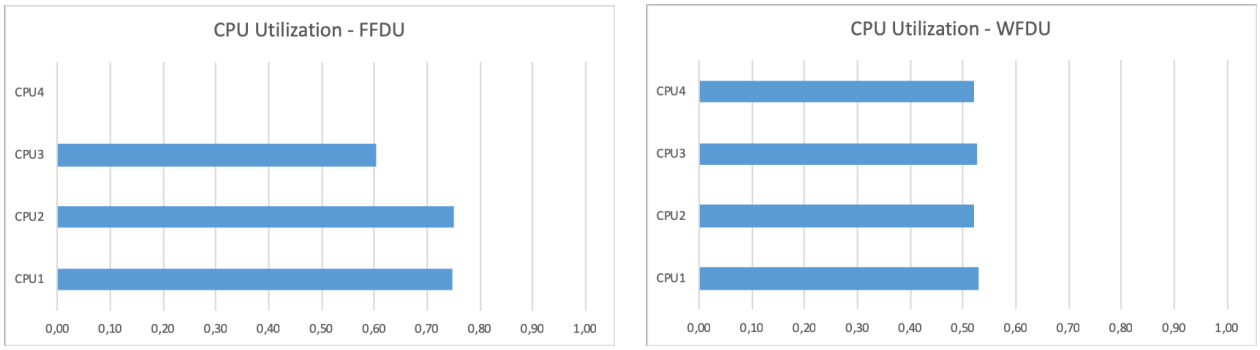


Figure 5.2: Experimental setup

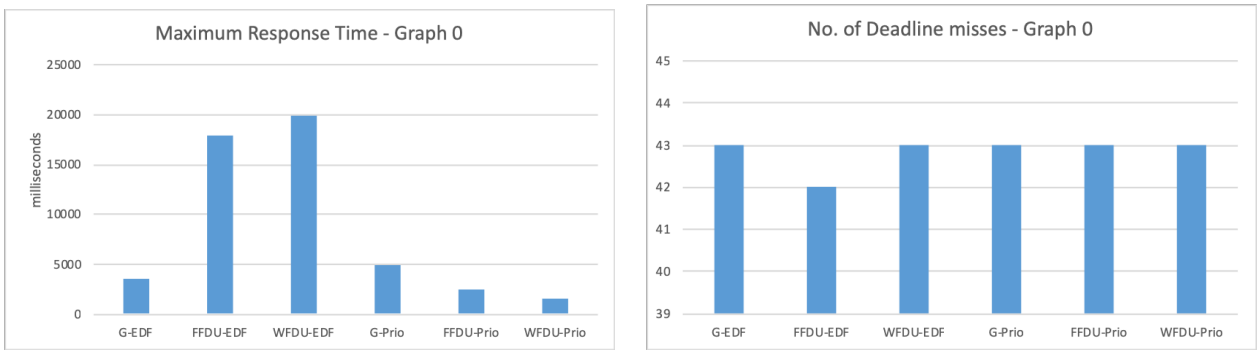
Test Case 1



(a) **(b)**

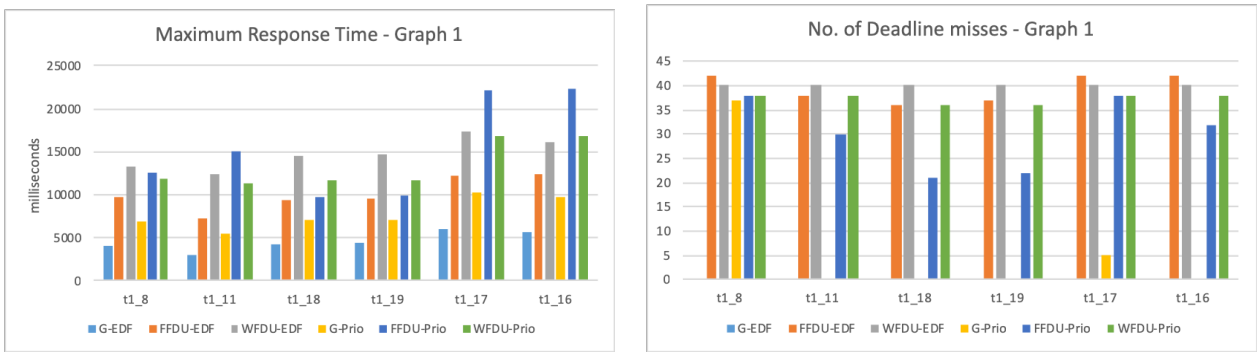
Figure 5.3: Core Utilization - Test Case 1

- The mapping of FFDU gives a minimum core count of 3 for scheduling the given task set with a total utilization of 2.1
- The mapping of WFDU balances the utilization of all the cores with the utilization of all the cores being just over 0.5
- Given a implicit deadline on TASK_GRAPH 0, the WFDU-Priority Scheduler achieves the shortest average response time among all the schedulers.
- Given a implicit deadline on TASK_GRAPH 0 with a deadline/period of 900ms, none of the six schedulers were able to meet the deadline.



(a) **(b)**

Figure 5.4: TASK_GRAPH 0 - Test Case 1

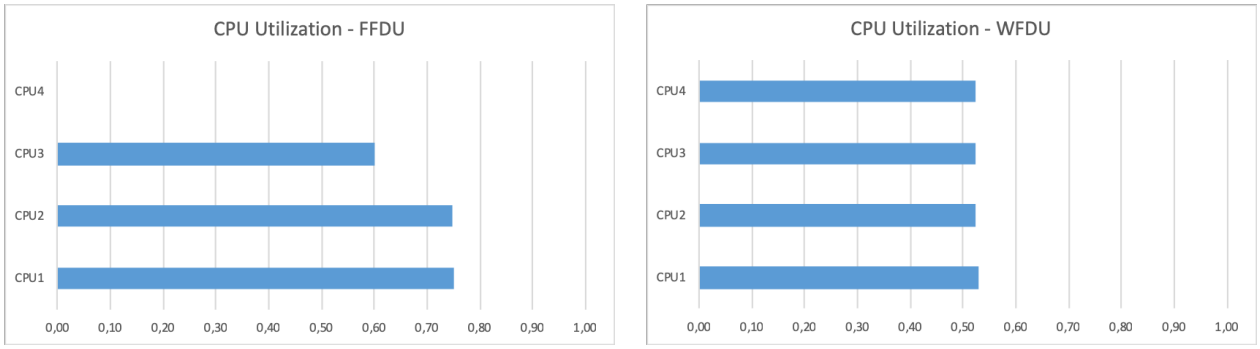


(a) **(b)**

Figure 5.5: TASK_GRAPH 1 - Test Case 1

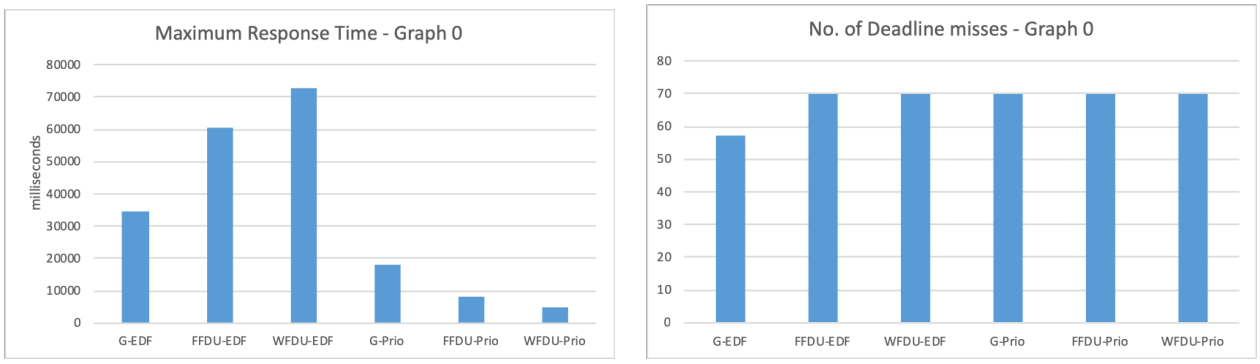
- Given six deadline tasks on TASK_GRAPH 1, the Global EDF scheduler achieves a minimum average response time among all the schedulers.
- Given TASK_GRAPH 1 with period of 1320ms and six bounded deadline tasks with a deadline of 500ms, 500ms, 600ms, 600ms, 800ms and 900ms respectively, both the Global-EDF and Global-Priority schedulers were able to meet all the deadlines.

Test Case 2



(a) **(b)**

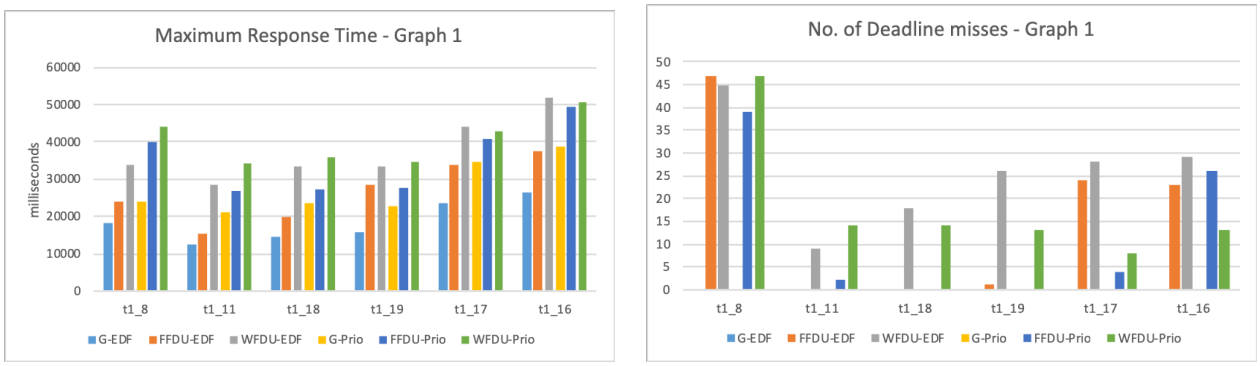
Figure 5.6: Core Utilization - Test Case 2



(a) **(b)**

Figure 5.7: TASK_GRAPH 0 - Test Case 2

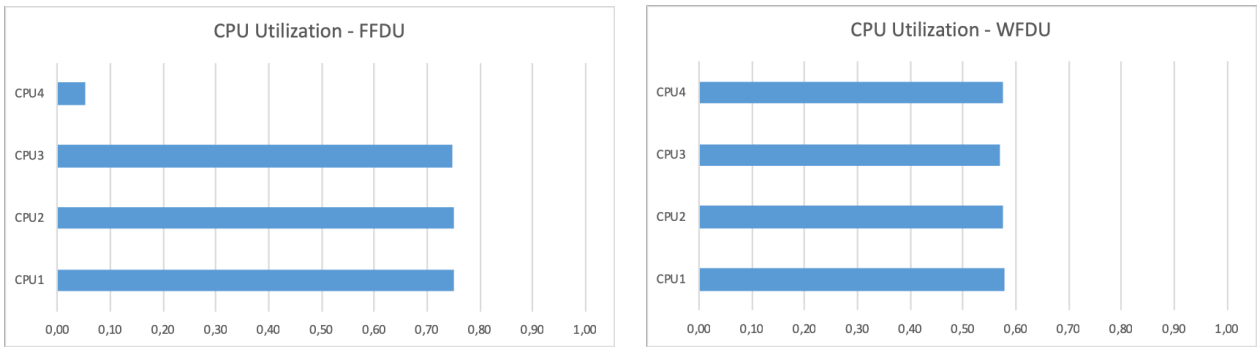
- The mapping of FFDU gives a minimum core count of 3 for scheduling the given task set with a total utilization of 2.1
- The mapping of WFDU balances the utilization of all the cores with the utilization of all the cores being just over 0.5
- Given a implicit deadline on TASK_GRAPH 0, the WFDU-Priority Scheduler achieves a minimum average response time among all the schedulers.
- Given a implicit deadline on TASK_GRAPH 0 with a deadline/period of 2700ms, none of the six schedulers were able to meet the deadline.
- Given six deadline tasks on TASK_GRAPH 1, the Global EDF scheduler achieves a minimum average response time among all the schedulers.
- Given TASK_GRAPH 1 with period of 5280ms and six bounded deadline tasks with a deadline of 2000ms, 2000ms, 2400ms, 2400ms, 3200ms and 3600ms respectively, both the Global-EDF and Global-Priority schedulers were able to meet all the deadlines.



(a) **(b)**

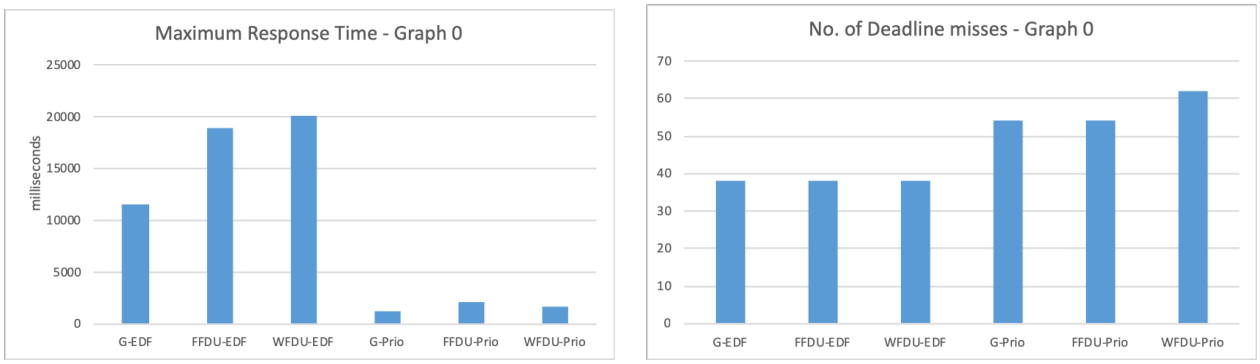
Figure 5.8: TASK_GRAPH 1 - Test Case 2

Test Case 3



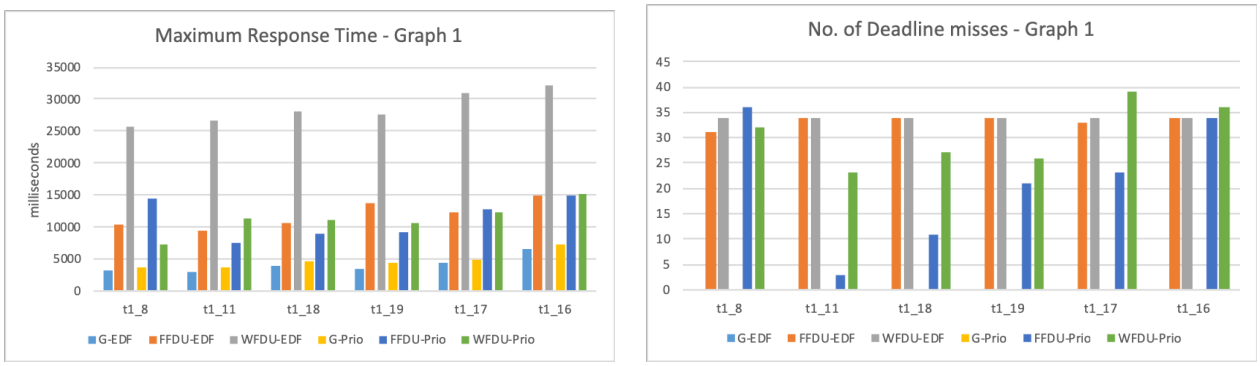
(a) **(b)**

Figure 5.9: Core Utilization - Test Case 3



(a) Maximum Execution time of t0_11 **(b)**

Figure 5.10: TASK_GRAPH 0 - Test Case 3

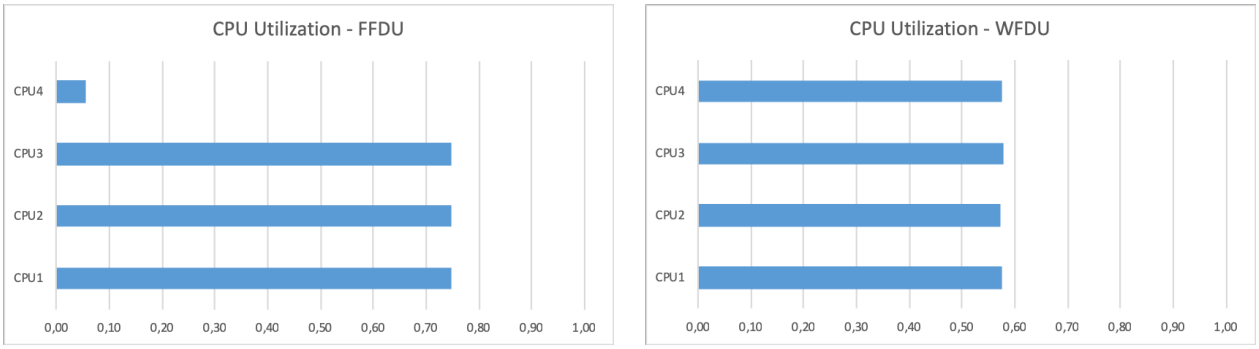


(a) **(b)**

Figure 5.11: TASK_GRAPH 1 - Test Case 3

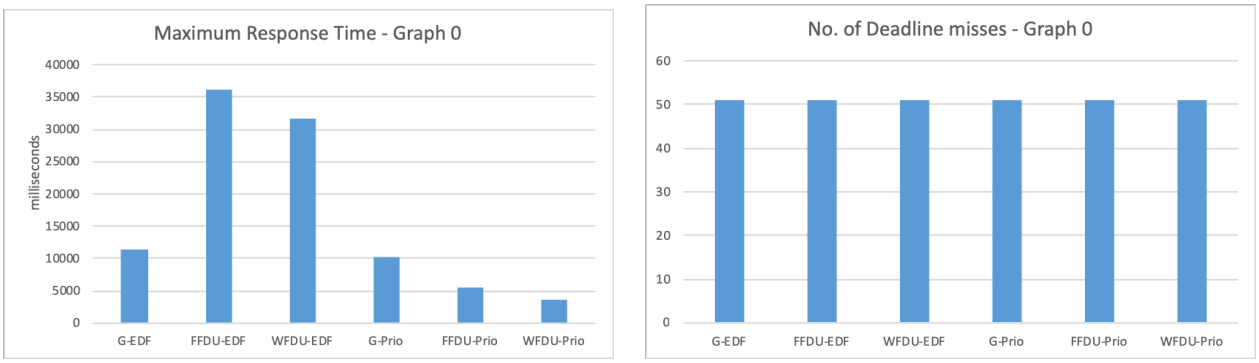
- The mapping of FFDU gives a minimum core count of 4 for scheduling the given task set with a total utilization of 2.3. Core 4 has an utilization of about 0.05.
- The mapping of WFDU balances the utilization of all the cores with the utilization of all the cores being just under 0.6
- Given a implicit deadline on TASK_GRAPH 0, the Global-Priority Scheduler achieves a minimum average response time among all the schedulers.
- Given a implicit deadline on TASK_GRAPH 0 with a deadline/period of 900ms, all the three variation of EDF algorithms a achieve comparatively shorter average response times.
- Given six deadline tasks on TASK_GRAPH 1, the Global EDF scheduler achieves the minimum average response time among all the schedulers.
- Given TASK_GRAPH 1 with period of 1320ms and six bounded deadline tasks with a deadline of 500ms, 500ms, 600ms, 600ms, 800ms and 900ms respectively, both the Global-EDF and Global-Priority schedulers were able to meet all the deadlines.

Test Case 4



(a) **(b)**

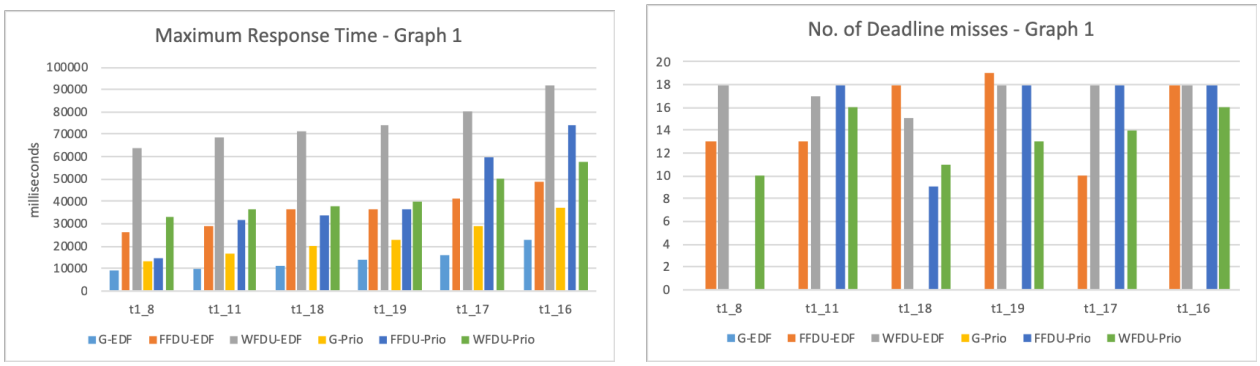
Figure 5.12: Core Utilization - Test Case 4



(a) Maximum Execution time of t0_11 **(b)**

Figure 5.13: TASK_GRAPH 0 - Test Case 4

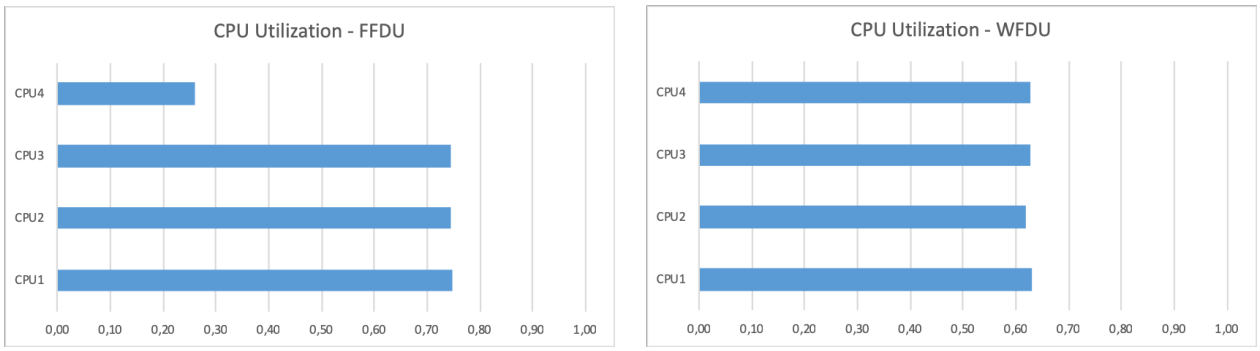
- The mapping of FFDU gives a minimum core count of 4 for scheduling the given task set with a total utilization of 2.3. Core 4 has an utilization of about 0.05.
- The mapping of WFDU balances the utilization of all the cores with the utilization of all the cores being just under 0.6
- Given a implicit deadline on TASK_GRAPH 0, the WFDU-Priority Scheduler achieves a minimum average response time among all the schedulers.
- Given a implicit deadline on TASK_GRAPH 0 with a deadline/period of 1800ms, all the six schedulers were unable to meet the deadline.
- Given six deadline tasks on TASK_GRAPH 1, the Global EDF scheduler achieves a minimum average response time among all the schedulers.
- Given TASK_GRAPH 1 with period of 5280ms and six bounded deadline tasks with a deadline of 2000ms, 2000ms, 2400ms, 2400ms, 3200ms and 3600ms respectively, both the Global-EDF and Global-Priority schedulers were able to meet all the deadlines.



(a) **(b)**

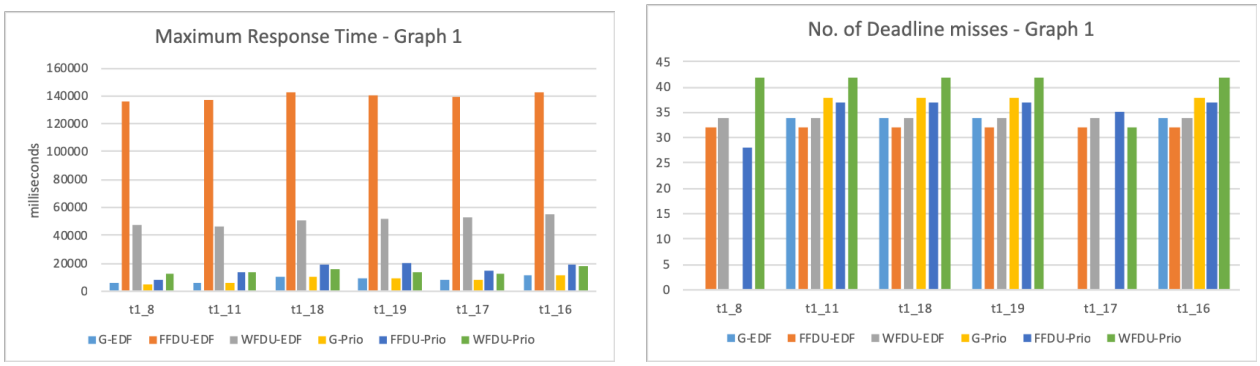
Figure 5.14: TASK_GRAPH 1 - Test Case 4

Test Case 5



(a) **(b)**

Figure 5.15: Core Utilization - Test Case 5

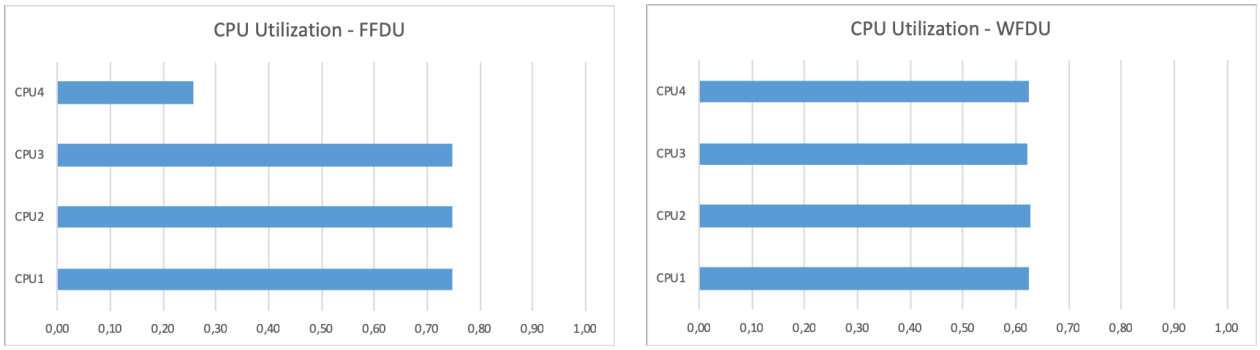


(a) **(b)**

Figure 5.17: TASK_GRAPH 1 - Test Case 5

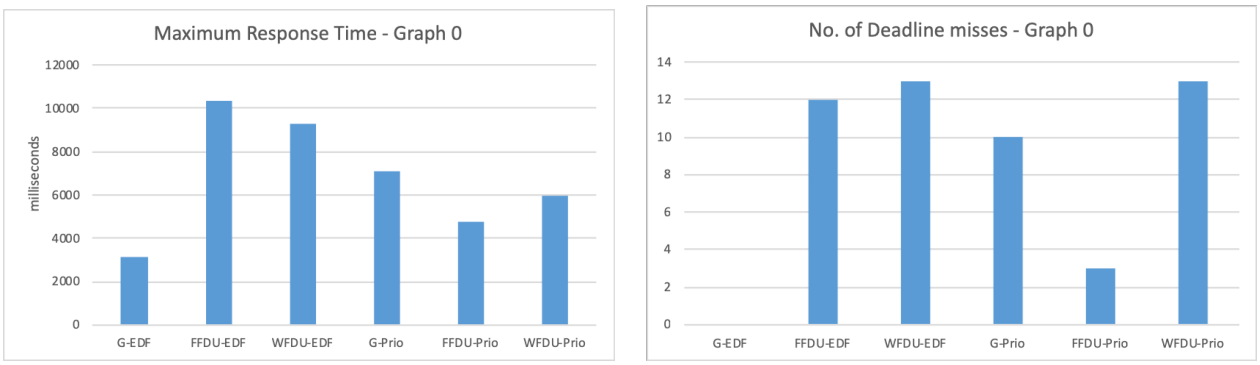
- The mapping of FFDU gives a minimum core count of 4 for scheduling the given task set with a total utilization of 2.5. Core 4 has an utilization of about 0.25
- The mapping of WFDU balances the utilization of all the cores with the utilization of all the cores being just over 0.6
- Given a implicit deadline on TASK_GRAPH 0, all the three variation of Priority schedulers achieve comparatively a minimum average response times.
- Given a implicit deadline on TASK_GRAPH 0 with a deadline/period of 900ms, only the Global-Priority scheduler was capable of meeting the deadline.
- Given six deadline tasks on TASK_GRAPH 1, FFDU-EDF and WFDU-EDF have relatively higher response times.
- Given TASK_GRAPH 1 with period of 1320ms and six bounded deadline tasks with a deadline of 500ms, 500ms, 600ms, 600ms, 800ms and 900ms respectively, only the Global-EDF scheduler was able to meet all the deadlines.

Test Case 6



(a) **(b)**

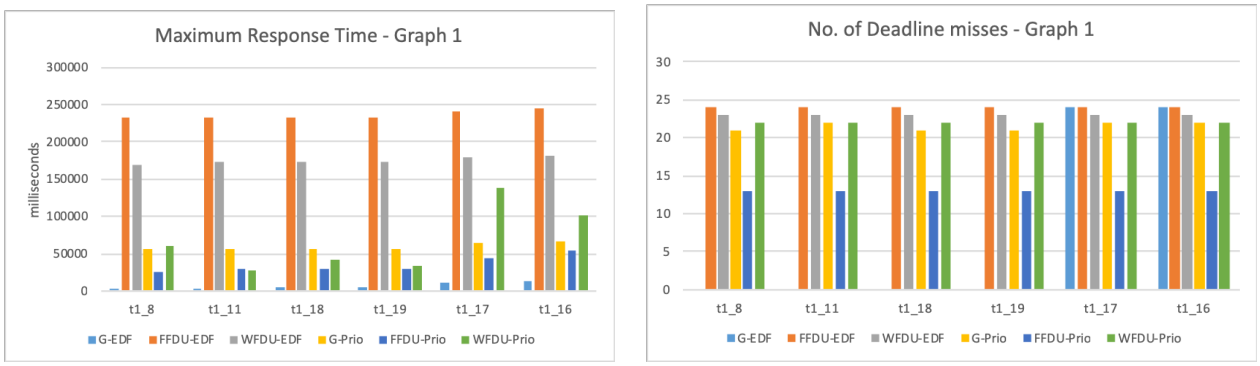
Figure 5.18: Core Utilization - Test Case 6



(a) Maximum Execution time of t0_11 **(b)**

Figure 5.19: TASK_GRAPH 0 - Test Case 6

- The mapping of FFDU gives a minimum core count of 4 for scheduling the given task set with a total utilization of 2.5. Core 4 has an utilization of about 0.25
- The mapping of WFDU balances the utilization of all the cores with the utilization of all the cores being just over 0.6
- Given a implicit deadline on TASK_GRAPH 0, the Global-EDF Scheduler achieves the shortest average response time among all the schedulers.
- Given a implicit deadline on TASK_GRAPH 0 with a deadline/period of 3600ms, only the the Global-EDF Scheduler was able to meet the deadline.
- Given six deadline tasks on TASK_GRAPH 1, the Global-EDF Scheduler achieves the shortest average response time among all the schedulers.
- Given TASK_GRAPH 1 with period of 1320ms and six bounded deadline tasks with a deadline of 500ms, 500ms, 600ms, 600ms, 800ms and 900ms respectively, all the six

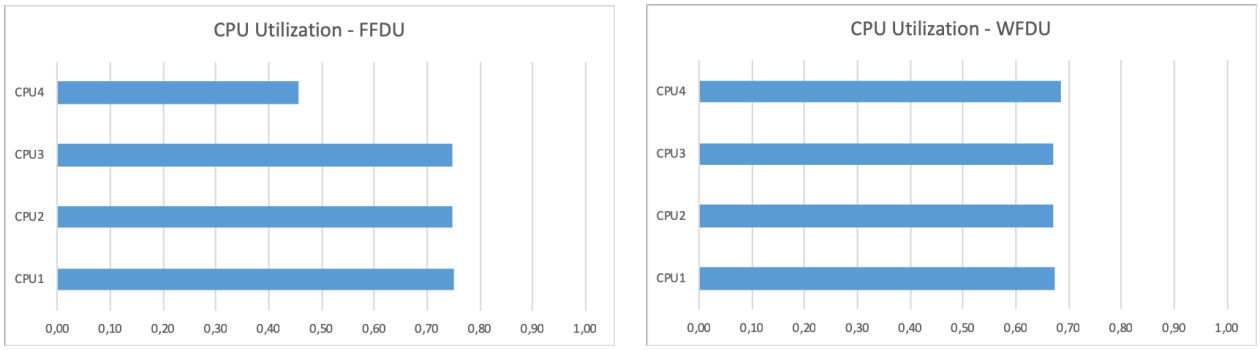


(a) **(b)**

Figure 5.20: TASK_GRAPH 1 - Test Case 6

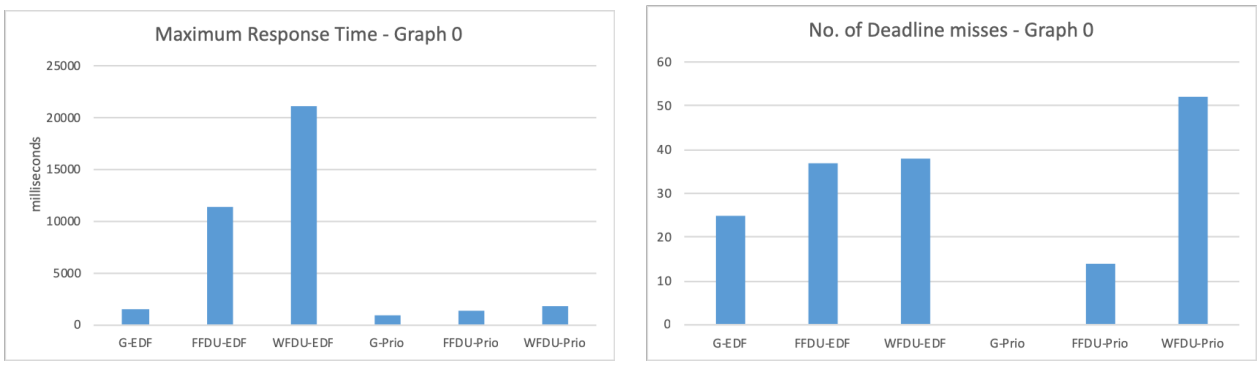
schedulers were unable to meet the deadline. The FFDU-Priority scheduler was able to achieve fewer deadline misses.

Test Case 7



(a) **(b)**

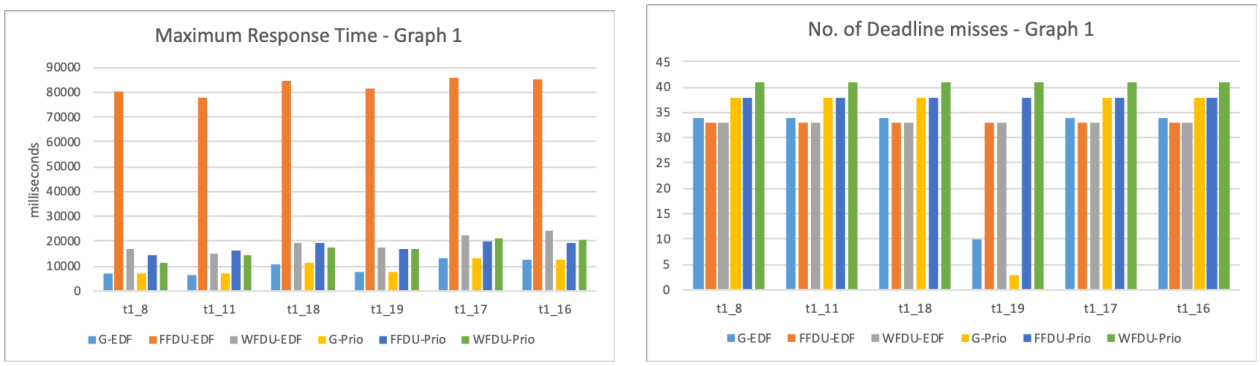
Figure 5.21: Core Utilization - Test Case 7



(a) Maximum Execution time of t0_11 **(b)**

Figure 5.22: TASK_GRAPH 0 - Test Case 7

- The mapping of FFDU gives a minimum core count of 4 for scheduling the given task set with a total utilization of 2.7. Core 4 has an utilization of about 0.45.
- The mapping of WFDU balances the utilization of all the cores with the utilization of all the cores being just under 0.7
- Given a implicit deadline on TASK_GRAPH 0, the Global-EDF Scheduler achieves the shortest average response time among all the schedulers. FFDU-EDF and WFDU-EDF schedulers causes significantly higher response times.
- Given a implicit deadline on TASK_GRAPH 0 with a deadline/period of 900ms, only the the Global-Priority Scheduler was able to meet the deadline.
- Given six deadline tasks on TASK_GRAPH 1, both the Global-EDF and Global-Priority schedulers achieve comparatively shorter average response times.

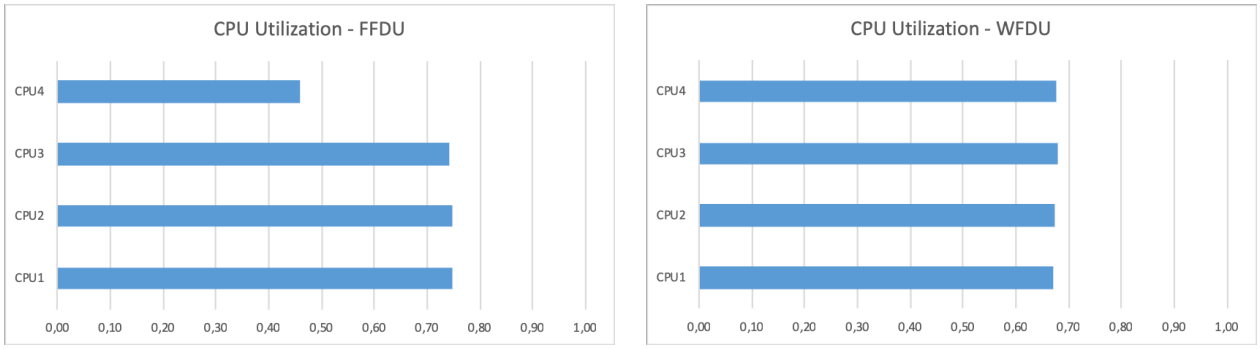


(a) **(b)**

Figure 5.23: TASK_GRAPH 1 - Test Case 7

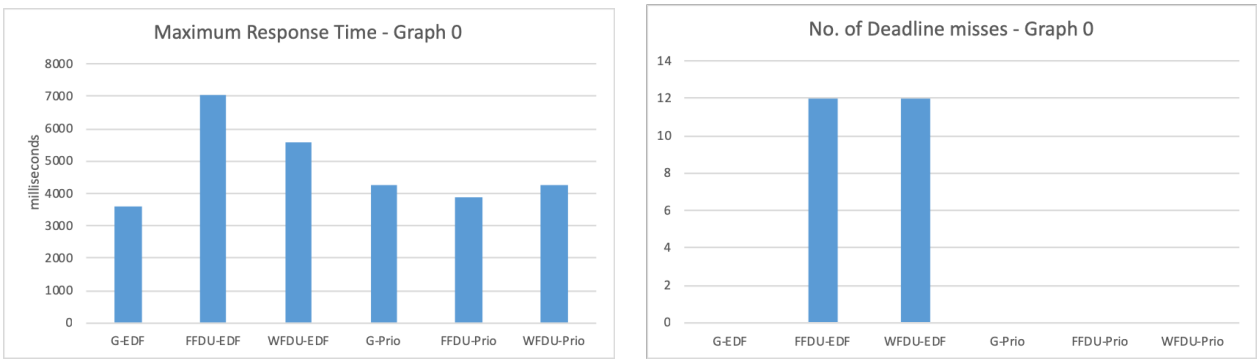
- Given TASK_GRAPH 1 with period of 1320ms and six bounded deadline tasks with a deadline of 500ms, 500ms, 600ms, 600ms, 800ms and 900ms respectively, all the six schedulers were unable to meet the deadline.

Test Case 8



(a) **(b)**

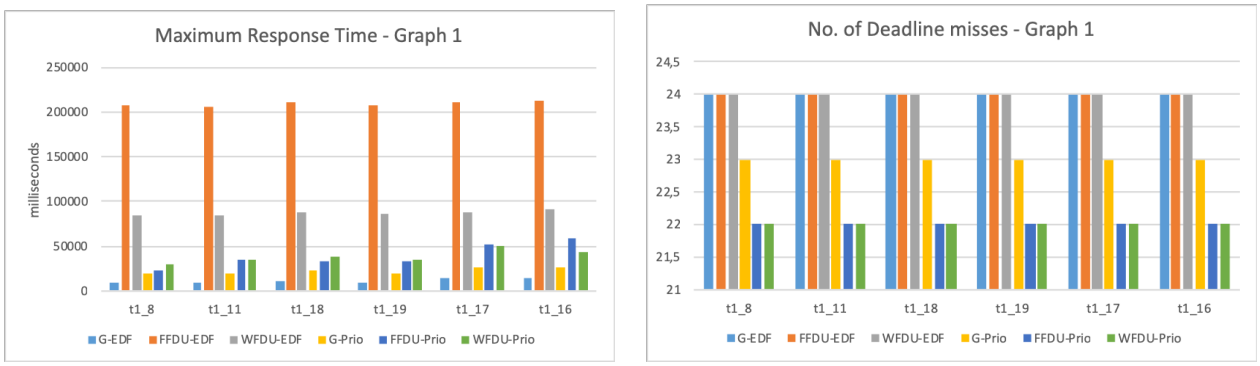
Figure 5.24: Core Utilization - Test Case 8



(a) Maximum Execution time of t0_11 **(b)**

Figure 5.25: TASK_GRAPH 0 - Test Case 8

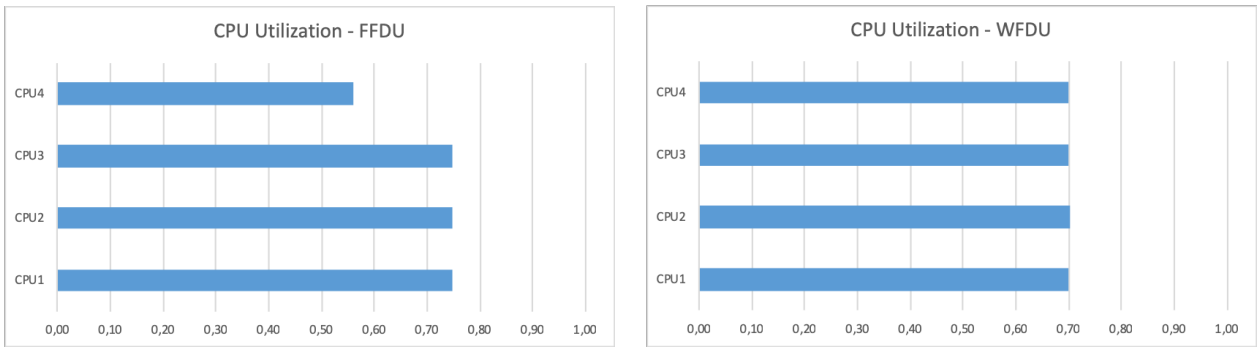
- The mapping of FFDU gives a minimum core count of 4 for scheduling the given task set with a total utilization of 2.7. Core 4 has an utilization of about 0.45.
- The mapping of WFDU balances the utilization of all the cores with the utilization of all the cores being just under 0.7.
- The shortest average response time of TASK_GRAPH 0 is achieved by Global-EDF scheduler.
- Given a implicit deadline on TASK_GRAPH 0 with a deadline/period of 3600ms, all the schedulers except FFDU-EDF and WFDU-EDF were able to meet the deadline.
- Given six deadline tasks on TASK_GRAPH 1 the Global-EDF algorithm achieves the shortest average response time among all the other schedulers.
- Given TASK_GRAPH 1 with period of 1320ms and six bounded deadline tasks with a deadline of 500ms, 500ms, 600ms, 600ms, 800ms and 900ms respectively, all the six schedulers were unable to meet the deadline.



(a) **(b)**

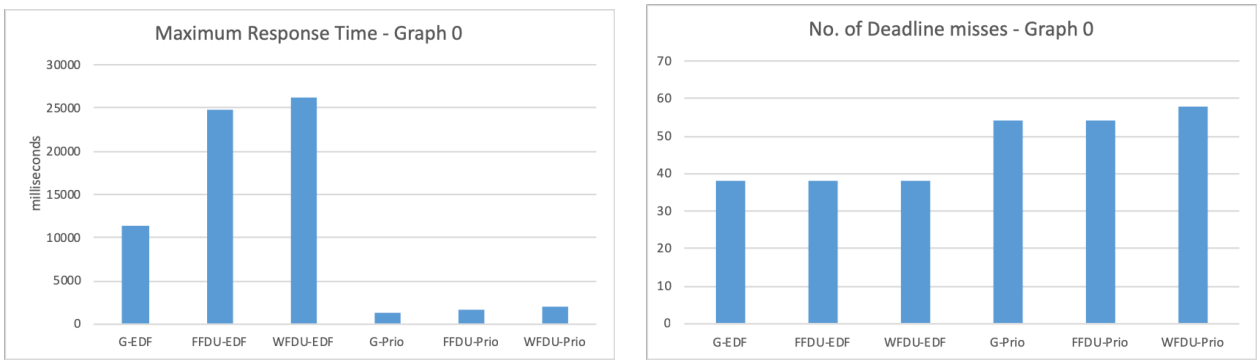
Figure 5.26: TASK_GRAPH 1 - Test Case 8

Test Case 9



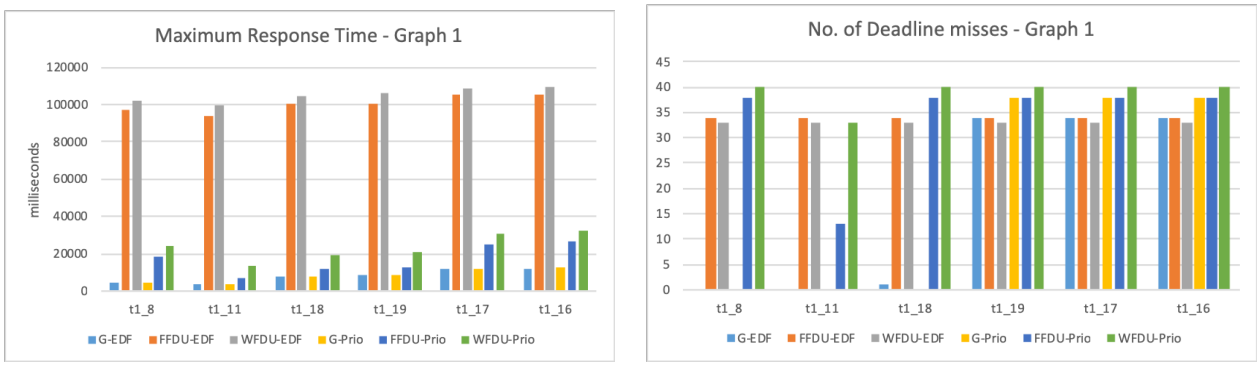
(a) **(b)**

Figure 5.27: Core Utilization - Test Case 9



(a) Maximum Execution time of t0_11 **(b)**

Figure 5.28: TASK_GRAPH 0 - Test Case 9

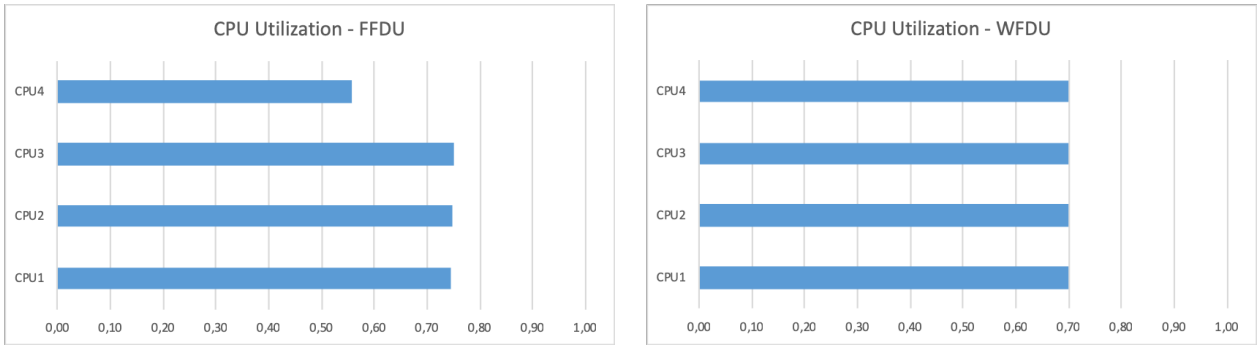


(a) **(b)**

Figure 5.29: TASK_GRAPH 1 - Test Case 9

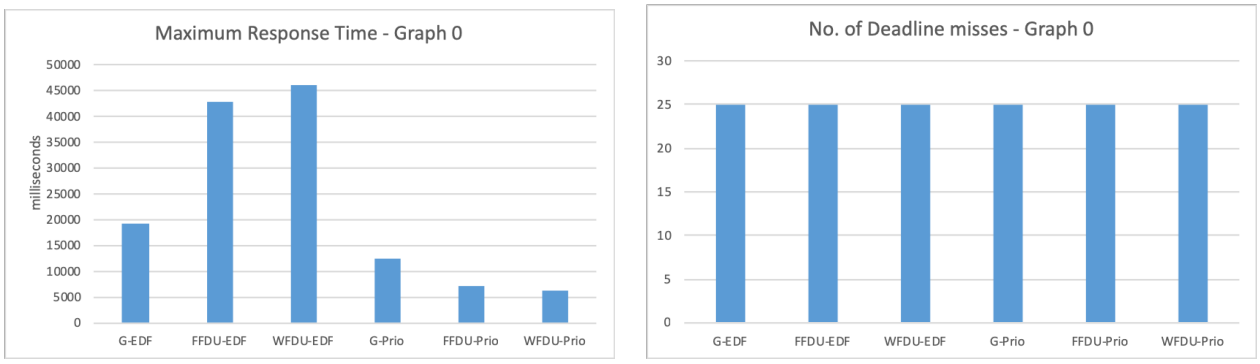
- The mapping of FFDU gives a minimum core count of 4 for scheduling the given task set with a total utilization of 2.7. Core 4 has an utilization of about 0.55.
- The mapping of WFDU balances the utilization among all the cores with the utilization of all the cores were about 0.7
- All the three variation of Priority schedulers achieves comparatively shorter average response time for TASK_GRAPH 0.
- Given a implicit deadline on TASK_GRAPH 0 with a deadline/period of 900ms, all the six schedulers were unable to meet the deadline.
- Given six deadline tasks on TASK_GRAPH 1, both the Global-EDF and Global-Priority algorithms were able to achieve shorter average response times.
- Given TASK_GRAPH 1 with period of 1320ms and six bounded deadline tasks with a deadline of 500ms, 500ms, 600ms, 600ms, 800ms and 900ms respectively, all the six schedulers were unable to meet the deadline.

Test Case 10



(a) **(b)**

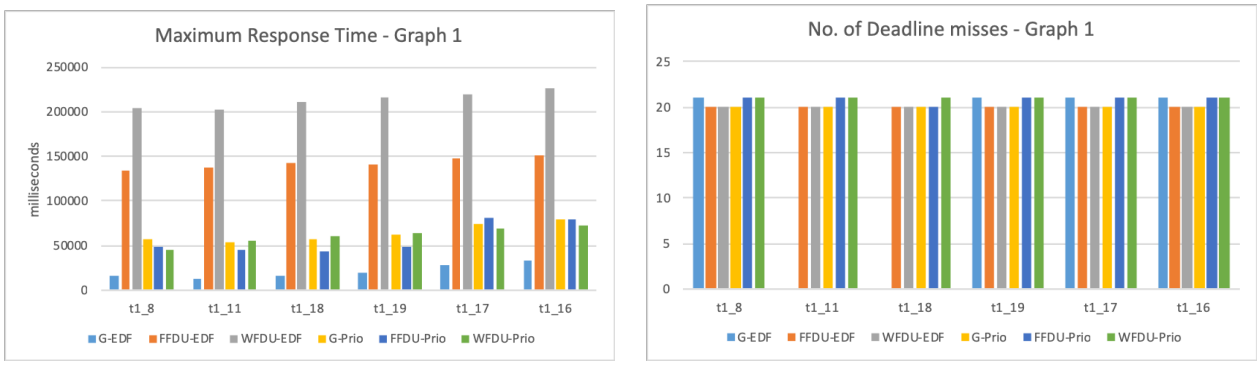
Figure 5.30: Core Utilization - Test Case 10



(a) Maximum Execution time of t0_11 **(b)**

Figure 5.31: TASK_GRAPH 0 - Test Case 10

- The mapping of FFDU gives a minimum core count of 4 for scheduling the given task set with a total utilization of 2.7. Core 4 has an utilization of about 0.55.
- The mapping of WFDU balances the utilization among all the cores with the utilization of all the cores were about 0.7
- FFDU-Priority and WFDU-Priority schedulers achieves comparatively shorter average response time for TASK_GRAPH 0.
- Given a implicit deadline on TASK_GRAPH 0 with a deadline/period of 2700ms, all the six schedulers were unable to meet the deadline.
- Given six deadline tasks on TASK_GRAPH 1, Global-EDF scheduler achieves smaller average response times.
- Given TASK_GRAPH 1 with period of 3960ms and six bounded deadline tasks with a deadline of 1500ms, 1500ms, 1800ms, 1800ms, 2400ms and 2700ms respectively, all the six schedulers were unable to meet the deadline.



(a) **(b)**

Figure 5.32: TASK_GRAPH 1 - Test Case 10

6 Conclusion

The multi-core scheduling problem has been a prevalent research topic in the field of real-time computing. Due to the increasing demand of multi-core processors for on-board processing in space missions and RTEMS's popularity in space applications, RTEMS was choosen for this study. In this thesis, the real-time performance of RTEMS SMP schedulers was experimentally analysed on an ARM Cortex A7 cluster. The schedulers were studied in both the Global and Partitioned scheduling approaches. The bin-packing problem, in assigning the threads to the available cores, in the partitioned approach were solved by using the well known FFDU and WFDU heuristics algorithms.

Work done during this thesis can be summarized in the following steps:

- Literature review on the existing Thread Mapping strategies
- Selection of two thread mapping policies
- Generation of a multi-rate sample task set for carrying out the experiment
- Implementation of RTEMS application based on the generated task set
- Evaluation of the Global scheduling approaches with the EDF and Deterministic Priority scheduling policies
- Evaluation of the two Partitioned scheduling approaches with the EDF and Deterministic Priority scheduling policies

A total of ten experiments were carried out for each of the six scheduling policies. The utilization and period of sample task set was also varied. Unlike many mathematical studies in this topic, the tasks are not considered to be independent. Task set was defined using DAGs with precedence relations. The outcome of this study is summarized below:

- FFDU heuristics achieved a tighter packing of threads on every core and eventually achieves the minimum cores needed for the given task set. For the minimum utilization of 2.1, FFDU heuristics achieved a minimum count of 3 cores. For the maximum utilization of 2.8, while using all the 4 cores, the utilization on the fourth core was 0.55
- WFDU heuristics balances the overall system load among all the available cores. For the minimum utilization of 2.1, the utilization of all the cores are balanced to a value of about 0.55. For the maximum utilization value of 2.8, the balanced utilization of all the cores were about 0.7
- For the test cases considered in this study, the Global-EDF scheduler(default RTEMS scheduler), was able to achieve better schedulabilty than the five other scheduling algorithms. However the EDF schedulers are prone to domino effect[SRS98].

- From the results it is also evident that the potential of Deterministic Priority schedulers cannot be ignored. The Global and Partitioned Deterministic priority algorithms achieved better response times for all the test cases.
- For the task sets with low utilization values, the WFDU-Priority scheduler achieved significantly smaller response times than the five other scheduling algorithms.
- None of the six schedulers were able to schedule the given task set if their utilization exceeds 2.5
- It can be clearly seen that the global scheduling approach achieved better schedulabilty than the partitioned scheduling approach. This was an expected behaviour since global schedulers are work conserving.
- Unless task migration has to be prevented, the FFDU and WFDU partitioned approach is not a suitable approach for task set with precedence constraints.

6.1 Problems encountered

Some of the challenges faced during this thesis are listed below:

- RTEMS approach in setting deadline and periods made the implementation of DAG harder, especially, if the deadlines are not equal to the period. The software overhead needed to achieve this might have a negative impact on the results.
- Debugging an SMP system is difficult and the hardware and software support for debugging is limited for Raspberry Pi2. During the early stages of this study, the Zynq 7000 SoC on a Xilinx Zedboard was used for testing and debugging purposes.

6.2 Suggestions for future research

Possible directions for future work that could enhance this study are:

- To understand thread migration in Global scheduling, some experimental thread assignment data were collected. But due to time limitations, the data was not processed during this study. Processing them will add value to this study.
- FFDU and WFDU partition schemes did not provide any significant improvements to schedulability of the given task set. So a new partitioning methodology, especially based on criticality of tasks can be considered.
- From the results it can be clearly seen that the utilization bound of 0.75 on each CPU used for partitioning is really high to acheive schedulability. Lesser CPU utilization bound can be considered for future studies.

- This study can be extended by generating different DAG structures using the new TGFF algorithms. Two of the new graph structures supported by the new algorithm are graphs with multiple start nodes and single end nodes, graphs that contain massively parallel tasks.
- In this study, for the partitioned scheduling approach, the schedulers assigned to each core are the same. A hybrid partitioning scheme with a combination of different schedulers can be studied.

List of Acronyms

OS	Operating System
SoC	System-on-Chip
RTOS	Real Time Operating System
GPOS	General Purpose Operating System
RTEMS	Real-Time Executive for Multiprocessor Systems
RSB	RTEMS Source Builder
SMP	Symmetric Multiprocessing
NP	Non-deterministic Polynomial time
ISA	Instruction Set Architecture
MMU	Memory Management Unit
PID	Proportional-Integral-Derivative
ESA	European Space Agency
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DM	Deadline Monotonic
RM	Rate Monotonic
EDF	Earliest Deadline First
DAG	Directed Acyclic Graph
ECSS	European Cooperation for Space Standardization
FFDU	First-Fit Decreasing Utilization
WFDU	Worst-Fit Decreasing Utilization
TGFF	Task Graphs For Free
CPU	Central Processing Unit
TFTP	Trivial File Transfer Protocol
POSIX	Portable Operating System Interface
NGMP	Next Generation Microprocessor
API	Application Programming Interface
AGC	Apollo Guidance Computer
RISC	Reduced Instruction Set Computer
BSP	Board Support Package
GPIO	General Purpose Input Output

Bibliography

[a2013] *Application Processors for Embedded Applications*, 2013.

[ABJ01] Andersson, Björn, Sanjoy Baruah Jan Jonsson: *Static-priority scheduling on multiprocessors. Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, 193–202. IEEE, 2001.

[AQC14] Abdallah, Nadine, Audrey Queudet Maryline Chetto: *Task partitioning strategies for multicore real-time energy harvesting systems. 2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 125–132. IEEE, 2014.

[Bar07] Baruah, Sanjoy: *Techniques for multiprocessor global schedulability analysis. 28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 119–128. IEEE, 2007.

[BBA10] Bastoni, Andrea, Bjorn B Brandenburg James H Anderson: *An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. 2010 31st IEEE Real-Time Systems Symposium*, 14–24. IEEE, 2010.

[BC07] Bertogna, Marko Michele Cirinei: *Response-time analysis for globally scheduled symmetric multiprocessor platforms. 28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 149–160. IEEE, 2007.

[BCB+08] Brandenburg, Björn B, John M Calandrino, Aaron Block, Hennadiy Leontyev James H Anderson: *Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, 342–353. IEEE, 2008.

[BCL05] Bertogna, Marko, Michele Cirinei Giuseppe Lipari: *Improved schedulability analysis of EDF on multiprocessor platforms. 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, 209–218. IEEE, 2005.

[BCL08] Bertogna, Marko, Michele Cirinei Giuseppe Lipari: *Schedulability analysis of global scheduling algorithms on multiprocessor platforms*. IEEE Transactions on parallel and distributed systems, 20(4):553–566, 2008.

[Bon16] Bonata, Luca: *RTEMS Internals Manual -how the kernel works- author Luca Bonato version v1.0*, 2016.

[But11] Buttazzo, Giorgio C.: *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd , 2011.

[ca7] *Cortex-A7*. `https://developer.arm.com/ip-products/processors/cortex-a/cortex-a7`.

[CCKF13] Chang, Che-Wei, Jian-Jia Chen, Tei-Wei Kuo Heiko Falk: *Real-time partitioned scheduling on multi-core systems with local and global memories*. *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 467–472. IEEE, 2013.

[CHS+14] Cederman, D., D. Hellström, J. Sherrill, G. Bloom, M. Patte M. Zulianello: *Rtems SMP for LEON3/LEON4 multi-processor devices*. European Space Agency, (Special Publication) ESA SP, 09 2014.

[Con19] Contributors, Wikipedia: *Embedded system*, 09 2019.

[CY12] Chishiro, Hiroyuki Nobuyuki Yamasaki: *Experimental evaluation of global and partitioned semi-fixed-priority scheduling algorithms on multicore systems*. *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 127–134. IEEE, 2012.

[DB09] Davis, Robert Alan Burns: *Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems*. 398–409, 12 2009.

[DGM14] Digalwar, Mayuri, Pravin Gahukar Sudeept Mohan: *Design and development of a real time scheduling algorithm for mixed task set on multi-core processors*. *2014 Seventh International Conference on Contemporary Computing (IC3)*, 265–269. IEEE, 2014.

[DJ06] Darera, Vivek N Lawrence Jenkins: *Utilization bounds for RM scheduling on uniform multiprocessors*. *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*, 315–321. IEEE, 2006.

[DRW98] Dick, R. P., D. L. Rhodes W. Wolf: *TGFF: task graphs for free*. *Proceedings of the Sixth International Workshop on Hardware/Software Codesign. (CODES/CASHE'98)*, 97–101, March 1998.

[DS14] Dario Socci, Peter Poplavko, Saddek Bensalem Marius Bozga: *Multiprocessor Scheduling of Precedence-constrained Mixed-Critical Jobs*. TR-2014-11, Verimag Research Report, 2014.

[GSHT13] Giannopoulou, Georgia, Nikolay Stoimenov, Pengcheng Huang Lothar Thiele: *Scheduling of mixed-criticality applications on resource-sharing multicore systems*. *Proceedings of the Eleventh ACM International Conference on Embedded Software*, 17. IEEE Press, 2013.

[HZW+13] Han, Jian-Jun, Dakai Zhu, Xiaodong Wu, Laurence T Yang Hai Jin: *Multiprocessor real-time systems with shared resources: Utilization bound and mapping*. IEEE Transactions on Parallel and Distributed Systems, 25(11):2981–2991, 2013.

[Joe19] Joel Sherrill, Gedare Bloom: *Scheduling and Thread Management with RTEMS*. URL: http://www.rtems.com/sites/default/files/EWiLi-RTEMS-JoelSherrill-2013.pdf, 12 2019.

[kel19] *www.militaryaerospace.com*, 2019.

[KKLR11] Kandhalu, Arvind, Junsung Kim, Karthik Lakshmanan Ragunathan Rajkumar: *Energy-aware partitioned fixed-priority scheduling for chip multi-processors*. *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, 1, 93–102. IEEE, 2011.

[KYD11] Kong, Fanxin, Wang Yi Qingxu Deng: *Energy-efficient scheduling of real-time tasks on cluster-based multicores*. *2011 Design, Automation & Test in Europe*, 1–6. IEEE, 2011.

[l4d] *GR-CPCI-GR740 Quad-Core LEON4FT Development Board*. https://www.gaisler.com/index.php/products/boards/gr-cpci-gr740.

[LA10] Leontyev, Hennadiy James H Anderson: *Generalized tardiness bounds for global multiprocessor scheduling*. Real-Time Systems, 44(1-3):26–71, 2010.

[leo] *RTEMS-SMP Improvement for LEON multi-core*. https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Microprocessors. Accessed: 2019-12-21.

[LKR10] Lakshmanan, Karthik, Shinpei Kato Ragunathan Rajkumar: *Scheduling parallel real-time tasks on multi-core processors*. *2010 31st IEEE Real-Time Systems Symposium*, 259–268. IEEE, 2010.

[LQ11] Lu, Jun Qinru Qiu: *Scheduling and mapping of periodic tasks on multi-core embedded systems with energy harvesting*. *2011 International Green Computing Conference and Workshops*, 1–6. IEEE, 2011.

[LZLQ07] Liu, Yi, Xin Zhang, He Li Depei Qian: *Allocating tasks in multi-core processor based parallel system*. *2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007)*, 748–753. IEEE, 2007.

[NE12] Negrean, Mircea Rolf Ernst: *Response-time analysis for non-preemptive scheduling in multi-core systems with shared resources*. *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, 191–200. IEEE, 2012.

[NKN08] Nemati, Farhang, Johan Kraft Thomas Nolte: *Towards migrating legacy real-time systems to multi-core platforms*. *2008 IEEE International Conference on Emerging Technologies and Factory Automation*, 717–720. IEEE, 2008.

[NN10] Nedu, Vaidehi T.R. Nair: *Multicore Applications in Real Time Systems*. 01 2010.

[Pro19] Project, RTEMS: *RTEMS Classic API Guide 5.d954241*, 2019.

[ras] *Raspberry Pi 2 Model B*. https://www.raspberrypi.org/products/raspberry-pi-2-model-b/.

[rte] *RTEMS Classic API Guide*. `https://docs.rtems.org/branches/master/c-user/overview.html`.

[SEGY11] Stigge, M., P. Ekberg, N. Guan W. Yi: *The Digraph Real-Time Task Model*. *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 71–80, April 2011.

[SHG] Själander, Magnus, Sandi Habinc Jiri Gaisler: *LEON4: Fourth Generation of the LEON Processor*.

[SJPL08] Seo, Euiseong, Jinkyu Jeong, Seonyeong Park Joonwon Lee: *Energy Efficient Scheduling of Real-Time Tasks on Multicore Processors*. IEEE Transactions on Parallel and Distributed Systems, 19:1540–1552, 11 2008.

[SRS98] Stankovic, John A., Krithi Ramamritham Marco Spuri: *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[SSF+09] Silva, Helder, José Sousa, Daniel Freitas, Sergio Faustino, Alexandre Constantino Manuel Coutinho: *RTEMS improvement-space qualification of RTEMS executive*. 1st Simpósio de Informática-INFORUM, University of Lisbon, 2009.

[SSRM12] Shekhar, Mayank, Abhik Sarkar, Harini Ramaprasad Frank Mueller: *Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems*. *2012 24th Euromicro Conference on Real-Time Systems*, 331–340. IEEE, 2012.

[Tan09] Tan, Pengliu: *Task scheduling of real-time systems on multi-core architectures*. *2009 Second International Symposium on Electronic Commerce and Security*, 2, 190–193. IEEE, 2009.

[Ull75] Ullman, Jeffrey D.: *NP-complete scheduling problems*. Journal of Computer and System sciences, 10(3):384–393, 1975.

[Ver] Verhoef, Marcel: *RTEMS SMP Qualification*. `http://microelectronics.esa.int/gr740/rtems-smp-02122018-FSC_v3.pdf`.

Annex

Listing 1: timing.h

```
// Configure Test Number
#define TEST_13


/// Common value
#define T0_0_EXE_MODIFY 2
#define T1_0_EXE_MODIFY 1
///


#if defined(TEST_1)

#define GR00_PERIOD 900.0
#define GR01_PERIOD 1320.0

#define D0_MODIFY 1
#define D1_MODIFY 1

#define d0_0_DEADLINE 100*D0_MODIFY
#define d0_1_DEADLINE 200*D0_MODIFY
#define d0_2_DEADLINE 200*D0_MODIFY
#define d0_3_DEADLINE 300*D0_MODIFY
#define d0_4_DEADLINE 400*D0_MODIFY
#define d0_5_DEADLINE 500*D0_MODIFY
#define d0_6_DEADLINE 500*D0_MODIFY
#define d0_7_DEADLINE 600*D0_MODIFY
#define d0_8_DEADLINE 700*D0_MODIFY
#define d0_9_DEADLINE 800*D0_MODIFY
#define d0_10_DEADLINE 800*D0_MODIFY
#define d0_11_DEADLINE 900*D0_MODIFY

#define d1_0_DEADLINE 100*D1_MODIFY
#define d1_1_DEADLINE 200*D1_MODIFY
#define d1_2_DEADLINE 300*D1_MODIFY
#define d1_3_DEADLINE 300*D1_MODIFY
```

```
#define d1_4_DEADLINE 400*D1_MODIFY
#define d1_5_DEADLINE 400*D1_MODIFY
#define d1_6_DEADLINE 400*D1_MODIFY
#define d1_7_DEADLINE 500*D1_MODIFY
#define d1_8_DEADLINE 500*D1_MODIFY
#define d1_9_DEADLINE 500*D1_MODIFY
#define d1_10_DEADLINE 600*D1_MODIFY
#define d1_11_DEADLINE 500*D1_MODIFY
#define d1_12_DEADLINE 500*D1_MODIFY
#define d1_13_DEADLINE 700*D1_MODIFY
#define d1_14_DEADLINE 700*D1_MODIFY
#define d1_15_DEADLINE 800*D1_MODIFY
#define d1_16_DEADLINE 900*D1_MODIFY
#define d1_17_DEADLINE 800*D1_MODIFY
#define d1_18_DEADLINE 600*D1_MODIFY
#define d1_19_DEADLINE 600*D1_MODIFY

#define t0_5_EXE_TIME 38.0
#define t0_4_EXE_TIME 17.0
#define t0_7_EXE_TIME 16.0
#define t0_6_EXE_TIME 47.0
#define t0_1_EXE_TIME 99.0
#define t0_0_EXE_TIME 90.0-T0_0_EXE_MODIFY
#define t0_3_EXE_TIME 8.0
#define t0_2_EXE_TIME 10.0
#define t0_9_EXE_TIME 389.0
#define t0_8_EXE_TIME 241.0
#define t1_8_EXE_TIME 17.0
#define t1_9_EXE_TIME 66.0
#define t1_4_EXE_TIME 18.0
#define t1_5_EXE_TIME 86.0
#define t1_6_EXE_TIME 66.0
#define t1_7_EXE_TIME 28.0
#define t1_0_EXE_TIME 28.0-T1_0_EXE_MODIFY
#define t1_1_EXE_TIME 94.0
#define t1_2_EXE_TIME 122.0
#define t1_3_EXE_TIME 155.0
#define t0_11_EXE_TIME 49.0
#define t0_10_EXE_TIME 166.0
#define t1_18_EXE_TIME 16.0
#define t1_19_EXE_TIME 15.0
#define t1_16_EXE_TIME 1.0
#define t1_17_EXE_TIME 98.0
#define t1_14_EXE_TIME 82.0
#define t1_15_EXE_TIME 25.0
```

```
#define t1_12_EXE_TIME 95.0
#define t1_13_EXE_TIME 38.0
#define t1_10_EXE_TIME 2.0
#define t1_11_EXE_TIME 3.0

#endif

#if defined(TEST_2)

#define GR00_PERIOD 2700.0
#define GR01_PERIOD 5280.0

#define D0_MODIFY 3
#define D1_MODIFY 4

#define d0_0_DEADLINE 100*D0_MODIFY
#define d0_1_DEADLINE 200*D0_MODIFY
#define d0_2_DEADLINE 200*D0_MODIFY
#define d0_3_DEADLINE 300*D0_MODIFY
#define d0_4_DEADLINE 400*D0_MODIFY
#define d0_5_DEADLINE 500*D0_MODIFY
#define d0_6_DEADLINE 500*D0_MODIFY
#define d0_7_DEADLINE 600*D0_MODIFY
#define d0_8_DEADLINE 700*D0_MODIFY
#define d0_9_DEADLINE 800*D0_MODIFY
#define d0_10_DEADLINE 800*D0_MODIFY
#define d0_11_DEADLINE 900*D0_MODIFY

#define d1_0_DEADLINE 100*D1_MODIFY
#define d1_1_DEADLINE 200*D1_MODIFY
#define d1_2_DEADLINE 300*D1_MODIFY
#define d1_3_DEADLINE 300*D1_MODIFY
#define d1_4_DEADLINE 400*D1_MODIFY
#define d1_5_DEADLINE 400*D1_MODIFY
#define d1_6_DEADLINE 400*D1_MODIFY
#define d1_7_DEADLINE 500*D1_MODIFY
#define d1_8_DEADLINE 500*D1_MODIFY
#define d1_9_DEADLINE 500*D1_MODIFY
#define d1_10_DEADLINE 600*D1_MODIFY
#define d1_11_DEADLINE 500*D1_MODIFY
#define d1_12_DEADLINE 500*D1_MODIFY
#define d1_13_DEADLINE 700*D1_MODIFY
#define d1_14_DEADLINE 700*D1_MODIFY
#define d1_15_DEADLINE 800*D1_MODIFY
#define d1_16_DEADLINE 900*D1_MODIFY
```

```
#define d1_17_DEADLINE 800*D1_MODIFY
#define d1_18_DEADLINE 600*D1_MODIFY
#define d1_19_DEADLINE 600*D1_MODIFY

#define t0_5_EXE_TIME 16.0
#define t0_4_EXE_TIME 803.0
#define t0_7_EXE_TIME 588.0
#define t0_6_EXE_TIME 42.0
#define t0_1_EXE_TIME 77.0
#define t0_0_EXE_TIME 110.0-T0_0_EXE_MODIFY
#define t0_3_EXE_TIME 334.0
#define t0_2_EXE_TIME 147.0
#define t0_9_EXE_TIME 164.0
#define t0_8_EXE_TIME 203.0
#define t1_8_EXE_TIME 748.0
#define t1_9_EXE_TIME 653.0
#define t1_4_EXE_TIME 157.0
#define t1_5_EXE_TIME 20.0
#define t1_6_EXE_TIME 122.0
#define t1_7_EXE_TIME 118.0
#define t1_0_EXE_TIME 291.0-T1_0_EXE_MODIFY
#define t1_1_EXE_TIME 477.0
#define t1_2_EXE_TIME 83.0
#define t1_3_EXE_TIME 110.0
#define t0_11_EXE_TIME 618.0
#define t0_10_EXE_TIME 408.0
#define t1_18_EXE_TIME 125.0
#define t1_19_EXE_TIME 12.0
#define t1_16_EXE_TIME 194.0
#define t1_17_EXE_TIME 187.0
#define t1_14_EXE_TIME 329.0
#define t1_15_EXE_TIME 174.0
#define t1_12_EXE_TIME 69.0
#define t1_13_EXE_TIME 204.0
#define t1_10_EXE_TIME 150.0
#define t1_11_EXE_TIME 1.0

#endif
```

Listing 2: ffdu.h

```
#if defined(TEST_1)

x->status = rtems_task_set_scheduler(x->id0[9], x->schd[0],T0_9_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[8], x->schd[0],T0_8_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[5], x->schd[0],T0_5_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[11], x->schd[0],T1_11_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[10], x->schd[0],T1_10_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[16], x->schd[0],T1_16_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[10], x->schd[1],T0_10_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[3], x->schd[1],T1_3_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[1], x->schd[1],T0_1_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[0], x->schd[1],T0_0_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[2], x->schd[1],T1_2_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[17], x->schd[1],T1_17_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[1], x->schd[1],T1_1_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[12], x->schd[2],T1_12_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[5], x->schd[2],T1_5_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[14], x->schd[2],T1_14_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[11], x->schd[2],T0_11_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[6], x->schd[2],T0_6_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[9], x->schd[2],T1_9_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[6], x->schd[2],T1_6_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
```

```
x->status = rtems_task_set_scheduler(x->id[13], x->schd[2],T1_13_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[7], x->schd[2],T1_7_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[0], x->schd[2],T1_0_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[15], x->schd[2],T1_15_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[4], x->schd[2],T0_4_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[7], x->schd[2],T0_7_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[4], x->schd[2],T1_4_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[8], x->schd[2],T1_8_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[18], x->schd[2],T1_18_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[19], x->schd[2],T1_19_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[2], x->schd[2],T0_2_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[3], x->schd[2],T0_3_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );

#endif


#if defined(TEST_2)
x->status = rtems_task_set_scheduler(x->id0[4], x->schd[0],T0_4_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[11], x->schd[0],T0_11_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[7], x->schd[0],T0_7_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[5], x->schd[0],T0_5_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[10], x->schd[1],T0_10_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[8], x->schd[1],T1_8_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[3], x->schd[1],T0_3_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[9], x->schd[1],T1_9_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
```

```
x->status = rtems_task_set_scheduler(x->id[1], x->schd[1],T1_1_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[8], x->schd[1],T0_8_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[0], x->schd[1],T0_0_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[19], x->schd[1],T1_19_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[11], x->schd[1],T1_11_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[14], x->schd[2],T1_14_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[9], x->schd[2],T0_9_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[0], x->schd[2],T1_0_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[2], x->schd[2],T0_2_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[13], x->schd[2],T1_13_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[16], x->schd[2],T1_16_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[17], x->schd[2],T1_17_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[15], x->schd[2],T1_15_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[4], x->schd[2],T1_4_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[1], x->schd[2],T0_1_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[10], x->schd[2],T1_10_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[18], x->schd[2],T1_18_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[6], x->schd[2],T1_6_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[7], x->schd[2],T1_7_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[3], x->schd[2],T1_3_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[2], x->schd[2],T1_2_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[6], x->schd[2],T0_6_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[12], x->schd[2],T1_12_PRIO);
```

```
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[5], x->schd[2],T1_5_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );

#endif
```

Listing 3: wfdu.h

```
#if defined(TEST_1)

x->status = rtems_task_set_scheduler(x->id0[9], x->schd[0],T0_9_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[6], x->schd[0],T1_6_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[0], x->schd[0],T1_0_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[4], x->schd[0],T1_4_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[2], x->schd[0],T0_2_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[8], x->schd[1],T0_8_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[17], x->schd[1],T1_17_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[5], x->schd[1],T1_5_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[6], x->schd[1],T0_6_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[13], x->schd[1],T1_13_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[4], x->schd[1],T0_4_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[18], x->schd[1],T1_18_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[11], x->schd[1],T1_11_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[10], x->schd[2],T0_10_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[0], x->schd[2],T0_0_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[12], x->schd[2],T1_12_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[14], x->schd[2],T1_14_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[9], x->schd[2],T1_9_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[7], x->schd[2],T1_7_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[7], x->schd[2],T0_7_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
```

```
x->status = rtems_task_set_scheduler(x->id[19], x->schd[2],T1_19_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[3], x->schd[2],T0_3_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[3], x->schd[3],T1_3_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[1], x->schd[3],T0_1_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[2], x->schd[3],T1_2_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[1], x->schd[3],T1_1_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[11], x->schd[3],T0_11_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[5], x->schd[3],T0_5_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[15], x->schd[3],T1_15_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[8], x->schd[3],T1_8_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[10], x->schd[3],T1_10_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[16], x->schd[3],T1_16_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );

#endif


#if defined(TEST_2)
x->status = rtems_task_set_scheduler(x->id0[4], x->schd[0],T0_4_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[8], x->schd[0],T0_8_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[0], x->schd[0],T1_0_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[16], x->schd[0],T1_16_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[1], x->schd[0],T0_1_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[7], x->schd[0],T1_7_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[12], x->schd[0],T1_12_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[11], x->schd[1],T0_11_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
```

```
x->status = rtems_task_set_scheduler(x->id[9], x->schd[1],T1_9_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[9], x->schd[1],T0_9_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[13], x->schd[1],T1_13_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[4], x->schd[1],T1_4_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[6], x->schd[1],T1_6_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[6], x->schd[1],T0_6_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[5], x->schd[1],T1_5_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[7], x->schd[2],T0_7_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[3], x->schd[2],T0_3_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[14], x->schd[2],T1_14_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[0], x->schd[2],T0_0_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[15], x->schd[2],T1_15_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[18], x->schd[2],T1_18_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[3], x->schd[2],T1_3_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[19], x->schd[2],T1_19_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[10], x->schd[3],T0_10_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[8], x->schd[3],T1_8_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[1], x->schd[3],T1_1_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[2], x->schd[3],T0_2_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[17], x->schd[3],T1_17_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[10], x->schd[3],T1_10_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[2], x->schd[3],T1_2_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id0[5], x->schd[3],T0_5_PRIO);
```

```
assert( x->status == RTEMS_SUCCESSFUL );
x->status = rtems_task_set_scheduler(x->id[11], x->schd[3],T1_11_PRIO);
assert( x->status == RTEMS_SUCCESSFUL );

#endif
```

Listing 4: system.h

```
/* configuration information */
#include <bsp.h>

#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_MICROSECONDS_PER_TICK    1000 /* 1 millisecond */
#define CONFIGURE_MAXIMUM_PROCESSORS 4
#define CONFIGURE_MAXIMUM_PERIODS 50 // Max. Classic API Rate Monotonic P
#define CONFIGURE_MAXIMUM_TASKS 50
#define CONFIGURE_MAXIMUM_SEMAPHORES 50
#define CONFIGURE_MAXIMUM_BARRIERS 40

#define CONFIGURE_RTEMS_INIT_TASKS_TABLE

#define CONFIGURE_MAXIMUM_TIMERS 50

// Global Schedulers
//#define CONFIGURE_SCHEDULER_EDF_SMP //(default)
//#define CONFIGURE_SCHEDULER_PRIORITY_AFFINITY_SMP
//#define CONFIGURE_SCHEDULER_PRIORITY_SMP
//#define CONFIGURE_SCHEDULER_SIMPLE_SMP

/* Partitioned Scheduling

//STEP1
#define CONFIGURE_SCHEDULER_EDF_SMP
#include <rtems/scheduler.h>

//STEP2
RTEMS_SCHEDULER_EDF_SMP(sh0,4);
RTEMS_SCHEDULER_EDF_SMP(sh1,4);
RTEMS_SCHEDULER_EDF_SMP(sh2,4);
RTEMS_SCHEDULER_EDF_SMP(sh3,4);

//STEP3
#define CONFIGURE_SCHEDULER_TABLE_ENTRIES
```

```
RTEMS_SCHEDULER_TABLE_EDF_SMP(sh0,rtems_build_name('S', 'C', 'H', '0')),
RTEMS_SCHEDULER_TABLE_EDF_SMP(sh2,rtems_build_name('S', 'C', 'H', '2')),
RTEMS_SCHEDULER_TABLE_EDF_SMP(sh3,rtems_build_name('S', 'C', 'H', '3'))
//STEP4
#define CONFIGURE_SCHEDULER_ASSIGNMENTS
RTEMS_SCHEDULER_ASSIGN(0, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
RTEMS_SCHEDULER_ASSIGN_NO_SCHEDULER,
RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
RTEMS_SCHEDULER_ASSIGN(2, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY)

*/

#include <rtems/confdefs.h>
```

Listing 5: task.cc

```
#include "graph.h"

void spin0_0(rtems_interval ticks){
        rtems_interval start;
        rtems_interval now;
        start = rtems_clock_get_ticks_since_boot();
        do {
                now = rtems_clock_get_ticks_since_boot();
        }while(now-start < ticks );
}

void t0_0_init(gr01_context *x){
        // RM Object for Periodic timer
        x->status = rtems_rate_monotonic_create(rtems_build_name('
␣␣␣␣␣␣␣␣p', '0','0','0'),&x->period0);
        assert( x->status == RTEMS_SUCCESSFUL );

        // RM Object for Deadline timer
        x->status = rtems_rate_monotonic_create(rtems_build_name('
␣␣␣␣␣␣␣␣r', '0','0','0'),&x->rm0[0]);
        assert( x->status == RTEMS_SUCCESSFUL );

        x->status = rtems_semaphore_create(rtems_build_name('a',
        '0','0','0'),0,RTEMS_COUNTING_SEMAPHORE | RTEMS_PRIORITY,0,
        &x->sem0[0]);
        assert( x->status == RTEMS_SUCCESSFUL );
        x->status = rtems_semaphore_create(rtems_build_name('a',
        '0','0','1'),0,RTEMS_COUNTING_SEMAPHORE | RTEMS_PRIORITY,0,
```

```
        &x->sem0[1]);
        assert( x->status == RTEMS_SUCCESSFUL );

}

void t0_0_work(gr01_context *x){
        printf("t0_0 on CPU: %d \n", rtems_scheduler_get_processor());
        spin0_0(RTEMS_MILLISECONDS_TO_TICKS(t0_0_EXE_TIME));
        x->counter0[0] = x->counter0[0]+1;
        rtems_semaphore_release(x->sem0[0]);
        assert( x->status == RTEMS_SUCCESSFUL );
        rtems_semaphore_release(x->sem0[1]);
        assert( x->status == RTEMS_SUCCESSFUL );
}

rtems_task t0_0(rtems_task_argument argument){
        gr01_context *x = (gr01_context*) argument;
        t0_0_init(x);
        printf("t0_0: Init completed \n");

        x->status = rtems_barrier_wait(x->b0,RTEMS_WAIT);
        assert( x->status == RTEMS_SUCCESSFUL );
        while(1){
                while(1){
//                        if ( rtems_rate_monotonic_period( x->period0,
RTEMS_MILLISECONDS_TO_TICKS(GR00_PERIOD)) == RTEMS_TIMEOUT )
//                                break;
                        rtems_rate_monotonic_period( x->period0, RTEMS_MI
                        // GPIO 6 ON

                        // Width
                        x->status =  rtems_task_wake_after(RTEMS_MIL
                        LISECONDS_TO_TICKS(T0_0_EXE_MODIFY));
                        assert( x->status == RTEMS_SUCCESSFUL );

                        // GPIO 6 OFF

                        x->status = rtems_barrier_wait(x->c0,RTEMS_WAIT);
                        assert( x->status == RTEMS_SUCCESSFUL );

//                        if ( rtems_rate_monotonic_period( x->rm0[0],
RTEMS_MILLISECONDS_TO_TICKS(d0_0_DEADLINE))
== RTEMS_TIMEOUT )
//                                break;
                        rtems_rate_monotonic_period( x->rm0[0],RTEMS_MILLI
```

```
                        x->status = rtems_barrier_wait(x->d0,RTEMS_WAIT);
                        assert( x->status == RTEMS_SUCCESSFUL );

                        t0_0_work(x);
                        x->status = rtems_rate_monotonic_cancel( x->rm0[0
                        assert( x->status == RTEMS_SUCCESSFUL );
                }
//              printf("t0_0:missed deadline or an iteration \n");
        }
        rtems_task_exit();
}
```