JAIST Repository
https://dspace.jaist.ac.jp/

Title	分散リアルタイムシステムにおけるスケジューリングに関する研究
Author(s)	馬場, 久
Citation	
Issue Date	2004-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1773
Rights	
Description	Supervisor:田中 清史, 情報科学研究科, 修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

分散リアルタイムシステムにおける 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

馬場 久

2004 年 3 月

修 士 論 文

分散リアルタイムシステムにおける 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　堀口 進 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

210072 馬場 久

提出年月: 2004 年 2 月

Copyright © 2004 by Hisashi Baba

概 要

近年の急速な技術革新によって，かつては実現困難であった分散リアルタイムシステムの構築が可能になってきた．単一プロセッサ上のリアルタイムスケジューリングでは，最適なアルゴリズムが提案されている．それに対して，マルチプロセッサのシステムにおけるスケジューリングでは，最適性を求めることが困難である．本研究では，複数のプロセッサを持つ分散リアルタイムシステムにおいて，新たなタスク割り当ての手法を提案する．提案手法では，プロセッサの負荷の指標として，タスクの静的優先度と余裕時間を新たに導入する．シミュレーションによって提案方式の評価を行ない，静的優先度が高いタスクのデッドラインオーバーを減少できることを確認した．

目 次

第 1 章　はじめに **1**
1.1 研究の背景と目的 1
1.2 本論文の構成 1

第 2 章　リアルタイムスケジューリング **3**
2.1 リアルタイムタスクの属性 3
2.2 リアルタイムスケジューリング 4
2.2.1 ハードリアルタイムとソフトリアルタイム 4
2.2.2 スケジューリングアルゴリズムの分類 5
2.2.3 WCET 6
2.2.4 スケジューリングアルゴリズム 7
2.3 マルチプロセッサ上のスケジューリング 9
2.3.1 オンラインスケジューリング 9
2.3.2 オフラインスケジューリング 11
2.3.3 その他の問題 11
2.3.4 問題の分割 13

第 3 章　負荷の算出 **14**
3.1 負荷の指標 14
3.2 適応型動的優先度を用いた負荷計算法 15

第 4 章　評価 **17**
4.1 システムモデル 17
4.1.1 プロセッサとスケジューリング 17
4.1.2 タスクセット 18
4.2 分散リアルタイムスケジューリングシミュレータ 18
4.3 各オーバヘッド 19
4.4 スケジュール評価関数 20
4.5 シミュレーション条件 21
4.6 シミュレーション結果 23
4.7 考察 42

第 5 章　関連研究 **48**
5.1 Cheddar 48

第 6 章　おわりに **49**
6.1 まとめ 49
6.2 今後の課題 49

謝辞 **50**

参考文献 **51**

図 目 次

2.1 タスクの時間属性 4
2.2 リアルタイム性と価値関数 5
2.3 マルチプロセッサでの EDF の使用 10
2.4 Richard's anomalies の例 12

3.1 余裕時間の違いによって負荷が異なる例 15

4.1 システムモデルの模式図 17
4.2 シミュレーション 1 のデッドラインオーバー率 24
4.3 シミュレーション 1 の平均応答時間 25
4.4 シミュレーション 1 の遅れタスクの静的優先度平均 26
4.5 シミュレーション 2 のデッドラインオーバー率 27
4.6 シミュレーション 2 の平均応答時間 28
4.7 シミュレーション 2 の遅れタスクの静的優先度平均 29
4.8 シミュレーション 3 のデッドラインオーバー率 30
4.9 シミュレーション 3 の平均応答時間 31
4.10 シミュレーション 3 の遅れタスクの静的優先度平均 32
4.11 シミュレーション 4 のデッドラインオーバー率 33
4.12 シミュレーション 4 の平均応答時間 34
4.13 シミュレーション 4 の遅れタスクの静的優先度平均 35
4.14 シミュレーション 5 のデッドラインオーバー率 36
4.15 シミュレーション 5 の平均応答時間 37
4.16 シミュレーション 5 の遅れタスクの静的優先度平均 38
4.17 シミュレーション 6 のデッドラインオーバー率 39
4.18 シミュレーション 6 の平均応答時間 40
4.19 シミュレーション 6 の遅れタスクの静的優先度平均 41
4.20 シミュレーション 7 のデッドラインオーバー率 43
4.21 シミュレーション 7 の平均応答時間 44
4.22 シミュレーション 7 の遅れタスクの静的優先度平均 45

表 目 次

2.1 タスクセット 9
2.2 オフラインスケジューリングの計算時間 [18] 11

3.1 負荷の指標と平均応答時間の関係 [9] 14

4.1 負荷計算のオーバヘッド 19
4.2 スケジューリングのオーバヘッド 20

第1章　はじめに

1.1　研究の背景と目的

一般の PC やワークステーションで実行されるほとんどのプログラムにとって，プログラムの正しさ (correctness) は，計算の論理的な結果だけに依存する．それに対し，リアルタイムシステムでは，プログラムの正しさが論理的結果だけではなく結果が生成される時間にも依存する [17], [19]．つまり，決められたデッドライン (deadline, 締切時刻) までにタスク処理を完了することが重要である．

近年の急速な技術革新によって高性能なコンピュータシステムをより低コストで実現することが可能となり，それと共にリアルタイムシステムの適用分野が広がってきた．現在では，身近な通信機器・オフィス機器・家電製品など様々な分野でリアルタイムシステムの需要が高まっている．そのような流れの中で，かつては実現困難であった分散リアルタイムシステムの構築が可能になってきた．特に，ネットワークの普及にともない，社会系インフラシステム (航空管制システム，交通制御システム等) ・制御系システム (工場におけるロボット制御，機器監視システム等) ・マルチメディア系システム (Video on Demand 等) の分散処理化が注目を集めつつある [5]．

単一プロセッサ上のリアルタイムスケジューリングは長年にわたって研究されており，タスクのデッドラインを守るために様々なアルゴリズムが提案されている．それに対して，マルチプロセッサのシステムにおけるスケジューリングでは，最適性を求めることが困難な場合が多い [18]．特に，システム稼働中にスケジューリングを行なうオンラインスケジューリングの場合，発見的 (heuristic) 手法に頼らざるを得ない状況にある．

本研究では，複数のプロセッサを持つ分散リアルタイムシステムにおいて，より効果的な発見的スケジューリングを実現することを目的とする．そのために，タスクの優先度を考慮した新たなプロセッサ割り当ての手法を提案する．

1.2　本論文の構成

本論文の構成を以下に示す．

第 2 章では，リアルタイムスケジューリングについて説明する．特に，単一プロセッサシステムにおけるスケジューリングとマルチプロセッサシステムにおけるそれとの違いを説明する．

第 3 章では，本研究の提案と方法について説明する．

第 4 章では，提案方式の評価や従来方式との比較を行なう．
第 5 章では，関連研究を紹介する．
第 6 章では，まとめを行なう．

第2章　リアルタイムスケジューリング

本章では，まず 2.1 節でリアルタイムタスクの属性について説明する．次いで 2.2 節でリアルタイムスケジューリングに関して述べる．2.3 節では，本研究でターゲットとするマルチプロセッサシステムにおけるスケジューリングについて，研究の現状を述べる．

2.1　リアルタイムタスクの属性

リアルタイムタスクは以下のような時間属性を持つ (図 2.1).

- **起動要求時刻 (release or request time) または到着時刻 (arrival time) (a)**
  タスクに実行要求が出され実行可能になった時刻.
- **計算時間 (computation time) または実行時間 (execution time) (C)**
  プロセッサがタスクを中断することなく実行するために必要な時間.
- **絶対デッドライン (absolute deadline) (d)**
  タスクの実行が完了していなければならない締切時刻.
- **相対デッドライン (relative deadline) (D)**
  $D = d - a$.
- **開始時刻 (start time) (s)**
  タスクが最初に実行を開始した時刻.
- **終了時刻 (finishing time) (f)**
  タスクの処理が完了した時刻.
- **遅れ時間 (lateness) (L)**
  $L = f - d$ と定義され，デッドラインより前に処理が終了した場合は負になる.
- **超過時間 (tardness or exceeding time) (E)**
  $E = \max(0, L)$．デッドライン後にタスクが処理を実行している時間.

- **余裕時間 (laxity or slack time) (X)**

  $X = D - C$. 処理の開始が遅れてもデッドラインを越えない最大の時間.

- **周期 (period) (T)**

  同じタスクが複数回起動されるときの起動間隔.

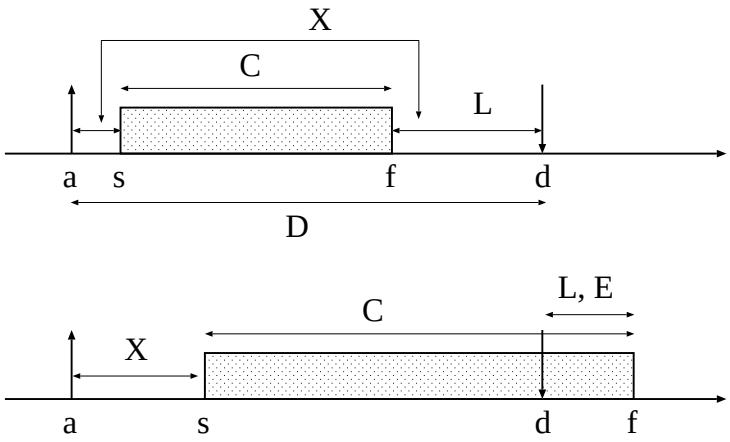


図 2.1: タスクの時間属性

一般に，タスクは起動の規則性によって周期的 (periodic)，非周期的 (aperiodic)，偶発的 (sporadic) の 3 種類に分けられる．周期タスクは，一定の周期 T に従って規則的に起動する．起動された個々のタスクをジョブ (job) またはインスタンス (instance) と呼ぶ．また，初回の起動時刻をフェーズ (phase) と呼び，Φ で表わす．非周期タスクは，再発するが規則的ではない．偶発的タスクは，起動時刻を予期できないので非周期タスクの一種と考えることができる．ただし，最小起動間隔が決まっている点が異なる．

偶発的タスクを非周期タスクに含めて，単純に周期タスクと非周期タスクとに分類している文献も多い．本論文では，再発するか否かに関係なく，規則的 (周期的) でないタスクをすべて非周期タスクと呼ぶ.

2.2 リアルタイムスケジューリング

2.2.1 ハードリアルタイムとソフトリアルタイム

リアルタイム性には，大きく分けて次の 2 種類がある．*

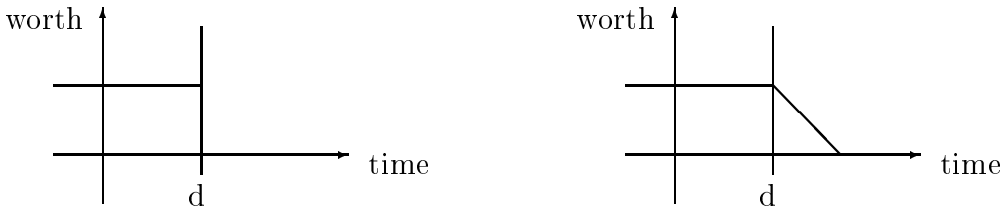
*さらにファームリアルタイム (firm real-time) という分類が加わることもある．その定義の場合，デッドラインを過ぎたタスクの価値がファームリアルタイムではゼロになり，ハードリアルタイムでは $-\infty$ となる．デッドラインを過ぎたファームリアルタイムタスクの処理を実行するのは無駄であり，場合によってはシステムを停止する必要があるかもしれないが，その結果はシステムに損害を与えるほど致命的ではない.

- ハードリアルタイム (hard real-time)
- ソフトリアルタイム (soft real-time)

ハードリアルタイムの場合，タスクは時間制約を必ず守らなければならず，少しでもデッドラインを過ぎてはならない．ハードリアルタイムシステムにおいて，あるタスクがデッドラインオーバーを起こした場合，そのタスク (またはシステム) の価値はゼロになる．(図 2.2(a)) システムに損害を与える可能性があるので，システムを緊急停止する措置がとられることもある．ハードリアルタイムシステムの例としては，航空宇宙・自動車関連の機器や，発電所などが挙げられる．

ソフトリアルタイムの場合，過負荷状態の時などはデッドラインを多少過ぎても許容される．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，システム自体に損害が及ぶことはない．(図 2.2(b)) ソフトリアルタイムシステムの例としては，電話交換器や家電製品などが挙げられる．

システム全体がハードリアルタイムかソフトリアルタイムか，と決めることもできるが，現実的には一つのシステム内に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が混在することもある．



(a) ハードリアルタイム　　(b) ソフトリアルタイム

図 2.2: リアルタイム性と価値関数

2.2.2 スケジューリングアルゴリズムの分類

スケジューリングアルゴリズムの分類法にはいくつかある．ハードリアルタイムかソフトリアルタイムかはその一つであるが，それ以外の主なものを以下に挙げる．

- **最適 (optimal)，発見的 (heuristic)**
  スケジュール評価関数 (4.4 節参照) の最適値 (通常は最小値) を与えるスケジューリングを，最適であるという．それに対して，最適性を保証できないものの，実用に堪える程度のスケジューリングを発見的であるという．

- **オフライン (off-line)，オンライン (on-line)**
  システム稼働前 (例えばコンパイル時など) にスケジューリングを済ませておく方式をオフライン方式という．事前に全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，柔軟性にも乏しい．しかし，最適なスケジューリングを行なうことが可能であり，実行時のオーバヘッドが少ないという特長がある．
  それに対して，オンライン方式ではシステム稼働中にスケジューリングを行なう．具体的にはタスクの起動時と終了時にそれぞれスケジューリングが行なわれる．この方式の特徴はオフライン方式のそれと反対で，柔軟性はあるが最適なスケジューリング結果を得ることが難しく，実行時のオーバヘッドが大きくなる．

- **静的 (static)，動的 (dynamic)**
  タスクの優先度が静的か動的かという意味の分類である．静的優先度 (固定優先度 (fixed priority) とも呼ばれる．) とは，一度決められたら変化しないタスク優先度のことである．それに対して，タスクの時間属性などに基づいて変化する可能性があるのが動的優先度である[†]．

- **プリエンプティブ (preemptive)，ノンプリエンプティブ (non-preemptive)** [‡]
  タスク A が処理を実行している途中，さらに高い優先度を持つタスク B が起動された場合に，タスク A の処理を中断してタスク B に処理が移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，タスク B の優先度がどれほど高くてもタスク A の処理が完了するまで実行権が移動しないなら，ノンプリエンプティブであるという[§]．単一プロセッサにおいては，プリエンプティブなほうが最適なスケジューリングを見つけやすい．

2.2.3 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分からなければデッドラインを保証するスケジューリングは不可能なので，最長の実行時間を見積もることが行なわれる．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 WCET (Worst-Case Execution Time, 最悪実行時間) と呼ぶ．

[†]なお，動的/静的スケジューリングという言葉がそれぞれオンライン/オフラインスケジューリングの意味で用いられることもあるので注意が必要である．

[‡]preemptive という言葉は「先取権がある」「中断可能である」「横取りできる」などと訳される．

[§]ちなみに，優先度の高いタスクが優先度の低いタスクから実行権を奪うことを「プリエンプトする」というが，その反対を「ブロック (block) する」という．

2.2.4 スケジューリングアルゴリズム

単一プロセッサにおけるリアルタイムスケジューリングは従来から広く研究されており，様々な手法が提案されている．一般に，リアルタイムシステムでは優先度に基づくスケジューリングが行なわれるが，タスクの優先度を決定する方法には主に次の 2 種類がある．

- 設計者が事前に静的優先度を設定する
- タスクの属性に応じて優先度を決定する

前者については議論の余地が少ないが，後者については実際に数多くのアルゴリズムが提案されており，条件によっては最適性が証明されているものも存在する．“スケジューリングアルゴリズム” と呼ぶことが多いが，正確にはタスクの優先度付け方法である．ここでは，その代表的なものを紹介する．

RM (Rate Monotonic) [12]
周期タスク用のアルゴリズムであり，周期と相対デッドラインが等しい ($T = D$) という条件の下，周期の短いタスクの優先度を高く設定する．他の条件として，タスクは互いに独立で順序制約がないこと，資源制約がないこと，プリエンプティブであることが必要である．
このアルゴリズムは，オンライン・静的である．また，RM 法を用いてスケジュールできない周期タスクセットは，他のアルゴリズムを用いてもスケジュールできないことが証明されており，その意味で最適である．

EDF (Earliest Deadline First) [12]
絶対デッドラインの早いタスクに高い優先度を与える方法．Deadline Driven Scheduling とも呼ばれる．非周期タスクも扱える．前提条件は，タスクは互いに独立で順序制約がないこと，資源制約がないこと，そしてプリエンプティブであることである．
このアルゴリズムは，オンライン・動的である．スケジュール可能なタスクセットであれば必ず EDF でスケジュールできること，最大遅れ時間 (4.4 節参照) を最小にすることがそれぞれ証明されており，最適スケジューリングであるといえる．

LLF (Least Laxity First) [14]
余裕時間の短いタスクに高い優先度を与える方法．前提とする条件や，分類 (オンライン・動的・最適) は，EDF 法の場合と同じである．
LLF 法も最適スケジューリングであると言われているが，余裕時間は時々刻々と変化するので，厳密に実現しようとするとスケジューリングやコンテキスト切り替えが頻発しオーバヘッドが大きくなるという欠点がある．

DM (Deadline Monotonic) [11]

RM 法を拡張した方法であり，周期ではなく相対デッドラインに基づいてタスク優先度を決める方法．RM 法では $T = D$ という前提条件があったが，実際のシステムでは次の起動要求時刻より先にデッドラインを迎えることがある．そのような状況，つまり $T \geq D$ であることが，DM 法の条件である．他の条件，分類 (オンライン・静的・最適) は RM 法の場合と同じである．

FCFS (First Come First Service)

タスクを到着順に処理していく方法．リアルタイムシステムのためのスケジューリング方法というわけではないので，デッドラインについては考慮されていない．
実用の RTOS で静的優先度に基づくスケジューリングが行なわれていて，同じ優先度を持つタスクが複数個 到着したとき，FCFS で実行されるのが基本である．

適応型動的優先度 (ADP, Adaptive Dynamic Priority) [10]

静的優先度を基準とし，これに動的要因を導入することによって優先度を更新する優先度付け方法．アプリケーション開発者が各タスクの重要度，および実行順序を考慮して設定した値は重要であるため，静的優先度を基準と考え，動的要因として，タスクの余裕時間と周期タスクの場合の周期を取り入れている．余裕時間は LLF に従い，余裕時間に応じて優先度を高めに変更する．周期タスクに関しては RM に従い，周期に応じて優先度を高めに変更する．動的優先度 $Pri_{dynamic}$ は次式で表わされる¶.

$$Pri_{dynamic} = Pri_{static} - (\frac{\gamma}{T} + \frac{\delta}{L})$$

$$\begin{aligned} Pri_{static} &: \text{静的優先度} \\ T &: \text{周期} \\ L &: \text{余裕時間} \\ \gamma, \delta &: \text{定数} \end{aligned}$$

また，ADP 法では余裕時間の見積もりに関し，WCET ではなく PET (Predictive execution time) を用いる．PET とは，タスクの実際の実行時間に基づき動的に変動する時間である．各タスクはシステム起動後に複数回 実行される可能性があるため，実行毎にタスクの実行時間に基づいて PET の再計算を行い，その結果が次回の実行時間の見積もりに反映される．この計算はタスクの終了時に行なわれ，終了したタスクの実行時間と前回の PET の値の加重平均から算出する．これにより，経験的に実際の実行時間を反映する PET の値を得る．あるタスクの i 回目の実行終了時の PET の再計算は以下の式で与えられる．ただし，β は定数である．

$$PET_i = \beta \times Pri_{i-1} - (1 - \beta) \times (last\ execution\ time)$$

¶ $Pri_{static}, Pri_{dynamic}$ は値が小さいほど優先度が高い.

2.3 マルチプロセッサ上のスケジューリング

マルチプロセッサを持つリアルタイムシステムは需要が高まってきているが，単一プロセッサ上でのスケジューリングに比べると，マルチプロセッサ上のスケジューリングに関する知見は少ない．しかも，分かっていることは否定的なことが大半である．本節で述べるように，単一プロセッサ上のスケジューリングでは成立したことがマルチプロセッサの場合に成立しないことも多い．

2.3.1 オンラインスケジューリング

マルチプロセッサの場合，オンライン方式で最適なスケジューリングは得られないことが知られている [14], [18].

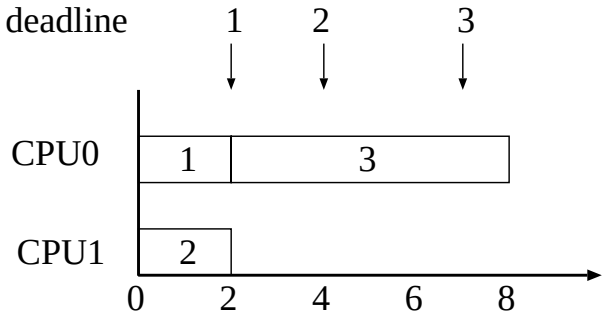
定理 1
2 個以上のプロセッサを持つシステムにおいて，タスクのデッドライン・実行時間・起動要求時刻が事前に分からない場合 (つまりオンラインスケジューリングの場合)，デッドラインに基づくスケジューリングアルゴリズムで最適なものは存在しない．

例えば，単一プロセッサでは最適であった EDF 法も，マルチプロセッサの場合は最適ではない [14]．例として，表 2.1 のようなタスクのスケジューリングを考える．プロセッサが二つあるとして EDF 法を用いると，図 2.3(a) のようになり，タスク 3 がデッドラインをオーバーする．同じ条件下でも，図 2.3(b) のようにスケジューリングすれば，全てのタスクのデッドラインが守られる．

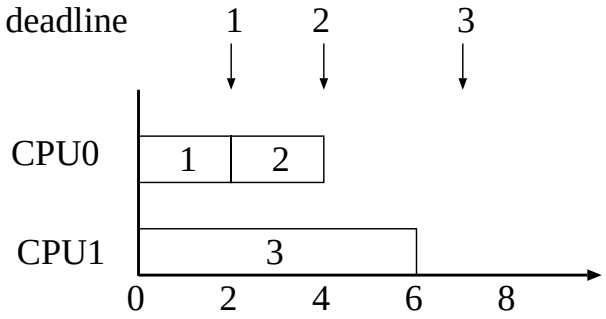
表 2.1: タスクセット

	1	2	3
a	0	0	0
C	2	2	6
d	2	4	7

このように最適スケジューリングが得られないので，発見的手法に頼らざるを得ない．いくつかの発見的手法が提案されているが，EDF 法のような単純な方法に比べると，スケジューリングのための計算が複雑になるという問題がある．また，確率的挙動を定量化できていないので，デッドラインの保証が要求される場面では使えない [6].



(a) EDF 法の場合



(b) 最適な例

図 2.3: マルチプロセッサでの EDF の使用

2.3.2 オフラインスケジューリング

ノンプリエンプティブなオフラインスケジューリングについての研究成果を表 2.2 に示す．この表によると，ほとんどの条件下でスケジューリング問題が NP 完全[‖]であることが分かる．なお，資源制約 (resource constraints) とは，複数のタスクによって使用される共有資源が存在し，したがって排他アクセスを実現しなければならないという制約である．また，順序制約 (precedence constraints) とは，タスク間に順序関係 (依存関係) があり，あるタスクが終了するまではそのタスクに依存する別のタスクが開始できないという制約である．表中の「順序関係が木構造」とは，あるタスクが依存しているタスクは高々一つ，逆に，あるタスクに依存しているタスクは複数存在しうる状況を指す．また,「タスクの実行時間が 1 または 2 単位」とは，全てのタスクで実行時間が同一，もしくはその 2 倍の実行時間を持つタスクが含まれてもよいことを意味する．

表 2.2: オフラインスケジューリングの計算時間 [18]

CPU 数	資源制約	順序制約	タスクの実行時間	計算量
2	なし	任意	同一	多項式時間
2	なし	なし	任意長	NP 完全
2	なし	任意	1 または 2 単位	NP 完全
2	あり	木構造	同一	NP 完全
3	あり	なし	同一	NP 完全
N	なし	木構造	同一	多項式時間
N	なし	任意	同一	NP 完全

Stankovic は，タスクの実行時間が同一でないことと，資源制約の存在が，直ちに問題を NP 完全にすると述べている [18].

2.3.3 その他の問題

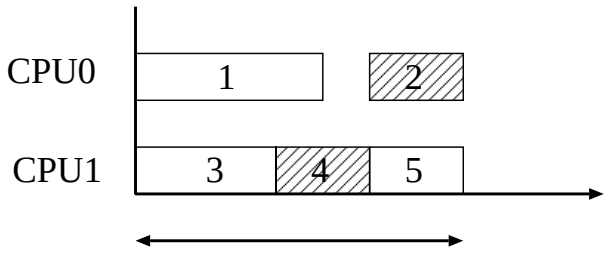
単一プロセッサではノンプリエンプティブよりプリエンプティブのほうが最適なスケジューリングを見つけやすいことは先に述べた．それに対してマルチプロセッサの場合，条件によってはプリエンプティブスケジューリングに優位性がなくなることが知られている [18], [13].

単一プロセッサ上のスケジューリングにはない，もう一つの問題は，Richard's anomalies と呼ばれるものである．

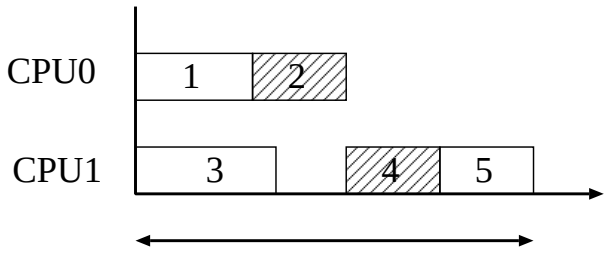
[‖]NP 完全というクラスに属する問題を解くには，指数関数的な時間が必要であると考えられている．しかし，このクラスに属する問題が一つでも多項式時間で解けたら，全ての NP 完全問題が多項式時間で計算できると予想されている．

定理 2 [7]
マルチプロセッサ上のスケジューリングにおいて，プロセッサ数を増やす，タスク実行時間を短くする，順序制約を緩くするなどの措置によって，全てのタスクを実行するのにかかる時間が長くなることがある．

図 2.4 に例を示す．タスク 1, 2 はプロセッサ 0 に，タスク 3, 4, 5 はプロセッサ 1 に，それぞれ割り当てられることが決まっているとする．また，タスク 2 と 4 は共通の資源を排他的に使用するとする．図 2.4(a) でタスク 2 が開始できずに待たされているのは，タスク 4 が資源を占有しているからである．ここで，タスク 1 の実行時間を短くすると，図 2.4(b) のように全体の処理時間が長くなる．



(a) 最適なスケジューリング例



(b) タスク 1 の実行時間を短くした場合

図 2.4: Richard's anomalies の例

特にハードリアルタイムシステムでは WCET を大きく見積もる傾向があるが，通常，タスクは WCET よりも早く終了する．このときオンラインスケジューリングによって次のタスクを開始すると Richard's anomalies に陥る可能性がある．したがって，タスクが早く終了しても，WCET が経過するまでプロセッサをアイドル状態にするなどの対策が必要である．

2.3.4 問題の分割

以上のように，マルチプロセッサ上のスケジューリング問題を解くのは難しい．しかし，単一プロセッサでのスケジューリングなら，比較的 簡単である．このことから，問題を次の 2 段階に分割する方法がある．

1. タスクをプロセッサに割り当てる
2. 各プロセッサ毎に，割り当てられたタスクをスケジューリングする

ただし，タスク割り当てもほとんどの場合 NP 困難な問題であり [3],[11]，発見的手法が多く提案されている．

負荷均衡 (utilization balancing)

タスク起動の度に，その時点で負荷が最小のプロセッサに割り当てるなどして，プロセッサ同士の負荷均衡を図る方法.

箱詰め問題の解法の応用

大きさが異なってもよい複数の品物を，容量が有限の複数の箱 (bin) にいかにして詰め込むかというアルゴリズムの問題があり，箱詰め問題 (bin-packing problem) という．ここで，品物をタスク，箱をプロセッサとみなすと，タスク割り当てに応用できる．

箱詰め問題は NP 完全問題の一つであり，最適解を多項式時間で得ることは難しい．RM 法と組み合わせたスケジューリング手法などが提案されている [4].

第3章　負荷の算出

3.1　負荷の指標

本研究では，マルチプロセッサにおける，オンライン方式の発見的スケジューリングを対象としている．2章で説明した，タスク割り当てと個々のスケジューリングを切り分けて扱う方法を採用する．特に，リアルタイムシステムにおいて，負荷分散によるデッドラインオーバーの減少を図る．

発見的な負荷均衡を行なうとき，負荷の指標がどのような影響を及ぼすかに関する研究がある [9]．この研究では，homogeneous な Unix ワークステーションを接続して分散環境を構築し，負荷の指標と平均応答時間の関係を調べている．各ワークステーションは，負荷などの情報を定期的に (8 秒毎に) ブロードキャストし，スケジューラはその情報に基づいてプロセス割り当てを行なう．

その研究結果を表 3.1 に示す.「1 分間の負荷平均」とは，実行キューのタスク数を過去 1 分間に計測した結果の平均値である．この研究ではリアルタイムシステムをターゲットにしているわけではないが，この表から，負荷の指標の違いがシステム全体のパフォーマンスに影響を及ぼすことが分かる．したがって，負荷の指標の選択は重要である．

表 3.1: 負荷の指標と平均応答時間の関係 [9]

負荷の指標	平均応答時間 [sec]
実行キューのタスク数	13.576
システムコールの呼出し率	14.423
CPU コンテキストスイッチの割合	14.747
CPU のアイドル時間 [%]	15.692
空きメモリ容量	16.710
1 分間の負荷平均	17.932

3.2 適応型動的優先度を用いた負荷計算法

負荷の指標として一般によく用いられるのは，主に以下のものである．

- CPU 使用率
- タスク (プロセス) の数
- Ready タスクの数

上記の指標は有用であるが，リアルタイムシステムでの使用を前提としていない．そこで，本研究では，適応型動的優先度法の概念を取り入れ，よりリアルタイムシステムに適した負荷の指標を提案する．特徴を以下に示す．

1. 実用の RTOS では静的優先度に基づくスケジューリングが広く行なわれていることから，負荷の指標としてタスクの静的優先度を導入する
2. リアルタイムシステムでは，タスクのデッドラインを保証することが重要であるという点に着目し，負荷の指標としてタスクの余裕時間を導入する

図 3.1 は，二つのプロセッサにそれぞれ二つのタスクが割り当てられている状況を示している．プロセッサ 0 と 1 を比較した場合，CPU 使用率やタスク数は全く同様であるが，余裕時間の長さが異なる．余裕時間が短いタスクを抱えるプロセッサ 1 のほうが，負荷が高いとみなすのが，本研究の提案である．(ただし，厳密にはタスクの静的優先度も考慮する．)

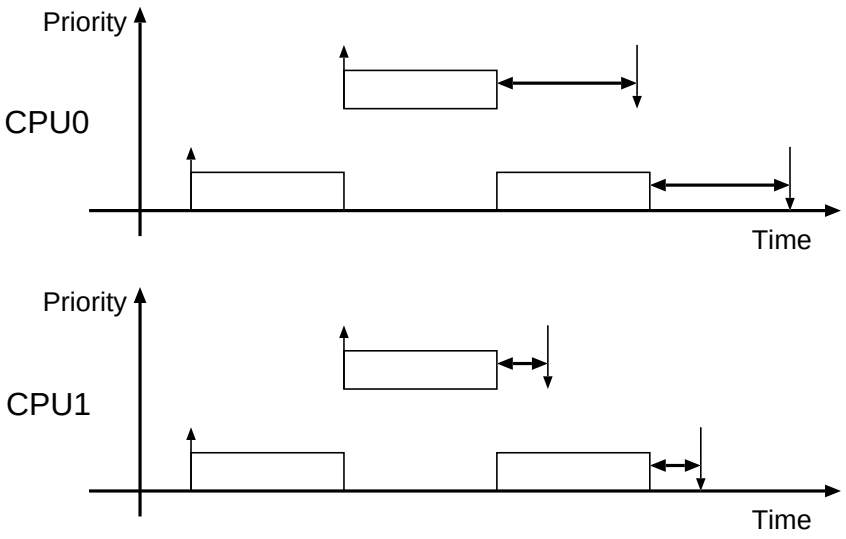


図 3.1: 余裕時間の違いによって負荷が異なる例

負荷の算出式を次に示す.

$$\rho_i = \frac{1}{M_i} \sum_{k=1}^{N} (\frac{P_k}{P_{lv}} + \frac{c_k}{X_k})$$

ρ : プロセッサの負荷
M : プロセッサの相対能力
P : タスクの静的優先度
P_{lv} : 優先度レベル
c : タスクの残り実行時間
X : タスクの余裕時間
i : プロセッサ番号
k : タスク番号
N : プロセッサ上のタスク数

この式では，P_k/P_{lv} の項で静的優先度 P_k によりそのタスクがどの程度 重要か，を基準にとる*．c_k/X_k の項で，余裕時間を用いて値を調整する．余裕時間が少ないほど，この項の値は大きくなる．分子に定数ではなく残り実行時間 c を与えたのは，次の理由による．

- タスクの余裕時間 X と実行時間 C が共に長いタスクの場合，プロセッサを長時間占有するのにその負荷を過小評価してしまう可能性があるので，それを防ぐため
- 項全体を無次元にそろえるため

これらを各タスク毎に計算して合計をとるので，タスク数に応じて計算結果が大きくなる傾向がある．つまり，この式はタスク数もある程度 反映している．

最後にプロセッサの相対能力 M で割る．分散システムでは，プロセッサの処理能力がそれぞれに異なる可能性がある．例えば，処理能力に 2 倍の差がある二つのプロセッサに，同じタスクセットが割り当てられているとする．速いほうのプロセッサは，遅いほうのプロセッサが処理を完了するのに必要な時間の半分の時間で処理を終える．残り半分の時間はアイドル状態であり，負荷はゼロである．CPU の相対能力で割ることによって，能力差を吸収した負荷値を算出できる．

非リアルタイム OS である Unix などの場合，CPU 使用率とは，ある期間のうち CPU がアイドル状態ではなかった時間の割合を指すことが多い．つまり，過去から現在までの情報に基づく指標である．それに対して提案方式では，余裕時間などの，現在から未来に関する情報に基づいて計算している点がリアルタイムシステムに適している．

*ここでは数値が大きいほうが優先度が高いとする．逆の場合は分子を $P_{lv} - P_k$ にすればよい．

第4章　評価

4.1　システムモデル

本評価で想定する分散リアルタイムシステムの構成モデルについて説明する．模式図を図 4.1 に示す.

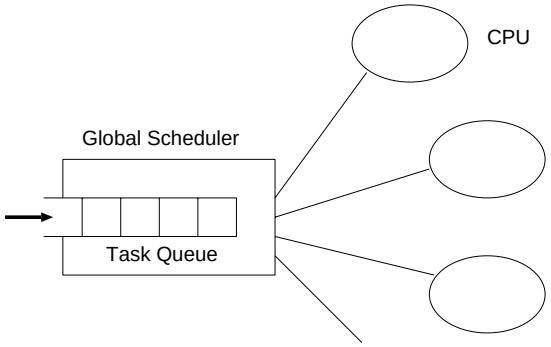


図 4.1: システムモデルの模式図

4.1.1　プロセッサとスケジューリング

システムは一つ以上の (通常は複数の) プロセッサから成る．各プロセッサの処理能力は，均等でも不均等でもよい．ただし，タスクがどのプロセッサに割り当てられても実行できるよう，バイナリレベルで互換性を持つものとする.

各プロセッサにタスクを割り当てるグローバルスケジューラ (以下，GS と表記する) が，システムに一つだけ存在する．GS は，システム内で最高速のプロセッサと同じ速度で動作するものとする．GS と各プロセッサは大域時刻 (global clock) に同期して動作する．このシステムは同期システム (synchronous system) を仮定する [8].

全てのタスク起動要求は GS が受け取り，GS はスケジューリングを実行した後，選択したタスクをプロセッサの一つに送信する．一つのタスクを送信するのに GS は 1 クロック消費するとする．言い換えると，GS は 1 クロックにつき 1 タスクしか送信できない.

GS は各プロセッサとの通信路を持っており，タスクやその他の情報を交換する．送受信されるデータのサイズや量にかかわりなく，全ての通信は固定時間で行なわれる．ただ

し，その時間は各プロセッサ毎に異なっていてもよい．

各プロセッサでは任意のアルゴリズムに基づいてオンラインスケジューリングが行なわれる．タスクの実行は，デッドラインを過ぎたかどうかにかかわりなく，処理が完了するまで継続される．(つまり，ソフトリアルタイムである．)

各プロセッサはタスクが到着したとき，終了したときにスケジューリングを行なうが，それと同時に負荷計算を行なう．本シミュレータでは，負荷計算を先に行なって結果を GS へ送信し，次いでスケジューリングを実行する．GS は受け取った最新の情報を参照して，負荷が最小のプロセッサにタスクを送信する．

4.1.2 タスクセット

タスクセットは基本的に周期タスクから成るが，非周期タスクも含んでよい．各々のタスクは独立した処理の単位であり，その間に依存関係，順序関係はないものとする．また，資源制約についても考慮しない．プロセスが一つ以上のスレッドで構成されるのと同様に一つのタスクをサブタスクに分割して取り扱う研究もあるが，本研究ではタスクを唯一かつ最小の処理単位とする．

タスクの静的優先度・実行時間・相対デッドライン・周期などは，それぞれ任意とする．タスク実行時間は，実行時の条件によって，起動する度に異なるのが通常である．しかし，本評価では WCET 分の CPU 時間を各タスクに費やすことにする．

4.2 分散リアルタイムスケジューリングシミュレータ

本研究では，ANSI C を用いてスケジューリングシミュレータを実装した．シミュレーション実行時に選択可能な条件について主なものを以下に説明する．

負荷の指標

プロセッサの負荷を示す指標として，タスク数，提案方式から選択する．また，リアルタイムスケジューリングの研究分野で広く用いられている CPU 使用率 $\sum(C/T)$ も採用する．ただし，この式は周期タスクに限定されるので，非周期タスクに対応するために次式を用いる [2], [6].

$$\begin{aligned} \rho_i &= \frac{\sum_{d_k \le d_i} c_k}{d_i - t_a} \\ \rho &= \max(\rho_i) \end{aligned}$$

ここで，t_a は最新のタスク起動要求時刻である．まず，ρ_i を各タスク毎に計算する．$\sum_{d_k \ge d_i} c_k$ は，そのタスクと，そのタスクより早い絶対デッドラインを持つタスクの，残り実行時間の合計である．各タスクの負荷 ρ_i の最大値を，そのプロセッサの負荷とする．CPU 使用率 $\sum(C/T)$ と区別するため，本論文では負荷 ρ を CPU 負荷率と呼ぶ．

スケジューリングアルゴリズム (タスクの優先度付け方法)

タスクの優先度付け方法としては，静的優先度に基づく方法・FCFS・EDF・LLF・RM・DM から任意に選択できるものとする．ただし，静的優先度に基づく方法でスケジューリングを行なっていて同じ優先度を持つタスクが複数個 到着したときは，FCFS で実行することとする．なお，使用するアルゴリズムは，GS や各プロセッサで異なってもよい．

他に変更可能なオプションとして以下のものを実装した．

- プロセッサ数
- プロセッサの相対能力
- プロセッサと GS 間の通信時間
- シミュレーション終了時刻

4.3 各オーバヘッド

スケジューリングと負荷計算には，それぞれ方式やタスク数に応じてオーバヘッドが発生する．

本研究では，負荷計算のオーバヘッドを決めるために，シミュレータプログラムの該当部分を sparc[16] アセンブリ命令にコンパイルしてその命令数を方式毎にカウントした．タスク 1 個の場合に実行される命令数を基に，タスクが n 個の場合に実行される命令数を算出した．結果を表 4.1 に示す．この結果を目安として，シミュレーション時のオーバヘッドを決定した．

表 4.1: 負荷計算のオーバヘッド

方式	計算式	オーダー	命令数	実装オーバヘッド
タスク数	—	$O(1)$	—	10
CPU 使用率	$\sum(C/T)$	$O(n)$	$49 \times n$	$5 \times n$
CPU 負荷率	$\max(\frac{\sum_{d_k \le d} c_k}{d - t_a})$	$O(n^2)$	$(22 \times \frac{n}{2} + 50) \times n$	$(n+5) \times n$
提案方式	$\frac{1}{M}\sum(\frac{P}{P_{lv}} + \frac{c}{X})$	$O(n)$	$87 \times n + 14$	$9 \times n + 1$

また，スケジューリングのオーバヘッドについては表 4.2 に示す．ただし,「静的優先度に基づく方法」を HPF と表記する*.

どのオーバヘッドも，命令数をそのままシミュレーション時のクロック数として用いず，方式ごとの比率を保ったまま，タスクセットの時間属性に応じて定数部分を変更した．

*Highest Priority First の略．静的優先度に基づくスケジューリング法を指す言葉ではないが，便宜上この表記を用いる．

表 4.2: スケジューリングのオーバヘッド

方式	オーダー	実装オーバヘッド	備考
FCFS, HPF	$O(1)$	10	タスクがキューに優先度順に並んでいるならば，キューの先頭タスクを選ぶのは固定時間で可能.
EDF, RM, DM	$O(n)$	$4 \times n$	キューをたどって最高優先度のタスクを探すので，タスク数に比例した時間が必要.
LLF	$O(n)$	$10 \times n$	上記と同様だが，タスク毎に余裕時間を計算する必要があるので，オーバヘッドは大きくなる.

4.4 スケジュール評価関数

スケジューリング手法の性能を評価するために，スケジュール評価関数 (cost function) を用いる．デッドラインオーバー数や平均応答時間が主に用いられるが，これらを含めてスケジュール評価関数の例を次に示す [1].

平均応答時間 (average response time):

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

全計算時間 (total completion time):

$$t_c = \max_i(f_i) - \min_i(a_i)$$

重み付き終了時間 (weighted sum of completion times):

$$t_w = \sum_{i=1}^{n} w_i f_i$$

ここで，w_i はタスク毎に適当に決めた重みである．本研究では，静的優先度を重みとする.

最大遅れ時間 (maximum lateness):

$$L_{max} = \max_i(f_i - d_i)$$

最大遅れタスク数 (maximum number of late tasks):

$$N_{late} = \sum_{i=1}^{n} miss(f_i)$$

ただし,

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

本研究ではタスクの静的優先度を重視しているため，その点を考慮したスケジュール評価関数として「遅れタスクの静的優先度平均 $\bar{t_p}$」を新たに提案する.

$$\bar{t_p} = \frac{1}{n} \sum_{i=1}^{n} miss(P_i)$$

ただし,

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

また，本研究ではシミュレーションの観測時間を固定とするため，終了するタスクの数が条件によって異なる．単純にデッドラインオーバーの数 N_{late} での比較はできないので，公平な評価を行なうために，次のデッドラインオーバー率 R_{late} を定義する.

$$\begin{aligned} R_{late} &= \frac{N_{late}}{N_{comp}} \\ N_{comp} &: \text{終了タスクの数} \end{aligned}$$

4.5 シミュレーション条件

シミュレーション実行時の条件について述べる．一部の条件を変更して複数回シミュレーションを行なったが，その共通の条件を以下に示す.

タスクセットは，以下の条件を満たすように定義した.

- 静的優先度 P: 64 段階 (最低優先度:0, 最高優先度:63)
- タスク実行時間 C: 100〜1000 クロック
- タスク実行時間 C/相対デッドライン D = 10〜50%
- 相対デッドライン D は最低 1000 クロック
- 周期 $T \geq$ 相対デッドライン D

- 周期タスクの起動回数は無限
- 非周期タスクは最低 10 回起動

その他の共通条件は，以下のとおりである．

- 観測時間: 1000000 クロック
- GS との通信時間は全てのプロセッサで同一
- スケジューリングアルゴリズムは全てのプロセッサで同一 (GS が使用するアルゴリズムとは異なってもよい)

以下の条件でそれぞれシミュレーションを行なった．

シミュレーション 1

- プロセッサ数: 36 個
- タスクセット: 周期タスクのみ 200 個
- プロセッサの処理能力: 同一
- 通信，スケジューリング，負荷計算のオーバヘッド: なし

シミュレーション 2

- シミュレーション 1 にスケジューリングオーバヘッドを追加

シミュレーション 3

- シミュレーション 2 に負荷計算のオーバヘッドを追加

シミュレーション 4

- シミュレーション 3 に通信オーバヘッドを 500 クロック追加

シミュレーション 5

- シミュレーション 4 の条件で，タスク 60 個 (タスクセットの 30%) を非周期タスクに変更

シミュレーション 6

- シミュレーション 5 の条件で，プロセッサを 48 個に増やし，うち半分の 24 個の処理能力を 5/6 倍に変更

シミュレーション 7

- シミュレーション 5 の条件で，プロセッサを 64 個に増やし，うち 24 個の処理能力を 2/3 倍に変更

ただし，非周期タスクを含むシミュレーション 5〜7 では，周期タスク用のスケジューリング法 (RM, DM) と負荷の指標 (CPU 使用率) を使用しない．

4.6 シミュレーション結果

各シミュレーションについてタスクセットを 2 種類用意し，結果の平均を取った．シミュレーション結果を図 4.2〜4.22 に示す．デッドラインオーバー率，平均応答時間，遅れタスクの静的優先度平均について評価した．比較のために四つの方式を並べて示す．

なお，横軸には GS が用いたスケジューリングアルゴリズムと各プロセッサが用いたそれとを示す．例えば「EDF x LLF」と表記してある場合，GS は EDF 法で，各プロセッサは LLF 法でスケジューリングしたことを意味する．

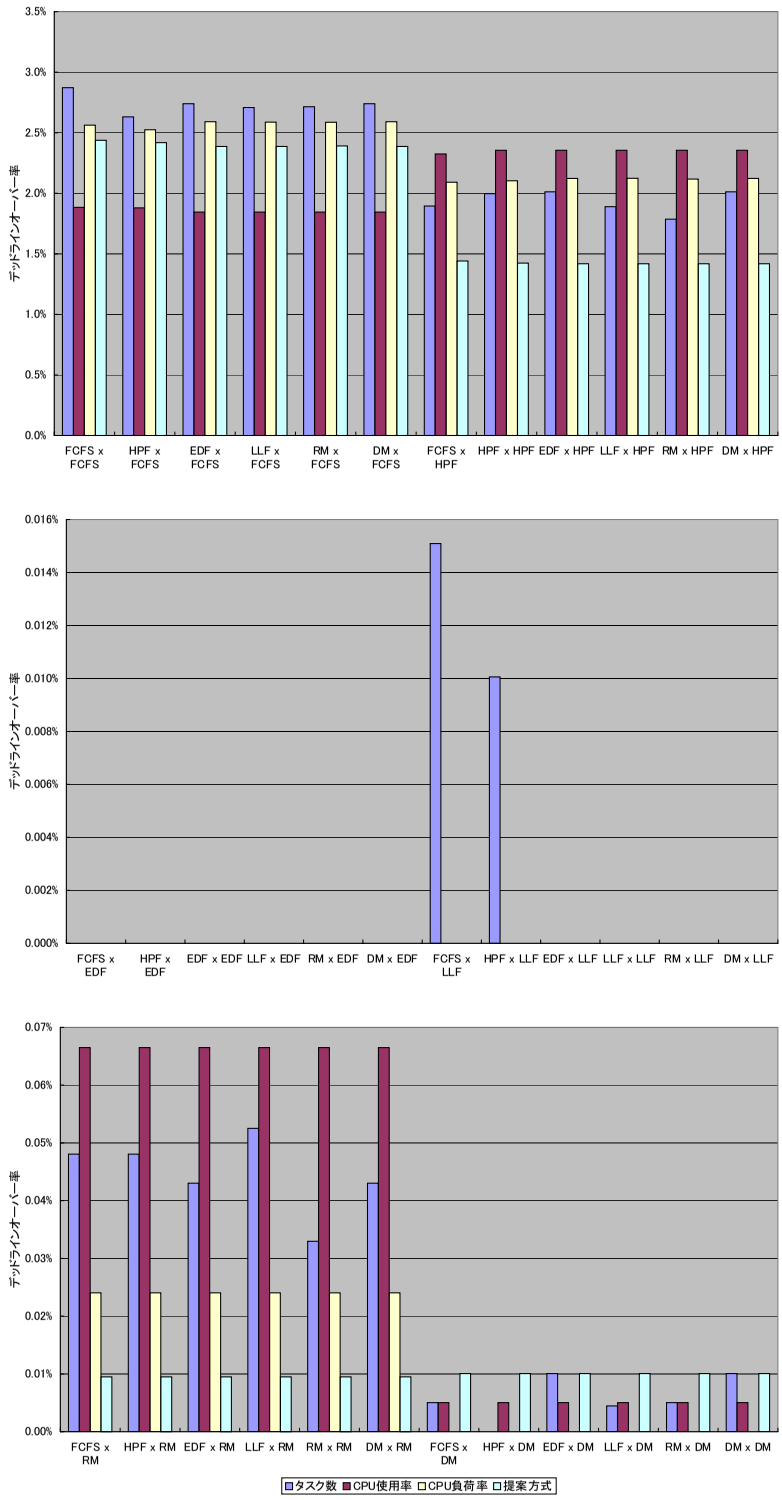


図 4.2: シミュレーション 1 のデッドラインオーバー率

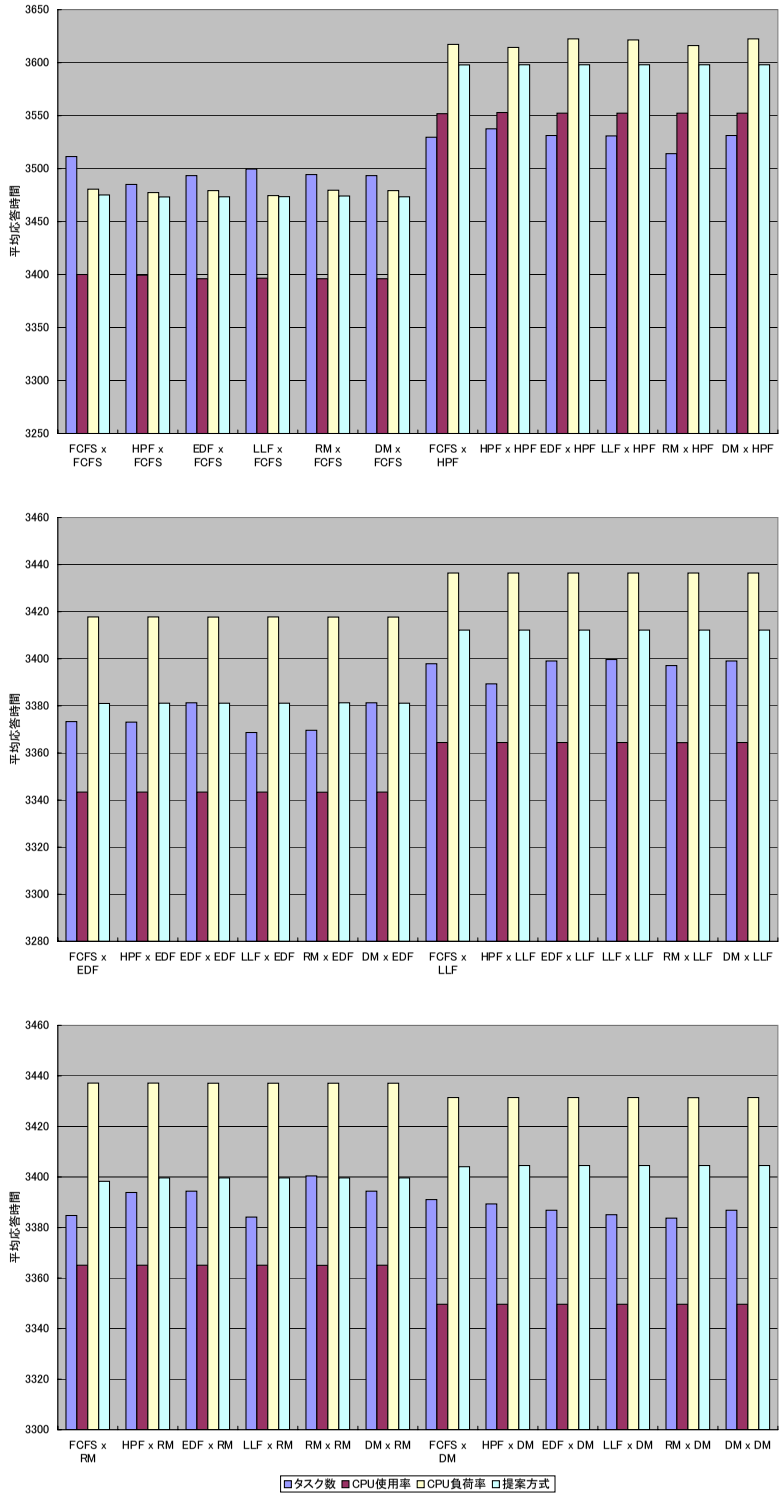


図 4.3: シミュレーション 1 の平均応答時間

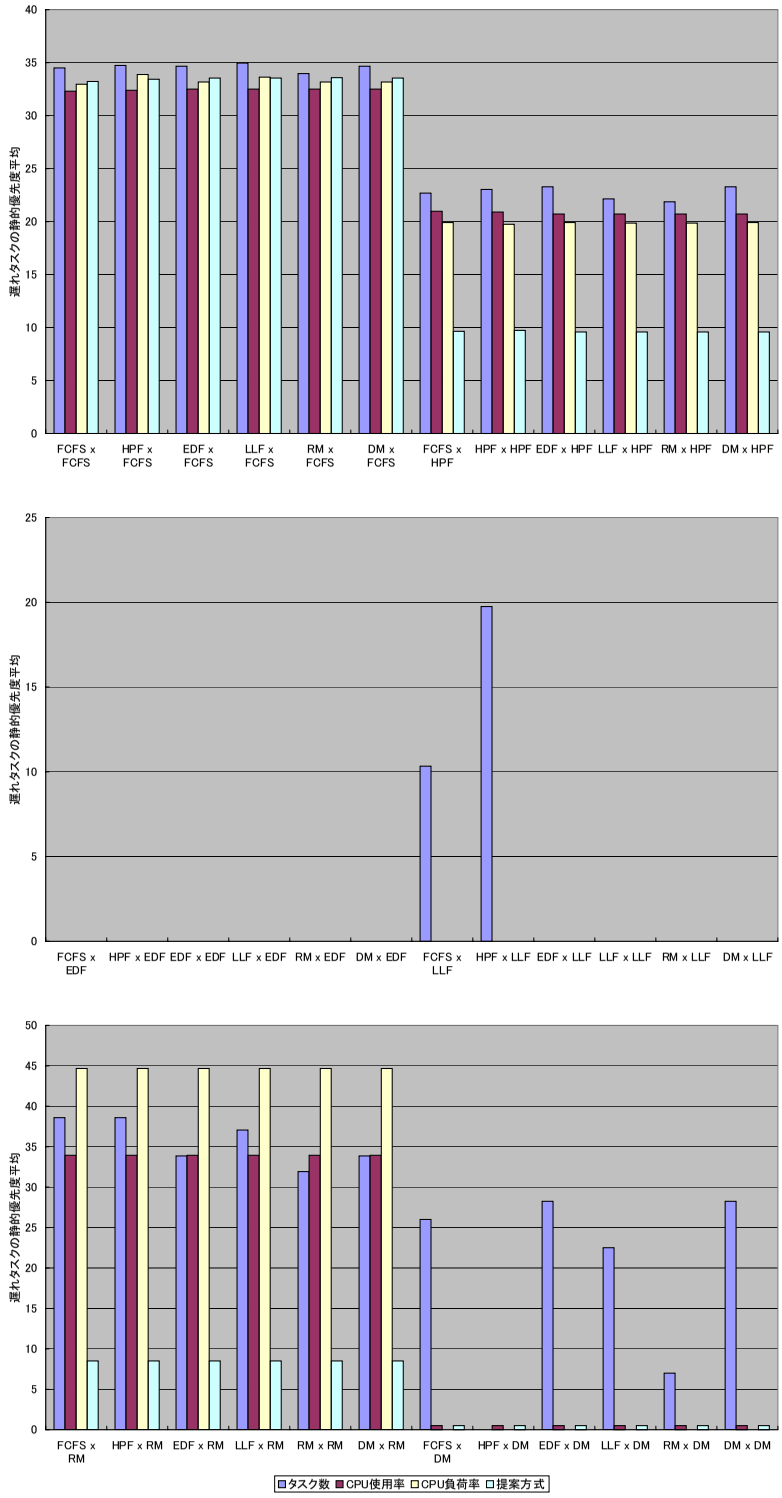


図 4.4: シミュレーション 1 の遅れタスクの静的優先度平均

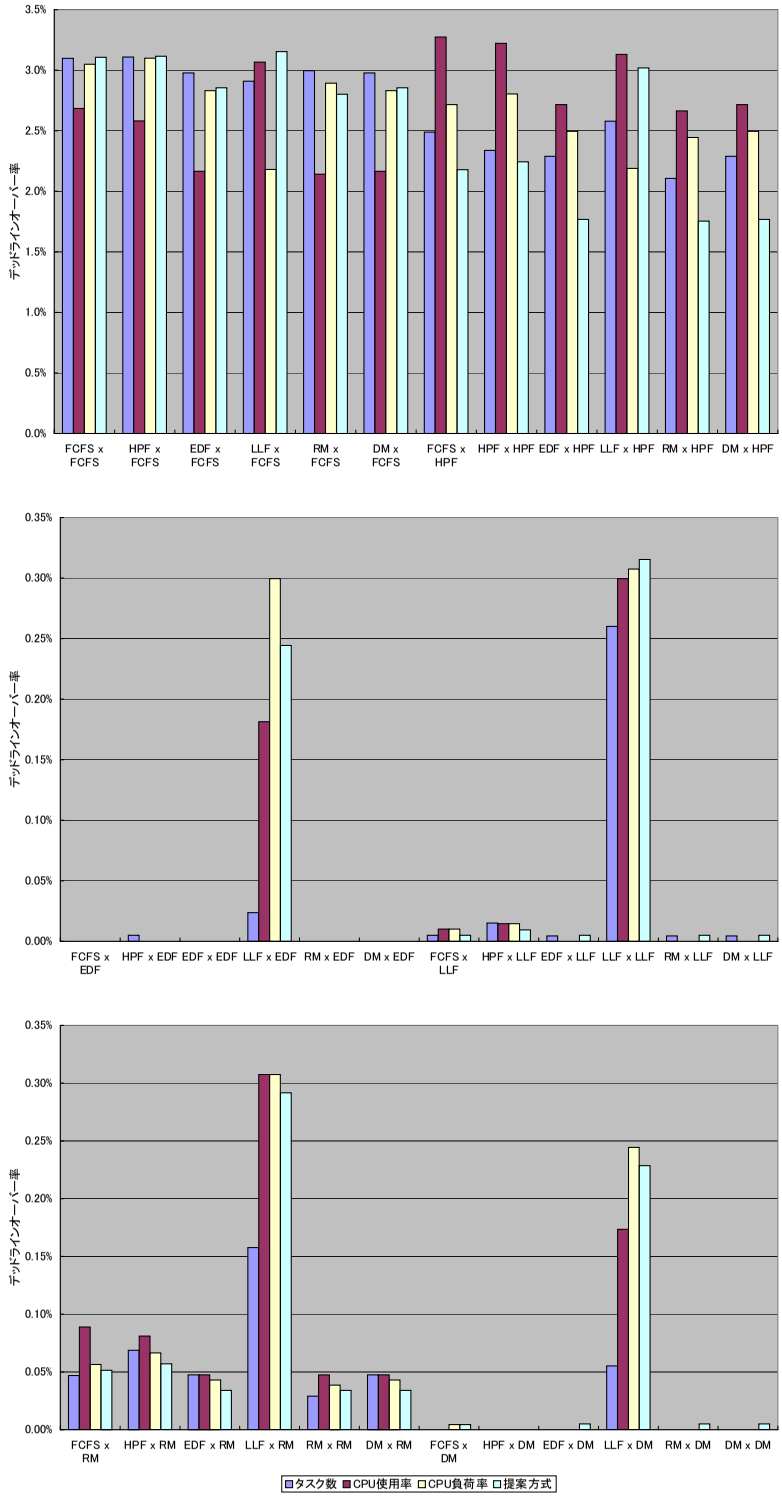


図 4.5: シミュレーション 2 のデッドラインオーバー率

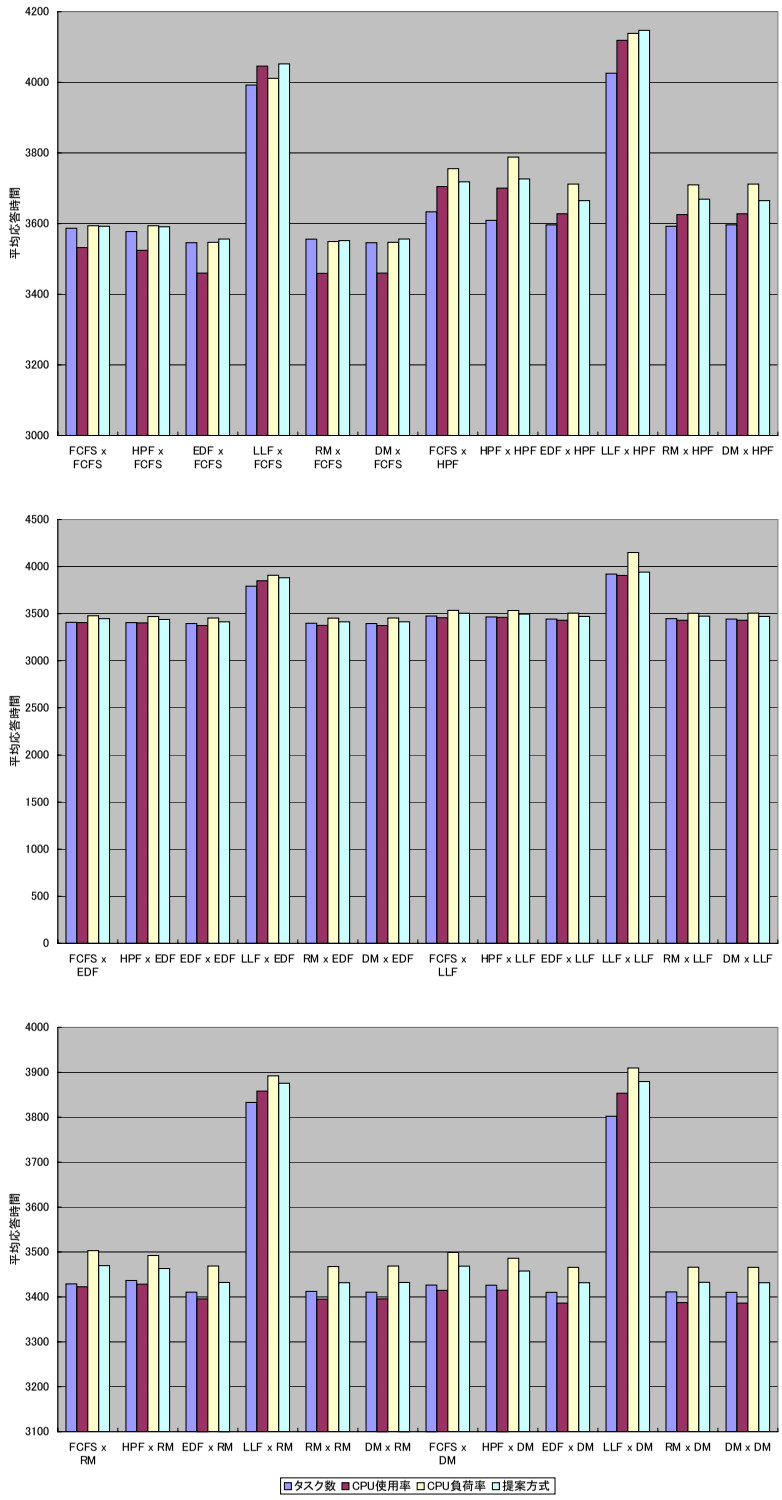


図 4.6: シミュレーション 2 の平均応答時間

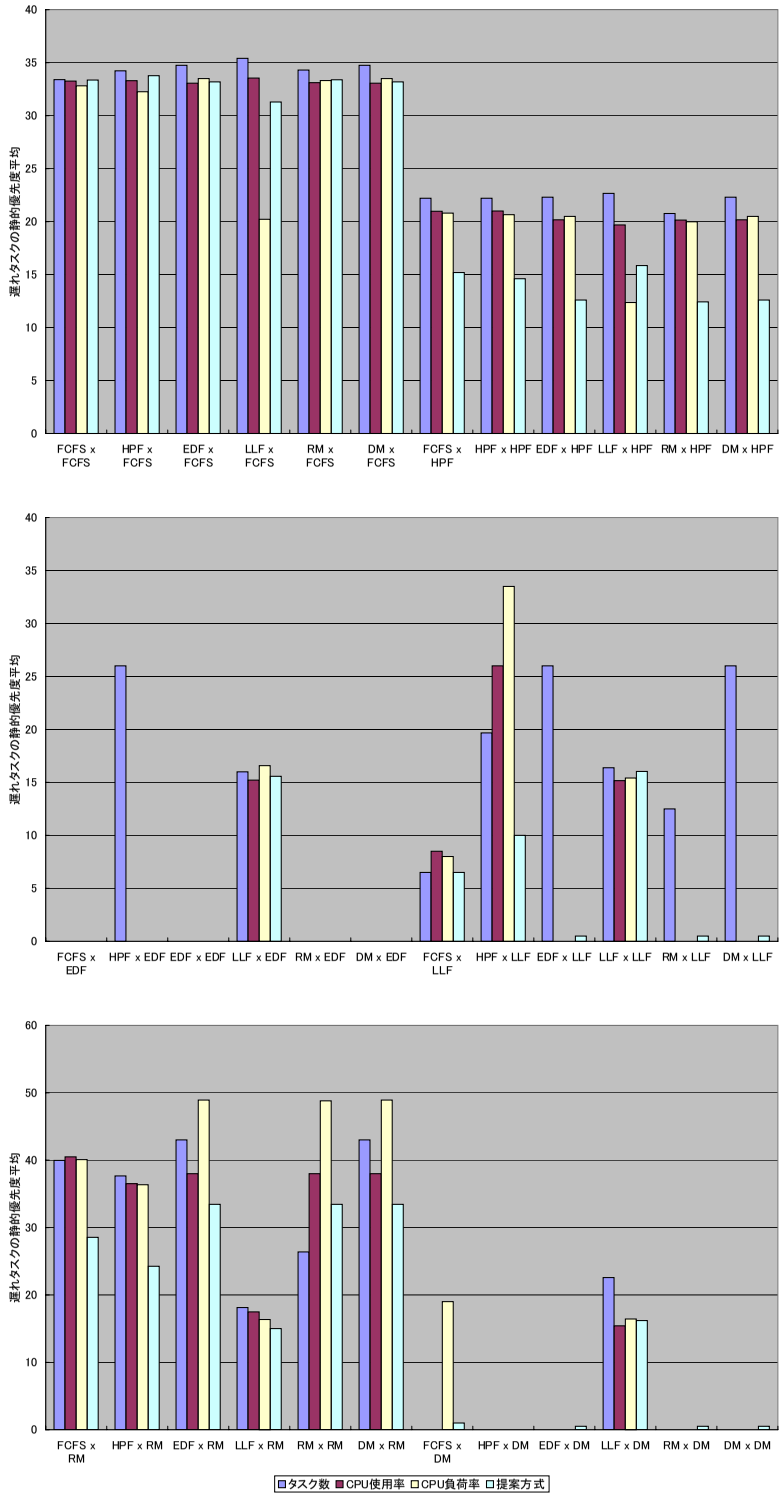


図 4.7: シミュレーション 2 の遅れタスクの静的優先度平均

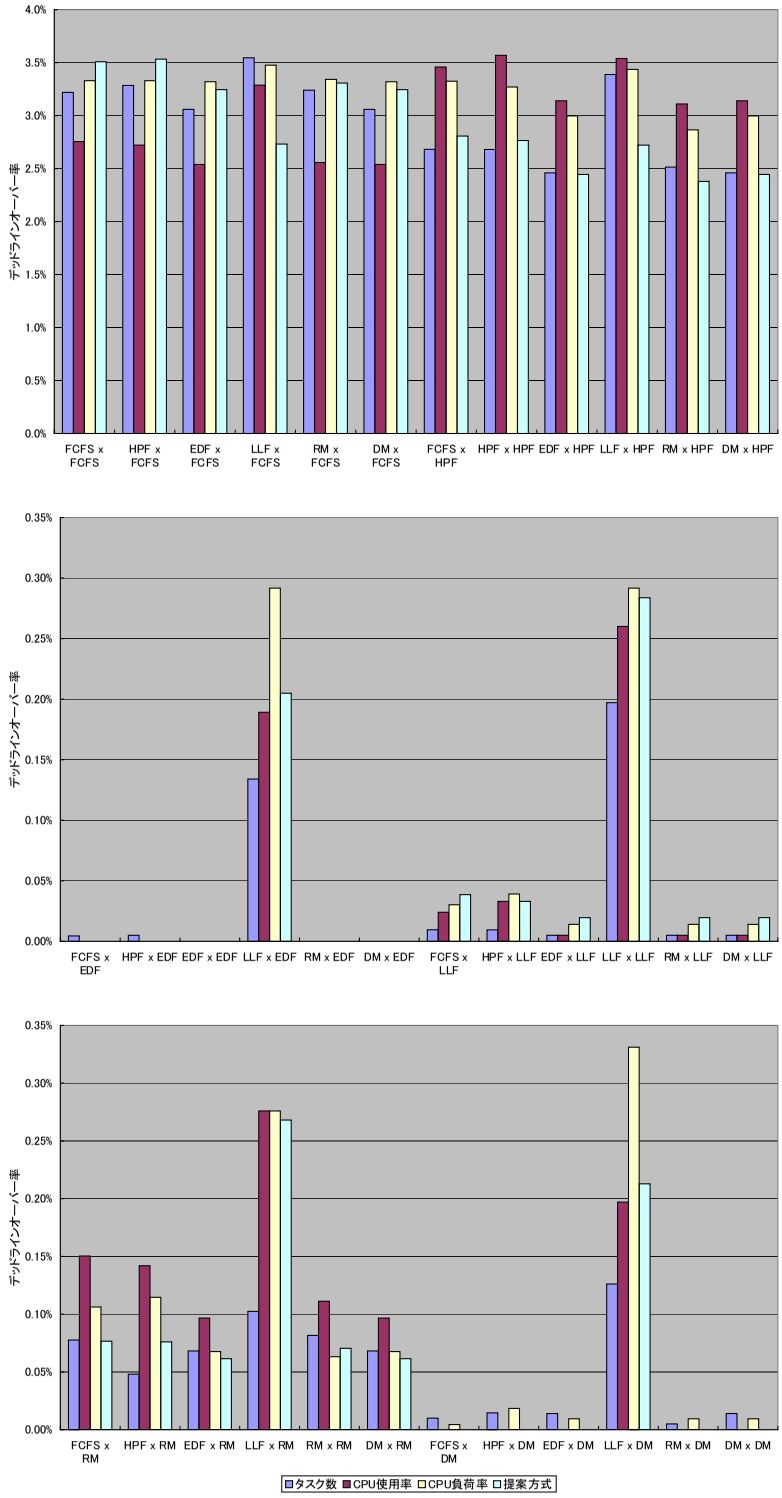


図 4.8: シミュレーション 3 のデッドラインオーバー率

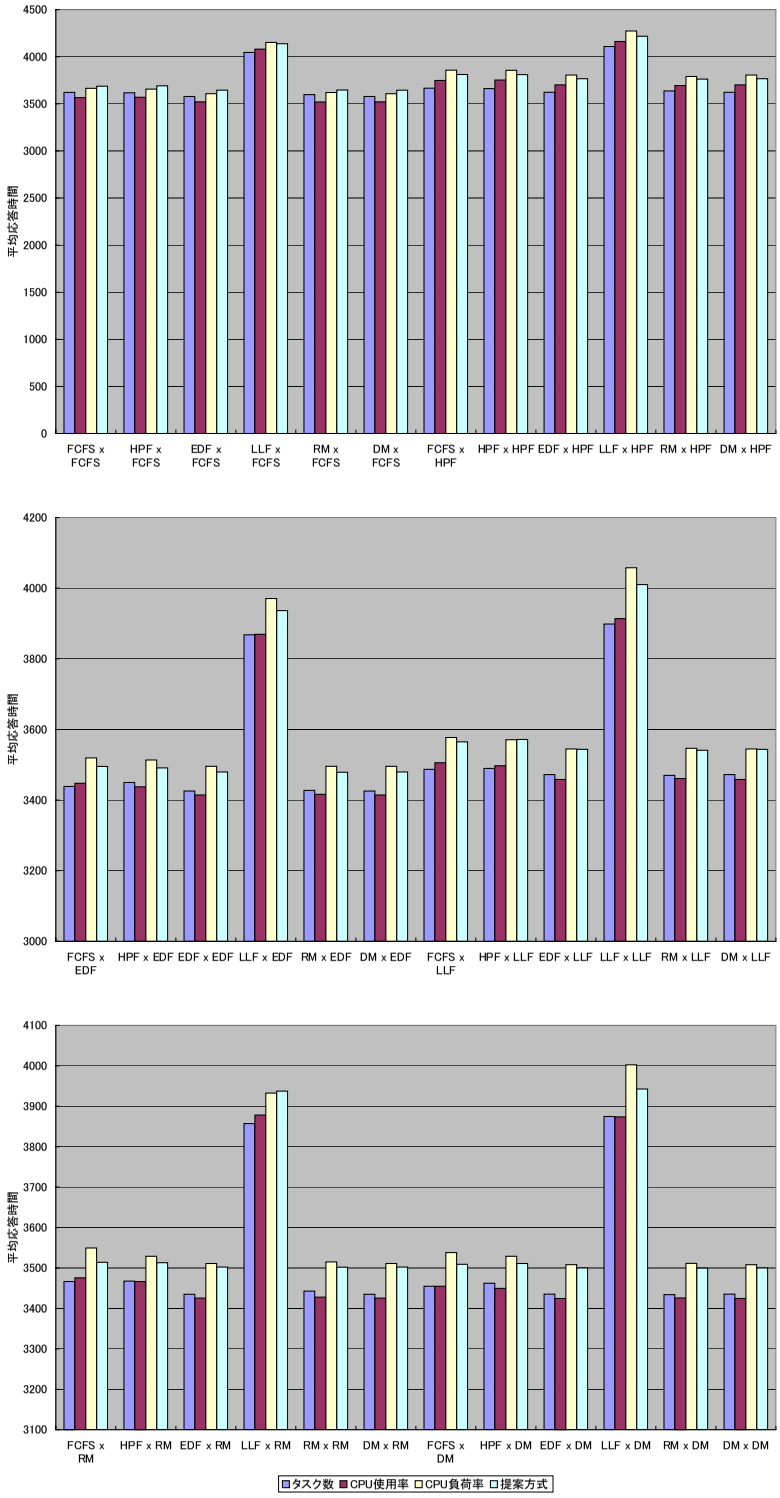


図 4.9: シミュレーション 3 の平均応答時間

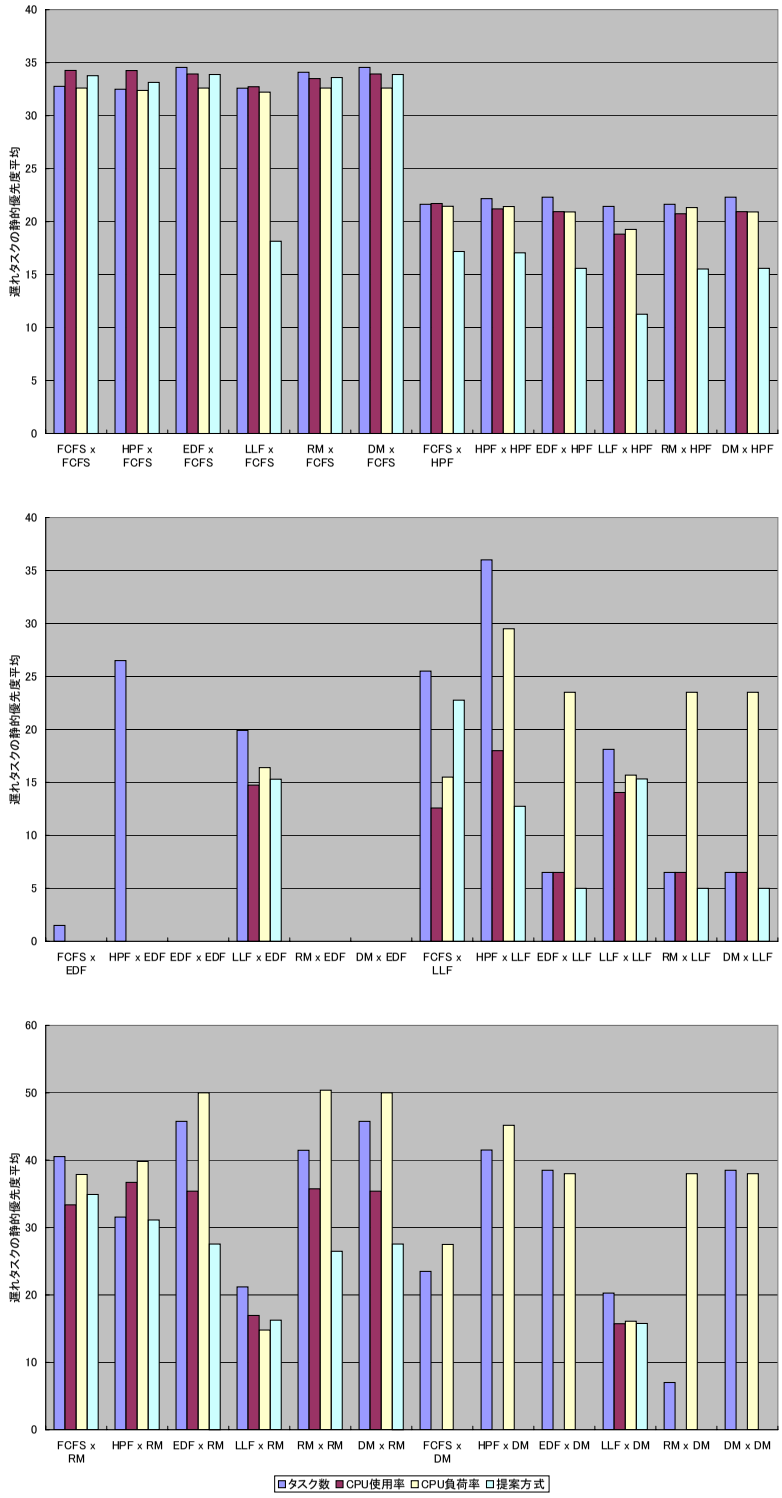


図 4.10: シミュレーション 3 の遅れタスクの静的優先度平均

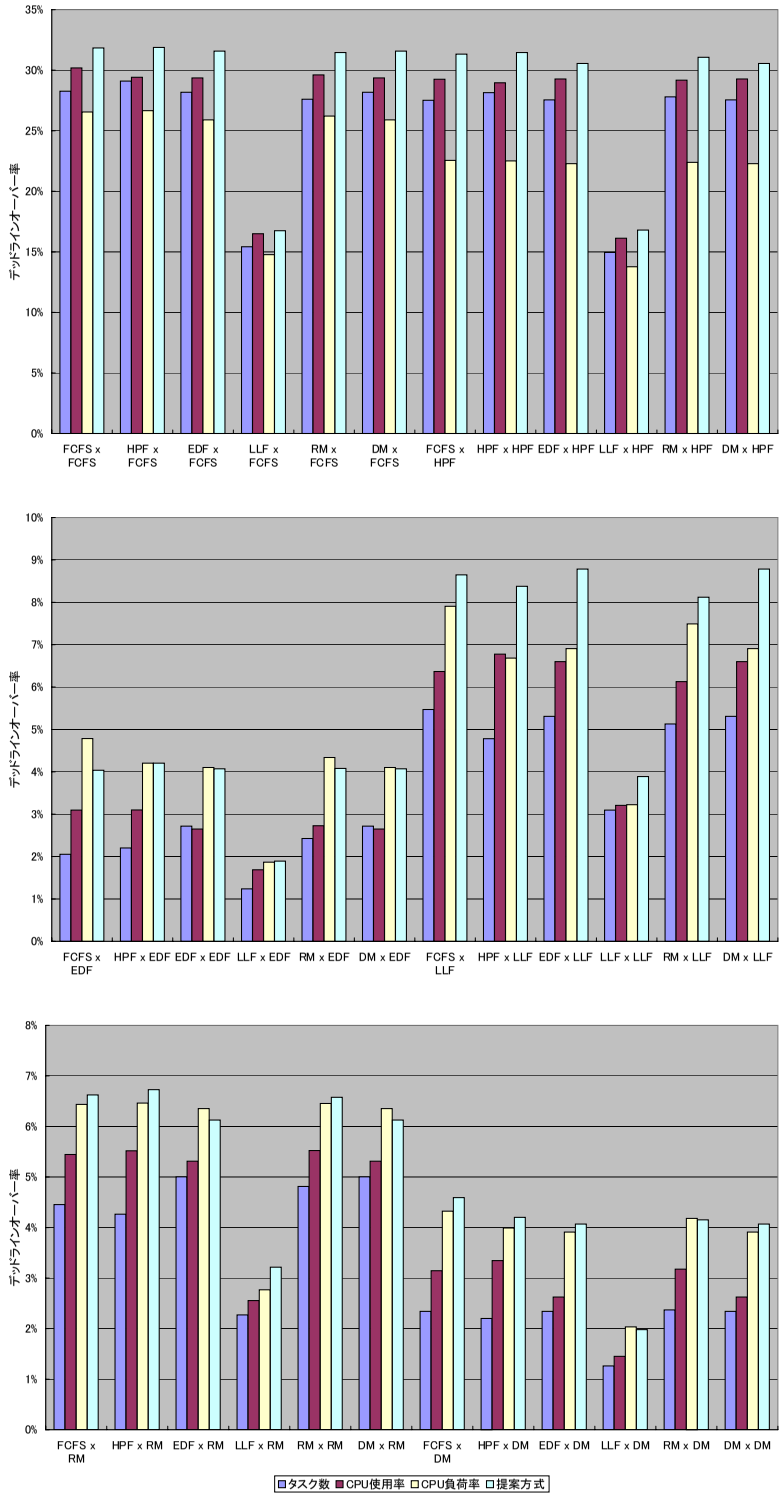


図 4.11: シミュレーション 4 のデッドラインオーバー率

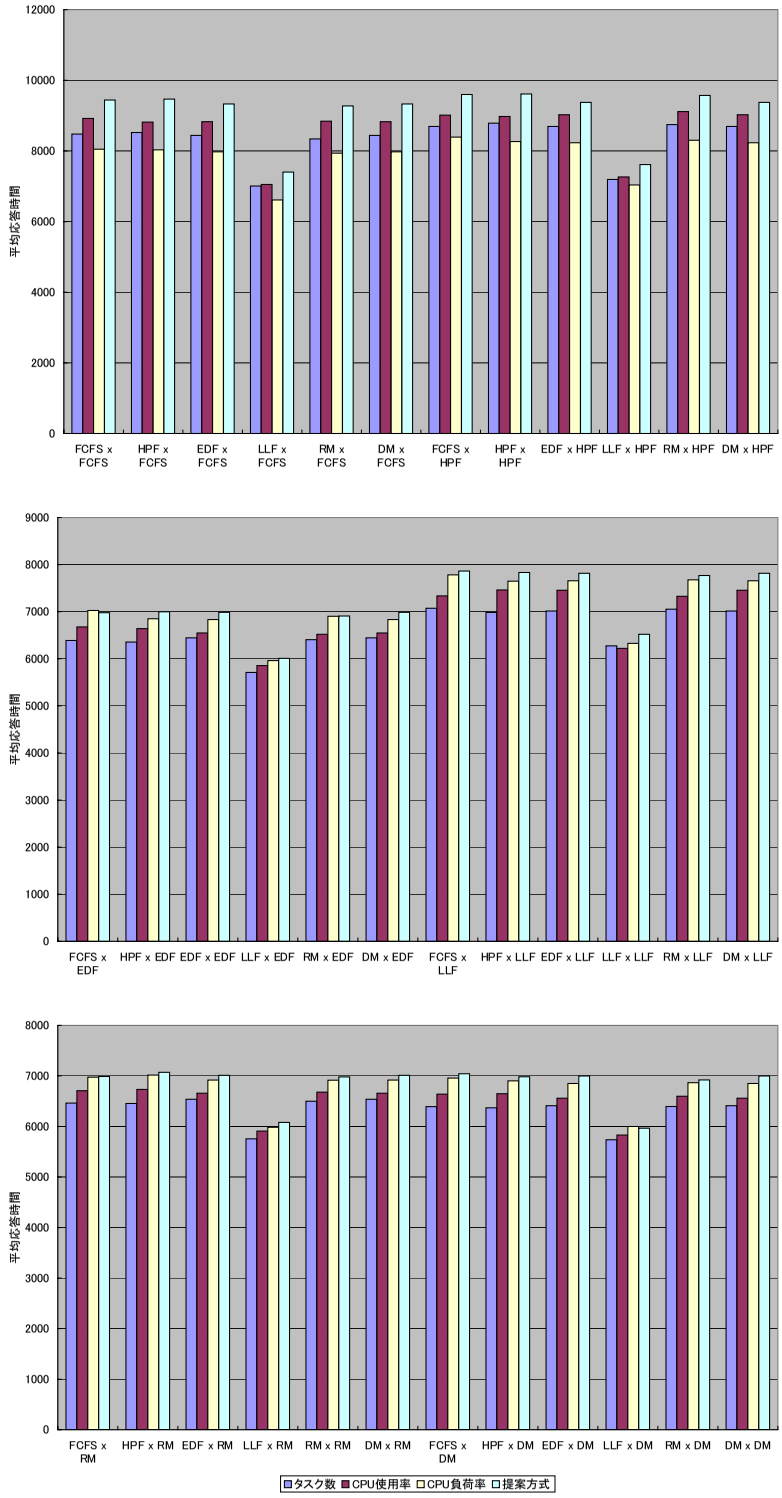


図 4.12: シミュレーション 4 の平均応答時間

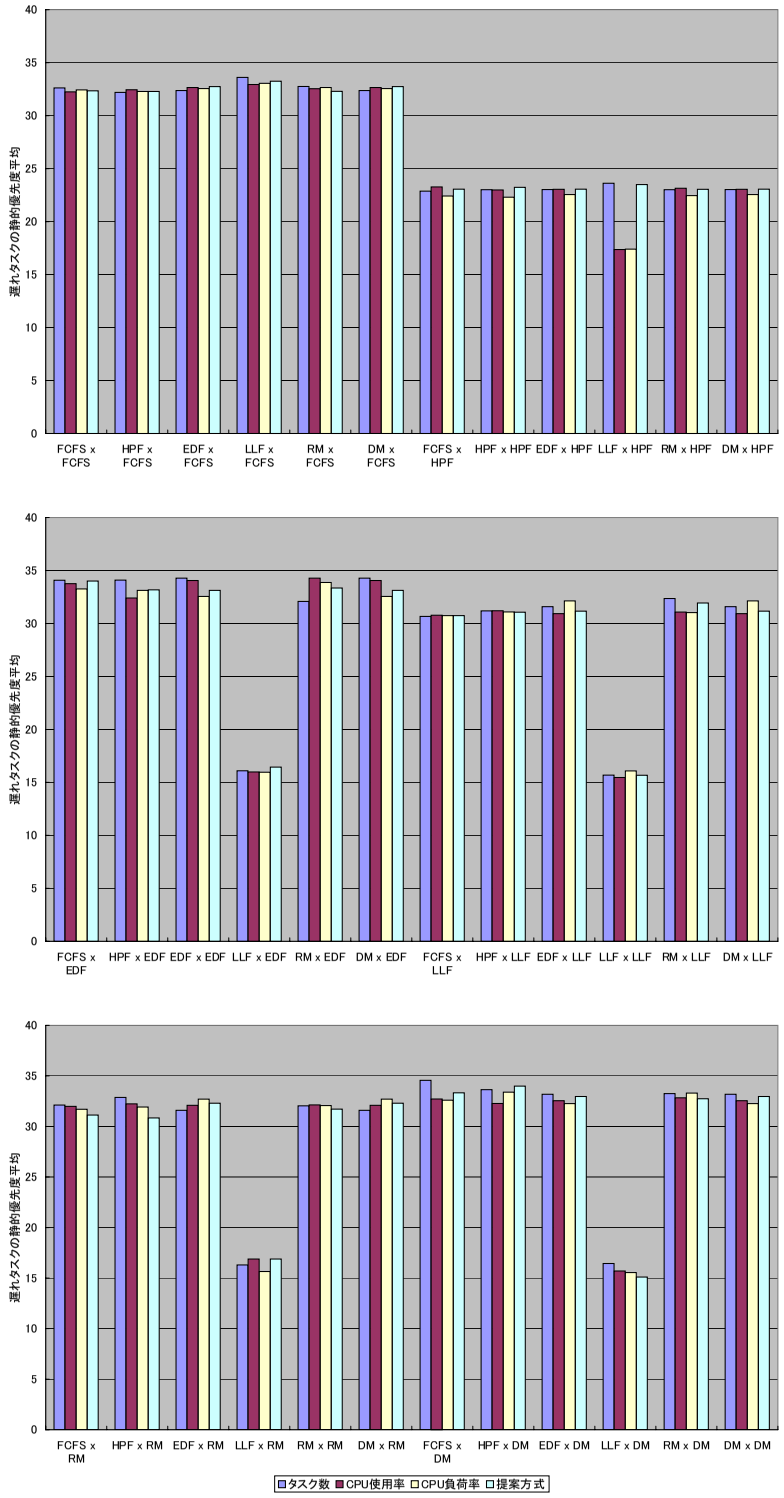


図 4.13: シミュレーション 4 の遅れタスクの静的優先度平均

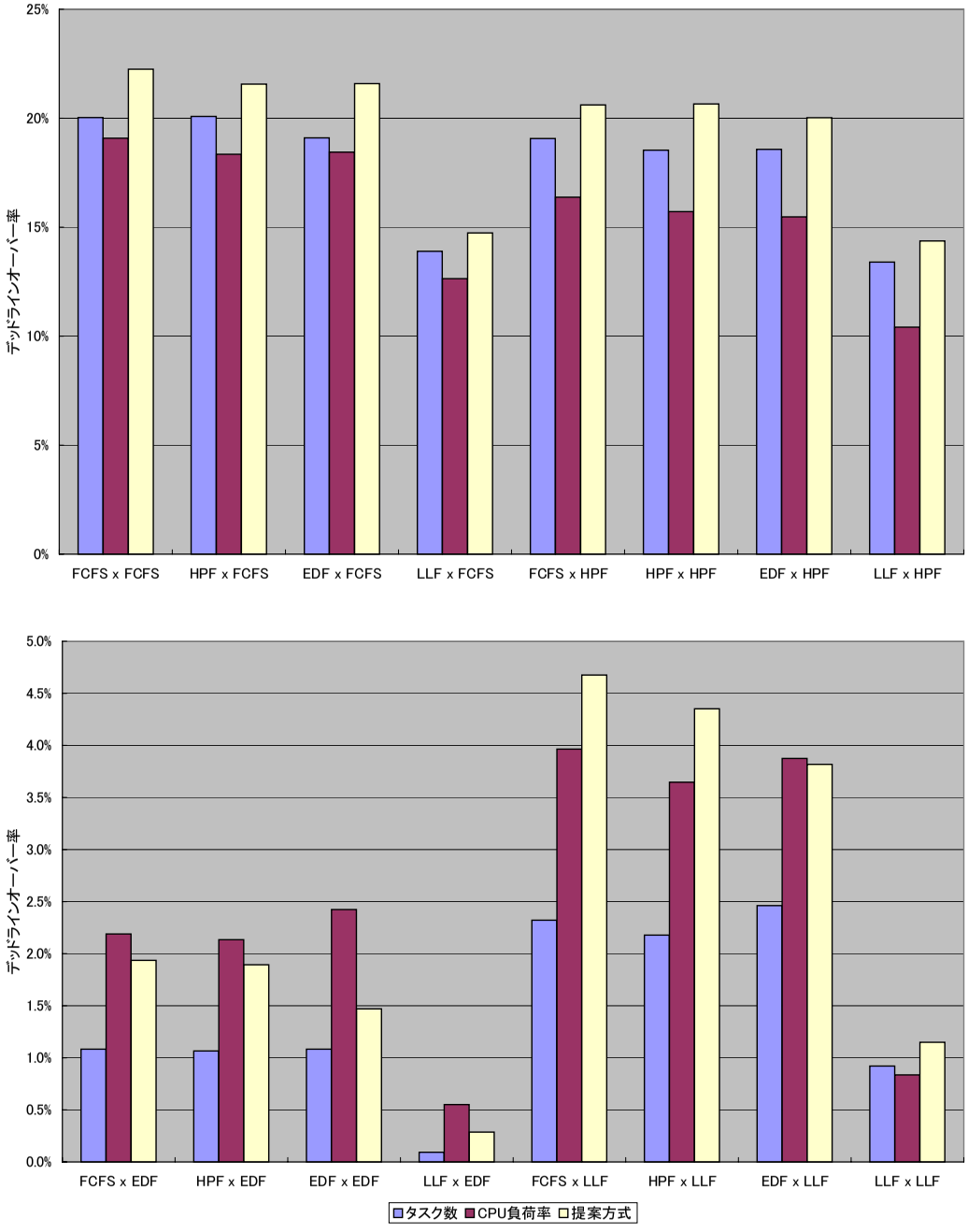


図 4.14: シミュレーション 5 のデッドラインオーバー率

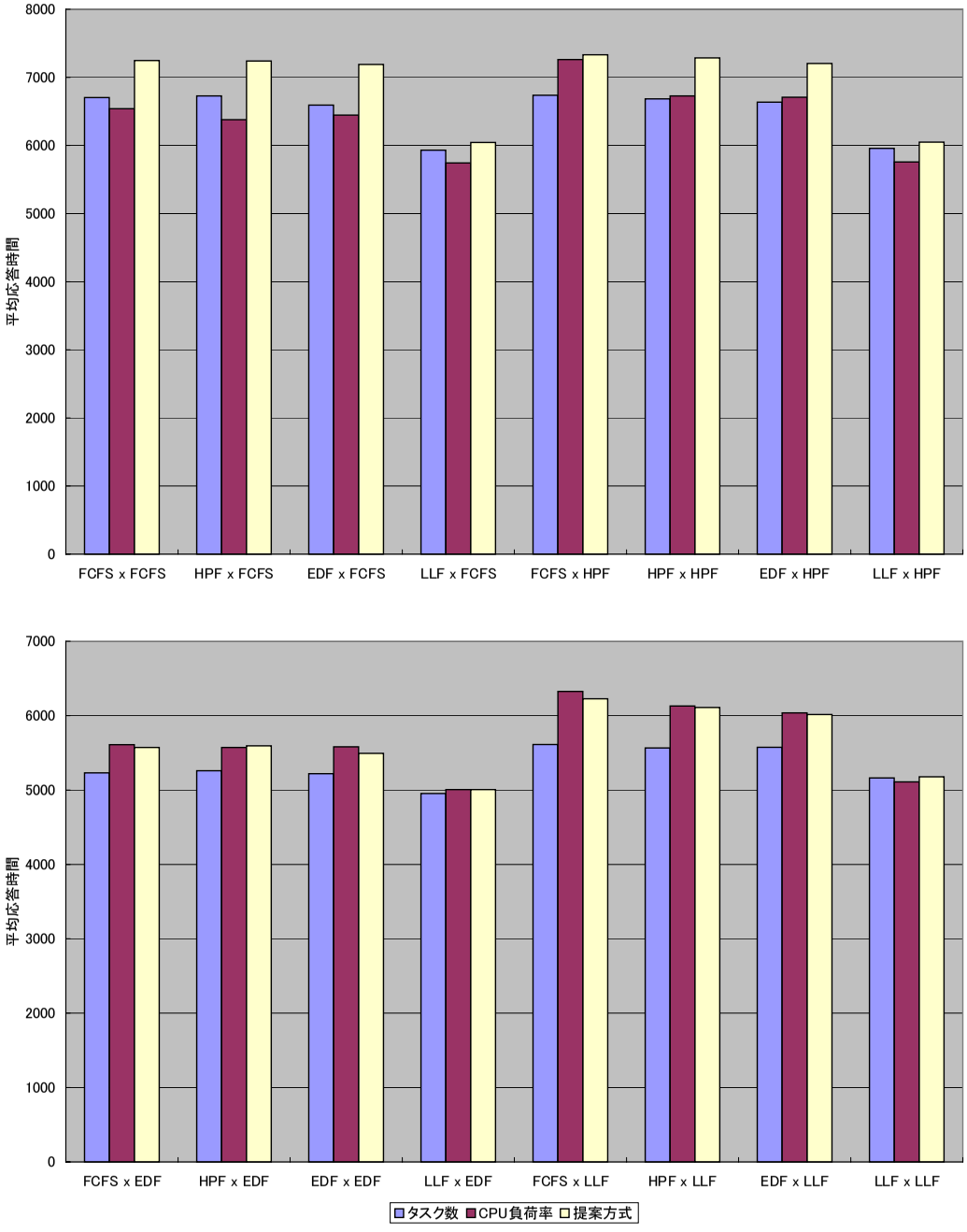


図 4.15: シミュレーション 5 の平均応答時間

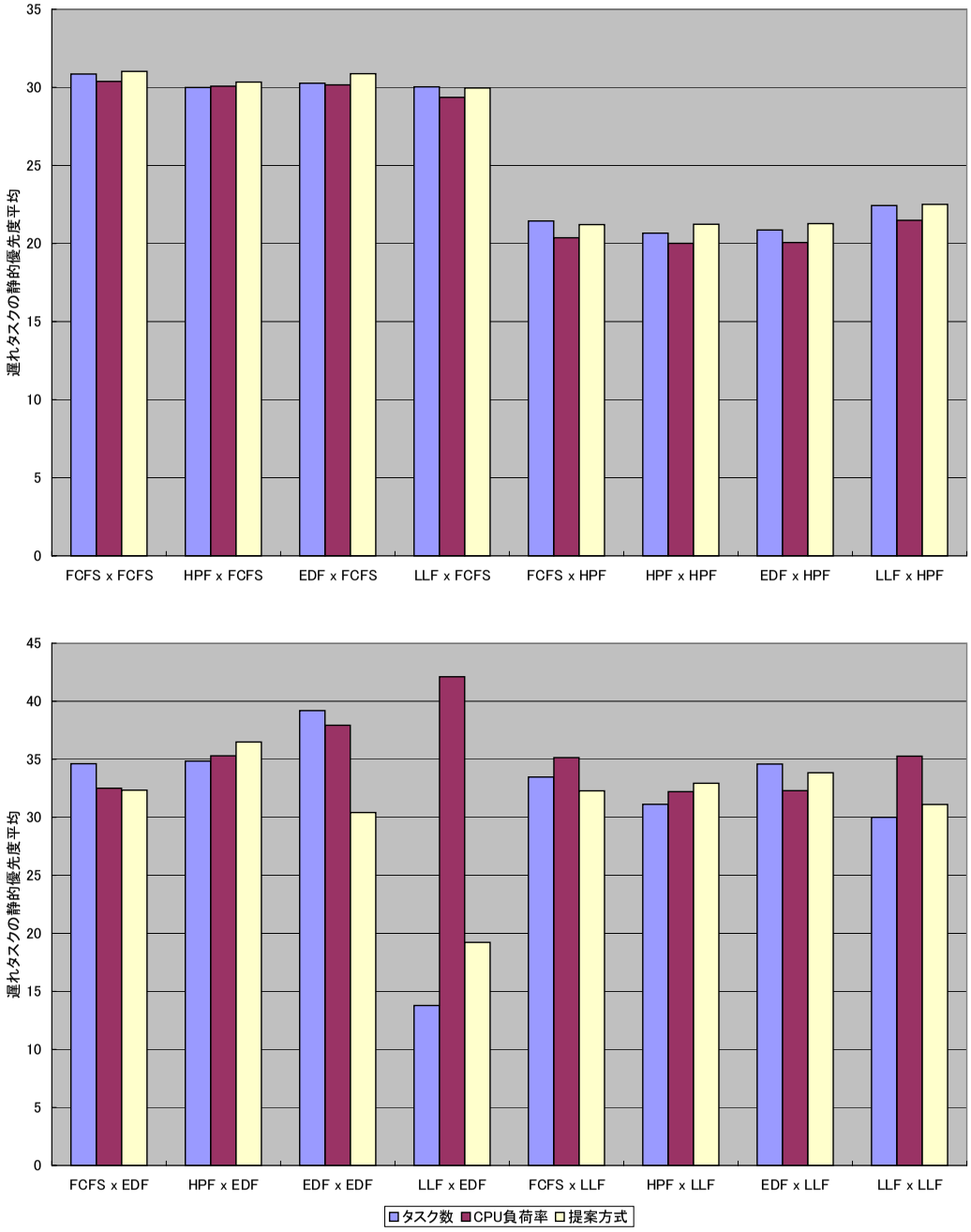


図 4.16: シミュレーション 5 の遅れタスクの静的優先度平均

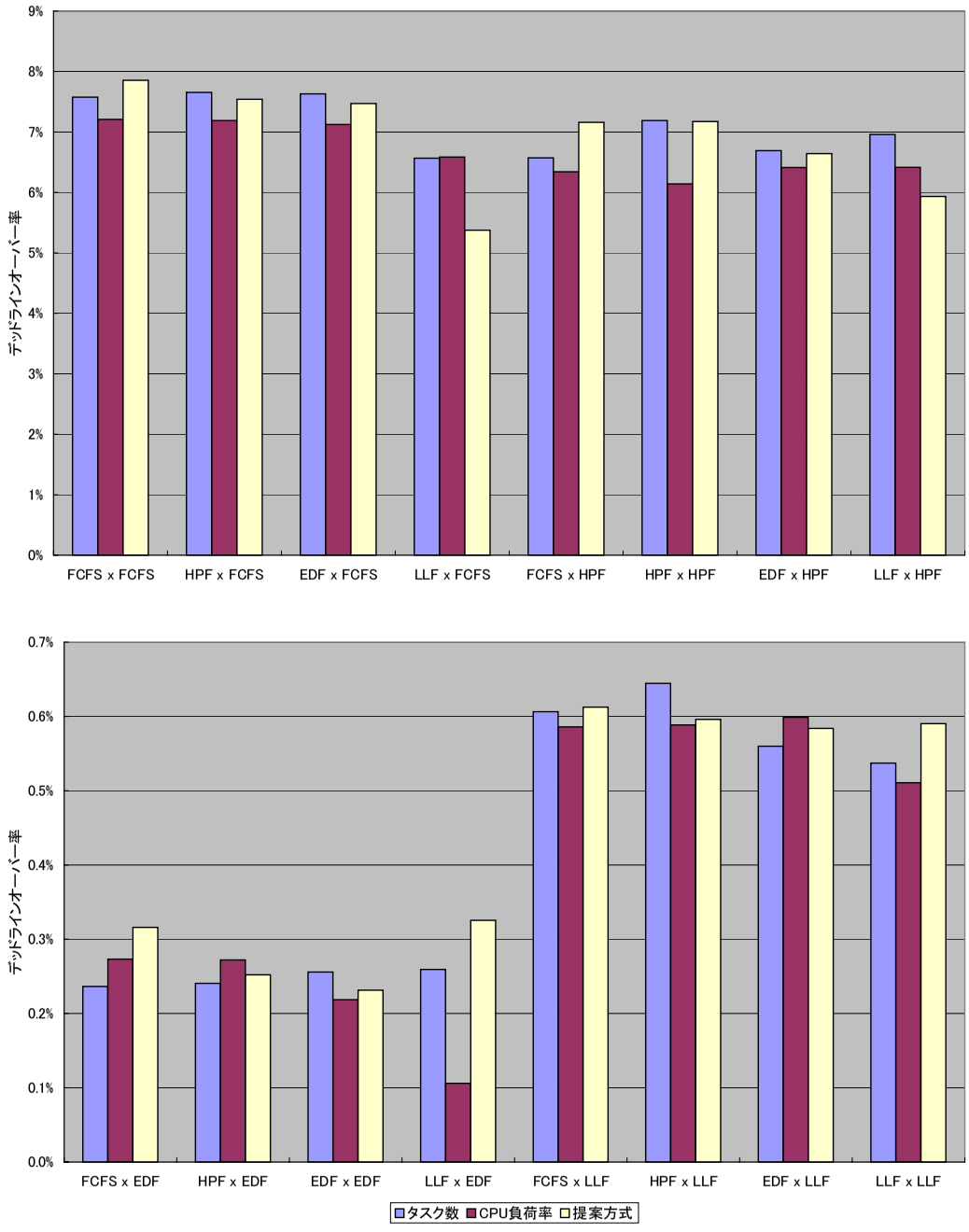


図 4.17: シミュレーション 6 のデッドラインオーバー率

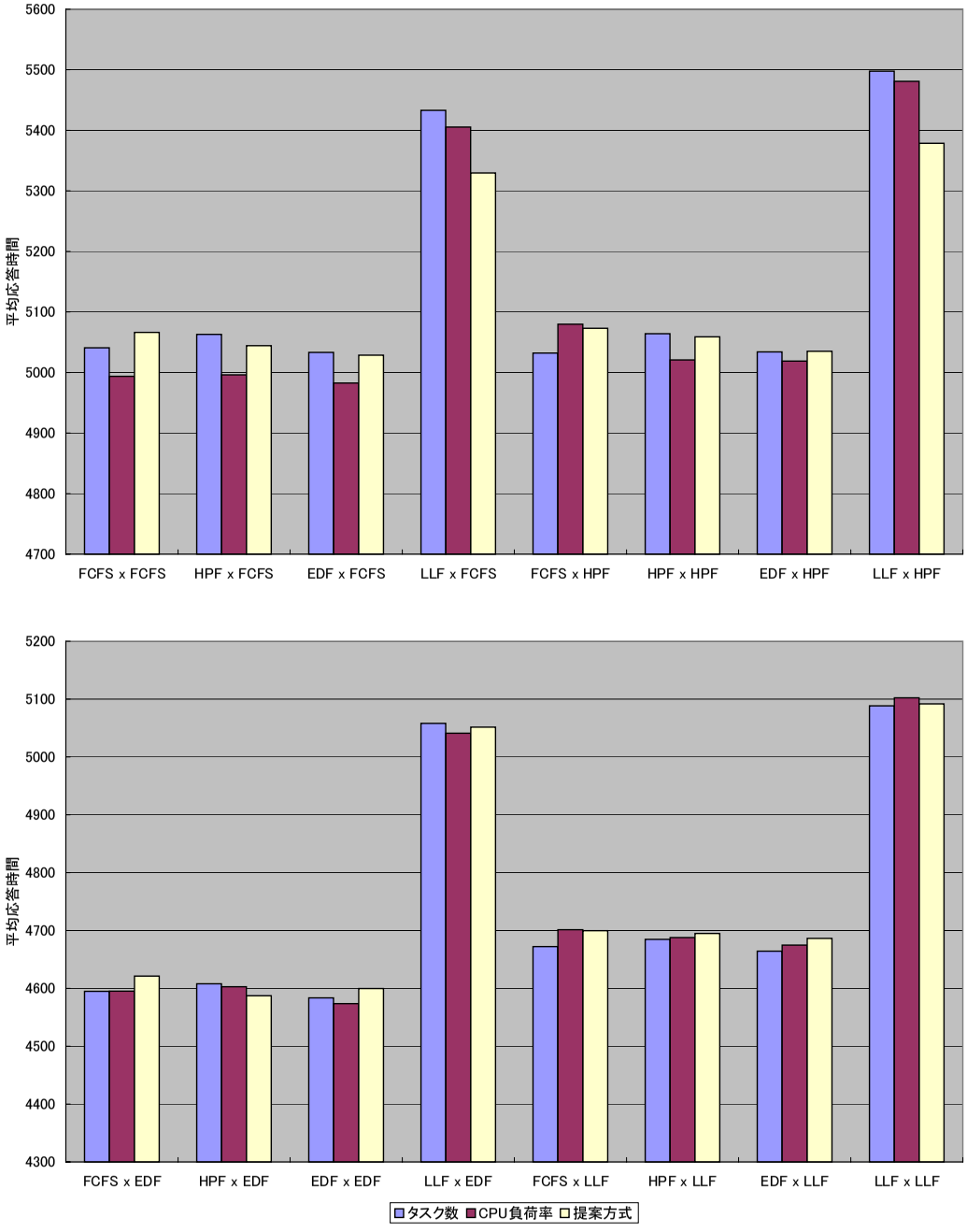


図 4.18: シミュレーション 6 の平均応答時間

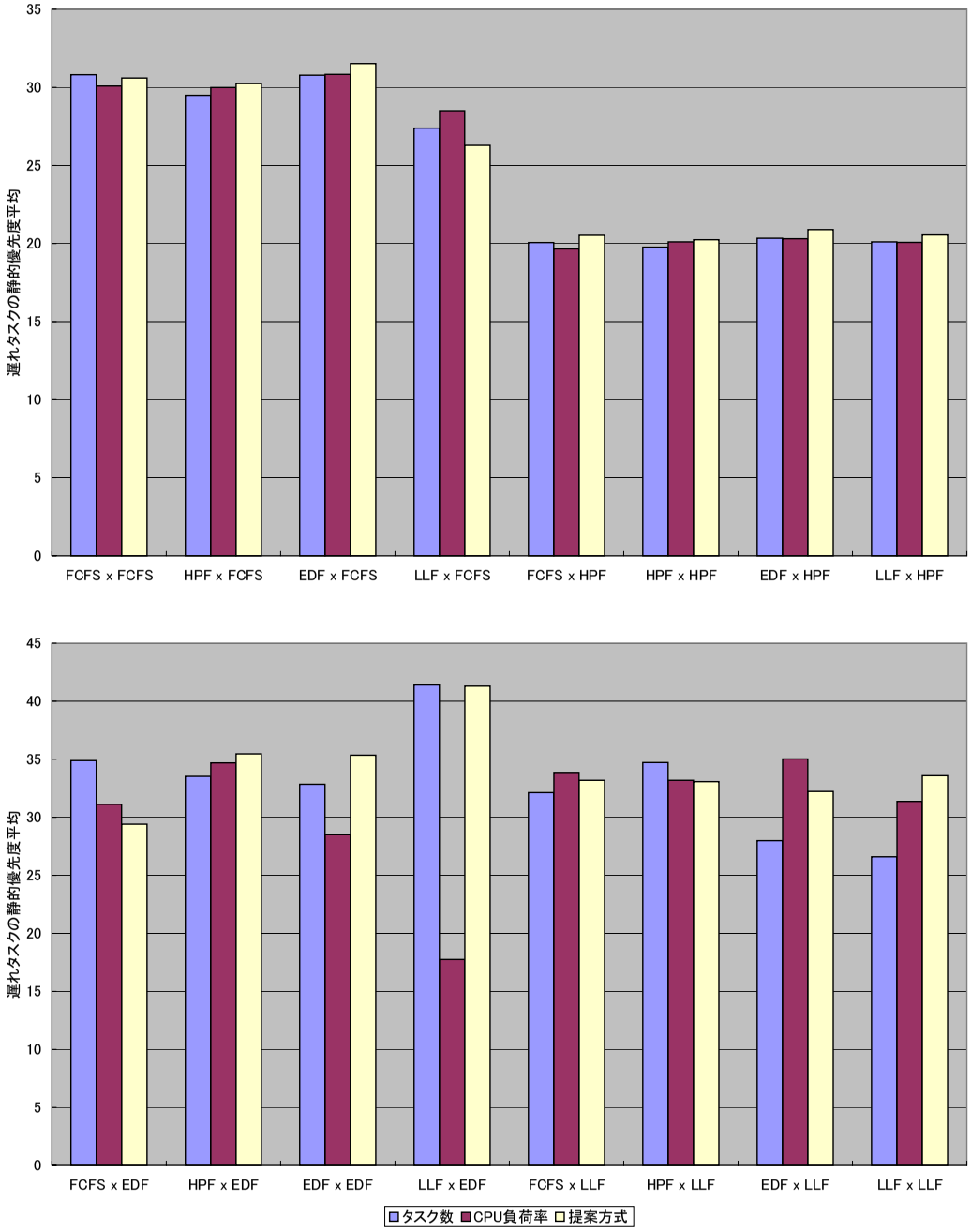


図 4.19: シミュレーション 6 の遅れタスクの静的優先度平均

4.7 考察

各シミュレーション結果から，従来方式と提案方式を比較，考察する．

- **シミュレーション 1**
  - デッドラインオーバー率 (図 4.2)
    各プロセッサの使用するアルゴリズムによって傾向がはっきり分かれている．プロセッサが HPF 法か RM 法でスケジューリングする場合，提案方式が最も優れている．FCFS と LLF， HPF と LLF の組合わせの場合のデッドラインオーバー率は，誤差とみなして差し支えないほど小さい．
  - 平均応答時間 (図 4.3)
    各プロセッサの使用するアルゴリズムが HPF 法の場合を除いて，CPU 使用率を用いた場合が最も良い結果になっている．各プロセッサの使用するアルゴリズムが FCFS 法の場合を除いて，CPU 負荷率を用いた場合が最も悪い結果になっている．提案方式の優位性は特に認められない．
  - 遅れタスクの静的優先度合計 (図 4.4)
    デッドラインオーバー率のグラフ (図 4.2) と似ているが，プロセッサ上で RM 法，DM 法を使用している場合がかなり異なる．CPU 負荷率と RM 法の組合わせではデッドラインオーバー率が低かったが，このグラフから，静的優先度の高いタスクを多くミスしていたことが分かる．
- **シミュレーション 2**
  - デッドラインオーバー率 (図 4.5)
    各プロセッサで HPF または RM を使用している場合，提案方式が優れているといえる．1% 未満の数値なので大きくはないが，GS が LLF 法を用いた場合の結果が悪くなっている．
  - 平均応答時間 (図 4.6)
    GS が LLF 法を用いた場合の結果が悪くなっている．
  - 遅れタスクの静的優先度合計 (図 4.7)
    各プロセッサで HPF または RM を使用している場合，提案方式が優れているといえる．負荷の指標としてタスク数を用いた場合の結果が悪くなっている例が目立つ．
- **シミュレーション 3**
  - デッドラインオーバー率 (図 4.8)
    シミュレーション 2 の結果と大差ない．

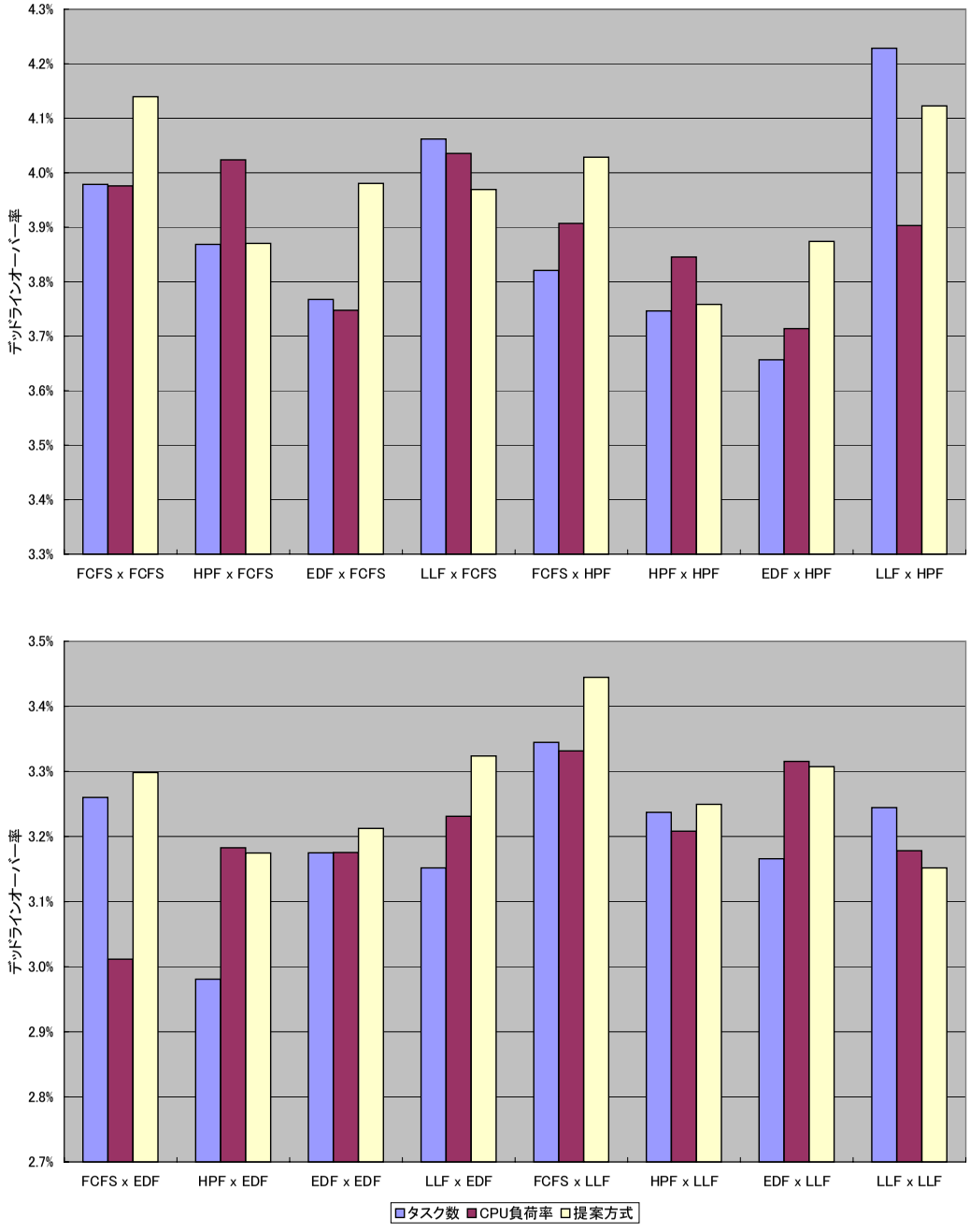


図 4.20: シミュレーション 7 のデッドラインオーバー率

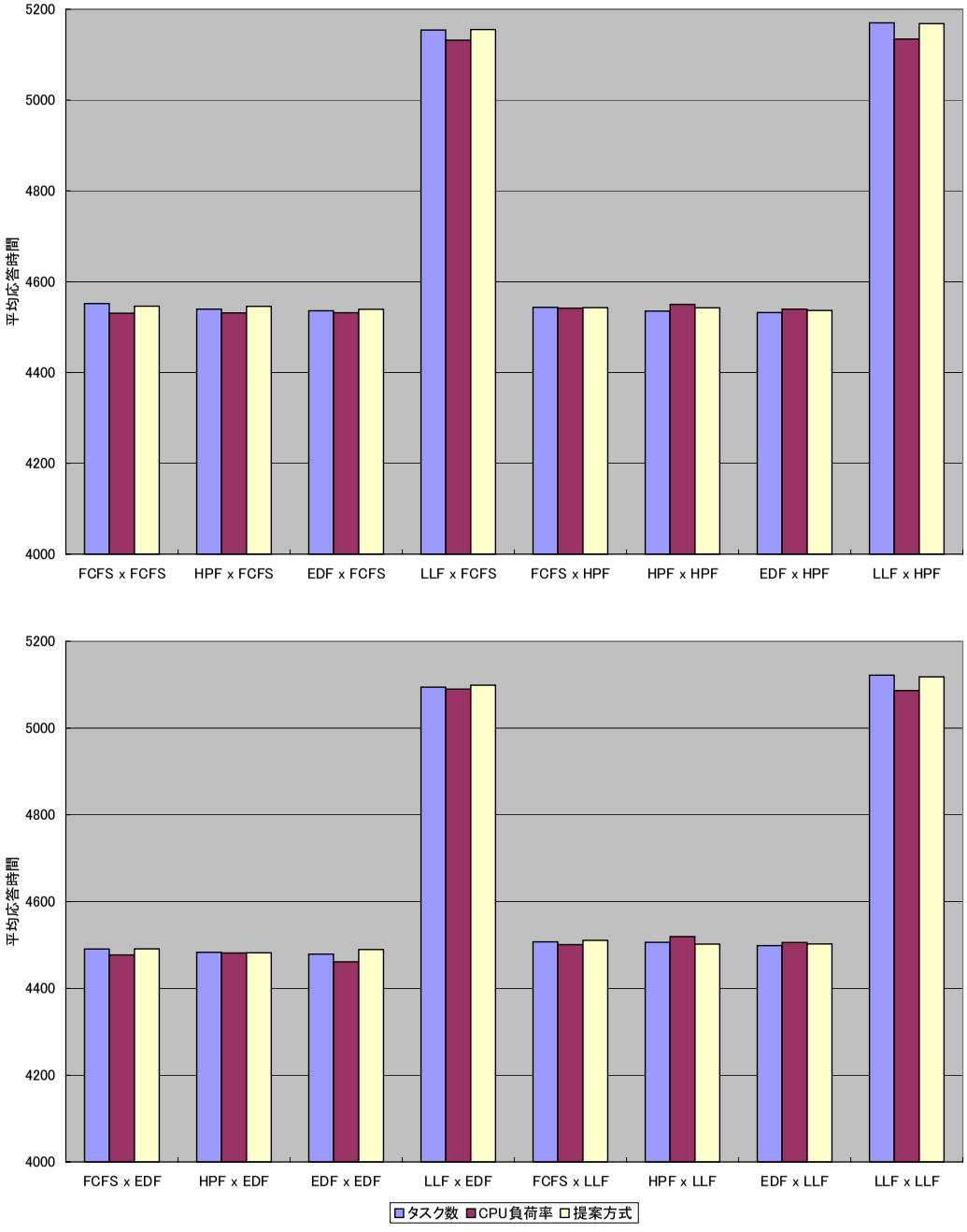


図 4.21: シミュレーション 7 の平均応答時間

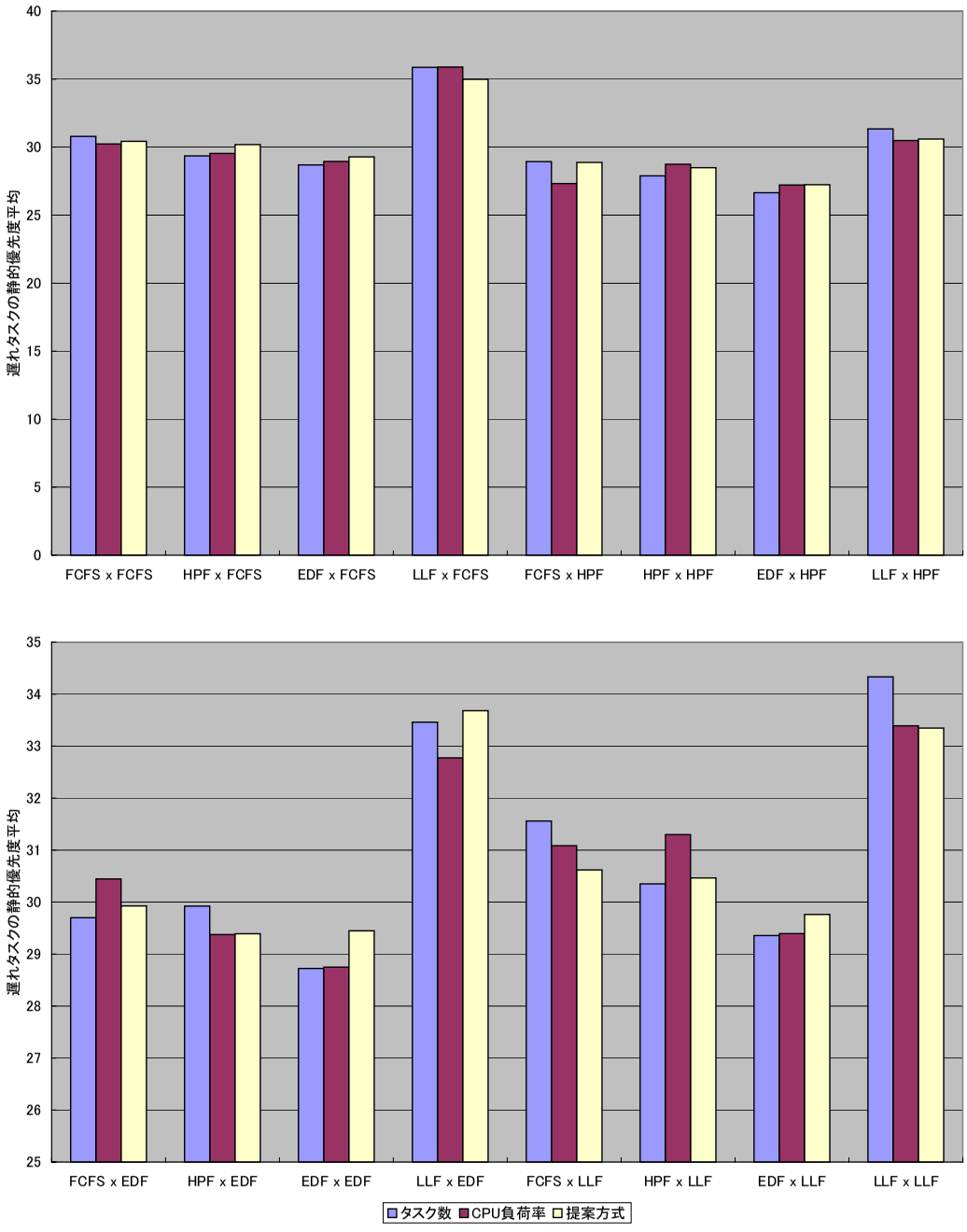


図 4.22: シミュレーション 7 の遅れタスクの静的優先度平均

– 平均応答時間 (図 4.9)
わずかな差だが，CPU 使用率，タスク数，提案方式，CPU 負荷率の順で結果が良い．GS が LLF 法を用いた場合の結果が悪くなっている．

– 遅れタスクの静的優先度合計 (図 4.10)
提案方式が良い結果を出しているパターンが多い．タスク数や CPU 負荷率を用いたことによって，静的優先度の高いタスクがオーバーしている例が目立つ．GS が LLF 法を使用している場合，デッドラインオーバー率 (図 4.8) と平均応答時間 (図 4.9) が悪い結果となったが，このグラフでは RM 法や DM 法と組み合わせると良い結果となっている．オーバーしたタスクの数は多いが，静的優先度の低いタスクが多いことが原因である．

- **シミュレーション 4**

– デッドラインオーバー率 (図 4.11)
提案方式が悪い結果になった例が多い．GS が LLF 法を用いた場合の結果が比較的 良くなっている．

– 平均応答時間 (図 4.12)
総じて，タスク数，CPU 使用率，CPU 負荷率，提案方式の順に良い結果を出している．ただし，プロセッサで FCFS 法 または HPF 法を用いている場合は CPU 負荷率が最も良い．

– 遅れタスクの静的優先度合計 (図 4.13)
方式の違いによる差は特に認められない．ただし，GS が LLF 法を用いた場合の結果が良くなっている．

- **シミュレーション 5**

– デッドラインオーバー率 (図 4.14)
プロセッサが FCFS 法か HPF 法を使用している場合の結果は，CPU 負荷率，タスク数，提案方式の順に良い．プロセッサが EDF 法か LLF 法を使用している場合に，タスク数を負荷の指標にすると良い結果が出ている．GS が LLF 法を用いた場合の結果が良くなっている．

– 平均応答時間 (図 4.15)
GS が LLF 法を用いた場合の結果がやや良い．提案方式は従来手法とほぼ同じかやや悪い結果となった．

– 遅れタスクの静的優先度合計 (図 4.16)
提案方式のデッドラインオーバー率 (図 4.14) は従来手法と比べて悪い結果が多かったが，このグラフではその差が縮まっている．提案手法によって静的優先度の高いタスクがデッドラインオーバーを避けることができたことが分かる．

- **シミュレーション 6**
  - デッドラインオーバー率 (図 4.17)
    負荷の指標の違いによる差異には特に際立った規則性は見られないが，プロセッサが EDF 法を用いた場合の結果が全体的に良い．
  - 平均応答時間 (図 4.18)
    負荷の指標の違いによる差は特に見られないが，GS が LLF 法を用いた場合の結果が悪い．
  - 遅れタスクの静的優先度合計 (図 4.19)
    プロセッサが HPF 法を用いた場合の結果が全体的に良い．
- **シミュレーション 7**
  - デッドラインオーバー率 (図 4.20)
    GS が FCFS 法を用いている場合，提案方式との相性が悪い．
  - 平均応答時間 (図 4.21)
    負荷の指標の違いによる差は特に見られないが，GS が LLF 法を用いた場合の結果が飛び抜けて悪い．
  - 遅れタスクの静的優先度合計 (図 4.22)
    デッドラインオーバー率 (図 4.20) ほどの違いはない．ただし，GS が LLF 法を用いた場合の結果が比較的 悪くなっている．

全体を通して，以下のことが結論できる．

- 全体的に，GS のタスク優先度付け法よりも，各プロセッサ上のそれが大きな影響を及ぼす．
- GS が LLF 法を用いている場合，スケジューリングオーバヘッドが比較的 大きいため，パフォーマンスに悪影響が及ぶことがある (シミュレーション 2)．しかし，他のオーバヘッド (負荷計算，通信遅延) を合わせて考慮した場合，パフォーマンスは比較的 良くなる．オーバヘッドのためにタスクの余裕時間が減少するため，余裕時間を考慮した LLF 法が適していたといえる．
- 通信遅延を考慮にいれると，提案方式の優位性が少なくなった．提案した負荷算出式は，タスクの残り実行時間や余裕時間といった常に変化するパラメータを含むため，通信遅延によって算出結果が古くなったことが原因である．

第5章　関連研究

5.1　Cheddar

Cheddar[15] とは，フランス Brest 大学にて開発中のリアルタイムスケジューリングシミュレータである．教育目的で開発されており，GPL に従うフリーソフトウエアとして配布されている．Ada, GtkAda で書かれており，Solaris, Linux, Windows に対応している．

タスクセットやプロセッサなどを定義してシミュレーションを行なうが，定義した情報は XML 形式で保存できるようになっている．本研究で開発したシミュレータと同様，様々なスケジューリングアルゴリズムを試すことができる．さらに，共有資源，共有バッファを用いたシミュレーションも可能である．

しかし，Cheddar はマルチプロセッサスケジューリングに対応していない．複数のプロセッサを用いたシミュレーションはできるが，どのタスクをどのプロセッサで実行するかはユーザーが指定しなければならない*．また，Cheddar ではスケジューリング等のオーバヘッドを考慮していない．

*マルチプロセッサスケジューリングは次期バージョンで対応予定とされている．

第6章　おわりに

6.1　まとめ

本研究では，分散リアルタイムシステムにおけるオンラインスケジューリングを対象とし，タスク割り当てのための新たな負荷の指標を提案した．負荷の指標に適応型動的優先度法の概念とタスクの余裕時間を導入し，またプロセッサ能力の違いを考慮した．

シミュレータを実装して様々な条件でシミュレーションを行ない，結果の傾向を分析した．プロセッサ間の通信時間が十分に短い場合，提案手法によって，静的優先度が高いタスクのデッドラインオーバーを減少できることが分かった．

6.2　今後の課題

今後の課題としては，以下の点が挙げられる．

- さらに他の条件でシミュレーションを行なう
- さらに他のスケジュール評価関数で結果を比較する
- タスクの順序制約，資源制約をシミュレートできるようにする
- 実装途中だったためシミュレーションの際に使用できなかったが，GS が負荷計算するパターンもシミュレートできるようにする

また，本論文の範囲を越えるため詳細には立ち入らなかったが，分散リアルタイムスケジューリングを考えるときには以下のような設計事項も問題となる．

- WCET を求める問題．本研究では WCET が既に求まっている前提でタスクを定義したが，WCET を求める問題は NP 完全問題である．
- 通信の問題．本研究では通信が全て固定時間で行なわれるとしたが，リアルタイム通信を実現するには専用のプロトコルなどが必要である．
- クロック同期問題．本研究では，GS と各プロセッサで同一の大域時間が存在するという仮定をした．分散システムにおいて同期をとることは，上記の通信に関する問題もあり容易ではない．

謝辞

本研究を進めるにあたり，熱心かつ懇切丁寧にご指導を賜りました，田中 清史助教授に深く感謝いたします.
貴重な助言や指摘をしてくださいました，日比野 靖教授，堀口 進教授に深く感謝いたします.

以下の田中研究室のメンバーに感謝します.
同じリアルタイムスケジューリング関連の研究に携わる仲間として，互いに悩みを相談しあったり研究資料を共有したりした，大崎哲弥氏に感謝します.
評価結果のグラフを，Excel で作成した後 EPS に変換する方法を教えてくれた越前智史氏に感謝します。
シンプルに保とうと努めた結果 殺風景になりがちだった私の机や本棚をいつも賑やかに飾ってくれた山本達也氏と吉兼 寛氏に感謝します.
中間審査のリハーサル後，研究室メンバーの中で私の発表が最も理解しやすいと褒めてくださった今井俊晴氏に感謝します．単に研究内容が簡単だからという気もしましたが，それでも大変嬉しく，励みになりました.
役立ちそうな記事が購読雑誌に載っていればすぐに持ってきてくださった，荻野 雅氏に感謝します.
同じく田中研究室の先輩，後輩の皆様に感謝します.

計算機アーキテクチャ講座 日比野研究室の皆様に感謝します.

参考文献

[1] G. C. Buttazzo, "Hard Real-Time Computing Systems : Predictable Scheduling Algorithms and Applications," Kluwer Academic Publishers, 1997.

[2] G. C. Buttazzo and J. A. Stankovic, "RED: Robust Earliest Deadline Scheduling," Technical Report UMCS-1993-025, University of Massachusetts, July 1993.

[3] M. L. Dertouzos, A. K. Mok, "Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks," IEEE Transaction on Software Engineering, Vol. 15, No. 12, Dec. 1989.

[4] S. K. Dhall and C. L. Liu, "On a Real-Time Scheduling Problem," Operations Research, Vol. 26, No.1, pp. 127–150, 1978.

[5] 土井美和子，天野真家 (編)，"人間支援のための分散リアルタイムネットワーク," 情報処理学会，Vol. 44, No. 1, pp. 1–51, Jun. 2003.

[6] 藤倉俊幸，"リアルタイム/マルチタスクシステムの徹底研究," インターフェース増刊 TECH I, Vol. 15, CQ 出版，Jan. 2003.

[7] R. L. Graham, "Bounds on the Performance of Scheduling Algorithms," Computer and Job Scheduling Theory, John Wiley and Sons, pp. 165-227, 1976.

[8] 亀田恒彦，山下雅史，"アルゴリズムシリーズ 6 分散アルゴリズム," 近代科学社，July 1994.

[9] T. Kunz, "The Influence of Different Workload Description on a Heuristic Load Balancing Scheme," IEEE Transactions on Software Engineering, pp. 725–730, July 1991.

[10] 栗谷一路，"リアルタイム OS における優先度を用いた動的スケジューリングの研究," 北陸先端科学技術大学院大学 情報科学研究科 修士論文，Mar. 2003.

[11] J. Y. T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks," Performance Evaluation, Vol. 2, No. 4, pp. 237–250, Dec. 1982.

[12] 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, 1973.

[13] R. McNaughton, "Scheduling with Deadlines and Loss Functions," Management Science, Vol. 6, No. 1, pp. 1–12, Oct. 1959.

[14] A. K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," Ph.D. thesis, MIT Cambridge MA, 1983.

[15] "The Cheddar project : a free real time scheduling simulator," http://beru.univ-brest.fr/~singhoff/cheddar/

[16] SPARC International, Inc., "The SPARC Architecture Manual: Version 8," Prentice Hall, 1992.

[17] J. A. Stankovic, "Distributed Real-Time Computing: The Next Generation," Invited paper for Jounal of the Society of Instrument and Control Engineers of Japan, Vol. 31, No. 7, pp. 726–736, Jan. 1992.

[18] J. A. Stankovic, M. Spuri, M. D. Natale, and G. C. Buttazzo, "Inplications of Classical Scheduling Results for Real-Time Systems," IEEE Computer, Vol. 28, No. 6, pp. 16–25, June 1995.

[19] A. S. Tanenbaum (著)，水野忠則 他 (訳)，"分散オペレーティングシステム，" プレンティスホール，Aug. 1996.