社団法人　情報処理学会　研究報告
IPSJ SIG Technical Reports

2005－ＳＬＤＭ－119（17）
2005／3／18

SMT機構における実行時間変動を考慮した実時間スケジューリング手法

加藤　真平†　　橘内　和也†　　小林　秀典†　　山崎　信行†

† 慶應義塾大学大学院理工学研究科開放環境科学専攻
〒 223-8522 横浜市港北区日吉 3-14-1
E-mail: †{shinpei,kitsunai,kobahide,yamasaki}@ny.ics.keio.ac.jp

あらまし　Simultaneous Multithreading (SMT) はスーパースカラに細粒度マルチスレッディングを統合したプロセッサアーキテクチャであり，実時間システムにおいてもその有用性が期待されているが，これまで SMT プロセッサを利用した実時間処理の研究はほとんど行われていない．SMT プロセッサの特性としてタスクの実行時間変動が挙げられる．実時間処理を考えた場合，タスクの実行時間が変動してしまうとデッドラインミスを起こす可能性があるので，大きな問題となる．本研究では，SMT プロセッサにおける既存の実時間スケジューリングアルゴリズムの挙動を吟味し，問題点を明確にする．そして，実行時間の変動を抑制し SMT プロセッサにおいても実時間処理が可能なスケジューリングアルゴリズムの設計および実装を行い，その性能を評価する．評価の結果，既存の EDF アルゴリズムと比べ本論文で提案したアルゴリズムは実行時間変動を小さく抑えることができた．

キーワード　実時間スケジューリング，SMT プロセッサ

Real-Time Scheduling Algorithm Bounding Execution Time Variation on a SMT Architecture

Shinpei KATO†, Kazuya KITSUNAI†, Hidenori KOBAYASHI†, and Nobuyuki YAMASAKI†

† Keio University
3-14-1 Hiyoshi, Kouhoku-ku, Yokohama, Kanagawa 223-8522 Japan
E-mail: †{shinpei,kitsunai,kobahide,yamasaki}@ny.ics.keio.ac.jp

Abstract　Simultaneous Multithreading is a processor architecture that combines the superscalar with fine-grained multithreading. Although it is expected to come to support real-time systems, very few works have been done in the area of real-time processing in SMT processors in the past. One of SMT-specific characteristic is an execution time jitter. From the viewpoint of real-time processing, it would be a critical problem since there is a possibility of deadline miss if the execution time fluctuates. In this paper, we examine a behavior of the existing real-time scheduling algorithm on an SMT processor and clarify the issues. We then design and implement a new scheduling algorithm which enables real-time processing on an SMT processor and evaluate it. The result proved that our algorithm held execution time jitter low compared to EDF algorithm.

Key words　Real-Time Scheduling, SMT Processor

1. はじめに

近年の実時間システムは，実時間処理の他に高性能処理も要求する．Simultaneous Multithreading (SMT) [1], [2] はスーパースカラに細粒度マルチスレッディングを統合したプロセッサアーキテクチャで，1 サイクルごとに複数のスレッドから命令フェッチ/発行を行い，同時に命令を実行する．8way-SMT アーキテクチャは従来のスーパースカラやマルチスレッディングに比べ 2〜3 倍の IPC を実現することが広く知られており [3]，実時間システムにおいてもその有用性が期待されている．

従来のプロセッサアーキテクチャとは違って，SMT では複数のスレッドが同時に実行可能となり，多くのハードウェア資源を共有する．各スレッドは毎サイクル必要となるハードウェア資源を取り合う．これは，あるスレッドの実行時間と IPC がそのスレッドと同時に実行しているスレッドによって影響を受けることを意味する．システム全体のタスク数が，SMT プロセッサにより提供されるハードウェアコンテキストの数より多い場合，タスクはスケジューラによってマルチプログラムされ

る．もし，同時に実行するタスクの組み合わせ (同時実行セット: *co-scheduled task set*) が毎回異なれば，同時に実行するタスクから受ける影響も異なり，結果として実行時間は毎リリースごとに変わってしまう．我々はこの現象を実行時間変動 (*ETV: Execution Time Variation*) と定義する．実行時間変動は実時間スケジューリングにいくつかの問題をもたらす．単一スレッドで計測された実行時間を基に実時間スケジューリングを行うと，実際の実行時間は単一スレッド時よりも長くなる可能性が高いので多くのデッドラインミスが起こってしまう．シミュレーションを通して最悪実行時間 (*WCET: Worst-Case Execution Time*) が計算できたとしても，実行時間変動問題によって平均実行時間 (*AvCET: Average-Case Execution Time*) との差が大きくなるとシステムの利用率の低下を招いてしまう．システム利用率が大きく低下してしまうと，SMT プロセッサを実時間システムに適用する利点が小さくなってしまう．よって，SMT プロセッサにおいて効率的に実時間タスクを扱えるスケジューリング手法の提案が必要である．

2. SMT と実時間スケジューリング

2.1 SMT の特性

すでに述べたとおり，SMT プロセッサにおいて実時間タスクをスケジューリングする場合，実行時間の変動が問題となる．この挙動を理解するために，我々は事前にいくつかの簡単な実験を行った．

図 1 は 2 つのスレッドをもつ SMT プロセッサを利用して，整数演算中心の PD 制御タスクを 4 つの異った状況で実行した場合の実行時間の変動を示している．

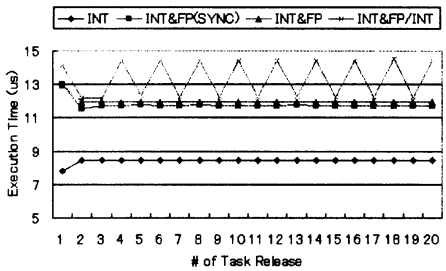


図 1 (INT) 単一スレッドで実行した場合の ETV. (INT&FP(SYNC)) リリース時間を同期させて浮動小数点演算中心の PD 制御タスクと同時に実行した場合の ETV. (INT&FP) リリース時間を同期させずに浮動小数点演算中心の PD 制御タスクと同時に実行した場合の ETV. (INT&FP/INT) 浮動小数点演算中心の PD 制御タスクと整数演算中心の PD 制御タスクの両方と同時に実行した場合の ETV.

単一スレッドで実行した場合，利用可能なハードウェア資源が全て占有できるので実行時間は短く変動も小さい (INT)．次に，もう一方のスレッドに計測対象タスクとは異った性質のタスクが 1 つ存在し，両タスクのリリース時間を同期/非同期の 2 つの場合でそれぞれ実行してみると，計測対象のタスクの実行時間は多少長くなるが実行時間の変動はほとんど見られない (INT&FP(SYNC), INT&FP)．最後に，計測対象のタスクと同じ性質をもつタスクをもう一方のコンテキストに加えて実行してみると，今度は計測対象タスクの実行時間が変動した (INT&FP/INT)．この結果から，同時に実行するタスクが同じであれば，たとえタスクのリリース時間がずれても実行時間の長さにはさほど影響がないことがわかる．一方で，毎回のリリースで同時に実行するタスクが異ると，実行時間の変動も激しくなることもわかる．

2.2 EDF の特性

新たな SMT 実時間スケジューリングアルゴリズムを提案する前に，SMT プロセッサにおいて既存の実時間スケジューリングアルゴリズムではどのような挙動が起こるのかを考察する．

Earliest Deadline First (EDF) は単一プロセッサでは最適なアルゴリズムとして知られている [4]．しかし，マルチプロセッサのような複数のコンテキストを有するシステムでは最適ではないことが証明されている [5]．それにもかかわらず，これまで SMT 実時間スケジューリングについてほとんど研究が行われていないことを考慮すると，EDF は適当な例の 1 つであると考えられる．本論文でも，その簡潔さと実用性から既存のアルゴリズムの代表として EDF を採用する．マルチプロセッサシステムで EDF を利用する際は，優先度が同じタスクが存在した場合にどのタスクを選択するかが重要になってくるが [6]，ここでは簡単化のため，ラウンドロビン方式を採用する．

図 2 は，表 1 にあるタスクセットが与えられたとき，2 つのスレッドをもつ SMT プロセッサで EDF を利用するとどのようにスケジュールされるかを示している．

Task	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
C/T	1/5	2/5	2/5	2/10	3/10	3/10

表 1 タスクセット 1

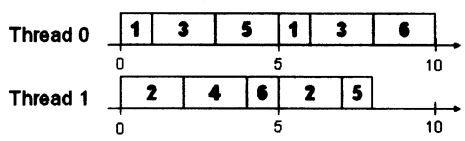


図 2 2way-SMT プロセッサでの EDF スケジューリングの様子

τ_3 に注目すると，1 回目のリリースでは，τ_3 は τ_2, τ_4 と同時に実行されている．しかし，2 回目のリリースでは，τ_2, τ_5 と同時に実行されている．2.1 節で論じたように，同時実行セットが変わると実行時間が変動する可能性がある．もし，実際の実行時間が予測した実行時間を超えてしまうと，図 2 の Thread 0 のように利用率が 1 に近いなコンテキストではデッドラインミスが起こってしまうかもしれない．仮に，シミュレーションなどで最悪実行時間が計算できたとしても，実行時間の変動が激しいと，つまり最良実行時間と最悪実行時間の差が大きいと，システム全体の利用率が低下してしまう．言い替えると，実行時間の変動が小さければ，システムの予測性も高まり利用率の低下も防げることができると考えられる．

3. Weight Combinated Scheduling

本論文では，実行タスクの組み合わせが変わらなければタスクの実行時間はさほど変動しないという特性に着目し，Weight Combinated Scheduling (WCS) と呼ばれる新しい実時間スケジューリング手法を提案する．WCS では，タスクの重み (C/T) を用いて，すべての実行タスクの組み合わせを固定する．タスクの重みが等しければ，ある時間内でのタスクの実行時間は等しくなる．よって，常に同じタスクを組み合わせることができる．

マルチプロセッサシステムにおいて，実時間スケジューリングはグローバルスケジューリング方式とパーティショニング方式に分類される [7], [8]．グローバルスケジューリング方式とは，システムにはスケジューラとタスクキューがそれぞれ 1 つだけ存在し，各タスクはどのコンテキストでも実行される可能性がある．一方，パーティショニング方式では，スケジューラとタスクキューはコンテキストごとに存在し，各タスクは静的にどれかのコンテキストに割り当てられる．

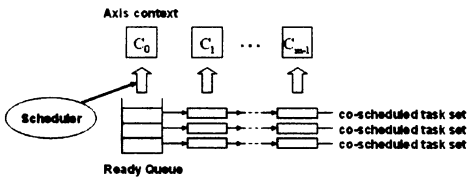


図 3　WCS のアーキテクチャ

図 3 は WCS のアーキテクチャを示している．図を見るとわかるように，WCS は各タスクが実行されるコンテキストが固定されるという点でパーティショニング方式といえる．しかし，コンテキストごとにスケジューラやタスクキューが用意されているわけではない．スケジューラとタスクキューは軸スレッド (Axis Thread) のみが保持する．スケジューラは軸スレッドで実行されるタスクについては，タスクキューからディスパッチし，その他のスレッドで実行されるタスクについては，軸スレッドで実行されるタスクの同時実行セット (co-scheduled task set) によって自動的に決まる．また，WCS はそのアーキテクチャとスケジューラが独立しているので，単一プロセッサにおける様々な実時間スケジューリングアルゴリズムと組み合わせて利用することができる．

3.1 システムモデル

本論文で想定するタスクはすべて周期タスクである。システムが開始してから未知のタスクが到着することはない。本論文で利用される記号を以下に示す。

- τ_i: システムに投入された i 番目のタスク
- D_i: τ_i の相対デッドライン
- T_i: τ_i の周期
- C_i: τ_i の計算時間
- N: システムに投入された全タスク数
- M: システムに存在する論理 CPU 数 (スレッド数)

3.2 WC-EDF

最初のステップとして，EDF を拡張した *WC-EDF* を設計する．WC-EDF は，図 3 の scheduler に EDF を採用する手法である．同時実行セットの決定には First-Fit-Decreasing-Utilization(FFDU) アルゴリズム [9] を応用する．

Co-scheduled Task Set Decision with FFDU

```
/* task[] is sorted by the non-increasing
   order of their weight */
task[] = SortByWeight();
top = 0;
/* Construct the task queue */
while(top < N) {
  /* Only the task on the axis thread is
     stored in the task queue */
  AddToTaskQueue(task[top]);
  t = task[top++];
  /* Make a co-scheduled task set */
  for (1...M-1) {
    /* set pointer to the next task */
    t->CoNext = task[top++];
    t = t->CoNext;
  }
}
```

まず，タスクを重みの大きい順にソートする．次に，先頭のタスクをタスクキューに格納する．繰り返すがタスクキューに格納されたタスクは軸スレッドで実行することになる．そして，残りのスレッド分のタスクをポインタ $(CoNext)$ で順につなげていく．すべてのスレッド分のタスクが決定されたらまた軸スレッド用のタスクから決定していく．先に示した表 1 をスレッド数 2 で例にとると，$\{\tau_2, \tau_3, \tau_5, \tau_6, \tau_1, \tau_4\}$ という順にソートする．これを順にスレッドに割り振るので，同時実行セットは $(TH_0, TH_1) = \{(\tau_2, \tau_3), (\tau_5, \tau_6), (\tau_1, \tau_4)\}$ になる．

同時実行セットが決定したら，軸スレッド (TH_0) で EDF に従ってタスクを選択し，そのタスクの同時実行セットをその他のスレッドで実行する．

WC-EDF を使って表 1 にあるタスクセットをスケジュールすると図 4 のようになる．

図 4 を見ると，各タスクの同時に実行する相手のタスクが変わらないことがわかる．これにより，各タスクの実行時間変動を抑制することができる．

3.3 FFDU Tie-Breaking

3.2 節で述べた設計ではいくつかの問題が生じる．まず，FFDU で同時実行セットを決定する際，重みが同じで実行時間や周期が違うタスクがあった場合，どのような順でソートするかが問題になる．FFDU はもともと箱詰め問題を解決するための手法なので，利用率 (重み) の大小しか考慮しない．し

WC-EDF Algorithm

```
/* Decide co-scheduled task set again if
   necessary */
if (There are new tasks) {
  ffdu();
}
/* Select the axis task from the task queue */
task[0] = SelectByEDF(TaskQueue);
/* Select the other thread task */
for (i=1,...M-1) {
  task[i] = task[i-1]->CoNext;
}
/* Execute the runnable tasks */
for (i=0,...,M-1) {
  RunIfRunnable(task[i]);
}
```

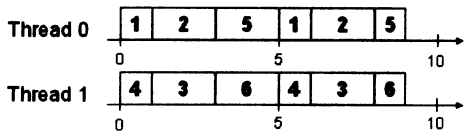
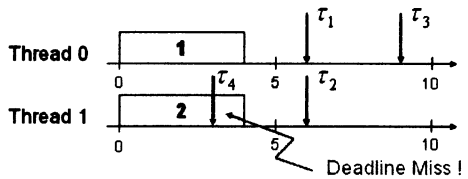


図 4 WC-EDF スケジューリングの様子

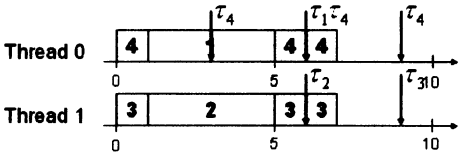
かし, WC-EDF では利用率が等しかった場合の優先付け (*Tie-Breaking*) の手法も重要になってくる. 表 2 に示すタスクセットを使って例を示す.

Task	τ_1	τ_2	τ_3	τ_4
C/T	4/6	4/6	3/9	1/3

表 2 タスクセット 2



(a) FFDU で生成した同時実行セット



(b) FFDUP で生成した同時実行セット

図 5 パーティショニングの Tie-Breaking による影響

Tie-Breaking の方法をラウンドロビンで FFDU すると, 同時実行セットは $(TH_0, TH_1) = \{(\tau_1, \tau_2), (\tau_3, \tau_4)\}$ となる. この同時実行セットで WC-EDF によりスケジューリングを行うと, 図 5(a) に示すように τ_4 がデッドラインミスを起こしてしまう. そこで我々はこの問題に対処すべく, 周期の短いタスクが優先されるように FFDU を行う *FFDUP(First-Fit-Decreasing-Utilization-Priority)* を提案する. FFDUP を用いて同時実行セットを決定し, WC-EDF によりスケジューリングを行うとデッドラインミスを起こさずスケジューリング可能となる (図 5(b)).

3.4 重み誤差

WC-EDF には, 同時実行セットの各タスクの重みが等しくない場合にどのようにスケジューリングを行うかという, もう 1 つの大きな問題が残っている. 表 1 や表 2 に示されるタスクセットでは, すべての同時実行セットの各タスクの重みは等しい. しかし, 一般的に同時実行セット内のタスクの重みが等しくない場合も存在する. 重みが違う 2 つのタスク $\{\tau_1, \tau_2\} = \{1/3, 1/4\}$ をセットにしてスケジューリングした場合, τ_1 の 4 回目の周期で τ_2 が実行できない (図 6).

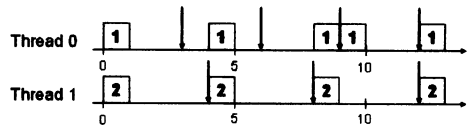


図 6 重み誤差があるタスクをスケジューリングする様子

この問題に対する最も簡単な対応策は, 同時実行セットのタスクが実行できないときは空きスロットにしておくことである. この手法を採用した場合, スケジューラビリティテストは同時実行セットの数 $S(S = \lceil \frac{N}{M} \rceil)$ を用いて式 (1) で表される.

$$\sum_{i=0}^{S-1} max\{\frac{C_k}{T_k} \mid k \in CoScheduleSet(i)\} \leq 1 \qquad (1)$$

この手法は, ロボットのような実行時間と周期の組み合わせがさほど多くないシステムでは効果的である. しかし, 実行時間と周期の組み合わせが多数存在するシステムではシステム全体の利用率が著しく低下する恐れがある. そこで, 同時実行セットを 1 つのタスクでなく複数のタスク (サブセット) で構成するよう FFDUP を改良する (*FFDUP-B*). 同時実行の単位をタスクセットにすることでシステム全体の利用率を増加することができる.

改良した FFDUP-B では, 軸スレッドのタスクの重みとその他のスレッドのタスクの重みとの差が, 最も重みの小さいタスク (*bottom* ポインタが指すタスク) より大きければ, その *bottom* ポインタが指すタスクをサブセットに格納する. タスクセット τ_1, τ_2, τ_3=1/2, 1/4, 1/4 が与えられた場合, $\tau_1 - \tau_2 \geq \tau_3$ となるので, τ_2 と τ_3 はサブセットとなる. これにより, 同時に実行するタスクが常に同じになるとは限らないが, 組み合わせは変わらない. WC-EDF では, サブセット内のタスクも EDF に従って選択する.

Co-scheduled Task Set Decision with FFDUP-B

```
/* task[] is sorted by the order of their weight
   (Ties are broken for shorter periods) */
task[] = SortByWeight();
top = 0; /* Top pointer to task[] */
bottom = N - 1; /* Bottom pointer to task[] */
/* Construct the task queue */
while(top != bottom) {
  /* Only the task on the axis thread is
   * stored in the task queue */
  AddToTaskQueue(task[top]);
  t = task[top++];
  /* Make a co-scheduled task set */
  for (j = 1; j < M; j++) {
    /* Set next task in co-scheduled task set */
    t->CoNext = task[top++];
    /* Add the task pointed by bottom to
     * the subset if necessary */
    while(task[bottom].weight <=
      AxisSet->weight-TotalWeight(t->SubNext) {
      /* set the next task in subset */
      t->SubNext = task[bottom--];
    }
    t = t->CoNext;
  }
}
```

表 3　命令発行ユニットの概要

アクティブスレッド数	8
キャッシュスレッド数	32
優先度の指定	256level
命令フェッチ数	8
同時命令発行数	4
同時命令完了数	4
整数レジスタ数	32bit × 32entry × 8set
整数リネームレジスタ数	32bit × 64entry
浮動小数点レジスタ数	64bit × 8entry × 8set
浮動小数点リネームレジスタ数	64bit × 64entry

4. 評　　価

4.1 評価環境

評価には，我々の研究室にて研究開発が行われている *Responsive Multithreaded Processor* (*RMTP*) [10] の次期バージョンとなる *μRMTP* を使用した．*μ*RMTP は現在チップ化作業中のため，評価は Cadence 社の NC-Verilog を用いたクロックレベルシミュレーションにより行なった．CPU クロックは 100MHz に設定した．使用した RMTP の命令発行機構の概要を表 3 に，命令演算機構の概要を表 4 にそれぞれ示す．

ここで，アクティブスレッドとキャッシュスレッドについてであるが，アクティブスレッドとキャッシュスレッドをハードウェアによって高速にコンテキストスイッチを行える機能を有しているため，このような構成になっている．

Improved WC-EDF Algorithm

```
/* Partition tasks if there are new tasks */
if (There are new tasks) {
  ffdup_b();
}
/* Select the axis task from the task queue */
task[0] = SelectByEDF(TaskQueue);
/* Get the head tasks of each subset in
 * co-schedule set where task[0] is */
head[0...M-1] = GetCoScheduleSet(task[0]);
/* Select each task from each subset */
for (i=1...M-1) {
  head[i] = head[i-1].CoNext;
  task[i] = SelectByEDF(head[i]);
}
/* run the tasks in sync */
for (i=0...M-1)
  RunIfRunnable(task[i]);
```

表 4　命令演算ユニットの概要

整数演算器	4 + 1 (Divider)
浮動小数点演算器	2 + 1 (Divider)
64bit 整数演算器	1
整数ベクトル演算器	1 (8IU × 2 line)
浮動小数点ベクトル演算器	1 (4FPU × 2 line)
分岐ユニット	2
メモリアクセスユニット	1
同期ユニット	1

4.2 周期タスクセットによる評価

使用するハードウェアスレッド数は 4 つとし，実行は開始時から 5*ms* まで行った．

周期タスクには，ロボットのモータ制御に用いられる PD 制御を行うタスクを使用した．各タスクの単一で実行した場合の実行時間は 10*μsec*, 周期は実用性を考慮して 200*μsec* と 1*msec* の 2 種類, 実行時間は単一スレッドで約 10*μsec* のタスクを計 30 個用意した．タスクセットには浮動小数点演算中心のタスクと整数演算中心のタスクが混在している．スケジューリングにおけるタスクの最小実行時間単位は 10*μsec* とした．

事前に，シミュレータで最悪実行時間を計測したところ 2 種類のタスクの最悪実行時間は表 5 のようになった．

Task	FP-intensive	INT-intensive
WCET	24.68*μsec*	23.87*μsec*

表 5　最悪実行時間

図 7 は，WC-EDF(with FFDUP-B) と EDF の実行時間変動

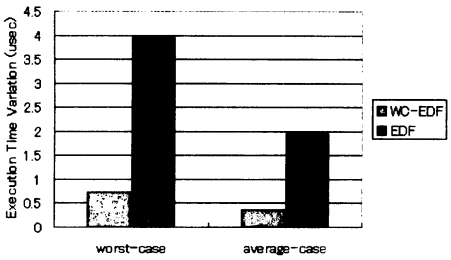


図 7　各アルゴリズムの実行時間変動の比較

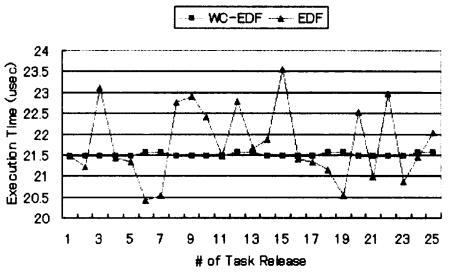


図 8　各アルゴリズムの実行時間変動の比較

の比較を示している. worst-case は, それぞれのアルゴリズムで実行した場合に最も大きな変動をみせたタスクの実行時間変動を表している. average-case は全タスク平均の実行時間変動を表している. EDF に比べ本論文の提案手法である WC-EDF の方が実行時間変動が小さいのがわかる. EDF では最大約 $4\mu sec$ の変動が見られたが, 数 μsec の粒度のモータ制御ではデッドラインミスの原因になりかねない.

図 8 は, より具体的に, WC-EDF と EDF で変動にもっとも大きな差の見られたタスクの実行時間の比較を示している. ほとんど実行時間に変動がない WC-EDF に対して EDF ではかなり大きな変動がある. また, 変動以外にも実際の実行時間に関しても, WC-EDF を使えばこのタスクの最悪実行時間を $22\mu sec$ と設定できるが, EDF では $24\mu sec$ と設定しないとデッドラインミスが起こる可能性がある. 平均実行時間に大した違いはないので, EDF ではシステムの利用率を低下させてしまう.

このような変動は, EDF では実行タスクの組み合わせが固定されてないため, 同時に実行されるタスクが整数演算中心のタスクの場合と浮動小数点演算中心のタスクになったりする場合があるためと考えられる. 一方, WC-EDF では常に同じ組み合わせが選択されるので実行時間に大きな変動は見られない. また, WC-EDF ではすべてのタスクが軸スレッドのタスクに同期してリリースされるのに対し, EDF では実行可能なタスクが存在したら即座に実行するため, 毎回のリリース時間がずれ実行時間に影響を及ぼしているとも考えられる.

5. 結論と今後の課題

本論文では, SMT プロセッサの実行時間の変動を考慮した Weight Combinated Scheduling (WCS) Algorithm を提案した. WCS の 1 つである WC-EDF では同時に実行するタスクが変らなければ実行時間はさほど変動しないことを利用して, 同時に実行するタスクの組み合わせを固定してスケジューリングを行った. シミュレーションによる評価では, 従来の EDF に比べ実行時間の変動を小さくすることに成功した. これにより予測性の高い実時間処理が可能となると考えられる.

今後の課題としては, 未知なるタスクが到着した場合に再度同時実行セットを決定する必要があるので, その際のオーバヘッドも考慮に入れなければならない. また, システム全体の利用率を向上させるために, 同時実行セット決定アルゴリズムに改良の余地があると考えられる. タスクセットに関しても, より実用的なシステムを目指すためには, メモリアクセスや I/O 処理を有するタスクを混在させて評価する必要がある.

謝辞　本論文の研究は, 文部科学省の科学技術振興調整費の支援による. また本研究の一部は, 科学技術振興機構 CREST の支援による.

文　献

[1] D. Tullsen, S. Eggers and H. Levy: "Simultaneous Multithreading: Maximizing On-Chip Parallelism", Proceedings of Annual International Symposium on Computer Architecture (1995).

[2] D. Tullsen, S. Eggers, H. Levy, J. Lo and R. Stamm: "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", Proceedings of Annual International Symposium on Computer Architecture (1996).

[3] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm and D. Tullsen: "Simultaneous Multithreading: A Platform for Next-Generation Processors", IEEE Micro, **17**, pp. 12–18 (1997).

[4] C. Liu and J. Layland: "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", Journal of ACM, **20**, pp. 46–61 (1973).

[5] S. Dhall and C. Liu: "On a real-time scheduling problem", Operations Research, **26**, pp. 127–140 (1978).

[6] J. Goossens, S. Funk and S. Baruah: "EDF scheduling on multiprocessor platforms: some (perhaps) counterintuitive observations", Proceedings of International Conference on Real-time Computing Systems and Applications (2002).

[7] B. Andersson and J. Jonsson: "Fixed-Priority Preemptive Multiprocessor Scheduling: To Partition or not to Partition.", Proceedings of International Conference on Real-Time Systems and Applications (2000).

[8] S. Lauzac, R. Melhem and D. Mosse: "Comparison of Global and Partitioning Schemes for Scheduling Rate Monotonic Tasks on Multiprocessor", Euromicro Workshop on Real Time Systems (1998).

[9] S. Davari and S. Dhall: "An On Line Algorithm for Real-Time Tasks Allocation", Proceedings of Real-Time Systems Symposium (1978).

[10] N. Yamasaki: "Design Concept of Responsive Multithreaded Processor for Distributed Real-Time Control", Journal of Robotics and Mechatronics, **16**, pp. 194–199 (2004).