An Algorithm for Assigning Real-time Tasks with Timing Constraints into Dynamically Reconfigurable Processor

Tomoya Kitani†, Keisuke Nishi‡, and Teruo Higashino‡
†Shizuoka University, Japan ‡Osaka Univeristy, Japan

Abstract - In this paper, we focus on the problem of implementing a periodic concurrent system with timing constraints into multi-context dynamically reconfigurable processors (DRP). A concurrent system has multiple tasks that can be executed in parallel. Moreover, some tasks in a specific set of processes might be required to synchronize each other. We propose a method for assigning tasks into a multi-context DRP such that timing constraints of the system are satisfied and the size of the program area required on each context for implementing the given system is minimized. However this problem is a combinatorial problem. Thus we propose a heuristic algorithm for solving the problem efficiently. Experimental results show that the proposed method can derive a quasi-optimal assignment in a short time.

Keywords: real-time system, dynamically reconfigurable processor, task assignment, heuristic algorithm

1 Introduction

Recently it has been required to manufacture a wide variety of embedded systems in small quantities in a short period of time. For the purpose, many embedded systems have been implemented to reconfigurable devices. With the advent of modern field programmable devices, many systems are implemented using small sized reconfigurable devices such as Dynamically Reconfigurable Processors (DRPs). These devices have many logical areas and can utilize these areas efficiently. A system can be implemented into a DRP by decomposing and allocating its multiple contexts into its logical areas.

Many systems are generally real-time systems with timing constraints. A proper decomposition or implementation of a real-time system into a multi-context DRP requires that the decomposed contexts satisfy the timing constraints of the given system. This is usually a hard task since the execution of decomposed contexts involves context switching and proper scheduling by considering synchronization constraints of the system. Moreover, a proper decomposition can take into account reducing the size of the program area required on each context for implementing the given system.

In this paper, a behavior (or a specification) of a system is assumed to be given as a set of periodic concurrent processes with timing constraints [1], [2]. Periodic systems are used in many real-time application areas such as communication systems, multimedia systems, routers and so on [3]–[5]. We assume that all processes have the same time period N_t. A process consists of a finite length of sequences of tasks, which can be represented in the form of a tree structure. A task corresponds to a functional module of the system. We assume that each task has several attributes such as its starting time, execution time, size (representing the program area necessary for implementing the task), timing constraint, temporal ordering relation with other tasks in the same process, and synchronizing condition with specific processes in the system.

We focus on a problem of decomposing a given periodic concurrent system and assigning its tasks into a multi-context DRP. The multi-context DRP can change its contexts dynamically in order to switch executable sub-modules. All the tasks on the same context can be executed in parallel. However, tasks assigned to different contexts cannot be executed simultaneously. Thus, we need assign those tasks carefully into a multi-context DRP so that all the tasks do not violate their timing constraints. We present a method for decomposing a given concurrent system and assigning its tasks into a multi-context DRP in such a way that the timing constraints of the system hold and that the maximum size of the programmable logic areas used on the target contexts is minimized. We propose a heuristic algorithm to solve this problem since it is a combinatorial problem. The proposed algorithm consists of three steps. First, given a system, the algorithm derives all tasks which satisfy the timing constraints of the system. Next, it constructs a task dependence graph about tasks' execution time range satisfying their timing constraints. Then, it assigns the tasks into contexts of a DRP according to the graph.

In the experimental results, we have confirmed that the proposed heuristic algorithm derives quasi-optical assignment results for large size examples in short time.

2 Related Work

A lot of research work has been carried out on partitioning and scheduling systems into reconfigurable systems [6]–[13]. The work in [6] targets a general multi-context reconfigurable architecture and focuses on context scheduling considering the overheads of context switching.

The work in [7] deals with temporal partitioning and scheduling of data flow graphs into reconfigurable computers. In particular, the work considers minimizing the additional number of configurable logic blocks of an FPGA.

In [8] a method for the synthesis and temporal partitioning of reconfigurable systems is provided. The method considers minimizing the number of ports of a DRP internal memory. In [9], a 0-1 non-linear programming model is given for temporal partitioning and high-level synthesis for implementing a system of reconfigurable processors. The model considers minimizing the size of data communication among partitioned tasks. A data scheduler for multi-context reconfigurable architectures is provided in [10]. The scheduler minimizes the size of data communication between DRP's external memory and its Programmable Logic Area (PLA).

In [11] and [12] partitioning methods are presented for reconfigurable systems considering parallelization of tasks, accurate reconfiguration overhead, and minimization of the total computation time of the system. In [13] a method for task scheduling with configuration prefetching during reconfiguration overhead is given. An ILP model for minimizing the total computation time of the system is provided. The model takes into account reconfiguration overhead and a given limit on the size of the programmable logic area.

Similar to our work, the above described work targets reconfigurable architectures. However, the system models considered in the above work consider order relations among tasks for tasks with no timing constraints. The work given in [5] uses a similar task model as the one presented in this paper, i.e. concurrent, periodic, and with time constraints. However, unlike our work, the work given in [5] does not consider reconfigurable architectures. In this paper, we use concurrent periodic task model with timing constraints over multi-context reconfigurable processors.

3 Target concurrent system and DRP

3.1 Real-time system model

3.1.1 Definition of system

In this paper, a concurrent periodic real-time system is described as a 4-tuple $System = (Processes, Tasks, clock, N_t)$. $Processes$ denotes a set of processes. $Tasks$ denotes a set of tasks used in the system. $clock$ is an integer counter that counts the elapsed time from the initial state of the system, and N_t is an integer that denotes the time period of the system.

A process in $Processes$ consists of a finite length of sequences of tasks in $Tasks$ that can be represented in the form of a tree structure. The root node of a tree structure represents the initial state. A sequence of consecutive tasks starting from the initial state (root node) and ending at a leaf node is called a *path*. All processes of the system have the same time period N_t. Each process executes tasks in one of its paths until it reaches the end of the path. If the time period N_t has passed before reaching the end of the path, $clock$ is reset to 0 and the process returns to its initial state. Note that we assume any process does not have loops. A simple example of a concurrent system `System1` is shown in Fig. 1. `System1` consists of 2 processes: `Process1` and `Process2`. `Process1` consists of two paths {`P11` and `P12`}, and `Process2` consists of two paths {`P21` and `P22`}.

In our model, we assume some tasks can synchronize among a specific set of processes. The synchronization relation as well as the behavior of each process is formally specified as follows using the operators in LOTOS language [14].

$$S ::= (S \ |[task\_list]|\ S) \ \mid \ (S \ ||| \ S) \ \mid \ P$$
$$P ::= \ task;\ P \ \mid \ (P[]P) \ \mid\mid \ task$$

Here, S denotes a concurrent system, and P denotes a process. In S, $|[task\_list]|$ denotes the synchronous parallel operator where $task\_list$ is a list of tasks to be synchronized

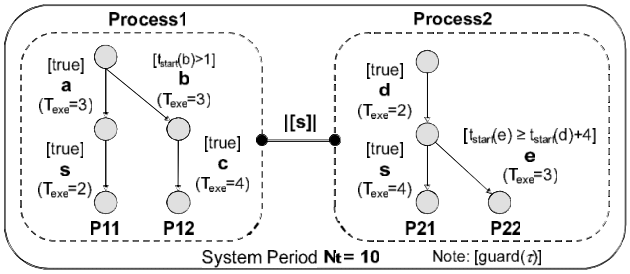


Figure 1: An example of a system, `System1`

between its operator's both sides of processes. If the same task s is specified in two processes $P1$ and $P2$, and if those processes are combined with the synchronous parallel operator $P1|[s]|P2$, the task s in $P1$ and $P2$ must be executed simultaneously. The operator $|||$ is the asynchronous parallel operator, and it denotes that its operator's both sides of processes can run in parallel without any synchronization. In P, the operator $[]$ is the choice operator, and $(P1[]P2)$ denotes that either $P1$ or $P2$ is executed. The choice operator $[]$ corresponds to a branch in a given process's tree structure. $task; P$ denotes the sequential composition of tasks.

Each task τ has a variable $t_{start}(\tau)$ and four attributes ($T_{exe}(\tau)$, $\tau_{prev}(\tau)$, $guard(\tau)$, $size(\tau)$). Although each task may appear in many paths of a given process if it can appear only once in any path of the process, here we assume that each task can appear only once in each process. The integer variable $t_{start}(\tau)$ denotes the starting time of task τ. $T_{exe}(\tau)$ is an integer value denoting the duration (or execution time) of task τ. Here, we assume the execution of a task is not interrupted by other tasks. $\tau_{prev}(\tau)$ denotes the previous task of τ. A $guard(\tau)$ denotes timing constraints that τ has to satisfy, and it is represented as a logical product of timing constraints, each of which is represented as a linear inequality over variables $\{t_{start}(\tau')|\tau'$ is an ancestor task of $\tau\}$. $size(\tau)$ denotes the size of τ, and it corresponds to the size of the program area necessary for implementing τ.

In `(Process1|[s]|Process2)` in Fig. 1, task `s` in `Process1` and `Process2` must be executed simultaneously. When the synchronous task `s` is executed simultaneously, both the guards in the two processes must hold. Given a system of n processes, $pr_1, \cdots, pr_n$, each of which has a set of finite lengths' paths $Paths^{pr_1}, \cdots, Paths^{pr_n}$, a *path-combination* is a list of n paths $\{< p_1, \cdots, p_n > \mid p_1 \in Paths^{pr_1}, \cdots, p_n \in Paths^{pr_n}\}$. All (syntactical) path combinations in Fig. 1 are `m1:<P11,P21>`, `m2:<P11,P22>`, `m3:<P12,P21>`, and `m4:<P12,P22>`.

However, the path combinations `m2` and `m3` cannot be executed until their leaf nodes since the synchronous task `s` cannot be synchronized between two processes. The path combinations `m1` and `m4` are syntactically possible path combinations. We describe about behavior of a system as follows.

3.1.2 Behavior of system

A task τ is $valid$ ($executable$) in a path-combination m, denoted $valid_{task}(\tau, m)$, if and only if it satisfies the following

three constraints:

The previous task $\tau_{prev}(\tau)$ has finished its execution.

$guard(\tau)$ is true.

τ ends its execution before the time period N_t.

Those constraints can be represented using the following linear constraints:

$$\begin{cases} guard(\tau) \wedge (t_{start}(\tau) + T_{exe}(\tau) \leq N_t) \\ \quad (\tau \text{ is the initial task of a path}). \\ (t_{start}(\tau') + T_{exe}(\tau') \leq t_{start}(\tau)) \ \wedge \ guard(\tau) \\ \quad \wedge (t_{start}(\tau) + T_{exe}(\tau) \leq N_t) \\ \quad (\text{otherwise}) \ (\tau' \text{ denotes } \tau_{prev}(\tau)). \end{cases}$$

A *valid path* of a process consists of a sequence of only valid tasks.

If a path combination m which consists of valid paths satisfies the following conditions, m is called a *valid path combination*:

m contains all synchronized tasks $Task_{sync}(\tau)$ of each task τ in m.

Each task τ which should be synchronized with tasks can start at the time when the synchronized tasks start.

The latter constraint can be represented using the following linear constraint:

$$\forall \tau' \in Task_{sync}(\tau) \, (t_{start}(\tau) = t_{start}(\tau'))$$

For this reason, a path combination which consists of only valid paths could not be valid.

A system is execute a valid path combination each period. If the system executes tasks in any invalid path combination, the system cannot reach their leaf nodes before the time period N_t has passed. This corresponds to either case of (i) a deadlock occurs, or (ii) a given task cannot be executed before the time period N_t has passed. Here, we focus on the implementation of all tasks in all valid path combinations of a given concurrent system.

3.2 Dynamically Reconfigurable Processors

Recent DRP consists of multiple contexts with several tiny processors called Processing Elements (PEs). The contexts of a DRP represent logical configurations of a PLA, and the DRP can switch its contexts dynamically with small overheads. Recently, many DRP's have been developed by companies and research institutes [15]–[19]. NEC Elect. Corp. has developed a multi-context DRP with the same program area size [15]. IPFlex Inc. has developed DAPDANA-2 [16], Singh et al. at U.C. Irvine have developed MorphoSys [17], Goldstein et al. in Carnegie Mellon Univ. have developed PipeRench [18], and Mei et al. in IMEC have developed ADRES [19]. DRPs can implement the systems at their programmable logic areas efficiently. Additionally, DRPs can hold many configurations, which are context assignments of a given system, without changing their logical area size. For example, the system implemented into a DRP can handle computation load flexibly by utilizing several configurations that mount the same function but with different performance and power consumption.

In this paper, we adopt a multi-context DRP architecture with the same program area size and a constant switching overhead, which is close to the multi-context DRP model of NEC Elect. Corp. in [15]. We define the target multi-context DRP as $DRP = (N_c, max_{cont}, T_{switch})$, where N_c denotes the number of contexts of the DRP, and max_{cont} and T_{switch} denote the size limit of the contexts and the context switching overhead, respectively.

When we implement a concurrent system defined above on the target multi-context DRP, it might occur a case that we cannot execute all valid path combinations of a given concurrent system due to the shortage of contexts, the context switching overhead and/or the size limit of contexts. For a given concurrent system Sys, if an implementation Imp on a multi-context DRP can execute all valid path combinations in Sys, then we say that Imp is a correct implementation of Sys. Hereafter, we discuss about a method to derive a correct implementation Imp on a target multi-context DRP from a given concurrent system Sys.

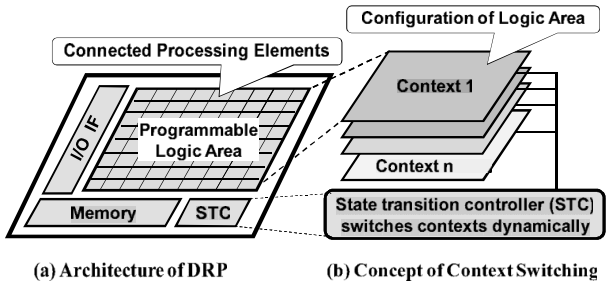


Figure 2: An architecture of DRP

4 System Decomposition into DRP

A system is decomposed into contexts of a DRP in order to implement a system on DRP. The decomposed system should also satisfy the timing constraints of the system.

In this section, we propose a heuristic algorithm to assign tasks in a system into contexts of a DRP so that all the tasks execute satisfying their timing constraints.

4.1 Problem Definition

Inputs:

A system $System = (Task, clock, N_t)$, and a DRP $DRP = (N_c, T_{switch})$.

Output:

N_c sets of tasks each of which is assigned into a context.

Constraints:

A set of valid path-combinations of the system is invariant whenever the system is assigned into the DRP and all the valid path-combinations are executable.

Objective:

To minimize the size of contexts of the DRP.

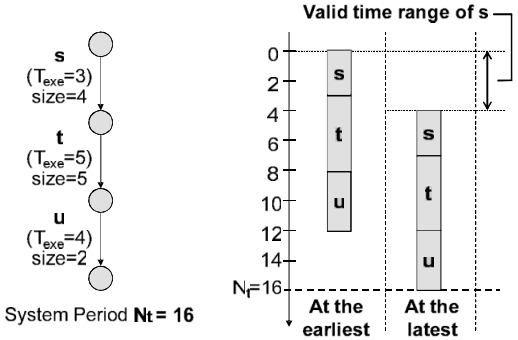
4.2 Proposed Heuristic Algorithm

The proposed algorithm consists of three parts: (1) to derive all tasks in valid path combinations, (2) to construct a task dependence graph about tasks' valid time range, and (3) to assign tasks into contexts according to the graph. For simplification of this problem, we assume that the context switching overhead T_{switch} is zero. In our future work, we will consider about the overhead.

4.2.1 Valid Time Range of Each Task

First, we derive all tasks in valid path combinations of a given system.

Given a task τ in a path p, a *valid time range* of τ in p, denoted $vrange(\tau, p)$, is defined as the range between the earliest startable time $t_{est}(\tau, p)$ of τ in p, and the latest startable time $t_{lst}(\tau, p)$ of τ in p. The earliest starting time $t_{est}(\tau, p)$ is calculated as the minimal value that makes $valid_{task}(\tau, p)$ true in p. Moreover, if τ is the last task of p, then $t_{lst}(\tau, p)$ is calculated as the maximal value that makes $valid_{task}(\tau, p)$ true in p. Otherwise, given the task τ' of p and its previous (parent) task τ of τ', $t_{lst}(\tau, p)$ is calculated as the maximal value of (a) the maximal value of $t_{start}(\tau)$ that makes $valid_{task}(\tau, p)$ true in p and (b) $(t_{lst}(\tau') - T_{exe}(\tau))$. An example of the valid time range of tasks in a path is shown in Fig. 3.



(a) A valid path **p1** (b) A valid time range

Figure 3: Valid time range

Given a path p and a task τ in p, the earliest startable time $t_{est}(\tau, p)$ and the latest startable time $t_{lst}(\tau, p)$ are calculated from $guard(\tau)$ as follows. Here, τ and τ' are tasks in a path p, k denotes a coefficient value, and D denotes a set of terms of the other tasks' variables and constant terms. If an inequality in $guard(\tau)$ is expressed as $t_{start}(\tau) < k \cdot t_{start}(\tau') + D$ then

$$t_{lst}(\tau, p) \leftarrow \min\left(t_{lst}(\tau, p), k \cdot t_{lst}(\tau', p) + D\right) \quad \text{and} \tag{1a}$$

$$t_{est}(\tau', p) \leftarrow \max\left(t_{est}(\tau', p), \frac{t_{lst}(\tau, p) - D}{k}\right), \tag{1b}$$

$t_{start}(\tau, p) < D - k \cdot t_{start}(\tau', p)$ then

$$t_{lst}(\tau, p) \leftarrow \min\left(t_{lst}(\tau, p), D - k \cdot t_{est}(\tau', p)\right) \quad \text{and} \tag{2a}$$

$$t_{lst}(\tau', p) \leftarrow \min\left(t_{lst}(\tau', p), \frac{D - t_{est}(\tau, p)}{k}\right), \tag{2b}$$

$t_{start}(\tau) > k \cdot t_{start}(\tau') + D$ then

$$t_{est}(\tau, p) \leftarrow \max\left(t_{est}(\tau, p), k \cdot t_{est}(\tau', p) + D\right) \quad \text{and} \tag{3a}$$

$$t_{lst}(\tau', p) \leftarrow \min\left(t_{lst}(\tau', p), \frac{t_{lst}(\tau, p) - D}{k}\right), \tag{3b}$$

or $t_{start}(\tau) > D - k \cdot t_{start}(\tau')$ then

$$t_{est}(\tau, p) \leftarrow \max\left(t_{est}(\tau, p), D - k \cdot t_{lst}(\tau', p)\right) \quad \text{and} \tag{4a}$$

$$t_{est}(\tau', p) \leftarrow \max\left(t_{est}(\tau', p), \frac{D - t_{lst}(\tau, p)}{k}\right). \tag{4b}$$

A task τ in a path p is valid if $t_{est}(\tau, p)$ and $t_{lst}(\tau, p)$ are defined (i.e., they have values) and $(t_{est}(\tau, p) \leq t_{lst}(\tau, p))$. Otherwise, task τ in path p is invalid, and p is also invalid.

Next, we consider about synchronizations among tasks. A task τ and a set of synchronized tasks $Task_{sync}(\tau)$ with τ should start their execution simultaneously. Thus, given τ and $Task_{sync}(\tau)$, $test(\tau, p)$ and $tlst(\tau, p)$ are recalculated as follows:

$$t_{est}(\tau', p') \leftarrow \max\big(t_{est}(\tau, p), \forall\tau''(t_{est}(\tau'', p''))\big), \quad \text{and} \tag{5a}$$

$$t_{lst}(\tau', p') \leftarrow \min\big(t_{lst}(\tau, p), \forall\tau''(t_{lst}(\tau'', p''))\big). \tag{5b}$$

Here, $\tau', \tau'' \in \{\tau\} \cup Task_{sync}(\tau)$, and p' and p'' denote paths which contain τ' and τ'', respectively. $t_{est}(\tau, p)$ and $t_{lst}(\tau, p)$ of all tasks in a path are updated as recalculated by Eqs. (1a)–(4b) whenever at least one of them in the path is recalculated. A valid path consists of valid tasks after the recalculation about synchronization.

Finally, we derive a *valid process*. Given a process pr, a task τ in pr and all the valid paths $p_1, \cdots p_n$ which contain τ in pr, $test(\tau, p_i)$ and $tlst(\tau, p_i)$ $(i \in \{1, \cdots, n\})$ are recalculated as follows:

$$t_{est}(\tau, p_i) \leftarrow \max_{j \in \{1, \cdots, n\}} \big(t_{est}(\tau, p_j)\big), \quad \text{and} \tag{6a}$$

$$t_{lst}(\tau, p_i) \leftarrow \min_{j \in \{1, \cdots, n\}} \big(t_{lst}(\tau, p_j)\big). \tag{6b}$$

Here, $j \in \{1, \cdots, n\}$. $t_{est}(\tau, p)$ and $t_{lst}(\tau, p)$ of all tasks in a path are updated too whenever at least one of them in the path is recalculated. A valid process consists of valid paths after the recalculation about .

A *valid system* consists of valid processes. In each period a valid system executes one of path combinations which consist of paths in each valid process one by one.

4.2.2 Construction of Task Dependency Graph about Valid Time Range

In a valid system, a task starts its execution within its valid time range. $st(\tau)$ denotes the time when an implemented task τ into a DRP starts its execution. We call $st(\tau)$ a scheduled time of τ. We can decide $st(\tau)$ within the valid time range of τ. We call this decision of $st(\tau)$ a *scheduling*. A task τ is running on a DRP between the time $st(\tau)$ and $st(\tau) + T_{exe}(\tau)$. We call this time range a *running time range*.

Given two tasks τ_1 and τ_2, they should be assigned into a same context of a DRP if the running time range of τ_1 and that of τ_2 overlap. On the other hand, if the running time ranges do not overlap, τ_1 can be assigned into different contexts from τ_2 so that the maximum size of contexts is reduced.

Given a task τ in a path p, the *earliest end time* and the *latest end time* of τ in p are calculated as $t_{est}(\tau, p) + T_{exe}(\tau)$ and $t_{lst}(\tau, p) + T_{exe}(\tau)$, respectively, as shown in Fig. 4. The running time range of τ depends on the scheduling of $st(\tau)$. However, the time range when τ is surely running can exist regardless of the scheduling of τ. We call such time range a *surely running range* as shown in the figure. If $t_{lst}(\tau) \leq t_{est} + T_{exe}(\tau)$, the surely running range of a task τ is given as the time range between $t_{lst}(\tau)$ and $t_{est}(\tau) + T_{exe}(\tau)$, otherwise the surely running range is empty. Note that $t_{est}(\tau)$ and $t_{lst}(\tau)$ denotes $t_{est}(\tau, p)$ and $t_{lst}(\tau, p)$ in any path p in a valid process, because $t_{est}(\tau, p)$ is same as $t_{est}(\tau, p')$ in a valid process regardless of paths p and p'. The runnable range of a task τ is given as the time range between $t_{est}(\tau)$ and $t_{lst}(\tau) + T_{exe}(\tau)$ as shown in the figure. Finally, given two

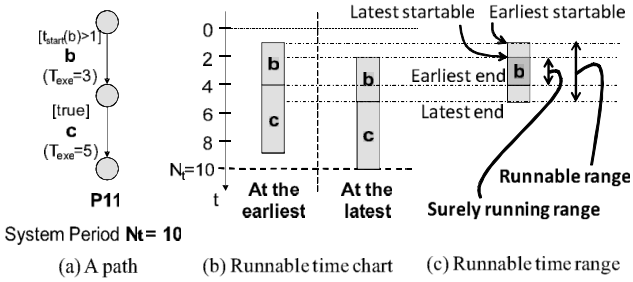


Figure 4: Runnable time range

task τ_1 and τ_2, they should be assigned into a same context if the surely running range of τ_1 and that of τ_2 overlap. Also, they can be assigned into different contexts if the runnable range of τ_1 and that of τ_2 do not overlap.

In the proposed algorithm, we construct a graph where a node represents a task in a valid system and an edge between two tasks represents a relation whether they should/can be assigned into a same context. We call this graph a *task dependency graph*. We give a single edge between any two tasks the runnable range of which overlaps each other. We also give a double edge between any two tasks the surely running range of which overlaps each other. However, we do not give any edge between two tasks if at least one of the following conditions satisfies:

> The tasks are in the same process and are not in the same path, and
>
> All the paths which contain one of the task are not synchronized with all the paths which contain the other task.

For example, Fig. 5 shows the runnable time range and the task dependency graph of the system in Fig. 1.

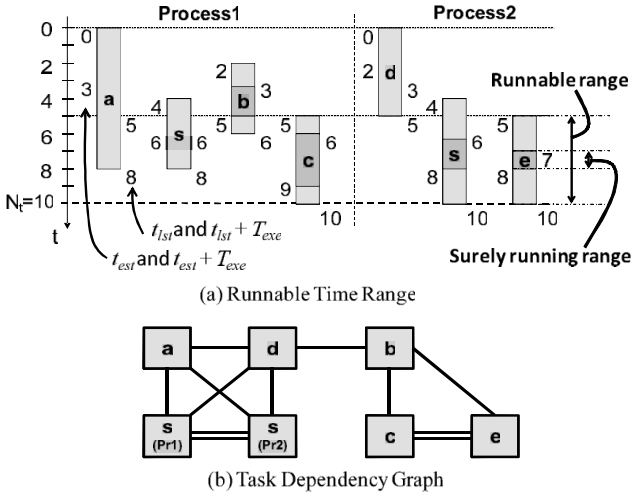


Figure 5: Example of `System1`

4.2.3 Assignment Tasks into Contexts using Task Dependency Graph

In this phase, we decompose the task dependency graph of a given system into several smaller graphs of which the maximum size is as small as possible. If the number of derived graphs is larger than the number of contexts N_c, the graphs are merged into N_c graphs. Finally, each the graph corresponds to each context of a DRP.

Tasks connected by double edges in a task dependency graph should be assigned into a same context of a DRP, because they have run all together for at least one clock. Thus, we decompose a task dependency graph keeping the size of the maximum connected graph with double edges small.

In a task dependency graph, a single edge can be changed to a double edge or no edge due to the scheduling of the edge's both end tasks. That is calculated by updating the runnable time range of each tasks. Figures 6 and 7 show how to decompose a connected tasks.

Figure 6 shows an example of decomposing tasks connected by double edges. By assign Task `b` into two contexts redundantly, the size of the maximum connected tasks by double edges can be reduced although the total size of implemented tasks into contexts becomes larger. However, tasks in a clique shaped by double edges cannot be decomposed.

Figure 7 shows an example of decomposing tasks connected by single edges. Tasks connected by single edges can be assigned into a same contexts. It is also important to decompose tasks connected by single edges and make the size of the maximum connected tasks by single edges smaller. However, as shown in Fig. 7, after the decomposition, the single edge between Task `a` and `b` has been changed to a double edge. Thus, it needs to be carefully chosen how a connected task is decomposed.

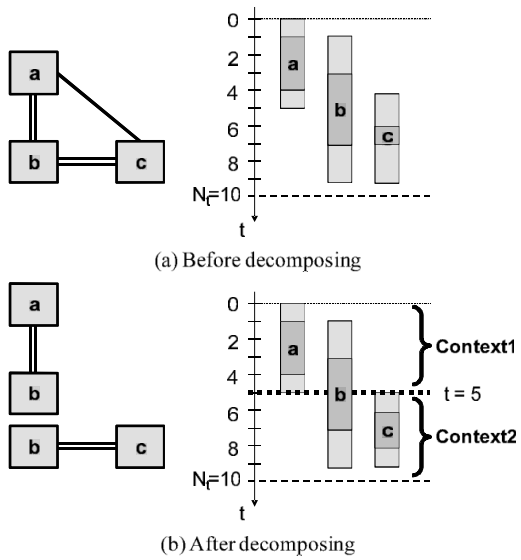


Figure 6: Example of Decomposing Tasks connected by Double Edges

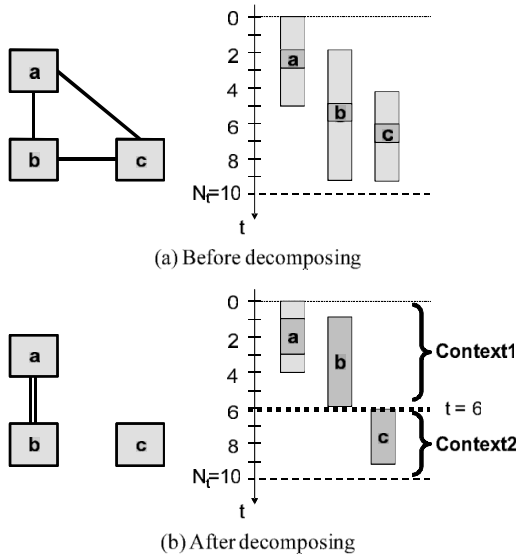


Figure 7: Example of Decomposing Tasks connected by Single Edges

Finally, after an assignment into contexts, the size of the maximum context is at least $\frac{S}{N_c}$. Here, S denotes the sum of the tasks in a valid system, calculated as $S = \sum_\tau size(\tau)$ and note that N_c is the number of contexts of a DRP. Therefore, in our algorithm, we decompose connected tasks in the following order of priority:

1. the maximum connected tasks by only double edges and the size of them is larger than $\frac{S}{N_c}$,
2. the maximum connected tasks by double edges and single edges and the size of them is larger than $\frac{S}{N_c}$,
3. the maximum connected tasks by only double edges, or
4. the maximum connected tasks by double edges and single edges.

A connected tasks is decomposed as follows.

Step 1. C denotes a set of contexts. C is initialized to the set of all tasks.

Step 2. Select one context c from C by the priority as mentioned above. $C \leftarrow C/\{c\}$.

Step 3. Sort the tasks in c by t_{est}.

Step 4. Find a clock t when the tasks are divided into two groups G_1 and G_2. G_1 consists of tasks with $t_{est} \leq t$ and G_2 consists of tasks with $t_{est} > t$ so that the total size of each group is (almost) the same.

Step 5. Find the cut set between G_1 and G_2.

Step 6. Given a double edge in the cut set and a task in G_1 is an end of the edge, the task is assigned to two new contexts c_1 and c_2. Other tasks in G_1 are assigned to c_1 and the tasks in G_2 are assigned into c_2.

Step 7. $C \leftarrow C \cup \{c_1, c_2\}$.

Step 8. Update the runnable time range of all the task.

Step 9. If all connected tasks in C are clique then end, else goto Step 2.

We call a length between t_{est} and t_{lst} a *deadline slack*. When a graph is decomposed to two subgraphs, it is important to allocate the deadline slack to the subgraphs. In this algorithm, we give half of the deadline slack to each of the subgraphs. Later, we evaluate how the deadline slack is allocated into two subgraphs.

After the decomposition, we have derived the sets of tasks corresponding assignment into contexts. However, the number of the sets C can be larger than N_c. Thus, we merge the sets into N_c sets. In the proposed method, the smallest set of C and the second-smallest set are merged until the number of sets becomes N_c.

5 Evaluation

We have compared the performance of the proposed heuristic algorithm. We implemented the algorithm in Java 5.0 on a Core2 Duo 1.2GHz CPU, 2GB RAM with Microsoft Windows XP Professional. We consider systems with m concurrent processes, each of which consists of n paths. For each $< m, n >$, we randomly derive 20 systems. In the following experiments, we have considered systems with the time period $N_t = 30$ and DRPs with four contexts ($N_c = 4$).

First, we evaluated how the deadline slack is divided into two subgraphs at a ratio of. Table 1 shows the summary of the obtained results for cases $5:5$, $6:4$, $7:3$, and $8:2$. This results shows that it is the best for a deadline slack to be divided into two subgraphs at a ratio of $5:5$.

Next, we have compared the performance of the proposed heuristic algorithm with a general ILP solver w.r.t. the execution time necessary for obtaining a task assignment and w.r.t. the quality of the obtained solutions. We used the ILP solver GLPK[20] in our experiments. For each derived system, we have derived its corresponding linear constraints, applied the proposed algorithm and the ILP solver, and found assignments. Then we calculate the average execution time of the algorithm and the ILP solver.Table 2 shows the summary of the obtained results for cases $< 2, 2 >$ and $< 2, 4 >$.

Table 1: Maximum Context Size against Ratio of Deadline Slack

$< m, n >$	$< 2, 2 >$	$< 4, 4 >$	$< 8, 8 >$
$5 : 5$	43.9	84.8	128.7
$6 : 4$	45.6	85.4	131.6
$7 : 3$	47.1	100.6	130.8
$8 : 2$	62.3	112.0	180.3

Table 2: Two Solver Results and Execution Time

$< m, n >$	$< 2, 2 >$		$< 2, 4 >$	
	size	time	size	time
ILP	26.7	0.16 sec.	35.0	> 10 h.
Proposed	27.7	78 msec.	39.0	125 msec.

The results clearly show that the proposed algorithm outperforms the ILP in respect to execution time, and the difference becomes remarkable as the size of the system becomes bigger. For small systems $< 2, 2 >$, our method assigned the systems with the average logical area size 27.7 within 78ms and the ILP solver assigned it with the size 26.7 within 0.16sec. For larger systems $< 2, 4 >$, our method assigned the systems with the average logical area size 39.0 within 125ms and the ILP solver assigned it with the size 35.0 within 10hours.

6 Conclusion

In this paper, we have focused on a problem implementing a periodic concurrent system with real-time constraints into a multi-context DRP. We have proposed a heuristic algorithm to solve the problem efficiently. In the experimental results, we have confirmed that the proposed heuristic algorithm derives quasi-optical assignment results for large size examples in short time.

Considering the context switching overhead to our proposed algorithm and applying it to several types of real systems are our future work.

REFERENCES

[1] B.P. Dave, and N.K. Jha, "CASPER: Concurrent Hardware-Software Co-Synthesis of Hard Real-Time Aperiodic and Periodic Specifications of Embedded System Architectures," *Proc. of DATE '98*, pp.118–125 (1998).

[2] T. Kitani, Y. Takamoto, K. Yasumoto, A. Nakata, and T. Higashino, "A Flexible and High-Reliable HW/SW Co-Design Method for Real-Time Embedded Systems," *Proc. of RTSS2004*, pp. 437–446 (2004).

[3] K. Ramamritham, "Allocation and Scheduling of Complex Periodic Tasks," *Proc. ICDCS '90*, pp. 108–115 (1990).

[4] D.-T. Peng, and K.G. Shin, "Assignment and Scheduling of Communication Periodic Tasks in Distributed Real-Time Systems," *Proc. ICDCS '89*, pp. 190–198 (1989).

[5] C.-J. Hou, and K.G. Shin, "Allocation of Periodic Task Modules with Precedence and Deadline Constraints in Distributed Real-Time Systems," *IEEE Trans. Comput.*, vol. 46, no. 12, pp. 1338–1356 (1997).

[6] R. Maestre, R. Hermida, F.J. Kurdahi, N. Bagherzadeh, and H. Singh, "Optimal vs. Heuristic Approaches to Context Scheduling for Multi-Context Reconfigurable Architectures," *Proc. ICCD'00*, pp. 575–576 (2000).

[7] K.M.G. Purna, and D. Bhatia, "Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers," *IEEE Trans. Comput.*, vol. 48, no. 6, pp. 579–590 (1999).

[8] B. Ouni, A. Mtibaa, and M. Abid, "Synthesis and Time Partitioning for Reconfigurable Systems," *Design Automation for Embedded Systems*, vol. 9, no.3, pp. 177–191 (2004).

[9] M. Kaul, and R. Vemuri, "Optimal Temporal Partitioning and Synthesis for Reconfigurable Architecture," *Proc. of DATE '98*, pp. 389–396 (1998).

[10] M. Sanchez-Elez, M. Fernandez, R. Maestre, R. Hermida, and N. Bagherzadeh, and F.J. Kurdahi, "A Complete Data Scheduler for Multi-Context Reconfigurable Architectures," *Proc. of DATE 02*, pp. 547–552 (2002).

[11] S. Banarjee, E. Bozorgzadeh, and N. Dutt, "Physically-Aware HW-SW Partitioning for Reconfigurable Architectures with Partial Dynamic Reconfiguration," *Proc. DAC '05*, pp. 335–340 (2005).

[12] S. Banarjee, E. Bozorgzadeh, and N. Dutt, "Considering run-time reconfiguration overhead in Task Graph Transformations for dynamically reconfigurable architectures," *Proc. FCCM '05* pp. 273–274 (2005).

[13] F. Redaelli, M.D. Santambrogio, and D. Sciuto, "Task Scheduling with Configuration Prefetching and Anti-Fragmentation techniques on Dynamically Reconfigurable Systems," *Proc. DATE '08*, pp. 519–522 (2008).

[14] ISO, "Information Processing System, Open Systems Interconnection LOTOS," *ISO 8807* (1989).

[15] H. Nakano, T. Shindo, T. Kazami, and M. Motomura, "Development of Dynamically Reconfigurable Processor LSI", *NEC Technical Journal*, Vol. 56, No. 4, pp. 99–102 (2003). http://www.necel.com/.

[16] T. Sunagawa, K. Ide, and T. Sato, "Dynamically Reconfigurable Processor Implemented with IPFlex's DAPDNA Technology", *IEICE Trans. Info. and Sys.*, Vol. E87-D, No. 8, pp. 1997-2003 (2004).

[17] H. Singh, M.-H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481 (2000).

[18] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Trans. Comput.*, vol. 33, no. 4, pp. 70–77 (2000).

[19] B. Mei, A. Lambrechts, D. Verkest, J.-Y. Mignolet, and R. Lauwereins, "Architecture Exploration for a Reconfigurable Architecture Template," *IEEE Trans. Design & Test of Comp.*, vol. 22. Issue 2, pp. 90–101 (2005).

[20] GNU Project, "GLPK (GNU Linear Programming Kit)", http://www.gnu.org/software/glpk/.