



저작자표시-비영리-동일조건변경허락 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.
- 이차적 저작물을 작성할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



동일조건변경허락. 귀하가 이 저작물을 개작, 변형 또는 가공했을 경우에는, 이 저작물과 동일한 이용허락조건하에서만 배포할 수 있습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

Master's Thesis

Scheduling of Tasks on
Componentizing Kernel During
Evolution of Tasks

August 2015

Graduate School of Seoul National University
Computer Science and Engineering
Bezabh Yonas Kidanemariam

Scheduling of Tasks on Componentizing Kernel During Evolution of Tasks

Professor Chang-Gun Lee

Submitting a master's thesis

August 2015

Graduate School of Seoul National University

Computer Science and Engineering

Bezabh Yonas Kidanemariam

Confirming the master's thesis written by___

August 2015

Chair Bernhard Egger (seal)

Vice Chair Chang-Gun Lee (Seal)

Examiner Srinivasa Rao Satti (Seal)

Abstract

Scheduling of Tasks on Componentizing Kernel During Evolution of Tasks

Bezabeh Yonas

School of Computer Science and Engineering

The Graduate School

Seoul National University

This thesis proposes how to find a feasible schedule with minimum system degradation during the evolution of the software components on Componentizing kernel. On the componentizing kernel, the software component can expand and their system utilization can be increased at run time. This might lead to the system to be un-schedulable. We proposed a method to reschedule the software components at run time by changing their offset (by increasing the release time). But increasing the release time of the software components might cause performance degradation of the system. We maintained the performance of the system by introducing weight for each transaction. To find the best solution in short period of time, we developed a heuristic algorithm.

Keywords : Componentizing Kernel, Software
Evolution, Transaction

Student Number : 2013-23849

Contents

1. Introduction	1
2. Related Work	6
3. Background and Problem Description	9
3.1 Background	9
3.2 Problem Description	11
4. Proposed solutions	14
4.1 Exhaustive Search	17
4.2 Heuristic approach	17
5. Experiment.....	27
6 Conclusion and Future Work	32
7. References	33

List of Figures

Figure 1: Illusion of Physically isolated ECU with SW component..	2
Figure 2: Normal real-time system	3
Figure 3: Componentizing Kernel	3
Figure 4: Transactions $\{\Gamma_i, \Gamma_j\}$ and SW component mapping with ECU	10
Figure 5: Mapping the software component on ECU resource	10
Figure 6: SW component expansion at run time	13
Figure 7: SW components before and after evolution	13
Figure 8 a,b: Example of possible schedules by changing the offset of transaction	16
Figure 9: Comparing the degradation value of the systems	28
Figure 10: Time complexity of the Heuristic and Exhaustive approach	29
Figure 11: Comparing Heuristic approach with Exhaustive search by finding the solution for any given system	31

List of Tables

Table 1: All feasible schedule	19
Table 2: Schedulable	22
Table 3: System degradation	25

1. Introduction

The vehicle system is complex as modern cars have 60 to 70 ECU (Electronic Control Unit) components with 5 communication networks [1]. Despite the complexity, the automaker wants to include more facilities to maximize the safety and the comfort for their customers. However, an ECU component performs only one specific task [2]. As a result of this, automaker needs to add additional ECU components for every additional facility. Adding a new ECU for every additional facility is not feasible, as the ECU and the network to integrate the ECU will increase the cost, occupy space and increase the vehicle weight.

To solve this problem, different research work has been going on. One of the studies related to this issue is using multi-core ECU [1], [3], [4]. Multi-core ECU handles more than one critical task at a time [1]. However, the current software design in automotive industry doesn't support multi-core ECU, particularly the multi-core scheduling in the open research area. The current automotive industry uses a single-core ECU. As a result, there is physical isolation of hardware components from the corresponding software components. Therefore, we should use one ECU for each software component. To narrow this gap, a componentizing based kernel has been developed [5]. It uses a single ECU for multiple tasks by componentizing the ECU resource [5].

It assigns invariant delay to each software component to create an illusion that each software component has independent hardware resources, as we can see from Figure 1. The software component

1(sw1), software component 2 (sw2) and software component 3(sw3) shared one ECU component. However because of componentizing kernel, the software components look like they access the ECU independently.

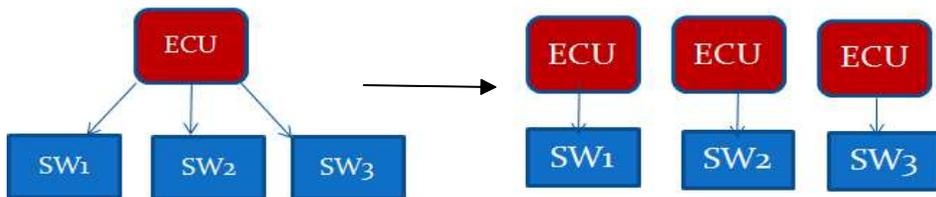


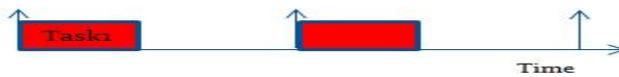
Figure1: Illusion of Physically isolated ECU component with SW component

On componentizing kernel the hardware share S for the software components SW is scheduled by constant bandwidth server (CBS). If the CBS are scheduled by EDF and the sum of their shares is less than 100% each CBS's cumulative share can be guaranteed before the associated deadline. Therefore, we can add software components to share the hardware component if the summation of their utilization is less than 100%.

Unlike normal real-time operating system in componentizing kernel, each software component physical properties, like time change is invariant to the surrounding software component. As we can see from Figure 2, in a normal real-time operating system, task1 is preempted by task 2 if it has higher priority than task 1. As a result, task 1 is delayed by task 2. As we can see from Figure 3, software component 1 is not delayed by software component 2 because in a componentizing kernel, the software

component has physical properties invariant with the surrounding software component.

1 task



2 tasks

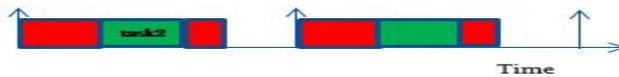
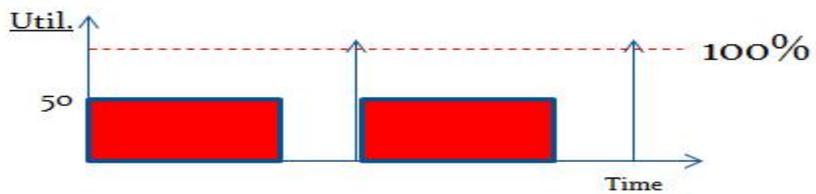


Figure 2: Normal real-time system

1 SWC



2 SWCs

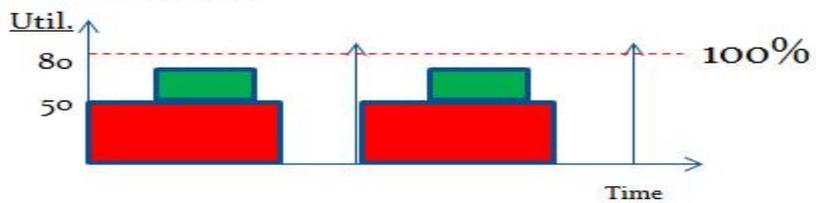


Figure 3: Componentizing Kernel

Even if the software component is not affected by the surrounding environment, the software component can evolve and make the system un-schedulable. This might cause a disaster to the Cyber Physical System (CPS). The automotive system is a mission critical system and we should give a guarantee that requires each task to meet its deadline. For these reasons, we are motivated to develop a method to reschedule the software components after the evolution of one or more software components in the system to meet their deadline. However, there are some tasks which are very critical and there are other tasks which are not critical in the automobile system. If the tasks are critical, the tasks should meet their deadline, or else it will cause a catastrophic effect to the system. But there are tasks which are not critical and even if we couldn't meet their deadline, it will not cause a catastrophic result. We can postpone such kind of tasks. But, postponing such kind of tasks might cause system degradation. Therefore, we give a weight for each task. We use this value during rescheduling of the tasks (software components).

The software evolution is an action of system dynamics which make the software system continuously maintained and improved. If one or more parts of the system changes throughout time, the process is a software evolution [6]. The purpose of the evolution is to make the system adjust to the change in the environment. Then, the user will be more satisfied with the system. There are several reasons for software evolution fixes errors, enhances functionality,

and improves (overall) performance. Two decades ago, software needed to be corrected occasionally and have a new release issued approximately once a year. One decade ago, software needed a major release twice a year. Today, software needs to be changed on an ongoing basis with a major enhancement taking place within days or weeks rather than months or years.

The purpose of this thesis is to propose a rescheduling mechanism for componentizing kernel during the software component evolution.

The rest of this thesis is organized as follow: we survey the related works of the evolution of software component concept with on the fly evolution, real-time system evolution and resource utilization issues in Section 2. The background and the problem formulation will be described in Section 3. In Section 4, we are going to propose a mechanism to find the feasible schedule with minimum system degradation by the exhaustive search and heuristic approach. Section 5 contains the experiment result of the two approaches and their comparison. The Conclusion and future works are described in Section 6.

2. Related Work

There are several studies that have taken place in the area of software evolution. There are research works in monitoring software evolution. These research works are important to identify the rate of the software evolution in the given system for example in linux kernel. This study helps to predict the future software evolution rate and this will be a useful input to design the proper design for the future software evolution.

To monitor software evolution, it requires to use measurement. There are research works which monitored the evolution of the software components by number of line of codes [7] others by using module count. However, this research works unlike to our research work they are not supporting real-time system or on the fly evolution (updating the system without stoping).

Qian Zhao et al. [8] want to consider the histories of evolution behavior of the software to analyze the present state and to predict its future development. This research also does not support on the fly evolution of software components.

YingHui et al. [9] designed a framework for software evolution based on the object-oriented paradigm. By giving out a detailed analysis about the process of software change transmission and implementation during software lifecycle, they proposed a software evolution framework. Software change starts from requirements changes in software lifecycle. Therefore, based on the object-oriented technology, the change transmits in the following

order: scenario, use case, object, component and software architecture. They proposed the ontology system to understand and describe requirement in the stage of requirement analysis. The process of understanding the software evolution requirement in semantic way and able to transmit this change in the software lifecycle, lead to a dynamic software evolution. However, they didn't consider real-time system in their research.

Regarding the real-time system evolution, there is a research work which shows how the real-time system evolves by using organic programming [10].

Since the evolution of real-time system response to environmental changes, it becomes more difficult for the system to adapt itself and manage its stability at run time. But organic programming concept enables the real-time operating system to be able to adapt itself to the new circumstances and to manage its data in order to preserve all real-time constraints. To archive this goal they propose tasks in real-time application to behave like objects do in real world. Objects in real world adapt to the environment and they change their behavior according to a set of influential factors. Similarly, there are many situations that a real-time application modification of behavior or structure is needed as a result of task update or arrive.

However the research work didn't show how the hardware resources are shared by the software component and how the hardware resource constraint will be kept .

Therefore, the research work is not suitable for small embedded systems. However, in our research work we considered the

constraint of the hardware resource in addition to the evolution of software components in real-time system. Therefore, we want to utilize the hardware resource efficiently during the evolution of software component by keeping real-time constraints.

There is a research work on evolution of software component regarding to the hardware resources for embedded system [12]. This research proposed a method to update the hardware during the evolution of the software component and to update the software when the hardware is updated. Unlike to our approach this approach it is not on the fly.

3. Background and Problem Description

In this chapter, we describe the background of the research work and we formulate the research problem.

3.1. Background

Automaker's engineers design end-to-end control transactions from sensors to actuators using the traditional model-based design method. As a result of such design, each end-to-end control transaction Γ_i is generally given by a directed acyclic graph (DAG) G_i , consisting of $|\Gamma_i|$ software components $\{C_{i1}, C_{i2}, \dots, C_{i|\Gamma_i|}\}$.

As we can see from Figure 4, each transaction Γ_i should be triggered periodically in a period, P_i . Then, each successive software component triggered by the event in its input port will produce an event on its output port or trigger the next software component within the deadline. Eventually, the final SW component in the (DAG) G_i should be completed before the end-to-end deadline D_i . As a consequence, a transaction Γ_i is represented by a three-tuple (G_i, P_i, D_i) .

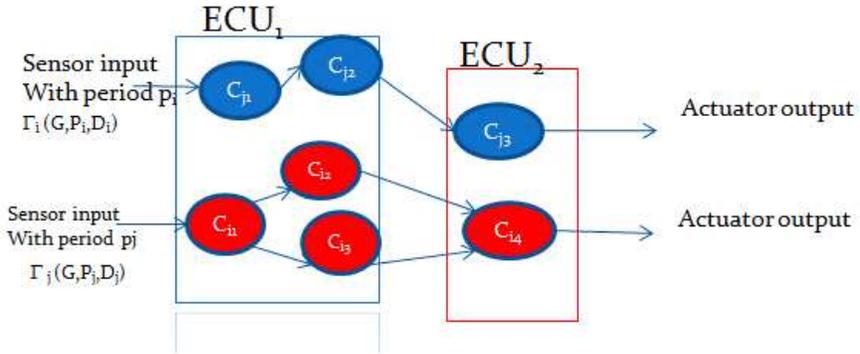


Figure 4: Transaction $\{\Gamma_i, \Gamma_j\}$ and SW component mapping with ECU

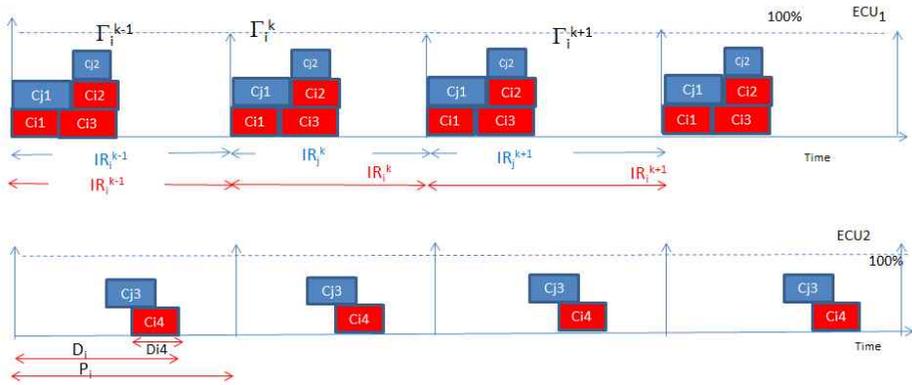


Figure 5: Mapping the software component on ECU resource

Γ_i^{k-1} represent transaction “i” at the state of k-1, Γ_i^k represent transaction “i” at the state of k and Γ_i^{k+1} represent transaction “i” at the state of k+1.

Each transaction has a weight (V). This value is given and we use this value to prioritize the transactions during rescheduling of the evolved system. As we can see from Figure 5, the software components that are found in one transaction can

be executed in one or more ECU resources. But the software utilization in the corresponding ECU resources shouldn't be greater than 100%.

Each transaction has end to end delay (D_i) and inter arrival time (IR) which is the time difference between the previous transaction and the next transaction. It can change during the software evolution. But the period will not be changed during the evolution of the software components. On the Figure 4, we have 2 ECU components with two transactions and each transaction has three or four software components.

Each software component expressed in a componentizing kernel, in the Y-axis share of the software component utilization and in the x-axis the execution time of the software components.

3.2. Problem Description

The software component in componentizing kernel will not be affected by another software component that is found in another transaction. But the software component could be expanded or evolved during the system upgrade. Therefore, if the software components evolve, their utilization might be increased and would be more than 100%, which means the system will be un-schedulable. If we keep running the system without rescheduling the software components, it might cause a disaster.

The Automotive system is a very critical system and if one of the critical software components couldn't meet its deadline, it might risk human life. Therefore, we want to make sure that after the software component evolves, we should be able to reschedule the system and all software components meet their deadline with minimum system degradation or cost.

We have two choices to reschedule the software component. We can reschedule the system by stopping the system or we can reschedule while the system is running. The cyber physical system is real-time system and we want the system to be online while we are updating the system. Therefore, we proposed a mechanism to reschedule the system during the evolution of the software components on a componentizing kernel without stopping the system.

To archive our goal, we changed the offset (the release time) of the transactions to make sure the utilization of the transactions is under 100%. As we can see from Figure 6, after evolution the utilization of the two transactions is more than 100%, therefore, the system will not be schedulable. As a result, we should reschedule the system by changing the offset of the transactions. But increasing the offset will cause the system performance to be degraded. Therefore, we should take this into consideration before increasing the offset value.

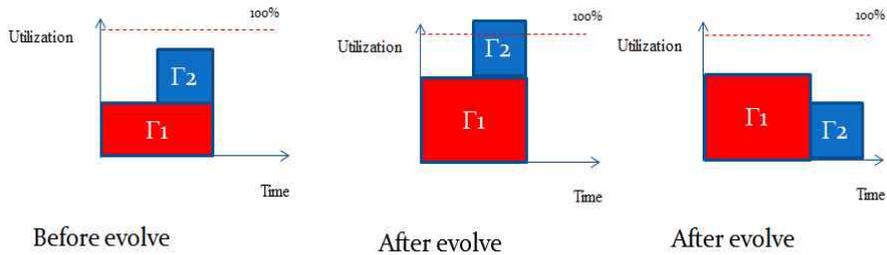


Figure 6: SW component expansion at run time

Γ_1 is transaction one and Γ_2 is transaction two.

The soft real-time system is less restrictive [11]. If certain deadlines are missed the system performance will be lower. However, the system will continue to operate. Therefore, by meeting the deadline of the most priority task in the system we can increase the overall performance of the system. To do that it is required that the critical processes receive higher priority over less critical ones.

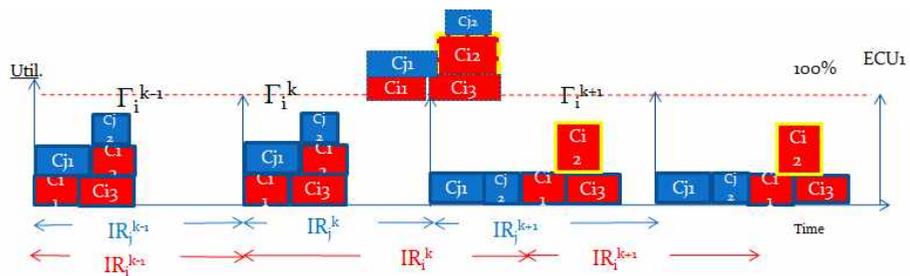


Figure 7: SW components before and after evolution

Figure 7 explains about the evolution of software components in the same transaction. During evolution of a software component in a transaction, if the evolution makes the system to be un-schedulable, we should change the offset of the software component which is evolved. However, to keep the end to end delay of the transaction for all software components that are found in the transaction, they should move together by the same value in the same direction with the evolved software component.

We will give the same weight to the software component in the same transaction. Because, we cannot change one software component in a transaction without affecting another software component in the same transaction.

4. Proposed Solution

We could find the feasible schedules by changing the offset of the transactions, as we can see from figure 6. However, we could have many feasible schedules. To select the best feasible schedule from the possible schedules, we need to calculate each feasible schedule degradation value. The degradation value will be calculated as follow.

The example given in Figure 8, there are 2 transactions, “i” and “j”. Transaction “i” has three software components and

transaction “j” has two software components. Then to find the degradation value of figure 8.a (DV1), we can use the following formula.

$$DV_1 = (\text{Max}(\text{IR}_j^k - \text{IR}_j^{k-1}), 0)V_j + (\text{Max}(\text{IR}_i^k - \text{IR}_i^{k-1}), 0)V_i$$

We get the positive or zero value of the inter arrival time difference of the schedules after the software component (ci2) evolves. Then, we add the multiplication of this inter arrival time with the corresponding transaction weight.

We did the same procedure as we did above to find the degradation value of (DV2) for the second feasible schedule figure 8.b,

$$DV_2 = (\text{Max}(\text{IR}_j^k - \text{IR}_j^{k-1}), 0)V_j + (\text{Max}(\text{IR}_i^k - \text{IR}_i^{k-1}), 0)V_i$$

The following variables are used in the above formula.

- IR_j^k : The inter arrival time after the evolution of software components.
- IR_j^{k-1} : The inter arrival time before the evolution of software components.
- $(\text{Max}(\text{IR}_j^k - \text{IR}_j^{k-1}), 0)$: It calculates a positive or zero inter arrival time difference for transaction “j” before and after the evolution of software component (ci2).
- $(\text{Max}(\text{IR}_i^k - \text{IR}_i^{k-1}), 0)$: It calculates a positive or zero inter arrival time difference for transaction “i” before and after the evolution of software component (ci2).
- V_j is the weight value for transaction “j”.
- V_i is the weight value for transaction “i”.
- DV_1 : it is the degradation value for feasible schedule figure 8.a.
- DV_2 : it is the degradation value for feasible schedule of

figure 8.b.

After finding all the degradation value for all feasible schedules, we will take the minimum degradation value as the best feasible schedule for our solution. In our case, we have two feasible schedules so we have two degradation values. Therefore, we select the minimum degradation value as a solution $\text{Min}(DV_1, DV_2)$.

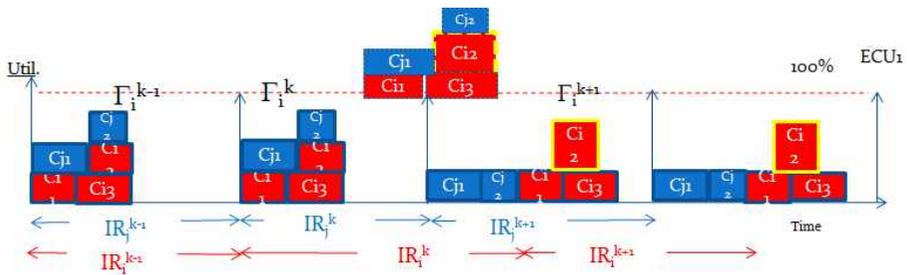


Figure 8.a :Example of possible schedules by changing the offset of transaction

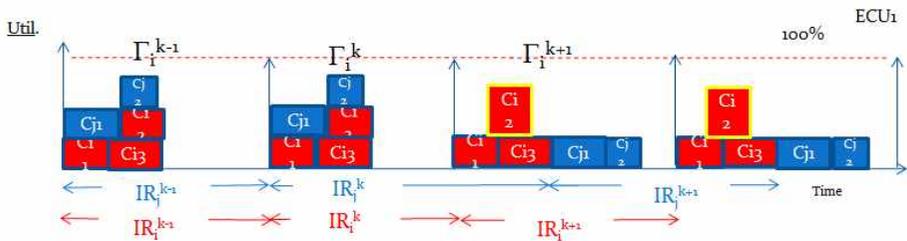


Figure 8.b : Example of possible schedules by changing the offset of transaction

If we have many transactions and software components, it is not easy to find all feasible schedules after the software component is evolved. Therefore, we need to have a mechanism to find all feasible schedules. We can use exhaustive search to find these feasible schedules. However, it is not practical as it takes long time to find the feasible schedules. Therefore, we developed a heuristic approach.

4.1 Exhaustive search

The exhaustive search will check all possible combinations of the offset values of the software component which will make the system schedulable. This approach is like a bin packing problem which is NP-Hard problem. This approach will find the best solution for sure since we will search exhaustively whole the solution space. We use this approach in our experiment to verify the correctness of our Heuristic solution.

4.2 Heuristic approach

In the heuristic approach, we first sort the transactions based on their weight in decreasing order. Then, we fix the search space of each transaction by limiting the search space. To limit the search space, we use the offset value instead of the period value in the search space. To do that we will follow the

following approach.

The transaction with highest weight, its search space will be fixed by adding 0 to its offset. The next transaction with the higher weight, its search space will be fixed by the summation of its offset with one. We will fix the search space for the other transactions in decreasing order of the transaction weigh by adding their offset with an integer number which increments from zero to the period.

The algorithm to find the feasible offset values is as follow.

Table 1 All feasible schedule

Input: Transaction $\{\Gamma_1, \Gamma_2, \dots, \Gamma_n\}$, offset of transactions $\{\text{offsetinput}[1], \text{offsetinput}[2], \dots, \text{offsetinput}[n]\}$, period $\{P\}$

Output: All possible offset values $\text{schedule}[\cdot].\text{offsetoutput}[m]$

begin procedure

1. Sort $\{\Gamma_1, \Gamma_2, \dots, \Gamma_n\}$ as the decreasing order their weight
2. $M =$ multiplication of (Number of transaction * P)
3. $R =$ Multiplication of ($\text{offsetinput}[1], \text{offsetinput}[2], \dots, \text{offsetinput}[n]$)
4. struct fes{ $\text{offsetoutput}[n]$ }
5. fes schedule [M] , feschedule [R]
6. $a=0, i=0, n$ is initialized by the number of transaction in the system
7. do Recursive ($\text{offsetinput}[\cdot], \text{offsetoutput}[\cdot], n, \text{index}, P$)
8. if $\text{index} == n$
9. $a++$
10. for $m=0$ to n
11. $\text{schedule}[a].\text{offsetoutput}[m] = \text{offsetinput}[m]$
12. end for
13. end if
14. else
15. for $j=0$ to P
16. $\text{offsetoutput}[i]=\text{offsetinput}[j]$
17. do recursive ($\text{offsetinput}, \text{offsetoutput}, n, (i+1), P$);
18. end for

```

19.   end else
20.   end recursive
21.  s=0
22.  for b=0 to M
23.    for g=0 to P
24.      If (schedule[b].offsetoutput[g] < offsetinput[g] + g )
25.        fchedule[s].offsetoutput[g]=schedule[b].offsetoutput[g]
26.        s++
27.      End If
28.    End for
29.  End for
30.

```

end procedure

Description for table one

Line 1 Sort the transaction based on their weight. Line 2 ,we calculate array size of M which is a multiplication of the period with the number of transaction n. line 3 ,we find the array size of R which is a multiplication of the number of offset values in each transaction. Line 4 and 5, we created a structure fes with array of object schedule [M] and fesschedule[R] with array of object schedule [R]. Line 6 , initialized the value of “i” and “a”

From line 7 to 20, it is a recursive function to find the transactions period combination with repetition.

Line 22 to line 29 select the feasible schedule based on the transaction weight. The transaction which has higher weight its search space will be limited up to the initial offset value and the remaining transaction search space will be limited by the addition of an increasing integer number from 1 to P with corresponding transactions offset value.

The above algorithm is to find the feasible offset values. To know if the offset values are schedulable or not we will use the next algorithm.

Table 2: Schedulable

Input: Transaction $\{\Gamma_1, \Gamma_2, \dots, \Gamma_n\}$, $R =$ Multiplication of (offset[1], offset[2], ..., offset[n]), All possible feasible schedules offset feschedule[].offsetoutput[], number of software component in each transaction {transaction[sw]}, the utilization of the software components in the transaction $\{\Gamma[i][j].utilization\}$, execution time of the software component in the transaction execution[], and period

Output: All feasible schedules offset: feasible[].offset[]

begin procedure

1. for m to R
2. for i to n
3. for j=0 to i[sw]
4. $\Gamma[i][j].window1 = feschedule[m].offsetoutput[i]$
5. $\Gamma[i][j].window2 = \Gamma[i][j].window1 + execution [j]$
6. end for
7. for t = 0 to period
8. totalutilization=0
9. if $\Gamma[i][j].window1 < t$ and $t < \Gamma[i][j].window2$
10. totalutilization=totalutilization+ $\Gamma[i][j].utilization$
11. end if
12. if (totalutilization <100)
13. u++
14. end if
15. end for
16. if (u== period +1)
17. feasible[].offset[] =feschedule[].offsetoutput[]
18. end if
19. u=0

20. end for

21. end for

end procedure

Description for table 2

Line 1, a loop for the feasible schedules.

Line 2, a loop for number of transaction.

Line 3, a loop for the number of software components in each transaction.

Line 4, we assign the window1 value by the offset of the transaction for each transaction.

Line 5, we assign the window two value of each software component in a transaction as the summation of window 1 value of each software component in a transaction with the execution time of the software component.

Line 6, end of for loop.

Line 7, a loop for the period.

Line 8, initialized the totalutilization to zero.

Line 9, checking if the software components in each transaction could be schedule with in the period.

Line 10, summation of the utilization of the software components in any transaction at a given time.

Line 11, end of if statement

Line 12, checking the utilization less than 100. If the utilization is greater than 100, we will not increase the value of “u“ which

means the given schedule will not be considered as the feasible schedule.

Line 13, increase the value of u.

Line 14, end of if statement,

Line 15, end of for loop.

Line 16, checking the utilization if it is less than 100 for whole time in a given period.

Line 17, selecting the feasible schedules from the given schedule.

Line 18, end of if statement.

Line 19, initialized the value of “U” to zero after the complete period.

Line 20 and Line 21, end of for statement.

The following algorithm is to find the minimum degradation value.

Table 3: System degradation

Input: offset value of the feasible schedules
feasible[i].offset[i], offset values of transaction before the
evolution of software component {preoffset[1], preoffset[2],
.... preoffset[n]}, Critical value for transaction Γ_n { V_1, V_2
, V_3 V_n } ,

Output: Minimum system degradation (min)

begin procedure

1. min= ∞
2. for i=0 to R do // number of feasible schedules
3. for n=0 to m do // number of transactions
4. DVT[n] = (Max((feasible[i].offset[n] - preoffset[n]) , 0)) * V[n]
5. deg+= DVT[n]
6. end for
7. if (i==1)
8. min=deg
9. end if
10. if (min > deg)
11. min=deg
12. end if
13. end for

end procedure

Description for the table 3

Line 1, initialized the minimum value with a very big number.

Line 2, loop for the feasible schedules.

Line 3, loop for the transactions.

Line 4, find the degradation value for one transactions ($DVT[n]$) in a feasible schedule.

Line 5, find the summation of the degradation values of the transactions in one feasible schedule.

Line 6, end of for loop.

Line 7, checking if there is only one transaction

Line 8, if the transaction is one we take the first degradation value as the final solution.

Line 9, end of if statement.

Line 10, comparing the minimum (min) with the degradation value(deg).

Line 11, If the minimum value is greater than the degradation value, replace minimum value by assign degradation value.

Line 12, end of if statement.

Line 13, end of for loop.

5. Experiment

We evaluate the proposed approaches through simulation. We make a set of transactions with the following parameters.

- We use from 2 to 7 system controlled transactions.
- Each transaction could have from 3 to 5 software components.
- Each software component has a share in the ECU resource. The share of SW component is randomly determined from 10% to 50%.
- Each transaction should keep its end to end delay when we find the possible feasible schedules.
- We implemented the system by using two ECU resources. We implemented our system for two ECU component due to the following reason. The software component are distributed to the ECU components on componentizing kernel. Implementing a system for two ECU component is almost similar to implementing for n number of ECU except the software component will be distributed to n number of ECU. Therefore, it is easy to extend the number of ECU to n number of ECU based on the requirement of a system.
- we generated 100 different systems which are dynamically generated for both exhaustive search and heuristic approach.

We simulate the exhaustive search and our heuristic approach to find the feasible schedule with minimum system degradation. Our objective is to find the schedulable system with minimum

system degradation after the evolution of any software component in one of the transactions. We take average of the degradation value of the 100 systems in our experiment.

Figure 9 compares the degradation value of the exhaustive search with the heuristic approach. The exhaustive search value after transaction four is better than the heuristic approach.

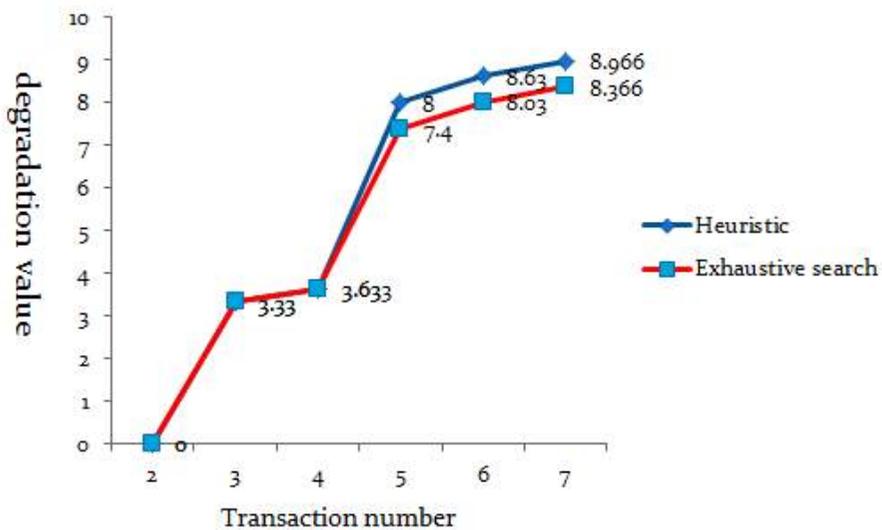


Figure 9: Comparing the degradation value of the systems

Even if the exhaustive search finds better result compared to the heuristic approach after transaction four, it takes very long time to find the feasible schedule with minimum system degradation.

Figure 10 compares the time complexity of the two approaches.

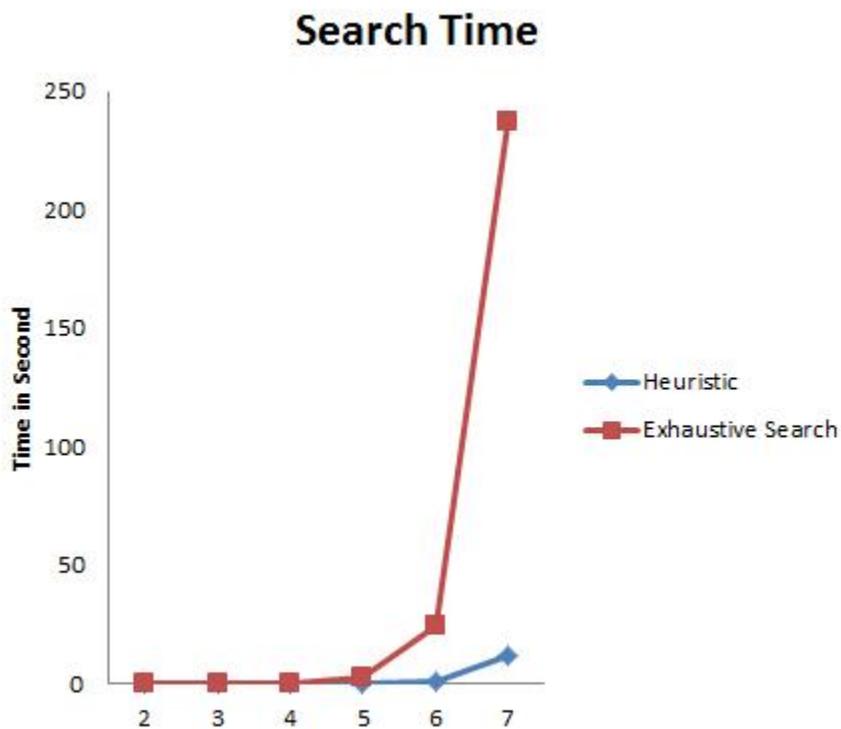


Figure 10 Searching time of the Heuristic and Exhaustive approach

When the transaction increased the exhaustive search takes long time to find the solution but the heuristic approach takes comparatively short time to find the solution. Therefore, it is

more practical to find the feasible schedules with minimum system degradation during the evolution of the software component on Componentizing kernel.

we took another experimental setup to check how many systems wouldn't have a solution in our heuristic approach while they have a solution in exhaustive search.

- We use from 2 to 7 system controlled transactions.
- Each transaction could have from 3 to 5 software components.
- Each software component has a share in the ECU resource. The share of SW component is randomly determined from 10% to 50%.
- Each transaction should keep its end to end delay when we find the possible feasible schedules.
- We implemented the system by using two ECU resources.
- We generated 100 different systems which are dynamically generated for both exhaustive search and heuristic approach. We count how many dynamically generated system has a feasible schedule in exhaustive search but have no feasible schedule using our heuristic approach.

Figure 11, shows how many evolved system couldn't have a feasible a solution in our heuristic approach while the exhaustive search could find a solution.

As we can see from Figure 11 our heuristic could find solutions for all evolved systems up to transaction six but for transaction seven ,our heuristic couldn't find a solution for about 5% of the evolved system however the exhaustive search could find a solution.

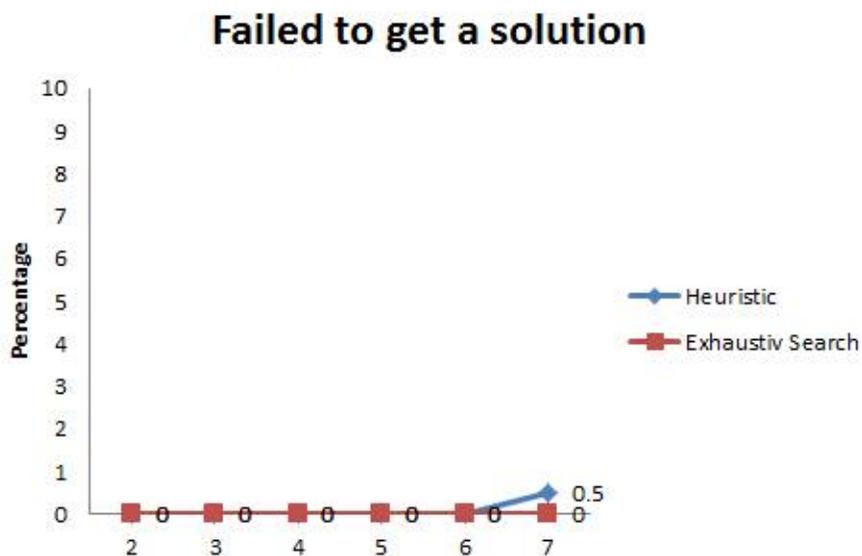


Figure 11 Comparing Heuristic approach with Exhaustive search by finding the solution for any given system

6. Conclusion and Future Work

This thesis proposed an approach to evolve the software component on Componentizing kernel. In addition, it presents how we can find the best schedule which has minimum system degradation during the evolution of the software components. It also proposed the evolution of software component on real-time system with limited resource during the evolution of the software components.

In the future, we are planning to further optimized our heuristic approach. Our heuristic approach couldn't find a solution for 5% of the evolved system when the number of transaction is seven. But we can minimized or avoid this problem by increasing the search space whenever the heuristic approach couldn't find a solution.

7. References

- [1] Abinesh S., Kathires M. and Neelaveni.R., “Analysis of Multi-Core Architecture for Automotive Applications,” In IEEE Embedded Systems (ICES) International Conference, pp 76 - 79, 3-5 July 2014,
- [2] Jian HU, Gangyan LI and Jun XU, ”Component-Based ECU Design Method of Passenger Car Information Integrated Control System,” In International Conference on Automation and Logistics Shenyang China, pp 365-370, August 2009.
- [3] Thomas Herpel and Reinhard German, “A Simulation Approach for the Design of Safety-Relevant Automotive Multi-ECU Systems,” In SoSE, pp 1-8,2009.
- [4] Santosh Kumar Jena and M. B Srinivas, ”On The Suitability of Multi-Core Processing for Embedded Automotive Systems,” In CVEST, IIIT Hyderabad, pp 315-322 ,2012
- [5] Jong-Chan Kim, Kyoung-Soo We, Chang-Gun Lee, Kwei-Jay Lin and Yun Sang Lee, “HW Resource Componentizing for Smooth Migration from Single-function ECU to Multi-function ECU,” In SAC’12, pp 1821-1828, Mar., 2012.
- [6] Hongji Yang, Martin Ward, “Successful Evolution of Software Systems”, 2003.
- [7] Michael W. Godfrey and Qiang Tu, “Evolution in Open

- Source Software: A Case Study,” In software maintenance, pp 131 - 142, 2000.
- [8] Qian Zhao, Huiqiang Wang, Guangsheng Feng and Xu Lu, “Software Evolution Method Considering Software Historical Behavior,” In Internet Computing for Science and Engineering Evolutionary Computation IEEE , pp.36-41, 2009.
- [9] YingHui Wang, XiuQing He and QiongFang Wang, “Lifecycle based Study Framework of Software Evolution,” Evolutionary Computation, IEEE Transactions on, vol.7, no.2, pp.204,223, April 2003
- [10] Franz-Josef RammigLial Khaluf, Norma Montealegre, Katharina Stahl and Yuhong Zhao, “Organic Real-time Programming -Vision and Approaches towards Self-Evolving and Adaptive Real-time Software, “In IEEE ISORC, pp 1-8, 2013
- [11] Nilabja Roy, Nathan Hamm, Manish Madhukar, Douglas C. Schmidt and Larry Dowdy,” The Impact of Variability on Soft Real-Time System Scheduling,” In IEEE Conference on Embedded and Real-Time Computing Systems and Applications, pp 527-532, 2009
- [12]. Brian Dougherty, Jules White, Chris Thompson, and Douglas C. Schmidt, “Automating Hardware and Software Evolution Analysis,” In IEEE ECBS, pp 265-274,2009.

