平成 30 年度

修士学位論文

データ駆動型マルチプロセッサにおける分散スケジューリング機構の研究

A Study on Decentralized Hardware Scheduler for Data-Driven Multiprocessor

1215095　　福田 和馬

指導教員　　岩田 誠

2019 年 2 月 28 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報学コース

要 旨

データ駆動型マルチプロセッサにおける
分散スケジューリング機構の研究

福田 和馬

近年の IoT (Internet of Things) デバイスは，様々な分野で活用が広がり，ますます，高機能化，高性能化が求められている．特に，ミッションクリティカルシステムでは，マルチコア上でのリアルタイム処理が必要となる．

マルチコア上でのリアルタイム処理には，パーティショニング方式とグローバル方式がある．前者はタスク実行前に静的にタスク割り当てを決定するため，実行中の動的な負荷変動には対応できない．また，静的なタスク割り当ては NP 困難な問題である．後者は，コアやタスクの実行状態に応じて動的にタスク割り当てを決定するため，スケジューリングオーバヘッドをゼロと考えると，高い稼働率を達成できる．しかし，実際にはスケジューリングオーバヘッドの大小で稼働率が左右される．また，コア数が増加するとオーバヘッドも増加し，スケーラビリティが低下するため，マルチコア化による性能向上には分散スケジューリングが必須となる．

本研究では，複数タスクを多重に処理可能なデータ駆動型プロセッサ DDP に着目し，相互結合網内のスイッチ用スケジューラと DDP コア内のスケジューラを統合した，分散スケジューリング機構を検討する．また，オーバヘッドを考慮した場合は，コア数が多くなると，提案方式がグローバル方式の EDF (Earliest Deadline First)，LST (Least Slack Time) に比べてスケジューリング性能が高いことを回路設計およびシミュレーションにより確認した．

キーワード　マルチコア，リアルタイムスケジューリング，セルフタイム型パイプライン，
データ駆動型プロセッサ

Abstract

A Study on Decentralized Hardware Scheduler for Data-Driven Multiprocessor

Kazuma FUKUDA

Recent internet of things (IoT) devices are used in various fields, and they are required to operate with higher functionality and performance. Especially, in mission critical systems, it is necessary to perform real-time processing on multicore system.

For real-time processing on multicore system, there are partitioning scheduling and global scheduling. The former decides the task assignment statically before executing the task. Therefore, it cannot cope with dynamic load fluctuations during execution. The latter decides the task assignment according to the execution state of the core and the task. Therefore, it can achieve higher utilization. Actually, the utilization depends on the scheduling overhead. In addition, as the number of cores increases, the overhead increases and the scalability decreases. Therefore, decentralized hardware scheduler is indispensable for performance improvement of multicore.

In this research, we focus on data-driven processor (DDP) which is possible to execute multiple tasks in parallel without task switching overhead. In this paper, a decentralized hardware scheduler is proposed, in which the switch scheduler in the interconnection network and the scheduler in the DDP are integrated. The result indicated that it had higher performance than the global scheduling under actual overhead.

key words multicore, real-time scheduling, self-timed pipeline, data-driven processor

目次

図目次

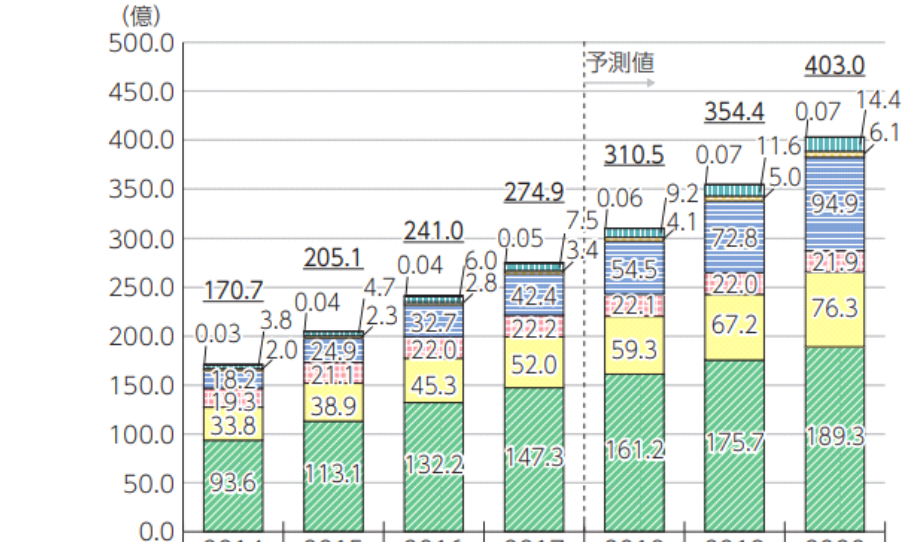
図目次

表目次

第 1 章

序論

近年の IoT (Internet of Things) デバイスは，図 1.1 に示すように，年々増加しており，2020 年には 403 億個にまで増加すると予測されている [1].



	2014	2015	2016	2017	2018	2019	2020
合計	170.7	205.1	241.0	274.9	310.5	354.4	403.0
軍事・宇宙・航空	0.03	0.04	0.04	0.05	0.06	0.07	0.07
自動車	3.8	4.7	6.0	7.5	9.2	11.6	14.4
医療	2.0	2.3	2.8	3.4	4.1	5.0	6.1
産業用途	18.2	24.9	32.7	42.4	54.5	72.8	94.9
コンピューター	19.3	21.1	22.0	22.2	22.1	22.0	21.9
コンシューマー	33.8	38.9	45.3	52.0	59.3	67.2	76.3
通信	93.6	113.1	132.2	147.3	161.2	175.7	189.3

図 1.1　世界の IoT デバイス数の推移及び予測 (文献 [1] より引用)

IoT デバイスは，産業，医療，自動車など，様々な分野のアプリケーションで使用され，高機能化，高性能化が進んでいる．これまでは，シングルコアの動作周波数を高めることで性能向上が図られてきたが，消費電力や発熱の面からシングルコアでの性能向上には限界が

ある．このような問題を解決するために，マルチコア化が進んでいる．加えて，自動車やロボットなどのミッションクリティカルシステムでは，所要の時間内に特定のエラーを回復できなかった場合，重大な事故につながる危険性があるため，リアルタイム性が要求される．よって，将来の IoT デバイスでは，リアルタイム処理機能を備えたマルチプロセッサが必要となる [2].

リアルタイム処理を実現するためには，アプリケーションに含まれるすべてのタスクの時間制約を保証するために，各タスクの優先度を考慮したリアルタイムスケジューリングが必要となる．本来のタスク実行の有効稼働率を低下させないためには，スケジューリング時に発生する様々なオーバヘッドを削減する必要がある．リアルタイムスケジューリングに対するソフトウェアによるアプローチとしては，リアルタイムオペレーティングシステム (RTOS) を用いることが一般的である．RTOS は，スケジューリングやリソース管理など，リアルタイムシステムに必要な機能を提供しており，RTOS を用いることでシステム開発の容易化，効率化を図ることができる．しかし，ソフトウェアによるアプローチは，計算上のオーバヘッドが大きく，単純なスケジューリングアルゴリズムを使用するため，平均稼働率が低下する問題がある．そこで，ハードウェアスケジューラに関する研究が各所で行われている [3][4][5]．スケジューリングを専用ハードウェアで行うことで，ソフトウェアのオーバヘッドを削減し，細粒度なスケジューリングが可能となる．

リアルタイムスケジューリングのアルゴリズムには，静的優先度方式と動的優先度方式がある．静的優先度方式では，各タスクの優先度はタスク実行前にあらかじめ割り当てられ，実行中に優先度が変化することはない．一方，動的優先度方式では，各タスクの優先度はタスク実行中に動的に変化する．一般的に，IoT デバイスにおけるリアルタイムタスクは，周期的，または非周期的に実行される．静的優先度方式では，タスクの優先度は実行時に更新されないため，非周期タスクを処理するためには動的優先度方式を採用する必要がある．しかし，動的優先度方式は，静的優先度方式に比べ，タスク実行中にタスクの切り替え (プリエンプション) が頻繁に発生する．一般的な組み込みマイクロプロセッサは，ノイマン型アーキテクチャ，すなわち，制御駆動方式を基本としており，プリエンプション時にコンテ

キストスイッチが発生するため，タスク切り替え時のオーバヘッドが大きい．そこで，コンテキストスイッチなしに，複数のタスクを多重に処理可能なデータ駆動型プロセッサ (以下，DDP：Data-Driven Processor) [6] に着目したハードウェアスケジューラ [7] [8] が提案されている．

また，マルチコアシステムでは，相互結合網と呼ばれるネットワークで複数のコアを接続し，コア間での通信を実現する．そのため，マルチコア上でリアルタイム処理を実現するためには，スケジューリングアルゴリズムに従って，タスクをひとつ，または複数のコアに割り当てる必要がある．タスクをコアに割り当てる方式として，パーティショニングスケジューリング方式とグローバルスケジューリング方式がある [9]．パーティショニングスケジューリング方式は，タスク実行前にあらかじめ静的に各コアにタスクを割り当て，各コアが独立してスケジューリングアルゴリズムを実行する方式である．対して，グローバルスケジューリング方式は，1 つのキューにタスクを入れ，全コアでタスクを共有し，高優先度タスクから順に各コアに割り当てる方式である．パーティショニングスケジューリング方式は，静的にコアにタスクを割り当てた後は，各コアが独立してスケジューリングを実行するため，実装が容易である．加えて，コア間でのタスクの移動 (マイグレーション) が発生しないため，オーバヘッドが小さいという利点がある．しかし，静的なタスク割り当ては，タスクを荷物，コアを箱としたビンパッキング問題として見ることができ，NP 困難な問題である [10]．また，タスク実行時には，分岐の有無やキャッシュのヒット/ミスヒットなどによる実行時間の変動や，非周期タスクの要求があるが，パーティショニングスケジューリング方式はこのような動的な負荷変動に対応できないため，実用的には使えない．一方，グローバルスケジューリング方式は，すべてのコアでタスクを共有しているため，コアやタスクの実行状態に応じて動的にタスク割り当てを決定することができる．そのため，スケジューリングオーバヘッドをゼロと考えると，パーティショニングスケジューリング方式に比べ，高い稼働率を達成することができる．しかし，実際にはスケジューリングオーバヘッドの大小で稼働率は左右される．グローバルスケジューリング方式では，コア間でタスクの移動が発生するため，マイグレーションのオーバヘッドが大きい．また，コアやタスクの実行状態を収

集，分析し，各コアにフィードバックする必要があるため，スケジューラのオーバヘッドが大きい．コア数が増加すると，実行状態を管理するための，ネットワーク遅延やデータ量が大きくなるため，オーバヘッドも増加する．そのため，スケーラビリティの観点からは，集中型よりも分散型で実行状態を管理するほうが望ましいと考えられる．

本研究では，DDP をマルチコア化した，データ駆動型マルチプロセッサに着目し，ネットワーク内のスイッチ用スケジューラと DDP コア内のスケジューラを統合した，分散スケジューリング機構 [11] を検討する．DDP は割り込みを必要とせず，タスクの到着をトリガにタスクの入出力が可能であり，各 DDP コアは独立して動作できる．そのため，DDP は分散スケジューリングと親和性が高いと考えられる．

本稿において，第 2 章ではマルチコアにおけるリアルタイムスケジューリング方式とその問題点を議論する．この議論を踏まえて，分散スケジューリング機構に必要な，ネットワークとコアの要件を述べる．

第 3 章では，第 2 章で述べた要件から，分散スケジューリング機構を構成する，ネットワーク内のスイッチ用スケジューラのアルゴリズムと，DDP コアのアーキテクチャについて述べる．

第 4 章では，アーキテクチャシミュレータを用いて，パーティショニングスケジューリング方式，グローバルスケジューリング方式と提案方式のスケジューリング性能評価を行い，結果を考察する．ネットワーク内のスイッチと DDP コアを 65nm CMOS 標準セルライブラリを用いて論理合成ツール，Synopsys 社 Design Compiler で論理合成を行い，性能パラメータを抽出後，シミュレーションベースで評価を行った．

第 5 章では，本研究のまとめと今後の課題を述べる．

第 2 章

マルチコアにおけるリアルタイムスケジューリング

2.1 緒言

本章では，リアルタイムシステムにおいてスケジューリングされるタスクが持つパラメータと分類をまとめ，動的優先度方式の代表的なアルゴリズムである EDF (Earliest Deadline First) と LST (Least Slack Time) の 2 つをマルチコアの観点で比較する．その後，マルチコアにおけるタスク割り当て方式であるパーティショニングスケジューリング方式，および，グローバルスケジューリング方式の概要と問題点について述べる．そして，その問題点を踏まえて，分散スケジューリング機構を構成するネットワークとコアの要件と設計方針を述べる．

2.2 タスクの定義

タスクは主に，図 2.1 に示すパラメータを持つ．タスクは，実行要求時刻に起動すると，実行可能状態になる．そして，スケジューリングされ，コアに割り当てられると実行される．各タスクは，自タスクの持つ絶対デッドライン時刻までに実行を完了する必要がある．絶対デッドライン時刻は，実行要求時刻と相対デッドライン時間の和で求められる．絶対デッドライン時刻を超えても実行が完了していないタスクはデッドラインミスとなる．また，絶対デッドライン時刻を超えることなく，タスク実行の開始を遅らせることができる最大遅延時

2.2 タスクの定義

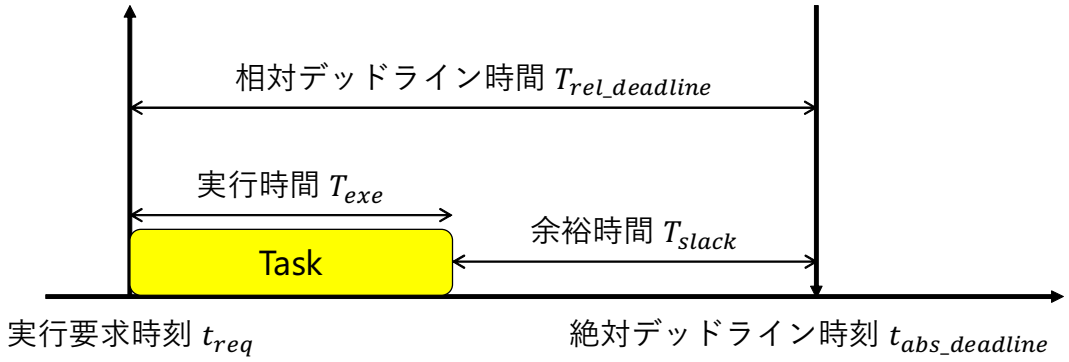


図 2.1　タスクのパラメータ

間を余裕時間と呼ぶ．各タスクは，最大で余裕時間がゼロになるまでタスク実行の開始を遅らせることができる．実行要求時刻における初期余裕時間は，

$$T_{slack} = (t_{abs\_deadline} - t_{req}) - T_{exe} \tag{2.1}$$

ある時刻 t における余裕時間は，

$$T_{slack} = (t_{abs\_deadline} - t) - T_{remain\_exe} \tag{2.2}$$

で求められる．T_{remain_exe} は，時刻 t における残り実行時間である．

タスクは，実行要求時刻のタイミングによって，周期タスクと非周期タスクに分類される．

- 周期タスク

  実行要求時刻にタスクが起動すると，実行要求時刻から絶対デッドライン時刻までを 1 周期として，周期的に起動する．

- 非周期タスク

  実行要求時刻が決まっておらず，非周期的に起動する．

また，時間制約の厳密性から，ハードリアルタイムタスクとソフトリアルタイムタスクに分類される [12]．

- ハードリアルタイムタスク

  デッドラインミスを起こした場合，システムに致命的な影響を与えるタスクである．車載の制御システムなどが該当する．

- ソフトリアルタイムタスク

  デッドラインミスを起こした場合，システムの性能が低下する可能性はあるが，致命的な問題にはならないタスクである．予約システムなどが該当する．

リアルタイム処理を実現するためには，各タスクがデッドラインミスを起こさないように，タスクの実行順序をスケジューリングする必要がある．リアルタイム処理性能 (スケジューリング性能) に関する指標には，主に以下がある．

- システム稼働率

  システムに含まれる各コアの稼働率の平均である．
- スケジューリング成功率

  要求されたタスクセット/タスクのうち，デッドラインミスを起こすことなく実行を完了したタスクセット/タスクの割合である．
- 実行時間

  実行要求時刻から，実行完了までに実際にかかる時間である．
- タスク実行時間のジッタ

  最小実行時間と最大実行時間の差である．

2.3　動的優先度方式アルゴリズムの比較

タスクの実行順序を決定するための様々なアルゴリズムが研究されているが，動的優先度方式の代表的なアルゴリズムに，EDF (Earliest Deadline First) [13] と LST (Least Slack Time) [14] がある．EDF では，すべての要求タスクのうち，絶対デッドライン時刻 $t_{abs\_deadline}$ の近いタスクから順に高い優先度が割り当てられる．一方，LST では，余裕時間 T_{slack} の短いタスクから順に高い優先度が割り当てられる．

マルチコアにおいて EDF と LST を比較した場合，EDF は優先度に実行時間を考慮していないため，タスクのデッドラインミスを予測することができない．対して，LST は，余裕時間がゼロ未満であるか否かによって，デッドラインミスを予測することができる．例とし

表 2.1　タスクの例

	実行要求時刻	実行時間	相対デッドライン時間	初期余裕時間
Task A	0	6	10	4
Task B	1	2	8	6
Task C	2	5	6	1

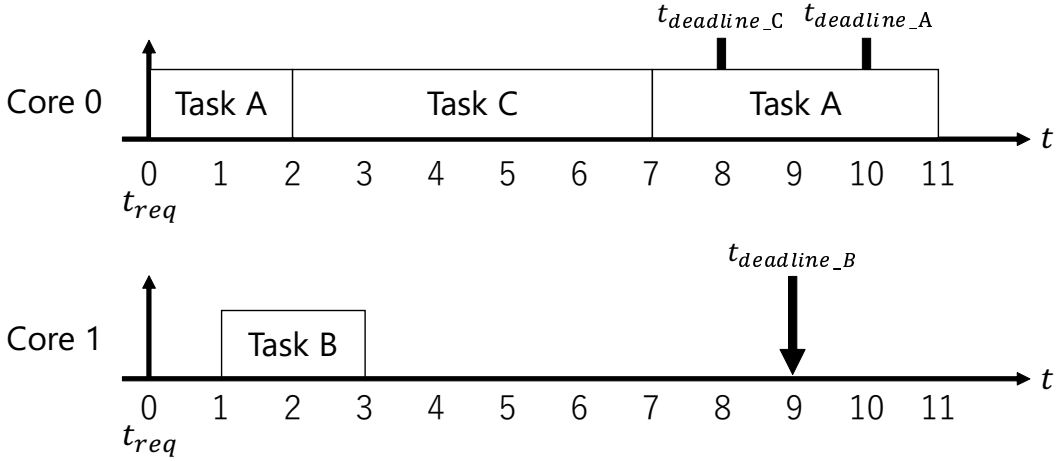


図 2.2　EDF によるスケジューリング結果

て，表 2.1 に示す 3 タスクを 2 コアで，EDF と LST によりスケジューリングした場合，図 2.2，図 2.3 のようになる．ここでは，マイグレーションは行わないと仮定する．

図 2.2 では，時刻 10 で Task A がデッドラインミスを起こしている．それに対して，図 2.3 では，すべてのタスクがデッドラインミスを起こさずに実行を完了している．例のように LST は EDF に比べ，デッドラインミスを起こさずにスケジューリングできるが，複数のタスクが同じ余裕時間を持つ場合，スラッシングと呼ばれる頻繁なタスク切り替えが発生し，オーバヘッドが増加する問題がある．そのため，LST を用いる場合は，スラッシングによるオーバヘッドを少なくする必要がある．あるいは，スラッシングの少ない改良 LST[15][16][17] の採用が必要になる．

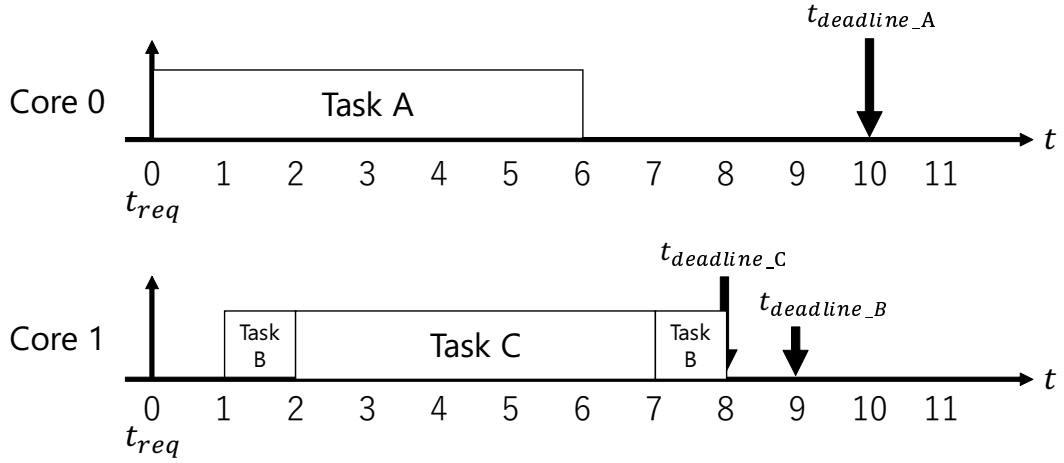


図 2.3　LST によるスケジューリング結果

2.4　マルチコアにおけるタスク割り当て方式の比較

マルチコアでは，スケジューリングアルゴリズムに基づいて，要求されたタスクをコアに割り当てる必要がある．マルチコアにおいて，タスクをコアに割り当てる方式として，パーティショニングスケジューリング方式とグローバルスケジューリング方式がある．

2.4.1　パーティショニングスケジューリング方式

パーティショニングスケジューリング方式の構成を図 2.4 に示す．パーティショニングスケジューリング方式では，各コアがそれぞれローカルキューとスケジューラを持つ．すべてのタスクは実行前にあらかじめ静的に各コアにタスクを割り当て，各コアのローカルキューで保持される．静的なタスク割り当ては，ビンパッキング問題に帰着するため，NP 困難な問題である．そのため，First-Fit，Next-Fit，Best-Fit，Worst-Fit などの近似アルゴリズム [18][19] が用いられる．タスク割り当て後，各コアはローカルキュー内のタスクを，スケジューリングアルゴリズムに基づいて実行する．

厳密に静的にタスクを割り当てるには，各タスクの最悪実行時間を正確に見積もる必要がある．しかし，実際には，分岐の有無やキャッシュのヒット/ミスヒットの影響による実行時間の変動がある．また，非周期タスクの到着時刻を予測することは困難であるため，非周期

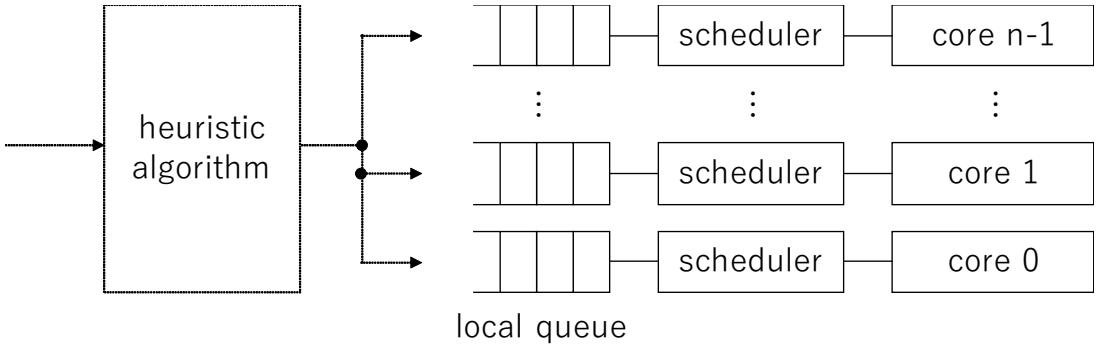


図 2.4　パーティショニングスケジューリング方式の構成

タスクに対応することができない．パーティショニング方式は静的な方式であり，このような実行時の動的な負荷変動には対応できないため，実用的には使えない．

2.4.2　グローバルスケジューリング方式

グローバルスケジューリング方式の構成を図 2.5 に示す．グローバルスケジューリング方式は，すべてのコアでひとつのグローバルキューとスケジューラを持ち，タスクを全コアで共有することができる．マイグレーションが可能であり，スケジューラはコアやタスクの実行状態に応じて動的にタスク割り当てを決定することができる．そのため，オーバヘッドをゼロと考えると高い稼働率を達成することができる．しかし，実際にはオーバヘッドの大小で稼働率は左右される．マイグレーション時には，タスク実行を別のコアに移動する必要があるため，オーバヘッドが生じる．また，スケジューラはタスクの切り替えを判断するために，コアやタスクの実行状態を集中的に収集，分析し，各コアにフィードバックする必要があるため，オーバヘッドが大きい．コア数が増加すると，実行状態を管理するためのネットワーク遅延やデータ量も増加するため，オーバヘッドがさらに大きくなり，スケーラビリティが低下する可能性がある．今後，性能向上の要請が高まり，コア数が増加することを考慮すると，スケーラビリティの観点から，スケジューラは集中型ではなく，分散型に動作できる方式が望ましいと考えられる．

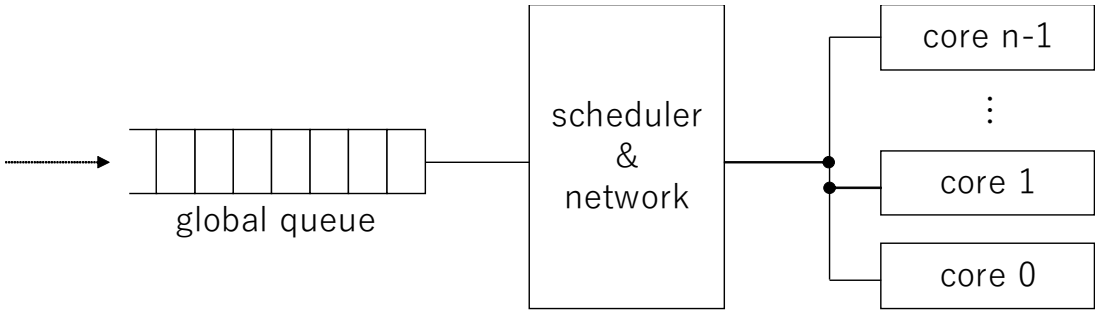


図 2.5　グローバルスケジューリング方式の構成

2.5　分散スケジューリング機構の要件

前節で述べたように，タスク実行中の動的な負荷変動に対応するためには，コアやタスクの実行状態を収集し，分析する必要がある．しかし，マルチコアのスケーラビリティを犠牲にすることなく，単一の集中型スケジューラですべての情報を管理することは困難である．そこで，本研究では，マルチコアシステムを構成するネットワークとコアが自律分散的に動作することで，分散的にスケジューリングが可能な，マルチコア向きの分散スケジューリング機構を検討することを目的とした．

スケーラビリティを犠牲にすることなく，スケジューリングの性能を低下させないためには，スケジューリング時に発生するオーバヘッドを削減する，あるいは，オーバヘッドの影響を受けない方法を検討する必要がある．分散スケジューリングにおける主要なオーバヘッドには，マイグレーション，プリエンプション，スケジューラによるものがある．タスクのマイグレーションのためには，すべての候補コアの状態を収集する必要がある．これは，分散スケジューリングに反するため，本研究では，マイグレーションなしを前提とする．そのため，各コアがローカルキューを持ち，ネットワーク内のスケジューラによって，タスクがコアに割り当てられた後は，コア内のスケジューラに従って，タスク実行が行われるものとする．プリエンプションとスケジューラのオーバヘッドを考慮した，ネットワークとコアの設計方針を以下に示す．

2.5.1　ネットワークの設計方針

マルチコアにおけるコア間の接続形態には，主に，バス，クロスバー，NoC (Network-on-Chip) がある [20]．バスは，実装が容易であるが，コア数が増加すると通信のボトルネックとなる．クロスバーは，すべての入力・出力の組合せにおいて距離 1 で通信が可能であるため，低レイテンシ，高スループットであるが，コア数が増加するとハードウェアコストが大きくなる．一方，NoC は，コア間が複数のスイッチ (ルータ) を介してネットワーク状に接続されており，バスやクロスバーに比べて，スケーラビリティに優れている [21][22]．本研究では，スケーラビリティの観点から，コア間は NoC のように複数のスイッチを介して接続されているものとする．したがって，分散スケジューリング機構は，ネットワーク内の各スイッチがスケジューラの役割を持ち，各スイッチが自律的に動作し，宛先を決定することで実現する．また，各スイッチがローカルに持つ情報から宛先を決定することで，スケジューラ全体のオーバヘッドを小さくする．コアに直接接続されているスイッチは，過去のタスク割り当てからコアの状態を判断することができる．しかし，コアに直接接続されないスイッチはコアの状態を精密に予測できないため，過去の通過タスクの情報から経験的に宛先を決定する方針を採る．また，デッドラインミスが起きないように，各コアの稼働率が均衡化するようにタスクを割り当てることが望ましい．

2.5.2　コアの設計方針

本研究では，コアとなるプロセッサ上で，動的優先度方式のスケジューリングを実行するものとする．よって，コアでは，タスク実行中に優先度の高いタスク実行が要求されると，プリエンプションが発生し，低優先度タスクの実行を一時的に中断し，高優先度タスクを優先して実行する必要がある．コアのスケジューリング性能を向上させるためには，タスク切り替え時に発生する，プリンプションのオーバヘッドを最小限に抑える必要がある．そのため，コアにはコンテキストスイッチなしに，複数のタスクを多重に処理可能な DDP を用いる．DDP は要求タスクの処理負荷が，DDP が多重処理可能な負荷を超えない限り，複数

タスクを多重に処理可能である．多重処理可能な負荷を超えると，DDP 内のスケジューラを動作させる必要がある．DDP 内のスケジューラのアルゴリズムには，2.3 節の議論から LST を採用する．LST は余裕時間が同じタスクが複数存在するとスラッシングと呼ばれる頻繁なタスク切り替えが発生するため，プリエンプションのオーバヘッドなしに複数タスクを処理できることは重要である．LST を採用した DDP は文献 [8] で提案されている．また，DDP は到着タスクをトリガとして，自律的に動作できるため，分散スケジューリングの観点からも有効であると考えられる．

2.6　結言

本章では，リアルタイムシステムにおけるタスクのパラメータと分類をまとめ，動的優先度方式のスケジューリングアルゴリズムである EDF と LST をマルチコアの観点から比較した．また，マルチコアのタスク割り当て方式であるパーティショニングスケジューリング方式，グローバルスケジューリング方式を比較した上で，その問題点を述べた．そして，その問題点を踏まえて，分散スケジューリング機構を構成するネットワークとコアの設計方針をまとめた．

次章では，本章での方針をもとに設計した，ネットワークとコアの詳細な構成について述べる．

第 3 章

分散スケジューリング機構

3.1 緒言

本章では，分散スケジューリング機構を構成するネットワークのアルゴリズム，および，DDP コアのアーキテクチャについて述べる．分散スケジューリング機構は，ローカル情報に基づき，自律的に宛先を決定可能なネットワーク内のスイッチ用スケジューラ，および，LST 機構を搭載した DDP コアを統合することで実現する．

3.2 全体構成

図 3.1 にネットワークと DDP コアを含む全体像を示す．ネットワークは多段相互結合網で構成した．図 3.1 は，7 コア，1 入出力ポートの場合の例である．以下，ネットワーク内の各スイッチのアルゴリズム，および，DDP コアのアーキテクチャを示す．

3.3 ネットワークの構成

ネットワーク内の各スイッチは，図 3.2 に示すセルフタイム型パイプライン (以下，STP : Self-Timed Pipeline) をベースとした，2 × 2 セルフタイム型スイッチモジュールで構成される．2 × 2 セルフタイム型スイッチモジュールの構成を図 3.3 に示す．STP は複数の転送制御回路 C (以下，C 素子 : Coincidence flip-flop) から構成される．CB 素子は分岐機能を持つ C 素子，CM 素子はマージ機能を持つ C 素子である．宛先となるパイプラインステージは，前段のロジックから出力される制御信号によって決定され，CB 素子に入力される．

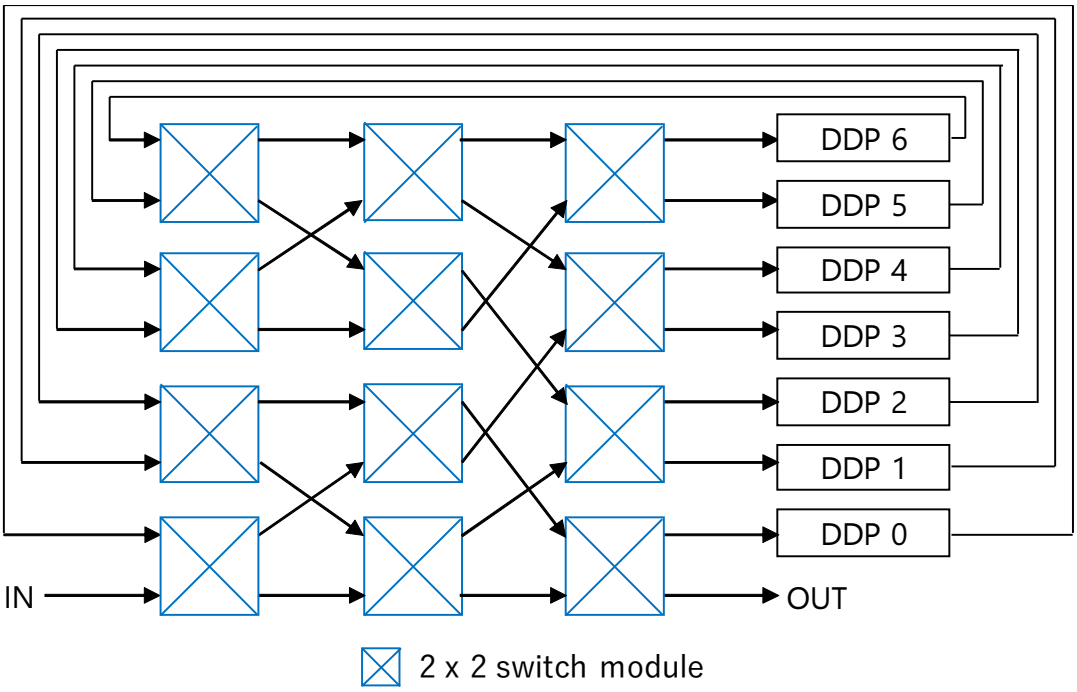


図 3.1　多段相互結合網の構成 (7 コアの場合)

タスクがパイプラインステージに到着すると，宛先となるパイプラインステージの C 素子と，データ転送要求信号 (Send 信号)，データ転送許可信号 (Ack 信号) をやりとりすることで，タスクが宛先のスイッチに転送される．CM は 2 つのパイプラインステージから到着するタスクを調停する．STP 自体は弾力性を有しているため，両方向から同時にタスクが到着した場合は，先着順などの事前に定義された方針に基づいて，タスク転送を調停することができる．有効なタスクが到着したパイプラインステージ間のみがハンドシェイクで局所的に動作するため，タスク転送時にのみ電力を消費する，自律的な省電力機能を有している [6]．具体的には，STP は図 3.4 のタイミングチャートに基づいて動作している．タスクを転送する場合，Send 信号を隣接する C 素子に転送する．そして，転送可能であれば，Ack 信号が返される．その後，CP 信号をデータラッチに送ることで，データラッチにタスクが取り込まれ，次のパイプラインステージにタスクを転送することができる．

タスクを転送するためには，各スイッチが自律的に宛先を決定するアルゴリズムが必要である．本研究では，2.5.1 項でも述べたように，各スイッチがローカルに持つ情報に基づい

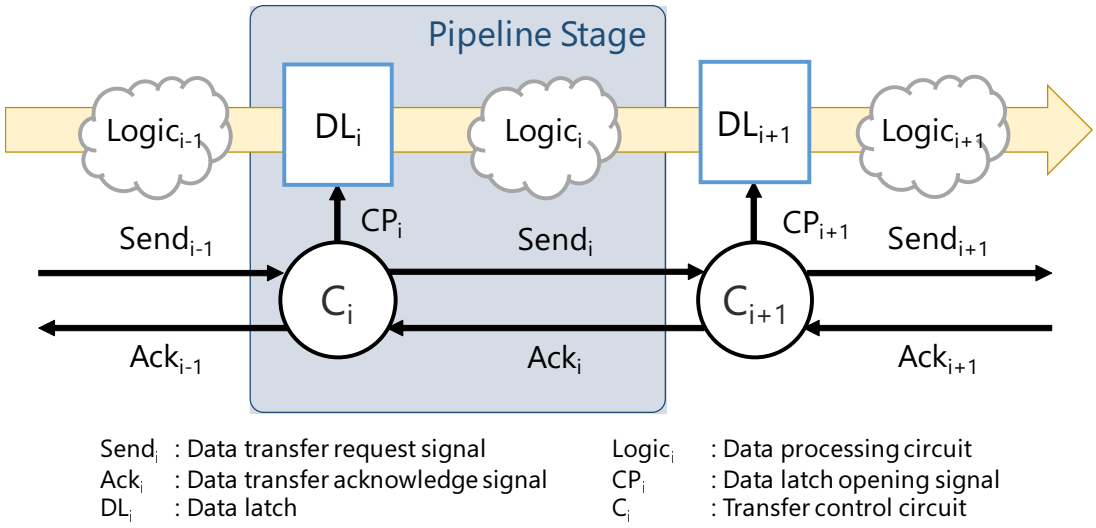


図 3.2　セルフタイム型パイプライン (STP) の構成

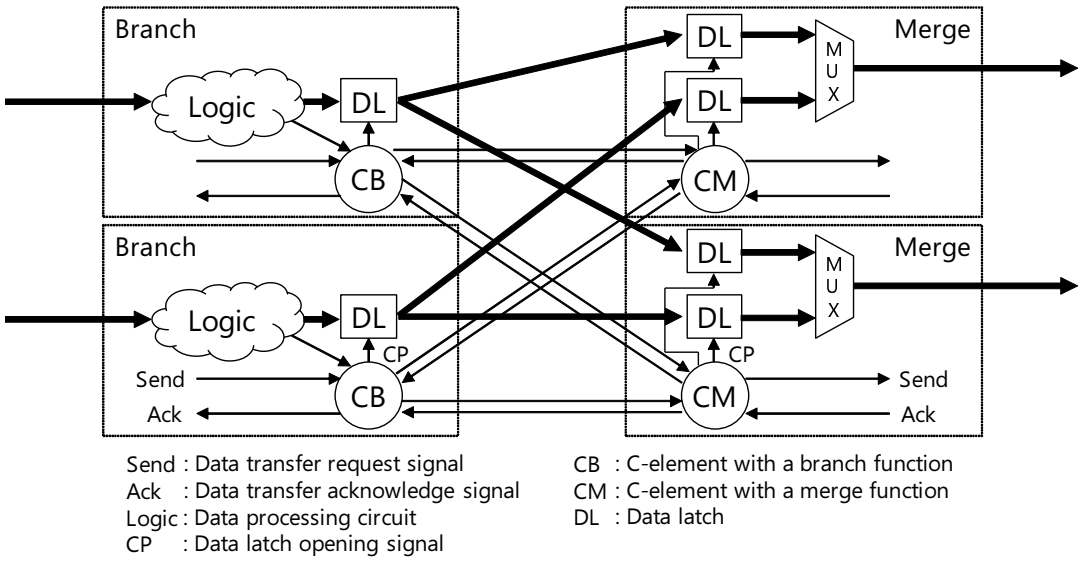


図 3.3　2 × 2 セルフタイム型スイッチモジュールの構成

て宛先を決定する方針を採った．各スイッチがローカルに判断できる情報として，過去の転送先，過去の通過タスクの稼働率がある．コアに直接接続されないスイッチは，コアの状態を精密に予測できないため，これらの情報をもとにして，経験的に宛先を決定する．コアに直接接続されるスイッチは，過去のタスク割り当てからコアの状態を判断することができるため，DDP コアが LST で動作していることを考慮し，余裕時間ベースのアルゴリズムを採用した．

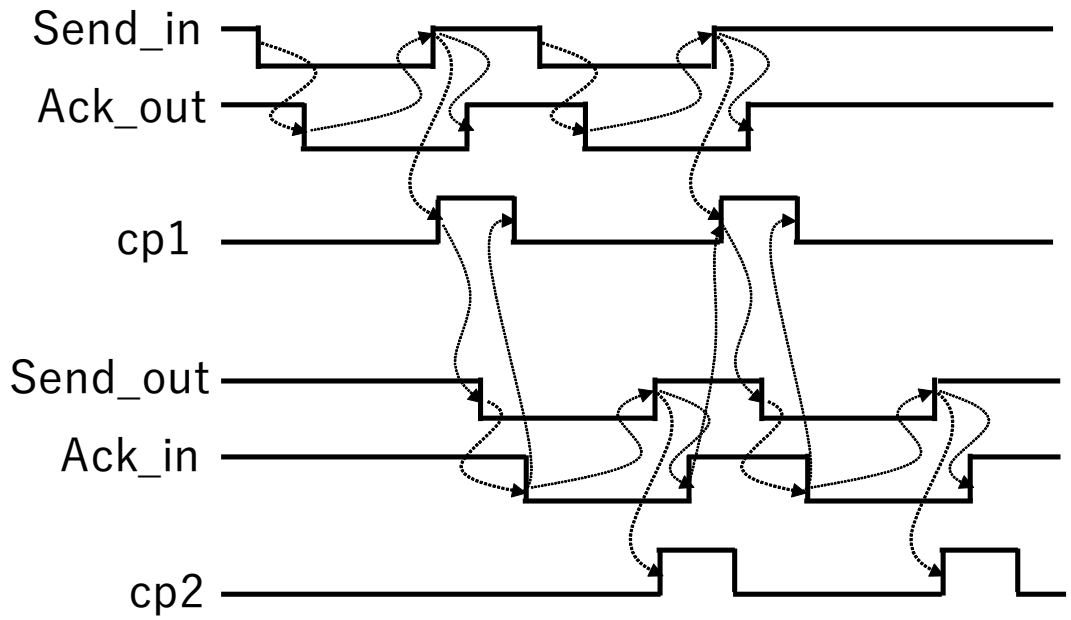


図 3.4　セルフタイム型パイプライン (STP) のタイミングチャート

3.3.1　コアに直接接続されないスイッチ

過去の転送先，過去の通過タスクの稼働率をもとにした，ラウンドロビン，稼働率ベース，タスク履歴ベースの 3 つのアルゴリズムを検討した．

ラウンドロビン

スイッチに到着したタスクを候補スイッチに，交互に転送する方式である．

稼働率ベース

図 3.5 に示すように，候補スイッチごとに，過去の通過タスクの稼働率を蓄積し，合計稼働率の小さい方にタスクを転送する方式である．各タスクの稼働率は，

$$Utilization = T_{rel\_deadline} \ / \ T_{exe} \tag{3.1}$$

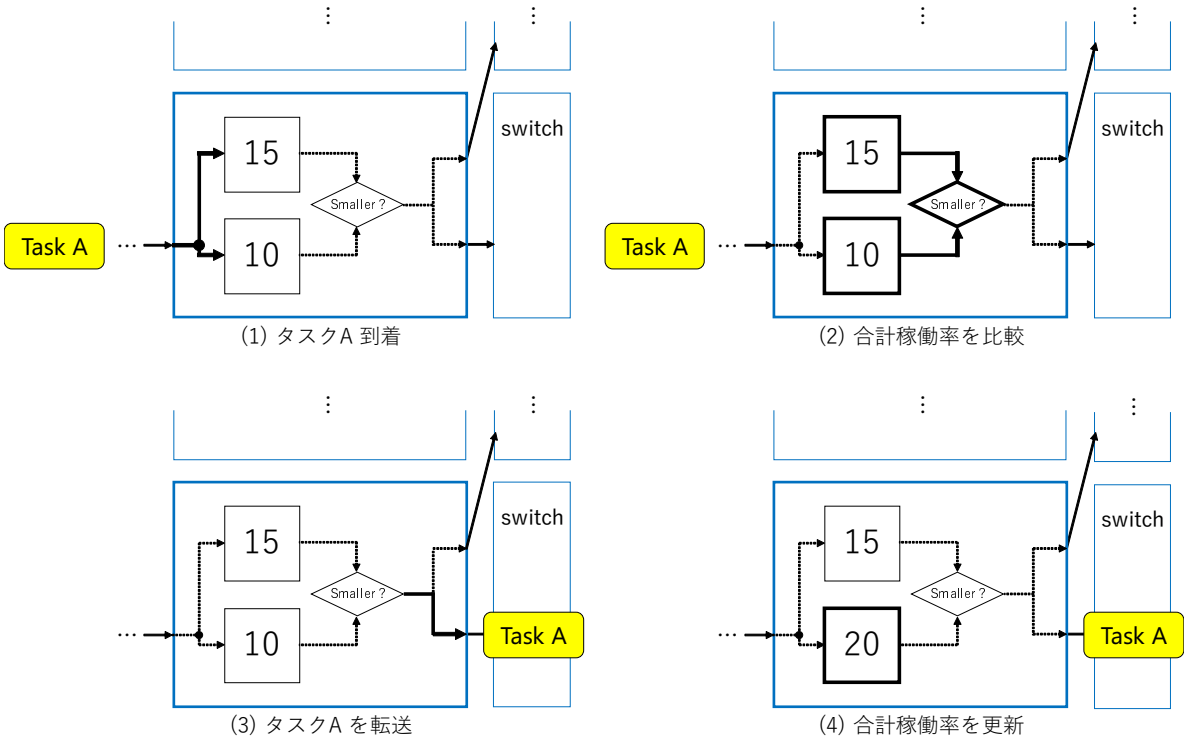


図 3.5　稼働率ベースアルゴリズムの動作例

で求められる．タスクが到着すると，以下のアルゴリズムでスイッチが動作する．図 3.5 では，すでに稼働率 10%，稼働率 15%のタスクがスイッチを通過しており，その後，稼働率 10%のタスク A がスイッチに到着したと仮定する．

1. タスク A がスイッチに到着する
2. 各候補スイッチの，過去の通過タスクの合計稼働率を比較する
3. 合計稼働率が小さい，スイッチにタスク A を転送する．
4. タスク A を転送した候補スイッチの合計稼働率に，タスク A の稼働率を加算する

他のタスクが到着した場合も，同様の動作を繰り返す．

タスク履歴ベース

タスク履歴ベースでは，各スイッチは，履歴テーブルを持ち，タスクが到着すると，履歴テーブルを参照して，宛先を決定する．各タスクにはタスク ID を付与する．同一のタスクは，同一のタスク ID を持つ．詳細な動作を以下に示す．履歴テーブル内の other には，常に最後に転送したタスクの宛先を記録する．図 3.6–図 3.9 では，履歴テーブルのエントリー数は 2 とし，各タスクの稼働率の大小関係は，(タスク ID: 3) < (タスク ID: 0) < (タスク ID: 1) < (タスク ID: 2) とする．

履歴テーブルに空きがある場合 (図 3.6)

履歴テーブルに空きがある場合は，到着タスクを履歴テーブルに記録する．宛先には，other に記録されている宛先とは別の宛先を選択する．other に何も記録されていない場合は，宛先に 0 を選択する．

到着タスクが履歴テーブルに記録されている場合 (図 3.7)

到着タスクと同じタスク ID が履歴テーブルに記録されている場合は，記録されている宛先とは別の宛先を選択し，テーブルを更新する．

到着タスクの稼働率 > 履歴テーブル内のタスクの最小稼働率 の場合 (図 3.8)

到着タスクの稼働率が，履歴テーブル内のタスクの最小稼働率よりも大きい場合は，最小稼働率のタスクの項目を，到着タスクで上書きする．宛先には，other に記録されている宛先とは別の宛先を選択する．

到着タスクの稼働率 ≤ 履歴テーブル内のタスクの最小稼働率 の場合 (図 3.9)

到着タスクの稼働率が，履歴テーブル内のタスクの最小稼働率と同じ，または，小さい場合は，到着タスクの情報をテーブルに記録せず，other に記録されている宛先とは別の宛先にタスクを転送する．よって，履歴テーブルに記録されていないタスクは，ラウンドロビンで割り当てることになる．

上記のアルゴリズムにより稼働率の大きなタスクがあるコアに偏らないようにする．

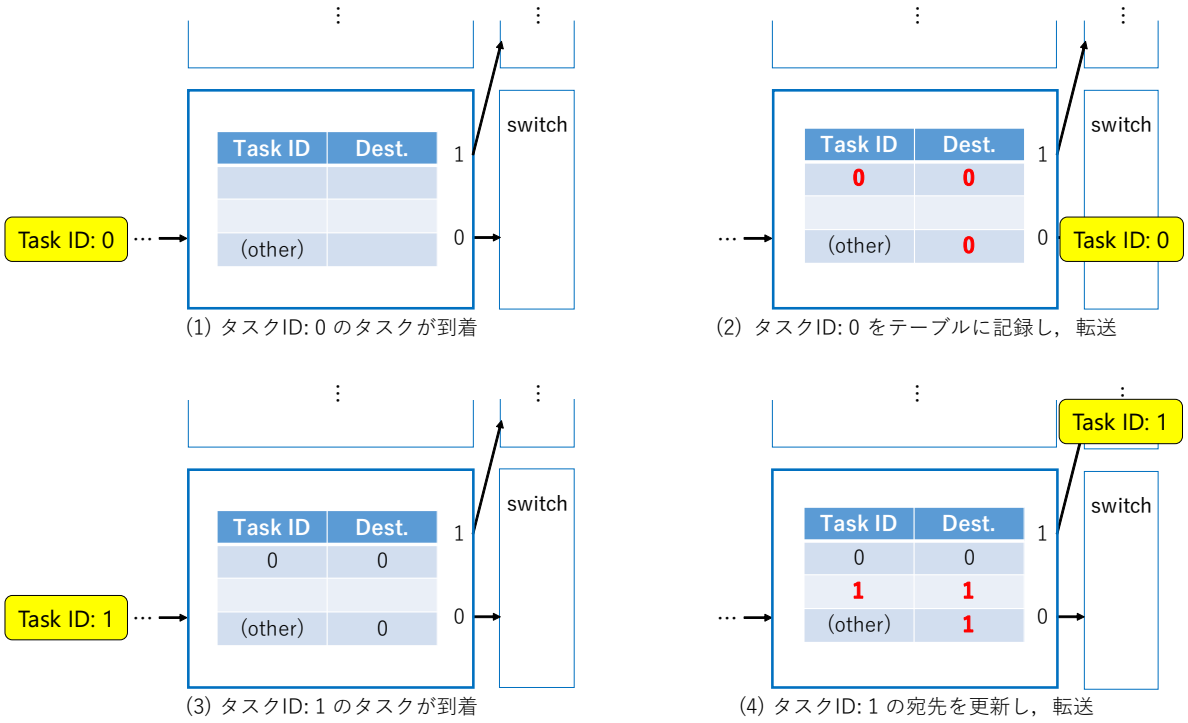


図 3.6　履歴テーブルに空きがある場合

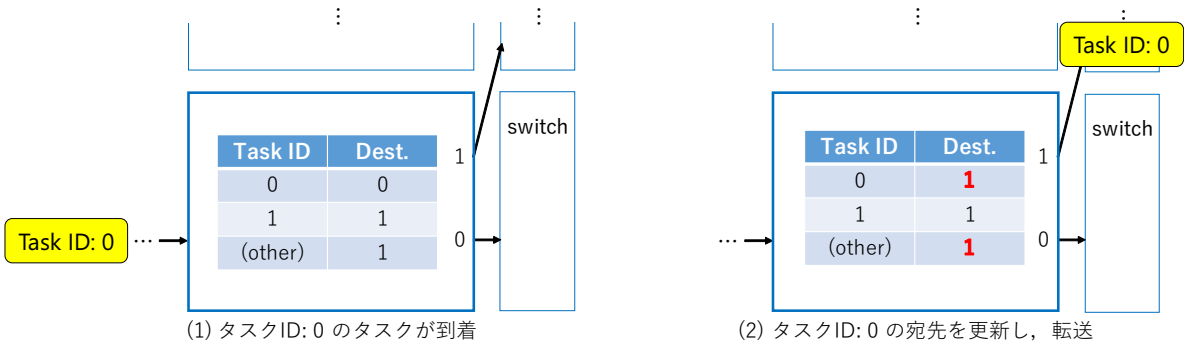


図 3.7　到着タスクが履歴テーブルに記録されている場合

3.3.2　コアに直接接続されるスイッチ

コアに直接接続されているスイッチには余裕時間ベースのアルゴリズムを採用した．図 3.10 に示すように，タスクが到着すると，以下のアルゴリズムでスイッチが動作する．

1. タスク A がスイッチに到着する
2. 到着タスクを各候補コアに割り当てたと仮定して，余裕時間を計算し，比較する

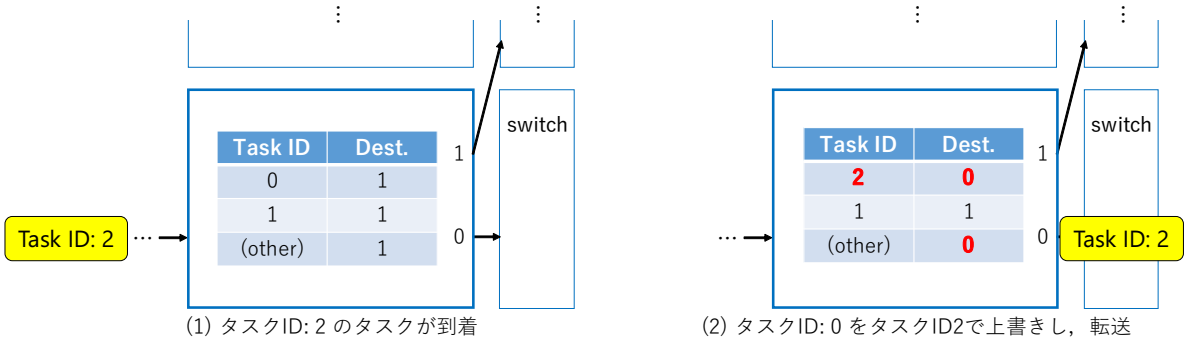


図 3.8　到着タスクの稼働率 > 履歴テーブル内のタスクの最小稼働率 の場合

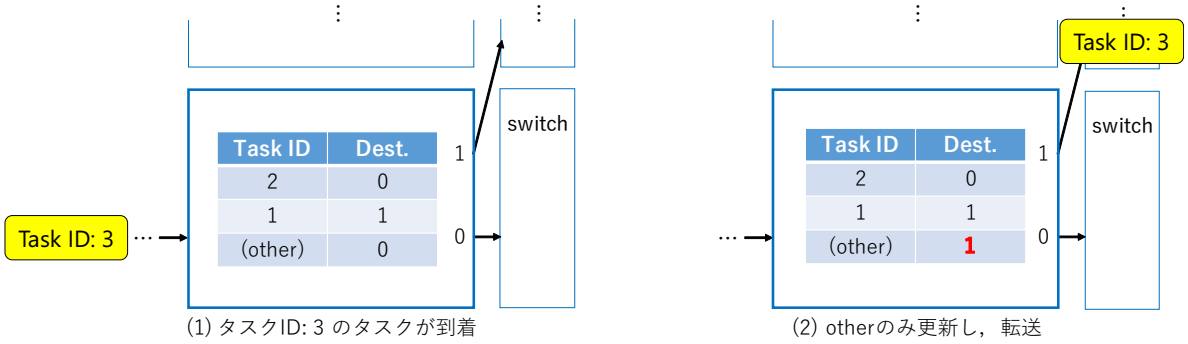


図 3.9　到着タスクの稼働率 ≤ 履歴テーブル内のタスクの最小稼働率 の場合

3. 最も余裕のある (余裕時間の長い) コアにタスクを割り当てる
4. タスク A を転送した候補コアの余裕時間を更新する

他のタスクが到着した場合も，同様の動作を繰り返す．

上記の余裕時間は，各候補コアに割り当てられたタスクの最小余裕時間である．ここでは，最小余裕時間を T'_{slack} と定義する．よって，到着タスクをすべて候補コアにそれぞれ割り当てたと仮定して，T'_{slack} を計算し，最大の T'_{slack} を有するコアにタスクが割り当てられる．

T'_{slack} を計算するための擬似コードをアルゴリズム 1 に示す．T'_{slack} の算出方法は以下の 3 つ分類できる．ここで，$t_{req}(i)$ は i 番目のタスクの実行要求時刻であり，$t_{req}(l)$ は，コアがアイドル状態になった後に割り当てられた最初のタスクである，l 番目のタスクの実行要求時刻である．すなわち，時刻 $t_{req}(l)$ から時刻 $t_{req}(i)$ まで，コアが連続的にビジー状態で

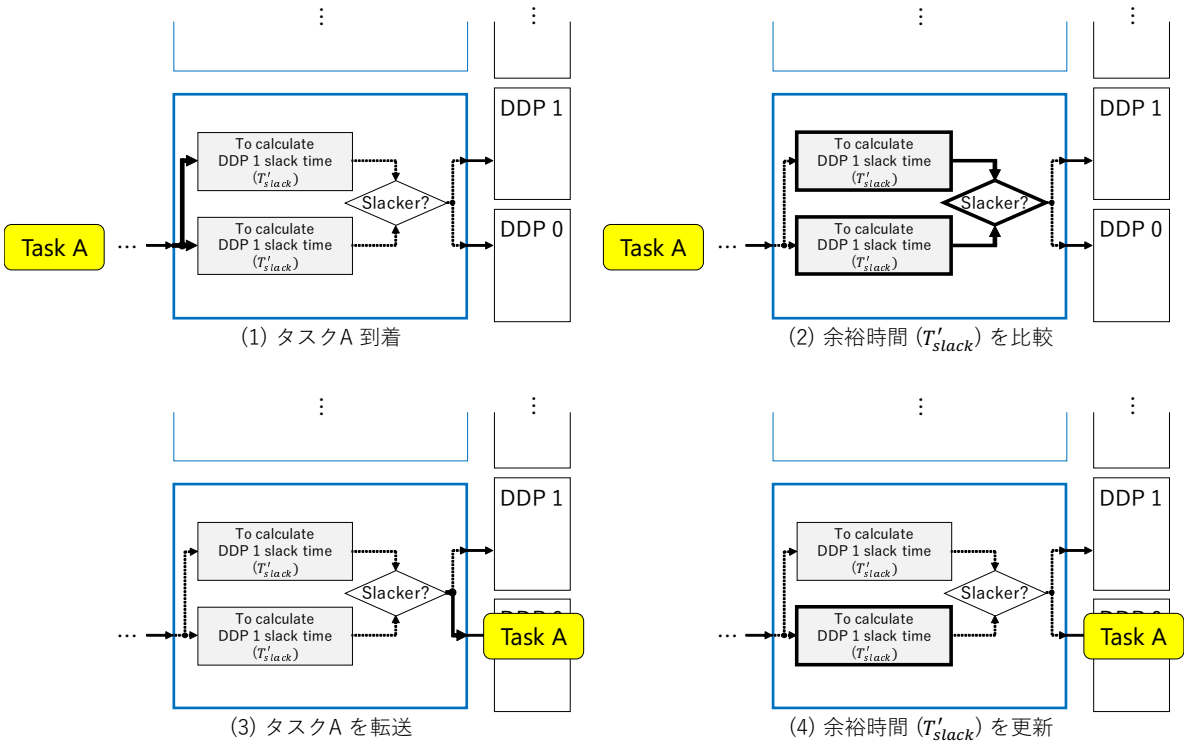


図 3.10　余裕時間ベースアルゴリズムの動作例

あったことを意味する．$T_{exe}(k)$ は，l 番目のタスクから i 番目のタスクまでを実行するのに必要な合計時間である．

Case 1: コアに実行中のタスクが存在しない場合

図 3.11 (a) は，タスク要求時にコア内でタスクが実行されてない場合である．このとき，要求タスクはすぐに実行を開始することができる．よって，T'_{slack} は，要求タスクの余裕時間 $T_{slack}(i)$ に設定され，i は l に割り当てられる．

Case 2: 要求タスクの余裕時間が，コア内の既存タスクの余裕時間よりも短い場合

図 3.11 (b) は，要求タスクが，すでにコアに割り当てられている 1 つまたは複数のタスクよりも前に実行を完了する場合である．よって，T'_{slack} は，前時刻の余裕時間 $T'_{slack}(i-1)$ と要求タスクの実行時間 $T_{exe}(i)$ の差で求められる．

Case 3: 要求タスクの余裕時間が，コア内の既存タスクの余裕時間よりも長い場合

図 3.11 (c) は，要求タスクが実行待ち状態になり，要求タスクよりも前に，既存のタス

Algorithm 1 最小余裕時間 T'_{slack} を算出する擬似コード

if $((t_{req}(i) - t_{req}(l)) - \sum_{k=l}^{i-1} T_{exe}(k) \geq 0)$ **then**

{/* Case 1 */}

$T'_{slack}(i) = T_{slack}(i)$

$l = i$

else if $T'_{slack}(i-1) > T_{slack}(i)$ **then**

{/* Case 2 */}

$T'_{slack}(i) = T'_{slack}(i-1) - T_{exe}(i)$

else

{/* Case 3 */}

$T'_{slack}(i) = T_{slack}(i) - (\sum_{k=l}^{i-1} T_{exe}(k) - (t_{req}(i) - t_{req}(l)))$

end if

クが実行される場合である．よって，T'_{slack} は，要求タスクの余裕時間 $T_{slack}(i)$ と，l 番目のタスクから $i-1$ 番目のタスクまでの残り実行時間の差で求められる．

3.3.3　タスク履歴ベーススイッチ回路の構成

本研究では，3.3.1 項，3.3.2 項で説明したアルゴリズムのうち，タスク履歴ベースを実現する回路構成を設計した．本研究では，マイグレーションなしを前提としているため，タスクは入力時にのみネットワークを通過する．よって，図 3.3 に示す 2 つの分岐機構のうち，下側の分岐機構のみタスクが通過するため，マージ機構には，1 方向からのみタスクが到着する．そのため，各マージ機構は 1 方向からの到着タスクに対する履歴テーブルを持つ．前段スイッチモジュール内のマージ機構が，後段スイッチモジュールのための履歴テーブルを持ち，前段スイッチモジュール内のマージ機構から出力された宛先情報が，後段スイッチモジュール内の分岐機構の制御信号として入力される．

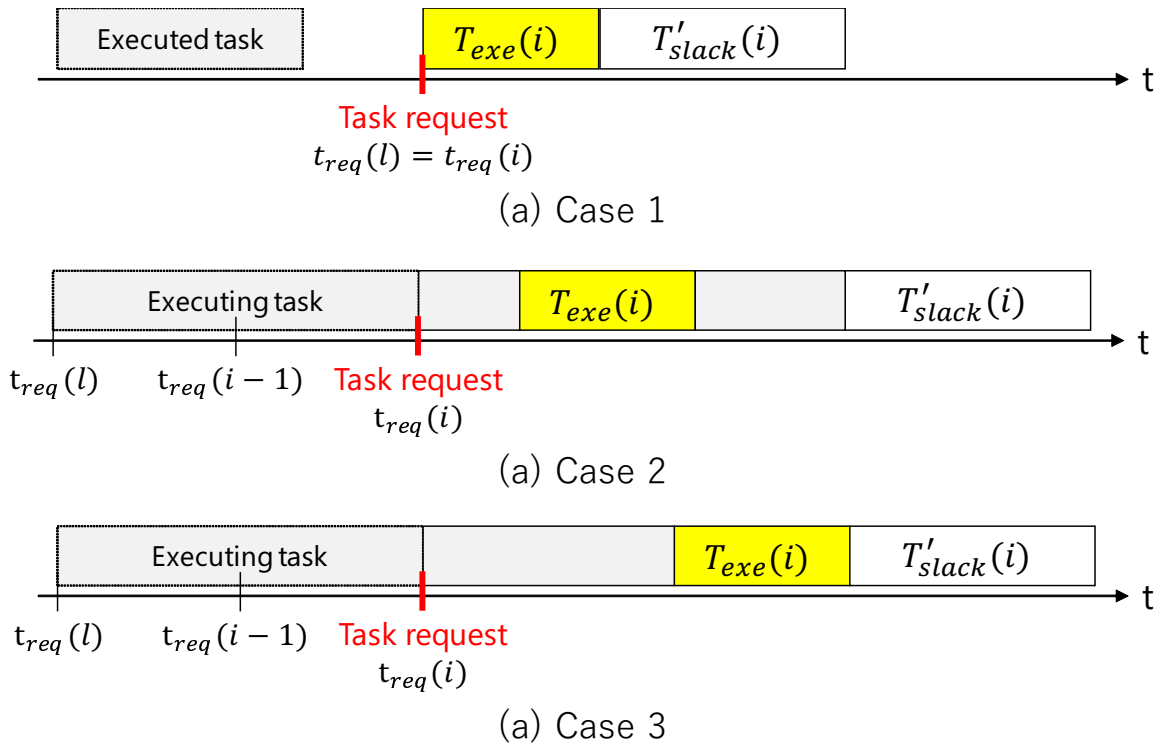


図 3.11　T'_{slack} 算出方法の分類

タスク履歴ベースを実現するための回路構成を図 3.12 に示す．ENTRY は履歴テーブルである．*packet_in_0* から，タスクが入力されると，ENTRY 内で保持しているタスク ID と到着タスクのタスク ID を比較し，図 3.6 – 図 3.9 のどの条件にあてはまるかを判断する．そして，*dest_sw* から宛先を出力する．到着タスクは，データラッチを通過後，後段のスイッチモジュールに転送される．タスクは，図 3.13，表 3.1 のフォーマットでパケットとしてネットワークに入力される．ネットワーク通過後は，*taskID*，*util* がパケットから削除され，DDP コアに入力される．ここでは，1 タスクは，1 パケットとして入力されるものとする．

3.4　コアの構成

コアには，DDP を用いる．DDP はパケット単位でタスクを実行するため，ネットワークのスイッチモジュールによって割り当てられたタスクを，細粒度にスケジューリングすることができる．DDP では，パケットが入力されると，環状パイプライン内をパケットが周回

3.4　コアの構成

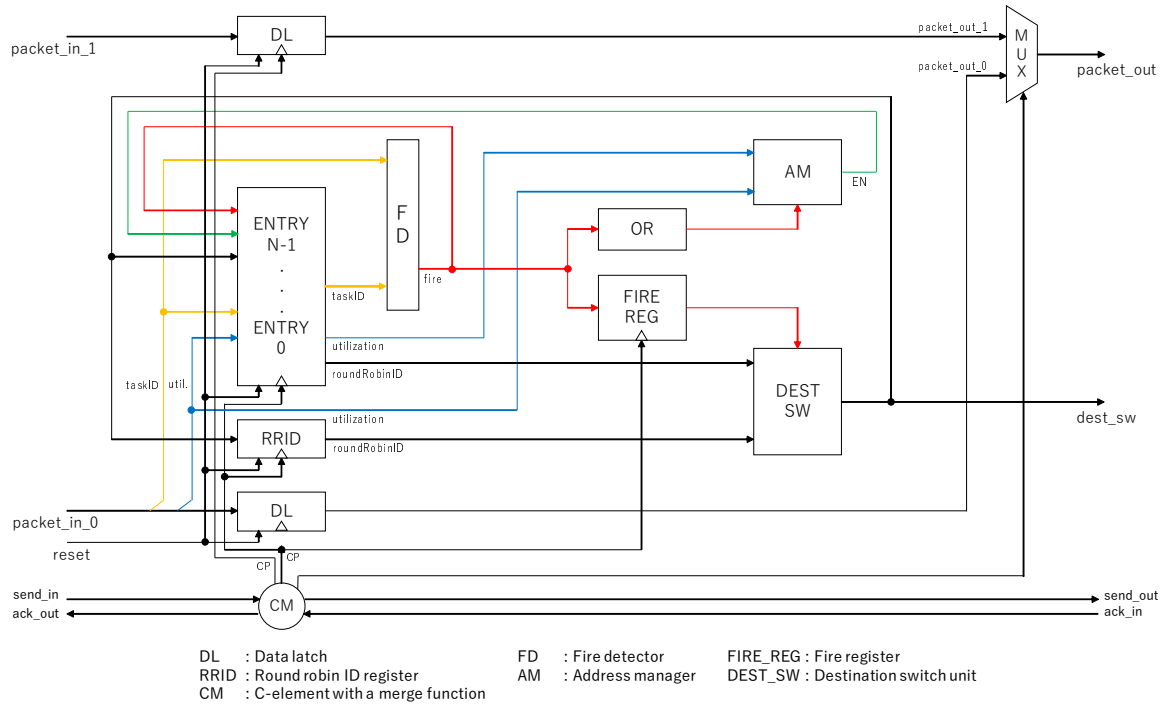


図 3.12　タスク履歴ベーススイッチ回路のブロック図

taskID	util	color	gen	dest	LR	CP	opc	C	Z	data

color : Identification　dest : Destination address　CP : Copy flag　C : Carry over flag　taskID : Task ID
gen : Generation　LR : Left or right flag　opc : Operation code　Z : Zero flag　util : Utilization

図 3.13　入力パケットフォーマット

し，周回後，次の命令をフェッチし，再び周回するか，外部に出力するか，を繰り返すことで処理が実行される．DDP は図 3.2 に示す STP を用いて実装されており，パケットの到着をトリガに，複数のパケットを多重に処理することができる．よって，環状パイプライン内のパケット数は，環状パイプラインの処理負荷を示している．リアルタイム処理を実現する場合，DDP 内のパケット数が多重処理可能な負荷を超えないように，パケットの優先度を考慮して，パケットの実行を中断/再開する必要がある．そのため，スケジューラの役割を持つ LST 機構を搭載した DDP を設計した [7][8]．LST 機構を搭載した DDP の構成を図 3.14 に示す．LST 機構を搭載した DDP は，従来の DDP に，優先度機構，負荷モニタ，タスクキューを追加することで実現できる．以下に，構成する各ステージの機能の概略を示す．

- パケット合流機構 (M: Merge)

表 3.1　パケットが保持する情報

フィールド名	名称	情報	ビット数
taskID	Task ID	タスクの識別情報	12bit
util	Utilization	タスクの稼働率	7bit
color	Color	パケットの識別情報	5bit
gen	Generation	パケットの世代 (順番)	5bit
dest	Destination	パケットの宛先	13bit
LR	Left or Right	パケットの左右の識別	1bit
CP	Copy flag	コピーの有無	1bit
opc	Operation code	命令	3bit
C	Carry flag	キャリーフラグ	1bit
Z	Zero flag	ゼロフラグ	1bit
data	Data	演算データ	32bit

DDP は，環状パイプライン内をパケットが周回することで，処理が実行される．よって，DDP 内部を周回する経路と外部からの入力経路がある．パケット合流機構では，内部周回パケットと外部入力パケットの合流を調停する．

- 定数読み出し機構 (CST: Constant memory)

  定数読み出し機構内のメモリには，定数演算を実行するために用いる定数値が格納されている．定数演算命令の場合は，メモリから取り出した定数値をパケットに付与する．定数演算命令でない場合は，メモリにアクセスすることなく，このステージを通過する．

- パケット待ち合わせ機構 (MM: Matching memory)

  DDP において 2 項演算は 2 つのパケットに保有されている，オペランド同士の演算として実行される．ペアとなるパケットは，*color*，*gen*，*dest*，*LR* から識別される．パケットはパケット待ち合わせ機構内の連想メモリに一時的に記憶され，ペアとなるパケットが到着すると，演算用のペアパケットを生成する．定数演算命令の場合は，この

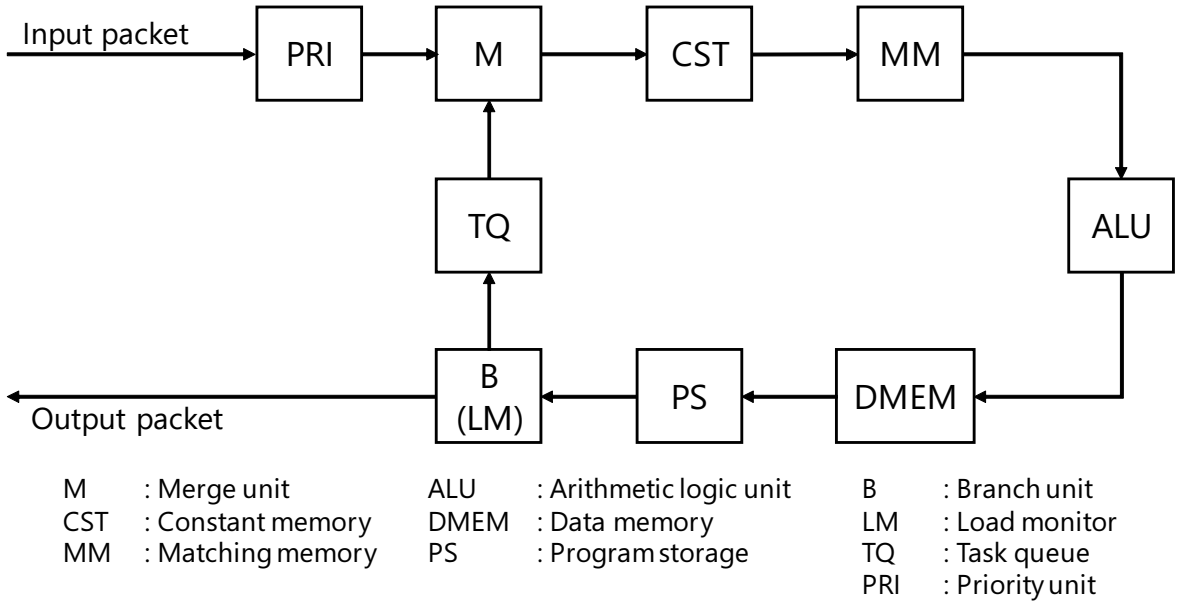


図 3.14　LST 機構を搭載した DDP の構成

ステージを通過する．

- 演算機構 (ALU: Arithmetic logic unit)

  パケットのオペレーションコードに従って，算術論理演算を実行する．演算後，演算結果，キャリーフラグ，ゼロフラグをパケットに書き込む．ロード/ストア命令の場合，演算機構では，データメモリのアドレスを計算し，パケットに付与する．

- データメモリアクセス機構 (DMEM: Data memory)

  ロード/ストア命令の場合は，データメモリにアクセスする．ロードしたデータは，パケットに付与する．

- 命令フェッチ機構 (PS: Program storage)

  命令フェッチ機構内のメモリには，実行するプログラムが格納されている．パケットの *dest* は，次の命令のアドレスを示している．よって，*dest* をアドレスとして，メモリから，次の宛先とオペレーションコードをフェッチし，パケットを書き換える．また，不要となったパケットの削除機能も備わっており，削除命令が実行されると，パケットを削除する．

- 分岐機構 (B: Branch)

パケットの分岐フラグに応じて，環状パイプラインの外部，または，内部にパケットを出力する．

- 優先度機構 (PRI: Priority)

  入力パケットの初期優先度を決定する．優先度機構には，各タスクの実行時間，相対デッドライン時間，優先度クラスが格納されている．優先度クラスは，ハードリアルタイムタスク，ソフトリアルタイムタスク，通常タスクのいずれかで定義される．パケットの *color* をアドレスとしてメモリにアクセスし，実行時間，相対デッドライン時間，優先度クラスをフェッチする．そして，相対デッドライン時間から実行時間を減算することで初期余裕時間を計算し，初期余裕時間と優先度クラスをパケットに付与する．

- 負荷モニタ (LM: Load monitor)

  DDP 内のパケット数から，処理負荷を観測する．パケット数は，特定のステージを通過するパケットをカウントすることで観測できる．DDP パケットに保持されている実行制御情報から，パケット数の増減を判断する．パケット増加の要因には，外部入力，コピー命令の実行がある．また，パケット減少の要因には，外部出力，削除命令の実行，パケット待ち合わせ機構において 2 つのパケットからペアパケットが生成された場合がある．処理負荷を示すパケット数の情報は，タスクキューに転送される．

- タスクキュー (TQ: Task queue)

  負荷モニタからのパケット数の情報をもとに，多重処理可能であるかどうかを判断し，多重処理可能な負荷を超える場合は，タスクキューに一部のパケットをキューイングすることで，パケットの中断/再開を行う．タスクキューにはソーターがあり，キューイングされたタスクは，余裕時間の短い順にソートされる．パケットがタスクキューに到着すると，キューイングされているタスクの余裕時間を更新する．そして，到着パケットの余裕時間とキュー内のタスクの余裕時間を比較し，余裕時間の短い (優先度の高い) パケットが次のステージに転送される．また，外部出力などにより，DDP 内のパケットが減少した場合は，キュー内のタスクの中で，最も余裕時間の短いタスクが次のステージに転送される．

タスクキューの回路構成を図 3.15 に示す．タスクキューは主に優先度キュー (PQ: Priority queue) および，パケット削除機能付き C 素子 (CE 素子) から構成される．*packet_in_q* からパケットが入力されると，パケットの優先度クラスに対応する優先度キューにエンキューされる．そして，最も優先度の高いパケットを優先度キューからデキューし，実行する．環状パイプライン内のパケット数が多重処理可能なパケット数を超える場合は，優先度の低いパケットを優先度キューに退避し，CE 素子を用いてパケットを削除することで実行を中断する．タスクキューは，優先度キューを管理するための 3 つの制御回路から構成される．

- CE_CTRL

  CE_CTRL は，負荷モニタからの情報に基づいて，パケットをキューイングするかどうかを判断する．キューイングする場合は，削除制御信号 (*del_ctrl_sig*) をアサートし，CE 素子を用いてパケットを削除する．
- Q_SEL

  Q_SEL は，MUX を制御する．MUX は，各優先度キューの出力パケットから最も優先度の高いパケットを選択する．各優先度キューから出力される *count* は，キューイングされているパケット数を示す．*count* がゼロの場合は，優先度キューが空であることを意味する．
- Q_CTRL

  Q_CTRL は，前時刻のパケットの状態を管理する PPS (Previous Packet State) からの情報に基づいて，優先度キューからパケットを削除する．PPS は前時刻のパケットが優先度キューにキューイングされたか，パイプラインステージに転送されたかを管理する．パケットがパイプラインステージに転送された場合は，パケットを優先度キューから削除する必要がある．そのため，削除するパケットがキューイングされている優先度キューに対して，パケット削除信号 (*packet_del_sig*) をアサートする．

タスクキュー内の優先度キューの回路構成を図 3.16 に示す．パケットが到着すると，SLACK_TIME_UPDATER により，ENTRY 内のすべてのパケットの余裕時間を更新す

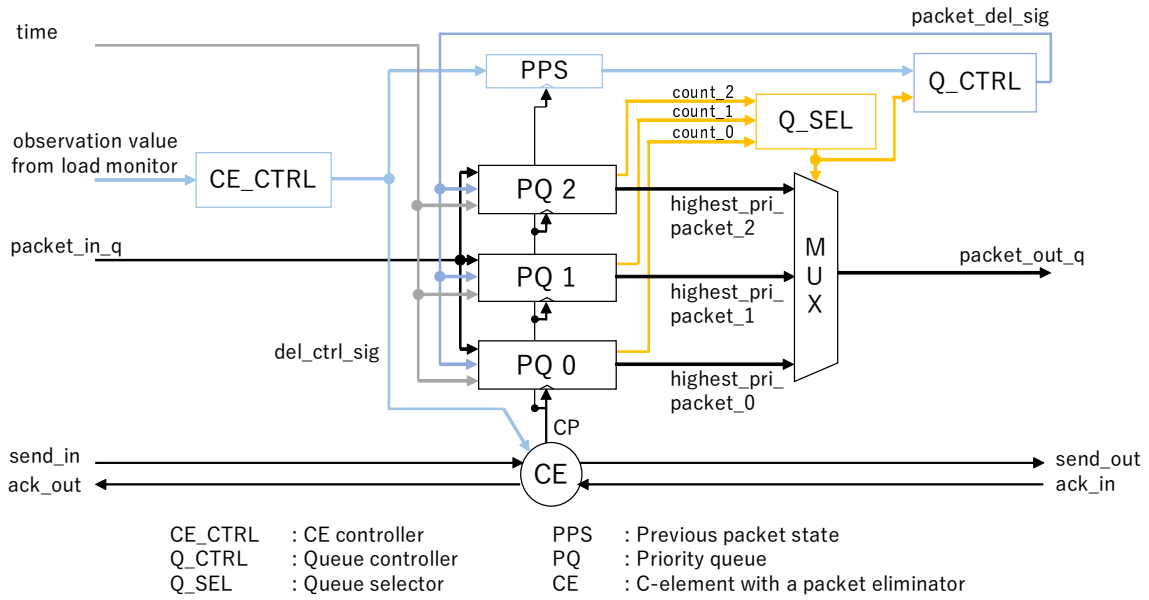


図 3.15　タスクキューの回路構成

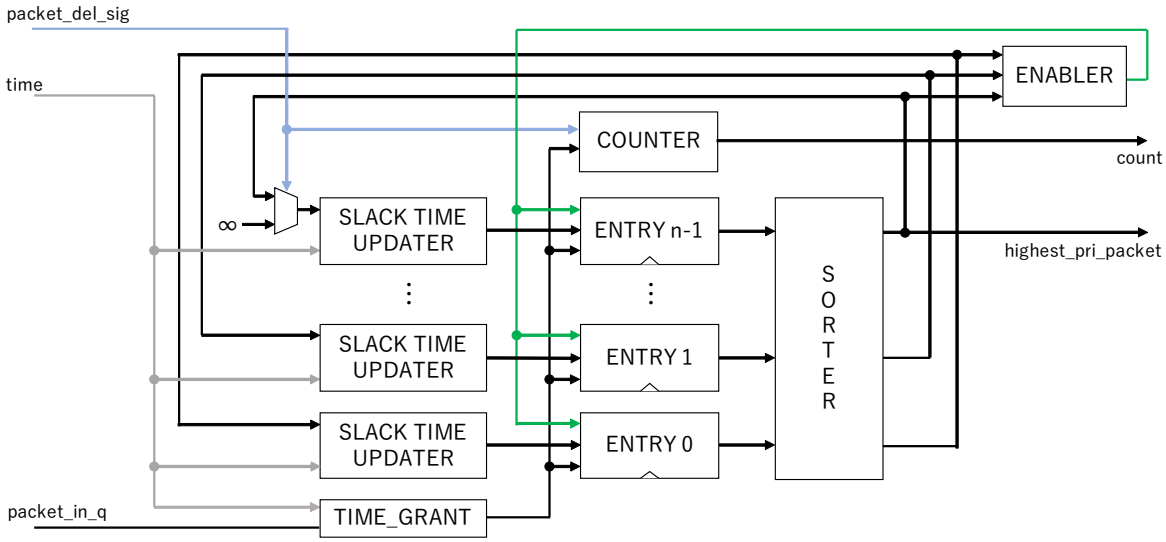


図 3.16　優先度キューの回路構成

る．その後，パケットはいずれか 1 つの ENTRY にキューイングされる．ENABLER は，同じパケットが複数の ENTRY にキューイングされないように，空の ENTRY のうち，1 つの ENTRY にのみ書き込み許可を与えるように各 ENTRY を制御する．ENTRY 内のパケットは，SORTER により余裕時間の短い順にソートされ，最も余裕時間の短いパケットが優先度キューから出力される．

3.5　結言

本章では，分散スケジューリング機構を構成する，ネットワーク内のスイッチ用スケジューラのアルゴリズム，および，コアとなる LST 機構を搭載した DDP のアーキテクチャを述べた．ローカルに持つ情報に基づいて宛先を決定可能なスイッチ用スケジューラは，ラウンドロビン，余裕時間ベース，タスク履歴ベース，余裕時間ベースのアルゴリズムによって実現できる．また，LST 機構を搭載した DDP は，従来の DDP に，優先度機構，負荷モニタ，タスクキューを追加することで実現できる．

次章では，分散スケジューリング機構のスケジューリング性能を評価した結果を述べる．

第 4 章

評価

4.1 緒言

本章では，第 2 章，第 3 章で述べた分散スケジューリング機構のスケジューリング性能を評価した結果を述べる．本来は，回路シミュレーションによって性能を評価するべきであるが，回路シミュレーションによるマルチコアシステムの評価には膨大な時間が必要である．よって，スイッチ，および，シングルコア DDP を 65nm CMOS 標準セルライブラリを用いて論理合成を行い，性能パラメータを抽出し，シミュレーションベースで評価を行った．一次的な評価として，ラウンドロビン，稼働率ベース，タスク履歴ベースの性能比較を行った後に，提案方式のスケジューリング性能を評価するために，グローバルスケジューリング方式の EDF，LST を比較対象として平均システム稼働率，および，スケジューリング成功率を比較した．

4.2 回路設計による性能パラメータの抽出

タスク履歴ベーススイッチ，LST 機構を搭載した DDP を 65nm CMOS 標準セルライブラリを用いて設計した．各回路は，Verilog-HDL を用いて設計し，Synopsis 社の Design Compiler によって論理合成を行った．

表 4.1　論理合成時の設定条件 (スイッチ)

Process	SOTB 65nm CMOS (V_{DD}: 0.75V)
Input packet	81 bit
History table	81 bit × 4 words

表 4.2　論理合成時の設定条件 (DDP)

Process	SOTB 65nm CMOS (V_{DD}: 0.75V)
# Stages of STP ring	10 stages
Input packet	62 bit
Task queue (TQ)	105 bit × 8 words × 3 queues
Constant memory (CST)	35 bit × 8192 words
Matching memory (MM)	62 bit × 96 words
Data memory (DMEM)	32 bit × 8192 words
Program storage (PS)	20 bit × 8192 words

4.2.1　回路仕様

タスク履歴ベーススイッチ，LST 機構を搭載した DDP を論理合成する際の設定条件はそれぞれ，表 4.1，表 4.2 のように設計した．スイッチへの入力パケットは 32 bit データを保持した 81 bit のパケットであり，DDP への入力パケットはネットワークでのスケジューリングのみに必要なフィールドを削除した，62 bit のパケットである．入力パケットに含まれる各フィールドは，表 3.1 に示すように設定した．また，タスク履歴ベースの履歴テーブル数は 4 とした．

4.2.2　面積評価

タスク履歴ベーススイッチ，LST 機構を搭載した DDP を 65nm CMOS 標準セルライブラリを用いて設計，論理合成を行った．その結果を表 4.3 に示す．タスクがスイッチを通過

表 4.3　論理合成後の面積評価

	switch based on task history	DDP based on LST [8]
time [ns]	20.7 (pass)	260.0 (circulate)
standard cells	2.3k	67.0k
area [mm^2]	0.0128	0.295

するために必要な時間は 20.7ns，DDP を 1 周するために必要な時間 (1 命令の実行時間) は 260.0ns となった．ASIC を設計する場合，各マクロ内やマクロ間に電源や配線領域を考慮する必要があるため，総セル面積は，セル密度を 80%として換算した結果を示している．スイッチの総論理セル数は，2.3k 個 (面積 : 0.0128 mm^2)，DDP の総論理セル数は，67.0k 個 (面積 : 0.295 mm^2) となった．

コア数を N とすると，多段相互結合網の段数は，$\log_2$N となる．よって，分散スケジューリング機構のスイッチの総面積 (ネットワークの面積) は，各スイッチの出力ポート数を 2 とすると，

$$0.0128 \times \log_2 \mathrm{N} \times (N \ / \ 2) \ \ [\mathrm{mm}^2] \tag{4.1}$$

コアの総面積は，

$$0.295 \times N \ \ [\mathrm{mm}^2] \tag{4.2}$$

から，おおよその値を見積もることができる．厳密には，ラウンドロビン，稼働率ベース，余裕時間ベースのスイッチを設計し，評価する必要がある．これらの論理合成結果をもとに，評価用アーキテクチャシミュレータを作成した．

4.3　マルチコアアーキテクチャシミュレータの評価

回路シミュレーション上で，複数の DDP コアに対して評価タスクセットを実行すると非常に多くの時間が必要となる．そこで，シミュレーション時間短縮のために，アーキテクチャシミュレータを作成した．作成したアーキテクチャシミュレータでは，コアの性能を示

表 4.4　シミュレータ評価時のタスクセット

Name	Exec. time [ms]	Period [ms]	Util.
T1　simple	0.06	2	3%
T2　monitor	0.10	10	1%
T3　compute	1.00	10	10%
T4　network	1.00	10	10%
T5　service	1.20	20	6%
T6　input	0.50	5	10%
T7　output	1.00	10	10%
T8　PWM	0.04	2	2%
		Total:	52%

表 4.5　シミュレータの稼働率評価

DDP コアの多重度	アーキテクチャシミュレータ	回路シミュレーション
2	32.1%	32.0%
4	21.6%	21.8%

すパラメータとして，DDP コアで多重処理可能なタスク数 (多重度)，DDP 内のタスクの周回時間 (1 命令の実行時間) を設定することができる．要求タスク数が多重度を超えない限り，タスクを同時に実行することが可能である．要求タスク数が多重度を超える場合は，周回時間が経過するごとに，要求タスクの中から多重度を超えない範囲で優先度の高いタスクを選択し，実行する．

アーキテクチャシミュレータの評価として，LST 機構を搭載した DDP を論理合成し，回路シミュレーションにより求めた稼働率と、シミュレータの推定稼働率を比較した．ここでは，コアの稼働率は，ハイパーピリオドに対して，DDP 内で少なくとも 1 つのタスクが実行されている時間の割合としている．比較には表 4.4 に示すタスクセットを用いた [5].

比較結果を表 4.5 に示す．多重度は DDP コアで多重処理可能なタスク数である．表 4.5

より，作成したシミュレータは，最大 0.2%の誤差で，論理合成後の DDP コアの稼働率を見積もることが可能であることを確認できた．よって，シミュレータは，DDP コアの稼働率を妥当な精度で推定するために用いることができると言える．以降，このシミュレータを用いて，スケジューリング性能評価を行う．

4.4　評価タスクセット

スケジューリング性能評価のために，評価タスクセットを作成した．前提として，本研究では，周期タスクのみを対象とする．よって，相対デッドライン時間を 1 周期として，周期的にタスクが起動する．以降，相対デッドライン時間を周期と呼ぶ．また，全タスクが時刻 0 で実行要求されるとする．ネットワークの入力ポートは 1 つであるため，同時刻に要求されたタスクは，DDP の周回時間 (1 命令の実行時間) が 250ns として，250ns ずつ遅れて順番に入力されるものとする．

4.4.1　タスクセット内のタスクのパラメータ

タスクセットは，実行要求時刻，実行時間，周期，余裕時間，稼働率，タスク ID のパラメータを持つ複数のタスクから構成される．各タスクはネットワーク入力前に，余裕時間を計算し，パラメータに追加する．余裕時間は式 4.3 により計算する．

$$\text{余裕時間} \;=\; \text{周期} \;-\; \text{実行時間} \tag{4.3}$$

各タスクの稼働率と周期は，以下の範囲からランダムに決定する．

- 稼働率：1% – 20%
- 周期：1ms, 2ms, 4ms, 5ms, 10ms, 20ms

周期は，ハイパーピリオドが 20ms となるように，上記のようにした．ハイパーピリオドは，すべてのタスクの周期の最小公倍数である．すなわち，ハイパーピリオドの時間分，シ

タスク ID	要求時刻 [ms]	実行時間 [ms]	周期 [ms]	余裕時間 [ms]	稼働率
0	0	0.06	2	1.94	3%
1	0	0.10	10	9.90	1%
0	0	0.06	2	1.94	3%
1	0	0.10	10	9.90	1%
⋮					

(a) ベースタスクセットの複製

タスク ID	要求時刻 [ms]	実行時間 [ms]	周期 [ms]	余裕時間 [ms]	稼働率
0	0	0.06	2	1.94	3%
1	0	0.10	10	9.90	1%
2	0	1.00	10	9.00	10%
3	0	0.04	2	1.96	2%
⋮					

(b) ランダムなタスクセット

図 4.1　タスクセットの種類

ミュレーションを実行すれば，全タスク実行の組み合わせを網羅することができる．また，稼働率と周期から，式 4.4 により，実行時間を計算する．

$$\text{実行時間} \;=\; \text{周期} \;\times\; (\text{稼働率} \;/\; 100) \tag{4.4}$$

4.4.2　タスクセットの生成方法

コア数を N，DDP コアで多重処理可能なタスク数 (多重度) を P，タスクセットに含まれるタスクの合計稼働率を $U_{taskset\_util}$ とおくと，システム稼働率 U_{sys_util} は，$U_{taskset\_util} \;/\; (N \;\times\; P)$ で定義される．生成するタスクセットのシステム稼働率に応じて，$U_{taskset\_util} \;/\; (N \;\times\; P) \leq U_{sys\_util}$ である限り，新しいタスクをタスクセットに追加し，追加したタスクの稼働率を $U_{taskset\_util}$ に加算する．最後に生成するタスクのみ，$U_{taskset\_util} \;/\; (N \;\times\; P) = U_{sys\_util}$ となるように，タスクの稼働率を調整した．

本研究では，図 4.1 に示す，2 種類のタスクセットを用いる．図 4.1 (a) は，ベースとなるタスクセットをランダムに作成し，ベースタスクセットを複製したものである．タスクセット内のタスクに周期性がある場合を想定している．合計稼働率が U_{sys_util} となるタスクセットをベースとして，それを，$(N \;\times\; P)$ 倍して作成する．以降，図 4.1 (a) のタスクセットを，周期性のあるタスクセットと呼ぶ．また，図 4.1 (b) はタスクセット内に含まれるすべてのタスクをランダムに作成したものである．以降，図 4.1 (b) のタスクセットを，ランダムなタスクセットと呼ぶ．

4.5　スイッチアルゴリズムの評価

3.3.1 項において，ネットワーク内のスイッチのうち，コアに直接接続されていないスイッチのアルゴリズムとして，ラウンドロビン，稼働率ベース，タスク履歴ベースを検討した．これらのアルゴリズムを，各コアの稼働率の標準偏差の平均，スケジューリング成功率の観点で評価した．コアに直接接続されていないスイッチには上記いずれかのアルゴリズム，コアに直接接続されているスイッチには余裕時間ベースを用いて各コアにタスク割り当てを行い，アーキテクチャシミュレータにおいてタスクを実行し，平均稼働率，スケジューリング成功率を計測した．スケジューリング成功率は，式 4.5 から計算する．

$$\text{成功率} = \frac{\text{全タスクがスケジューリングに成功したタスクセット数}}{\text{総タスクセット数}} \tag{4.5}$$

評価条件は以下の通りである．

- コア数：16
- DDP コアの多重度：2
- 評価タスク数：周期性のあるタスクセット，ランダムなタスクセット，各 100 セット
- システム稼働率：60%

評価結果を表 4.6，4.7，4.8 に示す．タスク履歴ベースは，履歴テーブル数を 1 – 10 としてそれぞれ評価し，最も性能の高かったテーブル数 4 の場合を示している．デッドラインミスが起きないようにするためには，各コアの稼働率が均衡化し，平均標準偏差が小さくなる方が望ましいと考えられる．ランダムなタスクセットにおいては，平均標準偏差，スケジューリング成功率ともに，すべてのアルゴリズムが近い値となった．周期性のあるタスクセットにおいては，タスク履歴ベースが他のアルゴリズムに比べてスケジューリング成功率が高い結果となった．ラウンドロビンは，ラウンドロビンの周期性とタスクセット内のタスクの周期性が一致すると，稼働率の高いタスクがあるコアに集中して割り当てられることになり，各コアの稼働率の偏りが大きくなることで，デッドラインミスが起こっていると考えられる．また，稼働率ベースは，タスクセット内の各タスクの稼働率の差が大きくなると，

表 4.6　スイッチアルゴリズムの評価結果 (ラウンドロビン)

	周期性のあるタスクセット	ランダムなタスクセット
平均標準偏差	9.6	9.5
スケジューリング成功率	41%	89%

表 4.7　スイッチアルゴリズムの評価結果 (稼働率ベース)

	周期性のあるタスクセット	ランダムなタスクセット
平均標準偏差	9.2	8.5
スケジューリング成功率	77%	88%

各候補スイッチに対応する合計稼働率に大きな差ができ，合計稼働率の小さい方に集中してタスクが割り当てられることで，デッドラインミスが起こっていると考えられる．タスク履歴ベースは，稼働率の大きなタスクをテーブルに記録することで，稼働率の大きなタスクがあるコアに偏らないようにしているため，他のアルゴリズムに比べ，スケジューリング成功率が高くなったと考えられる．

これらの結果をふまえて，分散スケジューリング機構を構成するネットワーク内のスイッチのうち，コアに直接接続されていないスイッチにはタスク履歴ベース，コアに直接接続されているスイッチには余裕時間ベース，コアには LST 機構を搭載した DDP を採用し，スケジューリング性能の評価を行った．

表 4.8　スイッチアルゴリズムの評価結果 (タスク履歴ベース)

	周期性のあるタスクセット	ランダムなタスクセット
平均標準偏差	4.8	9.3
スケジューリング成功率	95%	90%

4.6　スケジューリング性能評価

提案方式のスケジューリング性能を評価するために，LST を採用したパーティショニングスケジューリング方式 (p-LST)，EDF を採用したグローバルスケジューリング方式 (g-EDF)，LST を採用したグローバルスケジューリング方式 (g-LST)，および，提案方式を比較した．グローバルスケジューリング方式は，オーバヘッドをゼロと考えると高い性能を達成することができる．しかし，実用を考えると，実際にはオーバヘッドの大小で性能は左右される．よって，グローバルスケジューリング方式のオーバヘッドを考慮した上で，性能評価を行った．

4.6.1　オーバヘッドの定義

評価に用いる g-EDF，g-LST の概要を図 4.2 に示す．本研究では，スケジューリングオーバヘッドのうち，スケジューラのオーバヘッドを考慮した．図 4.2 内の $T_{aggregation}$ は，スケジューラが各コアから実行状態を集約する際のオーバヘッド，$T_{distribution}$ は，スケジューラから各コアにタスク実行を指示する際のオーバヘッド，$T_{scheduler}$ は，スケジューラがスケジューリングアルゴリズムを実行し，実行タスクを決定する際のオーバヘッドである．EDF では，外部からタスク実行が要求されたとき，コアでタスク実行が終了したときに，各タスクの優先度が変化する可能性がある．LST では，加えて，タスク実行中に余裕時間が変動すると，各タスクの優先度が変化する可能性がある．よって，各タスクの優先度の変化により，タスクの切り替えが必要な場合は，

$$overhead \ = \ T_{aggregation} \ + \ T_{distribution} \ + \ T_{scheduler} \tag{4.6}$$

外部からタスク実行が要求される，あるいはコアで実行中のタスクが終了したときに，タスクの切り替えが必要ない場合は，

$$overhead \ = \ T_{aggregation} \ + \ T_{scheduler} \tag{4.7}$$

として，$overhead$ の時間分，コアでのタスク実行を中断する．

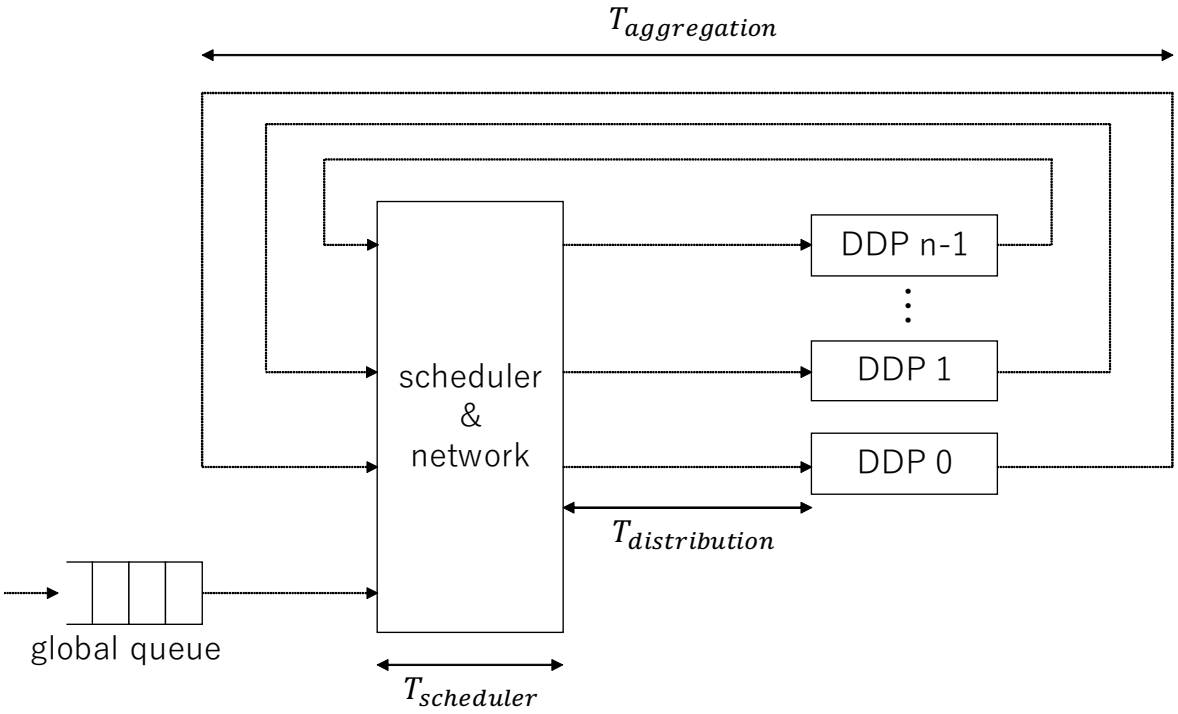


図 4.2　g-EDF，g-LST の構成

p-LST では，タスクの割り当ては，タスク実行前に静的に行われるため，オーバヘッドはないものとする．また，提案方式では，タスク入力時にのみ，ネットワーク内のスケジューラを通過する必要がある．ネットワーク通過による遅延時間は ns オーダーであるのに対し，タスクの実行時間は ms オーダーであるため，提案方式では，スケジューラのオーバヘッドは無視できるものとする．

4.6.2　評価条件

スケジューリング性能評価のための各パラメータは以下のように設定した．

- コア数：2, 4, 8, 16, 32, 64, 128
- DDP コアの多重度：2
- 各スイッチの履歴テーブル数：4
- 評価タスク数：周期性のあるタスクセット，ランダムなタスクセット，各 100 セット
- システム稼働率：60%

- $T_{aggregation}$，$T_{distribution}$：$\log_2$N ＊ weight
- $T_{scheduler}$：1

$T_{aggregation}$，$T_{distribution}$，$T_{scheduler}$ における，1 は，DDP の 1 命令実行時間 (250ns) に相当する時間と定義する．ネットワークは，多段相互結合網を前提としているため，$T_{aggregation}$，$T_{distribution}$ のオーバヘッドは，$\log_2$N とした．weight は，ネットワーク遅延の重みであり，ネットワーク遅延の増加によるスケジューリング性能への影響を評価するために用いる．weight は，1 – 30 の範囲の 1 刻みの値，および，0.1 – 1 の範囲の 0.1 刻みの値とする．また，各コアの稼働率は，コアの多重度が 2 であることから，式 4.8 で計算する．

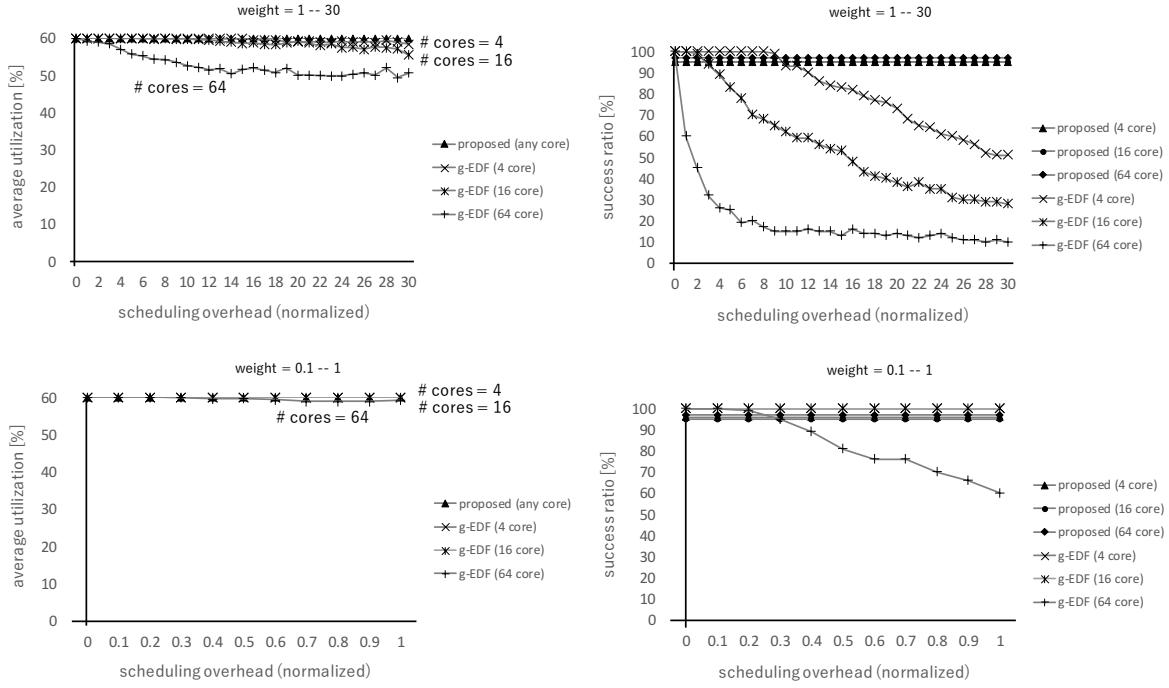
稼働率 ＝ (多重度 1 で実行中の稼働率 × 0.5) ＋ 多重度 2 で実行中の稼働率　(4.8)

多重度 2 で実行中は，別のタスクを多重に実行することはできない．しかし，多重度 1 で実行中は，別のタスクを受け入れることができるため，多重度 1 で実行中の稼働率を 0.5 倍する．以上の条件のもと，p-LST，g-EDF，g-LST，提案方式のスケジューリング性能評価として，平均システム稼働率，スケジューリング成功率を比較する．

4.6.3　評価結果

p-LST，g-EDF，g-LST，提案方式の平均システム稼働率，スケジューリング成功率を評価した．p-LST はすべてのコア数，タスクセットにおいて，スケジューリング成功率が 100%となった．タスク実行前の静的なタスク割り当てには，空き部分の大きなコアにタスクを割り当てる Worst-Fit アルゴリズムを用いた．すべてのタスクの実行状態があらかじめ静的に分かっている場合は，近似アルゴリズムを用いれば，最適解に近いタスク割り当てができると考えられる．しかし，p-LST は静的な方式であるため，実用的には使えない．よって，g-EDF との比較結果を図 4.3，図 4.4，g-LST との比較結果を図 4.5，図 4.6 に示す．各グラフには，4 コア，16 コア，64 コアの場合の値を示している．

g-EDF，g-LST はコア数が増加するほど，オーバヘッドが大きくなる．よって，64 コアの場合，g-EDF では，周期性のあるタスクセットは weight = 0.6 で，ランダムなタスク



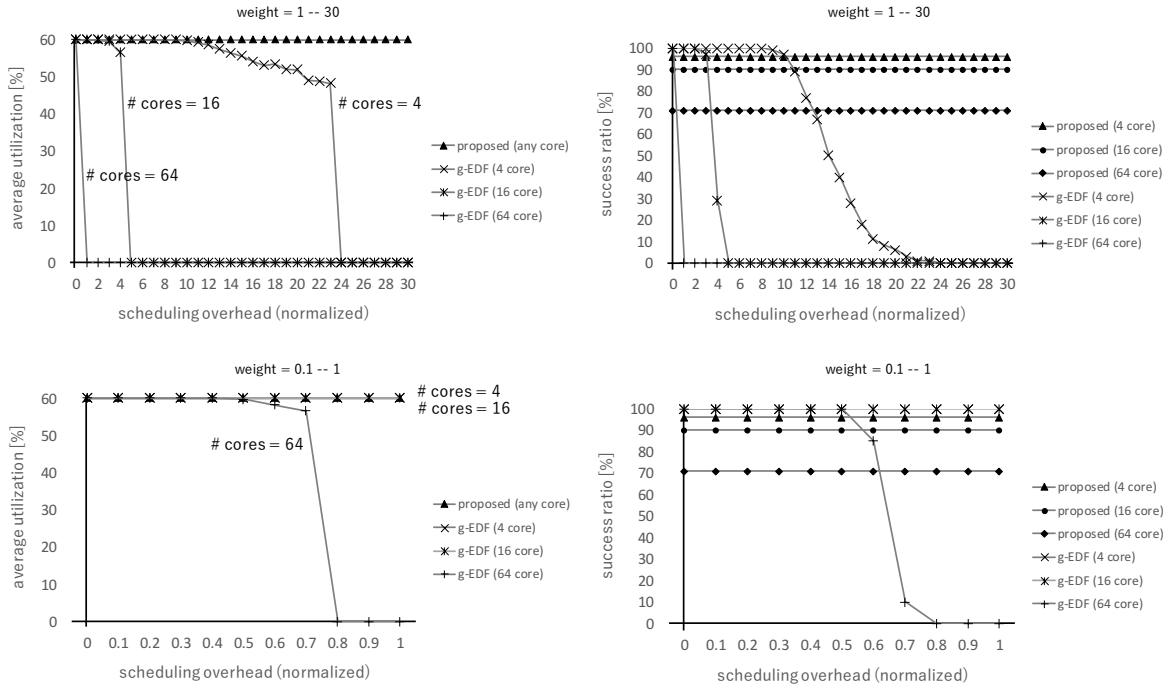
[1] 平均システム稼働率　　[2] スケジューリング成功率

図 4.3　提案方式と g-EDF の比較 (周期性のあるタスクセット)

セットは weight = 0.7 で，g-LST では，周期性のあるタスクセット，ランダムなタスクセットともに weight = 0.1 で，平均システム稼働率，スケジューリング成功率ともに，提案方式が g-EDF，g-LST に比べて高い結果となった．g-EDF，g-LST では，オーバヘッドが大きくなるほど，ハイパーピリオド内でのスケジューラの実行時間が増加するため，平均システム稼働率が低下する結果となった．提案方式は，オーバヘッドの影響がないため，平均稼働率は一定の値となった．また，2.3 節の議論から，LST は EDF に比べ，オーバヘッドをゼロと考えると，スケジューリング性能が高い．しかし，LST は余裕時間の変動によってタスク切り替えが頻繁に発生するため，同一の weight では，LST は EDF に比べ，平均システム稼働率，スケジューリング成功率が低くなった．

周期性のあるタスクセットは，ランダムに生成したベースタスクセットを複製して作成しているため，タスクセット内に含まれるタスクの種類は，ランダムなタスクセットに比べて少ない．そのため，周期性のあるタスクセットはランダムなタスクセットに比べ，タスクセット内に稼働率の小さなタスクが含まれる割合が大きくなる可能性が高い．稼働率の小さ

4.6 スケジューリング性能評価



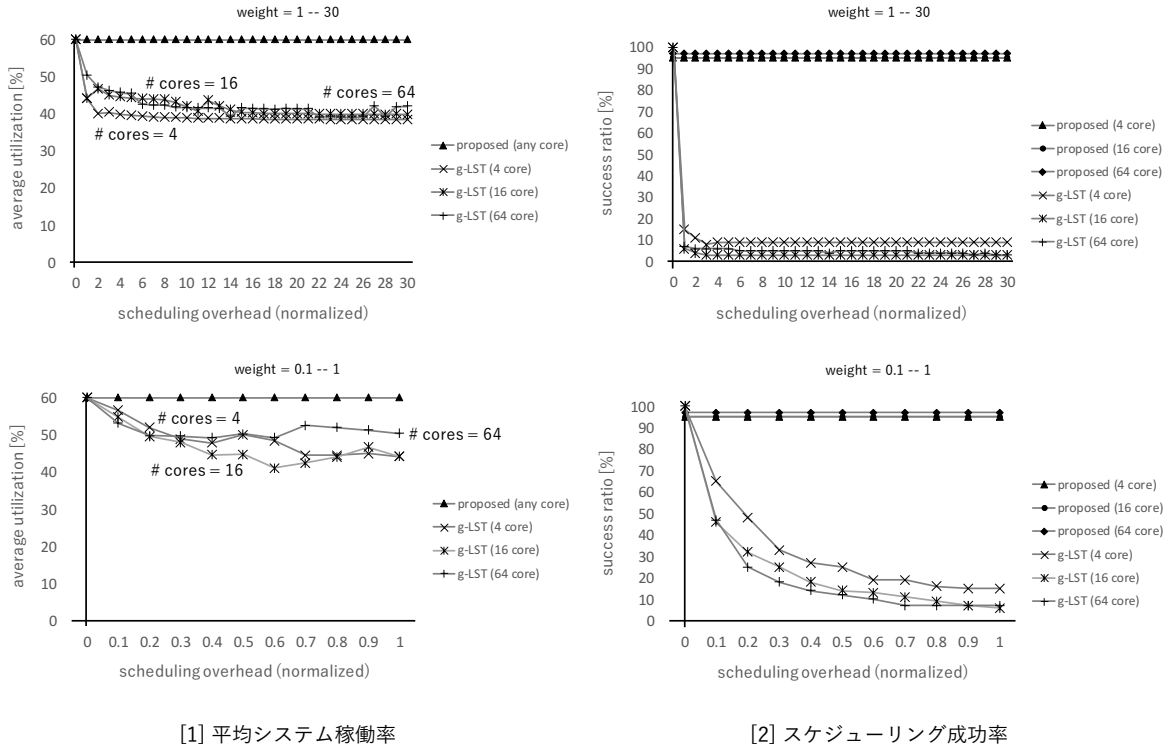
[1] 平均システム稼働率　　[2] スケジューリング成功率

図 4.4　提案方式と g-EDF の比較 (ランダムなタスクセット)

なタスクは稼働率の大きなタスクに比べ，オーバヘッドの影響により，タスク実行が遅れた場合に，デッドラインミスを起こす可能性が低い．よって，周期性のあるタスクセットは，ランダムなタスクセットに比べ，スケジューリング成功率が高くなったと考えられる．ランダムなタスクセットでは，スケジューリング成功率が 0%となった場合，すべてのタスクセットがデッドラインミスを起こしているため，平均システム稼働率が 0%になった．

また，提案方式では，周期性のあるタスクセットはタスクの種類が少ないため，タスクセットに含まれるタスクのうち，履歴テーブルで保持できるタスクの割合が大きくなる．よって，各コアの稼働率の偏りが小さくなり，ランダムなタスクセットに比べてスケジューリング成功率が高くなった．

平均システム稼働率，スケジューリング成功率の観点から，オーバヘッドを考慮した場合，コア数が多くなると，提案方式は g-EDF，g-LST に比べて，スケジューリング性能が高いことを確認できた．実用的なシステムを対象とした場合に，今回設定したオーバヘッドが妥当かどうかは今後検討が必要である．



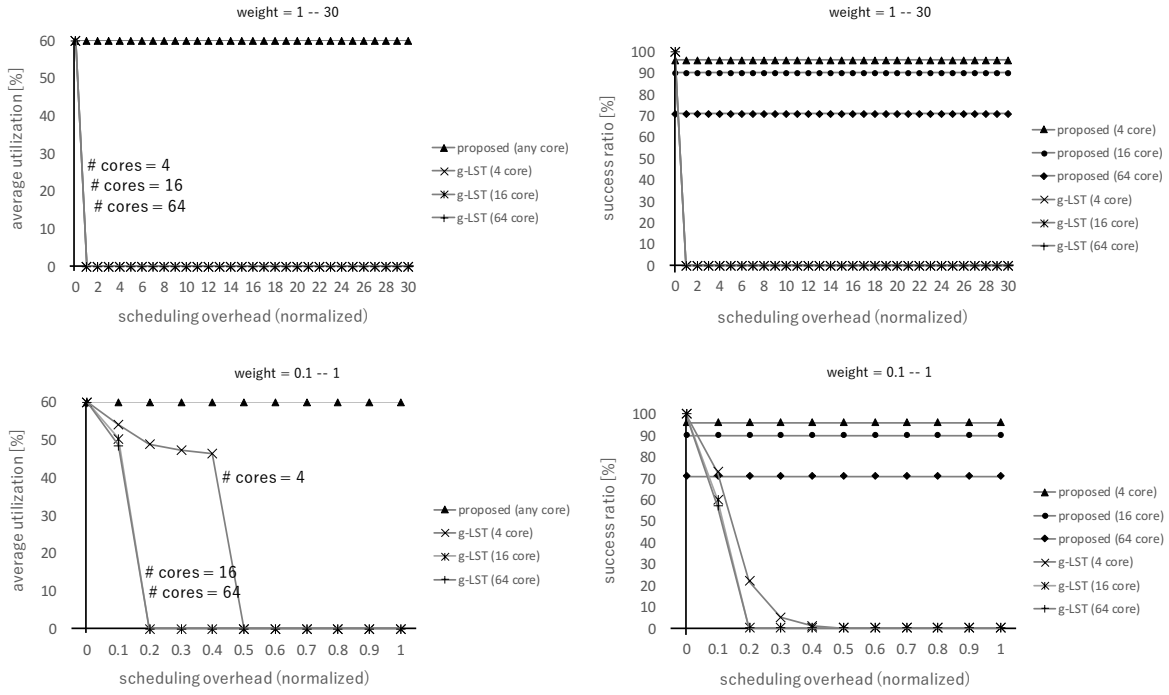
[1] 平均システム稼働率　　[2] スケジューリング成功率

図 4.5　提案方式と g-LST の比較 (周期性のあるタスクセット)

4.7　結言

本章では，提案方式の評価を行うために，65nm CMOS 標準セルライブラリを用いて Verilog-HDL による回路設計を行い，スイッチとシングルコア DDP の性能パラメータを抽出した．また，論理合成結果をもとに，論理合成後の DDP の稼働率を 0.2%の誤差で見積もり可能な評価用アーキテクチャシミュレータを作成した．

一次的な評価として，ネットワーク内スイッチ用スケジューラのアルゴリズムである，ラウンドロビン，余裕時間ベース，タスク履歴ベースの性能比較を行った結果，タスク履歴ベースのスケジューリング成功率が最も高かった．また，実際のシステムにおいて発生するスケジューリングオーバヘッドを考慮した性能を評価するために，g-EDF，g-LST と提案方式の平均システム稼働率と，スケジューリング成功率を比較した．64 コア以上の場合，g-EDF では，周期性のあるタスクセットは weight = 0.6 で，ランダムなタスクセットは weight = 0.7 で，g-LST では，周期性のあるタスクセット，ランダムなタスクセットと



[1] 平均システム稼働率　　[2] スケジューリング成功率

図 4.6　提案方式と g-LST の比較 (ランダムなタスクセット)

もに weight = 0.1 で，平均システム稼働率，スケジューリング成功率ともに，提案方式が g-EDF，g-LST に比べて高い結果となった．よって，今回設定したオーバヘッドの範囲内では，コア数が増加するほど，提案方式が g-EDF，g-LST に比べ，スケジューリング性能が高いことを確認できた．

第 5 章

結論

近年の IoT デバイスは，増加の一途を辿っており，2020 年には 403 億個にまで達すると予測されている．IoT デバイスは，様々な分野で活用が広がり，ますます，高機能化，高性能化が進んでいる．これまでは，シングルコアでの性能向上が図られてきたが，消費電力や発熱の面からシングルコアでの性能向上には限界があり，マルチコア化が進んでいる．特に，ミッションクリティカルシステムでは，リアルタイム性が要求されるため，将来の IoT デバイスでは，リアルタイム処理機能を備えたマルチプロセッサが必要となる．

マルチコア上でリアルタイム処理を実現するためには，スケジューリングアルゴリズムに従って，タスクをコアに割り当てる必要がある．タスクをコアに割り当てる方式として，パーティショニングスケジューリング方式とグローバルスケジューリング方式がある．パーティショニングスケジューリング方式は，タスク実行前に静的にタスク割り当てを決定するため，動的な負荷変動には対応できない．一方，グローバルスケジューリング方式は，コアやタスクの実行状態に応じて動的にタスク割り当てを決定するため，スケジューリングオーバヘッドをゼロと考えると，高い性能を達成することができる．しかし，実際にはスケジューリングオーバヘッドの大小で性能が左右される．また，タスクやコアの実行状態を集中的に集約/分析するとオーバヘッドが大きく，スケーラビリティが犠牲になる可能性があるため，マルチコアにおける性能向上には分散スケジューリングが必須となる．

本研究では，複数のタスクを多重に処理可能なデータ駆動型プロセッサ DDP に着目し，ネットワーク内のスイッチ用スケジューラと DDP コア内のスケジューラを統合した分散スケジューリング機構を検討した．具体的には，ローカル情報に基づき，自律的に宛先を決定可能なスイッチ用スケジューラのアルゴリズムの検討，および，LST (Least slack time) 機

構を搭載した DDP のアーキテクチャの検討を行った．

本稿において，第 2 章では，まず，リアルタイムシステムにおけるタスクのパラメータと分類をまとめた．そして，動的優先度方式のスケジューリングアルゴリズムである，EDF，LST を比較し，マルチコアでは，デッドラインミス予測性の観点から LST が有用であることを述べた．また，マルチコアにおける動的なタスク割り当て方式であるグローバルスケジューリング方式は，実用的には，マイグレーション，プリエンプション，スケジューラによるオーバヘッドが大きく性能が低下する問題がある．そこで，ネットワークとコアが自律分散的に動作可能な，分散スケジューリング機構の要件と設計方針について議論した．

第 3 章では，第 2 章で議論した方針をもとに設計した， 分散スケジューリング機構の構成について述べた．ネットワークの各スイッチは，コアに直接接続されるスイッチと，コアに直接接続されていないスイッチに分類できる．コアに直接接続されるスイッチは，過去のタスク割り当てからコアの状態を判断できることから，余裕時間ベースのアルゴリズムを採用した．コアに直接接続されないスイッチは，コアの状態を精密に予測できないため，過去の通過タスクの情報から宛先を判断する方針を取り，ラウンドロビン，稼働率ベース，タスク履歴ベースの 3 つのアルゴリズムを検討した．また，コアに用いる LST 機構を搭載した DDP のアーキテクチャについて述べた．

第 4 章では，第 3 章で設計した提案方式のスケジューリング性能評価について述べた．65nm CMOS 標準セルライブラリを用いてスイッチとシングルコア DDP の回路設計を行い，性能パラメータを抽出後，論理合成後の DDP の稼働率を 0.2%の誤差で見積もり可能なアーキテクチャシミュレータを作成した．一次的な評価として，ラウンドロビン，稼働率ベース，タスク履歴ベースの性能比較を行なった結果，タスク履歴ベースのスケジューリング成功率が最も高かった．また，スケジューリングオーバヘッドを考慮した性能評価のために，EDF を採用したグローバルスケジューリング方式 (g-EDF) ，LST を採用したグローバルスケジューリング方式 (g-LST) と提案方式の平均システム稼働率，スケジューリング成功率を比較した．その結果，コア数が多くなると，オーバヘッドを考慮した場合は，提案方式が g-EDF，g-LST に比べてスケジューリング性能が高いことを確認できた．

以下に本研究の今後の課題を示す．

- 様々なタスクセット，パラメータによる詳細な評価

  システム稼働率，タスクの稼働率を変更した場合のタスクセット，ソフトリアルタイムタスクから構成されるタスクセット，非周期タスクを含んだタスクセットによる，詳細なスケジューリング性能の評価が必要である．また，DDP コアの多重度，スケジューラのオーバヘッド，履歴テーブル数などのパラメータを変更した場合の評価，応答時間性能などの今回の評価とは別の指標による詳細な性能評価が必要である．

- 動的な負荷変動を考慮した性能評価

  分岐や繰り返し，メモリアクセスを含んだプログラムを用いて，動的な負荷変動が発生した場合の詳細な評価が必要である．

- スケジューリング成功率の向上

  提案方式は，システム稼働率 60%においてスケジューリング成功率が 100%に達していない．よって，オーバヘッドが小さい，コアやタスクの実行状態のフィードバック方法の検討，マイグレーションありを前提とした場合の分散スケジューリング機構の検討によるスケジューリング成功率の向上が必要である．

- DDP コア内スケジューラへの Fluid scheduling の適用

  マルチコアにおける最適なスケジューリングアルゴリズムである，Pfair[23][24] や LLREF[25][26] などの Fluid scheduling アルゴリズムは，スケジューリングのオーバヘッドが大きいため，実用的ではない．しかし，DDP は複数のタスクを多重に処理可能であり，タスク切り替え時にコンテキストスイッチが発生しないため，DDP コア内のスケジューラに Fluid scheduling アルゴリズムを適用することで，性能向上につながる可能性がある．

- 余裕時間ベーススイッチの回路設計

  余裕時間ベーススイッチの回路設計を行い，分散スケジューリング機構全体の回路面積を評価する必要がある．

- 提案回路の配置配線

  今回の評価は，論理合成の結果をもとに行ったため，配線遅延を含めたより厳密な評価が必要である．

今後，以上の課題を解決することで，提案方式の性能を厳密に評価し，さらなる性能向上を図る必要がある．また，IoT システムでは，面積や性能以外にも消費電力を小さくすることが重要である．よって，DVS (Dynamic Voltage Scaling) やパワーゲーティングなどの低消費電力技術を導入したスケジューリング [27][28] を検討することで，低消費電力化，高性能化が求められるマルチコアシステムへの応用が期待できる．

謝辞

本研究を進めるにあたり，日頃より懇切丁寧にご指導，ご鞭撻を賜りました岩田誠教授に心より深謝申し上げます．お忙しいにも関わらず，親身になって指導していただいたおかげで，本論文を完成させることができました．また，研究活動のみならず，様々な場面で大変お世話になりました．ここに感謝の意を表します．

本研究の副査をお引き受けくださり，様々な疑問点や改善点などを指摘していただいた，横山和俊教授，鵜川始陽准教授に心より感謝いたします．

研究室の同期として，日頃からご支援，ご協力いただいた，齋藤あかね氏，田原匡浩氏に心より感謝いたします．

研究室の後輩として，日頃からご支援，ご協力いただいた，修士 1 年の和田悠伸氏，学部 4 年の楠田健太氏，汐見興明氏，下出千晴氏，長野寛司氏，柳田海志氏，学部 3 年の井上聡氏，西岡周真氏に心より感謝いたします．

最後になりましたが，日頃からご支援いただいた関係者の皆様に心より御礼申し上げます．

参考文献

[1] 総務省, “平成 30 年度 情報通信白書 第 1 節 世界と日本の ICT 市場の動向,” http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h30/pdf/30honpen.pdf, 2019 年 1 月 26 日参照.

[2] Li Da Xu, Wu He, and Shancang Li, “Internet of Things in Industries: A Survey,” IEEE Transactions on Industrial Informatics, Vol. 10, No. 4, pp. 2233–2243, Feb. 2014.

[3] Sergio Saez, Joan Vila, Alfons Crespo, and Angel Garcia, “A Hardware Scheduler for Complex Real-Time Systems,” Proceedings of the IEEE International Symposium on Industrial Electronics, pp. 43–48, July 1999.

[4] Pramote Kuacharoen, Mohamed A. Shalan, and Vincent J. Mooney III, “A Configurable Hardware Scheduler for Real-Time Systems,” Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, pp. 96–101, June 2003.

[5] Yi Tang and Neil W. Bergmann, “A Hardware Scheduler Based on Task Queues for FPGA-Based Embedded Real-Time Systems,” IEEE Transactions on Computers, Vol. 64, No. 5, pp. 1254–1267, May 2015.

[6] Hiroaki Terada, Souichi Miyata, and Makoto Iwata,“DDMP’s: Self-Timed Super-Pipelined Data-Driven Multimedia Processors,” Proceedings of the IEEE, Vol. 87, No. 2, pp. 282–296, Feb. 1999.

[7] Kazuma Fukuda, Hiroki Shibuta, and Makoto Iwata, “Priority-Based Hardware Scheduler for Self-Timed Data-Driven Processor,” Proceedings of the 2017 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’17), pp. 245–251, July 2017.

[8] Yushin Wada, Kazuma Fukuda, and Makoto Iwata, “Least Slack Time Hardware Scheduler Based on Self-Timed Data-Driven Processor,” Proceedings of the 2018 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’18), pp. 249–255, July 2018.

[9] Mehrin Rouhifar and Reza Ravanmehr, “A Survey on Scheduling Approaches for Hard Real-Time Systems, ” International Journal of Computer Applications, Vol. 131, No. 17, Dec. 2015.

[10] Amit Kumar Singh, Piotr Dziurzanski, Hashan Roshantha Mendis, and Leandro Soares Indrusiak, “A Survey and Comparative Study of Hard and Soft Real-Time Dynamic Resource Allocation Strategies for Multi-/Many-Core Systems,” ACM Computing Surveys, Vol. 50, No. 2, Article 24, pp. 1–40, Apr. 2017.

[11] Kazuma Fukuda, Yushin Wada, and Makoto Iwata, “Decentralized Hardware Scheduler for Self-Timed Data-Driven Multiprocessor,” Proceedings of the 2018 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’18), pp. 256–261, July 2018.

[12] Giorgio C. Buttazzo, “Hard Real-Rime Computing Systems: Predictable Scheduling Algorithms and Applications - Third Edition,” Springer, 2011.

[13] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” Journal of the ACM, Vol. 20, No. 1, pp. 46–61, Jan. 1973.

[14] J. Hildebrandt, F. Golatowski, and D. Timmermann, “Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems,” Proceedings of the 11th Euromicro Conference on Real-Time Systems, pp. 208–215, June 1999.

[15] Xuefeng Piao, Sangchul Han, Heeheon Kim, Minkyu Park, and Yookun Cho, “Predictability of Earliest Deadline Zero Laxity Algorithm for Multiprocessor Real-Time Systems,” Proceedings of the Ninth IEEE International Symposium on Object and

Component-Oriented Real-Time Distributed Computing (ISORC'06), pp. 359–364, Apr. 2006.

[16] Shinpei Kato and Nobuyuki Yamasaki, "Global EDF-based Scheduling with Efficient Priority Promotion," Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 197–206, Aug. 2008.

[17] Myuuggwon Hwang, Dongjin Choi, and Pankoo Kim, "Least Slack Time Rate First: an Efficient Scheduling Algorithm for Pervasive Computing Environment," Journal of Universal Computer Science, Vol. 17, No. 6, pp. 912–915, Mar. 2011.

[18] J. E. G. Coffman, M. R. Garey, and D. S. Johnson, "Approximation Algorithms for Bin Packing: A Survey," PWS Publishing, pp. 46–93, 1996.

[19] Dong-Ik Oh and T. P. Baker, "Utilization bounds for n-processor rate monotone scheduling with static processor assignment," Real-Time Systems, Vol. 15, No. 2, pp. 183–192, Sep. 1998.

[20] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge, "A Survey of Multicore Processors," IEEE Signal Processing Magazine, Vol. 26, No. 6, Nov. 2009.

[21] 水野正之, 鳥居淳, 松谷宏紀, 野瀬浩一, "10 群 3 編 4 章 IP 間接続," 電子情報通信学会 知識ベース 知識の森, 2010.

[22] 天野英晴, 吉永努, 鯉渕道紘, 横田隆史, 松谷宏紀, "6 群 5 編 9 章 インタコネクション技術," 電子情報通信学会 知識ベース 知識の森, 2010.

[23] S. K. Baruah, N. K. Cohen, C. G. Plaxton, D. A. Varvel, "Proportionate Progress: A Notion of Fairness in Resource Allocation," Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, pp. 345–354, May 1993.

[24] James H. Anderson and Anand Srinivasan, "Early-release fair scheduling," Proceedings of the 12th Euromicro Conference on Real-Time Systems, pp. 35–43, June 2000.

[25] Hyeonjoong Cho, Binoy Ravindran, E. Douglas Jensen, “An Optimal Real-Time Scheduling Algorithm for Multiprocessors,” IEEE International Real-Time Systems Symposium, Dec. 2006.

[26] Hyeonjoong Cho, Binoy Ravindran, E. Douglas Jensen, “Synchronization for an optimal real-time scheduling algorithm on multiprocessors,” IEEE International Symposium on Industrial Embedded Systems, pp. 9–16, July 2007.

[27] Kei Miyagi, Shuji Sannomiya, Makoto Iwata, and Hiroaki Nishikawa, “Low-Powered Self-Timed Pipeline with Runtime Fine-Grain Power Supply,” Proceedings of the 2012 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’12), pp. 472–478, July 2012.

[28] Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo, “Energy-Aware Scheduling for Real-Time System: A Survey,” ACM Transactions on Embedded Computing System, Vol. 15, No. 1, Article No. 7, pp. 1–34, Feb. 2016.