Vol. 48 No. SIG 3(ACS 17) 情報処理学会論文誌：コンピューティングシステム Feb. 2007

Pfair スケジューリングにおけるコンテキストキャッシュの有効利用

船 岡 健 司[†] 加 藤 真 平[†] 山 崎 信 行[†]

Pfair スケジューリングは，複数の実行単位（コンテキスト）が存在する周期リアルタイムタスクのスケジューリングにおいて，現在知られている唯一の最適な手法である．しかしながら，Pfair スケジューリングでは，コンテキストスイッチやタスクマイグレーションが頻繁に起こるので，そのまま実際のシステムに応用するとパフォーマンスが大幅に低下する可能性がある．本論文では，Pfair スケジューリングのオーバヘッドを削減することを目的として，コンテキストキャッシュを有効に利用する手法を提案する．評価の結果，タスクの実行が分散されオーバヘッドが大きくなる状況下において，提案アルゴリズムが有効であることを示した．

An Effective Use of the Context Cache for Pfair Scheduling

Kenji Funaoka,[†] Shinpei Kato[†] and Nobuyuki Yamasaki[†]

Pfair scheduling is the only known optimal way for scheduling recurrent real-time tasks on multi-context environments. To apply Pfair scheduling to practical use causes the degradation of the performance due to frequent context-switches and task migrations. In this paper, we propose methods of the effective use of the context cache for Pfair scheduling to reduce these overheads. The simulation results show that the proposed algorithm is effective under dispersed task execution and high overhead conditions.

1. 序 論

テクノロジの発展と革新的なアイディアにより，プロセッサの性能は劇的に向上している．しかしながら，プロセッサの性能向上に起因して，熱や消費電力の増加が大きな問題となっており，1 プロセッサあたりの性能向上に限界が見えてきた．したがって，スレッドレベルの並列性に着目した Simultaneous Multithreading (SMT)[1] や Chip Multiprocessing (CMP)[2] が注目されている．組み込みシステムでは，パフォーマンスだけでなく消費電力やコストも重要である．それゆえ，SMT や CMP が広く用いられるようになると考えられる．また，組み込みシステムでは，ロボット制御や画像処理のように時間制約を持つシステムが多い．

リアルタイムシステムとは，処理の結果が論理的な正しさだけでなく，処理の完了した時刻にも依存するシステムである．このようなシステムでは，タスクのスケジュールを行うことにより，タスクに課せられた時間制約を守らなければならない．リアルタイム性を要求するタスクは，周期的な実行を行うものが多く，周期タスクに関する様々な研究がなされている．

SMT や CMP のように複数の実行単位（コンテキスト）が存在する場合，スケジューリングの手法は，パーティショニングとグローバルスケジューリングに大別される．パーティショニングとは，あらかじめタスクをそれぞれのコンテキストに割り振り，それぞれのコンテキストで独立してスケジューリングを行う手法である．グローバルスケジューリングとは，タスクをグローバルなキューに格納し，実行時にタスクを各コンテキストに動的に割り振る手法である．

パーティショニングでは，各コンテキストに割り振られたタスクは，シングルプロセッサのアルゴリズムでスケジュールすることが可能である．シングルプロセッサのアルゴリズムは古くから研究されており，Earliest Deadline First（EDF）が最適であると証明されている[3]．しかしながら，各コンテキストにタスクを割り振る操作は，箱詰問題に還元でき，NP 困難問題として知られている．よって，オンラインでのタスク割振りは，First Fit（FF）などのヒューリスティックな手法を利用した EDF-FF[4] などが用いられる．ここで厳密なアルゴリズムを用いると，大きなオーバヘッドが発生する可能性がある．したがって，オンラインで最適なスケジュールを保証することは困難である．

† 慶應義塾大学
Keio University

グローバルスケジューリングにおいてシングルプロセッサのアルゴリズムを使用した場合，Dhall's effect [5] により，スケジュール可能性が大きく低下する．EDF を改良した EDF-US[x] においても，コンテキスト数を M とすると，理論的に保証可能なタスクセットの重みは $(M+1)/2$ である[6]．保証可能なタスクセットの重みとは，すべての利用可能なプロセッサ時間を M としたとき，デッドラインミスすることなく利用可能なプロセッサ時間である．すでに多くのコンテキストを持つ SMT や CMP が実現されているにもかかわらず，すべてのデッドラインを守るには約半分の性能しか利用することができないということを示している．

Proportionate-fair [7]（Pfair）スケジューリングは，複数のコンテキストが存在する環境において，現在知られている唯一の最適なスケジューリング手法である．Pfair スケジューリングでは，タスクは固定長の特定時間ごとにスケジュールされる．それぞれのタスクには重み w が与えられ，すべての区間 L において，タスクは $w \cdot L$ の時刻が与えられる．Pfair スケジューリングをプロセッサスケジューリングに適用すると，頻繁なコンテキストスイッチとタスクマイグレーションにより，オーバヘッドが大きくなる．しかしながら，オーバヘッドを考慮しても，EDF-FF にひけをとらないという報告があり[8]，現実的な選択肢の 1 つとなりうる．また，Pfair スケジューリングを行うことにより，様々な恩恵を得ることが可能となる[8]．

リアルタイムスケジューリングアルゴリズムの最適性の証明は，スケジューリングのオーバヘッドが存在しないという仮定で行われる．従来のアルゴリズムでは，Pfair スケジューリングほどの頻繁なコンテキストスイッチが起こらないため，タスクの最悪実行時間にオーバヘッドを含めてしまうなどの手法により，大きな問題となることはなかった．しかしながら，Pfair スケジューリングでは，オーバヘッドにより大きくパフォーマンスが低下する．よって，効率的にプロセッサ資源を利用するためにオーバヘッドを削減する必要がある．オーバヘッドを削減する手法として，コンテキストスイッチの回数を削減する手法とコンテキストスイッチあたりのオーバヘッドを削減する手法が考えられる．本論文では，コンテキストキャッシュと呼ばれる技術を有効利用して後者の手法に焦点を当てる．

2. Pfair スケジューリング

Pfair スケジューリングでは，リアルタイムシステムを以下のようにモデル化する．システムのタスク数を N，コンテキスト数を M とし，タスクの集合をタスクセット τ とする．周期タスク T（$\in \tau$）は，最悪実行時間 $T.e$ と周期 $T.p$ によって特徴付けられる．タスク T の相対デッドラインは，周期 $T.p$ と等しい．$wt(T) = T.e/T.p$ をタスクの重みと呼び，$0 < wt(T) \leq 1$ である．$wt(T) \leq 1/2$ のタスクを light タスク，$wt(T) > 1/2$ のタスクを heavy タスクという．$\sum_{\forall T \in \tau} wt(T)$ をタスクセットの重みという．

Pfair スケジューリングでは，プロセッサ時間を固定長の時間に区切る．時間の間隔 $[t, t+1)$ を slot t と呼ぶ．時刻 t とは slot t の開始時刻である．各コンテキストの 1 つの slot では，最大で 1 つのタスクを実行可能である．タスクは，slot ごとに異なるプロセッサ上で実行される可能性があるが，同じ slot において 1 つのタスクを複数のコンテキスト上で並列実行することはできない．slot へのタスクの割当ては，式 (1) のスケジュール S によって定められる[7]．

$$S : \tau \times N \rightarrow \{0, 1\} \qquad (1)$$

タスク T が slot t へスケジュールされたときに限り，$S(T, t) = 1$ である．ただし，すべての t について $\sum_{\forall T \in \tau} S(T, t) \leq M$ である．Pfair を定義するために，lag という概念を導入する．タスク T の slot t における lag は，式 (2) で示される．lag は，理想の割当てと実際の割当ての差である．

$$lag(T, t) = wt(T) \cdot t - \sum_{u=0}^{t-1} S(T, u) \qquad (2)$$

スケジュール S が式 (3) を満たすとき，スケジュール S は Pfair であるという．

$$(\forall T, t :: -1 < lag(T, t) < 1) \qquad (3)$$

これは，理想のプロセッサへの時間の割当てと実際の時間の割当ての差が，つねにある一定の値 1 の間に収まっていることを意味する．式 (3) を満たすために，タスク T は固定長のサブタスクに分割される．タスク T の i 番目（$i \geq 1$）のサブタスクを T_i と表す．それぞれのサブタスクは，疑似リリース時刻 $r(T_i)$ と疑似デッドライン $d(T_i)$ を持ち，式 (4) で表される．

$$r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \quad d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil \qquad (4)$$

サブタスクは，疑似リリース時刻と疑似デッドラインの間に実行されなければならない．これをサブタスク T_i のウィンドウと呼ぶ．例として，$wt(T) = 8/11$ となるタスク T のウィンドウの様子を**図 1** に示す．この図は，Srinivasan ら[8] の図を参考にした．

最適な Pfair スケジューリングアルゴリズムとして，PF と PD，PD^2 が提案されている[7),9),10)]．理論的に

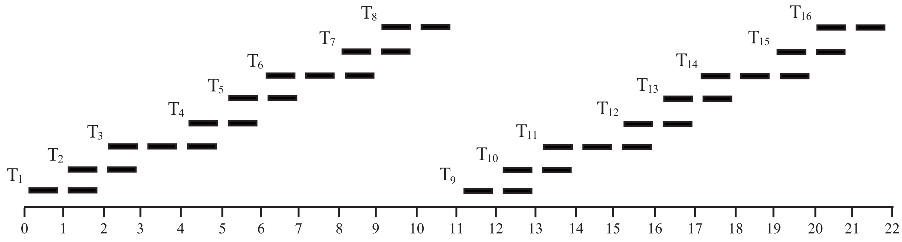

図 1 $wt(T)=8/11$ のウィンドウ
Fig. 1 Windows when $wt(T)=8/11$.

保証可能なタスクセットの重みは式 (5) で表される．これらのアルゴリズムでスケジュール不可能なタスクセットは，理論的な面のみをとらえると，いかなるアルゴリズムをもってしてもスケジュールできない．

$$\sum_{\forall T \in \tau} wt(T) \leq M \tag{5}$$

これらのアルゴリズムは，疑似デッドラインの早いサブタスクに高い優先度を与えてスケジュールを行うが，疑似デッドラインが等しい場合の判断手法が異なる．このうち，PD^2 が最も効率的に実装可能である．

PD^2 の 1 つ目の判断基準は，b ビット $b(T_i)$ である．b ビットは，現在のサブタスクのウィンドウが直後のウィンドウと重なっているときに 1 となる．PD^2 は，b ビットが 1 のタスクを優先してスケジュールする．これは，後続のサブタスクの実行可能時間を最大限に伸ばすためである．b ビットは，式 (6) で表される[11]．

$$b(T_i) = \left\lceil \frac{i}{wt(T_i)} \right\rceil - \left\lfloor \frac{i}{wt(T_i)} \right\rfloor \tag{6}$$

PD^2 の 2 つ目の判断基準は，グループデッドライン $D(T_i)$ である．グループデッドラインとは，連続してサブタスクの実行を行ったときに，その実行が必ず途切れる時刻である．PD^2 は，グループデッドラインの遅いタスクを優先して実行する．これは，グループデッドラインの早いタスクを後回しにすることにより，同じタスクを連続して実行する可能性が高まるためである．グループデッドラインは，式 (7) で表される[11]．

$$D(T_i) = \left\lceil \frac{\left\lceil \left\lceil \frac{i}{wt(T_i)} \right\rceil \times (1 - wt(T_i)) \right\rceil}{1 - wt(T_i)} \right\rceil \tag{7}$$

2.1 Pfair スケジューリングのオーバヘッド

Pfair スケジューリングは，slot ごとに全コンテキストのスケジュールを行わなければならない．よって，従来のアルゴリズムよりもスケジューリングのオーバヘッドが大きくなる．Srinivasan ら[8] は，これらのオーバヘッドを定式化している．スケジューリングには，大きく分けて 2 つの操作が必要である．1 つ目は，実行するタスクの選択である．これには，スケジュールごとにタスクのキューを操作しなくてはならない．2 つ目は，タスクの切替えである．これには，前に実行していたタスクを復元可能な状態にし，次に実行するタスクを実行可能な状態にする．本節では，コンテキストスイッチやタスクマイグレーションによるオーバヘッドに焦点を当てる．コンテキストスイッチやタスクマイグレーションによるオーバヘッドは，大きく分けて以下の 2 点を考えることができる．

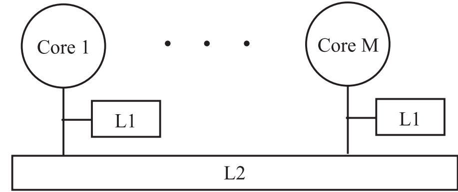

図 2 CMP アーキテクチャ例
Fig. 2 An example of CMP architecture.

第 1 点目は，コンテキストスイッチ自体のオーバヘッドである．コンテキストスイッチ時には，レジスタの値などのコンテキスト情報をメインメモリに保存して復元可能な状態にしなくてはならず，レイテンシの長い多くのメモリアクセス命令が必要となる．

第 2 点目は，コンテキストスイッチやタスクマイグレーションによって引き起こされる，ハードウェア資源の競合によるオーバヘッドである．競合するハードウェア資源には，キャッシュや TLB などがあげられる．ここでは，CMP におけるキャッシュを例にオーバヘッドを説明する．キャッシュの容量は限られているため，コンテキストスイッチが起こると他のタスクによりキャッシュが上書きされる．**図 2** に CMP のアーキテクチャ例を表し，このアーキテクチャから想定されるオーバヘッドを示す．このアーキテクチャでは，タスクマイグレーションが起こると，そのタスクに関する L1 キャッシュのエントリがすべて無効となる．また，頻繁なコンテキストスイッチにより，L1 だけでなく L2 の競合が発生し，大きなオーバヘッドと

なる．特に，タスク数が多く，頻繁にコンテキストスイッチが行われると，オーバヘッドが顕著に現れると考えられる．よって，頻繁なコンテキストスイッチやタスクマイグレーションは，タスクに依存したハードウェア資源に大きな影響を与える．

Pfair スケジューリングは，これらのオーバヘッドが存在しないという仮定の下で最適性が証明されている．実システムにおいて時間制約を守るには，これらのオーバヘッドを隠蔽しなくてはならない．多くのシステムでは，オーバヘッドをタスクの実行時間に含めて考えることにより隠蔽している．しかしながら，Pfair スケジューリングでは，オーバヘッドが非常に大きくなることから，システムのパフォーマンスが大幅に低下する可能性がある．よって，オーバヘッドを削減することにより，複数コンテキストが存在するシステムを有効に利用することが可能となる．

3. コンテキストキャッシュ

コンテキストキャッシュ[12]とは，プロセッサ内に各種レジスタ値などのコンテキスト情報を格納することが可能な専用オンチップメモリである．ソフトウェアで専用の命令を実行することにより，コンテキスト情報をコンテキストキャッシュに保存することが可能である．コンテキストキャッシュを利用することにより，Responsive Multithreaded Processor（RMT Processor）[12] では，コンテキストスイッチにソフトウェアでは 590 クロック必要であったものが，ハードウェアにより 4 クロックで実現可能となっている．我々の知る限り，RMT Processor はコンテキストキャッシュが実装されている唯一の実プロセッサであるが，それ以外のプロセッサにも容易に実装可能である．よって，今後は様々なシステムに利用されていくと考えられる．

RMT Processor は，優先度付 8 スレッド同時実行の SMT であり，命令を実行可能な 8 つのアクティブスレッドと，コンテキストをキャッシュ中に保存可能な 32 のキャッシュスレッドが存在する．**図 3** に，RMT Processor のコンテキストキャッシュを示す．アクティブスレッドの各種レジスタは，専用バスによりコンテキストキャッシュと接続されている．RMT Processor のスレッドの状態遷移図を**図 4** に示す．RMT Processor では，ソフトウェアからの命令によりスレッドを状態遷移させる．これにより，コンテキストをコンテキストキャッシュに格納することが可能となる．本論文では，RMT Processor に従い，実行可能なスレッドをアクティブスレッド，コンテキストキャッシュ中のスレッドをキャッシュスレッドと呼ぶことにする．**図 5** に，RMT Processor におけるコンテキストスイッチの概念例を示す．図 5 の 19 行目において，コンテキストキャッシュにエントリの空きがない場合，コンテキストキャッシュに収めるタスクを選択する必要がある．コンテキストキャッシュには，様々なレジスタの値を保存しなければならず，通常のキャッシュと比較して，1 エントリあたりの面積が大きくなる．したがって，多くのエントリをプロセッサに実装することは困難であり，どのコンテキストをコンテキストキャッシュに収めるかの判断は非常に重要である．

コンテキストキャッシュに格納しておいたコンテキストは，実行する際にコンテキストキャッシュに格納したままにしておく必要がない．このことから，実行中のコンテキストとコンテキストキャッシュに格納されているコンテキストをスワップすることにより，コンテキストキャッシュを有効に利用することが可能となる．メモリキャッシュやディスクキャッシュなどの従来のキャッシュシステムでは，読み込みや書き込みの対象エントリが追い出されるということはない．よって，コンテキストキャッシュと従来のキャッシュシス

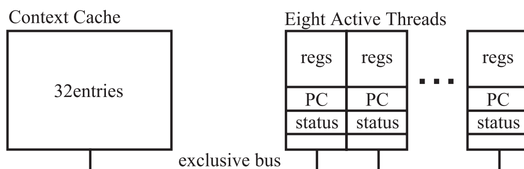

図 3 RMT Processor におけるコンテキストキャッシュ
Fig. 3 The context cache on RMT Processor.

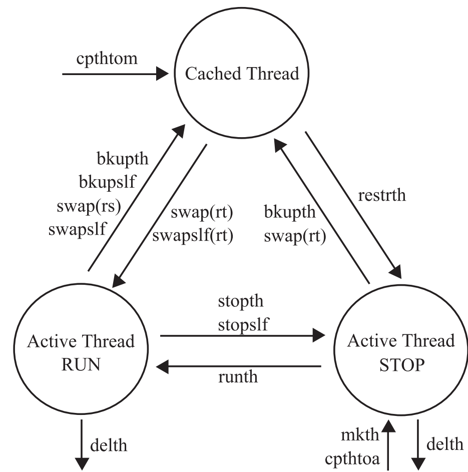

図 4 RMT Processor におけるスレッドの状態遷移図
Fig. 4 State transition chart of threads on RMT Processor.

```
Algorithm: ContextSwitch
 1: Let 𝒜 = {A_1, ..., A_M} be the active threads
 2: Let 𝒞 = {C_1, ...} be the cached threads
 3: Let P_i be the prev context on A_i
 4: Let N_i be the next context on A_i
 5:
 6: for all i such that 1 ≤ i ≤ M do
 7:     if P_i ≠ N_i then
 8:         if N_i ∈ 𝒞 then
 9:             swapth
10:         else if vacant entries of 𝒞 exist then
11:             if P_i exists then
12:                 cpthtom
13:             end if
14:             if N_i exists then
15:                 restore N_i from memory
16:             end if
17:         else
18:             if P_i exists then
19:                 if P_i goes to 𝒞 then
20:                     swapth
21:                 end if
22:                 save current context to memory
23:             end if
24:             if N_i exists then
25:                 restore N_i from memory
26:             end if
27:         end if
28:     end if
29: end for
```

図 5 RMT Processor におけるコンテキストスイッチ概念例
Fig. 5 A concept example of context switches on RMT Processor.

テムを同列に論じることはできない．

従来のソフトウェアのみによるコンテキストスイッチとの比較を行うために，ソフトウェアによるコンテキストスイッチの回数 n(soft-switch) を式 (8) のように定義する．ただし，n(soft-load) をソフトウェアによるコンテキスト復元の回数，n(soft-save) をソフトウェアによるコンテキスト保存の回数とする．

$$n(\text{soft-switch}) = \frac{n(\text{soft-load}) + n(\text{soft-save})}{2} \tag{8}$$

4. コンテキストキャッシュ置換アルゴリズム

すべてのタスクがコンテキストキャッシュに収まる場合，つねにコンテキストキャッシュの恩恵を受けることができる．しかしながら，タスク数がコンテキストキャッシュに収まる数を超える場合，コンテキストキャッシュを効率良くタスクに割り当てることが重要であり，コンテキストキャッシュ置換アルゴリズムが必要とされる．置換対象を決定するのに，ソフトウェアによるコンテキストスイッチより時間がかかると，コンテキストキャッシュを利用する意義が失われる．よって，非常に単純なアルゴリズムが求められる．

4.1 Farthest Weight[1/2]

スケジューリングアルゴリズム A とタスクセット τ が決定したとき，任意の時間の厳密なコンテキストスイッチ数を正確に導くことは困難である．よって，最もコンテキストスイッチ数が大きくなる状況を想定し，最悪コンテキストスイッチ数を以下のように定義する．

定義 1（最悪コンテキストスイッチ数） 連続して実行するサブタスク間ではコンテキストスイッチが起こらないと仮定したとき，すべてのスケジュール S において，最大のコンテキストスイッチ数を最悪コンテキストスイッチ数と定義する． ■

タスクが 1 つであれば，スケジュールは予測可能である．しかしながら，複数のタスクを実行する場合，スケジュールを動的に予測することは困難である．これは，複数のタスクを実行すると，実行中のタスクより優先度の低いタスクは実行できないというスケジュールの制約が生まれることによる．このことから，コンテキストスイッチ数を正確に予測することは難しい．補題 1 により，最悪コンテキストスイッチ数が最大となる場合を単純化して考えることが可能となる．

補題 1 タスク T について，他タスクからスケジュールの制約を受けるときの最悪コンテキストスイッチ数を $C(T)$，他タスクからスケジュールの制約を受けないときの最悪コンテキストスイッチ数を $C'(T)$ とする．$C(T)$ と $C'(T)$ の間には，$C(T) \leq C'(T)$ の関係が成立する．

証明 他タスクからスケジュールの制約を受けるときに可能なスケジュールの集合を $\mathcal{S}$，他タスクからスケジュールの制約を受けないときに可能なスケジュールの集合を $\mathcal{S}'$ とする．$\mathcal{S}$ と $\mathcal{S}'$ の間には，$\mathcal{S} \subseteq \mathcal{S}'$ の関係が成立する．よって命題は明らか． ■

これは，他のタスクを考慮せずにコンテキストスイッチ数が最大となるようにスケジュールすれば，そのときのコンテキストスイッチ数が，他のタスクによってスケジュールの制約がある状況での最悪コンテキストスイッチ数以上になることを示している．ここから，定理 1 により Pfair スケジューリングにおける最悪コンテキストスイッチ数が最大となる状況が導かれる．

定理 1 タスク T の重みを $wt(T)$ とする．Pfair スケジューリングにおいて，$wt(T) = 1/2$ のとき，タスク T の最悪コンテキストスイッチ数が最大となる．

証明 n を非負の整数とする．タスク T は，他タスクからスケジュールの制約を受けないと仮定す

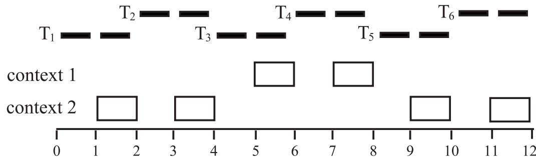

図 6 $wt(T)=1/2$ のときのスケジュール例
Fig. 6 An schedule example when $wt(T)=1/2$.

る．このとき，補題 1 より，最悪コンテキストスイッチ数が最大となる．タスク T が light タスクのとき，タスクを実行している時間より実行していない時間の方が長い．よって，いかなるサブタスクも連続して実行しないスケジュール S が存在する．このとき，$(n \cdot T.p, (n+1)T.p]$ における最悪コンテキストスイッチ数は，$2T.e$ である．単位 slot あたりの最悪コンテキストスイッチ数は，$2T.e/T.p = 2wt(T)$ となる．タスクが heavy タスクのとき，タスクを実行していない時間より実行している時間の方が長い．よって，いかなるサブタスクを実行しない slot も連続しないスケジュール S が存在する．このとき，$(n \cdot T.p, (n+1)T.p]$ における最悪コンテキストスイッチ数は，$2(T.p - T.e)$ である．単位 slot あたりの最悪コンテキストスイッチ数は，$2(T.p - T.e)/T.p = 2(1 - wt(T))$ となる．それゆえ，Pfair スケジューリングにおいて，タスク T の最悪コンテキストスイッチ数が最大となるのは $wt(T) = 1/2$ のときである． ■

定理 1 は，$wt(T) = 1/2$ のタスクがコンテキストスイッチを最も頻繁に起こす可能性があることを示している．$wt(T) = 1/2$ となるタスクが最悪コンテキストスイッチ数となる場合のスケジュール例を**図 6** に示す．この場合，このタスクにより，毎 slot に必ず 1 回のコンテキストスイッチが引き起こされる．Pfair スケジューリングアルゴリズムとして最も効率的な PD^2 は，複数のタスクが同じ疑似デッドラインを持つとき，b ビットとグループデッドラインにより $wt(T) \leq 1/2$ のタスクより $wt(T) > 1/2$ のタスクの方が優先する可能性が高いことから，特に重みの大きいタスクが存在するとこのような状態も起こりうる．

本論文では，重みが 1/2 から遠いタスクを置換するコンテキストキャッシュ置換アルゴリズム Farthest Weight[1/2]（FW）を提案する．FW は，タスク T の重みの遠さ $F(T)$ を式 (9) で定義し，$F(T)$ が最大となるタスクを含むキャッシュスレッドを置換する．

$$F(T) = |wt(T) - 1/2| \tag{9}$$

FW と従来のキャッシュシステムで広く用いられている Least Recently Used（LRU）や Least Frequently Used（LFU）の最大の相異点は，置換の判断に用いられる要素の所在である．FW では，キャッシュに収める対象に関する値を比較することによって置換を行う．それに対して，LRU と LFU はキャッシュエントリに関する値を比較することによって置換を行う．

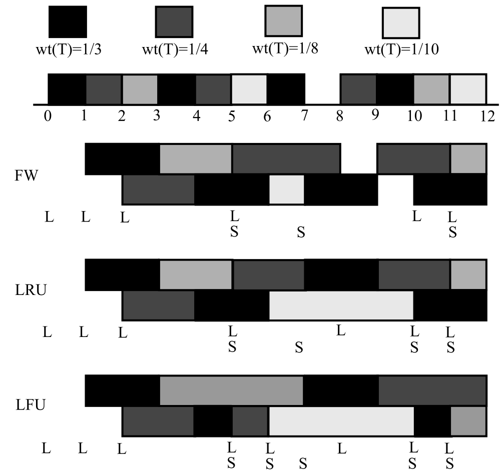

図 7 FW と LRU，LFU の比較例
Fig. 7 A comparison among FW, LRU and LFU.

FW と LRU，LFU の比較例を**図 7** に示す．アクティブスレッド数を 1，コンテキストキャッシュのエントリ数を 2 として，4 つのタスクをスケジュールした様子とキャッシュスレッドの様子を示した．それぞれのタスクの重みは，1/3，1/4，1/8，および 1/10 である．スケジューリングアルゴリズムは PD^2 である．キャッシュスレッドの下にソフトウェアによるメインメモリへの保存（S）と復元（L）が行われるタイミングを示した．LFU の場合，同じ利用頻度の場合にどのエントリを置換するか判断できないため，ここでは最悪のケースを示した．FW では，頻繁に実行する重み 1/3 のタスクに多くコンテキストキャッシュの割当てを行っている．LRU では，時刻 7 において重み 1/4 のタスクを置き換えてしまっていることから，FW と比較してソフトウェアによるコンテキストスイッチの回数が多い．この例の $[0, 12)$ におけるコンテキストスイッチ数は，式 (8) より，FW が 4.5 回，LRU が 5.5 回，LFU が 6.5 回であり，FW が最も優れている．

FW は，コンテキストスイッチ数が最大となるとき，最もコンテキストスイッチを起こすタスクを優先的にコンテキストキャッシュに格納するアルゴリズムである．したがって，コンテキストスイッチが頻繁に起こり，オーバヘッドが非常に大きくなる状況下で良い結果を出すアルゴリズムであると考えられる．

タスクの重みはタスクに固有の定数値である．よって，LRU や LFU のようにパラメータ値の更新を毎回

図 8 Strict 方式
Fig. 8 Strict Method.

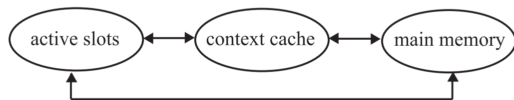

図 9 Lazy 方式
Fig. 9 Lazy Method.

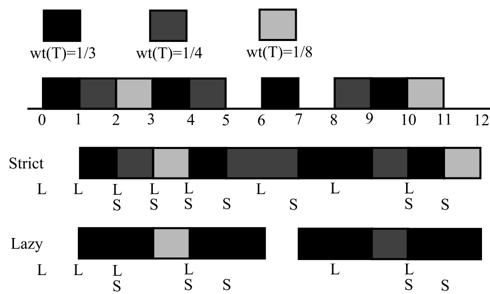

図 10 Strict と Lazy の比較例
Fig. 10 A comparison between Strict and Lazy.

行う必要はない．また，FW は実装方法により $O(M)$ もしくは $O(\log_2 M)$ で実現可能である．

4.2 置換方式

コンテキストキャッシュ置換方式として，以下の 2 つの手法がある．1 つ目は，**図 8** に示されるように，現在実行中のコンテキストをつねにコンテキストキャッシュに退避させる手法である．コンテキストキャッシュに空きがない場合は，置換アルゴリズムに基づいて，コンテキストキャッシュ内にあったコンテキストが追い出される．本論文では，この方式を Strict 方式と呼ぶ．2 つ目は，**図 9** に示されるように，実行中のコンテキストが直接メインメモリに格納される可能性のある手法である．キャッシュ内に含まれるコンテキストと現在のコンテキストを含む集合を考え，この集合の中から置換対象を選択する．現在実行中のコンテキストが選択された場合，直接メインメモリに格納される．本論文では，この方式を Lazy 方式と呼ぶ．

Strict 方式では，すでにキャッシュに格納されている要素の間のみで比較が行われる．Lazy 方式ではそれに加えて，これからキャッシュに格納される候補の要素も比較の対象となる．よって，これからキャッシュに格納される候補の要素も比較可能でなくてはならない．従来から広く用いられている LRU や LFU では，メタ情報としてキャッシュエントリの最終利用時刻や利用頻度が格納されている．これは，キャッシュに格納される要素自身の値ではないことから，Lazy 方式を適用することはできない．FW はキャッシュに格納される要素自身の重みを利用することから，Lazy 方式での置換を行うことが可能である．

従来のキャッシュシステムでは，多くが Strict 方式である．これは，キャッシュへのアクセスに局所性が大きいときに有効に働き，Lazy 方式と比較して情報を収める空間が少ない．しかしながら，Pfair スケジューリングでは，実行を分散させるという概念に基づいている．それゆえ，コンテキストキャッシュに格納されたエントリが即座に使われる可能性が低い．しかしながら，Strict 方式では，長時間使わない可能性の高いエントリでも必ずキャッシュに格納する．よって，Pfair のコンテキストキャッシュ置換アルゴリズムとして比較すると Lazy 方式が有効であると考えられる．

図 10 に Strict 方式と Lazy 方式の比較を示す．アクティブスレッド数を 1，コンテキストキャッシュのエントリ数を 1 として，3 つのタスクをスケジュールした様子とキャッシュスレッドの様子を示した．それぞれのタスクの重みは，1/3 と 1/4，および 1/8 である．スケジューリングアルゴリズムは PD^2，コンテキストキャッシュ置換アルゴリズムは FW である．キャッシュスレッドの下にソフトウェアによるメインメモリへの保存（S）と復元（L）が行われるタイミングを示した．Strict 方式では，あまり実行しないタスクも必ずコンテキストキャッシュに入ることから，Lazy 方式よりもメインメモリへの保存と復元が頻繁に行われている．逆に，Lazy 方式では多く実行されるコンテキストをコンテキストキャッシュに収めることができている．この例の $[0, 12)$ におけるコンテキストスイッチ数は，式 (8) より Strict 方式で 7.5 回，Lazy 方式で 5.5 回であり，Lazy 方式の方が優れている．

5. 評価

本章では，提案アルゴリズムと従来のキャッシュ置換アルゴリズムを比較して，提案アルゴリズムの方が Pfair スケジューリングにおいてコンテキストキャッシュを有効に利用できていることを示す．コンテキストキャッシュを有効に利用することによって，Pfair スケジューリングのオーバヘッドを削減してシステム全体のパフォーマンスを向上させることが可能となる．

5.1 評価方法

評価の指標となるキャッシュミス率とオーバヘッドを以下のように定義する．ソフトウェアによるコンテキストスイッチ数を n(soft-switch)，全コンテキスト

スイッチ数を $n(\text{switch})$ として，キャッシュミス率を式 (10) により定義した．

$$\text{miss ratio} = \frac{n(\text{soft-switch})}{n(\text{switch})} \quad (10)$$

また，1 slot の長さを $L(\text{slot})$，コンテキストスイッチに必要な時間を $T(\text{switch})$ として，オーバヘッドの割合を式 (11) により定義した．タスクは，このオーバヘッド以外のプロセッサ時間を利用して実行することが可能である．

$$\text{overhead ratio} = \frac{T(\text{switch})}{L(\text{slot})} \quad (11)$$

オーバヘッドの割合は，slot の長さによって大きく変化する．slot は，システムに求められる時間粒度によってシステムごとに異なる長さとなる．よって，絶対的な値ではなく，相対的な比に着目する必要がある．

評価の想定環境を述べる．現在コンテキストキャッシュが実装されているプロセッサは RMT Processor のみであることから，RMT Processor を想定してシミュレーションパラメータを決定した．プロセッサは，周波数 100 MHz で動作し，8 つのコンテキストと 32 エントリのコンテキストキャッシュを持つ．コンテキストスイッチは，ソフトウェアで行うと 590 クロック，コンテキストキャッシュを用いると 4 クロック必要となる．コンテキストキャッシュを利用しない場合，メモリキャッシュなどをコンテキストキャッシュの代わりに利用すれば 590 クロックよりも時間を短縮できる可能性がある．しかしながら，タスクの実行に利用できるメモリキャッシュ量が減少することからトレードオフの関係にあり，スワップ操作も行えない．本評価ではつねにメモリのみを利用した 590 クロックとする．本論文ではコンテキストの保存と復元のみに焦点を当てることから，コンテキストスイッチ時間にタスクキューの操作などの時間は含めない．

評価は，以下に示すシステムを想定したワークロードを用いて行った．ワークロード BI1 は，制御システムを想定した．制御システムでは，演算時間および周期が短く，時間制約の厳しいハードリアルタイムタスクが多い．ワークロード NO は，ロボットのような複合的なシステムを想定した．ロボットでは，BI1 で示したような制御タスクや，画像認識のように実行時間の長いタスクなど様々なタスクが混在する．NO と BI1 の 2 つの分布を用いることにより，異なる分布を持つタスクセットにおける置換アルゴリズムの比較を行う．評価では，単純な一様分布と二項分布を用いるが，実際のシステムでは様々な分布を持つことになる．

評価に用いたワークロードを**表 1** に示す．それぞれ

表 1 シミュレーションのワークロード
Table 1 Simulation workloads.

	分布	コンテキスト数	キャッシュ数	slot 長
NO-1	一様	8	32	1 ms
NO-01	一様	8	32	0.1 ms
BI1-1	二項	8	32	1 ms
BI1-01	二項	8	32	0.1 ms

のワークロードに対し，重みの異なるタスクセットを実行した．各ワークロードについて，タスクがすべてコンテキストキャッシュに収まってしまった場合の評価は示していない．すなわち，NO についてはタスクセットの重み 5.6 以上，BI1 についてはタスクセットの重み 3.2 以上の評価を示す．比較するアルゴリズムは，LRU と LFU，FW，および FW の Lazy 方式である FW-Lazy である．また，オーバヘッドの結果については，コンテキストキャッシュを利用しないときの結果を Software，コンテキストキャッシュが無限にあると仮定したときの結果を Hardware として示した．

評価に使用したタスクセットの生成方法について述べる．すべてのタスクの到着時刻を時刻 0 とする．タスクの周期を 100 以下の自然数からランダムに選択し，タスク重みが一様分布か二項分布となるように最悪実行時間を設定する．タスクの重みの和が目標値を超えた場合，そのタスクを破棄し，新しいタスクを再度生成する．タスクの生成に 100 回失敗した段階で生成を終了し，最後にタスクセットが目標値に到達する重みとなるタスクを生成し，これをタスクセットとする．タスクセットのタスク数がコンテキストキャッシュのエントリ数以下の場合，全タスクがキャッシュスレッドに収まってしまうことから，タスクセットを再度生成する．この操作を繰り返し，各ワークロードのタスクの重みごとに 100 のタスクセットを生成した．

重みが一様分布となるタスクセットの生成は，目標値 V に対して，$0 < wt(T) \leq \min(1, V)$ となるように一様乱数から重みを決定した．重みが二項分布となるタスクセットの生成は，以下の試行により行った．成功率が 0.1 の試行を 100 回行い，試行が成功した回数を 100 で割り，その結果を重みとした．

評価は，上記の各タスクセットを時刻 0 からハイパーピリオドまで実行することにより行った．ハイパーピリオドとは全タスクの周期の最小公倍数 $\text{lcm}(\forall T.p)$ であり，ハイパーピリオドごとに同じスケジュールが繰り返される．スケジューリングに用いたアルゴリズムは，PD^2 である．この際，同じタスクを 2 slot 以上連続して実行する場合は，そのタスクを同じコンテキストに割り当てた．この操作を 100 のタスクセットす

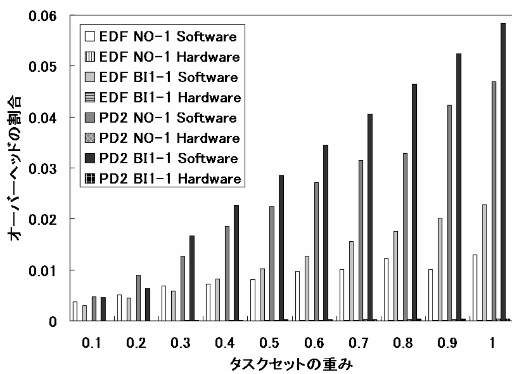

図 11 EDF と PD^2 のオーバヘッドの割合
Fig. 11 The overhead ratio of EDF and PD^2.

べてについて行い，それらの結果の平均値を求めることによりキャッシュミス率とオーバヘッドの割合を算出した．

5.2 予備評価結果

コンテキストキャッシュ置換アルゴリズムの比較を示す前に EDF と PD^2 のオーバヘッドの割合の比較を**図 11** に示す．複数コンテキストでの最適なスケジューリング手法は Pfair しか知られていないことから，シングルプロセッサアルゴリズムとして最適な EDF との比較を行った．両者の評価環境を揃えるため，コンテキスト数は 1 とした．オーバヘッドの割合の算出に用いたワークロードは NO-1 と BI1-1 である．Software の結果のタスクセットの重み 1 において，PD^2 は EDF と比較して NO-1 で約 3.6 倍，BI1-1 で約 2.6 倍のオーバヘッドが発生している．よって，PD^2 では EDF よりオーバヘッドが大きくなることが読み取れる．Software によるオーバヘッドが Hardware では大幅に削減されておりコンテキストキャッシュの有効性を確認することできる．

5.3 評価結果

タスク数と単位 slot あたりのコンテキストスイッチ数，キャッシュミス率では，ワークロード NO-1 と NO-01，BI1-1 と BI1-01 は同じ結果となることから，それぞれワークロード NO と BI1 として結果を示した．

タスク数と単位 slot あたりのコンテキストスイッチ数をそれぞれ**図 12** と**図 13** に示す．タスク数は，タスクセットの重みが増えるにつれて BI1 で大きく増加しているが，NO ではほとんど変化していない．よって，NO は BI1 と比較してコンテキストキャッシュに収まりやすい．単位 slot あたりのコンテキストスイッチ数は，NO で緩やかに増加しているのに対し，BI1 ではタスクセットの重みが増えるにつれて急激に増加している．よって，結果的に NO よりも BI1 では大き

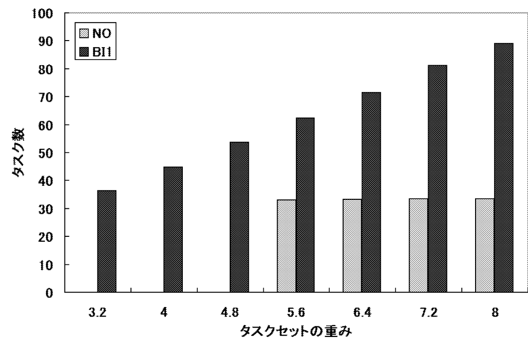

図 12 タスク数
Fig. 12 The number of tasks.

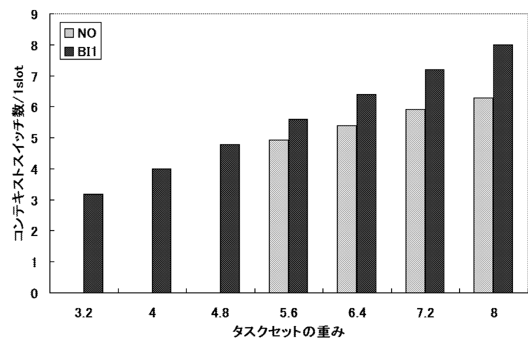

図 13 単位 slot あたりのコンテキストスイッチ数
Fig. 13 The number of context switches per 1 slot.

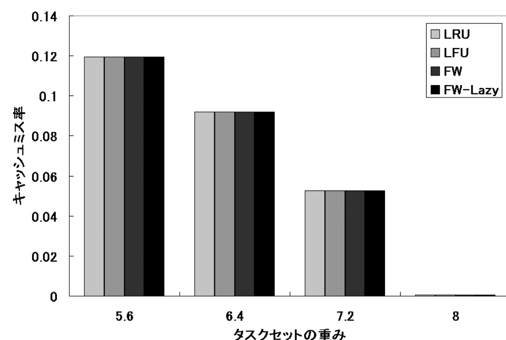

図 14 ワークロード NO におけるキャッシュミス率
Fig. 14 The cache miss ratio on work-load NO.

なオーバヘッドが発生することになる．タスクセットの重み 8 において，BI1 が 8 に近くなり全コンテキストの毎 slot でコンテキストスイッチが発生している．

ワークロード NO におけるキャッシュミス率を**図 14** に示す．NO では，タスクセットの重みが増加するにつれてキャッシュミス率が減少している．これは，タスクセットの重みが 8 に近くなるにつれて，空き slot が少なくなることによりキャッシュにヒットしやすくなっていること，また，タスクの重みが一様分布であることからキャッシュへのアクセスの局所性が高まっ

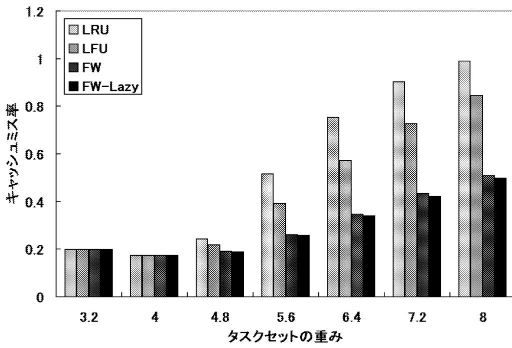

図 15　ワークロード BI1 におけるキャッシュミス率
Fig. 15　The cache miss ratio on work-load BI1.

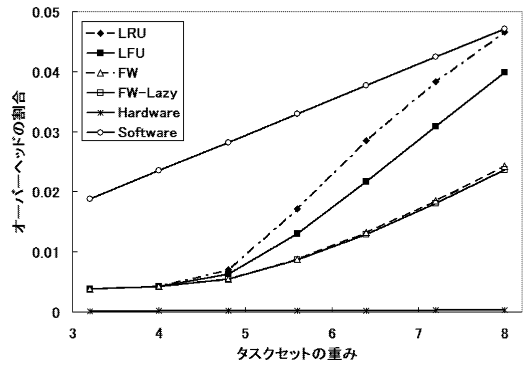

図 18　ワークロード BI1-10 におけるオーバヘッドの割合
Fig. 18　The overhead ratio on work-load BI1-10.

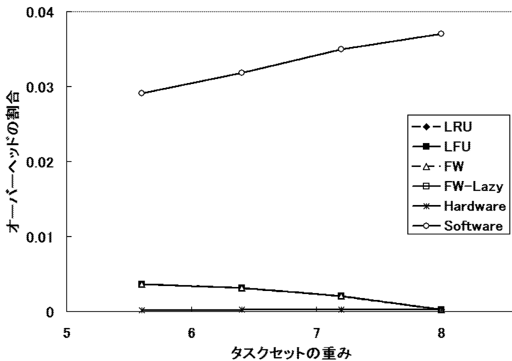

図 16　ワークロード NO-10 におけるオーバヘッドの割合
Fig. 16　The overhead ratio on work-load NO-10.

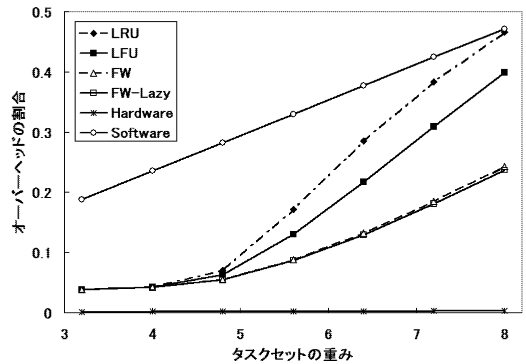

図 19　ワークロード BI1-01 におけるオーバヘッドの割合
Fig. 19　The overhead ratio on work-load BI1-01.

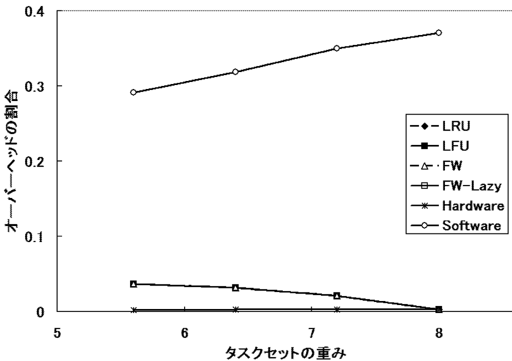

図 17　ワークロード NO-01 におけるオーバヘッドの割合
Fig. 17　The overhead ratio on work-load NO-01.

たことによると考えられる．したがって，アルゴリズムの違いによる結果の差異も見受けられない．

ワークロード BI1 におけるキャッシュミス率を**図 15** に示す．BI1 では，タスクセットの重みが増加するにつれてキャッシュミス率も増加している．これは，重みの近いタスクが混在することにより，タスクの実行が分散されたためであると考えられる．タスクセットの重みが 8 の際に，LRU ではキャッシュミス率が約 0.98 であり，コンテキストキャッシュをほとんど利用できていない．LFU は，LRU よりもキャッシュミス率を低く抑えることができている．しかしながら，LFU でもタスクセットの重みが 8 でキャッシュミス率が約 0.85 である．FW と FW-Lazy ではキャッシュミス率が約 0.51 と 0.50 であり，LRU や LFU と比較してキャッシュミス率を大きく減少させることに成功している．FW と FW-Lazy のキャッシュミス率の差は約 0.01 であるが，両者の実装はほとんど変わらないことから FW-Lazy が最も効率的なアルゴリズムである．

ワークロード NO-1 と NO-01 におけるオーバヘッドの割合をそれぞれ**図 16** と**図 17** に示す．図 14 で示されたとおり，NO では BI1 と比較してキャッシュミス率が小さい．よって，コンテキストキャッシュを有効に利用できており，オーバヘッドの割合もすべてのタスクがコンテキストキャッシュに収まると仮定した理想的な Hardware に近いものとなっている．

ワークロード BI1-1 と BI1-01 におけるオーバヘッドの割合をそれぞれ**図 18** と**図 19** に示す．図 15 で示されたとおり，BI1 では NO と比較してキャッシュ

ミス率が大きい．よって，タスクセットの重みが増加するにつれてオーバヘッドの割合もすべてのコンテキストスイッチをソフトウェアで行った Software に近づいている．タスクセットの重み 8 において，FW と FW-Lazy が Software の約 0.5 倍であり，オーバヘッドを最も減少させることが可能となっている．

以上の結果より，FW は従来の LRU や LFU よりもオーバヘッドが増加してしまうことはなく，特定のタスクセットにおいてオーバヘッドをほぼ半減させることが可能となる．特に重みの近いタスクが複数あるタスクセットでは，顕著に結果が表れている．FW と FW-Lazy は実装上の差異はほとんどないことから，FW-Lazy が最も効率的なアルゴリズムである．

6. 関連研究

Anderson ら[10] は，Pfair を拡張した Early-Release fair（ERfair）を提案した．ERfair では，システムの負荷が低い場合にコンテキストスイッチ自体の数を削減できる可能性がある．しかしながら，システムの負荷が高くなると Pfair との差が少なくなる．したがって，根本的な解決には，コンテキストスイッチあたりのオーバヘッドを削減しなくてはならない．

Moir ら[13]，および，Liu ら[14] は，タスクを特定のコンテキスト上で実行しなければならない場合の，Pfair スケジュールにおける手法を示している．本来は分散環境を想定した手法であるが，SMT や CMP で利用することにより，コンテキストに依存した資源の競合を抑えることができる可能性がある．

Echague ら[15] は，シングルプロセッサにおいてコンテキストスイッチのコストが非常に大きいタスクを想定した研究を行っている．EDF と DM（Deadline Monotonic）について，オフラインでのコンテキストスイッチ数を算出するアルゴリズムと，コンテキストスイッチが起こるかテストする手法を示した．

Lee ら[16] は，バッファキャッシュを対象として LRU と LFU を組み合わせた LRFU アルゴリズムを提案した．Alghazo ら[17] は，プロセッサキャッシュを対象とした SF-LRU アルゴリズムを提案し，ハードウェアでの実現手法を示した．Jiang ら[18] は，局所性が低いワークロードに有効なアルゴリズムを示した．本論文中でも述べたとおり，コンテキストキャッシュの概念は，従来のキャッシュシステムと異なる．我々の知る限り，コンテキストのキャッシュを対象とした置換アルゴリズムに関する研究は行われていない．

7. 結　論

本論文では，複数の実行単位が存在する環境において，唯一知られている最適なスケジューリング手法である Pfair スケジューリングのオーバヘッドを削減するため，コンテキストキャッシュの置換手法を提案した．提案手法は，タスクのパラメータのみを利用し，低いコストで実現可能である．提案手法と，ページやバッファキャッシュを対象とした置換手法と比較し，コンテキストスイッチが非常に頻繁に起こりタスクの実行が分散してしまう場合，提案手法が有効であることを示した．これにより，複数のコンテキストが存在する環境において，プロセッサ時間をより有効に利用することが可能となった．

本論文では，静的なアルゴリズムを提案し，従来のキャッシュシステムで用いられてきた置換アルゴリズムと比較した．今後の課題として，これらのアルゴリズムを柔軟に組み合わせることにより，さらにキャッシュミス率を削減していきたい．また，提案アルゴリズムは，スケジューリングアルゴリズムについていっさい考慮していなかった．しかしながら，PD^2 は，連続して実行するサブタスクを多くするアルゴリズムであり，重みが 1/2 よりも大きなタスクは，1/2 よりも小さなタスクよりも連続して実行される可能性が高いと考えられる．よって，PD^2 の特性を考慮することにより，さらにキャッシュミス率を削減できると考えられる．今回は，コンテキストスイッチ自体のオーバヘッドに焦点を当てたが，FW によってその他のオーバヘッドも削減していく手法についても考えていく．

謝辞　本研究は，科学技術振興機構 CREST の支援による．

参考文献

1) Tullsen, D.M., Eggers, S.J. and Levy, H.M.: Simultaneous Multithreading: Maximizing On-Chip Parallelism, *Proc. 22nd Annual International Symposium on Computer Architecture*, pp.392–403 (1995).
2) Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K. and Chang, K.: The Case for a Single-Chip Multiprocessor, *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.2–11 (1996).
3) Liu, C.L. and Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *J. ACM*, pp.46–61 (1973).
4) Lopez, J.M., Garcia, M., Diaz, J.L. and

Garcia, D.F.: Worst-Case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems, *Proc. 12th Euromicro Conference on Real-Time Systems*, pp.25–33 (2000).
5) Dhall, S.K. and Liu, C.L.: On a real-time scheduling problem, *Operations Research*, pp.127–140 (1978).
6) Baker, T.P.: An Analysis of EDF Schedulability on a Multiprocessor, *IEEE Trans. Parallel and Distributed Systems*, Vol.16, No.8, pp.760–768 (2005).
7) Baruah, S.K., Cohen, N.K., Plaxton, C.G. and Varvel, D.A.: Proportionate Progress: A Notion of Fairness in Resource Allocation, *Algorithmica*, Vol.15, No.6, pp.600–625 (1996).
8) Srinivasan, A., Holman, P., Anderson, J.H. and Baruah, S.: The Case for Fair Multiprocessor Scheduling, *Proc. International Parallel and Distributed Processing Symposium*, p.10 (2003).
9) Baruah, S.K., Gehrke, J.E. and Plaxton, C.G.: Fast Scheduling of Periodic Tasks on Multiple Resources, *Proc. 9th International Parallel Processing Symposium*, pp.25–28 (1995).
10) Anderson, J.H. and Srinivasan, A.: Early-Release Fair Scheduling, *Proc. 12th Euromicro Conference on Real-Time Systems*, pp.35–43 (2000).
11) Anderson, J.H., Black, A. and Srinivasan, A.: Quick-release Fair Scheduling, *Proc. 24th IEEE Real-Time Systems Symposium*, pp.130–141 (2003).
12) Yamasaki, N.: Responsive Multithreaded Processor for Distributed Real-Time Systems, *Journal of Robotics and Mechatronics*, Vol.17, No.2, pp.130–141 (2005).
13) Moir, M. and Ramamurthy, S.: Pfair Scheduling of Fixed and Migrating Periodic Tasks on Multiple Resources, *Proc. 20th IEEE Real-Time Systems Symposium*, pp.294–303 (1999).
14) Liu, D. and Lee, Y.-H.: Pfair Scheduling of Periodic Tasks with Allocation Constraints on Multiple Processors, *Proc. 18th International Parallel and Distributed Processing Symposium*, p.199 (2004).
15) Echague, J., Ripoll, I. and Crespo, A.: Hard Real-Time Preemptively Scheduling with High Context Switch Cost, *Proc. 7th Euromicro Workshop on Real-Time Systems*, pp.184–190 (1995).
16) Lee, D., Choi, J., Kim, J., Noh, S.H., Min, S.L., Cho, Y. and Kim, C.S.: LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies, *IEEE Trans. Comput.*, Vol.50, No.122, pp.1352–1361 (2001).
17) Alghazo, J., Akaaboune, A. and Botros, N.: SF-LRU Cache Replacement Algorithms, *Proc. Records of the 2004 International Workshop on Memory Technology, Design and Testing*, pp.19–24 (2004).
18) Jiang, S. and Zhang, X.: Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance, *IEEE Trans. Comput.*, Vol.54, No.8, pp.939–952 (2005).

(平成 18 年 7 月 21 日受付)
(平成 18 年 11 月 12 日採録)

船岡 健司（学生会員）
1982 年生．2006 年慶應義塾大学理工学部情報工学科卒業．現在同大学大学院理工学研究科開放環境科学専攻修士課程に在籍．リアルタイムシステム，オペレーティングシステム等の研究に従事．

加藤 真平（学生会員）
1982 年生．2004 年慶應義塾大学理工学部情報工学科卒業．2006 年同大学大学院理工学研究科開放環境科学専攻修士課程修了．現在同博士課程に在籍．リアルタイムシステム，オペレーティングシステム等の研究に従事．

山崎 信行（正会員）
1966 年生．1991 年慶應義塾大学理工学部物理学科卒業．1996 年同大学大学院理工学研究科計算機科学専攻博士課程修了．工学博士．同年電子技術総合研究所入所．1998 年 10 月慶應義塾大学理工学部情報工学科助手．同専任講師を経て 2004 年 4 月より同助教授．現在産業技術総合研究所特別研究員を兼務．並列分散処理，リアルタイムシステム，システム LSI，ロボティクス等の研究に従事．