修 士 論 文

非対称型マルチプロセッサにおける動的分散リアルタイムスケジューリングに関する研究

北陸先端科学技術大学院大学
情報科学研究科 情報システム学専攻

石川 智久

2007 年 3 月

修 士 論 文

非対称型マルチプロセッサにおける動的分散リアルタイムスケジューリングに関する研究

指導教官　田中清史 助教授

審査委員主査　田中清史 助教授
審査委員　日比野靖 教授
審査委員　井口寧 助教授

北陸先端科学技術大学院大学
情報科学研究科 情報システム学専攻

510007 石川 智久

提出年月: 2007 年 2 月

概 要

近年，組み込みシステムにおいて高性能化と低消費電力化を目的としたマルチプロセッサ化が進んでいる．組み込みシステムでは実行する処理が限定的なため，特定の処理に特化したプロセッサと汎用的なプロセッサを組み合わせた機能分散型マルチプロセッサが利用される．しかしながら，機能分散型マルチプロセッサは異種プロセッサから構成されるため動的な負荷分散は困難であり，負荷の偏りが生じた場合，計算資源に余裕があるにもかかわらずデッドラインミスが発生する．

本研究では機能分散型マルチプロセッサにおいて，デッドラインミスが予測された場合に異種プロセッサ間でタスクを移動する方式を提案・実装する．シミュレータ上でデッドラインミス数を計測し，提案方式を評価した結果，負荷に偏りがある状況でデッドラインミス数および応答速度が改善されることが確認できた．

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　研究の背景

近年，携帯電話やデジタル家電，カーナビゲーションシステムなどのさまざまな組み込みシステム分野において，システムの大規模化・複雑化が顕著である．それに伴いプロセッサに要求される性能も高まっているが，単独のプロセッサで十分な性能を得ることが困難になりつつある．これは，プロセッサの消費電力を増やすことなく動作周波数を上げることの難しさに起因している．携帯電話や携帯型オーディオ機器のようにバッテリー動作を前提とする組み込みシステムにおいて低消費電力化は直接商品価値に繋がるため，安易に消費電力が高いプロセッサを使用することはできない．そこで，1つのチップ上に複数のプロセッサコアを集積したオンチップマルチプロセッサが注目されている [1]．プロセッサの動作周波数は電圧の2乗に比例するため，1つの高速なプロセッサを用いる構成よりも低速なプロセッサを複数用いるマルチプロセッサ構成にすることで，処理性能の向上とともにシステム全体の電圧を下げることができる．

PCやワークステーションなどの汎用システムにおけるオンチップマルチプロセッサは，汎用的な処理に対する高速化が目的である．高速化の方法として並列処理と負荷分散があるが，後者では依存関係がない処理を複数のプロセッサに分散させ，個々のプロセッサの処理負荷を減らす負荷分散を行うことで高速化が実現される．システムが実行されるまでどのような処理要求がどのタイミングで発生するかわからないため，各プロセッサコアが均質な対称型マルチプロセッサ構成を用い，処理要求が発生した時に実行プロセッサを決定する方式が一般的である．

一方，組み込みシステムにおいては，特定の処理要求に対する応答速度向上を目的としてオンチップマルチプロセッサが用いられる．組み込みシステムでは動作するアプリケーションが事前に決められていることが一般的であるため，ボトルネックとなる高負荷な処理に対しては専用ハードウェアを用意することで効率的に実行し，その他の小さな処理を汎用プロセッサで実行することが一般的であった．最近では機能の複合化・高機能化に伴う多様な処理要求に対応するため，専用ハードウェアの代わりに再利用性が高いDSPなどの特定処理に特化したプロセッサアーキテクチャと汎用プロセッサを組み合わせた機能分散型マルチプロセッサ構成 [1] の利用が増えている．

機能分散型マルチプロセッサ構成を利用した組み込みシステムの増加とともに，それを

[1]実際に製品化された組み込みシステム向けのオンチップマルチプロセッサとしてTexas Instruments社のOMAP，ARM社のMPCore，東芝のMeP，NECエレクトロニクスのMP211などが発表されている．

サポートするリアルタイム OS の重要性が高まっていることから，μITRON [2] の機能分散型マルチプロセッサ向け拡張仕様に基づいた TOPPERS/FDMP カーネル [3] が実装された．従来の機能分散型マルチプロセッサのソフトウェア開発は，各プロセッサにシングルプロセッサ用のリアルタイム OS を載せ，プロセッサ間の同期・通信はアプリケーションレベルで実現していた．TOPPERS/FDMP カーネルではプロセッサ間の同期・通信がリアルタイム OS の機能として実装されているため，ソフトウェアの開発工数が削減され，かつ移植性が向上し，機能分散型マルチプロセッサを利用した組み込みシステムの開発が容易になった．

1.2 研究の目的

機能分散型マルチプロセッサ構成は各プロセッサの命令セットアーキテクチャが異なり，実行可能なバイナリやタスク制御ブロックの構造が異なる．システム実行中にプロセッサの負荷に応じてタスクを動的に分散させることは困難であるため，各タスクはシステム設計者によって静的にプロセッサに割り当てられ，各タスクを実行するプロセッサは固定される．このため特定の処理要求が頻発した場合，タスクの処理要求が集中してデッドラインミスや応答速度の低下が発生するプロセッサが存在する一方で，ほぼタスクがない低負荷なプロセッサが存在する状態が起こる．これは，投入した計算資源を有効に利用していない状態であり，高負荷プロセッサから低負荷プロセッサに負荷を分散することで，デッドラインミス数の削減および応答速度の向上が期待できる．

本研究は，機能分散型マルチプロセッサ構成において動的に負荷分散を行う方式を提案する．あるプロセッサでデッドラインミスが予測された場合，タスクを共有メモリ上へ一時的に保存し，負荷が少ないプロセッサが共有メモリからデッドライン時刻までに実行可能なタスクを取得することでタスクの移動を実現する．また，異種命令セットアーキテクチャ間でタスクが移動するが，各プロセッサはあらかじめ全てのタスクの実行可能なバイナリを生成し，タスク制御ブロックとともにローカルメモリに保持することで，異種命令セットアーキテクチャ間のタスク移動が可能となる．

1.3 本論文の構成

本論文の構成を以下に示す．

2 章では，本研究に関連する分野として，リアルタイムシステムと機能分散型マルチプロセッサについて述べる．

3 章では，本研究で提案する機能分散型マルチプロセッサにおけるタスクの移動方式について述べる．

4 章では，提案方式の評価を行う．

5 章では，本研究のまとめを行う．

第2章　リアルタイムシステムと機能分散型マルチプロセッサ

本章では，リアルタイムシステムの説明を行い，本研究で使用するスケジューリングについて述べる．また，組み込みシステムへの適用の観点から機能分散型マルチプロセッサと対称型マルチプロセッサの比較を行う．

2.1　リアルタイムシステム

2.1.1　タスクの時間情報

本研究におけるタスクの時間情報を以下に定義する（図 2.1）．

- **起動要求時刻（a）**
  タスクの実行要求が発行され実行可能状態に遷移した時刻．
- **開始時刻（s）**
  タスクが最初に実行を開始した時刻．
- **終了時刻（f）**
  タスクの実行が完了した時刻．
- **実行時間（C）**
  プロセッサがそのタスクの実行に費やした時間．
- **応答時間（R）**
  タスクの起動要求が発生してからタスクが完了するまでの時間．$f - a$ によって求まる．
- **相対デッドライン時間（D）**
  タスクの実行要求が発行されてから絶対デッドライン時刻までの時間．各タスクごと固有の値を持つ．
- **絶対デッドライン時刻（d）**
  タスクの実行が完了していなければならない締め切り時刻．$a + D$ によって求めることができ，タスクの起動要求が発行された時に確定する．

- **余裕時間（L）**
  タスクの実行が完了した時の絶対デッドライン時刻までの残り時間．$d - f$ によって求めることができ，タスクがデッドライン時刻を超過した場合は負の値となる．

- **周期（T）**
  規則的に繰り返し起動要求が発生するタスクの起動間隔．

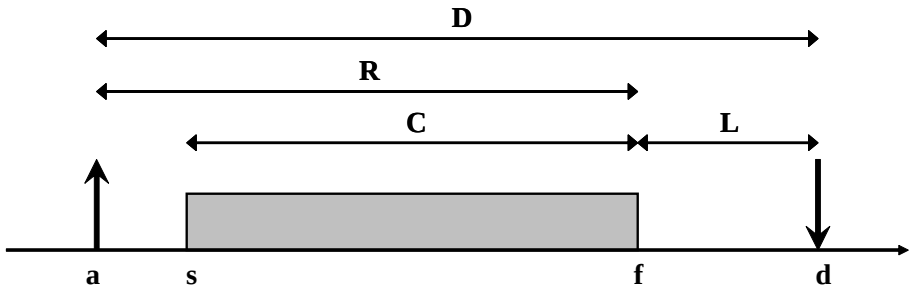


図 2.1: タスクの時間情報

一般に，タスクは起動の周期性によって周期タスクと非周期タスクに分類することができる[1]．周期タスクは一定の周期 T に従って規則的に起動するため，起動時刻を予測可能である．一方，非周期タスクは再発するが規則性がないので起動時刻を予測できない．

2.1.2 リアルタイム性の分類

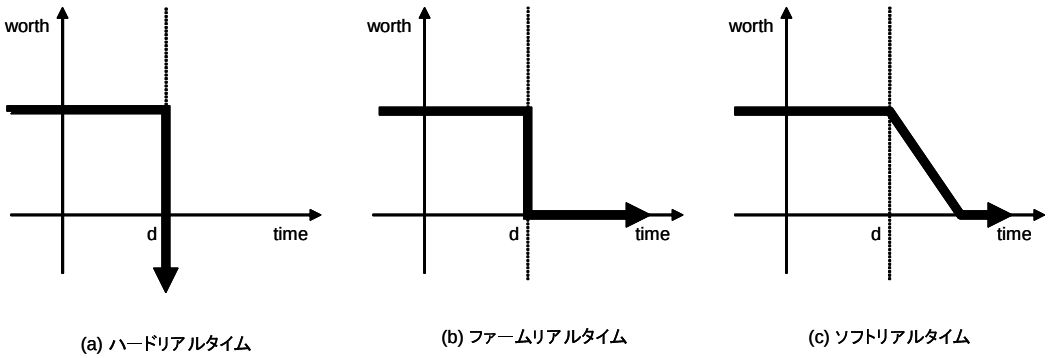
リアルタイム性には，主に以下の 3 種類がある．

- **ハードリアルタイム**
  ハードリアルタイム処理では，タスクは締め切り時刻を必ず守らなければならない．あるタスクがデッドラインミスを起こした場合，そのタスク（またはシステム）の価値はマイナス無限大になる（図 2.2a）．航空機や原子炉など，人命に関わる処理（Life Critical）や，システムに損害を与える可能性がある処理（Mission Critical）に適用される．

- **ファームリアルタイム**
  ファームリアルタイム処理は，あるタスクがデッドラインミスを起こした場合，そのタスクの価値はゼロになる（図 2.2b）．デッドラインを超過したタスクを実行する意味はなくなり，場合によってはシステムを停止する必要もあるが，その結果はシステムに損害を与えるほど致命的ではない．

[1]周期タスク，非周期タスクの他に偶発的タスクという分類が存在する．偶発的タスクは起動に規則性がない点において非周期タスクと同様だが，最低起動間隔が決まっている．本研究では偶発的タスクは非周期タスクに含める．

- **ソフトリアルタイム**
  ソフトリアルタイム処理は，高負荷状態の時などはデッドラインを多少過ぎることを許容される．タスクの価値は，デッドラインを過ぎると徐々に減少していくが（図2.2c），システムに損害を与えることはない．ソフトリアルタイムシステムの例として，マルチメディア処理などが挙げられる．



(a) ハードリアルタイム　(b) ファームリアルタイム　(c) ソフトリアルタイム

図 2.2: リアルタイム性の分類

本研究はソフトリアルタイムシステムを対象とし，デッドラインミスする可能性があるタスクを移動することでシステムの価値の低下を防ぐ．

2.1.3 最悪実行時間（WCET）

リアルタイムシステムに限らず，プログラムの実行時間は入力データなどの条件の違いによって変動する．しかし，タスクの実行時間がわからなければデッドラインを保障するスケジューリングは不可能なので，最長の実行時間の見積もりを行う．あらゆる条件でタスクを実行したとして，その中で最長の実行時間を最悪実行時間（Worst-Case Execuction Time: WCET）と呼ぶ．

2.1.4 タスク優先度

本研究におけるタスクの優先度は，値が小さいほど高優先度であることを意味する．また，タスク ID tid のタスクの優先度 P_{tid} は，各タスク間において以下の関係が成立するものとする（理由は 3.3.1 節で述べる）．

$$P_i \leq P_{i+1}$$

2.1.5 スケジューリングの分類

リアルタイムスケジューリングは性質や方法によって分類することができる．以下に主な分類方法を示す．

- **オフライン／オンライン**
  システム稼動前（例えばコンパイル時）にスケジューリングを済ませておく方法をオフライン方式という．事前にすべてのタスクについて起動要求時刻などが既知でなければならず，柔軟性に乏しい．しかし最適なスケジューリングを行うことが可能であり，実行時のオーバヘッドが小さいという特徴を持つ．一方，オンライン方式はシステム稼働中（例えばタスクの起動時と終了時）にスケジューリングを行う．この方式は最適なスケジューリング結果を得ることが難しく，また実行時のオーバヘッドが大きいという特徴を持つ．

- **静的／動的**
  タスクの優先度が静的か動的かによる分類である．静的優先度とは，一度決定したら変化しないタスク優先度のことである．それに対し，タスクの時間情報などに基づいて変化する可能性があるものを動的優先度という．

- **プリエンプティブ／ノンプリエンプティブ**
  タスク A の処理を実行中に，さらに高い優先度をもつタスク B が起動した場合に，タスク A を中断してタスク B の処理を開始することをプリエンプション（preemption）といい，プリエンプションの発生を許す性質をプリエンプティブという．逆に一度タスク A が開始されたら処理が完了するまで実行権を移動しないことをノンプリエンプションという．

組み込みシステム向け OS の仕様のひとつである μITRON におけるスケジューリングは，オンライン・静的・プリエンプティブに分類され，タスク間の優先順位に基づいてスケジューリングが行なわれる．タスク間の優先順位は，システム設計者が各タスクの重要度に合わせ事前に決めた優先度を基準として，以下のように決定される．

- 異なる優先度を持つタスク間の優先順位は，高い優先度を持つタスクの優先順位が高い．

- 等しい優先度を持つタスク間では First Come First Served（最初に来たものから処理される）方式がとられ，レディキューに追加された順にタスクの優先順位が高い．

本研究では，上記の静的優先度方式をノンプリエンプティブに変更したスケジューリングを行う．本スケジューリング方式におけるタスク実行権の遷移例を図 2.3 に示す．ある時刻においてタスク 2 が実行中であるとする．タスク 2 の実行中にタスク 4，タスク 1，タスク 5 の順で起動要求が発生する．タスク 1 はタスク 2 よりも優先度が高いが，タスク

2 の実行が終了するまで実行権は遷移しない（ノンプリエンプション）．タスク 2 の実行が終了すると，その時点の優先度が最も高いタスク 1 に実行権が遷移し，タスク 1 が実行される．タスク 1 の終了後は，優先度が最も高いタスクとしてタスク 4 とタスク 5 があるが，タスク 4 のほうが先に起動要求が発生してレディキューに追加されたため，タスク 4 が実行される．タスク 4 の終了後は，その時点の最高優先度を持つタスクであるタスク 5 が実行される．タスク 5 の終了後は，レディキューに実行タスクが存在しないためアイドル状態になる．その後，タスク 3 の起動要求発生し，タスク 3 の実行が開始される．

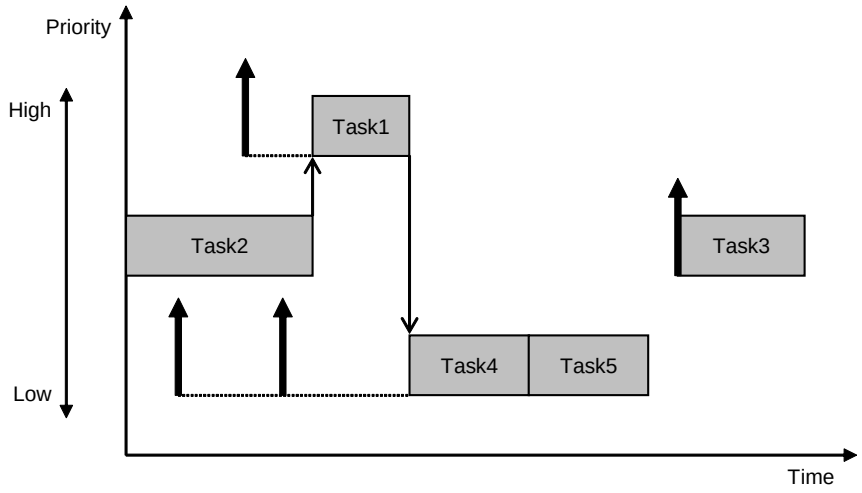


図 2.3: スケジューラの動作

2.2 リアルタイムシステムにおけるマルチプロセッサ

マルチプロセッサは以下のように分類することができる．

- 対称型マルチプロセッサ
- 機能分散型マルチプロセッサ

対称型マルチプロセッサでは，各プロセッサがすべての処理を等しく実行できる（図 2.4）．それに対して機能分散型マルチプロセッサでは，プロセッサごとに処理が限定される[2]（図 2.5）．

組み込みシステムでは事前に処理が決まっていることが一般的である．機能分散型マルチプロセッサでは，事前に判明している高負荷な処理に対して，特化したプロセッサや

[2]異なる種類のプロセッサを組み合わせて構築したマルチプロセッサシステムを非対称型マルチプロセッサと呼ぶが，各処理をどのプロセッサで実行するかを決めることになるため，機能分散型マルチプロセッサの一種と捉えることができる．

ハードウェアを用意することで，効率的に高負荷処理を実行することが可能である．したがって，1.1 節で述べたように，組み込みシステム分野では機能分散型マルチプロセッサ構成の利用が増えている．

しかしながら，機能分散型マルチプロセッサでは，プロセッサ間で負荷の偏りが生じた場合に負荷を分散することが困難である．本研究では，機能分散型マルチプロセッサにおいて，静的な負荷分散（システム設計者によるタスク割り当て）で対応できないケースに対して，動的な負荷分散を行う．

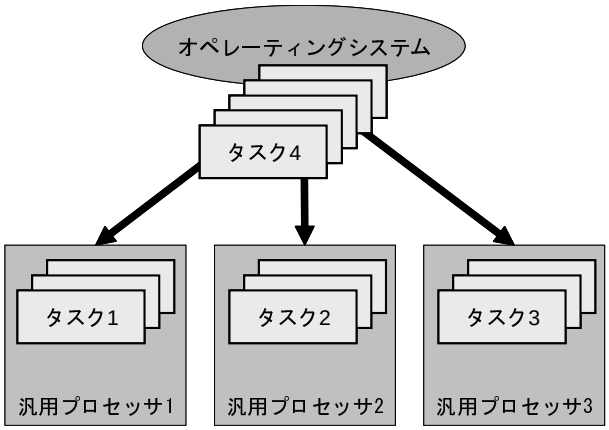


図 2.4: 対称型マルチプロセッサではタスクを実行するプロセッサは動的に決定する

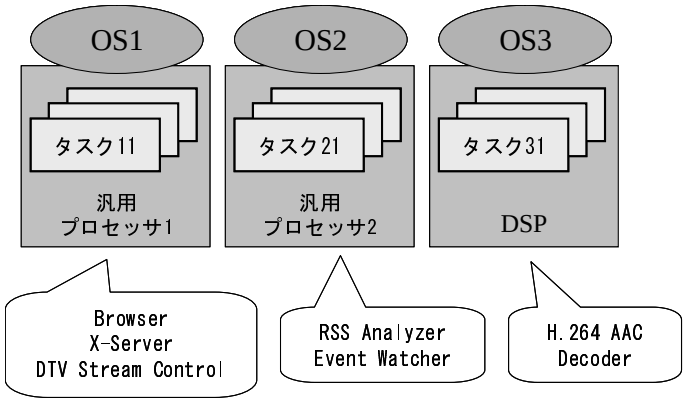


図 2.5: 機能分散型マルチプロセッサではタスクを実行するプロセッサは固定される

第3章　異種プロセッサ間におけるタスク移動方式

本章では，異種命令セットアーキテクチャ間におけるタスク移動を行うための方式とその実装方法について述べる．

3.1　命令セットアーキテクチャの違い

機能分散型マルチプロセッサは，各プロセッサの命令セットアーキテクチャ（Instruction Set Architechuer: ISA）が異なる．各 ISA ごとに実行可能なバイナリおよびタスク制御ブロック（Task Control Block: TCB）が異なるため，各プロセッサは静的に割り当てられたタスクのバイナリおよび TCB のみを保持する（図 3.1）．よって，通常の方法では静的に割り当てられたタスク以外を実行することはできない．

本方式では，各プロセッサはすべてのタスクに対して実行可能なバイナリを事前に作成し，TCB とともにメモリに保持することで，他プロセッサに割り当てられたタスクも同様に実行可能となる（図 3.2）．

TCB はタスクの起動要求発生時に初期化され，タスクの実行に必要な情報を保持する．タスクが中断した場合，プログラムカウンタやスタックポインタなどは TCB に格納され，タスクを再開する場合は TCB が保持する情報を利用する．プロセッサ間タスク移動において実行途中のタスクを移動先で再開するためには，TCB の内容も同様に移動する必要がある．TCB は ISA に依存するため，ISA が異なるプロセッサ間で移動が生じた場合は移動先の ISA に対応した TCB に変換する必要があるが，困難である．したがって，本方式では未実行状態のタスクに限定してタスク移動を行う．

3.2　タスクの移動方法

各プロセッサはスケジューラ実行時に負荷計算を行い，プロセッサの負荷状態を高負荷，通常，低負荷の 3 段階で判断する．高負荷プロセッサから低負荷プロセッサへ移動する際は，移動対象となるタスクの ID などを通信する必要がある．したがって，各プロセッサからアクセス可能な共有タスクプールを用意し，タスク移動に必要な情報は共有タスクプールを介して通信する（図 3.3）．

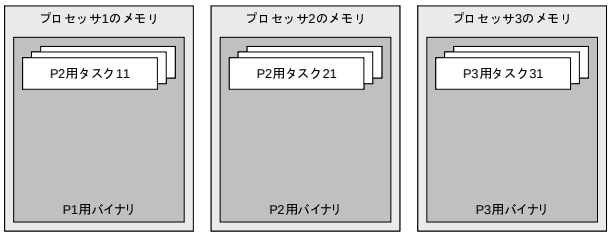


図 3.1: 各プロセッサは静的に割り当てられたタスクのバイナリを保持

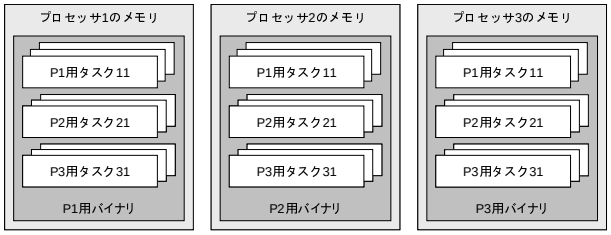


図 3.2: 各プロセッサは全タスクのバイナリを保持

以下に各状態におけるプロセッサの動作を示す.

- **高負荷状態**
  高負荷プロセッサは自身のレディキューから移動候補となるタスクを削除し，移動候補タスクの情報を共有タスクプールに書き込む.

- **通常状態**
  通常状態のプロセッサはタスクの移動・受け入れは行わず通常のスケジューリングを行う．機能分散型マルチプロセッサは，システム設計者による静的なタスク分配の時点で負荷分散がある程度行われている．よって，負荷の偏りが生じない場合はタスク移動は行わない.

- **低負荷状態**
  低負荷プロセッサは共有タスクプールから移動候補タスクを受け入れる．ただし，タスクを受け入れたことによるデッドラインミスの発生を避けるため，受け入れるタスクはスケジューリング可能なタスクに限定される．移動候補タスクを受け入れた場合は移動候補タスクを共有タスクプールから削除し，自身のレディキューに登録する.

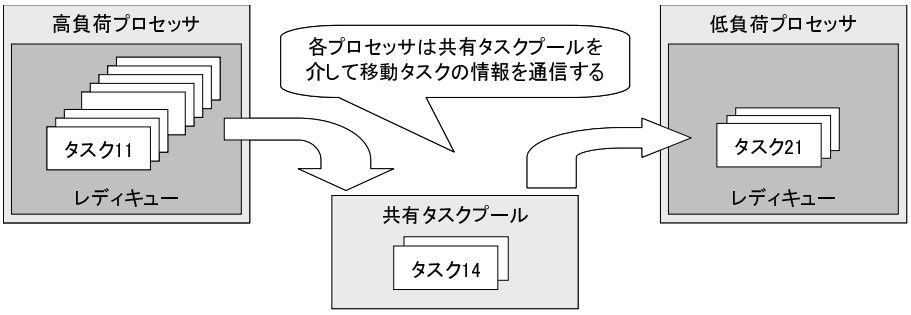


図 3.3: 共有タスクプールを使用したタスク移動

3.3 提案方式の実装方法

本節では本方式の実装例を示す.

機能分散型マルチプロセッサを構成する ISA として，以下を用いた.

- 汎用コア： MIPS32 ISA [4]
- 専用コア： Texas Instruments 社 TMS320C54x DSP ISA [5]

各プロセッサはそれぞれ独立したローカルメモリの他に，各プロセッサからアクセス可能な共有メモリに接続されている（図 3.4）．共有メモリの特徴としてローカルメモリに比べアクセスレイテンシが大きいことが挙げられる.

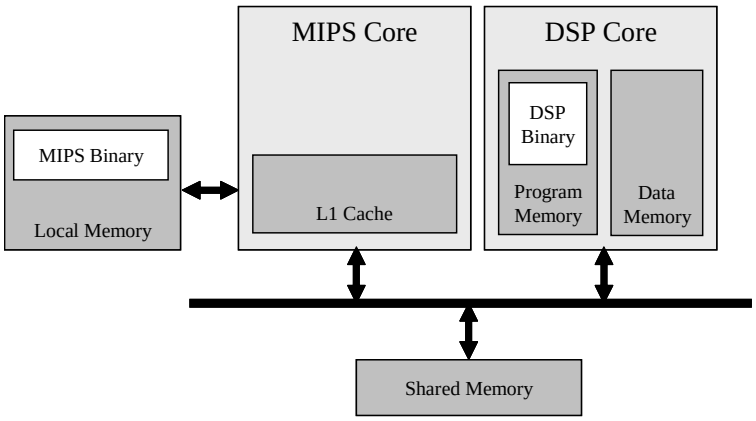


図 3.4: 本実装における機能分散方マルチプロセッサの構成

3.3.1　共有メモリの構造

共有メモリのアドレス空間

以下に MIPS および DSP のメモリアドレスマップを示す（図 3.5）.

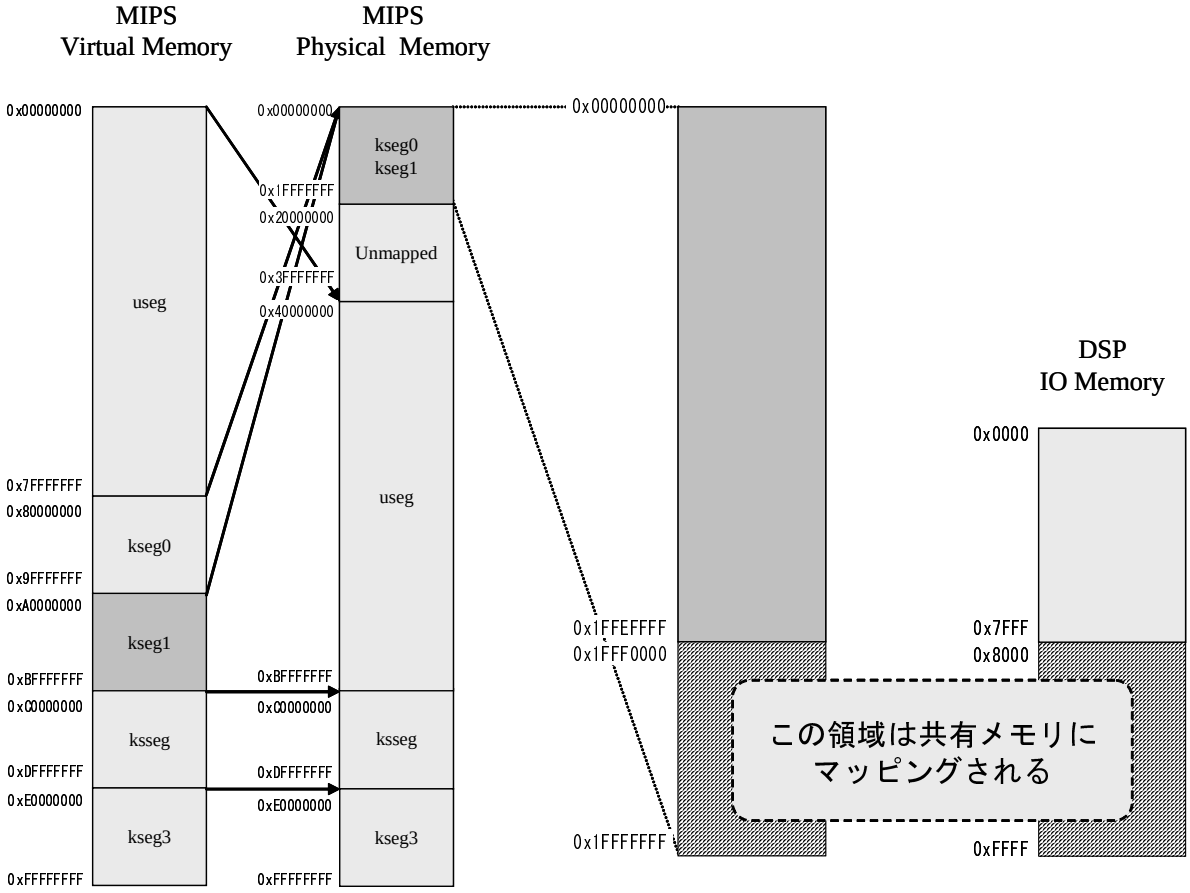


図 3.5: MISP,DSP のメモリマップ

MIPS の仮想メモリアドレスの 0xBFFF0000～0xBFFFFFFF と DSP の IO メモリアドレス[1]の 0x8000～0xFFFF は同一の共有メモリにマッピングされており，両プロセッサ間で通信を行う場合はこのアドレス領域を使用する．MIPS は L1 キャッシュを持つが，0xBFFF0000～0xBFFFFFFF は kseg1 領域[2]に含まれるため，キャッシュを介さずメモリへのアクセスが可能である．

[1]DSP はプログラムを保持するメモリのアドレス空間とデータを保持するメモリのアドレス空間が独立している．また，外部デバイスや外部メモリへアクセスするために 16bit の IO メモリアドレス空間を持つ．本実装では共有メモリを IO メモリ空間にマッピングした．また，DSP の共有メモリは見かけ上 MIPS の共有メモリの半分の 64KB だが，DSP の 1Byte は 16bit なので実質的には MIPS と同様のサイズを持つ．

[2]kseg1 はカーネルモードで使用される領域であり，物理メモリアドレス 0x00000000～0x1FFFFFFF にマップされ（図 3.5），キャッシュスルーでアクセスされる．

共有メモリ空間は，排他制御のためのロック変数と共有タスクプールに使用する．プロセッサ間タスク移動を行う時に，共有タスクプールを用いてプロセッサ間で移動するタスクの情報を通信する．

共有タスクプールのデータ構造

共有タスクプールはプロセッサ間タスク移動の際に，タスク移動に必要な情報を保持する領域である．タスク移動に必要な情報を以下に示す．

- **タスク ID**
  移動先プロセッサにおいて，タスクを起動するために必要となる．各プロセッサはすべてのタスクの TCB を保持しているため，タスク ID がわかれば移動タスクを実行可能である．

- **絶対デッドライン時刻**
  絶対デッドライン時刻はタスク起動時に決定する．移動タスクは移動元プロセッサにおいてすでに起動されており，移動先プロセッサにおいても移動元プロセッサで決まったデッドライン時刻までにタスクを完了する必要がある．

本実装では，タスク情報の書き込み先アドレスはタスク ID ごとに固定する．これにより，アドレスからタスク ID を求めることができるため，共有メモリへタスク ID を書き込む必要がなくなる．共有メモリへのアクセスはコストが大きいため，アクセス回数の削減はオーバヘッドの削減に繋がる．

加えて，タスク ID が小さいタスクほど静的優先度が高くなるようにタスク ID をつけることで，受け入れタスクの選択の際に，タスクの優先度の判断が容易になる．すなわち，最小のタスク ID が保持されるアドレスから優先して受け入れることで，静的優先度順のタスク受け入れが可能となる．

3.3.2 負荷計算

高負荷状態のプロセッサから低負荷状態のプロセッサへタスクの移動を行うために，各プロセッサの負荷状態を高負荷，通常，低負荷の 3 段階に分類する．以下に分類方法を示す．なお，N はパラメータとして与えられる定数である．

1. 各プロセッサのレディキューにおいて，優先順位が N 以内のタスクに対してデッドラインミス予測（後述）を行い，デッドラインミスを予測されたタスクが存在する場合は高負荷状態とする．
2. 1 に該当せず，レディキュー上のタスク数が N 以下の場合は低負荷状態とする．
3. 1, 2 の両方に該当しない場合は通常状態であるとする．

スケジューラは実行されるごとに負荷計算を行い，負荷に応じてタスクの移動・受け入れを行う．

デッドラインミス予測

タスク T_i のデッドラインミスを予測するためには，実行前に終了時刻 f_i を見積もる必要がある．

他のタスクの起動や割り込みが発生せず，かつスケジューリングオーバヘッドを無視できるという仮定のもと，時刻 c において優先順位でタスクを並べた結果が $T_1, T_2, T_3, ...$ のとき，

$$f_i = \begin{cases} E_i + c & (i = 1 \text{ のとき}) \\ E_i + f_{i-1} & (i > 1 \text{ のとき}) \end{cases}$$

となり，T_i は $d_i < f_i$ の時，デッドラインミスタスクとなる．

d_i はタスク起動時に求まるが，E_i はタスクの実行が終了するまでわからないため，実際には予測実行時間として $WCET_i$ を用いる．

しかしながら，WCET は実際の実行時間より大きく見積もられるため，実際のデッドラインミス数と比較して予測デッドラインミス数は大きくなる傾向にある．このため，実際の実行時間によってはデッドラインを守れるタスクを移動する可能性があり，その結果デッドラインミスするケースが増える．これを避けるために WCET に係数をかけ，WCET よりも小さい予測実行時間を用いることで予測デッドラインミスを抑制し，誤った予測によるミスタスクの増加を防ぐ．

3.3.3 タスク移動アルゴリズムと移動タスクの選択

高負荷状態におけるタスク移動

負荷計算の結果高負荷状態に分類された場合，図 3.6 で示すタスクの追い出し処理を実行する．移動対象タスクとして，負荷計算時にデッドラインミス予測が発生したタスクが選択される．タスクの追い出しは，移動対象のタスク ID によって決まる共有タスクプール上のアドレスに，移動対象タスクの絶対デッドライン時刻を書き込むことで実現する．また，過去に起動された同一タスクにおいて追い出し処理が実行され，かつ低負荷プロセッサに受け入れられていない場合は，共有タスクプール上に同一タスクが存在する．その場合，移動対象タスクは移動せずに起動したプロセッサで実行される．

タスクを共有タスクプール上に書き込んだ後，起動要求がキューイングされていた場合は，新たなデッドライン時刻を設定し，起動要求キューイング数をデクリメントし，自身のレディキューに再登録する．タスク移動の際にキューイングを含めないことで極端な負荷の移動を避け，また移動先プロセッサにおける受け入れ判断が容易になる．起動要求がキューイングされていない場合は，タスクは休止状態に遷移する．

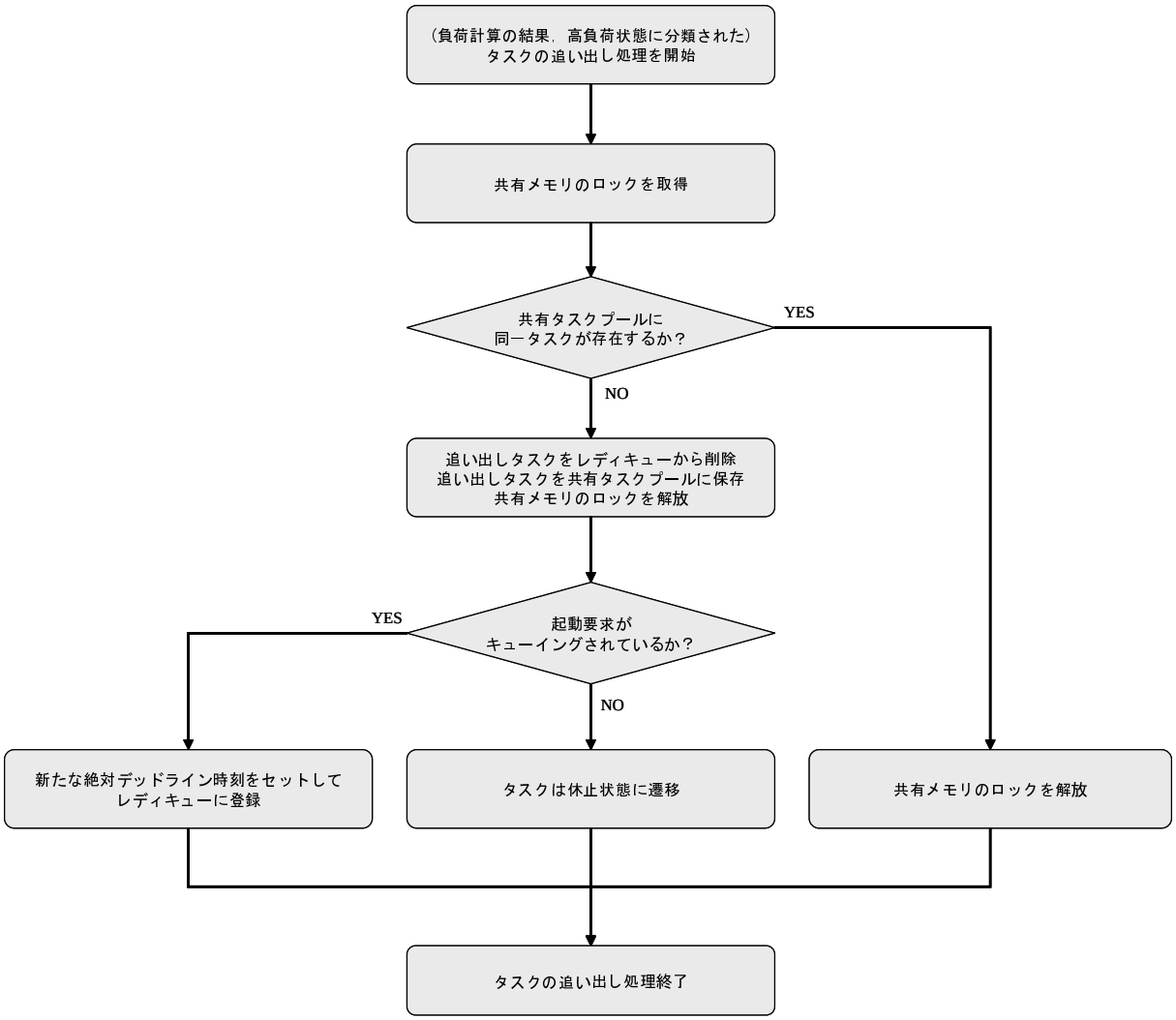


図 3.6: タスク追い出し処理のフローチャート

低負荷状態におけるタスク移動

負荷計算の結果低負荷状態に分類された場合，図 3.7 で示すタスクの受け入れ処理を実行する．静的優先度が高いタスクを優先して共有タスクプールから受け入れるため，共有タスクプール上の先頭（タスクセット中タスク ID が最小のタスクが保存される場所）からタスクの有無を調べる．タスクが存在し，かつそのタスクが休止状態である（受け入れ側プロセッサのレディキューに存在しない）場合は，受け入れ側プロセッサにおいてスケジューリング可能かを調べ，デッドラインミス予測が起こらなければタスクを受け入れる．このときのデッドラインミス予測には，受け入れ側プロセッサにおける WCET（またはそれを元にした値）を用いる必要がある．また，実行可能状態のタスクがない場合は，デッドラインミス予測を行わずにプール上のタスクを受け入れる[3]．

3.3.4 スケジューラの構成

本実装におけるスケジューラの動作を図 3.8 に示す．実行中のタスクがタスク終了のためのシステムコール ext_tsk を呼び出すとスケジューラが実行される．スケジューラはレディキューを参照しながら負荷計算を行い，高負荷時はタスクの追い出し処理を実行し，低負荷時はタスクの受け入れ処理を実行し，負荷の偏りを小さくする．また，通常負荷時はシステム設計者による静的なタスク割り当てを守り，タスク移動は行わない．負荷計算後，スケジューラはスケジューリングを行い最高優先順位のタスクの実行を開始する．

[3]本実装では，ソフトリアルタイムシステムを対象としているため，デッドラインを超過したタスクも実行する．従ってレディキューに実行タスクが存在しない場合は，デッドラインミスの可能性にかかわらず共有タスクプールのタスクを受け入れる．

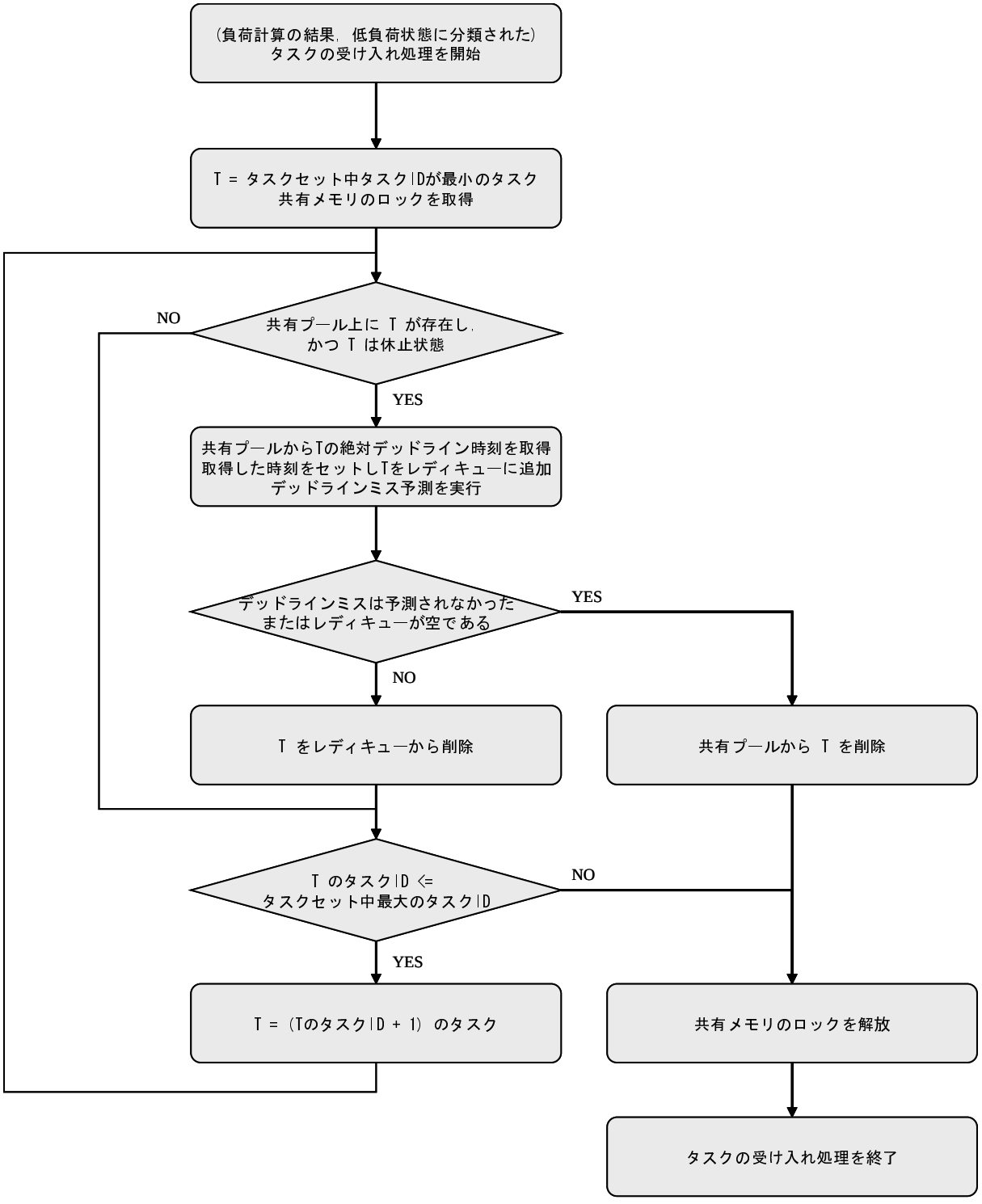


図 3.7: タスク受け入れ処理のフローチャート

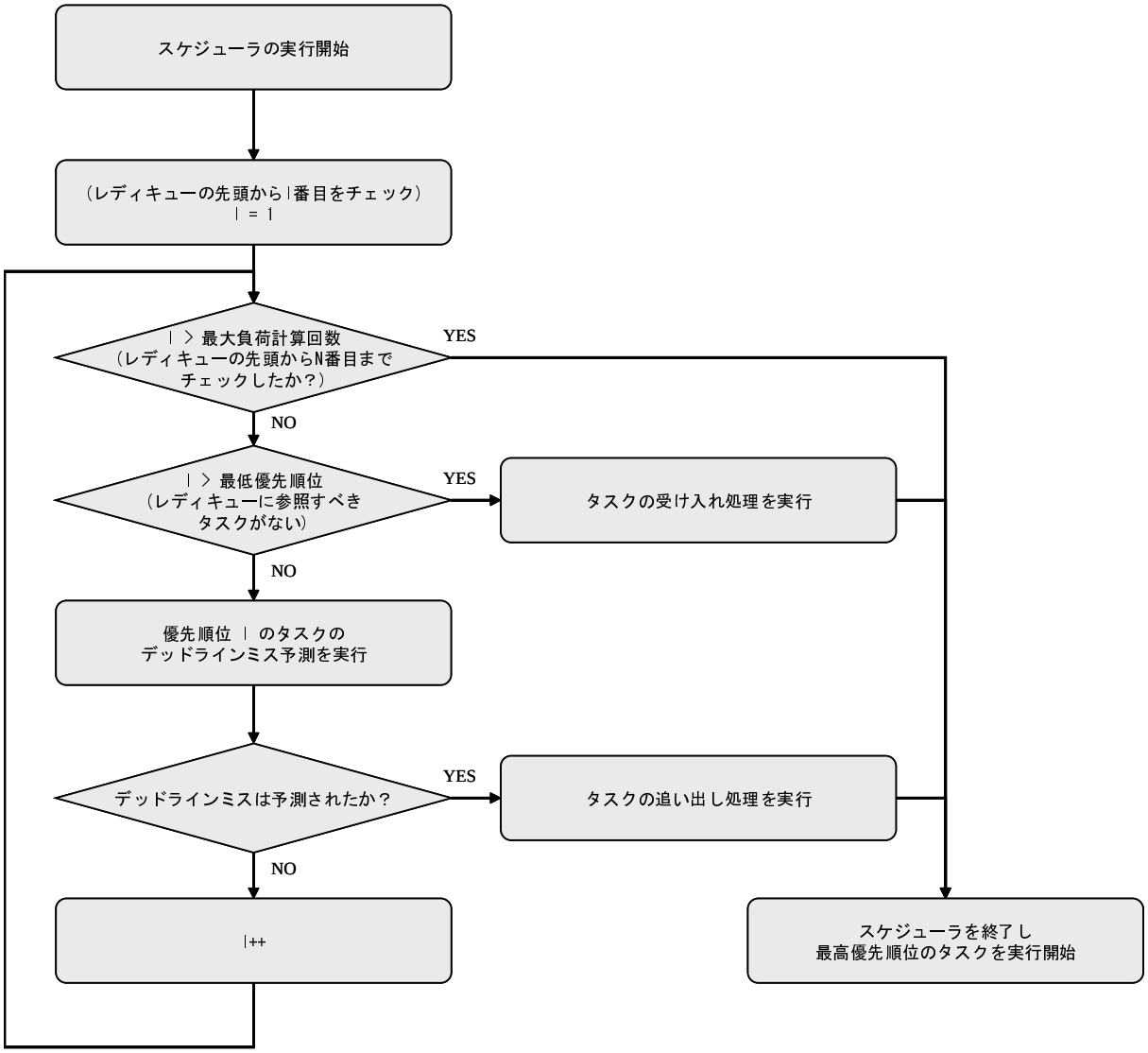


図 3.8: スケジューラのフローチャート

第4章　評価

本章では，作成したシミュレーション環境の説明を行い，提案方式の評価を行う．

4.1　シミュレーション環境

図4.1に本シミュレータのモデル図を示す．C言語で記述されたプログラムを各ターゲットプロセッサ用にコンパイルし，各プロセッサシミュレータに入力する．各シミュレータは入力プログラムを実行し，実行サイクル数，デッドラインミス数，タスクの応答時間，各CPU使用率などを出力する．

4.1.1　機能分散型マルチプロセッサシミュレータ

評価環境として作成した機能分散型マルチプロセッサのシミュレータはクロックサイクルレベルのMIPS ISAとDSP ISAのシミュレータから構成され，各プロセッサは共有メモリに接続されている（3.3節参照）．

MIPSシミュレータ

本研究における機能分散型マルチプロセッサで汎用コアとして使用されるMIPSのシミュレータは，ローカルメモリおよびL1命令キャッシュ，L1データキャッシュを持つ．キャッシュに関する設定を表4.1に示す．なお，キャッシュヒット時のメモリ参照は1サイクルで完了するものとする．表4.1に含まれない命令も同様に1サイクルで完了するものとする．

DSPシミュレータ

専用コアとして使用されるDSPの設定を表4.2に示す．DSPでは，命令を実行するために必要なサイクル数が各命令ごとに異なる．また，DSPはキャッシュを持たず，メモリアクセスは基本的に1サイクルで完了するが，複数サイクル必要とする命令も存在する．詳細は[5]に記載されている．

また，DSPは以下の特徴を持つ．

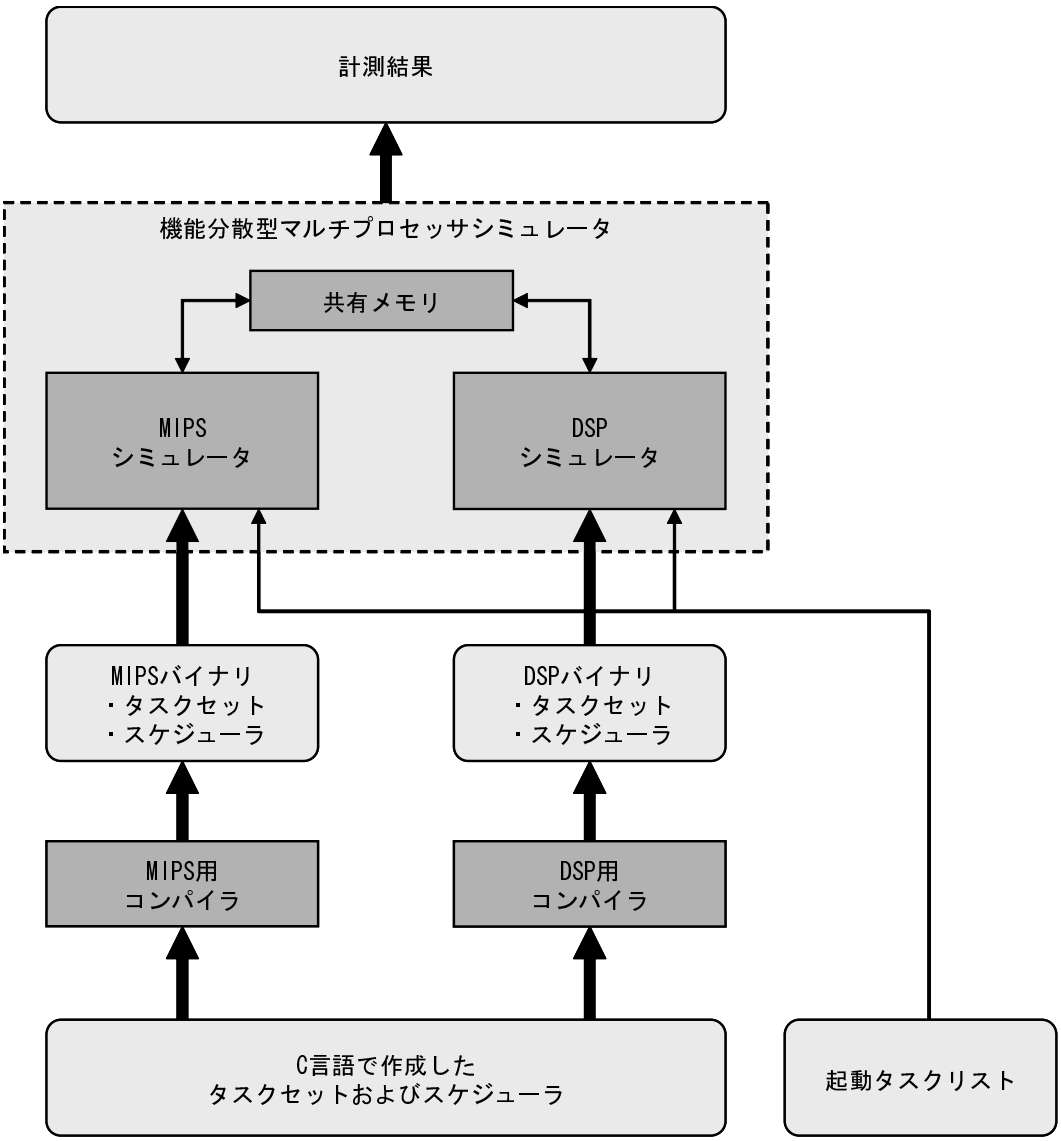

計測結果
機能分散型マルチプロセッサシミュレータ
共有メモリ
MIPS
シミュレータ
DSP
シミュレータ
MIPSバイナリ
・タスクセット
・スケジューラ
DSPバイナリ
・タスクセット
・スケジューラ
MIPS用
コンパイラ
DSP用
コンパイラ
C言語で作成した
タスクセットおよびスケジューラ
起動タスクリスト

図 4.1: 評価環境

ローカルメモリサイズ	16MB
L1 命令キャッシュサイズ	4KB
L1 命令キャッシュブロックサイズ	16Byte
L1 命令キャッシュウェイ数	2
L1 命令キャッシュミスペナルティ	10Cycle
L1 データキャッシュサイズ	4KB
L1 データキャッシュブロックサイズ	16Byte
L1 データキャッシュウェイ数	2
L1 データキャッシュミスペナルティ	10Cycle
乗算命令のレイテンシ	2Cycle
除算命令のレイテンシ	10Cycle

表 4.1: MIPS シミュレータの設定

プログラムメモリサイズ	64KB
データメモリサイズ	64KB

表 4.2: DSP のシミュレータの設定

- 可変長命令，算術論理演算のオペランドにメモリアドレスを指定可能など，CISC の特徴を持つ．
- コア上にプログラムメモリとデータメモリが存在し，サイズはそれぞれ 64KB である．プログラムメモリとデータメモリのアドレス空間は独立しており，アドレス幅はそれぞれ 16bit である．他に外部デバイスなどとのアクセスに使用される IO メモリ空間を持り，アドレス幅は同様に 16bit である．
- 40bit アキュムレータと乗算器を持ち，積和演算を高速に実行することができる．

共有メモリ

共有メモリのサイズは 64KB とし，各プロセッサにおける共有メモリへのアクセスレイテンシは 20Cycle とする．

4.1.2 入力バイナリの作成

本シミュレータは MIPS32 ISA のバイナリ（MIPS バイナリ）と TMS320C54x DSP ISA のバイナリ（DSP バイナリ）を入力ファイルとする．本方式では，各プロセッサはすべてのタスクについて実行可能でなければならない．すなわち，各入力バイナリは全タスクの実行コードを含む必要がある．よって，全タスクのソースコードを MIPS 用コンパイラと DSP 用コンパイラでコンパイルし，MIPS バイナリと DSP バイナリを作成する．MIPS 用

コンパイラとして，Cygwin[6] 上で動作するクロスコンパイラを作成した．クロスコンパイラのターゲットは mipsisa32-elf とした．DSP 用コンパイラは，Code Composer Studio（Texas Instruments 社が提供する開発環境 [7]）付属のコンパイラを使用した．

タスクを管理するシステムコールを作成するにあたり μITORN4.0 仕様 [8] を参考にした．使用可能なシステムコールの一覧を表 4.3 に示す．

cre_tsk	タスク管理機能	タスクの生成
act_tsk	タスク管理機能	タスクの起動
ext_tsk	タスク管理機能	自タスクの終了
cre_cyc	タスク管理機能	周期タスクの生成
snd_mbx	同期・通信機能	メールボックスへの送信
rcv_mbx	同期・通信機能	メールボックスからの受信
sta_cyc	時間管理機能	周期タスクの周期起動を開始
get_tim	時間管理機能	現在時刻の取得

表 4.3: 使用可能なシステムコール一覧

4.2 スケジュール評価関数

本方式の性能を評価するために，スケジュール評価関数を用いる．利用する評価関数を以下に示す．

- **デッドラインミス率**
  n 個のタスクが実行完了したとき，デッドラインミス率 M_{rate} を以下に定義する．

$$M_{rate} = \frac{1}{n}\sum_{i=1}^{n} miss(f_i)$$

  ただし，$miss(f_i)$ は以下の定義とする．

$$miss(f_i) = \begin{cases} 0 & (f_i \leq d_i \text{のとき}) \\ 1 & (\text{それ以外のとき}) \end{cases}$$

- **平均応答時間**
  n 個のタスクが実行完了したとき，平均応答時間 R_{avg} を以下に定義する．

$$R_{avg} = \frac{1}{n}\sum_{i=1}^{n}(f_i - a_i)$$

4.3 タスクセット

各タスクセットは，表 4.4 のタスクの組み合わせによって作成する．2.1.4 節で述べたように，タスク優先度はタスク ID が小さいタスクほど高優先度となる．MIPS-WCET は，事前に MIPS シミュレータで各タスクを仮実行し，その実行時間に基づいて決められた値である．DSP-WCET も同様に，事前に DSP シミュレータで仮実行し，その実行時間を基に決定した．また，各タスクの相対デッドライン時間は，各タスクの WCET と優先度に基づいて決定した．優先度が低いタスクは実行開始が遅くなる傾向があるため，優先度が低いタスクの相対デッドライン時間は大きい値になる．表内で DSP-WCET < MIPS-WCET となっているタスクは，積和演算を多様している傾向がある．

タスクセットは以下の規則に従ってタスクを組み合わせる．

1. 各タスクは WCET の値が小さいプロセッサに割り当てる．
2. 1. で割り当てられたタスク中から，各タスクセットごとに起動するタスクを選択する．

これにより，システム設計者による静的なタスク割り当て（負荷分散）を加味したタスクセットを作成することができる．

周期タスクの周期 T は，各タスクの WCET とプロセッサ負荷設定値 U に基づいて決定する．そのプロセッサ上で起動する周期タスクの数を n とするとき，T_i は以下の式で求まる．

$$T_i = \frac{WCET_i \times n}{U}$$

プロセッサ負荷設定値はシミュレータ実行時に与えられる入力データであり，この値を変更することで各プロセッサの CPU 使用率を調整する．また，非周期タスクは 1,000,000 クロックと 1,500,000 クロックに起動要求が発生するものとする．

計測方法

計測にあたり，プロセッサ負荷設定値を調整することで以下の 5 通りの負荷の偏りを作り出した．各ケースに対して 12 種類のタスクセットをシミュレータで実行し，その平均値を計測結果とする．

計測 1. MIPS と DSP の負荷の比率が 5:1 のケース．

計測 2. MIPS と DSP の負荷の比率が 2:1 のケース．

計測 3. MIPS と DSP の負荷の比率が 1:1 のケース．

計測 4. MIPS と DSP の負荷の比率が 1:2 のケース．

計測 5. MIPS と DSP の負荷の比率が 1:5 のケース．

タスク ID	優先度	周期／非周期	MIPS-WCET	DSP-WCET	相対デッドライン時間
1	1	周期	24584	9658	133422
2	1	周期	19517	16209	125820
3	2	周期	12929	25835	163272
4	2	周期	26170	13175	156068
5	2	周期	13578	27170	165944
6	3	周期	12330	28532	168668
7	3	周期	13883	27363	166328
8	3	周期	13410	33110	177824
9	4	周期	15584	31148	198625
10	4	周期	11855	40049	220875
11	4	非周期	42826	14154	217975
12	5	非周期	19026	38908	218025
13	5	非周期	23117	37602	214760
14	5	非周期	34877	28545	198100
15	6	非周期	21276	49682	278952
16	6	非周期	51598	23231	272886
17	6	非周期	11638	64754	324168
18	7	非周期	21316	57242	301632
19	7	周期	21000	61148	313350
20	8	周期	39585	43204	290271
21	8	周期	18575	75029	401655
22	8	周期	28844	71990	391022
23	9	周期	27570	70247	384918
24	9	周期	79863	33130	404791
25	9	周期	36771	75471	403202
26	10	周期	42142	105306	569432
27	10	周期	74627	82307	477432
28	10	周期	106396	42106	598040
29	10	周期	123420	61082	746136
30	10	周期	49248	125326	649512

表 4.4: 各タスクセットを構成するタスクの情報

4.4 スケジューラのオーバヘッド

実際にシミュレータで測定したスケジューラのオーバヘッドの平均値を表 4.5 に示す．ここでいうスケジューラのオーバヘッドとは，あるタスクが終了してから次のタスクが開始されるまでの時間を指す．

	タスク移動あり実行	タスク移動なし実行
MIPS	1081 サイクル	251 サイクル
DSP	1448 サイクル	310 サイクル

表 4.5: シミュレータ上で計測したスケジューラのオーバヘッドの平均値

表より，タスク移動を伴うスケジューラ実行のほうが，スケジューリングオーバヘッドが大きいことがわかる．これは主に共有メモリをアクセスするオーバヘッドが原因である．

4.5 シミュレーション結果

計測 1〜5 に対し，提案方式と従来方式[1]によるシミュレーションを行った．各計測に対し，システム全体のデッドラインミス率と全タスクの応答時間の平均値を観測する．なお，各計測の観測時間（全実行サイクル数）は 2,000,000 サイクルとする．シミュレーション結果を図 4.2〜図 4.11 に示す．

各ページの上段のグラフはプロセッサ使用率に対するデッドラインミス率である．縦軸はデッドラインミス率（%）を意味し，横軸はプロセッサ使用率（%）である．プロセッサ使用率 P_{usage} は，観測時間 t において実行時間 C のタスクが n 個完了した時，以下に定義される[2]．

$$P_{usage} = \frac{1}{t}\sum_{i=1}^{n} C_i$$

加えて，プロセッサ使用率に関して以下のことを定める．

1. グラフの横軸に対応するプロセッサ使用率の対象プロセッサは計測ごとに異なる．計測 1〜2 は MIPS の使用率，計測 3 は MIPS の使用率と DSP の使用率の平均，計測 4〜5 は DSP の使用率とする．この理由を以下に示す．負荷に偏りが生じた場合，デッドラインミスは主に高負荷プロセッサで発生する．提案方式に対する評価は，高負荷プロセッサのデッドラインミス数の削減率をもって行う．よって，高負荷プロセッサにおけるプロセッサ使用率に着目するため，上記をグラフの横軸とする．
2. 提案方式の結果は，同一タスクセットを従来方式で実行した時のプロセッサ使用率に対してプロットされる．提案方式では高負荷プロセッサから低負荷プロセッサへタス

[1]ここでいう従来方式とは，負荷計算およびタスク移動を行わないスケジューリングを指す．
[2]実行時間 C については 2.1.1 参照．

ク移動を行うため，従来方式とはプロセッサ使用率が異なる．従来方式のデッドラインミス率と提案方式のデッドラインミス率の違いを比較するため，上記のプロット方法を行う．

下段のグラフはプロセッサ使用率に対する平均応答時間である．縦軸は実行完了したタスクの応答時間の平均値（サイクル数）を意味し，横軸は上段のグラフと同様である．

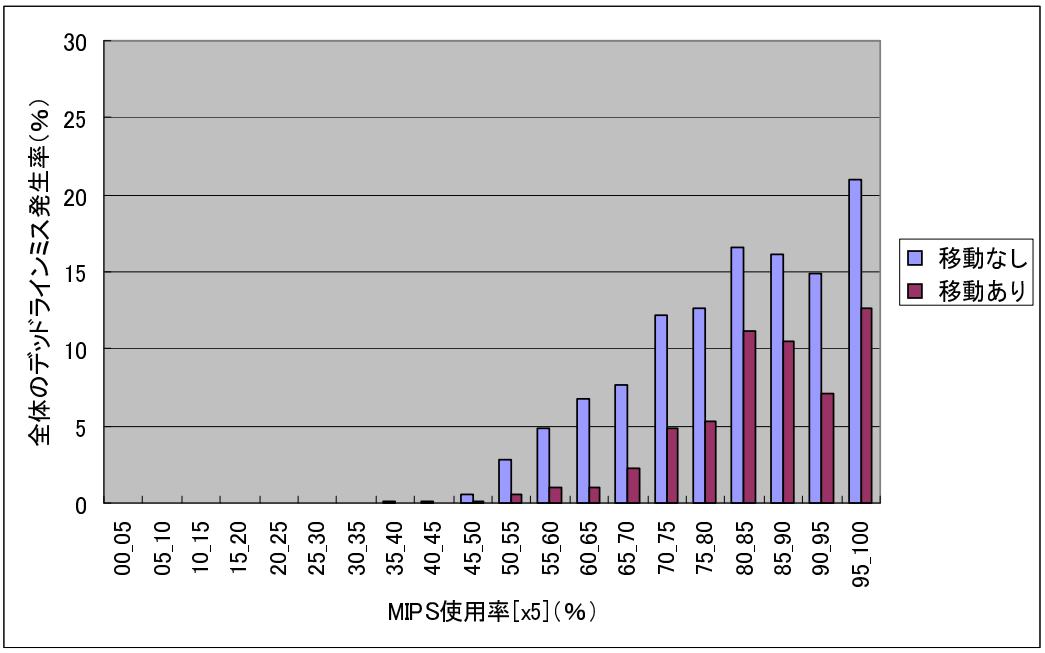


図 4.2: 計測 1：MIPS と DSP の負荷の比率が 5:1 の時の全体デッドラインミス率

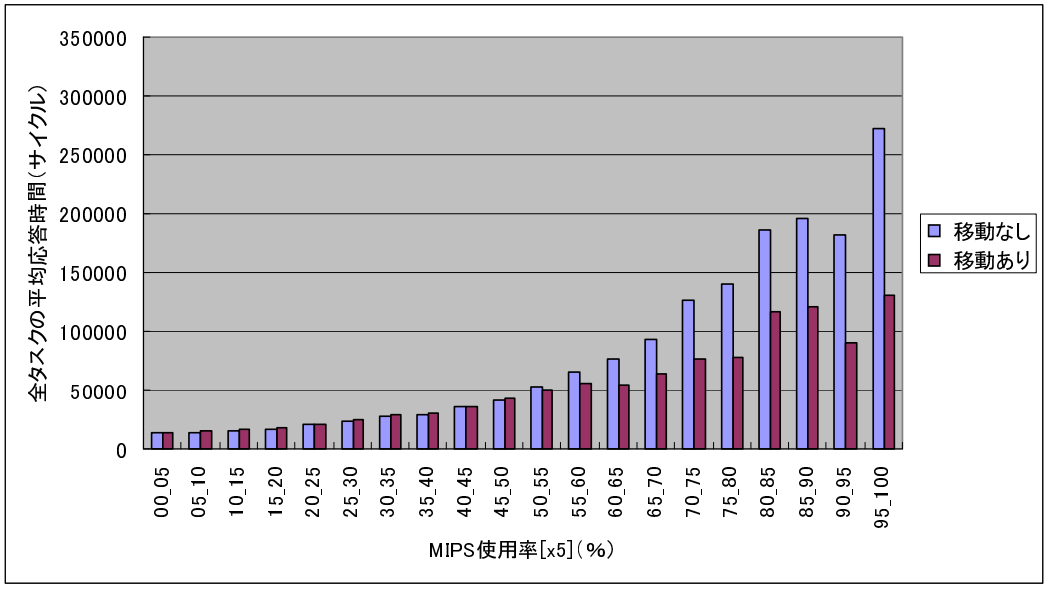


図 4.3: 計測 1：MIPS と DSP の負荷の比率が 5:1 の時の全タスクの平均応答時間

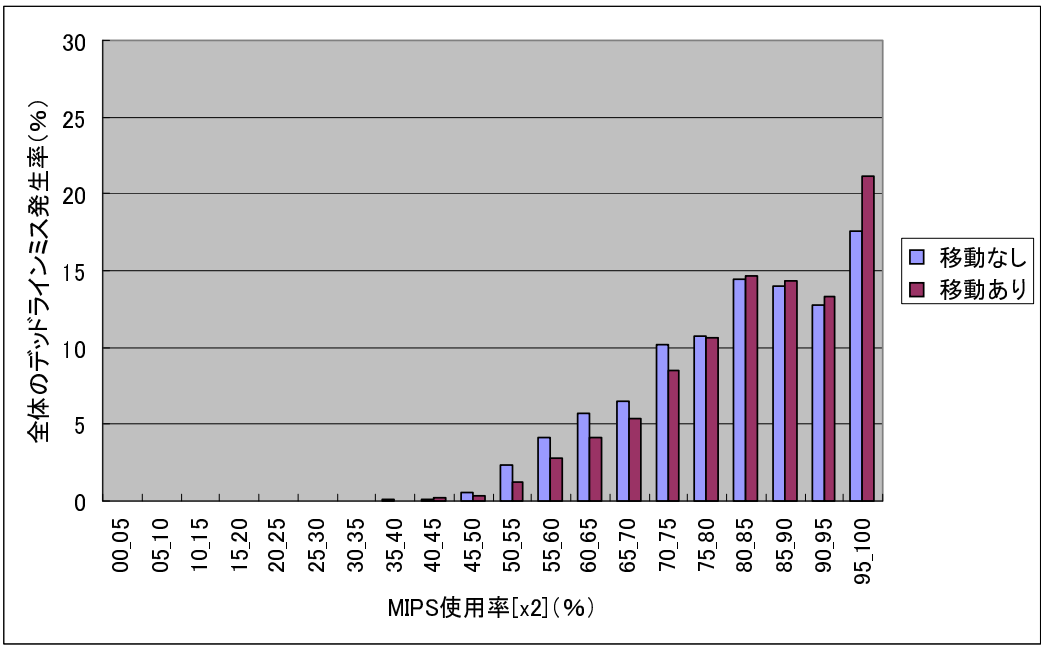


図 4.4: 計測 2：MIPS と DSP の負荷の比率が 2:1 の時の全体デッドラインミス率

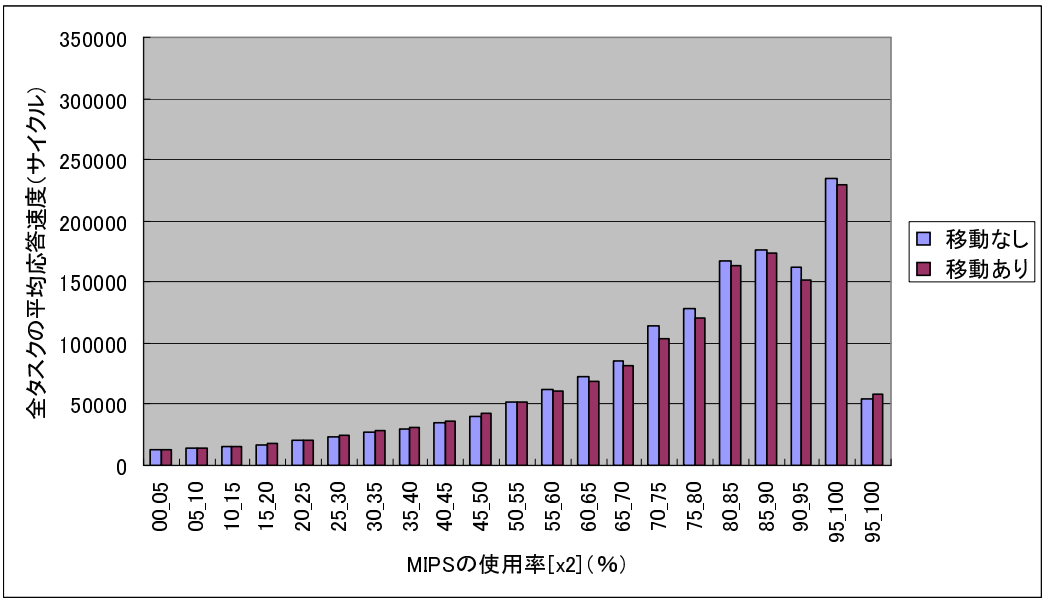


図 4.5: 計測 2：MIPS と DSP の負荷の比率が 2:1 の時の全タスクの平均応答時間

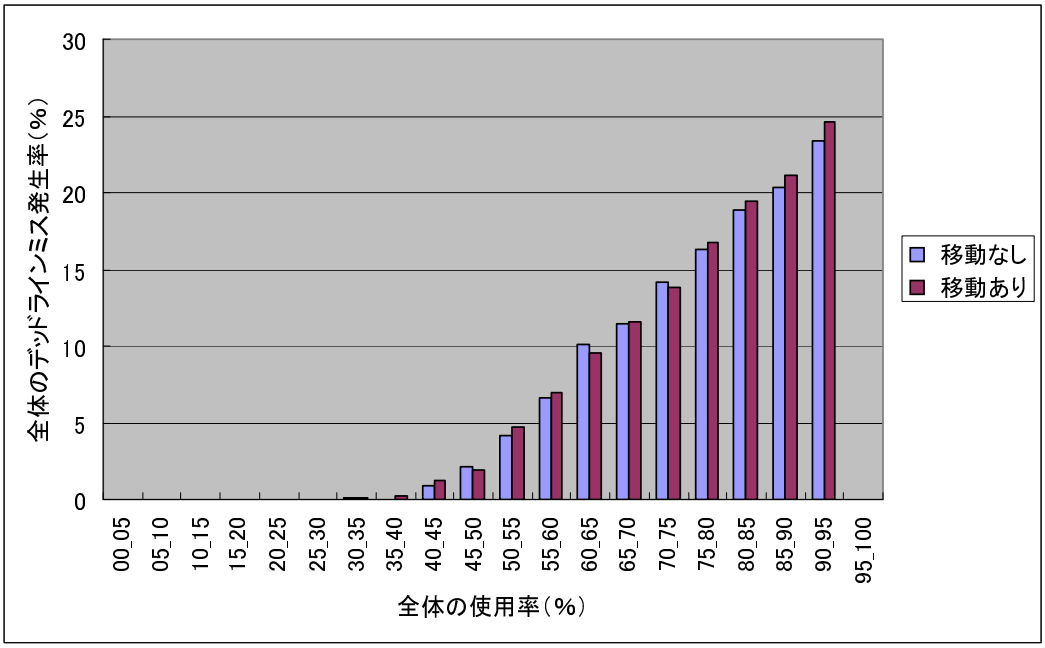


図 4.6: 計測 3：MIPS と DSP の負荷が均衡している時の全体デッドラインミス率

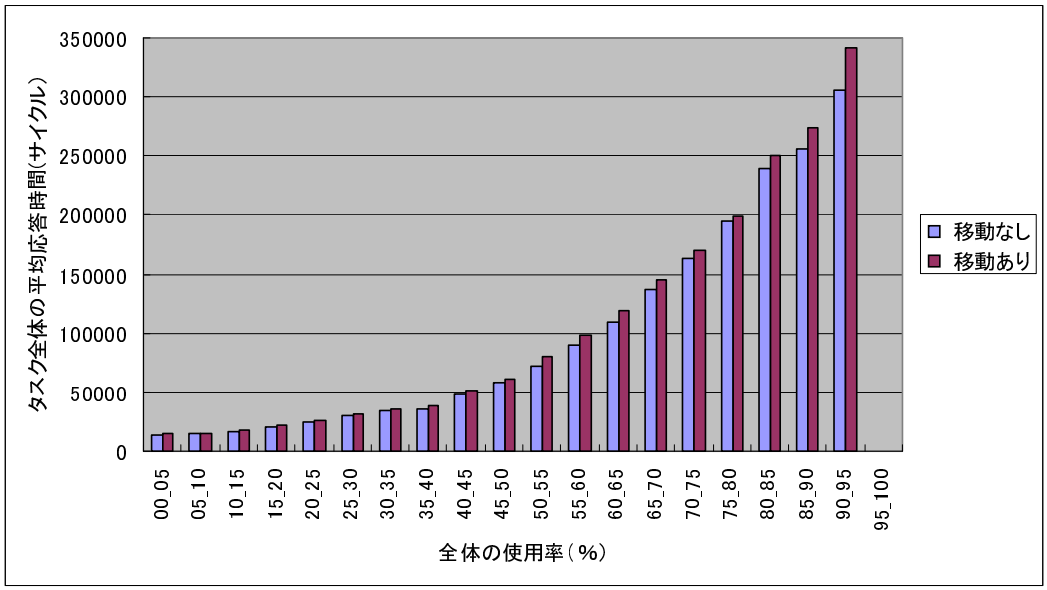


図 4.7: 計測 3：MIPS と DSP の負荷が均衡している時の全タスクの平均応答時間

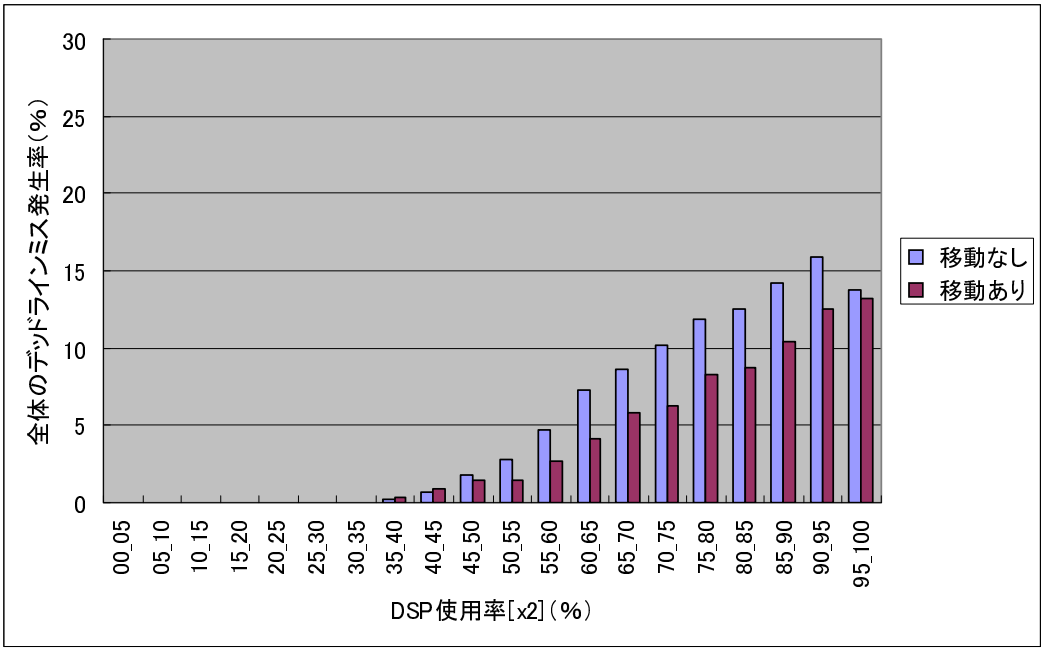


図 4.8: 計測 4：MIPS と DSP の負荷の比率が 1:2 の時の全体デッドラインミス率

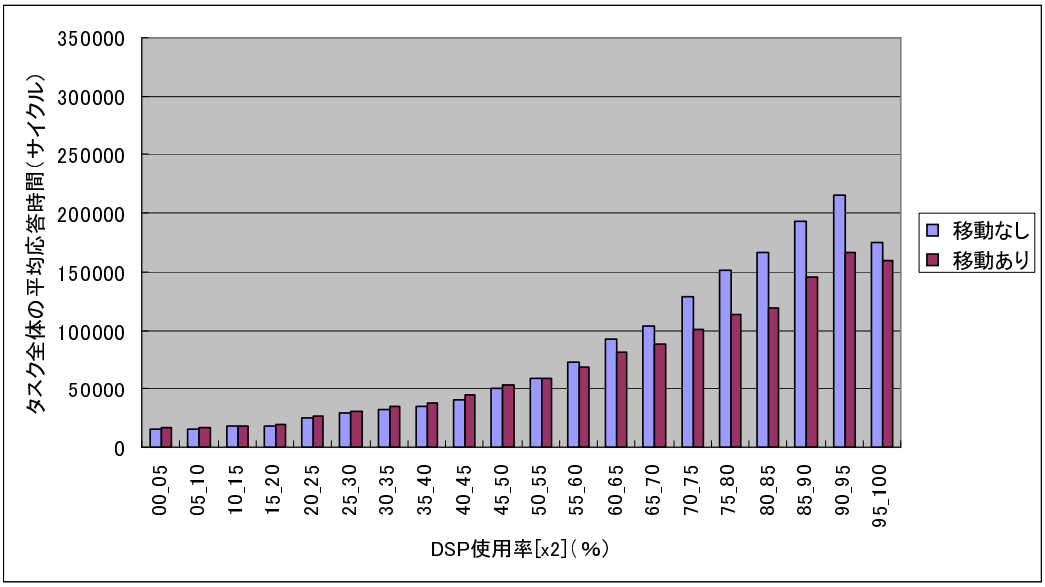


図 4.9: 計測 4：MIPS と DSP の負荷の比率が 1:2 の時の全タスクの平均応答時間

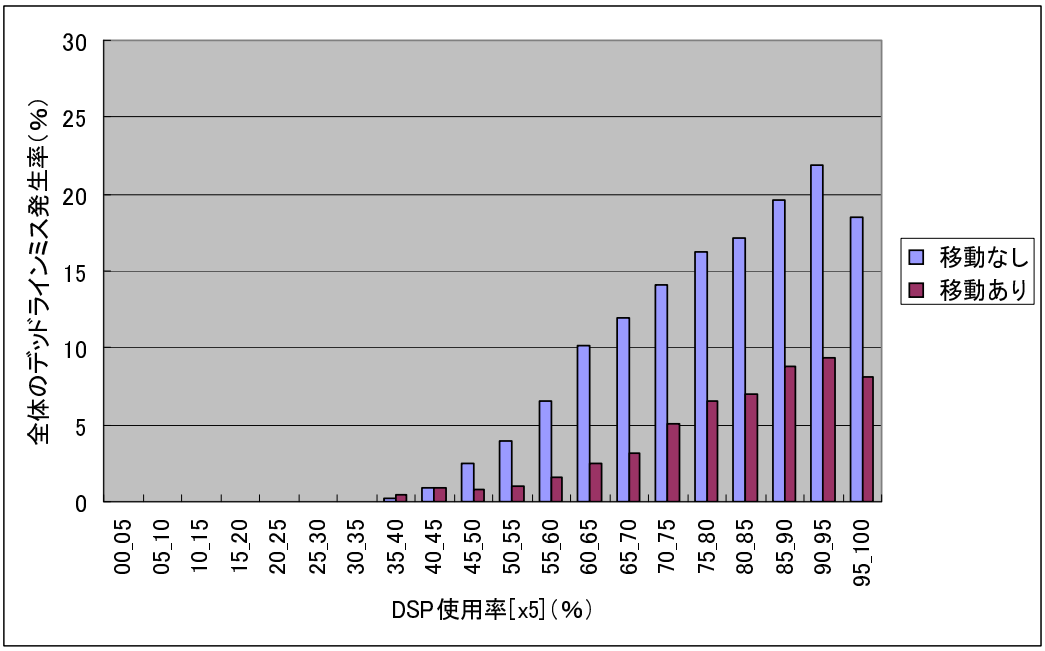


図 4.10: 計測 5：MIPS と DSP の負荷の比率が 1:5 の時の全体デッドラインミス率

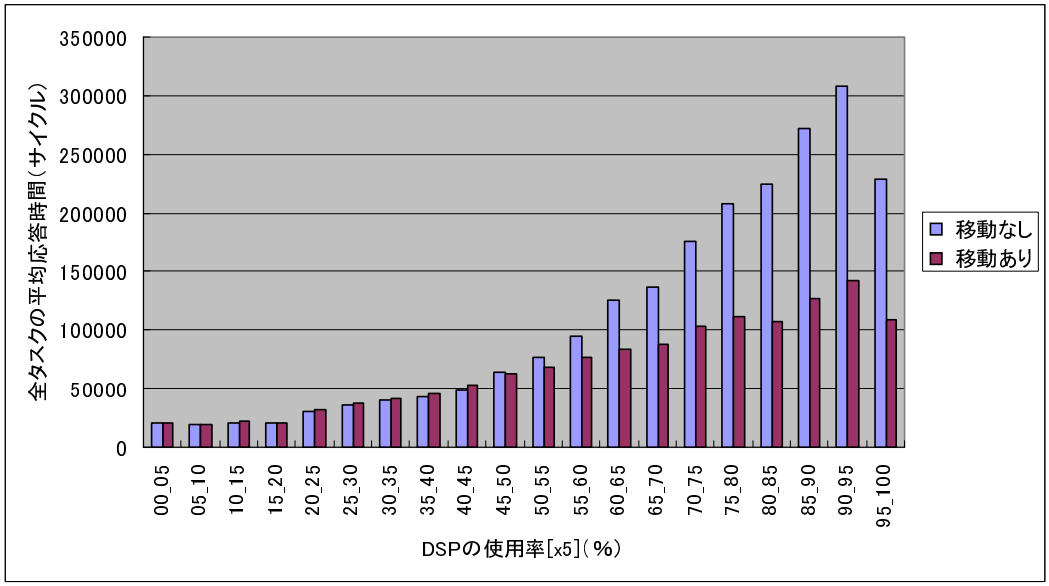


図 4.11: 計測 5：MIPS と DSP の負荷の比率が 1:5 の時の全タスクの平均応答時間

4.6 考察

各計測結果から提案方式と従来方式を比較，考察する．

4.6.1 デッドラインミス数と平均応答時間の比較

計測 1

計測 1 は，DSP の使用率が MIPS 使用率の約 20 %となった状況，つまり MIPS に比べ DSP にかかっている負荷が小さい状況に対する計測である．

図 4.2 に示されるように，提案手法は従来手法に比べ，全体的にデッドラインミス率が減少している．このことから，5 倍程度の負荷の偏りが発生した場合，提案手法は有効であるといえる．なお，使用率 85 %以上において，負荷が高まったにもかかわらずデッドラインミス率が低下する逆転現象がみられるが，組み込みシステムにおいてプロセッサ使用率が 80 %を超える状況は現実的ではないため，考慮しない．

応答時間に関しては，高負荷時では改善がみられるが，低負荷時は若干ながら悪化している（図 4.3）．デッドラインミスの発生が顕著になる MIPS 使用率 50 %以上において応答速度は改善傾向になる．この傾向は，タスクが移動し低負荷プロセッサで実行されたことで，タスクの実行開始が早まったためである．

計測 2

計測 2 は，DSP の使用率が MIPS 使用率の約 50 %となった状況に対する計測である．MIPS の負荷に比べ DSP の負荷は小さいが，計測 1 と比較すると DSP は高負荷である．

図 4.4 に示されるように，使用率 45〜80 %においては，デッドラインミス率は低下しているため提案方式は有効であるといえる．しかし，計測 1 と比較するとデッドラインミス率の改善幅は小さい．これは DSP の負荷が増えたため，共有タスクプール上のタスクを受け入れる回数が減ったことを意味する．また，平均応答時間に関しては，計測 1 と同様の傾向が見られる（図 4.5）．

表 4.6 は，計測 1 と計測 2 における共有タスクプールへのタスク移動が発生した回数をまとめたものである．追い出し成功とは，共有タスクプールにタスクを保存できたことを意味する．それに対して，追い出し失敗は共有タスクプール上にすでに同一タスクが存在し，タスクの追い出しができなかったことを意味する．この場合，追い出しに失敗したタスクは自身のプロセッサで実行される．受け入れ成功は，タスク受け入れ処理時に，共有タスクプール上のタスクをスケジューリング可能であったことを意味する．

共有タスクプール上でデッドライン時刻を過ぎたタスクが存在すると，アイドル状態のプロセッサが受け入れ処理を実行するまで共有タスクプールを占有する[3]．これは，後に

[3]アイドル状態とは，レディキューに実行可能なタスクがない状態を指す．アイドル状態のプロセッサにおいてタスクの受け入れ処理が発生すると，スケジューリングが可能かにかかわらずタスクを受け入れる．

同一タスクが起動してミス予測された場合に，タスク移動を妨げる．表 4.6 からも計測 2 は計測 1 と比較して，追い出し失敗回数が増え，受け入れ成功回数の減少していることを確認できる．

	追い出し成功回数	追い出し失敗回数	受け入れ成功回数
計測 1	1004	1533	289
計測 2	759	3527	85

表 4.6: 計測 1 と計測 2 におけるタスク追い出し／受け入れ回数の比較

計測 2 の考察をまとめると，計測 1 と同様にタスクの移動が発生し，デッドラインミス率および平均応答速度が改善されている．ただし，計測 1 と比較してその効果は小さい．よって，提案方式はタスクを受け取る低負荷側プロセッサの負荷のが小さいほど有効であるといえる．

計測 3

計測 3 は MISP と DSP の負荷が均衡している状況に対する計測である．図 4.6 からわかるように，デッドラインミス率に関しては，微差による改善と悪化が繰り返される．また，平均応答速度は全体的に悪化する傾向がみられる（図 4.7）．表 4.7 より，追い出し失敗回数が大きいことと受け入れ成功回数が小さいことがわかる．両プロセッサは自身を高負荷状態と判断し，タスクの追い出し処理を行うが，受け入れ処理を行うプロセッサが存在しない（あるいは受け入れ回数が少ない）ので，結果的にタスクを移動することができず，負荷計算が無駄になる．

以上のことから，負荷が均衡している状況では提案方式は有効とはいえない．

追い出し成功回数	追い出し失敗回数	受け入れ成功数
898	8096	31

表 4.7: 均衡状態におけるタスク追い出し／受け入れ回数

計測 4

計測 4 は，MIPS の使用率が DSP の使用率の 50 %となった状況の計測である．また，計測 4 は計測 2 における MIPS と DSP の負荷関係が逆転した状況といえる．計測 2 と比較すると図 4.8 より，低負荷時に若干デッドラインミス率が悪化しているが，高負荷時デッドラインミス率は計測 2 よりも大きく改善している．また，図 4.9 より，平均応答時間に関しても同様に負荷が高まるにつれ，改善される傾向にある．これはタスクが移動したことによって，移動タスクの開始時刻が早まったためである．

計測 5

計測 5 は，MIPS の使用率が DSP の使用率の 20 %となった状況の計測である．計測 4 と比べ，より DSP から MISP へタスク移動が生じやすくなった状況である．

図 4.10 より，デッドラインミス率は使用率 45 %以上において大きく改善していることがわかる．また応答速度も同様に 40 %以上において改善傾向が見られ，それ以下の低負荷状態では若干の性能低下が見られる（図 4.11）．

本計測においても計測 1,2,4 と同様に，高負荷時にはタスク移動したことによるデッドラインミス率改善と平均応答速度の改善がみられ，低負荷時にはスケジューリングオーバヘッドによる性能低下がみられる．

計測 1,2 と計測 4,5 を比較すると，DSP から MIPS へタスクが移動するケースの方がミス率が大きく改善されていることがわかる．以下に理由を述べる．

各タスクセットごとに，MIPS から DSP，および DSP から MIPS へタスク移動を行った際の実行時間の増加率（後述）を WCET を使用してそれぞれ求めた結果，前者は 2.36，後者は 1.82 であった．したがって，計測に使用したタスクセットにおいては，DSP から MIPS へのタスク移動のほうが，逆方向の移動よりも，成功する可能性が高いといえる．よって，DSP から MIPS へタスクを移動するほうが容易であったため，上記のような傾向がみられた．

なお，実行時間の増加率は以下のようにして求めた．あるタスクセットにおいて移動元プロセッサに静的に割り当てられた n 個のタスクに関して，自身で実行した場合の WCET を $WCET1$, 他方（移動先）のプロセッサで実行した場合の WCET を $WCET2$ とし，$Penalty$ を以下のように定義する．

$$Penalty = \frac{\sum_{i=1}^{n} WCET2_i}{\sum_{i=1}^{n} WCET1_i}$$

$Penalty$ は値が大きいほど移動先プロセッサでの実行時間の増加が大きいことを意味する．各タスクセットに対して上記の $Penalty$ を求め，平均した結果を実行時間増加率とする．

4.6.2 負荷移動率

各計測におけるプロセッサ使用率を提案方式と従来方式で比較し，提案方式による負荷の移動率を調べた．図 4.12〜図 4.16 は，各タスクセットにおける MIPS，DSP のプロセッサ使用率をプロットしたグラフである．縦軸はプロセッサ使用率（%），横軸は計測した 226 個のタスクセットを意味する．紺色のグラフはタスク移動を行わなかった場合の

MIPS のプロセッサ使用率，赤紫のグラフはタスク移動を行わなかった場合の DSP のプロセッサ使用率，黄色のグラフはタスク移動を行った場合の MIPS のプロセッサ使用率，水色のグラフはタスク移動を行った場合の DSP のプロセッサ使用率を意味する．従って，紺色のグラフと赤紫のグラフの差，および黄色のグラフと水色のグラフの差が負荷の移動率を示す．

計測 1

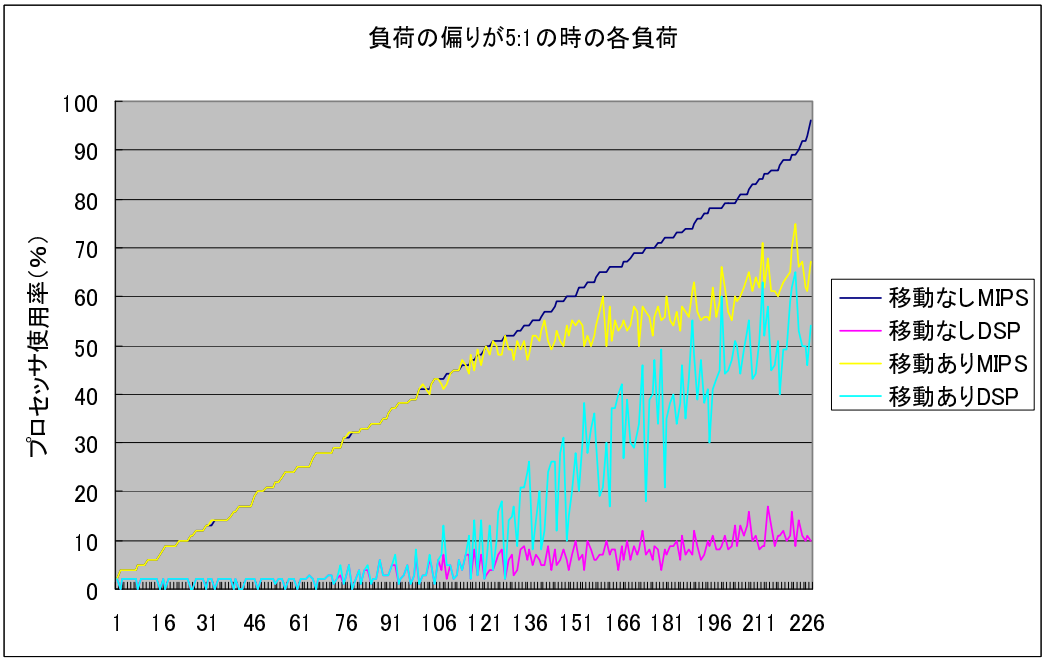


図 4.12: 計測 1：MIPS と DSP の負荷の比率が 5:1 の時の負荷の遷移

図 4.12 は計測 1 における負荷移動率を示したグラフである．このグラフはタスク移動を行わない場合の MIPS（高負荷）のプロセッサ使用率によってソートされている．つまり，横軸の値が大きいほど従来方式におけるプロセッサの負荷が高く，かつ負荷の偏りが大きいことを意味する．図 4.12 より，MIPS のプロセッサ使用率 40 %程度からタスク移動が発生し，使用率が高まるにつれて MIPS と DSP の使用率が近づいており，高負荷状態であるほど負荷の移動率が大きいことがわかる．

計測 2

図 4.13 は計測 2 における負荷移動率を示したグラフである．このグラフはタスク移動を行わない場合の MIPS のプロセッサ使用率によってソートされている．負荷が高まるにつ

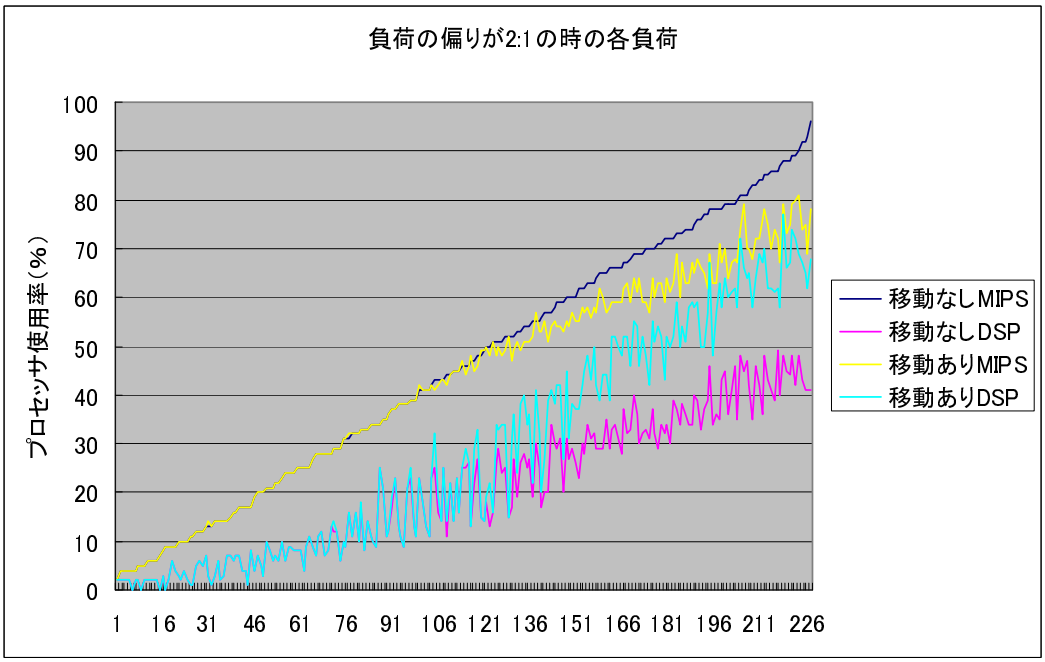


図 4.13: 計測 2：MIPS と DSP の負荷の比率が 2:1 の時の負荷の遷移

れて負荷移動率が大きくなる点は計測 1 と同様だが，MIPS，DSP ともに全体的な負荷移動率が減少している．これは，DSP の負荷が高まったことによって，DSP のタスク受け入れ数が減少したことと，それにより共有タスクプール上で移動タスクが競合して MIPS のタスク追い出しが失敗したことに起因している．

計測 3

図 4.14 は計測 3 における負荷移動率を示したグラフである．このグラフはタスク移動を行わない場合における MIPS のプロセッサ使用率によってソートされている．計測 1,2 と比較すると MIPS，DSP の負荷が均衡しており，互いに他方の負荷を受け入れる余裕がないため全体的な負荷移動率が小さい．プロセッサ使用率が高まっても DSP，MIPS のプロセッサ使用率に偏りができないため，計測 1,2 のように高負荷になるにつれ負荷移動率が増える傾向は見られない．また，若干ながら MIPS の負荷が高まり DSP の負荷が下がる傾向が見られるが，これは前節で述べた DSP から MIPS へのタスク移動が逆方向のタスク移動よりも成功しやすいことに起因している．

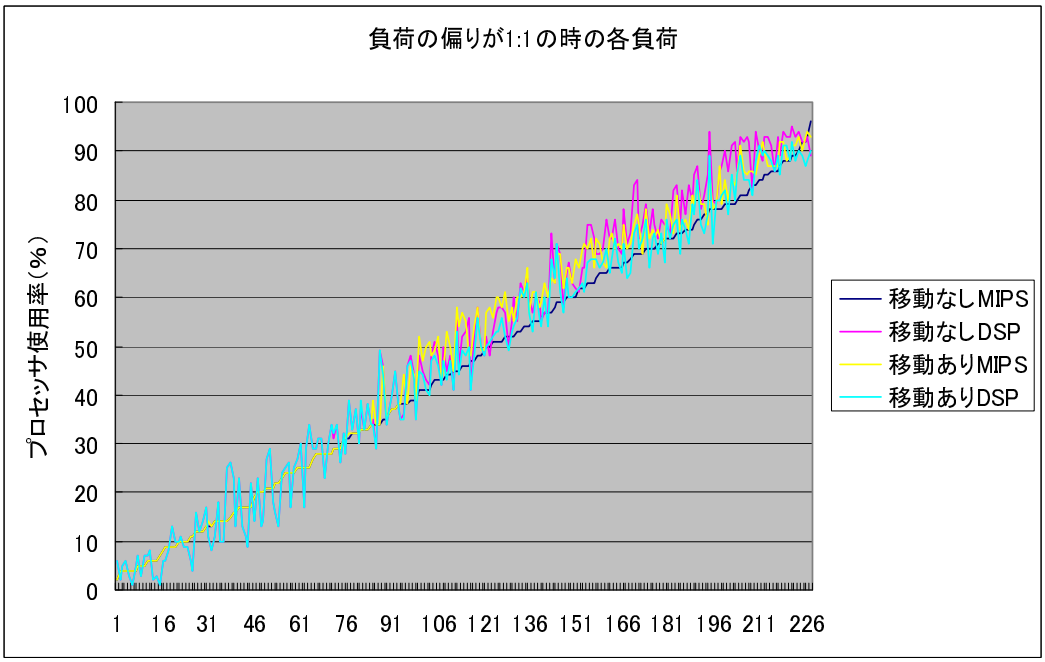


図 4.14: 計測 3：MIPS と DSP の負荷が均衡している時の負荷の遷移

計測 4

図 4.15 は計測 4 における負荷移動率を示したグラフである．このグラフはタスク移動を行わない場合における DSP（高負荷）のプロセッサ使用率によってソートされている．DSP から MIPS へと負荷が移動している点を除き，計測 2 と同様の傾向が見られる．

計測 5

図 4.16 は計測 5 における負荷移動率を示したグラフである．このグラフはタスク移動を行わない場合における DSP のプロセッサ使用率によってソートされている．DSP から MIPS へと負荷が移動している点を除き，計測 1 と同様の傾向が見られる．

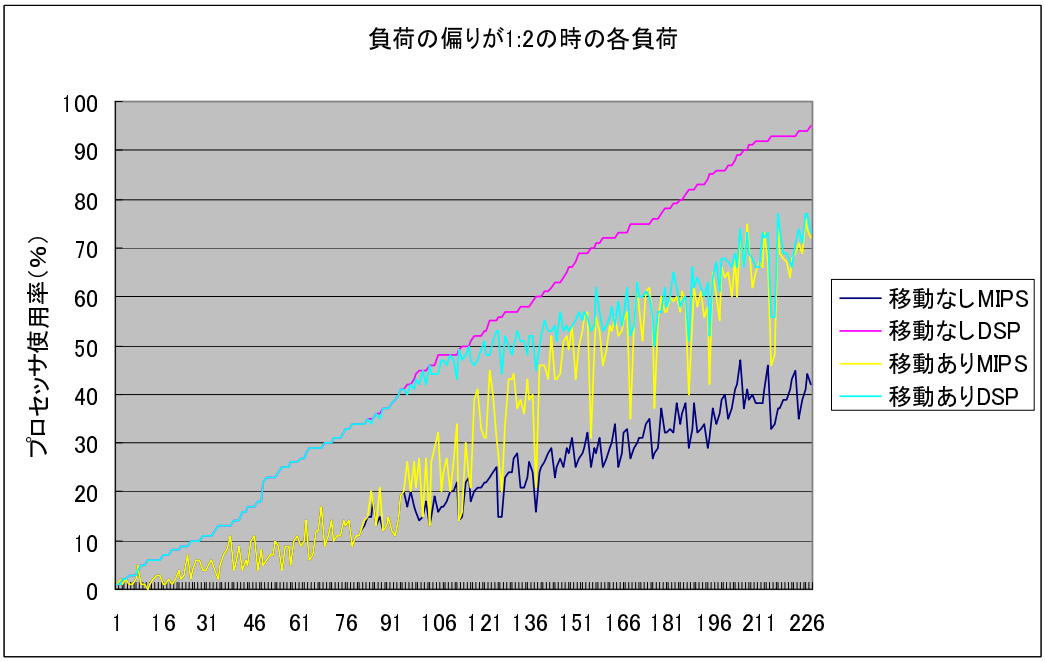


図 4.15: 計測 4：MIPS と DSP の負荷の比率が 1:2 の時の負荷の遷移

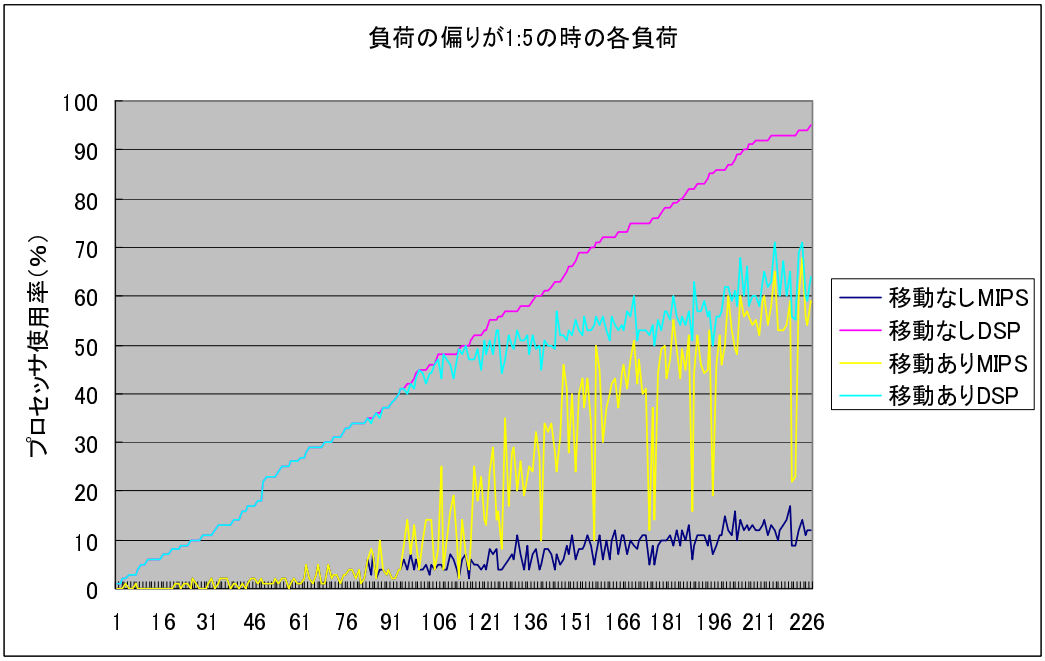


図 4.16: 計測 5：MIPS と DSP の負荷の比率が 1:5 の時の負荷の遷移

第5章　おわりに

5.1　まとめ

本研究では，機能分散型マルチプロセッサにおいて負荷の偏りが生じた場合に負荷分散を行うスケジューラを提案した．提案方式のスケジューラは，スケジューリングごとに負荷計算を行い，高負荷プロセッサから低負荷プロセッサへタスクを移動することで，高負荷プロセッサにおけるデッドラインミス数を削減する．

提案方式を評価するために，機能分散型マルチプロセッサのシミュレータとタスクセットを作成し，シミュレータ上で動作させることで提案方式と従来方式の比較を行った．デッドラインミス数および応答時間を計測した結果，2 倍以上の偏りが生じた際に，デッドラインミスが減少したことを確認した．また，デッドラインミスするタスクを移動することで平均応答時間が短縮されたことを確認した．

5.2　今後の課題

今後の課題として以下の項目が挙げられる．

- タスクの追い出し処理において，共有タスクプール上に同一タスクが存在するとタスクの追い出しは失敗する．計測 2,3 において，負荷の偏りが小さくなるにつれて追い出しに失敗する回数が増えることを確認した．今回は追い出し処理／受け入れ処理のオーバヘッド削減を優先したため実装しなかったが，1 タスクあたりの共有タスクプールを拡大，またはリスト構造とすることで，追い出し失敗回数の削減と受け入れ成功回数の増加を見込むことができる．

- 計測 3，すなわち負荷が均衡している場合は，提案方式は有効ではなかった．本方式では，各プロセッサは自身の負荷のみを判断基準としてタスク追い出し／受け入れを行う．これに対し，自身以外のプロセッサの負荷状態を観察し，他プロセッサとの負荷に開きがある場合のみタスク移動を行うように改良することで，負荷均衡時における性能低下を抑えることが見込める．

謝辞

本研究を行うにあたり，熱心かつ丁寧に指導してくださった北陸先端科学技術大学院大学 田中清史准教授に心から深く感謝いたします.

貴重なご助言を頂いた日比野靖教授，井口寧准教授に深く感謝いたします.

そして，本学での研究・日常生活・部活動において喜楽を共にした計算機アーキテクチャ講座の皆様に心から御礼申し上げます.

参考文献

[1] “機能分散型マルチプロセッサ向けのリアルタイム OS”, 高田広章, 本田晋也, 情報処理 47 巻 1 号 2006 年

[2] TRON Project, http://www.tron.org/

[3] TOPPERS Project, http://www.toppers.jp/

[4] “MIPS32 Architecture For Programmers” Revision 2.50, MIPS Technologies Inc., 2005

[5] “TMS320C54x DSP Reference Set”, Texas Instruments Inc., 2001

[6] Cygwin Project, http://cygwin.com/

[7] “Code Composer Studio User’s Guide”, Texas Instruments Inc., 2000

[8] “μITRON4.0 仕様” Ver.4.02.00, （社）トロン協会 ITRON 仕様検討グループ, 2004 年

[9] “See MIPS Run”, Dominic Sweetman,

[10] “Parallel Computer Architecture”, David E. Culler, Jaswinder Pal Singh,

[11] “μITRON4.0 標準ガイドブック”, 坂村健