Automatic Synthesis for Distributed Systems

Yang Yang

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-38
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-38.html

March 11, 2009

Automatic Synthesis for Distributed Systems

by Yang Yang

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Alberto Sangiovanni-Vincentelli
Research Advisor

Date

* * * * * *

Professor Kurt Keutzer
Second Reader

Date

Abstract

Distributed architectures are widely used in various application domains to cope with the increasing system complexity. In this work, we focus on finding a mapping from an application description to a distributed architecture platform, to optimize certain objectives while satisfying design constraints. Specifically, the mapping problem includes allocating functional tasks to distributed processors and allocating messages to buses, scheduling tasks and messages, as well as deciding buffer sizes. This synthesis problem is still largely open nowadays. In three case studies, we developed a set of algorithms to address various aspects of mapping, including applying on different types of architecture platforms, exploring different design variables and optimizing different optimization objectives.

The first two case studies focus on hard real-time distributed systems that collect data from a set of sensors, perform computations in a distributed fashion and based on the results, send commands to a set of actuators. Hard real-time requires that the tasks must satisfy hard end-to-end deadline constraints in the worst case. In the first project, our goal is to find a mapping to minimize total response time while satisfying end-to-end latency constraints. We designed a reinforcement machine learning algorithm, in which the rewards are evaluated based on the worst-case system performance analysis to guarantee the satisfaction of hard real-time constraints. The experimental results show the algorithm converges in reasonable time for non-trivial problems, and produces near optimal solution. In our second project, we measure the extensibility of the design solutions and then develop an efficient algorithm that optimizes this metric, other than just finding an satisfying mapping. Extensibility is defined as the amount by which the execution time of tasks can be increased without changing the system configuration while meeting the deadline constraints (as in [1]). With this definition, a design that is optimized for extensibility not only allows adding future functionality with minimum changes, but is more robust with respect to the variance of task execution times. Experimental results showed that our algorithm can significantly increase the extensibility of the system while meeting the latency constraints.

Our third project solves the buffer sizing problem in the mapping for Parallel Heterogeneous Platforms (PHPs). In this problem, we focus on distributed systems where processing units communicates through FIFOs between each pair of them. The work is concerned with determining the buffer sizes between processing elements. We want to minimize the buffer sizes while avoiding artificial deadlock [2]. Prior work in this field mainly focuses on the buffer sizing problem in uni-processor platforms [3]. The previous work that deals with mul-

tiprocessor buffer minimization [4] does not consider interleaving communication, where two active tasks on different processors can communicate large amounts of data using one-place buffers. In this work, we develop algorithms to address this problem. Theoretical as well as practical results are provided.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I'd like to thank my research advisor, Professor Alberto Sangiovanni-Vincentelli for his continued support and guidance. His feedback has greatly enhanced the quality of this work and provided guidance for future research directions. I'd like to thank Professor Kurt Keutzer for agreeing to be the second reader of this report. His valuable suggestions helped me improve the work.

Many thanks to Zhangxi Tan and Thomas Huining Feng, who were deeply involved in the first and the third projects respectively. This work would not have been possible without them. Many thanks to Abhijit Davare, Marco Di Natale, Eelco Scholte and Qi Zhu for their valuable advice from initial brainstormint to the implementation. Their ideas and insights were integral for this report.

Last but not least, I'd like to thank my parents and my husband for their love and support. Their continued encouragement has been vital to my pursuit of graduate studies at UC Berkeley.

1 Background of Hard Real-Time Distributed Systems

Hard real-time distributed systems are commonly used in cars, airplanes, industrial plants, building, etc. In these systems, a set of control tasks are executed on distributed implementation platforms consisting of multiple computation nodes (ECUs) that communicate with standard buses so that end-to-end latencies are within a given hard bound.

We consider systems based on *run-time priority-based scheduling* of tasks and messages. In the automotive domain, standards supporting this model are the OSEK operating system [5] standard and the CAN bus arbitration model [6]. The communication model considered in the projects, consists of the *periodic activation* with *asynchronous communication*, where all tasks are activated periodically and communicate by means of asynchronous buffers based on non-blocking read/write semantics. Similarly, message transmission is triggered periodically and each message contains the latest values of the signals that are mapped into it [7, 8].

More specifically, the execution model considered in this work is the following. Input data (generated by a sensor, for instance) are available at one of the system's computational nodes. A periodically activated task on this node reads the input data, computes intermediate results, and writes them to the output buffer from where they can be read by another task or used for assembling the data content of a message. Messages - also periodically activated - transfer the data from the output buffer on the current node over the bus to an input buffer on a remote node. Local clocks on different nodes are not synchronized. Tasks may have multiple fan-ins and messages can be multi-cast. Eventually, task outputs are sent to the system's output devices or actuators.

1.1 Mapping Problem

Our optimization problem is part of the mapping stage in the Platform-Based Design (PBD) [9] design flow, where the functionality of the design (what the system is supposed to do) and its architecture (how the system does it) are captured separately, and then "joined" together, i.e., the functionality is "mapped" onto the architecture.

During mapping, design variables are explored to optimize objective functions while satisfying

design constraints. The sets of variables, objectives and constraints vary for different systems. For instance, task scheduling is commonly explored in mapping since it might significantly affect the system performance. Tasks can be statically scheduled or dynamically decided during runtime, and the choice of which lead to different sets of design variables. For objectives and constraints, various systems have different focuses. In multimedia systems, metrics such as throughput, communication bandwidth are usually the most important ones, while in some hard real-time systems, the end-to-end latencies are chosen as the direct optimization objects. Besides the choice of variables, objects and constraints, there are also different modeling approaches. For example, task execution times can be modeled as fixed numbers, or treated as statistical variables and modeled by probability distributions.

In this work for hard real-time systems, the general mapping problem is formulated based on the specific characteristics of real-time systems. Function blocks communicate through signals, which represent the data dependencies. The architectural description is a topology of computational nodes connected by buses. Mapping deploys functional blocks to tasks and tasks to nodes. Correspondingly, signals can be mapped into local communication or packed into messages that are exchanged over the buses. Tasks and messages are scheduled based on static priorities, which are also explored during mapping. And a set of end-to-end latency constraints need to be satisfied in the worst-case. The *task allocation, signal to message packing, message allocation and priority assignment* are the design activities considered in this paper with the objective of optimizing total latency in the first project and task extensibility in the second project.

1.2 System Model

1.2.1 Problem Representation

The mapping problem is represented as a directed graph $\mathcal{G} = (\mathcal{T}, \mathcal{S})$. $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$ is the set of tasks that perform the computations. $\mathcal{S} = \{s_1, s_2, ..., s_m\}$ is the set of signals that are exchanged between task pairs. src_{s_i} and $\{dst_{s_i,j}\}$ denote the source task and the set of destination tasks of signal s_i, respectively (communication is of multicast type). The application is mapped onto an architecture that consists of a set of computational nodes $\mathcal{E} = \{e_1, e_2, ..., e_p\}$ connected through a

set of CAN buses $\mathcal{B} = \{b_1, b_2, ..., b_q\}$.

A task τ_i is periodically activated with period t_{τ_i}, and executed with priority p_{τ_i}. Tasks are scheduled with preemption according to their priorities, and a total order exists among the task priorities on each node. c_{τ_i} is the worst case computation time of τ_i, and r_{τ_i} is its worst case response time. Computational nodes can be heterogeneous, and tasks can have different execution times on different nodes. We use $c_{\tau_i,e}$ to denote the execution time of task τ_i on node e. In the following, the e subscript is dropped whenever the formula refers to tasks on a given node, and e is implicitly defined, or when the task allocation is (at least temporarily) defined, and the node to which the computation time (or its extensibility Δc) refers, is known.

For a signal s_i, the computational nodes to which the source task src_{s_i} and the destination task $dst_{s_i,j}$ are allocated are called source and destination nodes, respectively. If the source node is the same as all the destination nodes, the signal is local. Otherwise, it is global and must be packed into a message transmitted on one of the buses between the source node and all its destination nodes. Only signals with the same period, same source node and same communication bus can be packed into the same message. For message m_i, t_{m_i} denotes its period, p_{m_i} denotes its priority, and c_{m_i} denotes its worst case transmission time on a bus with unit speed. The worst transmission time on bus b_j is $c_{m_i}/speed_{b_j}$, where $speed_{b_j}$ is the transmission speed of b_j. r_{m_i} is the worst case response time on a bus with unit speed.

A path p on the application graph $\mathcal{G}$ is an ordered interleaving sequence of tasks and signals, defined as $p = [\tau_{r_1}, s_{r_1}, \tau_{r_2}, s_{r_2}, ..., s_{r_{k-1}}, \tau_{r_k}]$. $src(p) = \tau_{r_1}$ is the path's source and $snk(p) = \tau_{r_k}$ is its sink. Sources are activated by external events, while sinks activate actuators. Multiple paths may exist between each source-sink pair. The worst case end-to-end latency incurred when traveling a path p is denoted as l_p. The path deadline for p, denoted by d_p, is an application requirement that may be imposed on selected paths.

1.2.2 Performance Modeling

Utilization Since all the tasks are periodic, the worst case utilization of ECU e_k is evaluated by

$$U(e_k) = \sum_{i \in L(e_k)} \frac{c_{\tau_i}}{t_{\tau_i}} \tag{1}$$

where $L(e_k)$ is the set of tasks allocated to e_k.

When the utilization is bigger than 1, the ECU is overloaded. Some of the tasks in the ECU will have longer and longer response time as they keep running, and finally some response times will tend toward infinity.

Moreover, in 1973 Liu and Layland [10] proved that for a set of n periodic tasks with unique periods, a feasible schedule that will always meet deadlines exists if the CPU utilization is

$$U = \sum_{i=1}^{n} \frac{c_i}{t_i} \leq n(\sqrt[n]{2} - 1)$$

when $n = 2$, $U \leq 0.8284$; when the number of processes tends toward infinity, the expression tends toward $U \leq 0.693147$.

To make the response time finite, we must make the load of every ECU smaller than 1, moreover, smaller than 69.3% if possible to make sure the system is schedulable.

End-to-End Latency Analysis After tasks are allocated, some signals are local, and their transmission time is assumed to be zero. Others are global, and need to be transferred on the buses through messages. The time needed to transfer a global signal is equal to the transmission time of the corresponding message. Let r_{s_i} denote the worst case response time of a global signal s_i, and assume its corresponding message is m_j, then $r_{s_i} = r_{m_j}$.

The worst case end-to-end latency can be computed for each path by adding the worst case response times of all the tasks and global signals on the path, as well as the periods of all the global signals and their destination tasks.

$$l_p = \sum_{\tau_i \in p} r_{\tau_i} + \sum_{s_i \in p \wedge s_i \in GS} (r_{s_i} + t_{s_i} + t_{dst_{s_i}}) \tag{2}$$

where GS is the set of all global signals.

We need to include periods of global signals and their destination tasks because of the asynchronous sampling of communication data. In the worst case, the input global signal arrives immediately after the completion of the first instance of task τ_i. The event data will be read by the task on its next instance and the result will be produced after its worst case response time, that is, $t_{\tau_i} + r_{\tau_i}$ time units after the arrival of the input signal. The same reasoning applies to the execution of all tasks that are the destinations of global signals, and applies to global signals themselves. However, for local signals, the destination task can be activated with a phase equal to the worst-case response time of the source task, under the condition that their periods are harmonic, which is almost always true in practical designs. In this case, we only need to add the response time of the destination task. Similarly, it is sometimes possible to synchronise the queuing of a message for transmission with the execution of the source tasks of the signals present in that message. This would reduce the worst case sampling period for the message transmission and decrease the latency in Equation (2). In this work, we do not consider these possible optimizations and leave them to future extensions.

Response Time Analysis Computing end-to-end latencies requires the computation of task and message response times (signal response times are equal to the response times of the corresponding messages). The analysis in this section summarizes work from [11, 12].

Task Response Times In a system with preemption and priority-based scheduling, the worst case response time r_{τ_i} for a task τ_i depends on its computation time c_{τ_i}, as well as on the interference from higher priority tasks on the same node. r_{τ_i} can be calculated using the following recurrence:

$$r_{\tau_i} = c_{\tau_i} + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{r_{\tau_i}}{t_{\tau_j}} \right\rceil c_{\tau_j} \tag{3}$$

Where $hp(\tau_i)$ refers to the set of higher priority tasks on the same node.

Message Response Times Worst case message response times are calculated similarly to task response times. The main difference is that message transmissions on the CAN bus are not preemptable. Therefore, a message m_i may have to wait for a blocking time B_{max}, which is the longest transmission time of any frame in the system. Likewise, the message itself is not subject to preemption from higher priority messages during its own transmission time c_{m_i}. The response time can therefore be calculated with the following recurrence relation:

$$r_{m_i} = c_{m_i} + B_{max} + \sum_{m_j \in hp(m_i)} \left\lceil \frac{r_{m_i} - c_{m_i}}{t_{m_j}} \right\rceil c_{m_j} \tag{4}$$

For more details about performance modeling, please refer to [7, 8].

2 Allocation and Scheduling for Hard Real-Time Distributed Systems

2.1 Overview

In this project, we developed an algorithm based on machine learning to find an optimal mapping for hard real-time distributed systems described in Section 1. This synthesis problem is still largely open. In prior work, genetic algorithms [13] and simulated annealing [14] were used to solve this problem, but the performance needs to be improved.

Our goal is to find a mapping to minimize the total response time and satisfy end-to-end latency constraints. The problem is proved to be NP-hard. We developed a Q-learning based reinforcement machine learning algorithm, in which the rewards are evaluated based on the worst-case system performance analysis. The experimental results show the algorithm converges in reasonable time for non-trivial problems, while giving near optimal solutions.

2.2 Assumptions and Problem Formulation

In this project, we only consider hard real-time distributed systems with a single CAN bus, and we assume that each signal with its source node and destination node located in different ECUs is packed into an individual message. The design problem can be defined as follows. Given a set of design constraints including:

- end-to-end deadlines on selected paths
- utilization bounds on nodes and buses

explore the design space that includes:

- allocation of tasks to computational nodes
- assignment of priorities to tasks and messages

to minimize total worse case response time of tasks and messages.

The problem is proved to be NP-hard in [15]. It can be formulated as a Mixed Integer Linear Programming (MILP) problem, but is hard to solve as the size of problem is huge in practical.

2.3 System Performance Estimation

The system performance model is analyzed as in Section 1. This section describes two techniques we used in the performance estimation.

First, we use Newton's Method (described in Algorithm 1) to solve the implicit Equations (3) and (4) for response times. The Equation (3) is rewritten as

$$F(x) = c_{\tau_i} + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{x}{t_{\tau_j}} \right\rceil c_{\tau_j} - x = 0$$

We do the same thing for computing message response time with Equation (4).

Algorithm 1 NEWTON'S METHOD (TO SOLVE $F(x) = 0$)

```
1: x = initial guess
2: step = 0
3: while step < MAX ITERATION NUMBER do
4:     Evaluate dF = dF(x) and F = F(x) with current value of x
5:     if |F| < ERROR THRESHOLD and |Δx| < ERROR THRESHOLD then
6:         Newton's method converges, break
7:     else
8:         Solve dF * Δx = −F for Δx
9:         Update x = x + Δx
10:        step = step + 1
```

Since there are ceiling functions in the function $F(x)$, its derivative is not continuous. Rather than using the derivative of $F(x)$, we evaluate dF by the derivative of function $F'(x)$, which removes the ceiling functions in $F(x)$. Let d denote the derivative $dF'(x) = \sum_{\tau_j \in hp(\tau_i)} \frac{c_{\tau_i}}{t_{\tau_j}} - 1$. d is a constant and $d > -1$ when $hp(\tau_i) \neq \Phi$. (Note that $r_{\tau_i} \equiv x = c_{\tau_i}$when $hp(\tau_i) = \Phi$.)

The derivative $d \geq 0$ when $\sum_{\tau_j \in hp(\tau_i)} \frac{c_{\tau_i}}{t_{\tau_j}} \geq 1$, which implies the ECU is already fully loaded by the tasks with higher priority than τ_i. In this case, we set the response time of task τ_i to be infinity

to indicate that it cannot be added to this ECU.

The other case is that $-1 < d < 0$, which means $\sum_{\tau_j \in hp(\tau_i)} \frac{c_{\tau_i}}{t_{\tau_j}} < 1$. Convergence of Newton's method is not guaranteed in general unless we start close enough to the solution and the derivative is continuous. In addition, global convergence can be claimed if the function is monotonic. In our case, $F(x)$ and $F'(x)$ are both monotonic. Also we notice that $F(x) \geq F'(x)$ is always true for any x, and the equality is achieved if and only if x/t_{τ_j} is an integer for any $\tau_j \in hp(\tau_i)$. Besides, as shown in the Figure 1, if we start at point $x = 0$, after the first iteration we get to $x = x_1$, which is much closer to the exact solution x^*. However, since the derivative of $F(x)$ is not continuous, we still cannot guarantee the convergence of the Newton's Method. But we adjust the Newton's method according to the special properties of the function $F(x)$ to speed up its converging. We use $x = x + \alpha\Delta x$ instead of $x = x + \Delta x$ to reduce oscillation when updating x, where $\alpha = min\{1, \gamma/|\Delta x|\}$. γ is reduced every time Δx changes its direction. After all, if we still cannot get a converged solution, we set the solution to be the smallest x that we obtained during the iterations such that $F(x) \leq 0$. As this x is always larger than x^*, we get a conservative estimation of the response time and therefore guarantee that our algorithm will not violate the hard real-time constraints.

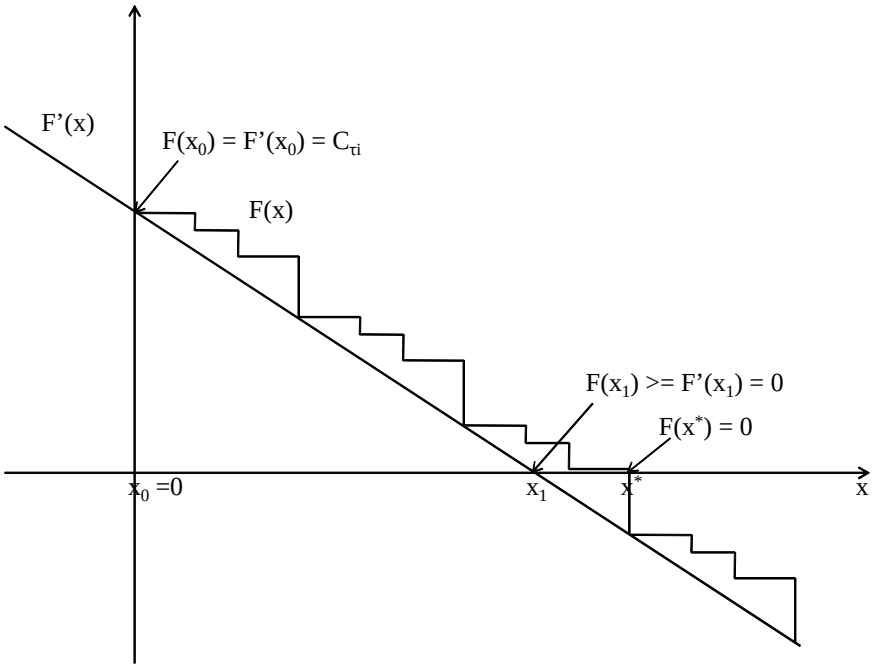


Figure 1: The Functions F(x) and F'(x) for Response Time Calculation

The worst case end-to-end latency for a computation spanning a path is evaluated by the sum of the worst case response time of all the tasks and messages on the path. Since it is the worst case

value, we can guarantee that the end-to-end latencies always satisfy the constraints in any case.

Second, we use priority groups instead of exact priorities to reduce the search space. As we discussed in Section 1, different tasks in an ECU should have different priorities. To reduce the search space, we provide each ECU with a limited number of priority groups (5 to 10 groups in usual cases). We assign a task of a priority group but not an exact priority. So there might be more than one task with the same priority group in an ECU. To evaluate the worst case response time of a task in this case, we use the Equation (3), where $hp^k(i)$ is the set of tasks in e_k with higher or the same priority groups as τ_i .

Similarly, messages in different ECUs should have different priorities so that the CAN bus can make deterministic resolution of the contention. In our algorithm, we assign messages in the same ECU with the same priority, and messages from different ECUs with different priorities. Since at any time there is only one messages from an ECU transmitted to the CAN bus, we can still use the Equation (4) to evaluate the worst case response time of a message.

2.4 Solving the Mapping Problem with Reinforcement Learning

As stated above, the optimal mapping problem is an NP-hard problem. Many of such problems are difficult not because we do not have fast computer with enough memory, but because it is too hard for the scheduler (either static or dynamic schedule) to choose the best task to run. Reinforcement learning (RL) is a modern machine learning technology that is more general than both supervised and unsupervised learning [5]. In reinforcement learning, we don't need to know the correct answer like those in supervised learning and the algorithm can be used online to adapt to runtime dynamic system. Some optimization and control techniques, such as control theory and dynamic programming, always assume a static system model. Therefore, they are limited by the size and complexity defined before the problem is actually solved. In RL, the computer is simply given a goal to achieve. Then the computer learns how to achieve the goal by trial-and-error with its environment.

To solve the problem, we choose an interactive setting and apply reinforcement learning to find an optimal solution. Under this framework, the environment is a setting where the task allocation

scheduler learns by trial-and-error iterations. This environment is observed by our RL task scheduler (agent). The observations come in form of sensors to current task allocation attempts, such as average task response time, communication latencies and etc. Our scheduler, choose its actions, i.e. allocate tasks to dedicate ECUs with some priority, based on some environment "states". As we formulate the problem above, the goal of the scheduler is to achieve lowest overall response time of tasks and messages in addition to satisfying communication latencies requirement. After every task is allocated, the scheduler will receive some reinforcement (reward) in the form of a scalar value. The scheduler will learn a sequence of best actions to achieve the maximum reward.

In our problem, the time complexity of calculating the precise model of environment (i.e. the performance of the distributed system) is very high. As this calculation happens in every trail of Reinforcement Learning, we use an abstract enviorment model to reduce the runtime. Therefore Q-learning [16] is chosen to be used in our work as it is a form of Reinforcement Learning algorithm that does not need a strong environment model and can be used on-line. In the following sections, we will describe the Q-learning algorithm used in our work as well as the design of environment and reward respectively.

2.4.1 Q-Learning

Let the environment state at time t be s_t, and assume that the learning system then chooses action a_t. The immediate result of this action is a reward R_t. After this action, the system will undergo a transition to the next state s_{t+1}. The objective is to choose a policy π (sequence of actions) maximizing discounted cumulative rewards over time. To be more specific, the total discounted return starting from time t is given by the following formula:

$$r(t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \cdots + \gamma^n r_{t+n} + \cdots \tag{5}$$

The discount factor γ is a number between $[0..1]$. It is used to weight near term reinforcement more heavily than distant future reinforcement. The closer it is to 1 the greater the weight of future reinforcements.

Q-learning is a simple incremental algorithm developed from the theory of dynamic programming for delayed reinforcement learning. In Q-learning, we define a state-action value function $Q^{\pi}(s,a)$, which is expected to return when starting in s, performing a, and following π. $Q^{\pi}(s,a)$ is updated by the following rule during the learning process.

$$Q(s_t,a_t) = (1-\alpha)Q(s_t,a_t) + \alpha[R(s,a_t) + \gamma max_{\alpha_{t+1}} Q(s_{t+1},a_{t+1})] \tag{6}$$

where α is a learning rate parameter, it should be close to 1 at the initial stage of learning and gradually approaches to 0 for converging to an optimal state value function. In each iteration i, we choose to make either a random action or the action from optimal policy $\pi(s) = argmax_{\alpha}Q(s,a)$ with a probability of ε. This is called ε-greedy policy. At first, ε is close to 0 for random exploring all possible states. Slowly, ε is increased to 1, hence it focuses mainly on optimal actions. It is also showed in [17] that the above Q state-value function update always converges.

2.4.2 Environment Design

To set up our Q-learning scheduler, we need an environment in which we evaluate trial actions. Therefore, we create a system simulator that emulates task running and provides performance measurements through an analytical model as described in Section 1.

State and Action Space Design We divide the states space in terms of number of tasks to be allocated. The initial state has all tasks un-allocated. The goal state has zero unassigned tasks. If one task is assigned to an ECU with a priority, then the system goes to a new state. Therefore, the number of states is exactly equals to the number of tasks need to be assigned, which is $O(m)$ and m is the number tasks. Without losing the generality, the schedule always makes scheduling in order from the first task to the last.

In each state, the scheduler can assign the current task to any ECU with a priority. So the action space is the cross product of number of ECUs and number of possible task priority groups. This is bounded by $O(n*p)$, where n is the number of ECUs in the system and p is the number of priority groups. Thus, the Q state-value function is bounded by $O(m*n*p)$. For most cases, m and n are

usually less than 100 and p is less than 10. If state-value is stored as double precision floating point that is 8-byte, the total required storage for Q is less than 1 Mega bytes. This can completely fit in the main memory of modern computer, even in the Level-2 data Cache. So, the Q-matrix update can be very efficient.

Reward Design Reward design is a very important part in our project. We spent most of our time designing reward that helps algorithm to converge faster and concentrate on interesting "good" choice. To be more specific, we have the following reward.

Case 1: $R(s,a) = -C$, where $C = [\sum_{\tau_i \in T} T(\tau_i) + \sum_{s_{i,j} \in S} T(s_{i,j})]^{-1}$, if any of the end-to-end latency constraints is violated, or the utilization of ECU that the task is allocated onto in the action a exceeds a threshold (which we set as 0.7 in the algorithm).

Case 2: $R(s,a) = 0$, when its next state is not the goal state. In addition, all the end-to-end latency constraints are satisfied, and utilization of every ECU is below the threshold.

Case 3: $R(s,a) = [\sum_{\tau_i \in T} r(\tau_i) + \sum_{s_{i,j} \in S} r(m_{i,j})]^{-1}$, i.e. the inverse of total response time, if its next state is the goal state. Besides, all the end-to-end latency constraints are satisfied, and utilization of every ECU is below the threshold.

Intuitively, in each state, if current assignment already violates the given constraints, we think this action is a "bad" move, therefore assigning a negative reward. On the other hand, only when we reach the goal state, can it have a positive reward which is given as the inverse of total response time.

Algorithm Description We start from the initial states and explore possible actions in the action space. When we reach the goal state, we call this process an *episode*. For all our experiments, we choose a constant of 0.8 as the discounted reward rate. The pseudo code of the Q-learning algorithm is shown in Algorithm 2.

Algorithm 2 Q-LEARNING BASED SYNTHEIS ALGORITHM

Set initial parameter $\varepsilon = 0$,$\gamma = 0.8$, $\alpha = 1$
Initialize matrix $Q = \Phi$
for each episode **do**
  while not reach goal state (all tasks are assigned) **do**
    Assign current task: select one ECU and a task priority using ε-greedy policy
    Using the current assignment, consider to go to the next state
    Get maximum Q value of the next state based on all possible allocations (actions) and Reward from simulator
    Compute and update Q value for current state
    Set next state to be current state
  Increase ε and decrease α
  if $\alpha = 0$ and $\varepsilon = 1$ **then**
    Return result
  if result converge or reach maximum episode **then**
    Return result

2.5 Experiment Results and Conclusions

The simulator environment and the learning algorithm are all written in C++ for maximum performance. We conduct a set of experiments varying different number of ECUs, tasks and priority groups. Case 1 has only 16 states. Case 2 has a moderate state space, while Case 3 is a fairly large example. The experiment settings and converged results are described in Table 1.

Test	# tasks	# signals	# priority groups	# ECUs	# states in searching space	# episodes before convergence
Case1	4	4	1	2	16	3,005,800
Case2	7	7	5	3	1.71e8	13,177,700
Case3	20	20	5	4	1.05e26	54,503,600

Table 1: Experiment Description

To demonstrate the effectiveness of our algorithm, we first show the change of standard deviation with learning episode as in Figures 2 and 3. We divide the episode data points for Case 1 into 200 bins and those for Case 2 into 70 bins. This is because the algorithm converges in different numbers of episode in both cases. We choose different bin sizes so that each bin contains similar number of data points in either case. In both cases, the standard deviation decreases as the algorithm approaches to the end. That means our algorithm starts focusing on the optimal solution.

We also plot the histogram of optimal solutions for the first 50000 episodes and last 50000 episodes in both cases, shown in Figure 4. On the histogram, we use 6 bins with equal size. Bin

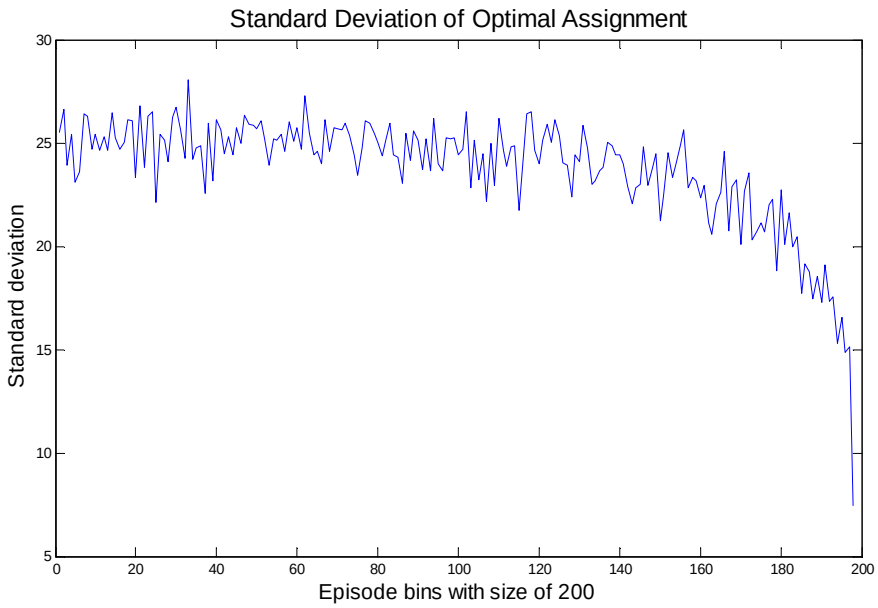


Figure 2: Changes in Standard Deviation of Optimal Assignments for Case 1

1 contains the optimal solution as it represents the bin with minimum total response time. Case 1 has limited solution space, but the algorithm tends to search equally across all possible solutions at the beginning. (Note: there is no possible solution for Case 1 falling in Bin 4 to 5.) When the algorithm is about to terminate because of convergence, it puts emphasis on the optimal solution (more points in Bin 1).

Similarly, the distribution for Case 2 is given in Figure 5. Case 2 has more states and is more representative than Case 1. Initially, the algorithm tends to perform random actions, so the graph looks like a Gaussian distribution. When the algorithm is about to finished, most of the efforts are spent near the optimal solution.

When we test the algorithm on the more complex Case 3, which has 20^{20} possible search states, the algorithm also converged within reasonable number of episodes (54 million compared to 13 million in Case 2). Unlike the situation in Figures 2 and 3, the standard deviation increases gradually. But, at the end of the learning period, the standard deviation starts to drop significantly. The reason is that the algorithm tries to explore more and more better actions with time. At last, the algorithm terminated because it satisfied our convergence condition (limited changes in 5000 consecutive valid solutions). Figure 6 shows the distribution of "optimal" solution at the first 50000 episodes and last 50000 episodes. Similarly as Case 1 and 2, at the end the system is trained to

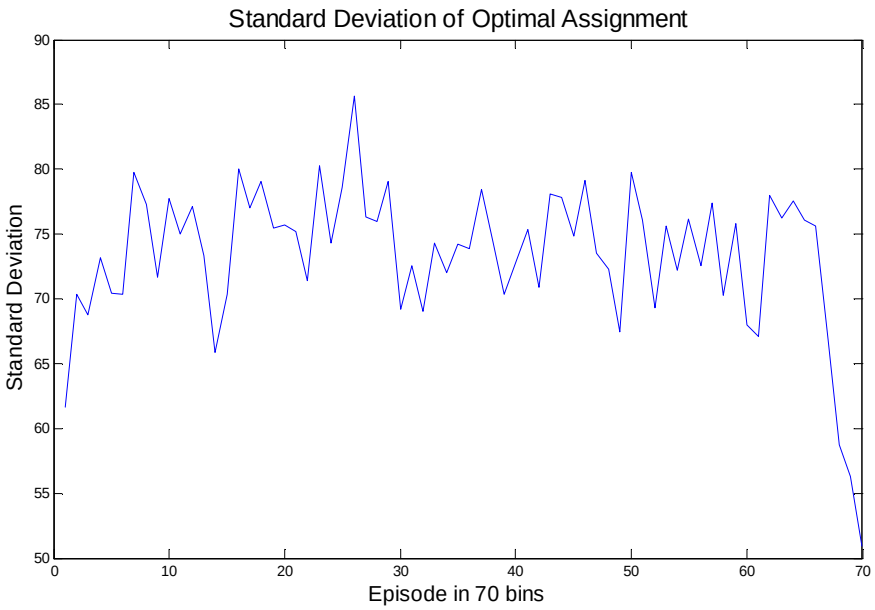


Figure 3: Changes in Standard Deviation of Optimal Assignments for Case 2

search “better” solutions. However, we should mention that there is a possibility for the algorithm to trap to local optimum for its huge state space. On the contrary, the algorithm tends to learn how to achieve the best solution with time. Longer learning time will help it get better result.

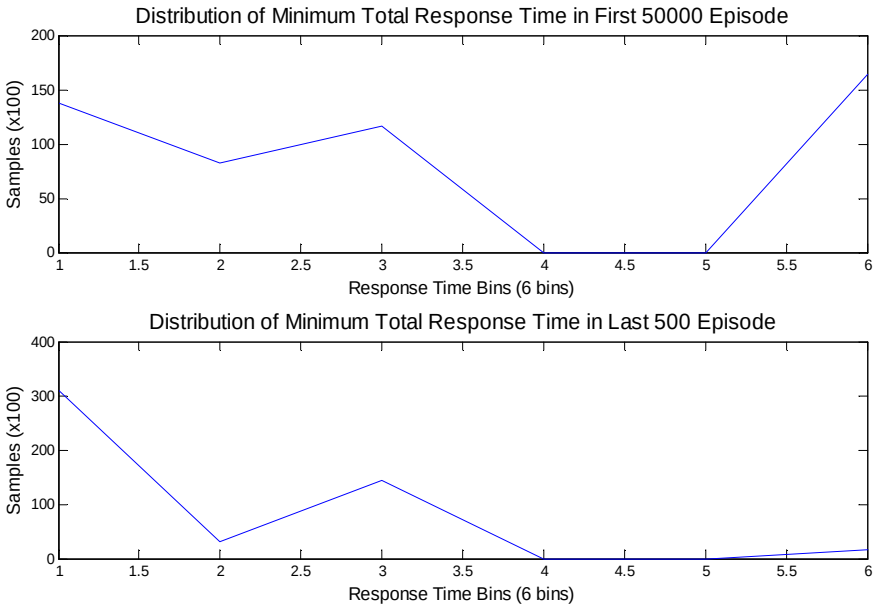


Figure 4: Distribution of Minimum Total Response Time for Case 1

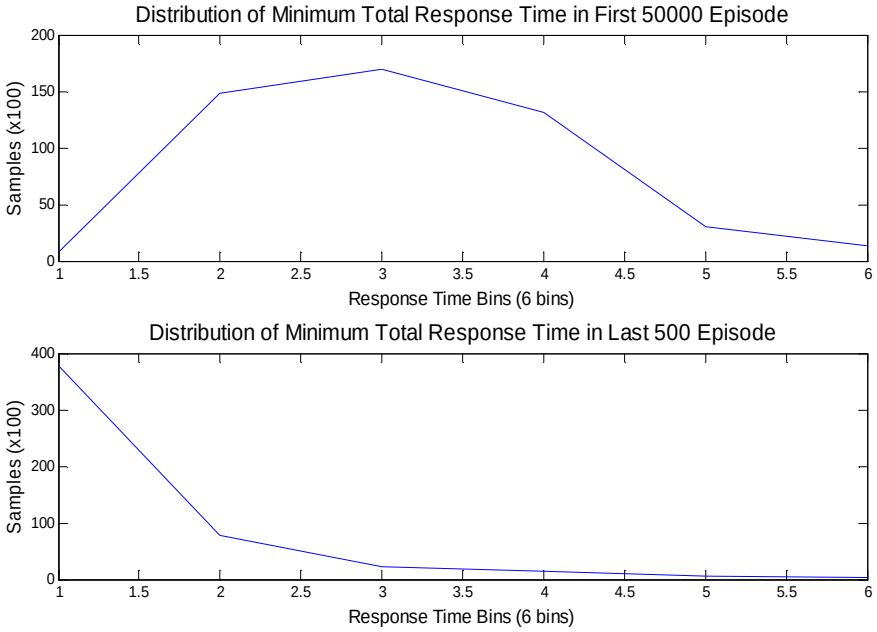


Figure 5: Distribution of Minimum Total Response Time for Case 2

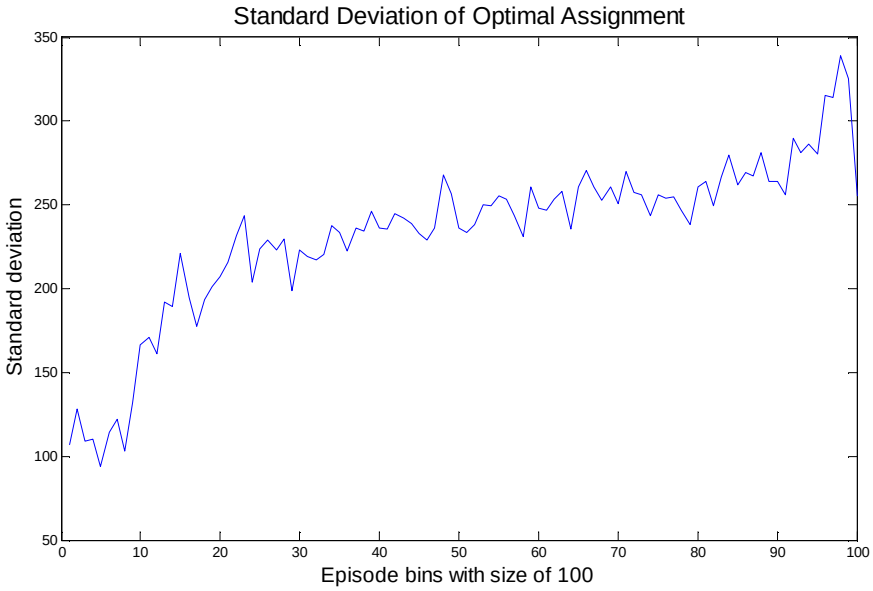


Figure 6: Changes in Standard Deviation of Optimal Assignments for Case 3

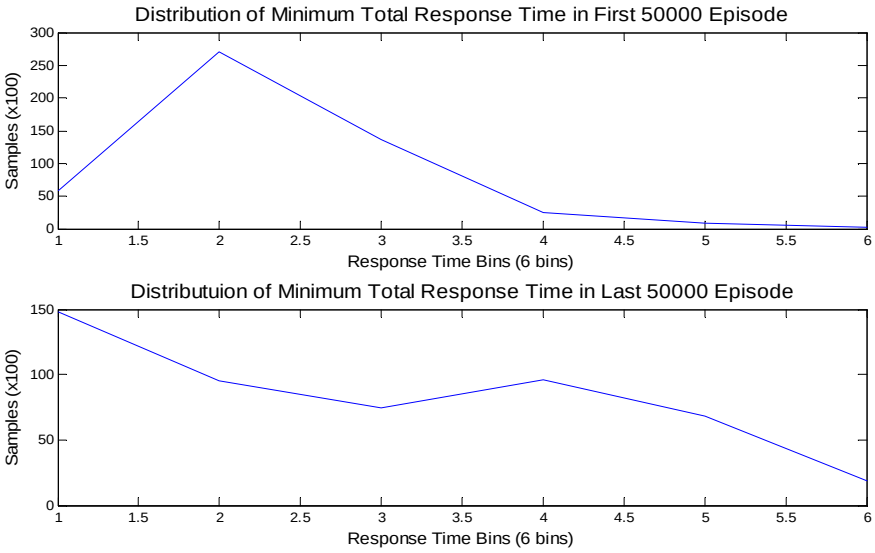


Figure 7: Distribution of Minimum Total Response Time for Case 3

3 Extensibility Optimization for Hard Real-Time Distributed Systems

3.1 Overview

Optimizing extensibility is important for embedded system design for several reasons. Firstly, in industrial settings it is often prohibitive to design a system without consideration of already existing systems. Moreover, as time progresses new functionality is introduced that requires existing functionality, and its allocation and architecture to remain in tact. Hence, providing a design that has as much room for such future functionality is desired. Secondly, current automated designs often focus on providing a *nominal* mapping. Of equal importance is the sensitivity of the solutions. By explicitly taking this into account through an extensibility measure, designs can be evaluated and optimized with respect to the sensitivity rather than a nominal point solution. In this work, the measure for the constraint is assumed to be given in a worst-case form, such that it can also be guaranteed that the mappings are feasible and will not violate any of the requirements.

Extensibility is defined as the amount by which the execution time of tasks can be increased without changing the system configuration while meeting the deadline constraints (as in [1]). With this definition, a design that is optimized for extensibility not only allows adding future functionality with minimum changes, but is more robust with respect to the variance of task execution times.

Analyzing and optimizing system extensibility have been addressed in the literature by several groups. Sensitivity analysis has been studied for priority-based scheduled distributed systems [18], with respect to end-to-end deadlines. Also, the evaluation of extensibility with respect to changes in the task execution times, when the system is characterized by end-to-end deadlines, is studied in [19]. These papers do not explicitly address system optimization. Task allocation, the definition of priorities, and the message configuration, are assumed as given.

For distributed systems with end-to-end deadlines, the optimization problem was partially addressed in [18], where the authors propose the use of genetic algorithms for optimizing priority and period assignments with respect to a number of constraints, including end-to-end deadlines

and jitter. In [7], an algorithm based on geometric programming was proposed for optimizing task and message periods in distributed systems, later extended in [8], to optimize jointly task and message allocations, as well as priority assignments. In [20] a design optimization heuristics-based algorithm for mixed time-triggered and event-triggered systems is proposed. The algorithm, however, assumes that nodes are synchronized. In [21], a SAT-based approach for task and message placement was proposed. The method provided optimal solutions to the placement and priority assignment. However, it did not consider signal packing.

The most relevant references to our approach are [22] and [1, 23, 24]. In the first work, task allocation and priority assignment are defined with the purpose of optimizing the extensibility with respect to changes in the task computation times. The proposed solution is based on simulated annealing and the maximum amount of change that can be tolerated in the task execution times without missing end-to-end deadlines is computed by scaling all task times by a constant factor. Also, a model of event-based activation for task and messages is assumed. In [1, 23, 24], a generalized definition of extensibility on multiple dimensions (including changes in the execution times of tasks, as in our research, but also period speedups and possibly other metrics) is presented. Also, a randomized optimization procedure based on a genetic algorithm is proposed to solve the optimization problem. These papers focus on the multi-parameter pareto optimization, and how to discriminate the set of optimal solutions. The main limitation of the proposed approach is however the complexity and the expected running time of the genetic algorithm proposed for the optimization. In addition, randomized optimization algorithms are difficult to control and give no guarantee on the quality of the obtained solution.

In contrast to them, our approach does not use randomized optimization, and works on much larger sized problems. Our algorithm consists of a first stage, based on MILP programming, where task placement (the most important variable with respect to extensibility) is optimized within the deadline and utilization constraints, and two heuristic algorithms, which then iteratively try to optimize signal-to-message packing and priority assignment, respectively. Our algorithm runs much faster than randomized optimization approaches (a 20x reduction with respect to simulated annealing in our case studies). As such, it is proven to be applicable to large-scale industrial systems as the case studies shown in the experimental section, which are of size comparable with the typical case of deployment of a set of additional functionality in a car. The shorter running time allows the use of the method not only for the optimization of a given system configuration,

but also for *architecture exploration*, where the number of system configurations to be evaluated and subject to optimization can be very large.

3.2 Problem Representation

The representation of the system is explained in Section 1. Next, we will describe our specific approach for this project.

3.2.1 Design Space and Extensibility Metric

The design problem can be defined as follows. Given a set of design constraints including:

- end-to-end deadlines on selected paths
- utilization bounds on nodes and buses
- maximum message sizes

explore the design space that includes:

- allocation of tasks to computational nodes
- packing of signals and allocation of messages to buses
- assignment of priorities to tasks and messages

to maximize *task extensibility*.

Task extensibility is defined as the weighted sum of each task's execution time slack over its period:

$$max. \quad S = \sum_{\tau_i \in \mathcal{T}} w_{\tau_i} \frac{\Delta c_{\tau_i}}{t_{\tau_i}} \tag{7}$$

where a task's execution time slack Δc_{τ_i} is defined as the maximum possible increase of its execution time c_{τ_i} without violating the design constraints, assuming the execution time of other tasks are not changed. w_{τ_i} is a preassigned weight that indicates how likely the task's execution time will be increased in future functionality extensions.

3.2.2 Formulation

Based on the Formulas (2), (3) and (4) for computing end-to-end latencies and response times, we construct a mathematical formulation that contains all the design variables. Part of the formulation is similar to the one in [8]: both explore the same set of design variables - task allocation, signal packing and message allocation, as well as task and message priorities. In [8], the problem was formulated as mixed integer linear programming (MILP). To reduce the complexity, the problem was divided into sub-problems and solved by a two-step approach.

However, in [8], the objective is to minimize end-to-end latencies, while in this work, we optimize task extensibility. The formulation of task extensibility with respect to end-to-end deadline constraints is a quite challenging task. In general, inverting the function that computes response times as a function of the task execution times is of exponential complexity in the simple case of single-CPU scheduling [25]. When dealing with end-to-end constraints, the problem is definitely more complex. A possible approach consists of a very simple (but possibly time-expensive) bisection algorithm that finds the sensitivity of end-to-end response times with respect to increases in task execution times (this is the solution used for performing sensitivity analysis in [18]).

Formally, if Δr_{ij} denotes the increase of task τ_j's response time r_{τ_j} when task τ_i's computation time c_{τ_i} is increased by Δc_{τ_i}, the end-to-end latency constraints and utilization constraints are expressed as follows:

$$\sum_{\tau_j \in p \wedge \tau_j \in (lp(\tau_i) \cup \{\tau_i\})} \Delta r_{ij} \leq d_p - l_p \qquad \forall p, \forall \tau_i \in \mathcal{T} \tag{8}$$

$$\frac{\Delta c_{\tau_i}}{t_{\tau_i}} + \sum_{\tau_j \in \mathcal{T}(e)} \frac{c_{\tau_j}}{t_{\tau_j}} \leq u_e \qquad \forall e, \forall \tau_i \in \mathcal{T}(e) \tag{9}$$

where $lp(\tau_i)$ refers to the set of tasks with priority lower than p_{τ_i} and executed on the same node

as τ_i, $\mathcal{T}(e)$ denotes the set of the tasks on computational node e, and u_e denotes the maximum utilization allowed on e.

The relation between Δr_{ij} and Δc_{τ_i} can be derived from Equation (3), as follows.

$$\begin{aligned}\Delta r_{ij} &= \sum_{\tau_k \in hp(\tau_j)} \left(\left\lceil \frac{r_{\tau_j} + \Delta r_{ij}}{t_{\tau_k}} \right\rceil - \left\lceil \frac{r_{\tau_j}}{t_{\tau_k}} \right\rceil\right) c_{\tau_k} \\ &+ \left\lceil \frac{r_{\tau_j} + \Delta r_{ij}}{t_{\tau_i}} \right\rceil \Delta c_{\tau_i} \qquad \forall \tau_j \in lp(\tau_i) \qquad (10)\\ \Delta r_{ii} &= \sum_{\tau_k \in hp(\tau_i)} \left(\left\lceil \frac{r_{\tau_i} + \Delta r_{ii}}{t_{\tau_k}} \right\rceil - \left\lceil \frac{r_{\tau_i}}{t_{\tau_k}} \right\rceil\right) c_{\tau_k} + \Delta c_{\tau_i} \qquad (11)\end{aligned}$$

For brevity, above formulas do not model task allocation and priority assignment as variables. In the complete formulation, they were expanded to include those variables.

Contrary to the problem in [8], in our case the formulation is nonlinear. It could be solved by non-linear solvers but the complexity is in general too high for industrial size applications. Therefore, we propose an algorithm that defines two steps: one in which mathematical programming is used, and a later refinement step, based on heuristics.

3.3 Optimization Algorithm

The flow of our algorithm is shown in Figure 8. First, we decide the allocation of tasks, since the choices of other design variables are restricted by task allocation. In the initial allocation stage, the problem is formulated as MILP and solved by an MILP solver. In the following signal packing and message-to-bus allocation stage, a heuristic is used. Finally, in the task and message priority assignment stage, an iterative method is defined to assign the priorities of tasks and messages. After all the above stages are completed, if the design constraints cannot be satisfied or if we want to further improve extensibility, the tasks can be re-allocated and the process repeated. Because of the complexity of the MILP formulation, we designed a heuristic for task re-allocation, based on the extensibility and end-to-end latency values obtained in the previous steps.

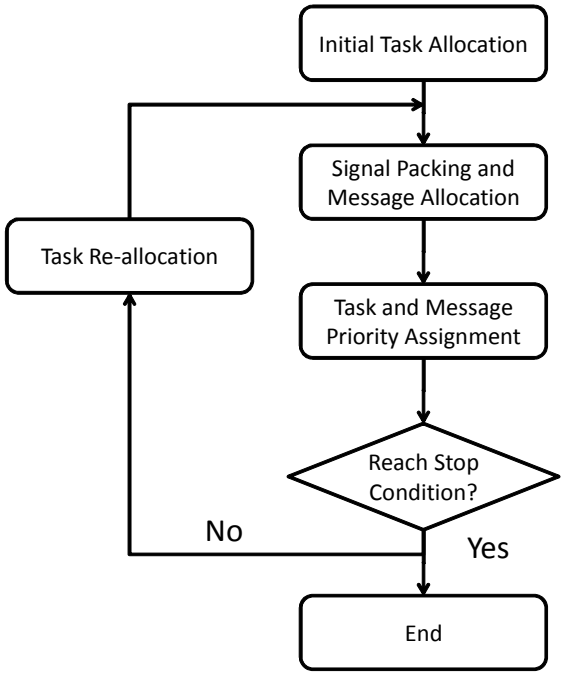


Figure 8: Algorithm Flow for Task Extensibility Optimization

3.3.1 Initial Task Allocation

In the initial task allocation stage, tasks are mapped onto nodes while meeting the utilization and end-to-end latency constraints. Utilization constraints are considered in place of the true extensibility metric to allow a linear formulation. In this stage, we also allocate signals to messages and buses assuming each message contains one signal only. The initial tasks and message priority assignment is assumed as given. In case the procedure is used to optimize an existing configuration, priorities are already defined. In case of new designs, any suitable policy, such as Rate Monotonic, can be used.

The MILP problem formulation includes the following variables and constraints:

Allocation constraints

$$\sum_{e \in E(\tau_i)} a_{\tau_i,e} = 1 \tag{12}$$

$$\sum_{b \in B(s_i)} a_{s_i,b} = g_{s_i} \tag{13}$$

$$1 - \sum_{j} \sum_{e \in \mathcal{E}} h_{src_{s_i},dst_{s_i,j},e} \leq g_{s_i} \tag{14}$$

$$0 \leq g_{s_i} \leq 1 \tag{15}$$

$$a_{\tau_i,e} + a_{\tau_j,e} - 1 \leq h_{\tau_i,\tau_j,e} \tag{16}$$

$$h_{\tau_i,\tau_j,e} \leq a_{\tau_i,e} \tag{17}$$

$$h_{\tau_i,\tau_j,e} \leq a_{\tau_j,e} \tag{18}$$

$$a_{s_i,b} + a_{s_j,b} - 1 \leq h_{s_i,s_j,b} \tag{19}$$

$$h_{s_i,s_j,b} \leq a_{s_i,b} \tag{20}$$

$$h_{s_i,s_j,b} \leq a_{s_j,b} \tag{21}$$

where $E(\tau_i)$ is the set of computational nodes that task τ_i can be allocated to, and $B(s_i)$ is the set of buses on which signal s_i can be transferred. The boolean variable $a_{\tau_i,e}$ indicates whether task τ_i is mapped onto computational node e, and the boolean variable $a_{s_i,b}$ represents whether signal s_i is mapped onto bus b. The value of the boolean variable g_{s_i} is 1 if s_i is a global signal, and 0 otherwise. $h_{\tau_i,\tau_j,e}$ defines whether τ_i and τ_j are on the same node e.

In detail, constraint (12) ensures that each task is mapped to one node and only one. Similarly, condition (13) enforces the mapping of global signals into a single bus. The definition of global signal is expressed by constraint (14) and (15). A signal is global if and only if its source task src_{s_i} and at least one of its destination tasks $dst_{s_i,j}$ are mapped into different nodes, then all of the corresponding $h_{src_{s_i},dst_{s_i,j},e}$ should be equal to 0. The following set of constraints (16), (17) , (18) ensures the consistency of the definitions of the h and a variables. Constraints (19), (20) , (21) enforce similar conditions on the set of signals.

Utilization constraints The following constraints enforce the utilization bounds on all nodes and buses considering the load of the current tasks (summation on the left-hand side of Equation (22)

and the additional load caused by extensions of the execution times ($z_{\tau_i,e}$, on the left-hand side of the equation). u_e and u_b are the utilization bounds on computational node e and bus b, respectively. Of course, the additional load caused by the extension Δc_{τ_i} must be considered only if the task is allocated to the node for which the bound is computed. This is represented by using an additional variable $z_{\tau_i,e}$, and the typical "big M" formulation in use in MILP programming for conditional constraints, where M is a large constant.

In our formulation, tasks can have different execution times depending on their allocation, and $c_{\tau_i,e}$ denotes the worst-case execution time of task τ_i on node e. Also, buses can have different speeds. c_{s_i} denotes the transmission time of the message that carries signal s_i on a bus with unit speed. At this stage, we assume each message will only contain one signal. The transmission time of that message on a bus with speed $speed_b$ is $c_{s_i}/speed_b$.

$$z_{\tau_i,e} + \sum_{\tau_j \in \mathcal{T}} a_{\tau_j,e} * c_{\tau_j,e}/t_{\tau_j} \leq u_e \tag{22}$$

$$\Delta c_{\tau_i}/t_{\tau_i} - M * (1 - a_{\tau_i,e}) \leq z_{\tau_i,e} \tag{23}$$

$$z_{\tau_i,e} \leq \Delta c_{\tau_i}/t_{\tau_i} \tag{24}$$

$$z_{\tau_i,e} \leq M * a_{\tau_i,e} \tag{25}$$

$$\sum_{s_i \in S} a_{s_i,b} * c_{s_i}/t_{s_i}/speed_b \leq u_b \tag{26}$$

End-to-end latency constraints

$$l_p \leq d_p \tag{27}$$

$$\sum_{\tau_i \in p} r_{\tau_i} + \sum_{s_i \in p} (r_{s_i} + t_{s_i} * g_{s_i} + t_{dst_{s_i}} * g_{s_i}) = l_p \tag{28}$$

$$\sum_{e \in \mathcal{E}} a_{\tau_i,e} * c_{\tau_i,e} + \sum_{\tau_j \in \mathcal{T}} \sum_{e \in \mathcal{E}} c_{\tau_j,e} * p_{\tau_i,\tau_j} * y_{\tau_i,\tau_j,e} = r_{\tau_i} \tag{29}$$

$$x_{\tau_i,\tau_j} - M * (1 - h_{\tau_i,\tau_j,e}) \leq y_{\tau_i,\tau_j,e} \tag{30}$$

$$y_{\tau_i,\tau_j,e} \leq x_{\tau_i,\tau_j} \tag{31}$$

$$y_{\tau_i,\tau_j,e} \leq M * h_{\tau_i,\tau_j,e} \tag{32}$$

$$0 \leq x_{\tau_i,\tau_j} - r_{\tau_i}/t_{\tau_j} < 1 \tag{33}$$

$$\sum_{b \in \mathcal{B}} (c_{s_i} + B_{max}) * a_{s_i,b}/speed_b + \sum_{s_j \in \mathcal{S}} \sum_{b \in \mathcal{B}} c_{s_j,b} * p_{s_i,s_j} * y_{s_i,s_j,b} = r_{s_i} \tag{34}$$

$$x_{s_i,s_j} - M * (1 - h_{s_i,s_j,b}) \leq y_{s_i,s_j,b} \tag{35}$$

$$y_{s_i,s_j,b} \leq x_{s_i,s_j} \tag{36}$$

$$y_{s_i,s_j,b} \leq M * h_{s_i,s_j,b} \tag{37}$$

$$0 \leq x_{s_i,s_j} - (r_{s_i} - \sum_{b \in \mathcal{B}} c_{s_i} * a_{s_i,b}/speed_b)/t_{s_j} < 1 \tag{38}$$

Latency constraints are derived from Equations (2), (3) and (4). r_{τ_i} is the response time of task τ_i, and r_{s_i} is the response time of the message that carries signal s_i. p_{τ_i,τ_j} is a parameter that denotes whether task τ_j has higher priority than task τ_j. We use a large constant M to linearize the relation $y_{\tau_i,\tau_j,e} = x_{\tau_i,\tau_j} * h_{\tau_i,\tau_j,e}$, similarly as in utilization constraints. Here x_{τ_i,τ_j} represents the number of interference from τ_j to τ_i. p_{s_i,s_j} and $y_{s_i,s_j,b}$ are similar parameters and variables for messages.

Objective function

$$max. \sum_{\tau_i \in \mathcal{T}} w_{\tau_i} * \Delta c_{\tau_i}/t_{\tau_i} \tag{39}$$

We recall here the objective function in (7), which represents the task extensibility. An alter-

native objective function can also include the optimization of latency, as shown in (40). K is the parameter used to explore the trade-off between task extensibility and latencies. The special case $K = 0$ is the original objective function (39).

$$max. \sum_{\tau_i \in \mathcal{T}} w_{\tau_i} * \Delta c_{\tau_i} / t_{\tau_i} - K * \sum_{p \in \mathcal{P}} l_p / d_p \tag{40}$$

In Section 3.4, we will report the experimental results with various values of K, to show the relationship between task extensibility and path latencies.

3.3.2 Signal Packing and Message Allocation

After the allocation of tasks is chosen, we use a simple heuristic to determine signal packing and message allocation. The steps are shown below.

1. Group the signals with the same source node and period as packing candidates.
2. Within each group, order the signals based on their priorities, then pack them according to the message size constraints (priorities are assumed given from an existing configuration or some suitable policy, as in the initial task allocation). The priority of a message is set to the highest priority of the signals that are mapped into it.
3. Assign a weight w_{m_i} to each message m_i based on its priority, transmission time and period. In our algorithm, we set $w_{m_i} = k_1 / p_{m_i} + k_2 * c_{m_i} / t_{m_i}$, where p_{m_i}, c_{m_i} and t_{m_i} are priority, transmission time on bus with unit speed and period of the message, k_1 and k_2 are constants. When multiple buses are available between the source and destination nodes, we allocate messages to buses according to their weights. Messages with larger weights are assigned first to faster buses.

Other more sophisticated heuristics or mathematical programming solutions have been considered. For instance, signal packing can be formulated as MILP as in [8]. However, from preliminary

experiments, there is no significant improvement that can outweigh the speed of this simple strategy.

3.3.3 Priority Assignment

In this stage, we assign priorities to tasks and messages, given the task allocation, signal packing and message allocation obtained from previous stages.

This priority assignment problem is proven to be NP-complete. Finding an optimal solution is generally not feasible for industrial-sized problems. Therefore, we propose an iterative heuristic to solve the problem.

The flow of this heuristic is shown in Figure 9. The basic idea is to define the local deadlines of tasks and messages over iteration steps, then assign priorities based on the deadlines. Intuitively, shorter deadlines require higher priorities and longer local deadlines can afford lower priorities.

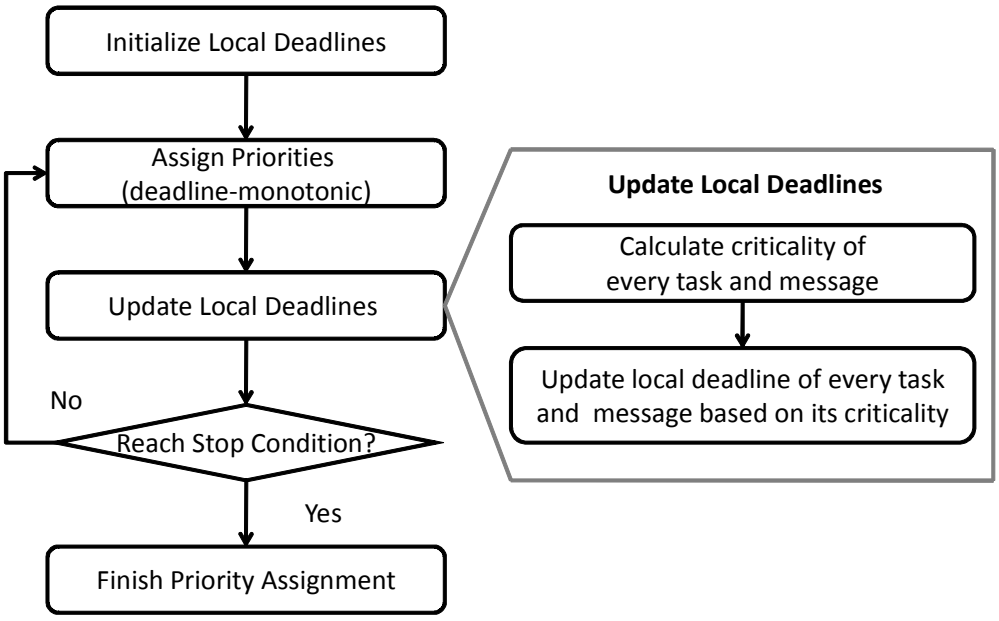


Figure 9: Iterative Priority Assignment Algorithm

Initially, the deadlines of tasks and messages are the same as their periods. Then, deadlines are modified, and priorities are assigned using the deadline-monotonic (DM) approach [26]. Of course, there is no guarantee that the DM policy is optimal in this case as for any system with

non-preemptable resources (the CAN bus), but there is no optimal counterpart that can be used here, and DM it is a sensible choice in the context of our heuristics.

During the iterations, deadlines are changed based on task and message *criticality*, as shown in Algorithm 3 and explained below.

Algorithm 3 UPDATE LOCAL DEADLINE (K_1)

```
1: Initialize the criticality ε of every task and message to 0
2: for all task τ_i do
3:    UB(Δc_τi) = t_τi * (u_e − Σ_{τj∈T(e)} c_τj/t_τj)
4:    c_τi = c_τi + UB(Δc_τi)
5:    for all task τ_j ∈ (lp(τ_i) ∪ {τ_i}) do
6:       update r_τj
7:    for all path p whose latency is changed do
8:       if l_p > d_p then
9:          for all tasks and messages o_j on p do
10:            ε_oj = ε_oj + w_τi * (l_p − d_p)/t_oj
11: for all task τ_i do
12:    ε^N_τi = ε_τi / max_{τi∈T}{ε_τi}
13:    d_τi = d_τi * (1 − K_1 * ε^N_τi)
14: for all message m_i do
15:    ε^N_mi = ε_mi / max_{mi∈M}{ε_mi}
16:    d_mi = d_mi * (1 − K_1 * ε^N_mi)
```

The criticality of a task or message, reflects how much the response times along the paths to which it belongs are affected by extensions in the execution times of other tasks. Tasks and messages with higher criticality are assigned higher priorities. To define the criticality ε of a task or a message, we increase the execution time of each task τ_i, by $UB(\Delta c_{\tau_i})$, the maximum amount allowed by utilization constraints.

$$UB(\Delta c_{\tau_i}) = t_{\tau_i} * (u_e - \sum_{\tau_j \in \mathcal{T}(e)} \frac{c_{\tau_j}}{t_{\tau_j}}) \tag{41}$$

Then the response time of τ_i and of lower priority tasks on the same node as τ_i is recomputed. The criticality of the affected task τ_j or message m_j (denoted as object o_j) is defined by adding up a term $w_{\tau_i} * (l_p - d_p)/t_{o_j}$ for each path p whose end-to-end latency exceeds the deadline after the increase $UB(\Delta c_{\tau_i})$, where w_{τ_i} is the weight of task τ_i. After repeating this operation for every task, the criticality of all tasks and messages is computed, denoted by ε_{o_j}. Criticality values are normalized, obtaining a value ε^N for each task and message and, finally, local deadlines are computed as $d = d * (1 - K_1 * \varepsilon^N)$. The procedure is shown in Algorithm 3. The parameter K_1 is initially set to 1,

then adjusted in the later iteration steps using a complex strategy that takes into account the number of iteration steps, the number of times the current best solution is found, and the number of times the priority assignment remains unchanged.

After local deadlines are updated, the stop condition is checked. If the number of iterations reaches its limit, or the upper bound of task extensibility is reached, the priority assignment will finish, otherwise we keep iterating.

The strategy of changing priorities based on local deadlines can also be found in [27]. Different from our algorithm, the goal is only to meet end-to-end latency constraints, therefore deadlines are updated based on the slack time of tasks or messages which indicate how much the local deadlines can be increased without violating latency constraints.

3.3.4 Task Re-allocation

After all the design variables are decided, we calculate the value of the objective function in Formula (7), and check the end condition. If the results are not good enough and the iteration limit has not been exceeded, we re-allocate the tasks and repeat the signal packing, message allocation and priority assignment.

We could use the same MILP based method for task re-allocation, by adding constraints to exclude the allocations that have been considered. However, solving the MILP is time consuming. To speed up the algorithm, we designed a local optimization heuristic that leverages the results of previous iterations, as shown in Algorithm 4.

Two operators are considered for generating new configurations: moving one task to a different node, or switching two tasks on different nodes. For each possible application of the previous operators on each task or task pair, that satisfies the utilization constraints, we compute the corresponding increase of the performance function Φ of Equation (40), which includes task extensibility and end-to-end latencies. The change that provides the largest increase of the performance function is selected. Parameter K_2 in cost function Φ provides the trade-off between task extensibility and end-to-end latencies. Initially, it is set to the same value as parameter K in Equation (40), which is

Algorithm 4 TASK RE-ALLOCATION (K_2)

Let $\Phi(M) = \sum_{\tau_i \in \mathcal{T}} w_{\tau_i} * UB(\Delta c_{\tau_i})/t_{\tau_i} - K_2 * \sum_{p \in \mathcal{P}} l_p/d_p$ for a mapping M

if current solution does not satisfy latency constraints **then**
  $K_2 += K_C$
$\Delta_{best} = MIN$
for all task τ_i and node e that τ_i is not on e **do**
  $\Delta_{\tau_i,e} = \Phi(M') - \Phi(M)$ {where M is the original mapping, M' is the new mapping after moving τ_i to e}
  if $\Delta_{\tau_i,e} > \Delta_{best}$ **then**
    best_move = τ_i *moves to* e
    $\Delta_{best} = \Delta_{\tau_i,e}$
for all task τ_i, τ_j that are not on the same node **do**
  $\Delta_{\tau_i,\tau_j} = \Phi(M') - \Phi(M)$ {M, M' similarly defined as above}
  if $\Delta_{\tau_i,\tau_j} > \Delta_{best}$ **then**
    best_move = *switch* τ_i *and* τ_j
    $\Delta_{best} = \Delta_{\tau_i,\tau_j}$
Execute best_move

used in the initial task allocation. If the current solution does not satisfy the end-to-end deadlines, we increase K_2 by a constant K_C to emphasize the optimization of latencies. K_C was set to 0.01 in our experiments.

3.3.5 Algorithm Complexity

The whole algorithm is polynomial except for the MILP based initial task allocation, which can be regarded as a preprocessing stage since we use heuristics for task re-allocation in following iterations.

Finding the *optimal* initial task allocation by MILP is a NP-hard problem. In practice, we set a timeout and use the best feasible solution. For the following stages, let $N_{\mathcal{S}}$ denote the number of signals, $N_{\mathcal{T}}$ denote the number of tasks, $N_{\mathcal{E}}$ denote the number of computational nodes, $N_{\mathcal{B}}$ denote the number of buses, and $N_{\mathcal{P}}$ denote the number of paths. The complexity of the signal packing and message allocation stage is $O(N_{\mathcal{S}} * log(N_{\mathcal{S}}) + N_{\mathcal{S}} * N_{\mathcal{B}})$. The complexity of the priority assignment is $O(N_{\mathcal{T}} * N_{\mathcal{P}} * (N_{\mathcal{T}} + N_{\mathcal{S}}) + N_{\mathcal{S}} * log(N_{\mathcal{S}}) + N_{\mathcal{T}} * log(N_{\mathcal{T}}))$ assuming the number of iterations is a constant. And the complexity of heuristic task re-allocation stage is $O(N_{\mathcal{E}} * N_{\mathcal{T}} * N_{\mathcal{P}} * (N_{\mathcal{T}} + N_{\mathcal{S}}) + N_{\mathcal{T}} * N_{\mathcal{T}} * N_{\mathcal{P}} * (N_{\mathcal{T}} + N_{\mathcal{S}}))$. This is the dominant stage. If we assume $N_{\mathcal{S}} \in O(N_{\mathcal{T}}^2)$, $N_{\mathcal{T}} \in O(N_{\mathcal{S}})$ and $N_{\mathcal{B}} \in O(N_{\mathcal{E}})$, which is usually the case in practice, we can simplify the total complexity of the

algorithm (except for the MILP based preprocessing stage) as $O(N_{\mathcal{E}} * N_{\mathcal{T}} * N_{\mathcal{P}} * N_{\mathcal{S}} + N_{\mathcal{T}} * N_{\mathcal{T}} * N_{\mathcal{P}} * N_{\mathcal{S}})$.

3.4 Case Studies

The effectiveness of the methodology and algorithm is validated in this section with an industrial example. In this case study, we apply our algorithm to an experimental vehicle that incorporates advanced active safety functions. This is the same example studied in [8].

The architecture platform consists of 9 ECUs (computational nodes) connected through a single CAN-bus with speed 500kb/s. For the purpose of our algorithm evaluation, we assumed that all ECUs have the same computational power, so that the worst case execution time of tasks does not depend on their allocation.

The subsystem that we considered consists of a total of 41 tasks executed on the ECUs, and 83 CAN signals exchanged between the tasks. Worst-case execution time estimates have been obtained for all tasks. The bit length of the signals is between 1 (for binary information) and 64 (full CAN message). The utilization upper bound of each ECU and bus has been set to 70%.

End-to-end deadlines are placed over 10 pairs of source-sink tasks in the system. Most of the intermediate stages on the paths are shared among the tasks. Therefore, despite the small number of source-sink pairs, there are 171 unique paths among them. The deadline is set at 300ms for 8 source-sink pairs and 100ms for the other two.

The experiments are run on a 1.7-GHz processor with 1GB RAM. CPLEX [28] is used as the MILP solver for the initial task allocation. The timeout limit is set to 1000 seconds. The parameter K in the MILP formulation is used to explore the trade-off between task extensibility and end-to-end latencies during initial task allocation. We test our algorithm with several different K values, and compare them with a system configuration produced manually. The results are shown in Figure 10.

The manual design is the initial definition of the system configuration provided by its designers. This initial configuration is not optimized, and there still exist paths that do not meet their

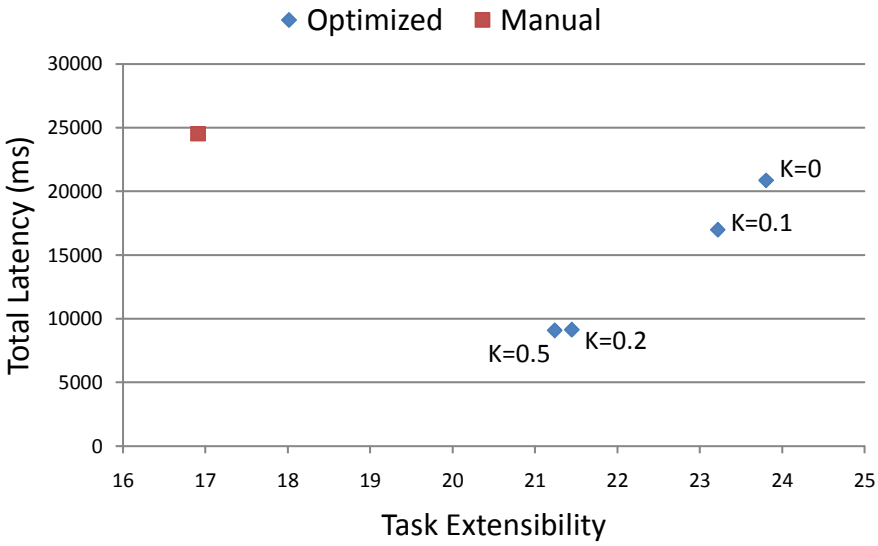


Figure 10: Comparison of Manual and Optimized Designs

deadlines. The total latencies of all paths is 24528.1ms and the task extensibility is 16.9113.

On the other side, in any of the four automatically optimized designs, all paths meet their deadlines. Different K values provide the trade-off between task extensibility and end-to-end latencies. When $K = 0$, we have the largest task extensibility at 23.8038, which is a 41% improvement over manual design. When $K = 0.5$, we have the shortest total end-to-end latency at 9075.46ms, which is 63% less than manual design. If a balanced design between extensibility and end-to-end latency is needed, intermediate values may be used. For $K = 0.1$, we obtain 37% improvement on task extensibility and 31% improvement on end-to-end latencies.

After the initial task allocation, each outer iteration of the signal packing and message allocation, priority assignment and task re-allocation takes less than 30 seconds for this example. And the optimization converges within 30 iterations for the various K values we tested. Figure 11 shows the current best task extensibility over 30 iterations for $K = 0$. Iteration 0 is the task extensibility after initial task allocation. The running time is 732 seconds for 30 iterations.

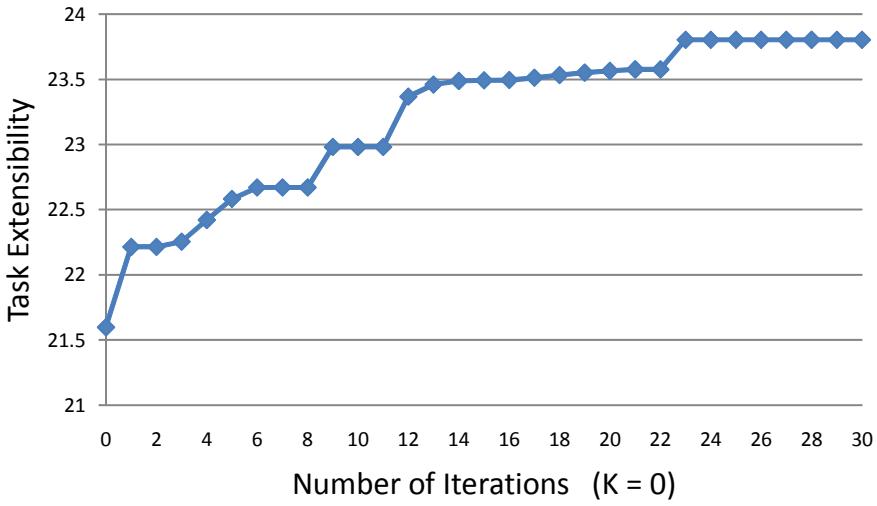


Figure 11: Task Extensibility over Iterations

4 Buffer-Sizing for Precedence Graphs on Restricted Multi-processor Architecture

4.1 Overview

This work solves a sub-problem in mapping for Parallel Heterogeneous Platforms (PHPs). The sub-problem is concerned with determining the buffer sizes between processing elements.

In the design flow, task allocation and scheduling are carried out before buffer sizing. Given a PHP with a number of processors and a set of tasks, a heuristic is used to allocate tasks onto processors and to schedule them. The scheduling algorithms utilized in the heuristic, such as [29], usually assume unlimited buffer sizes between processors, but this is not true in reality. Architectural platforms have finite-size buffers between processors. If the buffer size is too small, execution may deadlock. We call this kind of deadlock *artificial deadlock* [2], compared to *real deadlock* which can occur even if the buffer size were unlimited.

Prior work in this field mainly focuses on the buffer sizing problem in uni-processor platforms [3]. The previous work that deals with multiprocessor buffer minimization [4] does not consider *interleaving communication*, where two active tasks on different processors can communicate large amounts of data using one-place buffers.

In this project, we develop algorithms to address this problem. Theoretical as well as practical results are provided.

4.2 Problem Statements

4.2.1 Precedence DAG

The application, as the input of the buffer sizing algorithm, is represented as a *precedence DAG*, in which the vertices represent the tasks and the edges represent data dependencies. A precedence DAG is a common representation for the deployment of an application onto multiple processors. It

can be generated from statically schedulable dataflow descriptions, such as synchronous dataflow or cyclo-static dataflow.

4.2.2 Artificial Deadlock

Artificial deadlock [2] is a type of deadlock that occurs when the sizes of buffers between processors are reduced from infinity to some finite numbers. In buffer sizing, we want to optimize the objective function while avoiding artificial deadlock.

4.2.3 Problem Formulation

The buffer sizing problem is formulated as following. An instance of this problem is a 5-tuple $\langle V, E, W, P, M \rangle$. $V = \{v_1, v_2, \ldots, v_m\}$ is the set of vertices in the precedence DAG. $E = \{e_1, e_2, \ldots, e_n\}$ is the set of edges. We distinguish two disjoint subsets of E: $S = \{e | e \in E \wedge M(src(e)) = M(des(e))\}$ is the set of schedule edges, and $D = \{e | e \in E \wedge M(src(e)) \neq M(des(e))\}$ is the set of data edges. $W : D \rightarrow \Re^+$ is the weight function. M and E are acquired from the scheduling algorithm. $P = \{p_1, p_2, \ldots, p_l\}$ is the set of processors. $M : V \rightarrow P$ is the mapping from vertices to the processors that they are scheduled on.

We try to compute valid buffer sizes according to some minimum criteria without giving rise to artificial deadlock. If we use function $F : P \times P \rightarrow \Re^+$ to denote the buffer sizes between pairs of processors, the two problems that we are going to solve are:

- *Min max*: with $\langle V, E, W, P, M \rangle$ given, find a valid F such that $\max\{F(p_i, p_j) | \forall i, j\}$ is minimized.
- *Min total*: with $\langle V, E, W, P, M \rangle$ given, find a valid F such that $\sum\{F(p_i, p_j) | \forall i, j\}$ is minimized.

4.3 Observations

4.3.1 Write Blocked Cycles

In a precedence DAG, we classify the nodes which are blocked during execution into three categories.

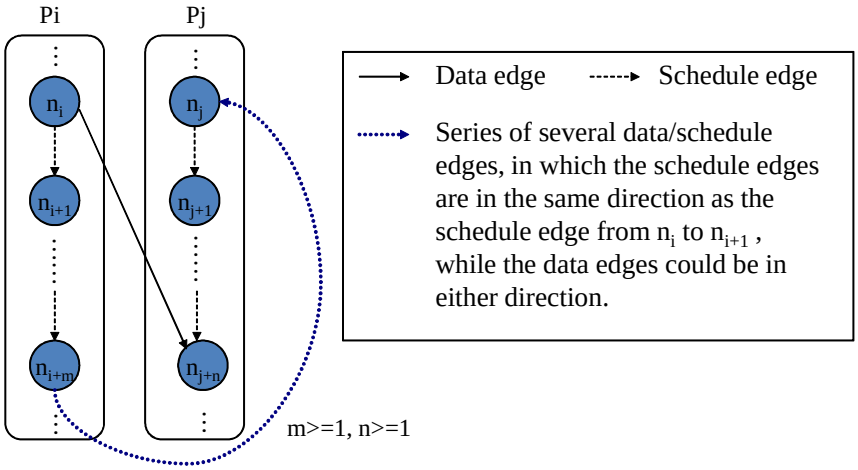
- read blocked node – the node is blocked because it can not read in enough tokens.
- write blocked node – the node is blocked because it can not finish writing all the produced tokens.
- scheduling blocked node – the node can not be fired because its previous node on the same processor has not finished execution.

We observed that it is impossible to have deadlock with only scheduling blocked nodes and read blocked nodes. Furthermore, if a precedence DAG has deadlock, it must have at least such a pattern called “write blocked cycle” (as shown in Figure 12), in which

- all the schedule edges are in the same direction;
- there must be one or more write blocked nodes, whose incoming degree is 0 in the cycle;
- there could be read blocked nodes, whose incoming degree is one or more in the cycle;
- if reversing the directed data edges from all the write blocked nodes, it becomes a directed cycle.

Based on the observation about write blocked cycle, we proved the following theorem by contradiction.

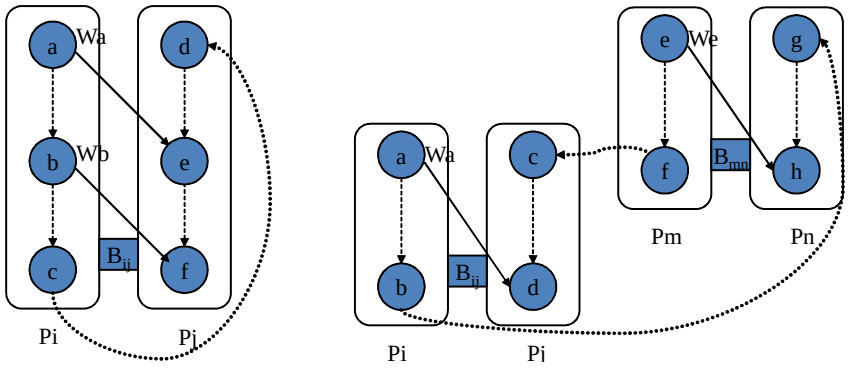
> Theorem 1: *There is artificial deadlock in a precedence DAG if and only if there exists a write blocked cycle in the graph.*



Buffer space from Pi to Pj < Token count on the write edge from n_i to n_{j+n}

Figure 12: Write Blocked Cycle

We can resolve the write blocked cycles by using enough communication or internal buffers. Figure 13 shows two different cases with write blocked cycles, and the way of increasing buffer size to resolve the deadlock.



Wa + Wb > Space(B_{ij}), a and b are write blocked.
Deadlock is solved by increasing buffer size to hold all the Wa + Wb tokens; or by increasing internal buffer in Pi or Pj.

Wa > Space(B_{ij}), We > Space(B_{mn}), a and e are write blocked.
Deadlock is solved by increasing the size of B_{ij} to hold all the Wa, or increasing the size of B_{mn} to hold all the Wb tokens; or using internal buffer.

Figure 13: Ways to Resolve Artificial Deadlock

4.3.2 Make Span

Make span is the maximum completion time for a set of processors. Insufficient buffer size can cause make span increase; on the other hand, it can also cause artificial deadlock. The question is

when the buffer size is enough to avoid artificial deadlock, will the make span also stay the same? The answer is no.

To prove it, we first assume that

- Interprocessor communication takes place through bounded size buffers with blocking reads and writes;
- Unlimited internal buffer space is available on each processor.

We have the conjecture that: For a task precedence graph, if insufficient buffer size leads to deadlock, reading and writing can be reordered in such a way that deadlock is eliminated and make span is not affected. If the conjecture is true, then we proved that the make span keeps the same if internal buffer space is unlimited.

The conjecture is proved to be false by the counterexample as shown in Figure 14. The example is scheduled in such a way that multiple paths are relatively critical. Reordering the reads and writes to eliminate the deadlock increases the length of some of the relatively critical paths, extending the make span, even if transition time $<<$ computation time of tasks.

In the example, edges (a,d) and (c,g) may be blocked due to insufficient buffer size. Without increasing the buffer size, there are 4 ways to resolve this:

1. Move communication (a,d) after b.
2. Move communication (a,d) before c.
3. Move communication (c,g) after d.
4. Move communication (c,g) before f.

Options 1 and 3 delay d and g by a large amount, and increase the makespan significantly Options 2 and 4 extend the critical paths that end at h and i.

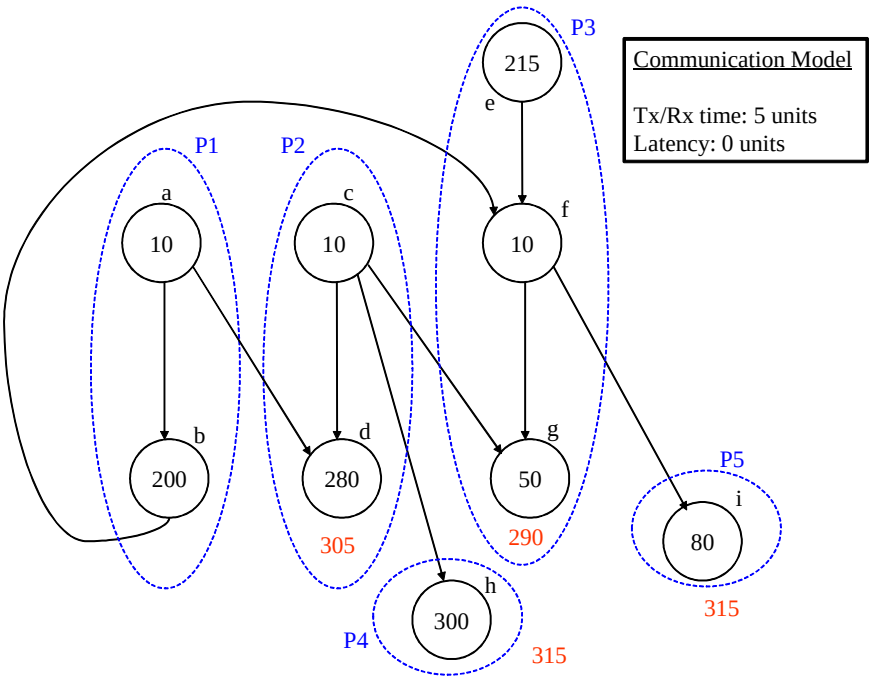


Figure 14: Counter Example - Make Span is Increased with Infinite Internal Buffer

4.4 Solving the Min Max Problem

4.4.1 Definitions

Free vertices are defined as the vertices with no incoming edges (e.g. a and c in Figure 15). *Free edges* are defiened as the edges starting from free vertices (e.g. (a,b), (a,d), (c,d), (c,e) in Figure 15). Our algorithm always deals with free edges. After a free edge is resolved, some other edges may become free.

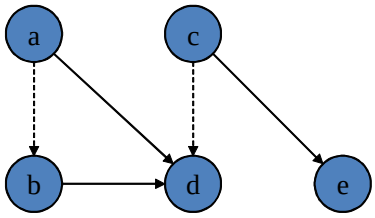


Figure 15: An Example of Free Vertices and Free Edges

Because any valid buffer size assignment should not produce artificial deadlock, we need to study how artificial deadlocks occur. A deadlock occurs when there is cyclic dependency. An

artificial deadlock is a special kind of deadlock where the cyclic dependency exists only because of buffer size. With the observation that a data edge implies bidirectional dependency if there is not enough buffer space for it, we transform the precedence DAG to a *dependency graph* by making all the data edges bidirectional. An example is shown in Figure 16.

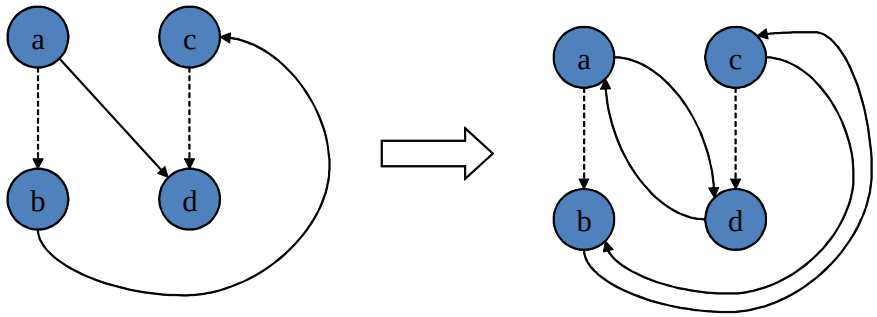


Figure 16: An Example of Tranforming Precedence DAG to Dependency Graph

We then proved the following theorem.

Theorem 2: *Artificial deadlock exists if and only if there is a cycle in the dependency graph (i.e. dependency cycle).*

We further observed that a dependency cycle must contain at least one data edge, because the precedence graph is acyclic. In addition, schedule edges do not affect buffers. Combining all these results, our algorithm to solve the min max problem only needs to deal with data edges in dependency cycles.

4.4.2 Algorithm Description

Our algorithm iterates over all the edges in the dependency graph. One edge is resolved and removed each time (hence, the edge set E changes over time). In iteration i, if we let $V_i^{free} = \{v | v \in V \wedge \bar{\exists} e \in E_i . des(e) = v\}$ and $E_i^{free} = \{e | e \in E_i \wedge src(e) \in V_i^{free}\}$, then our algorithm only needs to consider edges in E_i^{free}. Among all the edges in E_i^{free} that are also in dependency cycles, the algorithm always chooses the one e_i such that the buffer size required to complete the communication on e_i, $F_i(M(src(e_i)), M(des(e_i)))$, is minimal. It builds F_i by making it the same as F_{i-1} (initially, F_0 always returns 0), except that $F_i(M(src(e_i)), M(des(e_i)))$ becomes this new buffer size.

In detail, the way we use to remove edges are as follows. There are four types of free edges as shown in Figure 17:

1. Free schedule edge whose source has no other outgoing edges. For this type of free edge, we can just delete it. The reason is, as shown in Figure 17, *a* can finish immediately, then *b* becomes free.

2. Free data edge between two free vertices (ignoring the incoming data edges to the second vertex). We can also delete this type of free edge, because *a* and *c* can run simultaneously with interleaving communication.

3. Free data edge that is not type 2 and is not in a dependency cycle. This type of edge can be deleted, because *d* will be ready later, and *a* just needs to wait.

4. Free data edge that is not type 2 and is in a dependency cycle. We need to resolve blocking before deleting the edge: increasing buffer size if no space left; otherwise, useing the space first.

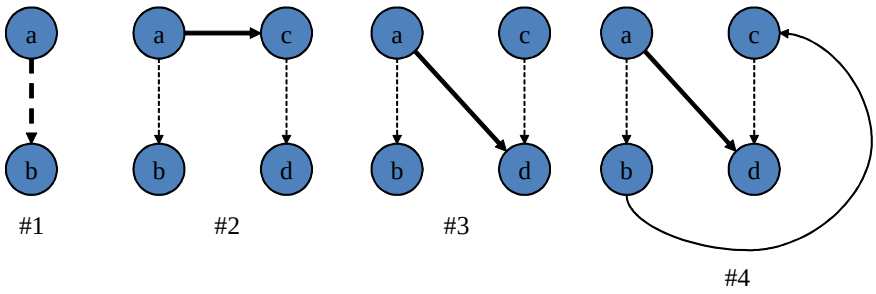


Figure 17: Four Types of Free Edges

In our algorithm, if edges of type 1, 2 or 3 exist, remove them first. Then there are only edges of 4 are left, choose one of them to resolve in a greedy manner: Among the edges of type 4, always pick the one *e* such that $F(M(src(e)), M(des(e)))$ is minimal after *e* is resolved.

4.4.3 Min Max Algorithm

To detect whether a data edge is in any dependency cycle, we develop an $O(|E|)$-time algorithm. With this, we develop an $O(|E|^2)$-time "Min Max"algorithm to compute valid buffer sizes while

minimizing the maximum buffer sizes. By "valid", we mean that there is no deadlock with such buffer sizes.

Linear Cycle Detection Algorithm To decide whether data edge from a to d is in a cycle: Without considering edge (a,d) in the dependency graph, can we find d by traversing the graph from a? Without considering edge (d,a) in the dependency graph, can we find a by traversing the graph from d? If either case is true, then return true; otherwise, false.

Quadratic Min Max Algorithm The Min Max Algorithm is decribed in Algorithm 5.

Algorithm 5 MIN MAX ALGORITHM

$\forall i \in P, \forall j \in P : space[i][j] = 0, fifo[i][j] = 0$
while E is not empty **do**
  $type = 0$, $sel\_src = None$, $sel\_des = None$, $min\_fifo = -1.0$
  for all edge $e = (src, des)$ **do**
    if src is free **then**
      if e is of type 1 **then**
        $type = 3$, $sel\_src = src$, $sel\_des = des$
      else
        if $type < 2$ and e is of type 2 **then**
          $type = 2$, $sel_src = src$, $sel_des = des$
        else
          if $type \leq 1$ **then**
            $new\_fifo\_size = calculate\_fifo(src, des)$
            if $minfifo < 0$ or $minfifo > fifo\_size$ **then**
              $type = 1$, $sel\_src = src$, $sel\_des = des$, $min\_fifo = fifo\_size$
  if $type = 3$ **then**
    remove edge (sel_src, sel_des)
  if $type = 2$ **then**
    remove edges (sel_src, sel_des) and (sel_des, sel_src)
  if $type = 1$ **then**
    if edge (sel_src, sel_des) is in a cycle according to the algorithm A_c **then**
      resolve blocking and update fifo and space
      remove edges (sel_src, sel_des) and (sel_des, sel_src)
  If sel_des becomes free, release the FIFO space that it has consumed

4.4.4 Proof of Optimality

We call the above algorithm A_m, and give the following theorem with detailed proof.

Theorem 3: *Assume that A_m terminates after iteration k. Let F be F_k computed by A_m. F is a valid buffer assignment, and $max\{F(p_i, p_j)|\forall i, j\}$ is minimized.*

Proof: We prove it by the way of induction: G is the complete precedence DAG. At step i, G_i is the sub-graph we have solved. $G - G_i$ is the sub-graph with only the remaining edges. F_i is the F function at step i.

- Base case: G_0 is empty. So G_0 is optimal.

- Induction step: Assume G_k is optimal, i.e. $max\{F_k(M(src(e)), M(des(e)))|e \in G_k\}$ is minimal. Prove G_{k+1} is also optimal:

G_{k+1} is obtained by either removing an edge of type 1, 2 or 3 (in which case G_{k+1} is obviously optimal), or updating buffer for an edge of type 4. In the latter case, we always pick an edge e_{k+1} such that $F_{k+1}(M(src(e_{k+1})), M(des(e_{k+1}))$ is minimum among such edges. Then,

$$\begin{aligned} & max\{F_{k+1}(M(src(e)), M(des(e)))|e \in G_{k+1}\} \\ = & max\{max\{F_k(M(src(e)), M(des(e)))|e \in G_k\}, \\ & F_{k+1}(M(src(e_{k+1})), M(des(e_{k+1}))\} \end{aligned} \tag{42}$$

is also minimal. So, G_{k+1} is optimal.

4.5 Solving the Min Total Problem

For the min total problem, in an intermediate step i, it cannot be determined which edge in E_i^{free} should be resolved so as to guarantee the total buffer size to be minimized at the end. This problem turns out to be an *NP*-hard problem.

We proved that the Min Total Problem is *NP*-hard by showing that any instance of the Feedback Arc Set (FAS) Problem [30], which is proven to be *NP*-hard, can be reduced to a min total problem in polynomial time. Details are omitted here.

Because of this, we developed another algorithm, which solves the min total problem in expo-

nential time.

The Min Total algorithm is very similar to the Min Max algorithm, except that if only free edges of type 4 are left, the Min Total algorithm picks them one by one in an arbitrary order, and each time it recursively computes the buffer size based on that choice. After finishing computing one buffer, it backtracks and picks another such edge to try again. This process ends when all possible sequences of choices are exhausted. The buffer with the minimum total size is returned. Because the exact min total problem is NP-hard, the algorithm has to be exponential.

4.6 Experimental Results

The algorithms are implemented in C++ with the Boost Graph Library (BGL). We have manually built a set of simple precedence DAGs, which covers most of the corner cases. We have also generated bigger random precedence DAGs with Task Graph For Free (TGFF) [31]. The result is shown in Figure 18. For those graphs, the above-described algorithms return the correct results to the Min Max problem and the Min Total problem, respectively. The big difference in time complexity is reflected with both small and large test cases. Furthermore, from the experiments, we obvserve that the results given by Min Max algorithm are good heuristics for those given by Min Total algorithm.

Test #	|V|	|P|	Min Max (A_m)			Min Total (A_t)			Lower Bound (L_n)	
			Max	Total	Time (s)	Max	Total	Time (s)	Max	Total
1	50	6	5	5	0.01	5	5	0.04	14	182
2	50	7	13	35	0.01	15	15	0.27	19	295
3	100	6	28	61	0.04	42	42	9.92	16	243
4	100	7	11	30	0.03	14	25	3.67	17	467
5	200	6	10	17	0.09	10	17	0.37	16	433
6	200	7	13	41	0.07	13	24	5.05	17	626
7	200	8	13	25	0.04	13	25	0.35	19	838
8	300	5	22	40	0.08	22	22	12.81	16	301
9	300	7	14	26	0.05	14	23	18.17	18	659
10	300	9	27	122	0.06	27	43	55.87	20	1232

Figure 18: Experiment Results of Buffer Sizing Algorithms

5 Conclusion and Future Work

We developed several algorithms to solve the mapping problem for distributed systems. The mapping space we explored includes task allocation, signal packing, message allocation, task and message scheduling, and buffer sizing.

We first developed a reinforcement learning algorithm to solve the allocation and scheduling of tasks and messages, while satisfying the latency constraints and optimizing the total response time. With careful state representation and reward design, the Q-learning algorithm converged within reasonable learning episodes and provided near optimal solutions for all test cases in different scales. One of the intriguing features of reinforcement learning is that it does not have a strong model assumption and the algorithm will work as long as it gets feedback from the environment. The whole learning procedure is an interactive process, and can adapt to dynamic system changes. Currently, the system performance estimation is based on Newton's Method. And in practical, the computation complexity of Newton's Method dominates the whole learning process. However, for reward calculation, high accuracy is not always necessary. Further optimization can be performed in the future.

We then presented a mathematical framework for defining an extensibility metric and for solving the related optimization problem in hard real-time distributed systems, by exploring task allocation, signal packing and message allocation, as well as task and message priorities. We formulated the mapping as a standard optimization problem, then proposed an algorithm based on mixed integer linear programming and heuristics. It was shown by an industrial cases study that this framework can effectively maximize extensibility while meeting the design constraints such as end-to-end latency constraints and utilization constraints. In the future, we will test our algorithms for multiple-bus examples. We also plan to extend our framework to include not only task extensibility but also message extensibility. Further, we would like to consider task and message scalability (i.e., how many new tasks and messages can be added to an existing system).

The allocation and scheduling procedures assume unlimited buffers sizes between processors. In real systems, we need to solve the buffer-sizing problem afterward. We developed the quadratic Min Max algorithm and exponential Min Total algorithm to determine the buffer sizes between

processing elements to avoid artificial deadlock and minimize the largest/total buffer sizes, respectively. Experimental results show that results given by Min Max algorithm are close approximations of those given by Min Total algorithm.

Overall, these three projects have shown the importance of formulating mapping problem and optimizing it during the platform-based design process. Specifically for hard-real time systems, a set of system metrics such as end-to-end latencies and extensibility can be optimized by designing automatic algorithms and utilizing mathematical tools. We think the methodology of separating functionality and architecture, then bridging them through a formal and automated mapping process can be generally applied to many application domains - besides the real-time systems introduced in this work, there are also works related to multimedia domain, communication domain, etc. The impact of mapping keeps increasing because of the trend of using more parallel systems, e.g., multi-core systems. And there are many interesting topics in this area. For instance, how to find right abstraction level for mapping, how to analyze the semantics during mapping to insure design correctness, how to refine a mapped system to implementation, etc. These topics form the general scope of our future work. Some of the related work can be found in [32, 33].

References

[1] R. Racu A. Hamann and R Ernst. A formal approach to robustness maximization of complex heterogeneous embedded systems. In *Proc. of the CODES/ISSS Conference*, October 2006.

[2] M. Geilen and T. Basten. Requirements on the Execution of Kahn Process Networks. In P. Degano, editor, *Proc. of the 12th European Symposium on Programming*, 2003.

[3] S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for signal processing systems. Technical Report CS-TR-4063, 1999.

[4] Marleen Adé, Rudy Lauwereins, and J. A. Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. In *DAC*, pages 64–69, 1997.

[5] OSEK. OS version 2.2.3 specification. Available at `http://www.osek-vdx.org`, 2006.

[6] R. Bosch. CAN specification, version 2.0. Stuttgart, 1991.

[7] Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *Proc. of the 44th DAC Conference*, 2007.

[8] Wei Zheng, Qi Zhu, Marco Di Natale, and Alberto Sangiovanni-Vincentelli. Definition of task allocation and priority assignment in hard real-time distributed systems. In *Proc. of the IEEE RTSS Conference*, pages 161–170, Washington, DC, USA, 2007. IEEE Computer Society.

[9] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design and Test of Computers*, 18(6):23–33, 2001.

[10] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[11] M. Gonzalez Harbour, M. Klein, and J. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1), January 1994.

[12] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Syst.*, 35(3):239–272, 2007.

[13] Razvan Racu, Marek Jersak, and Rolf Ernst. Applying sensitivity analysis in real-time distributed systems, 2005.

[14] A. Burns K. W. Tindell and A. J. Wellings. Allocating hard real-time tasks: An np-hard problem made easy, 1992.

[15] Giorgio Buttazzo Enrico Bini, Marco Di Natale. Sensitivity analysis for fixed-priority real-time systems, 2006.

[16] CJCH Watkins. Thesis: Learning from delayed rewards. 1989.

[17] Peter Dayan. The convergence of td() for general . *Real-Time Systems*, 8:341–362, 1992.

[18] Razvan Racu, Marek Jersak, and Rolf Ernst. Applying sensitivity analysis in real-time distributed systems. In *Proc. of the RTAS Conference*, San Francisco (CA), U.S.A., March 2005.

[19] R. Yerraballi and R. Mukkamalla. Scalability in real-time systems with end-to-end requirements. In *Journal of Systems Architecture*, volume 42, pages 409–429, 1996.

[20] Traian Pop, Petru Eles, and Zebo Peng. Design optimization of mixed time/event-triggered distributed embedded systems. In *Proc. of the CODES+ISSS Conference*, New York, NY, USA, 2003. ACM Press.

[21] Alexander Metzner and Christian Herde. Rtsat– an optimal and efficient approach to the task allocation problem in distributed architectures. In *Proc. of the IEEE RTSS Conference*, Washington, DC, USA, 2006.

[22] I. Bate and P. Emberson. Incorporating scenarios and heuristics to improve flexibility in real-time embedded systems. In *12th IEEE RTAS Conference*, pages 221–230, April 2006.

[23] R. Racu A. Hamann and R Ernst. Multi-dimensional robustness optimization in heterogeneous distributed embedded systems. In *Proc. of the 13th IEEE RTAS Conference*, April 2007.

[24] R. Racu A. Hamann and R Ernst. Methods for multi-dimensional robustness optimization in complex embedded systems. In *Proc. of the ACM EMSOFT Conference*, September 2007.

[25] Enrico Bini, Marco Di Natale, and Giorgio Buttazzo. Sensitivity analysis for fixed-priority real-time systems. In *Euromicro Conference on Real-Time Systems*, Dresden, Germany, June 2006.

[26] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[27] J.J.G. Garcia and M. G. Harbour. Optimized priority assignment for tasks and messages in distributed hard real-time systems. In *3rd Workshop on Parallel and Distributed Real-Time Systems*, 1995.

[28] ILOG CPLEX Optimizer. http://www.ilog.com/products/cplex/.

[29] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):175–187, 1993.

[30] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[31] Robert P. Dick, David L. Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *CODES*, pages 97–101, 1998.

[32] Qi Zhu. *Optimizing Mapping in System Level Design*. PhD thesis, EECS Department, University of California, Berkeley, Sep 2008.

[33] Abhijit Davare. *Automated Mapping for Heterogeneous Multiprocessor Embedded Systems*. PhD thesis, EECS Department, University of California, Berkeley, Sep 2007.