공학석사학위논문

A Component-Level Simplex Architecture based Framework to Enhance Reliability of Cyber-Physical Systems

사이버-물리 시스템의 신뢰도 향상을 위한 콤포넌트 수준 심플렉스 구조 기반의 프레임워크

2013년 8월

서울대학교 대학원

전기 · 컴퓨터 공학부

정 상 민

A Component-Level Simplex Architecture based Framework to Enhance Reliability of Cyber-Physical Systems

지도교수 이 창 건

이 논문을 공학석사학위논문으로 제출함

2013년 8월

서울대학교 대학원

전기 · 컴퓨터 공학부

정 상 민

정상민의 석사학위논문을 인준함

2013년 8월

위 원 장 조 유 근 (인)

부 위 원 장 이 창 건 (인)

위 원 하 순 회 (인)

Abstract

A Component-Level Simplex Architecture based Framework to Enhance Reliability of Cyber-Physical Systems

Sangmin Jeong
School of Computer Science and Engineering
The Graduate School
Seoul National University

Along with growing complexity and magnitude of Cyber-Physical Systems (CPSs), it is getting hard to guarantee their reliability. In this paper, we propose a novel framework that guarantees the reliability of CPSs by providing component-level Simplex architecture. The system-level reliability problem of CPSs is reduced to component-level problem when applying temporal isolation to each software component. Then, we can apply component-level simplex architecture, an approach for fault-tolerant for softwares. However, simplex architecture is based on redundancy of computations, so it requires additional

hardware resources. In order to overcome this resource requirement problem while guaranteeing system reliability, we propose a scheduling scheme consisting of speculative schedule optimization technique and runtime victim selection algorithm. This scheme enables CPS to guarantee system reliability, utilize hardware resources efficiently, and maximize operations of advanced functionalities. Experimental results confirm that the proposed scheme accepts 14% to 70% more transactions than baseline approach.

Keywords : **Cyber-Physical Systems, Simplex Architecture, Software fault-tolerance, Reliability, Schedule Optimization, Victim Selection Algorithm**

Student Number : 2011-20928

Contents

List of Figures

List of Tables

1. Introduction

Cyber-Physical Systems (CPSs) integrate physical world and cyber world by repeatedly detecting properties of physical world and performing reactions according to control algorithm executed in cyber world. Figure 1 describes a general concept of CPS. The system is composed of sensors and actuators on physical side and controllers on cyber side that are networked. Applications of CPS are various and numerous including traffic control system, smart greed, surgical operation robot, and autonomous automotive system. As CPSs become more advanced, they involve increasingly complex structures and complicated control algorithms. Therefore, it is important and difficult to guarantee system's reliability. By safety-critical nature of CPS, we need a way to operate system without catastrophical system failure even when control algorithm produces an error.

However, due to high complexity and magnitude of advanced CPSs, it is practically infeasible to confirm fault-tolerant operation of CPS at system-level for all scenarios. Recently, [2] proposed an approach to address high complexity of CPS, but it did not consider system reliability. [4] and [6] proposed fault-tolerant approaches, but [4] is based on impractical assumptions and in [6], system reliability is not always guaranteed. In this paper, we propose a component-level Simplex

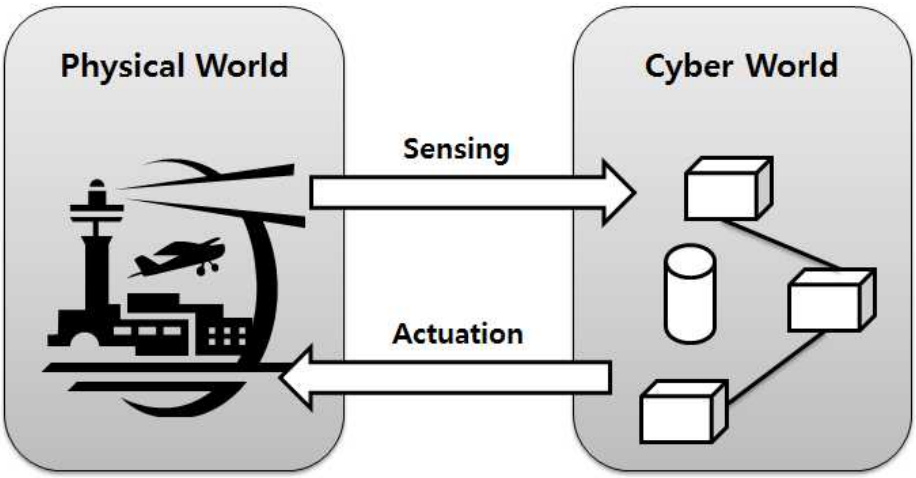


Figure 1. Cyber-Physical System

architecture based framework to ensure reliability of CPSs.

System-level reliability problem of CPS can be reduced to component-level problem by applying temporal isolation method proposed in [2]. This approach assigns invariant delay to each software component to create an illusion that each software component has independent hardware resources. After temporally isolated, software components' failure is also independent of one another. Therefore, system-level reliability can be guaranteed by independently guaranteeing reliability of each software component of the system.

To guarantee reliability of a software component, we adopt a dual version programming concept named Simplex architecture [3]. In Simplex architecture, each software component is composed of high-performance version and high-assurance

version. High-performance version provides advanced functionalities by means of more complicated algorithms than high-assurance one. As a result, it is difficult to verify whether high-performance version will be successfully executed. On the contrary, high-assurance version provides minimal functionalities for the system with simpler algorithms. Hence, it is possible to test high-assurance version for every scenario. Reflecting these properties, Simplex architecture outputs high-performance version's execution results by default and outputs high-assurance version's results when high-performance version has failed during execution. In this way, the component will always produce a reliable output, and the output is associated with advanced functionalities whenever possible.

Although Simplex architecture may ensure system reliability, it leads to another problem of additional resource requirements. Basic Simplex architecture executes both high-performance and high-assurance versions in preparation for the case where high-performance one fails. To overcome this hardware resource problem while guaranteeing overall system reliability, we propose a scheduling algorithm that is composed of two parts, offline speculative schedule optimization and runtime victim selection. In speculative schedule optimization step, we schedule transactions even over system resource capacity, guaranteeing system reliability. Then during runtime, we initially execute high-performance versions only and apply our victim selection

algorithm when a high-performance version fails. Experimental results confirm that our scheduling algorithm accepts 14% to 70% more transactions than basic Simplex architecture approach while guaranteeing system reliability and maximizing number of successful executions of high-performance versions.

The rest of this paper is organized as follows: Chapter 2 surveys related works. Chapter 3 reviews background of the framework proposed in this paper and describes the problem formulation. Chapter 4 presents our speculative schedule optimization technique that maximize resource efficiency while guaranteeing successful execution of software components and maximizing high-performance versions' executions. Chapter 5 presents our victim selection algorithm for guaranteeing every component to be successfully completed even when some high-performance versions fail. Chapter 6 presents the experimental results to evaluate this proposed framework. Finally, Chapter 7 concludes this paper and discusses the future work.

2. Related Work

2.1 Component-based Scheduling

[2] proposed a component-based scheduling algorithm to deal with high complexity issue of CPSs. Under traditional scheduling algorithms such as RM and EDF, concurrently scheduled SW components significantly affects temporal behavior of each other. In CPSs, these dependencies are huge due to high complexity of them, it makes hard validate and guarantee reliability of system. In order to resolve this problem, [2] is to provide the temporal isolation as if each SW component is executed on its dedicated HW component. In order to do this, their baseline approach is to permanently assign the required resource utilization to each SW component. However, since a SW component is active only for a short time during its period, a SW component wastes HW resource in most of time.

In order to reduce this waste, they provide the HW resource to each SW component only when it really requires HW resource. This short time window is called an “active window”. With this approach, a SW component C_{ij} is given the required resource utilization u_{ij} during delay bound d_{ij}.

Based on this approach, finding optimal schedule becomes a

kind of “bin packing problem”. In order to solve this problem, they shift the starting offset of each transactions and try to minimize the maximum peak utilization.

2.2 Software fault-tolerance approaches

Most of hardware fault-tolerance techniques are based on replication and major voting. Most of hardware faults are randomly generated by physical failures, thus it is almost impossible that replicated HW components produce the same fault at the same time. In contrast of hardware faults, software faults are time-invariant. So the same SW components produce the same results for the same inputs. Therefore, simply replicating a SW component does not improve the reliability of SW component.

However, N-version programming [4] is famous software fault tolerance approach and based on replication and major voting. It consists of N programs which are independently developed from the same program specification by different programmer groups. These independently developed programs are called “versions”. It is expected that each version has different software faults because different versions are independently developed. Thus, we can expect that major voting works.

However, there is a problem about this assumption. N-version

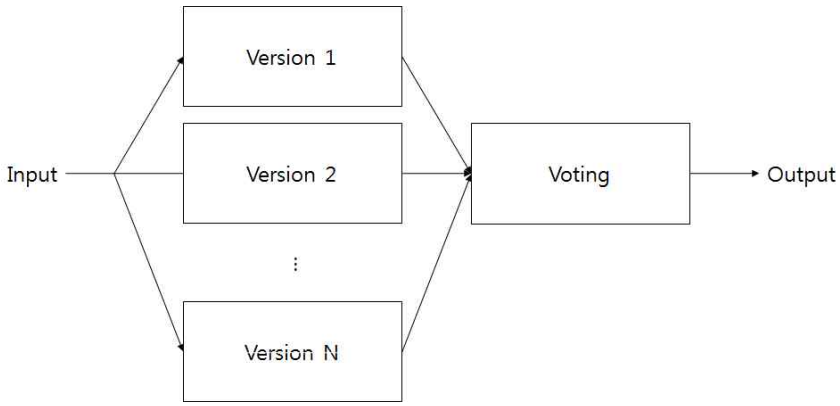


Figure 2. N-version programming

programming assumes that independently developed versions will fail independently so the probability of the majority of versions failing on the same inputs is extremely small. However, John C. Knight and Nancy G. Leveson argued that this assumption might be false [5]. Even among independently developed versions, they may share the same software faults. Thus, the probability of the majority of versions failing on the same inputs may be not that extremely small. As another problem, since N-version programming consists of N different versions, it requires N times cost and hardware resource. Thus, it is not practical for the resource constraint environment.

[6] proposed a software fault-tolerant scheduling algorithm and corresponding schedulability analysis. This algorithm also use redundancy to guarantee each SW components' reliability. It has two versions of programs, the primary and the alternate. The

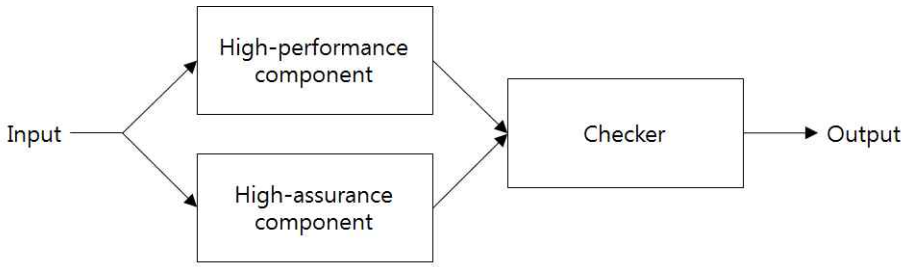


Figure 3. Simplex architecture

primaries produce better quality results but their reliabilities are not guaranteed. On the other hand, the alternates produce just acceptable results but they are always successfully completed when they are executed.

In order to efficiently use the resource, they proposed two algorithm, the checking available time (CAT) algorithm and the eliminating idle time (EIT) algorithm. CAT algorithm checks if there is enough available time to execute the primary. If there is not enough time to complete the primary, then its execution is cancelled. EIT algorithm reduce idle time by executing alternates to be executed if there is no primaries pending for execution.

However, this approach is not suitable for guaranteeing reliability of our target system. This approach is based on traditional scheduling algorithms such as RM and EDF. However, those scheduling algorithms cause complex temporal dependencies among SW components, and these dependencies make hard validate and guarantee reliability of system.

Simplex architecture is an software fault-tolerant approach

based on redundancy. It consists of two different versions of SW components, which are high-performance component and high-assurance component. A high-performance component is more complex because it contains more functions and produces good quality results. However, its execution is more prone to fail due to its high=level complexity and resource usage. On the other hand, a high-assurance component is more simpler because it contains only the minimum required functions and produces less precise but acceptable results. Due to its simplicity, it has been fully tested a priori, so its correctness is guaranteed.

Its operation mechanism is very simple. Two versions of components are executed concurrently. If high-performance component is completed successfully, then its result is chosen as a output. If]high-performance component fails, then the result of corresponding high-assurance component is chosen as a output.

3. Problem Formulation

In this paper, our periodic task system is formally defined as follows: Each transaction Γ_i is represented by a directed acyclic graph (DAG) $G_i = \{V_i, E_i\}$ consisting of $|\Gamma_i|$ SW components $\{C_{i1}, C_{i2}, \cdots, C_{i|\Gamma_i|}\}$, and triggered with the period P_i and should completed in the deadline D_i. As a consequence, a transactions Γ_i is represented by a three-tuple (G_i, P_i, D_i). Each SW component C_{ik} consists of two different versions, a high-performance component $C_{p,ik}$ and a high-assurance component $C_{a,ik}$. The high-performance component contains more functions and produces better quality results but it is prone to failure. The high-assurance component produce less precise results, but it is very reliable because it is easy to test due to its simplicity. Sum of their delay bound $d_{p,ik} + d_{a,ik}$ is equal to the SW component's delay bound d_{ik}. Usually $d_{p,ik} > d_{a,ik}$. The required resource utilizations of those two components are same as u_{ik}.

For given a set of N transations $\{\Gamma_1, \Gamma_2, \ldots \Gamma_N\}$, our objective is to efficiently use the HW resource while guarantee reliability of each SW component. More formally, our problem can be described as follows:

Problem Description: For given a set of transactions $\{\Gamma_1, \Gamma_2, \ldots \Gamma_N\}$ and a set of per-component specifications

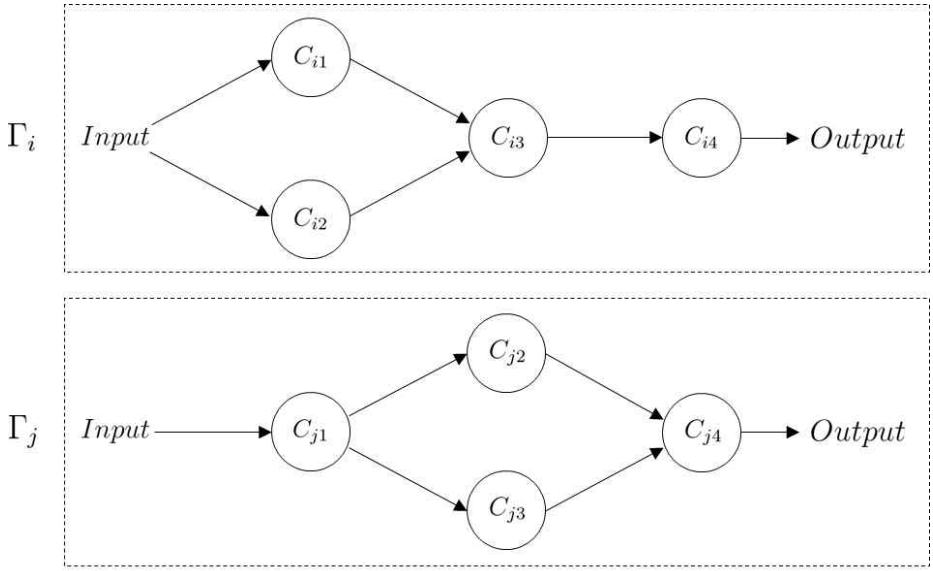


Figure 4. Example transactions

$(d_{p,ik}, d_{a,ik}, u_{ik})$s for C_{ik}s, our problem is how to complete as many high-performance components as possible while guaranteeing either the high-performance component or the high-assurance component of each transaction to be successfully completed by its delay bound d_{ik}.

Let the hyper period P_{hyper}, be the least common multiple of $P_1, P_2, ..., P_N$. Then, P_{hyper}/P_i is the number of transaction Γ_i in each hyper period. Without loss of generality, we can consider our problem for the first hyper period $[0, P_{hyper}]$ only. For the simplicity, we only consider harmonic transactions whose periods are divisible.

In this paper, we assume that 1) all transaction are

independent, that is, they have no data and temporal dependency and 2) all transactions are offset-free hence we can arbitrarily adjust each transaction's starting offset. These assumptions are valid through the rest of this paper.

4. Schedule Optimization

In this chapter, we describe our schedule optimization approach which is based on schedule optimization with active window based share provisioning of [2].

4.1 The Baseline Algorithm

Our proposed framework has two main objectives: 1) guarantee either the high-performance component or the high-assurance component of each transaction to be successfully completed by its delay bound d_{ik}; 2) completed as many high-performance components as possible. In order to satisfy these objectives, our baseline scheduling approach is to optimize schedule as simply reserving HW resource for executions of all high-assurance components.

In our system model, a SW component C_{ik} is given the resource utilization u_{ik} during delay bound d_{ik} every period P_i. Thus, our schedule optimization is to effectively stack these $u_{ik} \times d_{ik}$ blocks. In order to solve this bin packing problem, we use phase shifting optimization proposed in [2] as baseline algorithm. This baseline algorithm does not consider that each SW component consists of high-performance and high-assurance

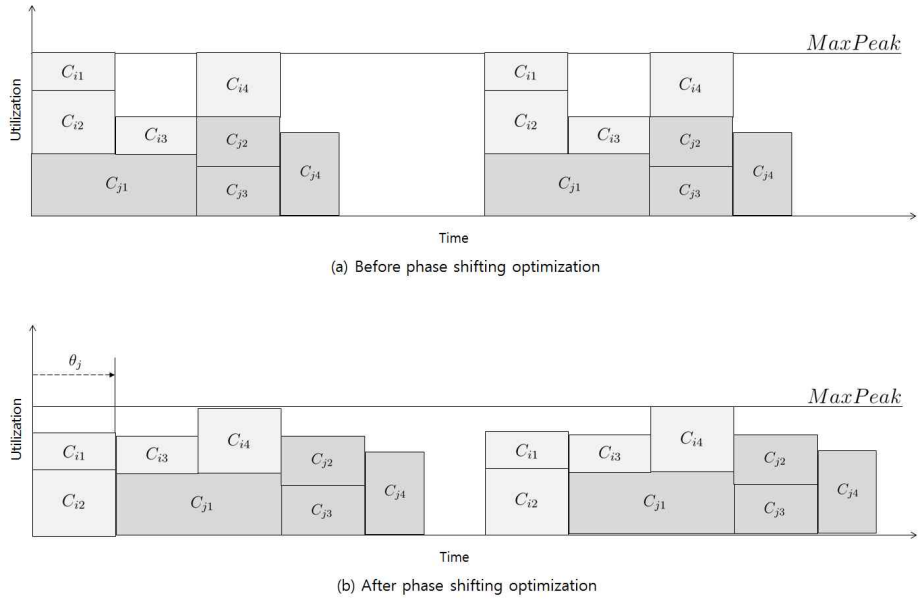
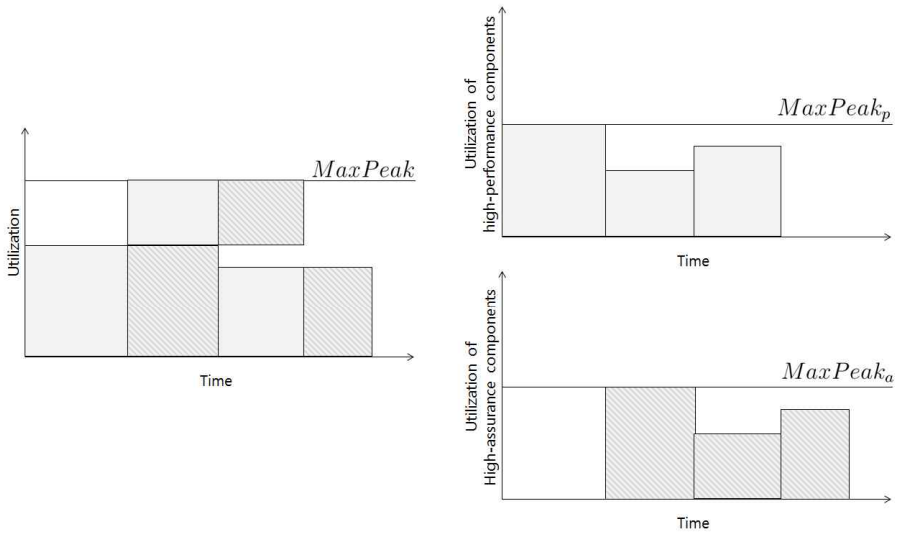


Figure 5. Example of phase shifting optimization

component at all, and consider each SW component as one block. Figure 5 shows a example of this phase shifting optimization. By shifting the phase of Γ_j as θ_j, we can reduce $MaxPeak$, which is the peak sum of the resource utilization and thus can schedule more transactions.

4.2 Schedule Optimization with Speculative Overbooking

The baseline scheduling approach always reserves HW resource for executions of all high-assurance components. Thus,



(a) Without separation high-performance components and high-assurance components

(b) With separation high-performance components and high-assurance components

Figure 6. Calculating MaxPeak with separation of high-performance components and high-assurance components

they waste HW resource when the result of high-assurance component is not needed. In order to reduce this resource waste, our proposed approach is to execute a high-performance component first and execute its corresponding high-assurance component only if that high-performance component fails.

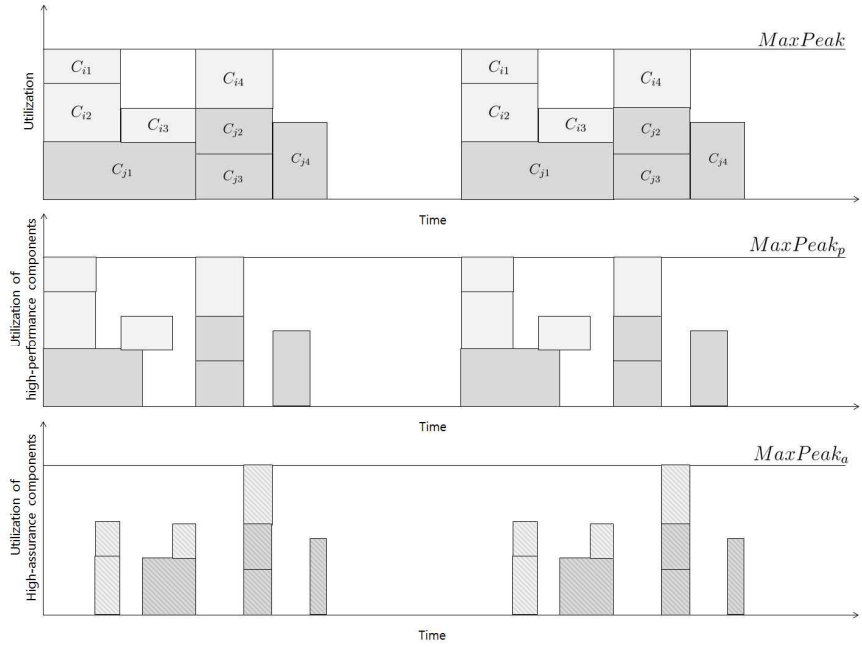
However, we cannot directly use these conserved HW resources because we don't know when high-assurance component needs to be executed in offline schedule optimization. In order to exploit these conserved HW resources, we speculatively schedule transactions over system capacity and expect the most of high-assurance component will not be

executed, so the system utilization will not be over system capacity in runtime. This approach follows the same principle of overbooking which is a common practice in the travel lodging industry.

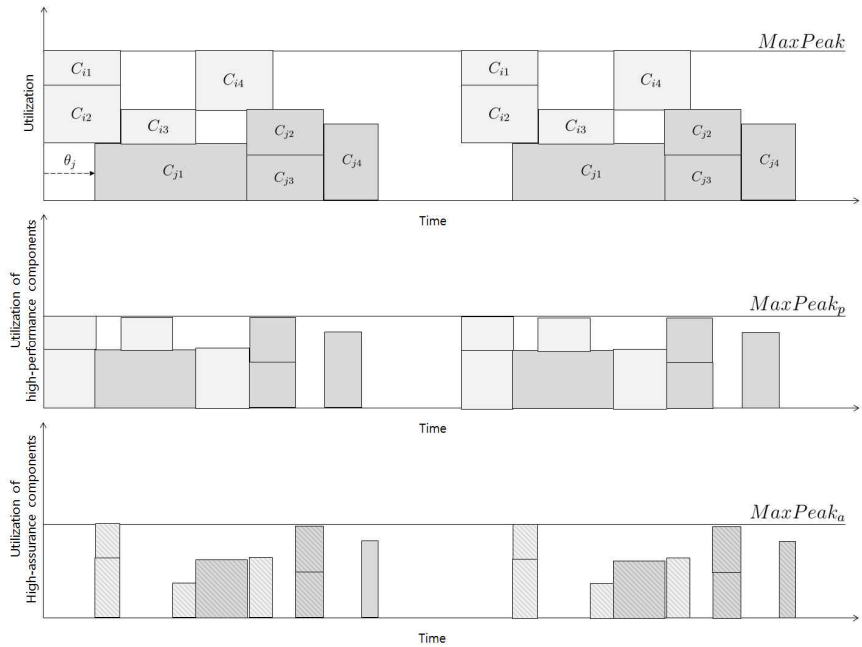
By using this strategy, we can exploit unused HW resources, but it also may allow the system utilization over the system capacity in the worst-case. When sum of share excesses HW resource capacity, some SW components should be selected as victims to reduce the system utilization. We will discuss this victim selection approach at chapter 5.

In order to speculatively optimize schedule, we consider the high-performance components and the high-assurance components separately. In our approach, *MaxPeak* just indicate degree of overbooking so it cannot be a proper measurement. Thus, as shown in Figure. 6, we calculate *MaxPeak* of the high-performance components and the high-assurance components separately. In order to achieve our objectives, following two constraints must be followed when schedule optimization is carried out.

- Constraint 1: The $MaxPeak_a$, *MaxPeak* of the high-assurance components, must not exceed 100%.
- Constraint 2: The $MaxPeak_p$, *MaxPeak* of the high-performance components, must not exceed 100%.



(a) Before phase shifting optimization



(b) After phase shifting optimization

Figure 7. Our proposed schedule optimization

Constraint 1 is obvious because reliability of system cannot be guaranteed if all of high-assurance component execution is not guaranteed. Constraint 2 is for preventing over-overbooking. $MaxPeak_p$ exceeding 100% implies that there are some high-performance component never be executed. Under these constraint, we conduct the phase shifting optimization.

Our proposed heuristic algorithm for the phase shifting algorithm is described as follows:

Table 1. Procedure for finding the near optimal phases for a set of harmonic transactions

FindOptimalPhases: Find the optimal phases $\{\theta_1,\theta_2,\cdots,\theta_k\}$ for a set of harmonic transactions $\{\Gamma_1,\Gamma_2,\cdots,\Gamma_k\}$
Input: Set of harmonic transactions $\{\Gamma_1,\Gamma_2,\cdots,\Gamma_k\}$
Output: Accepted Transactions number n, Near optimal phases $\{\theta_1,\theta_2,\cdots,\theta_n\}$
begin procedure
1. Sort $\{\Gamma_1,\Gamma_2,\cdots,\Gamma_k\}$ as the decreasing order of $Size(\Gamma)$
2. $\theta_1 = 0$
3. $i = 2$
4. **while** ($MaxPeak_p \le 100$ **and** $MaxPeak_a \le 100$) **do**
5. $\theta = 0$
6. $\theta_{best} = 0$
7. $MaxPeak_p^{smallest} = \infty$
8. $MaxPeak_a^{smallest} = \infty$
9. **while** ($\theta < P_i$) **do**

10. $MaxPeak_p^{\text{new}} = \max_{t \geq 0}\left(\sum_{j=1}^{i-1} U_{p,\Gamma_j}(t-\theta_j) + U_{p,\Gamma_i}(t-\theta)\right)$

11. $MaxPeak_a^{\text{new}} = \max_{t \geq 0}\left(\sum_{j=1}^{i-1} U_{a,\Gamma_j}(t-\theta_j) + U_{a,\Gamma_i}(t-\theta)\right)$

12. **if** $MaxPeak_p^{smallest} > MaxPeak_p^{\text{new}}$ **then**

13. $MaxPeak_p^{smallest} = MaxPeak_p^{\text{new}}$

14. $MaxPeak_a^{smallest} = MaxPeak_a^{\text{new}}$

15. $\theta_{best} = \theta$

16. **else if** $MaxPeak_p^{smallest} = MaxPeak_p^{\text{new}}$ **then**

17. **if** $MaxPeak_a^{smallest} > MaxPeak_a^{\text{new}}$ **then**

18. $MaxPeak_a^{smallest} = MaxPeak_a^{\text{new}}$

19. $\theta_{best} = \theta$

20. **end if**

21. **end if**

22. $\theta = \theta + 1$

23. **end while**

24. $\theta_i = \theta_{best}$

25. $i = i + 1$

26. **end while**

27. $n = i - 1$

end procedure

Line 1 sorts the transactions as the decreasing order of their size. The size of transaction Γ_i is calculated as follows:

$$Size(\Gamma_i) = \frac{u_{i1}d_{i1} + u_{i2}d_{i2} + \cdots + u_{i|\Gamma_i|}d_{i|\Gamma_i|}}{P_i} \quad (1)$$

Line 2 fixes the phase of Γ_1 to 0. Line 3 initialize the index that will be used in next while loop. The while loop from Line 4 to 26 iteratively places the transactions. that is, determines the phases of transactions. Lines 5 to 8 initialize the variables. The while loop from Line 9 to 23 determines the phase of transactions Γ_i. For the transaction Γ_i, the phases of transaction from Γ_1 to Γ_{i-1} are already determined. Thus, the sum of cumulated resource utilization so far and the transaction Γ_i's resource utilization is given in Line 10 and 11 as follows:

$$MaxPeak_p = \max_{t \geq 0}\left(\sum_{j=1}^{i-1} U_{p,\Gamma_j}(t-\theta_j) + U_{p,\Gamma_i}(t-\theta)\right) \quad (2)$$

$$MaxPeak_a = \max_{t \geq 0}\left(\sum_{j=1}^{i-1} U_{a,\Gamma_j}(t-\theta_j) + U_{a,\Gamma_i}(t-\theta)\right) \quad (3)$$

If $MaxPeak_p^{\text{new}}$ is smaller than the smallest $MaxPeak_p^{smallest}$ so far, Line 13 and 14 update $MaxPeak_p^{smallest}$ and $MaxPeak_a^{smallest}$, respectively. Then Line 15 updates the current best θ_{best}. If $MaxPeak_p^{\text{new}}$ is equal to $MaxPeak_p^{smallest}$, then Line 17 compares $MaxPeak_a^{\text{new}}$ with $MaxPeak_a^{smallest}$. If $MaxPeak_a^{\text{new}}$ is smaller than the smallest $MaxPeak_a^{smallest}$ so far, Line 18 updates $MaxPeak_a^{smallest}$ and Line 19 updates the current best θ_{best}. This

tie-brake is to reduce $MaxPeak_a$. The high-performance component is bigger than the corresponding high-assurance component. Thus, $MaxPeak_a$ is commonly lower than $MaxPeak_p$. However, we cannot ensure it, so we try to reduce $MaxPeak_a$ as possible. After completing this procedure, we can finally find the near optimal phases and the number of accepted transactions n.

5. Runtime Victim Selection

In chapter 4, we discussed our speculative schedule optimization approach. The schedule generated by our schedule optimization approach is overbooked, so the system utilization may exceeds the resource capacity in runtime. In this case, we should reduce the system utilization.

In order to solve this problem, our approach is to victimize some high-performance components and reclaim HW resource allocated to them. However, there is still a problem that which high-performance component is to be victimized. As shown in Figure 8, victim selection affects the number of victim would be needed. When p_2 is victimized, more victim is needed because a_2 make the resource utilization exceed 100 again. However, when p_4 is victimized, no more victim is needed. This implies that we can reduce the number of victims to properly select them.

Our intention is to minimize the number of victims which currently needed and the area of victims. The reason why we want to minimize the number of victims is because our objective is to complete as many high-performance components as possible. And as shown in Figure 8, victimizing bigger high-performance component may cause the situation that is needed more victims.

In order to select victim, first we should find the candidates which

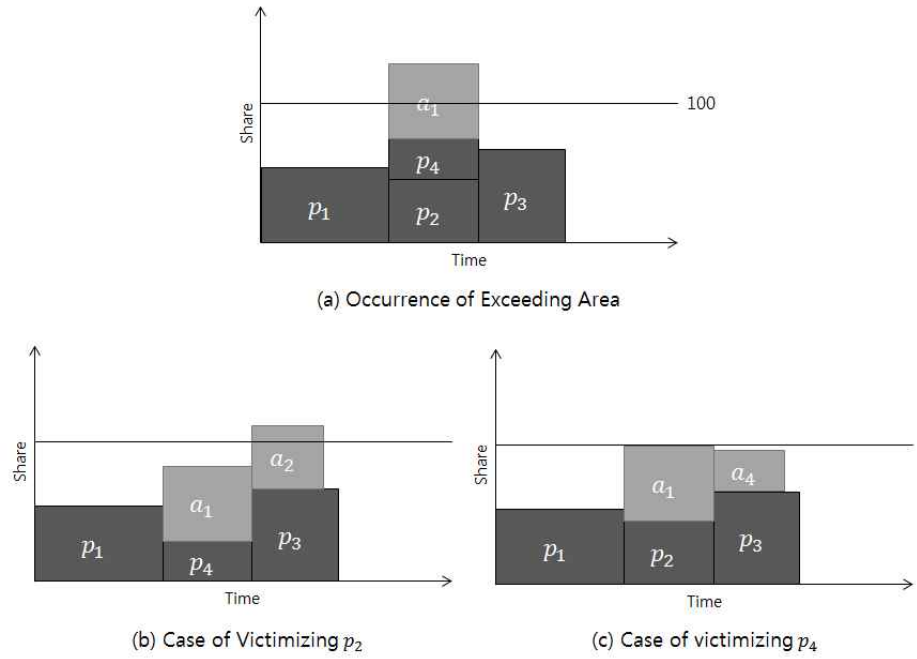


Figure 8. Two Cases of Victim Selection

can be victim. These candidates should overlap with exceeding area because victim should be able to reduce exceeding area. Victim cannot reduce exceeding area if it does not overlap with exceeding area. After we finish to find victim candidates, we select victim among candidates. In order to minimize the number of victims, we prefer to select a high-performance component which can eliminate whole exceeding area. For eliminating whole exceeding area, the victim should bigger than exceeding area, and its release time should not be later than the beginning point of exceeding area and its deadline should not be earlier than the end point of exceeding area. If there are several high-performance components which can eliminate whole exceeding

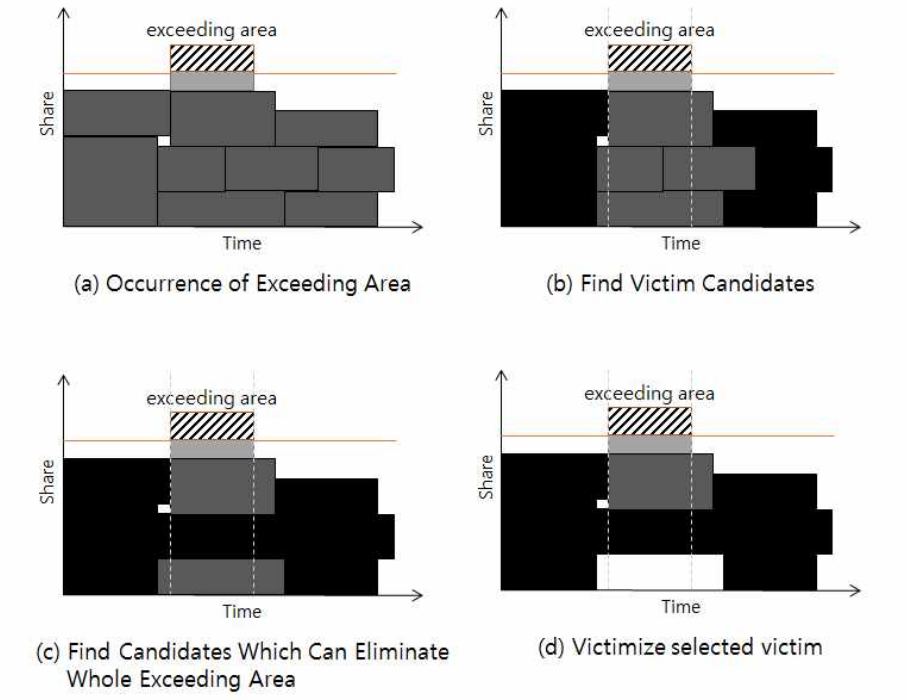


Figure 9. Example of Victim Selection

area, then we select the smallest one in aspect of size. If there is no such high-performance component, then we select one which can reduce exceeding area most, and find victim candidates and select a victim again.

Our victim selection algorithm is described as follows:

Table 2. Procedure for reducing the system utilization when it exceeds the system capacity in runtime

RuntimeVictimSelection: Victimize high-performance components until the system utilization $U \leq 100$

Input: Schedule which consists of high-performance components list

begin procedure

1. Sort
2. $begin = t_{curr}$
3. $end = begin$
4. **while** $MaxPeak(end) > 100$ **do**
5. $\quad end = end + 1$
6. **end while**
7. **while** $U > 100$ **do**
8. $\quad i = 1$
9. $\quad$ **while** $r_{p,i} + d_{p,i} < begin$ **do**
10. $\quad\quad i = i + 1$
11. $\quad$ **end while**
12. $\quad VictimCandidates = \{\ \}$
13. $\quad$ **while** $r_{p,i} < end$ **do**
14. $\quad\quad VictimCandidates = VictimCandidates \cup \{C_{p,i}\}$
15. $\quad$ **end while**
16. $\quad Max = ReductionArea(VictimCandidates_1)$
17. $\quad$ **for** $i = 2$ **to** $|VictimCandidates|$ **do**
18. $\quad\quad$ **if** $ReductionArea(VictimCandidates_i) > Max$ **then**
19. $\quad\quad\quad FinalVictimCandidate = \{VictimCandidates_i\}$
20. $\quad\quad\quad Max = ReductionArea(VictimCandidates_i)$
21. $\quad\quad$ **else if** $ReductionArea(VictimCandidates_i) = Max$ **then**
22. $\quad\quad\quad FinalVictimCandidates = FinalVictimCandidates \cup \{VictimCandidates_i\}$
23. $\quad\quad$ **end if**
24. $\quad$ **end for**
25. $\quad Min = Area(FinalVictimCandidates_1)$

```
26.      Victim = 1
27.      for i = 2 to |FinalVictimCandidates| do
28.          if Min > Area(FinalVictimCandidates_1) then
29.              Min = Area(FinalVictimCandidates_1)
30.              Victim = i
31.          end if
32.      end for
33.      Victimize(FinalVictimCandidates_victim)
34. end while
end procedure
```

This procedure would be activated when the resource utilization exceeds the resource capacity. First, Line 1 sorts the schedule as the increasing order of the release time r_p. Line 2 and 3 initialize the beginning point and the end point of the exceeding area. The while loop from Line 4 to 6 find the end point of exceeding area. The while loop from Line 7 to 34 repeatedly find the victim candidates and victimize high-performance component until the resource utilization is lower than the resource capacity.

In order to reduce searching space, the while loop from Line 9 to 11 increase the index until the high-performance component overlaps with the exceeding area. Line 12 initializes $VictimCandidates$. The while loop from Line 13 to 15 gather the victim candidates. Line 16 initialize Max as the area that can be reduced by victimizing $VictimCandidates_1$. The for loop from Line 17 to 24 find the victim candidates which can maximally

reduce the exceeding area. Then, the next for loop from Line 27 to 32 select victim, which has the smallest area, among *VictimCandidates* . After selecting a victim, Line 33 victimizes a high-performance component that is selected as a victim. After completing this procedure, the resource utilization is finally to be lower than the resource capacity.

6. Experiments

This section evaluates the proposed approach through simulation. We make a set of transactions with the following parameters:

- The periods of transactions are 300, 450, 900ms. N transactions are evenly divided into each period group.
- The deadline to period ratio $\frac{D_i}{P_i}$ whose default value is 0.5 if not otherwise mentioned.
- Each transaction has a randomly generated DAG consisting of three to five SW components.
- The share of SW components are randomly determined in range of [2, 10].

For generated transactions with the above parameters, we simulate schedule optimization. With this synthesized workload, we compare our proposed schedule optimization with the baseline optimization.

Figure 10 compares our proposed schedule optimization with the baseline optimization one in terms of the number of scheduled transactions as varying high-assurance component proportion of SW component. From the this graph, we can observe that our optimization always accepts more transactions

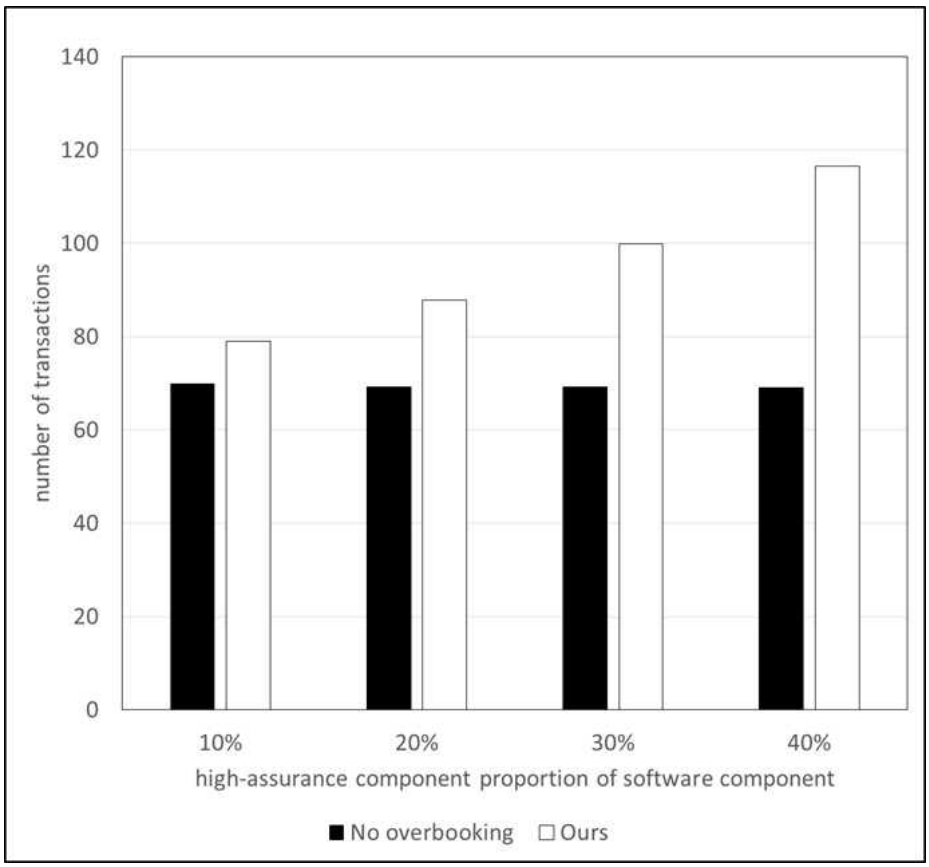


Figure 10. The number of accepted transactions

than the baseline optimization. Also, we can observe that the number of accepted transactions of the baseline optimization is constant as high-assurance component proportion of SW component increases. This is natural result because the baseline optimization does not consider high-performance component and high-assurance component separately so varying high-assurance component proportion does not matter. On the other side, we can observe increase of the number of acceptable transactions of our

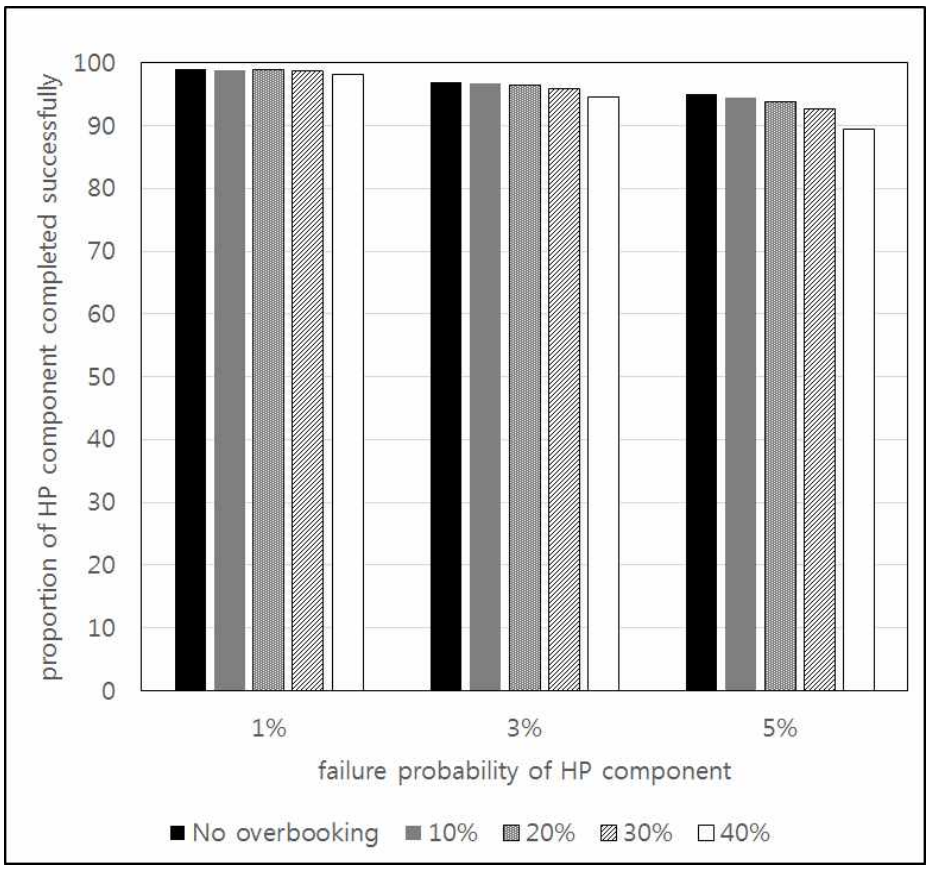


Figure 11. The proportion of high-performance component completed successfully

optimization as high-assurance component proportion of SW component increases. The bigger high-assurance component proportion grows, the smaller high-performance component proportion shrinks, so more transactions can be accepted until MaxPeak of high-performance components exceeds 100.

Figure 11 compares proportion of high-performance component completed successfully as varying failure probability of

high-performance component and high-assurance component proportion of SW component. From the graph, we can observe that difference between our optimization and the baseline optimization is small. It shows that our optimization adequately overbooks transactions without significant drop of proportion of high-performance components completed successfully.

7. Conclusion and Future Work

This paper proposes a framework for guaranteeing reliability of a CPS. Based on a framework proposed in [2], we provide temporal isolation to each SW components. It makes its reliability can be guaranteed by guaranteeing each SW component of the system. In order to provide software fault-tolerance to each SW component, we apply Simplex architecture. Along with this approach, we also modify schedule optimization algorithm proposed in [2] to efficiently use HW resource. And we propose victim selection algorithm to ensure the executions of high-assurance component when it need to be executed.

Our victim selection algorithm only consider the number and the area of SW components although importance and criticality of each SW component may be different. Thus, we are planning to modify our victim selection algorithm considering SW components' importance and criticality. Also, we plan to expand target HW resource in aspects of quantity and sort.

8. References

[1] Edward A. Lee, Cyber Physical Systems: Design Challenges, 2008 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), pages 363 - 369, May. 2008.

[2] 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, 2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), vol. 2, pages 28 - 31, Aug. 2011.

[3] Lui Sha , John B. Goodenough , Bill Pollak, Simplex Architecture: Meeting the Challenges of Using COTS in High-Reliability Systems, Journal of Defense Software Engineering, vol. 11, pages 7 - 10, 1998.

[4] L. Chen and A. Avizienis, N-version Programming : A Fault-Tolerance Approach to Reliability of Software Operation, Proc 8th IEEE International Symposium on Fault Tolerant Computing (FTCS8), vol. 1, pages. 3-9, 1978.

[5] John C. Knight and Nancy G. Leveson, An Experimental Evaluation of the Assumption of Independent in Multi-Version Programming, IEEE Transactions on Software

Engineering, vol. 12, pages. 96-109, 1986.

[6] Ching-Chih Han, Shin K. G. and Wu J, A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults, IEEE Transactions on Computers, vol. 52, issue. 3, pages 362-372, 2003.

[7] Jong-Chan Kim, Kyoung-Soo We, and Chang-Gun Lee, HW Resource Componentizing for Addressing the Mega-Complexity of Cyber-Physical Systems, in the 1st International Workshop on Cyber-Physical Systems, Networks, and Applications (CPSNA'11), Aug. 2011.

[8] 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 the 27th ACM Symposium on Applied Computing (SAC'12), 2012. vol. 2, pages 61-66, 2011.

[9] The Simplex Reference Model: Limiting Fault-Propagation Due to Unreliable Components in Cyber-Physical System Architectures, Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International, pages 400-412, Dec. 2007.

[10] D. Johnson, A. Demers, D. Ullman, M. Garey, and R. Graham. Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms. SIAM Journal of Computing, vol. 3, issue 4, pages 299 - 325, 1974.

[11] Stanley Bak, Deepti K. Chivukula, Olugbemiga Adekunle,

Mu Sun, Marco Caccamo, and Lui Sha, The system-level simplex architecture for improved real-time embedded system safety, In Proceedings of the Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE, pages 99 - 107, Apr. 2009.

[12] Jong-Chan Kim, Kyoung-Soo We, and Chang-Gun Lee, HW Resource Componentizing for Addressing the Mega-Complexity of Cyber-Physical Systems, in the 1st International Workshop on Cyber-Physical Systems, Networks, and Applications (CPSNA'11), Aug. 2011.

[13] D. Seto, B. Krogh, L. Sha, and A. Chutinan, The Simplex Architecture for Safe Online Control System Upgrades, Proceedings of the American Control Conference 1998, vol. 6, pages 3504 - 3508 Jun. 1998.

[14] J.-C. Kim, K.-S. We, C.-G. Lee, K.-J. Lin, and Y. S. Lee. ECU HW Componentizing. Technical report, May 2011. http://rubis.snu.ac.kr/~jckim/TR20110501.pdf.

[15] J. Lin, S. Sedigh, and A. Miller, Towards Integrated Simulation of Cyber-Physical Systems: A Case Study on Intelligent Water Distribution, in Proc. of the 8th IEEE Int'l Conference on Dependable, Autonomics and Secure Computing (DASC'09), pp. 690-695, Dec. 2009.

[16] D. Johnson. Near-Optimal Bin Packing Algorithms. PhD thesis, MIT, 1973.

[17] J.P.J. Kelly, "Specification of Fault-Tolerant Multi-Version

Software: Experimental Studies of a Design Diversity Approach," Ph.D. dissertation, University of California, Los Angeles, 1982.

요약(국문초록)

사이버-물리 시스템의 신뢰도 향상을 위한 콤포넌트 수준 심플렉스 구조 기반의 프레임워크

사이버 물리 시스템의 복잡도와 규모가 증가함에 따라 신뢰도를 보장하는 것이 점점 더 어려워지고 있다. 이 논문에서는 심플렉스 구조를 통해 사이버 물리 시스템의 신뢰도를 보장하는 프레임워크를 제안한다. 각 소프트웨어 콤포넌트에 시간 격리성을 제공함으로써 전체 시스템의 신뢰도 보장 문제를 각 콤포넌트 단의 신뢰도 보장 문제로 바꿀 수 있게 한다. 그 다음, 이 시간 격리성에 기반하여 소프트웨어 결함 내성 기법인 심플렉스 구조를 각 소프트웨어 콤포넌트에 적용하여 각 소프트웨어 콤포넌트의 신뢰성을 보장하고자 한다. 그러나 심플렉스 구조는 신뢰성을 보장하기 위해서 추가적인 소프트웨어 콤포넌트를 실행시키기 때문에 추가적인 하드웨어 자원을 요구한다. 이 추가적인 자원 요구를 줄이기 위해서 우리는 투기적 스케줄 최적화 기법과 희생자 선택 알고리즘을 제안한다. 이를 통해 CPS의 신뢰성을 보장하면서 하드웨어 자원을 효율적으로 사용하여 고기능의 작동을 최대화 것이 가능해진다. 실험을 통해 제안된 스케줄링 기법은 기본적인 방법보다 14% 에서 70%까지의 더 많은 트랜잭션을 받아들이는 결과를 확인하였다.

주요어 : 사이버 물리 시스템, 심플렉스 구조, 소프트웨어 결함 내성, 신뢰성, 스케줄 최적화, 희생자 선택 알고리즘
학 번 : 2011-20928