COMPARISON OF BUDGET BORROWING AND BUDGET ADAPTATION IN HIERARCHICAL SCHEDULING FRAMEWORK

Wenkai Wang
wwg13001@student.mdh.se

Examiner: Moris Behnam
Mälardalen University, Västerås, Sweden

Supervisor: Nima Khalilzad
Mälardalen University, Västerås, Sweden

January 13, 2016
Abstract

System virtualization technology is widely used in computing nowadays. In embedded domain, it is used as a solution to resource sharing among independent applications. One of the areas is to apply virtualization technique to real-time embedded systems with timing constraints. Multi-level adaptive hierarchical scheduling (AdHierSched) framework is a virtualized real-time framework, which runs in the Linux operating system. This virtualized framework has ability to adapt the CPU partition sizes according to their need through monitoring their demand during run-time, which yields more appropriate processor assignment. However, the performance of the virtualized framework is still unknown when the budget borrowing mechanism is enabled. To this end, in this thesis, we explore a new direction for performing the adaptation of CPU partition. We design and implement a budget borrowing mechanism for dynamic adaptation of resource parameters in AdHierSched framework. Extensive simulations are performed in this thesis, which are used to study and compare different adaptation mechanisms with our approach. From the results of experiments, we conclude that when the framework works only with budget borrowing controller, the results are not as good as only running a budget controller in the AdHierSched framework. However, while running both of the controllers at the same time, the experiments results are good enough. We also analyze the overhead of the framework at the end of the evaluation. Finally, we conclude the thesis by presenting the possible future work.

Key words: system virtualization, CPU partition, hierarchical scheduling, monitoring, budget borrowing.
Acknowledgements

First of all, a big thanks to my supervisor Nima Khalilzad for supporting and believing in me throughout the thesis process. I would like to thank my examiner Dr. Moris Behnam for his precious help and advice. I would also like to thank Meng Liu and Yong Du, who encourage me to keep on going and give me a lot of help during this time. A big thanks to Hang Yin who helps me improve the presentation of the thesis. Finally and most importantly, I would like to thank my parents and little brother for supporting me to study in Sweden.
# Table of Contents

1 Introduction ................................................. 6  
   1.1 Problem formulation .................................. 7  
   1.2 Thesis contributions .................................. 8  
   1.3 The structure of the thesis ............................ 8

2 Theoretical Background .................................... 9  
   2.1 Real-time systems ...................................... 9  
   2.2 Processor partition technology ......................... 9  
   2.3 CPU reservation technology ............................ 10  
   2.4 Adaptation mechanism .................................. 10  
   2.5 Server-based scheduling ................................ 10

3 The AdHierSched Framework ............................... 11  
   3.1 Application model ...................................... 11  
   3.2 Server model ........................................... 11  
   3.3 Task model ............................................. 11  
   3.4 System model ........................................... 11  
   3.5 Adaptation model ....................................... 11  
   3.6 A Linux kernel loadable model ......................... 13

4 Research Methodology ..................................... 14

5 Design ....................................................... 15  
   5.1 Overall view of the AdHierSched framework ............ 15  
   5.2 Server budget borrowing design ......................... 16  
      5.2.1 Server budget borrowing ............................ 16  
      5.2.2 Time to borrow budget .............................. 16  
      5.2.3 How much budget need to borrow ................... 17  
      5.2.4 Dealing with overload situations .................. 18

6 Evaluation .................................................. 19  
   6.1 Tasks .................................................. 19  
   6.2 Evaluation setup ....................................... 19  
   6.3 Evaluation conditions .................................. 19  
   6.4 Static tasks ............................................ 19  
      6.4.1 Case study 1 ....................................... 19  
      6.4.2 Case study 2 ....................................... 20  
   6.5 Dynamic tasks ......................................... 22  
      6.5.1 Case study 1 ....................................... 22  
      6.5.2 Case study 2 ....................................... 24  
   6.6 Overhead ............................................... 25

7 Related Work ............................................... 27

8 Summary and Conclusions ................................ 29

9 Future Work ................................................. 30

References ...................................................... 33

Appendix A Knowledge about Linux operation system .... 34  
   A.1 Jiffies .................................................. 34  
   A.2 Recompile the linux kernel ............................. 34
Appendix B  Random distributions  
B.1 Normal distribution .............................................. 35
B.2 Poisson distribution ............................................... 35
B.3 Exponential distribution ......................................... 35

Appendix C  Part of the implementation  
C.1 Stop Server Function ........................................... 36
C.2 Budget borrowing design flowchart ............................ 36

Appendix D  Part code implementation in this thesis  
D.1 Borrow budget function code .................................. 39
D.2 Redesign the stop server function ............................. 39
D.3 C# code that used to generate dynamic tasks Part A .......... 41
D.4 C# code that used to generate dynamic tasks Part B .......... 43
List of Figures

1. Diagram of a RTOS and GPOS running on top of a hypervisor [23]. . . . . . . . . . . . . . 6
2. Two-level hierarchical scheduling framework [7] . . . . . . . . . . . . . . . . . . . . . . . 7
3. CPU partition in real-time operating system[29] . . . . . . . . . . . . . . . . . . . . . . . 9
5. An example of control period [25] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
6. AdHierSched module [24] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
7. The research methodology flowchart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
8. Overall view of the AdHierSched framework . . . . . . . . . . . . . . . . . . . . . . . . . 15
9. Server borrow budget example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
10. When need to borrow budget . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
11. Task $\tau_j^1$ deadline miss ration compare . . . . . . . . . . . . . . . . . . . . . . . . . . 20
12. Run one static task under one server experiment . . . . . . . . . . . . . . . . . . . . . . . . 21
13. The structure of the case-study system with multi-tasks . . . . . . . . . . . . . . . . . . . . 22
14. The multiple static tasks running under different servers experiments results . . . . . . . 23
15. The distribution of execution time in task $\tau_0$ . . . . . . . . . . . . . . . . . . . . . . . . 23
16. The deadline miss ratio of dynamic task . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
17. Server budget adaptation during run-time . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
18. The dynamic tasks DMR when framework working under different conditions . . . . . . 25
19. The overhead in section 6.5 case study 2, when the framework works under condition 3 . 26
20. Setup the operating system HZ value . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
21. Normal distribution bell curve . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
22. Exponential distribution probability density curve . . . . . . . . . . . . . . . . . . . . . . . 36
23. Stop server design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
24. Borrow budget design flowchart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

List of Tables

1. The specification of $\tau_j^1$ and $S_j^1$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2. The specification of servers and tasks in the case study . . . . . . . . . . . . . . . . . . . . 22
3. The specification of server and dynamic task in case-study 1 . . . . . . . . . . . . . . . . . 22
4. The specification of servers and dynamic tasks in case study 2 . . . . . . . . . . . . . . . . 25
1 Introduction

Over the last decades, system virtualization has been widely used in real-time embedded systems to solve the stringent timing constraints problems [20]. The original purpose of using embedded virtualization technique was to create an environment for the real-time operating systems (RTOSs) which worked alongside other operating systems such as Linux or Windows on a single machine [19]. By using a software layer which is known as hypervisor, multiple operating systems can run on a single hardware platform [23]. Figure 1 is a simple example which shows a RTOS and general purpose operating system running on top of a hypervisor. The advantages of using embedded virtualization technology are clear. It not only increases the equipment utilization but also lowers capital expenditures. It also improves the security of the system.

A number of virtualized embedded systems or frameworks have been developed in the embedded system domain. One of them is the Multi-level adaptive Hierarchical Scheduling (AdHierSched) framework, which uses hierarchical scheduling technical and has the ability to adapt the processor partition sizes of soft real-time systems based on their need, which means monitoring their demand during run-time [25]. Hierarchical scheduling is a very popular research area in real-time systems. Hierarchical scheduling with the processor partitioning mainly divides the CPU into a certain mount of partitions and then assigns each partition to a subsystem. So different subsystems are temporally isolated. The minimum supply of the CPU resource to each partition can be derived.

Hierarchical Scheduling Framework (HSF) is hierarchically divided into different subsystems. A global (system level) scheduler is used to schedule all the subsystems. There is a local (subsystem level) scheduler which is used to schedule all the tasks or/and sub-subsystems in each subsystem. Figure 2 shows a two level hierarchical scheduling system. Each subsystem is assigned a fraction of the total processor time by the global scheduler. Each subsystem tries to use the assign processor time properly according to the local scheduler. Hierarchical scheduling technique has been used to deal with the increasing complexity of real-time systems for decades, which has the following advantages as follow:

1) Isolation
   CPU partitioning is usually used in hierarchical scheduling. By using this technique, the total system CPU bandwidth is divided into a certain amount of partitions. While each partition is assigned to a subsystem (A subsystem usually includes tasks or/and sub-subsystems). Since different subsystems are temporally isolated. The minimum supply of the CPU resource to each partition can be derived.

2) Decomposition
   The complex composite scheduling behaviours in hierarchical scheduling framework can be divided...
into small, simple schedulers. Thereby increasing the flexibility compared with a single global scheduler.

Because of these advantages, many hierarchical scheduling frameworks [13, 31, 12, 46, 27] have been developed since a two level hierarchical scheduling framework [13] was introduced to the open environment. Different resource management mechanisms are used in these frameworks. For example in [8], the authors implemented a hierarchical scheduling framework in a commercial operating system VxWorks, which demonstrates that it is feasible to implement a hierarchical scheduling framework in a commercial operating system without modifying the kernel source code. In [27], in order to address the low CPU utilization and numerous task deadline miss problems in dynamic systems, the authors propose an adaptive CPU allocation method which assigns CPU portions to subsystems according to their current CPU need. They also introduce the notion of subsystem criticality when the system works under overload situation.

However, the performance of hierarchical scheduling framework is still unknown after enabling the budget borrowing adaptive mechanism. Budget borrowing mechanism is a method that the server borrows budget from its future if the running tasks in the server cannot finish execution before their deadlines. This is the focus of our thesis.

### 1.1 Problem formulation

The resource demand of real-time systems may vary throughout the run-time. For instance, the processor demand of video decoder tasks may change significantly depending on the content of the video frames. To deal with this property, the resource adaptation mechanism is a main approach. The aim of this thesis project is to find answers to the questions as below:

Since there are many different real-time frameworks which use budget adaptation mechanism to manage their resource. Several of them have already used budget borrow mechanism. How should we design our own budget borrow controller in AdHierSched framework?

There are many real-time tasks whose worst-case execution times are unknown a priori. If a server wants to borrow budget, there should be an algorithm which is used to estimate the amount of budget that needs to borrow. What is the suitable algorithm we should choose to estimate the amount of budget that needs to borrow?

Through comparing the performance of the budget borrowing mechanism with budget adapting mechanism by running different case-studies in a virtualized real-time system. Which one is better?
1.2 Thesis contributions

The goal of the thesis is to design and implement a budget borrowing adaptation mechanism in AdHierSched framework, and then compare it with another adaptation mechanism by running different case-studies.

In order to achieve our main goal, we make the following contributions in this thesis:

- We explore a new direction for performing the adaptation in hierarchical scheduling, which is budget borrowing adaptation mechanism. We design and implement a budget borrowing controller which is used to adapt the budget of the framework during run-time.
- Since some real-time tasks’ Worst Case Execution Times (WCET) are unknown a priori, it is very difficult to decide how much budget to borrow when the framework needs to borrow budget from the future. In order to solve this problem, we propose an algorithm which is used to estimate the amount of server budget that needs to be borrowed in a server period.
- We evaluate the performance of budget borrowing mechanism and compare it with budget adaptation mechanism in AdHierSched framework by using several case-studies. We use both static tasks and dynamic tasks in this thesis.
- We measure the extra calculation overhead of the budget borrowing mechanism in the framework, and compare it with the total overhead from the framework.

1.3 The structure of the thesis

This section is to give a clear overview to the potential readers about the structure of this thesis report. The rest of the thesis is organized as follow:

- Section 2 explains some background knowledge which is used in the AdHierSched framework such as CPU partition technique, adaptation mechanism, server-based scheduling and hierarchical real-time systems.
- Section 3 presents some basic theories about the AdHierSched framework.
- Section 4 presents the research method that is used in this thesis.
- Section 5 first presents the overview of the AdHierSched framework. Then we present detailed analysis of design issues about the budget controller.
- Section 6 presents the experimental evaluation and the analysis of results.
- Section 7 summarizes the related work on hierarchical scheduling and budget adaptation areas.
- Section 8 presents summary and conclusion of this thesis.
- Section 9 presents the future work.
2 Theoretical Background

In this section, we introduced some background knowledge which is relate to our thesis. For example processor partition, CPU reservation, adaptation mechanism and server-based scheduling.

2.1 Real-time systems

Real-time systems are computational systems whose correctness depend on both time and function [33]. The real-time system’s timing behaviour should be carefully analyzed and the correctness should be guaranteed. In real-time systems, tasks often perform the same functionality periodically until the end of the system’s life-time. Every instance of a task execution is called a “Job”. At each point of time, there may be more than one job ready for execution. So all the jobs should be scheduled in a proper way and finish execution before their deadlines. There are two categories of real-time systems: hard real-time system and soft real-time system.

- Hard real-time system: In this kind of real-time systems, violation of timing requirements will cause a catastrophic consequence. It’s very important to ensure there is absolutely zero timing violation in the system. A car’s air-bag system is an example of hard real-time system. When a car accident happened, the air bag should be inflated on time. Otherwise the driver may lose his/her life.

- Soft real-time system: In soft real-time systems, violation of timing requirements can be tolerated. It will not cause a catastrophic consequence, but it has some effect on the Quality of service (QoS). For example a video player system is a soft real-time system.

2.2 Processor partition technology

Embedded systems’ security is increasingly widespread nowadays. Almost all embedded systems are physically or wirelessly connected to the outside world. This connection increases the possibilities to remote control and monitor the system, but this also offers a good opportunity for cyber terrorists to infiltrate the embedded systems. In order to achieve the security operation, embedded system software is becoming much more complex, which undermines the reliability of the system. Processor partition technique is an approach to address these reliability and security issues [29]. Figure 3 is an embedded partition system example. The CPU is separated into numerous individual sections and each section acts like a separate system [3], which improves the whole system’s security and reliability. The resource partition technique was first advocated in Integrated Modular Avionics (IMA) [41]. Federated style of architecture [10] is used in current generation aircraft implementing digital flight control functions. Each function has its own computer system in this architecture. Even though the fault containment is inherent by using federated approach, the amount of hardware requirement is an obvious disadvantage. In IMA, the common computing resource to several subsystems or functions are provided by a single computer system. The system not only obtains the same fault tolerance requirement as the federated approach but also increases the resource utilization.

![CPU partition in real-time operating system](image-url)
2.3 CPU reservation technology

Reservation is a mechanism which is used to control resource access [21]. "A reservation is a guarantee on either memory or CPU for a virtual machine" [18]. CPU reservation is defined in megahertz (MHZ), it acts differently compared with memory reservation. When the CPU cycles are not used by their virtual machine, they will be redistributed to other active virtual machines, which make sure that the CPU resource is not wasted on the physical host [14]. In order to protect processor resource which is reserved between different applications on Real-Time Mach [38], in Carnegie Mellon University, a processor reservation system was originally developed [37]. This system was designed for supporting higher level resource management policies. The kernel abstraction of the system is called "reserve". It is not only used to measure the processor usage of every program but also used to track the reservation. This system shows that by using the reservation mechanism, applications can achieve predictable real-time performance.

2.4 Adaptation mechanism

The definition of adaptability is "An aggregate measure of key software characteristics that support customization of software functionality after initial development" [15]. The essential purposes of adaptability are: 1) Improve the system’s versatility. 2) Decrease the real-time system’s life cycle maintenance costs. Adaptive scheduling is the process of identifying, prioritizing, and coordinating the time requirement for tasks who need to be transmitted based on not only load situations but also on resource availability [44, 2]. Various adaptive scheduling algorithms have been proposed, which are used to improve the performance of different frameworks in real-time systems. For example, in [28], the authors proposed an interesting adaptive scheduling algorithm, which combine Earliest Deadline First (EDF) algorithm and Ant Colony Optimization algorithm [16]. The experiment results show this combination algorithm is fast and efficient in both under loaded and overloaded situations. In [34], a new framework with the ability of adaptation, reclaiming and borrowing capacity is presented. The experiment results show that this new framework has quite low deadline miss ratios and tardiness.

2.5 Server-based scheduling

Priority based scheduling such as Rate monotonic [42] and Earliest deadline first scheduling [22] are well known and widely used in real-time systems. However, these scheduling algorithms have problems to deal with aperiodic tasks which are event driven tasks. Therefore server-based scheduling algorithms emerged in this background. A server works like a periodic task but with a certain amount of budget. When the server scheduler selects a server, the server begins to run and serve the corresponding task until the server budget exhausts or the task finish its execution. If the task does not finish its execution when the server exhausts its budget, the task will continue to execute after the server budget is replenished [36]. Some most used server-based scheduling algorithms are listed below:

- **Background scheduling**: It is the simplest algorithm which is used to handle soft aperiodic tasks. The aperiodic tasks will be executed if no periodic task is ready. The poor response time is the obvious disadvantage of this scheduling when there is high periodic load.

- **Polling server** can be seen as a periodic task. It is scheduled together with other periodic tasks which can prove an on-line guarantee for the aperiodic tasks requests. Using polling server improves the task response time compares with background scheduling.

- **Total bandwidth server** is a dynamic priority server. Every time an aperiodic request enters the system, it will be assigned a deadline. The server schedules the aperiodic task who has an earlier deadline.
3 The AdHierSched Framework

Since we need to design and implement a budget borrowing mechanism in AdHierSched framework. In this section we present the background knowledge of this framework.

3.1 Application model

In AdHierSched framework, an application $A_i$ is composed of $a_i$ real-time tasks $\tau_{ij}$ and $b_i$ sub-applications $A_{im}$. Where $i$ means the $i^{th}$ application, $j$ and $m$ means the $j^{th}$ task and $m^{th}$ sub-application in application $A_i$. The term “inner components” is used to refer as both tasks and applications in $A_i$. Each application is assigned to a periodic server which will be explained in Section 3.2. Hard real-time application and soft real-time application are the two types of applications in AdHierSched framework. The application has inherit feature. For example, if an application is a soft real-time application, its children inner components are soft real-time applications or soft real-time tasks. A fixed CPU reservation is assigned to a hard real-time application during its life-time in the system. The utilized calculation algorithm is from [43]. However, a dynamic CPU reservation is assigned to a soft real-time application and the budget in a soft real-time application is a function of time $B_i(t)$.

3.2 Server model

The framework uses periodic server which is from periodic resource model [43]. The server works periodically. In each server period, a predefined amount of CPU time is provided to its children. If there is no task or application executing in the server. The server idle its CPU allocation. For a server $S_{ij}$ belonging to application $A_i$, there are four parameters in this server: period $T_{ij}$, budget $B_{ij}$, priority $Pr_{ij}$ and importance value $\zeta_{ij}$. The importance value is only used when the system is overloaded and it shows the relative importance of a server compared with other servers that exist in the same system who have the same parent. When a system is overloaded, in order to serve server who has higher importance value, the system has to sacrifice the servers who have lower importances. A soft real-time server’s budget can be adapted during run time through a budget controller, which will be explained in Section 3.5.

3.3 Task model

Two types of periodic tasks are used in this framework: 1) Task with fixed execution time (referred as static task). 2) Task with dynamic execution time(referred as dynamic task). Both types of task release periodically. A periodic task $\tau_{ij}$ which belongs to application $A_i$ is represented below:

$$\tau_{ij} = \langle P_{ij}, C_{ij}, Pr_{ij}, D_{ij} \rangle$$

where $P_{ij}$, $C_{ij}$, $Pr_{ij}$ and $D_{ij}$ represent period, worst-case execution time, task priority and task deadline respectively. In this thesis we assume that the task’s deadline is equal to the task’s period. If a task is a hard real-time task, $C_{ij}$ is available. $C_{ij}$ is kept unknown until it finishes the execution in soft real-time task. In AdHierSched framework, a new job is allowed to be executed only when the one before it finishes its execution.

3.4 System model

The AdHierSched framework is a multi-level hierarchical scheduling framework. As shown in Figure 4. There are hard real-time and soft real-time applications in one single processor system. As mentioned before, each sub-application can also include $m_i$ sub-sub-applications and $n_{ij}$ tasks. Because each application has a local scheduler, the global scheduler in a parent application schedules all the local scheduler from its children applications. So the scheduler system is also a multi-level hierarchical system.

3.5 Adaptation model

Since a soft real-time server’s budget is a function of time $B_i(t)$. The server budget is adapted according to its need at each time point by on-line monitoring its inner components’ demand. There is a budget controller [25] which is triggered periodically. This period is call a control period $P_{ctrl}$. The relationship between server period and control period is:
which means that the budget controller trigger once every $u$ server period. For example, in Figure 5, server $S_j$ period $P_j = 10$ ms, $u = 4$, so the budget controller is triggered every 40 ms. A server is assigned

$$P_{ctrl} = u \cdot P_j$$

a sufficient budget $B_i(k)$ in each server period, the value of $B_i(k)$ is:

$$B_i(k) = \lambda \cdot (b_i(k) + r_i(k))$$

Where $b_i(k)$ represents the total required budget in the $k$th control period. There is an estimator which is used to estimate the value of $b_i(k)$. $r_i(k)$ is the amount of work that should be finished in control period
$P_{k-1}^{\text{ctrl}}$, but it is postponed to $P_k^{\text{ctrl}}$. The value of $b_i(k) + r_i(k)$ is the total demand budget in the $k$th control period. But there are $u$ server periods in each control period, the budget that assigned to each server is $(1 / u) * (b_i(k) + r_i(k))$ in each server period.

### 3.6 A Linux kernel loadable model

As shown in Figure 6, AdHierSched framework is a Linux kernel loadable module which is used to arrange the real-time tasks in the Linux kernel. There is a Linux run queue in the kernel. The framework inserts a real-time task which has to run into this queue. Then the task’s state is set to running. When the task has to stop, the framework removes it from the run queue and sets task state to sleep.

![Figure 6: AdHierSched module [24]](image-url)
4 Research Methodology

In order to answer the questions that we propose in the problem formulation section and achieve the thesis goal, we roughly followed the research method in paper [39]. In the first step, we study the most used technology in hierarchical scheduling framework. Then a detailed study about the AdHierSched framework is performed, because our design and implementation are based on this framework. In the next step, we design and implement our budget borrowing mechanism in AdHierSched framework. We test the budget borrowing controller in the AdHierSched framework to make sure it is compatible well with the framework. We compare the budget borrowing controller with the budget controller by running different case-studies at last. The methodology flowchart is shown in Figure 7.

![Figure 7: The research methodology flowchart](image-url)
5 Design

In this section, we first give an overview about the AdHiersched framework. Then we present the design details about the budget borrowing controller. The important parts of the implementation are explained in the Appendices.

5.1 Overall view of the AdHierSched framework

Our framework is a multi-Level hierarchical scheduling framework. There is a global scheduler which is used to schedule all the subsystems. In each subsystem, there are two controllers: (i) Budget adaptation controller which is also called budget controller; (ii) Budget borrowing controller. We have already explained how budget controller works in Section 3.5 in the adaptation model. The design of the budget borrowing controller and how we handle the overload situations are presented in the coming section. The framework can work either with budget controller or budget borrowing controller. It can also works with both of them at the same time.
5.2 Server budget borrowing design

In this section, we present the design details about our budget borrowing controller. We explain what is server budget borrowing mechanism. When should the budget borrowing controller be called and how much budget should the server borrow every time. We also use a method to deal with the overload situation.

5.2.1 Server budget borrowing

The idea of server budget borrowing which is used in this thesis is from paper [30, 34]. In order to ensure task $\tau^j$ can finish its execution before deadline, the server $S_j$ sometimes needs to borrow budget from its future which is the next coming server period $S^j_{i+1}$. To make this idea easier to understand, here is an example. We assume that task $\tau^1$, task period $P^1 = D^1 = 10$, task execution time $C^1 = 6$. $\tau^1$ runs under server $S^1$, server period $T^1 = 10$, budget $B^1 = 4$. The changes of server budget during run time is shown in Figure 9. Since task execution time is 6, the server budget is 4. In order to make sure that task can finish its execution before deadline, server $S^1$ need to borrow budget from server period $T^2_1$. The amount of budget needs to be borrowed is 2. As you can see from Figure 9, in server period $T^2_1$, the server budget is less than before, which may have a bad effect on the coming task. In order to solve this problem, we can use the budget controller to adapt the server budget at the same time.

![Figure 9: Server borrow budget example](image)

In the above example, we assume that the execution time of the task is recognized. However, for some of the soft real-time tasks, their execution times are unknown. In order to use this budget borrowing mechanism in AdHierSched framework, an estimator which is used to estimate the amount of budget that need to be borrowed should be designed and implemented. It is also important to know when the server need to use the budget borrow controller.

5.2.2 Time to borrow budget

Since the task execution time is unknown, we do not know when the task finishes its execution. As a result, the server does not know when to borrow budget. In order to solve this problem, we design a new way to decide when the server needs to borrow budget from future.

We assume that dynamic task $\tau^j$ whose execution time $C^j$ is unknown. Its deadline is equal to the task period: $P^j = D^j$. As mentioned before, we call every instance of a task execution as a “job”. We add a boolean variable into the framework. The variable is `job_state_check` which is used to check whether a job finishes its execution or not. When a job begin to execute, we set the value of the variable to false which means that the job is not finish yet. Since we do not know the execution time of the job, we do not know when the job finishes its execution when it finishes. When the job finishes its execution, we set the value of `job_state_check` to true.

Since we assume that task’s deadline is equal to the task period. When server does not have any budget, the job needs to stop its execution and waits until the server budget been replenished. We need to calculate the time interval between the job deadline and job stop time $t_k$. We call this time interval $\Delta t$. 

16
\( \Delta t \) needs to be compared with server period. When \( \Delta t \) is smaller than server period, meanwhile the value job state check is false, then we know this job will miss its deadline. Because during \( \Delta t \) this job can only wait. In order to make sure this job can execute during this \( \Delta t \) time, we need to use the borrowing mechanism to assign a certain amount of budget to server before the server stops. As shown in Figure 10, task \( \tau_j \) and server \( S_j \) are running in the framework. Task \( \tau_j \) has to stop at \( t_1 \), because server \( S_j \) runs out its budget and will be replenished in server period \( T^j_2 \). \( \Delta t_1 \) is bigger than the server period \( T^j_2 \), we know that we do not need to borrow budget right now. When the time goes to \( t_2 \), we compare the time interval \( \Delta t_2 \) with server period \( T^j_2 \). Because \( \Delta t_2 \) is smaller than server period \( T^j_2 \) this time. If \( \tau_j \) can not finish its execution, it will miss its deadline. In order to avoid this, we can use server budget borrow mechanism to borrow budget from future, which is server period \( T^j_3 \). By doing so, the task may have a chance to finish its execution before its deadline.

Figure 10: When need to borrow budget

5.2.3 How much budget need to borrow

To answer this question, first we need to know how much budget we can borrow. Therefore, calculating the free bandwidth of the system is necessary. Because the total of the CPU bandwidth is 1, we just need to exclude the bandwidth of all the applications from the parent bandwidth. The calculation is shown in equation (1), where \( U_{\text{free}} \) is the free bandwidth, \( U_i \) is the utilization of server \( i \). By using the free CPU utilization, we can get the maximum budget that we can borrow in this server period. In equation (2), \( B_{\text{Max}} \) represents the maximum amount of budget that can be borrowed, \( T^j \) represents the period of server \( S_j \), \( S_j \) is the server who needs to borrow budget from future.

\[
U_{\text{free}} = 1 - \sum_{i=1}^{k} U_i \quad (1)
\]

\[
B_{\text{Max}} = U_{\text{free}} * T^j \quad (2)
\]

In order to estimate the amount of borrowed budget, we use Autoregressive (AR) model [4] to estimate the new server budget in this thesis. Autoregressive indicates AR is a regression of the variable against itself. A new good quality value is forecasted according to the history values. The definition of AR can be clearly understood as shown below:

\[
B_{k+1} = c + [\theta_1^* B_1 + \theta_2^* B_2 + \ldots + \theta_k^* B_k] + \epsilon_k,
\]
Where $c$ is a constant value, $e_t$ represents White noise, $\theta_k$ is the autoregression coefficients and $B_{k+1}$ is the new estimate value.

In the adaptation model, the server will be assigned a new amount of budget at the beginning of each server period. In our thesis project, there are two budget values will be estimated by using AR model when budget controller is called:

1. $b_{k}^{opt}$ is the optimum estimated budget value, which is assigned to the server at the beginning of each server period;
2. $b_{k}^{pes}$ is the pessimistic estimated budget value, this value is used only when the server need to borrow budget from the future server.

The equations that are used to calculate $b_{k}^{opt}$ and $b_{k}^{pes}$ are shown in (4), (5).

$$b_{k}^{opt} = [\omega_1 \beta_1(1) + \omega_2 \beta_2(2) + \ldots \omega_k \beta_k(k-1)] + e_{k}^{opt}$$

$$b_{k}^{pes} = [\omega_1 \beta_1(1) + \omega_2 \beta_2(2) + \ldots \omega_k \beta_k(k-1)] + e_{k}^{pes}$$

where $\omega_k$ is the autoregression coefficient of observation $k$, $\beta_{k-1}$ is the $(k-1)$th server budget, $e_{k}^{opt}$ and $e_{k}^{pes}$ are Gaussian white noises. When a server begins to run, $b_{k}^{opt}$ amount of budget is assigned to it. However, if the server need to borrow budget according to section 5.2.2, the amount of budget that can be borrowed is $\theta$:

$$\theta = b_{k}^{pes} - b_{k}^{opt}$$

5.2.4 Dealing with overload situations

When a new server budget $b_{k}^{opt}$ is assigned to a server, we need to think about the overload situations of the system. We use the same way as in paper [25] to decide the last value of $b_{k}^{opt}$. However, when the server needs to borrow budget, in order to avoid overload after the borrow budget. We need to compare the borrow budget value $\theta$ with the maximum borrow budget value $B_{Max}$. Two situations can happen:

1. $\theta < B_{Max}$: we assign the $\theta$ budget to the server.
2. $\theta > B_{Max}$: we assign $B_{Max}$ budget to the server.

However, when we use budget borrowing mechanism and budget adaptation mechanism at the same time, the overload situation may happen. When overload situation happens, the applications with higher importance value will steal budget from lower importance applications, which make sure that the more important applications can have a bigger chance to meet their deadline.
6 Evaluation

This chapter contains the experiments that are conducted on the developed system. We first present the tasks that are used in the evaluation. Then we present the set-up of the system and the evaluation conditions. The evaluation results are presented last.

6.1 Tasks

We used two types of tasks to evaluate the framework. (i) Static tasks whose period and execution time keep fixed; (ii) Dynamic task whose period is fixed, but the execution time changes during the run-time. The dynamic tasks are generated by using C# code with different algorithms such as standard distribution, Poisson distribution.

6.2 Evaluation setup

The hardware set-up about the experiment in this thesis project is declared here. An Intel(R) Core (TM) i3-M330 processor which clocked at 2.13 GHz is used in the experiments. The hardware is equipped with a 1 GB memory. The operating system is Ubuntu 14.04.02 with Linux kernel version 3.8.2. The scheduler resolution used in the kernel is one millisecond.

During the experiments, the autoregression coefficients $\omega_k$ in equations (4) and (5) are set to $1/\eta$. The Gaussian white noises $e^{opt}_k$ and $e^{pes}_k$ are set to $1/2 \times std(\beta_j(k), \eta)$ and $2/3 \times std(\beta_j(k), \eta)$ respectively. $std$ means the standard deviation of its $\eta$ previous $\beta_j(k)$. Since both budget controller and budget borrowing controller have negligible overhead, we set the trigger period of both controllers equal to the corresponding server period.

6.3 Evaluation conditions

Three different approaches are used to evaluate the performance of the framework in this thesis:

- **Condition 1**: Tasks run with only budget borrowing controller.
- **Condition 2**: Tasks run with only budget controller.
- **Condition 3**: Tasks run with both budget controller and budget borrowing controller.

Task deadline miss ratio and server workload in terms of CPU utilization were recorded. The collected data from the evaluation were compared with each other.

6.4 Static tasks

In order to ensure the framework works properly, we evaluated the framework by running static tasks first in this section. The tasks’ deadline miss ratios are shown in different figures.

6.4.1 Case study 1

<table>
<thead>
<tr>
<th>Task/Server</th>
<th>Period (ms)</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau^j_1$</td>
<td>200</td>
<td>175</td>
</tr>
<tr>
<td>$S^j_1$</td>
<td>100</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: The specification of $\tau^j_1$ and $S^j_1$

In this experiment, we used one static task $\tau^j_1$ and a server $S^j_1$. The specification of $\tau^j_1$ and $S^j_1$ used in this case study are shown in Table 1. The initial server budget is based on the average workload which is 80 ms in this case. Task $\tau^j_1$ ran 100 jobs under each condition and its execution time tracing figures are shown in Figure 12. The deadline miss ratio (DMR) is shown in Figure 11.
When the framework works under condition 1, we traced the task execution time and the amount of server budget consumption. Part of the traced results are shown in Figure 12. In Figure 12 (a), because of the budget borrowing controller, the first job can finish its execution before deadline. However, because of the budget of the future server is borrowed by the server before it. Even though the fourth server period can borrow a limited amount of budget from its next server period, the server budget is still not enough for the second job to finish its execution before deadline, which leads to a Domino effect, all the coming jobs after the second job cannot finish their execution on time. Figure 12 (b) shows the results that when the framework works with only budget controller. In the first job execution period, server budget controller adapts the server budget to make sure that there is enough budget for the next job. According to result shown in Figure 12 (b). After the first two server periods, the server can always have enough budget for the coming job to finish execution before their deadlines. When the framework works under condition 3, the evaluates results are shown in Figure 12 (c). All the jobs meet their deadlines. This is mainly because when the server does not have enough budget, the borrow budget controller and budget controller work together, which adapt server budget at the same time. Therefore all the jobs can meet their deadlines. The server borrows budget from the next coming server and the budget controller assigns more budget to the server if the previous server period does not have enough budget.

Figure 11 shows the task deadline miss ratio when \( \tau^j \) work under different conditions. The data bar shows that when the framework works under condition 1, the task DMR is 99%. The DMR value is decreased to 1% when the framework works under condition 2. At last we evaluate the framework under condition 3 and the task DMR value is zero.

6.4.2 Case study 2

We evaluate the controllers by running four tasks and three servers in the framework in this case-study. The structure of the case-study system is illustrated in Figure 13. Task \( \tau^0 \) and \( \tau^1 \) run in server \( S^0 \). Task \( \tau^2 \) and \( \tau^3 \) running under server \( S^1 \) and \( S^2 \) respectively. The specification of \( \tau^i \) and \( S^i \) used in this case-study are shown in Table 2. All the tasks and servers are soft real-time tasks and servers. The server importance value is assumed by the server order number (i.e. \( \varsigma_0 > \varsigma_1 > \varsigma_2 \)).

We record the performance of the framework when it work under these three conditions. The evaluation results are shown in Figure 14. The data bar represents the task deadline miss ratio when task runs under different servers. When running the framework under condition 1, the tasks DMR value is 10% under server \( S^0 \). The DMR value are 6% and 0% under server \( S^1 \) and \( S^2 \) respectively. When the framework running under condition 2, the results are much better than the results from condition 1. The reason is that after enabling the budget controller, the server runs in this framework had the budget adapt ability. The server’s budget keeps updating in each budget control period, which ensure the server has
sufficient budget for the tasks. When running the framework under condition 3, the DMR of tasks from server $S^0$ and server $S^1$ declines to zero. However, the DMR value of task $\tau^3$ is increased to 2%, this is mainly because each server in AdHierSched framework has an importance value. When the total CPU utilization is bigger than one, the system is overloaded. So the framework serves the server with higher importance value, which is paid by sacrificing less important applications.
Figure 13: The structure of the case-study system with multi-tasks

<table>
<thead>
<tr>
<th>Task/Server</th>
<th>Hard-Soft</th>
<th>Period</th>
<th>Execution time/Budget</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^0$</td>
<td>SRT server</td>
<td>100</td>
<td>50</td>
<td>-</td>
</tr>
<tr>
<td>$\tau^0$</td>
<td>SRT task</td>
<td>200</td>
<td>30</td>
<td>99</td>
</tr>
<tr>
<td>$\tau^1$</td>
<td>SRT task</td>
<td>200</td>
<td>21</td>
<td>98</td>
</tr>
<tr>
<td>$S^1$</td>
<td>SRT server</td>
<td>100</td>
<td>30</td>
<td>-</td>
</tr>
<tr>
<td>$\tau^2$</td>
<td>SRT task</td>
<td>200</td>
<td>28</td>
<td>97</td>
</tr>
<tr>
<td>$S^2$</td>
<td>SRT server</td>
<td>100</td>
<td>20</td>
<td>-</td>
</tr>
<tr>
<td>$\tau^3$</td>
<td>SRT task</td>
<td>200</td>
<td>15</td>
<td>96</td>
</tr>
</tbody>
</table>

Table 2: The specification of servers and tasks in the case study

6.5 Dynamic tasks

In this section, we used dynamic tasks to evaluate our controllers. In the first case-study, we use only one dynamic task and one server. In case-study 2, we use multiple dynamic tasks and servers. The evaluation results are shown below.

6.5.1 Case study 1

In this experiment, one dynamic task and one server are used. The initial server budget is assigned based on the average workload. The task period is 200 milliseconds. The task execution time is ranged between 40 and 90 milliseconds. The distribution of all the job execution times are shown in Figure 15. Here we use 20 sets of execution sequence for each dynamic task. The specification of server and task are shown in Table 3.

<table>
<thead>
<tr>
<th>Task/Server</th>
<th>Period</th>
<th>Execution time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau^0$</td>
<td>200</td>
<td>40 - 90</td>
</tr>
<tr>
<td>$S^0$</td>
<td>100</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3: The specification of server and dynamic task in case-study 1
The task’s deadline miss ratio is illustrated in Figure 16. Since there are 20 set of execution sequence for each dynamic task, the results are different each time. The highest, lowest and average deadline miss ratios are included in Figure 16. When the framework running under condition 1, the task average deadline miss ratio is 51%, the highest deadline miss ratio is 93% and the lowest deadline miss ratio is 29%. When the framework works under condition 2, the results are much better than when the framework works under condition 1. The average DMR value is 19%. When the framework running under condition 3, the task average deadline miss ratio is only 13%.

During task run-time, the server’s budget adaptation is recorded. We use 20 sets of execution sequence for the dynamic task in this case-study, but we only shown one group of the server budget adapt state which is in Figure 17. The server’s initial budget is 35. When the framework works with neither budget controller nor budget borrowing controller, the server budget is the red line which is a constant value 35. The light green line shows the framework works under condition 1. If the server does not have enough budget for the executing task, it could borrow budget from the next coming server. Therefore sometimes the server maximum budget can reach to 70. When the server budget is 70, the next coming server budget
will be equal to zero. This may cause more coming jobs missing their deadlines. So this budget adaptation method is not good when using it alone. The black line shows the server budget changes when using budget controller. The blue line shows how server adapts budget when using both budget controller and budget borrowing controller.

6.5.2 Case study 2
In this part, we use five dynamic tasks and three periodic servers in this case study, and another 20 sets of execution sequence for each dynamic task are evaluated in the AdHierSched framework. The results are quite different depending on the random task execution time. The tasks and server parameters are shown in Table 4. From row three we can find that $\tau^0$'s execution time is 25 - 45, which means that the execution time is a random value between 25 and 45. $\tau^0$ and $\tau^1$ run under server $S^0$. $\tau^2$ and $\tau^3$ run under server $S^1$. $\tau^4$ runs under server $S^2$.

Different tasks deadline miss ratios are shown in Figure 18. Since we use 20 sets of execution sequence for each dynamic task, the maximum deadline miss ratio, the average DMR and the minimum DMR are
<table>
<thead>
<tr>
<th>Task/Server</th>
<th>Hard-Soft</th>
<th>Period(ms)</th>
<th>Execution time/Budget(ms)</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^0$</td>
<td>SRT server</td>
<td>100</td>
<td>50</td>
<td>-</td>
</tr>
<tr>
<td>$\tau^0$</td>
<td>SRT task</td>
<td>250</td>
<td>25-53</td>
<td>99</td>
</tr>
<tr>
<td>$\tau^1$</td>
<td>SRT task</td>
<td>200</td>
<td>20-55</td>
<td>98</td>
</tr>
<tr>
<td>$S^1$</td>
<td>SRT server</td>
<td>100</td>
<td>35</td>
<td>-</td>
</tr>
<tr>
<td>$\tau^2$</td>
<td>SRT task</td>
<td>200</td>
<td>12-33</td>
<td>97</td>
</tr>
<tr>
<td>$\tau^3$</td>
<td>SRT task</td>
<td>250</td>
<td>13-37</td>
<td>96</td>
</tr>
<tr>
<td>$S^2$</td>
<td>SRT server</td>
<td>100</td>
<td>20</td>
<td>-</td>
</tr>
<tr>
<td>$\tau^4$</td>
<td>SRT task</td>
<td>250</td>
<td>8-15</td>
<td>95</td>
</tr>
</tbody>
</table>

Table 4: The specification of servers and dynamic tasks in case study 2

![Task deadline miss ratio under different conditions](image)

Figure 18: The dynamic tasks DMR when framework working under different conditions

shown in figure 18. When the framework works under condition 1, all the tasks have deadline miss. When the framework worked under condition 2, the results are better than the framework works under condition 1. However, when the framework worked under condition 3, the deadline miss ratios of task $\tau^0$, $\tau^1$ and $\tau^2$ are slight lower than when the framework works under condition 2. But the deadline miss ratio of $\tau^3$ and $\tau^4$ change in a different way. Because the total CPU utilization is bigger than one when the framework works under condition 3 in this case-study, so overload happens during the evaluate process. The system assigns resource according the application importance value. The importance value of servers order by the server number (i.e. $\varsigma_0 > \varsigma_1 > \varsigma_2$). Since $S^0$ is the most important application in the system, the framework sacrifices the applications with a lower importance value in order to avoid $\tau^0$ and $\tau^1$ have a high DMR value, which also is the reason why the DMR values of task $\tau^3$ and $\tau^4$ are higher than when the framework works under condition 2.

6.6 Overhead

After excluding the Linux scheduler overhead, we have three sources of overhead:

(1) Hierarchical scheduling overhead which means the calculation when the tasks are scheduled in a hierarchical manner.

(2) Overhead of using budget controller to calculate and adapt server budget.

(3) Overhead of using budget borrowing controller to calculate and borrow server budget.
The overhead is measured by using time stamps. We use time stamps to measure the time length of timer handler and task_finish_job(task_id) function. The time interval is calculated and then divided by the total running time of the experiment. We measure the overhead in section 6.5 case study 2. When the framework works under condition 3, the proportion of the three sources of overhead are shown in Figure 19. The extra calculation overhead of budget controller and budget borrowing controller are almost the same. Both of them are quite small. The total average overhead is 0.224% of the CPU time. We believe this amount of extra overhead is acceptable in reality.

![Figure 19: The overhead in section 6.5 case study 2, when the framework works under condition 3](image-url)
7 Related Work

In this section, we study the state of the art related to hierarchical scheduling and budget adaptation mechanisms.

Nima et. al. [27] used feedback scheduling techniques to assign and adapt CPU resource to subsystems in hierarchical scheduling framework. The authors designed a PI controller, this controller has the ability to adapt the server budget when there is some change of the tasks execution time during run-time. The authors also designed an overload controller which is used to assign resource to subsystems under an overload situation. Each subsystem has a criticality value. When overload happened, the framework assigns the CPU resource depend on the critical value of the subsystems. The subsystems with higher critical values are more prevailing in receiving the CPU resource than the lower critical subsystems. The evaluation results from this paper shows that the PI controller and the overload controller can not only decrease the a system’s deadline misses but also achieve a higher CPU utilization of the system. However, the performance of the PI controller is not so well when there are multiple tasks. In order to improve the performance of the hierarchical scheduling framework, Nima et. al. [26] designed a fuzzy controller instead of the PI controller. After compared the evaluation results with the results from PI controller, the fuzzy controller performance better than PI controller. In our thesis we used a similar hierarchical scheduling framework but with our own budget adaptation mechanism.

Mikael et. al. [5] presented a loadable kernel module which is Hierarchical Scheduling Framework recorder. This recorder is based on the REal-time SCHeduler (RESCH) framework in Linux operating system. It has the ability to trace and recorder tasks and servers without any kernel modifications. The authors first evaluated the overhead of the HSF recorder through comparing with task switch patch [6]. Then the tracing ability and accuracy of the recorder are evaluated by using a Multimedia example. The authors compared the evaluation results with the results from another recorder which named Ftrace [9]. The evaluation results shows that the HSF recorder not only has a lower overhead but also a good choice when the user only want to monitor a subset of Linux tasks. In our thesis, the budget borrowing controller is based on AdHierSched framework which is running in Linux kernel.

Caixue et. al. [30] presented a four principles which makes the slack management more effective in an EDF-based system. Based on this four principles, the authors implemented SRAND, SLAD, SLASH and BACKSLASH scheduling algorithms in the Linux 2.6 kernel. Each algorithm add one principle to the previous algorithm. Since BACKSLASH algorithm used all the four principles, its performance is the best one among the four algorithms. The authors then compared BACKSLASH with CBS, and CASH. The results shows that the BACKSLASH algorithm distinctly decrease the deadline miss ratio of the workload. Meng et. al. [34] present a new reservation-based scheduling framework which is called CARB-EDF. It is a similar framework with BACKSLASH. But CARB-EDF framework has the traits of capacity adaptation, reclaiming and borrowing. In this paper, the future execution times of tasks are estimated by using Chebyshev’s inequality [45]. There are two phases in CARB-EDF framework, initial phase and runtime phase. During the initial phase, the scheduler reserves some budget which is used for the budget adaptation later, then the available budget is assigned to all the tasks fairly. In the runtime phase, the core function is use the Chebyshev inequality to estimate the bound of the task execution time. In the evaluation part, the authors compared the performance of CARB-EDF framework with BACKSLASH algorithm, the results shows that the former has lower deadline miss ratio and less overhead than the later. The extra overhead of the CARB-EDF framework was examined at the last. In our thesis, we also used budget borrowing mechanism and a budget controller to adapt the server budget during the run-time.

Giuseppe et. al. [32] presented a technique to assign the server parameters in hierarchical scheduling systems. The authors analyzed a fixed priority scheduler and designed a server which is used in hierarchical scheduling systems. Doerr et. al. [15] developed a Quality of Service framework with the ability to adapt resource during system configuration or even at run-time. The dynamic scheduling was studied and used in this framework. Adaptive resource management (ARM) was used to decrease the period of certain operations. In the application adaptation control section, the author used a resource manager to analyze the application state dynamically, which make sure the adaptation decisions from the framework were more optimal. In our thesis, we use a different resource adaptation mechanism to adapt the server budget.
A hierarchical loadable scheduler (HLS) architecture was designed and implemented in Windows 2000 by John [40]. The HLS’s feasibility was demonstrated through two ways: (i) A hierarchical scheduler infrastructure was designed. (ii) Implemented the HLS’s architecture in the Windows 2000 kernel. In order to establish the usefulness of HLS, the author first studied a number of scheduling behaviours and selected three types of useful scheduling behaviour. Then a new method was used to verify the correctness of the scheduling hierarchies, which means that the guaranteed scheduling behaviour was assigned to the corresponding application thread correctly. In order to increase the application predictability when a running application’s CPU time is stolen by the operating system, the author designed, implemented and evaluated Rez-C scheduler and Rez-FB scheduler.

At last we studied an interesting scheduling algorithm which was presented by Kotecha et. al. [28]. It is the EDF combine with Ant Colony Optimization (ACO) based scheduling algorithm. The basical idea of this real-time scheduling algorithm is very clear. When there is no overloaded, the system uses EDF algorithm. If the system overloaded, then the system stops using EDF scheduling algorithm and switches to ACO based scheduling algorithm. If the overload disappear, the system continues to use ACO based scheduling algorithm for a certain time and then it switches back to EDF algorithm. The ACO is a branch of the Swarm Intelligence. The ACO algorithms are computational models which are enlighten by the ants’ collective foraging behaviour. These smart ants deposit pheromone in the path, the pheromone is used to mark the shorter path so that could be used by other members of colony. This phenomenon is used to solve the optimization problems [16]. The evaluate results from [28] shows that this framework works well both under the single processor and multiprocessor real-time system. Even though the author combine this two algorithms together, however, both of them have limitations when the overload happens. For example, the EDF algorithm does not perform very well and the ACO based scheduling algorithm requires more execution time under the overload situations. We think this is an interesting scheduling algorithm and want to study its performance in AdHierSched framework in the future.
8 Summary and Conclusions

In real-time system, the resource adaptation is not only a very important approach to improve the flexibility of the system but also a way to decrease the life-cycle maintenance cost of the system. In this thesis, we explore a new direction for performing the resource adaptation in hierarchical scheduling real-time system. We design and implement a budget borrowing controller, which is used to adapt the resource for the AdHierSched framework during the run-time. Besides, we use Autoregressive model to estimate the new budget for servers, which forecasts a new good quality value according to the history values.

We evaluate the budget borrowing controller by using static tasks and dynamic tasks. When the framework running only with budget borrowing controller. The evaluation results are not good and the tasks deadline miss ratios are quite high. We compared the performance of budget borrowing controller with the budget controller by running different case-studies in this thesis. When the framework works only with budget borrowing controller, the evaluation results are not good as when the framework running with budget controller. However, when the framework works with both controllers, the evaluation results are good enough. Unfortunately, when running both controllers at the same time, the overload situation can easily happen, which is not good for the application who has a lower importance value.

We measured three types of overhead in this thesis. The extra overhead from budget adapt calculation of both controllers are very small compared with the total overhead, and the total overhead was also quite small which we think is acceptable in reality.

We can conclude that when we use budget borrowing mechanism alone, it does not work very well. However, when we combine it with other adaptation mechanisms, the results are good enough and the budget borrowing calculation overhead is very small compared with the hierarchical scheduling overhead.
9 Future Work

In this thesis, our future work can focus on several aspects which are listed below:

- We have explored the performance of budget borrowing controller in two-level AdHierSched framework. More than two-level systems running with budget borrowing controller can be investigated in this framework in the future.

- We have already used static tasks and dynamic tasks in our evaluation. We can investigate the performance of the budget borrowing controller by running real applications such as MPlayer video player.

- We used periodic server as our server-based scheduler in this thesis. Some other new server-based schedulers such as Bandwidth Sharing Server can be investigated in the future.

- Investigate and implement Ant Colony Optimization (ACO) techniques in AdHierSched framework to adapt server budget is another approach of the future.

- The AdHierSched framework runs on a single processor now, we can extend it to multiprocessors.
References


A Knowledge about Linux operation system

A.1 Jiffies

Jiffies is a global variable in the Linux kernel, one jiffy is the time interval between two successive clock ticks, after the system is booted, the system begins to count the tick that have occurred and then it stores the number of ticks in the jiffies. The kernel initializes jiffies to zero when boot a Linux operating system, whenever there is a timer interrupt happen, its value increases, which means if there are HZ (Hertz) timer interrupt in a second, then there much be HZ jiffies in a second, so the relationship between jiffies and seconds is shown in equation (2):

\[
\text{seconds} = \frac{\text{jiffies}}{\text{HZ}},
\]

(7)

The type of the jiffies variable is always unsigned long in the kernel, if the operating system running on 32-bits architectures with the HZ value equal to 100, then the jiffies value will overflow in about 497 days, but if the HZ value is 1000, then it will need about 49.7 day before the jiffies overflow. However if the operating system running on a 64-bits architectures with a reasonable HZ value, the jiffies will never overflow in any lifetime [35].

A.2 Recompile the Linux kernel

The first step is to get the Linux kernel source code from http://kernel.org/, choose the right kernel version and download it, then use the gcc compilers to configure the kernel by type the "make" command in the terminal, and the path that need be /usr/src/linux+version number, the "make" processor step will cost around three hours, after the "make" step finish, the kernel modules should be compile next. After the kernel modules finished the compile, now we can install the new kernel, this may take half an hour, the last two steps are create an initrd image and update the grub configuration file, the initrd images contains device drivers, these drivers are used to load the operating system.

While when all the steps above are finished, reboot the laptop, at the beginning of the boot process, press the Shift button from the keyboard (maybe different brand of computer should chose to press different button) and chose the new kernel, when finished the reboot process, open the terminal and type the command "grep HZ /boot/config-‘uname -r’" to check the HZ value, if the HZ value is not 1000, you need to open the terminal and type /usr/src/linux/.config, then according figure 20 modify HZ to 1000 and then recompile the kernel again.

![Figure 20: Setup the operating system HZ value](image)

(a) Step 1 (b) Step 2
B  Random distributions

B.1 Normal distribution

Normal distribution is a measure method that used in statistics, it is mainly used to dispersion of big data values or check the data variation, for example in social science and natural, there are many of real-valued random variables, usually these variables’ distribution are unknown, so we can use normal distribution to represent them. Because of the shape of the line, normal distribution is also called bell curve, in Figure 21, where \( \mu \) is the mean value of the distribution, \( \sigma \) is the standard deviation, we can find that the mean value \( \mu \) is right in the middle and most of the data is sitting around of the mean, the total area under bell curve is equal to one, another thing need to mention is that the bell curve is symmetrical, which means that the left area of the mean value is equal to the right side, both side are equal to 0.5. standard normal distribution is a special case of normal distribution, whose mean value is equal to zero and the population standard deviation is equal to one. We use the normal distribution method to create the dynamic task.

Figure 21: Normal distribution bell curve

B.2 Poisson distribution

Poisson distribution is a very important Mathematics method in statistics and probability theory, it is a discrete probability distribution, if we know an event happened average rate and the different events are independently with each other, then we can calculate the probability of a given number of events occurring in a fixed time or area interval [1], for example, suppose a fast-food restaurant can expect two customers every three minutes on average, so we can use poisson distribution to calculate the possibility that four or ten customers entry into the restaurant in nine minutes or any time interval, some other examples such as calculating the possibility that 50 cars pass a crossroad in 20 minutes [17]. According this feature I used Csharp create some dynamic tasks which were tested in the evaluation part.

B.3 Exponential distribution

Exponential distribution is another very important method in statistics and probability theory, it is used to repent the time between events in a Poisson process (Poisson process is a random process which is used
to count the number of events\cite{11}). From Figure 22 we can know the formula probability density function
and how it looks like, \( \lambda \) is called rate parameter, it is the parameter of the distribution and its value always
bigger than zero, the area under the curve line is equal to one. We also use exponential distribution create
some dynamic tasks.

\[
\begin{align*}
    f(x; \lambda) &= \begin{cases} 
        \lambda e^{-\lambda x} & x \geq 0, \\
        0 & x < 0.
    \end{cases}
\end{align*}
\]

Figure 22: Exponential distribution probability density curve

C Part of the implementation

C.1 Stop Server Function

The stop server function is a very important function in this framework, the borrow budget function will
be called in this function.

At the beginning of the function, we will check whether we need to stop a server or not, we must
check the active server is NULL or not before we call the stop server function (because in the linux kernel,
if you stop a function with a NULL pointer, the whole system will crush). If there is no active server right
now, then we just return \texttt{RET\_SUCCESS}, but if there is an active server, then we set the active server to
NULL. The next step is to compare the value of server current budget with the task consume budget in
this server period, if it is not bigger than server current budget, then we just remove the consume budget
which is used by the running tasks from server current budget, otherwise we set server current budget
to zero. Because AdHierSched framework is a multi-level framework, then we need to check the server’s
parent is NULL or not, if the server parent is NULL, we just remove the budget timer, otherwise we need
to add the consume budget to the server parent consume budget and then remove the budget timer.

Before we finish the stop server function, we need to check if there is task running in the system, if
there is a task running, then check the running task’s parent is this server or not is needed, if the task’s
parent is not this server, we just return \texttt{RET\_SUCCESS}, otherwise we get the error between the next job
release time and jiffies, then we check the error value is bigger than the server period or not, if the error
is bigger than the server period, we just stop the running task and search the ready task queue to see
whether this task in the queue or not, if it is already in the ready queue, we just return, but if this task is
not in the ready task queue, we need to insert it into the queue; but if the error is smaller than the server
period, then we need to check the \texttt{running\_task>task\_finish\_job\_flag} is true or not, if this
value is \texttt{false}, we stop the running task and check the task ready queue, if this value is \texttt{true}, we will
call the borrow budget function and set the server that we are going to stop as active server, then we insert
the server budget timer (we will get the timer value in the borrow budget function), the next is call the
run queue function and check the ready queue again, until now we can return \texttt{RET\_SUCCESS}.

C.2 Budget borrowing design flowchart

The budget borrowing controller design flowchart is shown in Figure 24.
Check the server is NULL or not

set the active server to NULL

Check the server current budget is bigger than the budget used by tasks

remove the used budget from the server current budget

set the server current budget to 0

Check the server parent is NULL or not

add the used budget by tasks to the server parent consume

remove the server budget timer

check the running task is not NULL &&
running_task-parent is S

get the error
between the next job release
time and jiffies, check whether the error
is bigger than the server period

check the value of
running_task-mask (from_jk_flag)

is true or not

call the borrow budget function;
set active server = S;
insert server budget timer and call
run queue function

search queue task to see the running task
in the ready queue or not

insert task into ready queue

return 1
Figure 24: Borrow budget design flowchart
D Part code implementation in this thesis

D.1 Borrow budget function code

```c
static inline int borrow_budget(struct Server *server)
{
    // calculate the max budget can borrow
    struct Server *s_num;
    int last_server_id = 0;
    long free_utilization = 0;
    long used_utilization = 0;
    int i;
    // struct Handler_Data *data_budget;
    server->borrow_budget_flag = true;
    // int max_borrow_budget = 0;
    // if(server->server_type == PERIODIC_SERVER)
    {
        last_server_id = get_last_server_id(&server_list.head) + 1;
        printf("borrow_budget", "there are %d server in the server list!!!", last_server_id);
        for(i = 0; i < last_server_id; i++)
        {
            s_num = find_server(&server_list.head, i);
            if(s_num->parent == NULL)
            {
                used_utilization += s_num->budget*100/s_num->period;
            }
        }
        free_utilization = 100 - used_utilization;
        server->max_borrow_budget = free_utilization * server->period/100;
        printf("borrow_budget", "the max_borrow_budget is ",
        server->max_borrow_budget);
        server->avg_server_budget = server->total_budget / server->cnt;
        if(server->avg_server_budget > server->budget)
        {
            server->borrowed_budget = server->avg_server_budget/2 +
            (server->avg_server_budget - server->budget);
        }
        else
        {
            server->borrowed_budget = server->avg_server_budget/2 +
            (server->budget - server->avg_server_budget);
        }
        server->current_budget = server->borrowed_budget;
        printf("borrow_budget", "After borrow the budget, the budget
        left in the server is ", server->current_budget);
    }
    return RES_SUCCESS;
}
```

D.2 Redesign the stop server function

```c
static inline int stop_server(struct Server *server)
{
    struct Task *prev;
    struct Handler_Data *data_budget;
    struct Queue_element *element;
    if(check_server(server) == RES_FAULT)
    {
        #ifdef DEBUG_SCHED
        print_warning("stop_server", "*********** STOPING NULL SERVER");
        #endif
    return RES_SUCCESS;/** It is OK to be NULL*/
    }
    active_server = NULL;
```
```c
#define DEBUG_SCHEDULE

print_int("stop_server", "consumed_budget", server->consumed_budget);
#endif

if (server->current_budget > (jiffies - server->timestamp))
{
    server->current_budget = server->current_budget -
    (jiffies - server->timestamp);
    print_int("stop_server", "consumed_budget", server->current_budget);
} else
{
    server->current_budget = 0;
}
#endif

if (server->parent != NULL)
{
    if ((jiffies - server->timestamp) > server->parent->period)
    {
        #ifdef DEBUG_SCHEDULE_ERRORS
        print_warning("stop_server", "consumed_budget more than period");
        #endif
        server->parent->consumed_budget += (jiffies - server->timestamp);
    }
    #ifdef JIFFY_TIMER
    remove_timer(&server->budget_timer);
    #endif
    #ifdef DEBUG_SCHEDULE
    print_warning("stop_server", "************** STOPING ACTIVE SERVER");
    #endif
    // if (running_task != NULL && running_task->parent == server)
    // {
    //     prev = running_task;
    //     stop_task(running_task);
    //     prev->state = TASK_WAITING;
    //     #ifdef DEBUG_SCHEDULE_TMP
    //     print_warning("stop_server", "************** calling
    //     release_hsf_task***********");
    //     #endif
    //     #ifdef DEBUG_SCHEDULE_TMP
    //     print_int("stop_server", "inserting task to queue", prev->id);
    //     release_hsf_task(prev); // to insert it into queue again
    //     prev->state = TASK_READY;
    // }

    if (running_task != NULL && running_task->parent == server)
    {
        prev = running_task;

        // the new check function about the task used budget and make
        // sure when need to borrow the budget.
        if (running_task->release_time - jiffies < server->period)
        {
            if (running_task->task_finish_job_flag == 1)
            {
                if (server->avg_server_budget > server->budget)
                    server->borrowed_budget = server->avg_server_budget/2 +
                    (server->avg_server_budget - server->budget);
                else
                    server->borrowed_budget = server->avg_server_budget/2 +
                    (server->budget - server->avg_server_budget);
                // here should has the borrow function.
                borrow_budget(server);
                print_int("stop_server", "the budget left in the server is ",
                        server->current_budget);
                //
                data_budget = (struct Handler_Data *) kcalloc(sizeof(struct Handler_Data)
```
D.3 C# code that used to generate dynamic tasks Part A

```csharp
namespace DistributionData {
    class Calculate {
        /// <summary>
        /// Normal distribution
        /// </summary>
        const int N = 100;
        const int MAX = 50;
        const double MIN = 0.1;
        const int MIU = 40;
        const int SIGMA = 1;
        static Random aa = new Random((int)(DateTime.Now.Ticks / 10000));
        public double AverageRandom(double min, double max) {
            int MINnteger = (int)(min * 10000);
            int MAXnteger = (int)(max * 10000);
            int resultInteger = aa.Next(MINnteger, MAXnteger);
            return resultInteger / 10000.0;
        }
        public double Normal(double x, double miu, double sigma) {
            return 1.0 / (x * Math.Sqrt(2 * Math.PI) * sigma) * 
```
Math.Exp\((-1 \cdot (\text{Math.Log}(x) - \text{miu}) \cdot (\text{Math.Log}(x) - \text{miu}) / (2 \cdot \sigma \cdot \sigma))\);

```java
public double Random_Normal(double miu, double sigma, double min, double max)
{
    double x;
    double dScope;
    double y;
    do
    {
        x = AverageRandom(min, max);
        y = Normal(x, miu, sigma);
        dScope = AverageRandom(0, Normal(miu, miu, sigma));
    } while (dScope > y);
    return x;
}
```

```java
public double RandExp(double const_a) // const_a is the value
{
    Random rand = new Random(Guid.NewGuid().GetHashCode());
    double p;
    double temp;
    if (const_a != 0)
        temp = 1 / const_a;
    else
        throw new System.InvalidOperationException("error happen");
    double randres;
    while (true)
    {
        p = rand.NextDouble();
        if (p < const_a)
            break;
    }    
    randres = -temp * Math.Log(temp * p, Math.E);
    return randres;
}
```

```java
Random ran;
public calculate()
{
    ran = new Random();
}
```

```java
public double ngtIndex(double lam)
{
    double dec = ran.NextDouble();
    while (dec == 0)
        dec = ran.NextDouble();
    return -Math.Log(dec) / lam;
}
```

```java
public double poisson(double lam, double time)
{
    int count = 0;
    while (true)
    {
        time -= ngtIndex(lam);
        if (time > 0)
            count++;
        else
```
D.4 C# code that used to generate dynamic tasks Part B

```csharp
namespace DistributionData
{
    class Model
    {
        static void Main(string[] args)
        {
            List<int> dat = new List<int>();
            int b = 0;
            int a = 0;
            calculate cal = new calculate();
            Console.WriteLine("there is no problem about this program!");
            // standard distribution
            while (b < 500)
            {
                // a = (int)cal.Random_Normal(40, 1, 40, 90);
                // dat.Add(a);
                // b++;
                //
                // Exponential Distribution
                // while (b < 500)
                // {
                //     // a = (int)cal.RoundExp (0.04);
                //     // if (40 < a &a < 90)
                //     //     {
                //     //         dat.Add(a);
                //     //         b++;
                //     //     }
                //     //
                // }
                //
                // Poisson Distribution
                while (b < 500)
                {
                    a = (int)cal.poisson(0.18, 400);
                    if (40 < a &a < 90)
                    {
                        dat.Add(a);
                        b++;
                    }
                }
                foreach (int c in dat)
                {
                    Console.WriteLine(c);
                }
            }
        }
    }
}
```