# Responsive Multithreaded Processorの命令供給機構

薄井弘之[†,☆]　内山真郷[†,☆]
伊藤務[†]　山崎信行[†]

分散リアルタイム処理用 *Responsive Multithreaded Processor* のプロセッシングユニットである *Responsive Multithreaded Processing Unit* (*RMT PU*) の命令供給機構を設計・実装する．*RMT PU* は 8 way の Simultaneous Multithreading（SMT）方式にリアルタイムシステムの優先度を導入してすべての機能ユニットを設計することで，各スレッドを優先度順に実行し，リアルタイム実行を行う．*RMT PU* の命令供給機構において，単純に優先度による制御を行うと優先度の低いスレッドが実行される機会が少なくなり，システム全体の性能が低下してしまう．そこで，パイプラインの状況によって低い優先度を持つスレッドを実行することで，高い優先度のスレッドの性能を維持しながら，プロセッサ使用率を向上させる．特に単一スレッドの性能を重視するポリシと，単一スレッドの性能の低下を抑制しつつ，全体の性能に比重を置くポリシの 2 種類を実装することで様々なデッドラインを持つタスクのスケジューリングを支援する．パイプラインの状況によって対応するスレッドのフェッチを止める方法が最も効果的であり，命令バッファ内命令数や，分岐命令数によるもので最高優先度スレッドの性能低下を 1%未満に抑えつつ，全体性能を 10%から 20%向上できた．また，パイプライン中命令数によってフェッチを止める方法では，全体性能を約 100%向上しながら，最高優先度スレッドの性能低下を 40%程度に抑えることができた．

## The Instruction Supply Mechanism for Responsive Multithreaded Processor

Hiroyuki Usui,[†] Masato Uchiyama,[†,☆] Tsutomu Ito[†]
and Nobuyuki Yamasaki[†]

We design and implement the instruction supply mechanism for *Responsive Multithreaded Processing Unit* (*RMT PU*), the processing unit of *Responsive Multithreaded Processor* for distributed real-time systems. Priority used in real-time systems is introduced into all functional units of the 8way Simultaneous Multithreading (SMT) in *RMT PU*. Each thread is executed in priority order to realize real-time execution. In the instruction supply mechanism of *RMT PU*, if control by the priority is performed simply, the opportunity for a thread with a low priority to be executed will decrease and the whole performance will decrease. Then, by selecting the thread which has low priority according to pipelines's situation, the processor utilization is raised without reducing the performance of the thread with the highest priority. We implemented the policy which considers especially performance of the thread with highest priority as important, and the policy which considers the whole performance controlling the fall of the performance of the thread with highest priority. By changing and performing these policy, scheduling tasks which have various deadline are supported. The policy of stopping fetch according to a pipeline's situation is the most effective. By the policy based on the number of the instructions in instruction buffer or the number of branch instructions, whole performance can be improved 20% from 10% suppressing the performance fall of the highest priority thread to less than 1%. By the policy based on the number of the instructions in pipeline, the performance fall of the highest priority thread is able to be suppressed to 40%, doing improvement of a whole performance in 100%.

## 1. はじめに

リアルタイム性とは処理の真偽が，結果の真偽だけではなく，時間にも依存する性質であり，狭義には与えられた時間制約（デッドライン）を守ることを意味する．リアルタイム性は，時間制約によってハードリ

† 慶應義塾大学
Keio University
☆ 現在，株式会社東芝
Presently with Toshiba Corporation

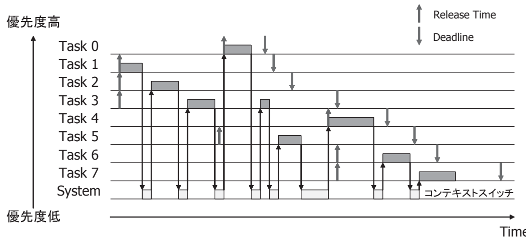


**図 1** シングルプロセッサにおけるリアルタイム実行
Fig. 1 Real-time execution on single processor.

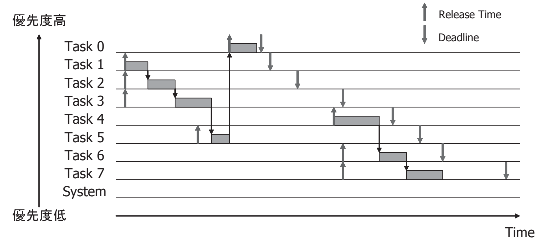


**図 2** マルチスレッドプロセッサによるリアルタイム実行
Fig. 2 Real-time execution on multithread processor.

アルタイム性とソフトリアルタイム性の 2 つに大別することができる．ハードリアルタイム性とは必ず時間制約を守らなければならず，時間制約を少しでも破ると価値が 0 になる性質のことである．ソフトリアルタイム性とは時間制約を多少破ることを許容する性質であり時間制約を破っても価値はただちに 0 にならないが，時間経過とともに価値が減少していく．

ハードリアルタイム性を持つタスクは時間粒度が小さく，デッドラインが 100 us から 10 ms 程度と短く，演算量は小さい場合が多いのに対し，ソフトリアルタイム性を持つタスクでは時間粒度が比較的大きく，デッドラインが 10 ms から 1 s 程度であり，演算量は比較的大きい場合もある．分散リアルタイムシステムにおいては，リアルタイムスケジューラがデッドラインや周期等の条件から優先度を各タスクに付与し，タスクはある一定時間間隔ごとに実行される．

**図 1** に Earliest Deadline First（EDF）でスケジューリングされたシングルプロセッサにおけるリアルタイム実行の例を示す．実行タスクの切替え時には，現在実行しているタスクのコンテキスト（レジスタセット，プログラムカウンタ，ステータスレジスタ等）をメモリに退避し，次に実行するタスクのコンテキストをプロセッサ内に復帰させるコンテキストスイッチが生じる．特に Preemptive なスケジューリングを行う場合，非リアルタイムシステムと比較して頻繁にコンテキストスイッチが起こるため，コンテキストスイッチのオーバヘッドが大きな問題となる．

この問題を解決するために，リアルタイム処理にマルチスレッド技術を応用する．マルチスレッドプロセッサは複数のスレッドコンテキストをプロセッサ内に保持するので，タスクとスケジューラ間のスイッチングのオーバヘッドを削減することができる．

さらに，マルチスレッドプロセッサにおいて実行スレッドを選択する際にリアルタイムシステムにおける優先度を用いる．**図 2** に 1 クロックあたり 1 命令処理するマルチスレッドプロセッサにおいて優先度を導入した場合の実行の様子を示す．優先度の高いタスクから実行され，他のタスクは待たされるため，ソフトウェアによるスケジューラとコンテキストスイッチを用いた従来方式と同様の実行が可能であり，コンテキストスイッチがない分効率が良くなる．可変優先度を用いるスケジューリングポリシの場合ではソフトウェアのスケジューラが必要となり，さらにプロセッサが保持できるコンテキスト数よりもタスク数が多い場合はスケジューラに加えて，コンテキストスイッチが必要となる．

我々は最初に，リアルタイム処理用のプロセッサとして独立した 4 本のパイプラインを持つブロックマルチスレッディングに優先度を導入したリアルタイム処理用プロセッサ[1] を設計・実装した．このプロセッサは 8 つのスレッドコンテキストをハードウェアで保持し，優先度の高い順に実行することで，複数スレッドの並列実行を行いながら，リアルタイム実行を可能にしている．8 スレッド以内ならばリアルタイム実行の際にコンテキストスイッチを必要としない．また，それ以上のスレッド数を扱う場合には，コンテキストキャッシュを用いた高速なコンテキストスイッチによりリアルタイム実行を支援する．しかし，各パイプラインが独立しているので，1 スレッドあたりの IPC が低いという問題点があった．

そこで，我々は Simultaneous Multithreading (SMT)[2,3] に優先度を導入してリアルタイム処理を行う *RMT PU*[4] を設計・実装した．*RMT PU* は並列分散リアルタイム処理用システム LSI である *Responsive Multithreaded Processor*（*RMT Processor*）[5] のプロセッシングユニットである．ここで，*RMT Processor* は並列分散リアルタイム処理に必要な機能のほとんどを 1 チップに集積したシステム LSI である．具体的には，リアルタイム処理機能（*RMT PU*），リアルタイム通信機能（*Responsive Link*），コンピュータ用周辺機能（DDR SDRAM I/Fs，PCI64，USB2.0，IEEE1394 等），制御用周辺機能（PWM 発生器，パ

ルスカウンタ等）を 1 チップ（TSMC 0.13 $\mu$m ダイサイズ 10.0 mm 角，約 10 M ゲート）に集積している．*RMT PU* は前述のブロックマルチスレッディングプロセッサの場合と同様に高優先度タスクから実行を行い，低優先度タスクの実行を待たせるため，リアルタイム実行が可能である．さらに，SMT アーキテクチャをとるために，前述の優先度付きブロックマルチスレッディングと比較して，プロセッサ使用率の向上や，シングルスレッドの性能向上を実現し，より多くのタスクをスケジューリングできるようになった．

しかし，この従来の *RMT PU* はすべての競合制御を単純に優先度に従って解決するため，優先度の低いスレッドの実行回数が極端に少なく，スレッドの並列実行によるレイテンシの隠蔽の効果も低いため，システム全体の性能は低下してしまう．そのため，スケジューリングできるタスクの時間粒度が細かくできても，スケジューリングできるタスクの総数は少なくなってしまう．

そこで，本研究では従来の *RMT PU* の優先度制御に加え，パイプラインの状況に応じて優先度の低いスレッドも選択して実行することで，優先度の高いスレッドの性能を維持しつつ，システム全体の性能を向上させることを目指す．

## 2. 関 連 研 究

文献 6) ではマルチスレッドプロセッサである Komodo microcontroller のデコード命令選択において，リアルタイムスケジューリングポリシを用いている．リアルタイム処理にマルチスレッドプロセッサを用いることで，コンテキストスイッチのオーバヘッドの削減を図るとともに，プロセッサの使用率の向上によるシステム全体の性能向上を図っている．Komodo microcontroller は 1 クロックあたり 1 命令処理する 4 ステージのパイプラインになっているため，各スレッドの性能は低い．今後，特にソフトリアルタイム処理においては，画像処理等高い処理能力を要求されることがあり，単一スレッドの処理能力の向上が求められる．複数スレッドをスーパスカラ上で実行する SMT アーキテクチャではシングルパイプラインのプロセッサと異なり，動的に最適化されるのでデータ依存等でもパイプラインがストールしにくく，単純に優先度のみを用いてフェッチを行うと優先度の低いスレッドのフェッチされる回数が減少し，全体の性能が低下してしまう．

文献 7) では SMT のフェッチスレッド選択に優先度を導入することで，foreground thread の性能をシングルスレッド実行時と比較して減少させずにシステム全体の性能をある程度向上させている．単純に優先度のみを用いて制御を行うと，プロセッサ全体での性能が低くなるので，扱うタスク数が多くなるとスケジューリングできなくなってしまう．

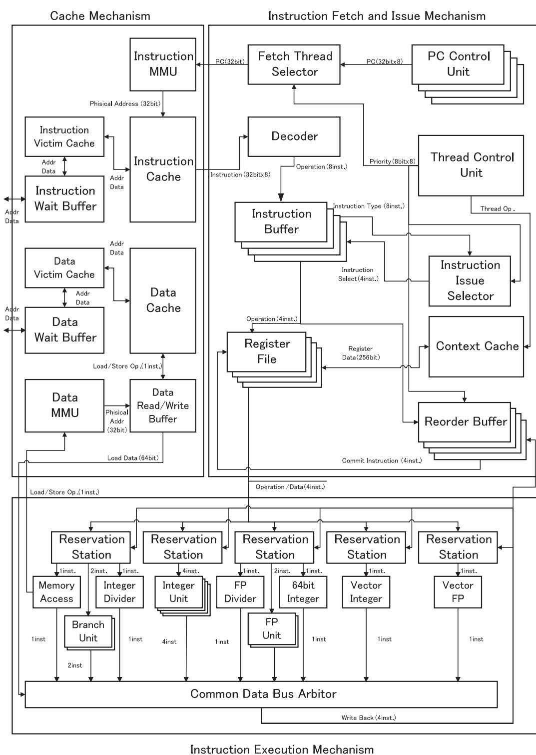


**図 3** *RMT PU* のブロック図
Fig. 3 Block diagram of *RMT PU*.

## 3. 設計および実装

### 3.1 Responsive Multithreaded Processing Unit

本機構の実装対象である *RMT PU* は複数スレッドのコンテキストを保持し，優先度に従って並列実行を行う．保持しているコンテキスト間ではソフトウェアによるコンテキストスイッチを行うことなく，ハードウェアによる優先度付きマルチスレッディング機構によって実行スレッドの切替えを行い，リアルタイム実行を可能にする．

*RMT PU* のブロック図を**図 3** に，プロセッサのパイプラインを**図 4** に示す．

*RMT PU* は図 3 に示すように命令供給機構，命令実行機構，キャッシュ機構の 3 つに大きく分けられる．命令供給機構はスレッドを管理し，優先度に従って実行ユニットに対して命令を送る．命令実行機構では命

| | |
|---|---|
| FS | Fetch Thread Select |
| IF1 | Instruction Fetch1(MMU) |
| IF2 | Instruction Fetch2(Select Way) |
| IF3 | Instruction Fetch3(Access) |
| DEC | Instruction Decode |
| IS | Issue Instruction |
| REG | Register Rename and Read |
| ES | Execute Instruction Select |
| EXE | Execute Instruction |
| WB | Write Back |
| CS | Commit Instruction Select |
| COM | Commit Instruction |
| MEM | Memory Access(Store) |

**図 4** *RMT PU* のパイプライン
Fig. 4 Pipeline of an *RMT PU*.

令発行ユニットから命令の演算を行う．

*RMT PU* では命令フェッチの際に，図 3 の Fetch Thread Selector において，優先度に従ってフェッチするスレッドを選択する（FS ステージ）．命令キャッシュのアクセスレイテンシは 3 クロックかかる（IF ステージ）．

フェッチした命令はデコードされ，図 3 の Instruction Buffer （命令バッファ）に格納される（DEC ステージ）．命令バッファはスレッドごとに用意されており，各スレッド 16 命令格納することができる．

命令バッファ内の命令は図 3 の Instruction Issue Selector で優先度に基づいて高い優先度の命令から発行される（IS ステージ）．以降のステージは通常のスーパスカラと同様である．

*RMT PU* の特徴として 32 個のスレッドコンテキストを退避，復帰可能なコンテキストキャッシュ（図 3 内 Context Cache）をオンチップで実装している．実チップの *RMT PU* のレジスタ数は GP32 個，FP8 個であり，コンテキストキャッシュとレジスタファイル間を幅の広いバス（GP256 bit，FP128 bit）でつなぐことで，並列実行可能なスレッドコンテキストとは別に 4 クロックでスレッドの切替えが可能となる．多数のタスクを Preemptive なリアルタイムスケジューリングで扱うと，コンテキストスイッチが頻繁に発生するので，コンテキストキャッシュは効果的に機能する．

**表 1** 命令供給機構の概要
Table 1 The outline of instruction supply mechanism.

| | |
|---|---|
| ハードウェアコンテキスト数 | 8 |
| 命令フェッチ数 | 8 |
| 命令デコード数 | 8 |
| 命令発行数 | 4 |
| 命令コミット数 | 4 |
| 整数レジスタ（32 bit）数 | $32 entry/thread$ |
| 浮動小数点レジスタ（64 bit）数 | $32 entry/thread$ |
| リオーダーバッファ数 | $128 entry$ |

### 3.2 命令供給機構

本研究では前述した *RMT PU* の命令供給機構に様々な制御ポリシを実装することで，様々な種類のタスクのスケジューリングを支援する．**表 1** に命令供給機構の概要を示す．

これらのパラメータは命令発行数を基準として予備評価を行い決定した．命令発行数が 2 命令では単一スレッド実行時の IPC が低くなり，命令発行数が増加するとレジスタファイルならびにリオーダバッファの読み出しポート数が増加するので，サイズの面で困難になる．そこで，命令発行数を 4 とし，それにあわせて命令コミット数も 4 とした．命令フェッチ，デコード数は 8 とする．命令発行数よりも多い命令数であるが，分岐命令により必ずしも最大限有効な命令が含まれるわけではないという点や，命令バッファに命令を貯めることで，優先度の低いスレッドが割り当てられるフェッチスロットをつくるという意図から 8 とした．リオーダバッファは従来の *RMT PU* におけるスレッドごとに 16 命令で合計 128 エントリの構成から変更し，スレッドの単独実行時の性能を向上させるために全スレッド共有で 128 エントリ用意した．リオーダバッファについては詳細を後述する．

同時実行可能スレッド数は，優先度を設定し，ハードウェアのみでリアルタイム実行が可能な最大スレッド数となる．*RMT PU* は組み込み用途向けのプロセッサであり，OS による動的スケジューリングなしでリアルタイム実行が可能であることは大きな利点となる．組み込み用途では 8 スレッド以内でシステム構築できる場合も多く，スレッド数を 8 と設定した．

命令供給機構の各ブロックモジュールとデータの流れを**図 5** に示す．

各スレッドは制御レジスタや PC，State を保持している．その制御レジスタ（図 5 中の Control Register）の 1 つとして優先度を保持するものがあり，その値を競合処理に用いる．静的に Rate Monotonic Scheduling を行う場合に 256 レベルの優先度があればスケジューリング可能という理論[8] に基づいて，本プロ

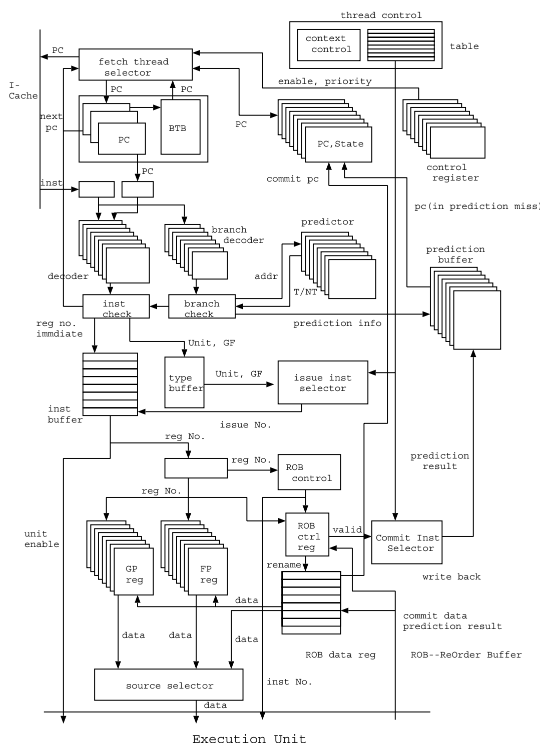


**図 5**　命令供給機構ブロック図
Fig. 5　Block diagram of an instruction supply mechanism.

セッサの優先度は 8 ビットで 256 レベルを用いる．

以下，命令供給機構内の各モジュールの機能について説明していく．

### 3.2.1　命令フェッチ

複数の実行可能なスレッドの中からフェッチするスレッドを fetch thread selector で選択してフェッチを行う．フェッチ要求を出してから実際にデータが命令キャッシュから返ってくるまでにキャッシュミスがない場合で 3 クロックかかるので，デコードステージによる分岐予測器に加えて Branch Target Buffer（BTB）を用いて投機的にフェッチを行う．

フェッチスレッドを選択する際，命令キャッシュのポートを複数にすることで複数のスレッドの命令でフェッチスロットを埋めることができるので，フェッチレートを向上させることができる．また，優先度に従って，優先度の高いスレッドが埋められなかったフェッチスロットを優先度の低いスレッドの命令で埋めることで，システム全体の性能を向上させることが可能である．しかし，キャッシュのポート数をたとえば 1 から 2 にすると，面積は約 2 倍になってしまう．同じサイズ制限の中でサイズを 1/2 にして 2 ポートにするか，ポートを 1 ポートにするかの選択では，優先度の高いスレッドを主に実行するという本プロセッサの目的から，1 スレッドが使用できるキャッシュの量が多くできる 1 ポートを採用する．

優先度の高いスレッドの実行を保証するために，フェッチスレッドの選択は優先度によって行う．優先度の低いスレッドが実行されないことで，システム全体の性能が低下することを防ぐために，最高優先度スレッドの性能をできるだけ維持しつつ，他のスレッドのフェッチを行うポリシを設計する．以下の 5 種類の場合について，そのスレッドのフェッチを止め，次の優先度のスレッドを選択する機構を設計する．

(1)　命令バッファ中の命令数
最高優先度スレッドの性能を落とさないためには，実行機構への命令供給を可能な限り絶やさない必要がある．そのためには，命令バッファ中につねに発行できるだけの命令が格納されている必要がある．命令発行まで 5 ステージあるのでスレッド選択の際に最高優先度スレッドの命令がバッファ内に 6 クロック分あれば，そのクロックに他のスレッドからフェッチを行ったとしても，次の最高優先度スレッドの命令フェッチがキャッシュミスしなければ，命令バッファ中の命令が不足することはない．1 クロックに 4 命令発行できるので，24 命令以上を閾値に設定することで，性能の低下を抑えることができる．

(2)　パイプライン中の条件分岐命令数
条件分岐命令を多く実行するほど分岐予測ミスの可能性が高くなり，実行資源を無駄にする可能性が高くなる．そのため，パイプライン中の条件分岐命令の数が閾値に達した場合に優先度の低いスレッドからフェッチを行う．

(3)　パイプライン中の未完の命令数
文献 3) における I-Count のことを指し，発行済みでコミットされていない命令数のことを指す．パイプライン中に同じスレッドの命令数が増加するとデータ依存関係のために全体性能が低くなる．そのため，パイプライン中の命令数が閾値に達した場合に優先度の低いスレッドからフェッチを行う．

(4)　フェッチパイプラインに占めているステージ数
Branch Target Buffer を用いた分岐予測ミスによるフェッチスロットの浪費を防ぐために，命令キャッシュのレイテンシである 3 ステージ中のそのスレッドが占めているステージ数を数え，閾値に達した場合は優先度の低いスレッドから

フェッチを行う．

(5) リザベーションステーション内の命令数
使う実行ユニットに偏りがある場合や，命令間に強い依存関係がある場合，パイプライン中の命令数による制御ではリザベーションステーションを一杯にしてしまう可能性がある．そこで，リザベーションステーション内の命令数を数え，閾値に達した場合に優先度の低いスレッドからフェッチを行う．

これらのポリシは図 5 の Fetch Thread Selector 内に設計した．それぞれのパラメータについてのカウンタを用意し，その値が閾値を超えた場合に，対応するスレッドのフェッチ要求線をマスクする．ポリシに従って選択されたスレッドの命令をキャッシュに要求する．

デコード時には bimodal 分岐予測器[9] を用いて分岐予測を行う．エントリ数はスレッドごとに 128 である．分岐予測器をスレッドごとに共通のものを使用すると，低い優先度スレッドの分岐の方向によって最高優先度スレッドの分岐予測ミス率が上がってしまうので，スレッド別に用意した．

分岐予測の結果に従い，有効な命令を図 5 における inst buffer（命令バッファ）に格納する．

#### 3.2.2 命令発行

命令バッファ内の命令の中から発行する命令を選択する．命令バッファを用いることで，命令フェッチと実行機構を分離し，一方がストールしてももう一方が動作することができる．Preemptive なスケジューリングをとる場合には，最高優先度スレッドが頻繁に切り替わるので，未完の命令が破棄されパイプライン中に空きスロットが多く生じてしまう．このとき，命令バッファに低い優先度スレッドの命令を格納しておくことで，実行資源を効果的に使用することができる．

命令バッファ内の命令から発行する命令を選択する際に，スレッドの優先度を用いて発行する命令を選択する．直接実行機構へ命令を供給するので，可能な限り優先度の高いスレッドの命令を発行する必要がある．

同じスレッドの命令を発行し続けるとデータ依存のために性能が低くなるので，フェッチスレッド選択と同様に，優先度の高いスレッドの性能を維持しつつ，全体の性能を向上させるために，優先度の高いスレッドが以下のパラメータについて閾値を超えた場合に命令発行を止め，低い優先度のスレッドから命令を発行する．

- パイプライン中の条件分岐命令数
- パイプライン中の命令数
- リザベーションステーション内の待ち命令数

パラメータは命令フェッチの場合に含まれるものと同じである．このポリシを図 5 の Issue Inst Selector 内に設計した．フェッチスレッド選択に用いたカウンタと同じものを利用し，閾値を超えた場合は対応するスレッドの命令発行の要求線をマスクする．

発行命令選択では命令バッファ内に命令のあるスレッドの中から優先度の高い 4 つのスレッドについて，発行する命令を選択することで，複雑化を抑えている．

#### 3.2.3 共有資源

最高優先度スレッドの性能維持のためには，命令フェッチスレッド選択や発行命令選択といったスロットに関する競合の解決に加え，共有資源の競合に関する制御が必要となる．本論文では，命令バッファ，リオーダバッファの制御機構を設計する．

これらの実行資源はスレッドごとに分割して用意することで，並列に実行されるスレッドが互いの性能に与える影響を抑えることができるが，サイズの問題からスレッド 1 つあたりの割当ては減少する．そのため，優先度の高いスレッドのバッファがすぐに一杯になってしまい，その分優先度の低いスレッドが実行されることになり，その他の共有資源を占有して優先度の高いスレッドに悪影響を与えてしまう．

そこで，本機構では命令バッファ，リオーダバッファをスレッド間で共有することで，単一のスレッドの性能の向上を図る．

これらのバッファは処理をスレッドごとに In-Order に行う必要があるので，スレッド間で同じバッファを使用すると，スレッドごとに最新の命令を探す必要があり，ハードウェアが複雑化してしまう．そこで，バッファをパーティション化（いくつかの単位に分割）する．1 つのパーティションを単一のスレッドのみが使用するように設計することで，各スレッドごとにパーティションの使用順番とパーティション内の現在位置だけ記憶しておけばよいので，ハードウェアが単純になる．

以上を実装した命令バッファを**図 6** に示す．リオーダバッファについても基本的な構造は同じであり，バッファに格納するデータが異なるのみである．

図 6 中の Inst Buffer Control（命令バッファ制御ユニット）でパーティションの管理を行う．命令バッファに書き込むアドレスはデコード時に決定される．Inst Buffer Control には各スレッドのパーティション内の先頭と末尾を示すポインタとパーティションの使用順を記録したキューが用意されている．図 6 において，スレッド 0 の命令がデコードステージにあるとすると，Inst Buffer Control は現在のパーティション

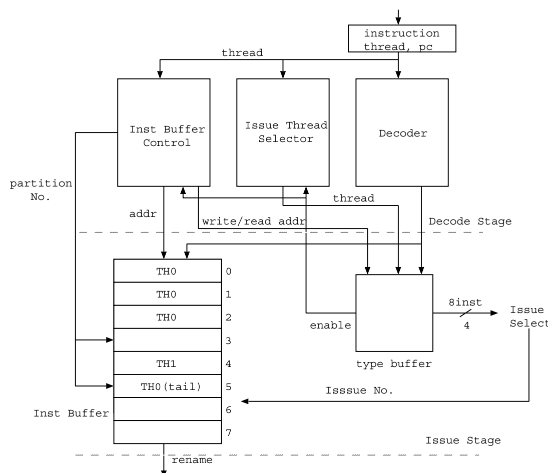


**図 6** 命令バッファのブロック図
Fig. 6 Block diagram of the Instruction Buffer.

内の位置を示すポインタから空きバッファ数を数え，足りない分は空のパーティションに書き込む．図 6 のように，末尾のパーティションが 5，末尾のパーティション内のポインタ（4 bit）が e のときに，8 命令書き込みがあると，書き込みアドレス（7 bit）は 5e, 5f, 30...34 というようになる．空のパーティションがない場合に現在のパーティション内の空きエントリ以上のフェッチがある場合は要求を止める．

命令発行時には同様に先頭のパーティション番号と，パーティション内の位置を示すポインタから読み出しアドレスを指定し，図 6 の type buffer から実行ユニット等の情報を読み出す．Issue Instruction Selector で選択された読み出しアドレスを用いてデータバッファから 4 つの命令を発行する．

バッファの分割数は同種類のタスクを複数取り扱うときのように，全体の性能を向上させたい場合を考えると，すべてのスレッドがストールしないために最低でも並列実行可能なスレッドコンテキスト数と同じ数があることが好ましい．また，パーティション内の命令数が少ないと，1 回の要求が 3 つ以上のパーティションを占有し，パーティションの割当てが複雑化してしまう．リオーダバッファは性能評価を行った結果 128 エントリで性能は頭打ちになるので，総数を 128 とし，分割数に関しては増やすと並列実行時の性能が高くなるので 16 とした．命令バッファのエントリ数は単純に優先度に基づいてフェッチを行った場合，64 エントリでは優先度の高いスレッドの命令でほとんど埋まってしまい，システム全体の性能上昇が少なくなるので 128 エントリとした．分割数に関してはシステム全体の性能に関してリオーダバッファよりも影響が小さいので，複雑化を抑えるために 8 とした．

また，共有バッファにおけるスレッド間の性能に対する影響を抑えるために，各スレッドが使用できるバッファのエントリ数をソフトウェアにより以下の方法でパーティション単位で設定する．

- 資源予約
  パーティションごとにスレッドの使用権を設定する．同じパーティションに複数のスレッドを割り当てることができる．
- 最大使用数設定
  スレッドごとに使用できる最大使用数を設定する．

資源予約方式ではスレッドごとに利用できる資源を個別に指定できるので，スレッド間の影響を除きやすい．しかしリアルタイムシステムではスレッドの停止，再開が繰り返されるので，資源を有効に利用するためには頻繁に設定し直す必要がある．

それに対し最大数設定方式ではスレッドのいくつかが停止している状況でも資源を有効に利用することができる．しかし設定した最大値の合計がバッファ数の合計を上回った場合に，優先度の低いスレッドがバッファを占有することで優先度の高いスレッドの実行を阻害してしまう．また，優先度の高いスレッドを除いた他のスレッド全部でいくらかのバッファを共有するといった細かい制御を行うことができない．

このように両者の手法ともに長短所があるので，本機構には両者を設計した．新しいパーティションが必要な場合に現在利用しているパーティションの数が最大数未満か，また空きパーティションの中にそのスレッドに割り当てられているものがあるかを調べる．

しかし，これらの制御のみでは最高優先度スレッドの性能を維持することは難しい．たとえば優先度の高いスレッドがスケジューリング可能な状態になり，最高優先度スレッドが切り替わる場合には，それまで実行されていた優先度の低いスレッドが実行資源を占有しており，優先度の高いスレッドの実行が阻害されてしまう．

そこで，優先度の高いスレッドの命令が格納されているパーティションの数が閾値以下で，空きパーティションがない場合に，現在その資源を利用しているスレッドの中から最も優先度の低いスレッドの命令を破棄する．命令バッファの場合ではそのスレッドの先頭の命令からフェッチし直し，リオーダバッファでは分岐予測ミスの場合と同様の機構を用いてパイプライン中の命令を破棄する．命令バッファにはその命令の PC 等の情報が格納されているので，追加するハードウェアはわずかで済む．

### 3.3 ソフトウェアによる設定

以上のような供給機構の制御機構を有効に利用するには，OS が頻繁に設定を変更する必要がある．たとえば，デッドラインまでの時間が非常に短いハードリアルタイムタスクがあるような場合では最高優先度のスレッドの性能を低下させるようなポリシを利用するべきではないし，デッドラインまでの時間が比較的長く，さらにデッドラインがお互いに近い位置にあるソフトリアルタイムタスクが複数あるような場合では優先度の低いスレッドの実行を必要以上に阻害し，全体の性能を低下すべきではない．

*RMT PU* は実行スレッド数が 8 スレッド以内で静的スケジューリングならばスケジューラによらないリアルタイム実行が可能であることが長所であるので，状況によって使用するポリシやバッファの割当てをハードウェアで変更する手法を設計することでハードウェアのみのリアルタイム実行時にプロセッサ資源を効率的に利用する．

各スレッドごとに，次のパラメータを設定し，最高優先度スレッドのパラメータに合わせて全体の制御パラメータを変更する．

- フェッチスレッド選択，発行命令選択ポリシ
- 最高優先度スレッドの場合の実行資源（命令，リオーダバッファ）の使用割合

優先度が 2 番目以下のスレッドは優先度の高いスレッドが使用する残りの実行資源を利用する．

スレッドの停止や優先度の変更等で実行資源の利用割合に変更があり，現在そのスレッドが獲得しているパーティションが使用できなくなった場合については解放を行わず，新しいパーティションに格納する場合に設定されたパラメータを適用する．前述したように最高優先度スレッドが格納できるパーティションがなくなり，最高優先度スレッドが使用できるパーティションを優先度の低いスレッドが使用している場合に低い優先度のスレッドのバッファを解放する機構が用意されているので，システム全体の性能をあまり低下させずに，スレッドの切替えを行うことができると考えられる．

## 4. 評 価

本研究は *RMT Processor* の実チップの構成の中で命令供給機構の箇所を変更したものである．そのため評価は実チップの設計・実装に用いた RTL によるシミュレーションによって評価を行った．

### 4.1 優先度特性

初めに，プロセッサ全体の優先度制御について評価を行う．スレッド選択ポリシや実行資源の使用量制御は使用しない．1 つのスレッドが 1 つのプログラムを実行し，その同じスレッド複数を同時刻に実行開始した場合の，各スレッドの単位時間あたりの命令実行数並びに実行にかかる時間を測定した．

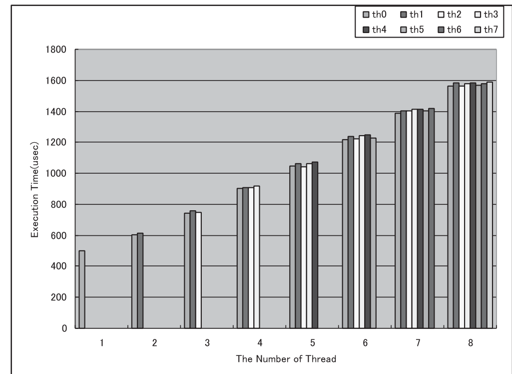


**図 7** 優先度を導入しない場合の実行時間
Fig. 7 Execution time without priority.

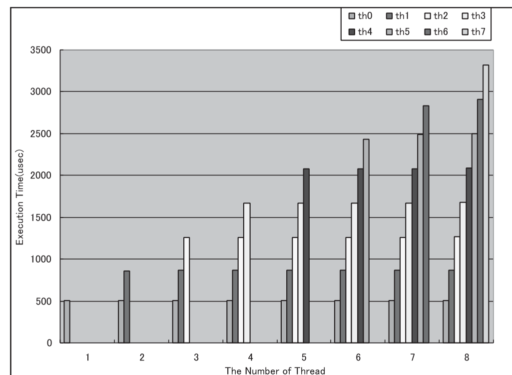


**図 8** 優先度を導入した場合の実行時間
Fig. 8 Execution time with priority.

評価プログラムとして逆離散コサイン変換（IDCT）を実行したときの，**図 7** に優先度を付加しない場合の実行結果，**図 8** に優先度を用いた場合の実行結果を示す．優先度を用いた場合では，スレッド 0 の優先度が最も高く，段階的に優先度が低くなり，スレッド 7 が最も低くなるように設定した．

同時実行したスレッドの実行時間に着目すると，図 7 に示すように，優先度を付加しない場合には，実行スレッド数にかかわらず，実行したすべてのスレッドがほぼ同時刻に実行が終了する．これは，すべてのスレッドが等しくスケジューリングされ実行されるからである．そのため，デッドラインの位置が大きく異なるようなスレッドを同時に実行すると時間制約の厳しいス

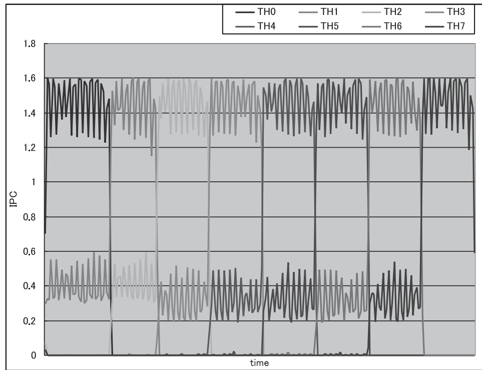


**図 9** 優先度を導入した場合の IPC
Fig. 9 IPC with priority.

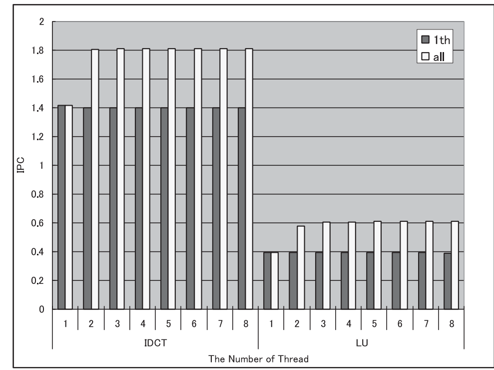


**図 10** 優先度を導入した場合の性能
Fig. 10 Performance with priority.

レッドがデッドラインミスを起こす可能性がある．

それに対し，優先度を付加した場合について図 8 の各スレッドごとの実行時間に着目すると，最も優先度の高いスレッド 0 の場合では 1 スレッド実行時には実行時間が 501.0 $\mu$sec であるのに対し，8 スレッド実行時の実行時間は 506.6 $\mu$sec で，その差は 1 スレッド実行時の 1.1%である．低い優先度を持つスレッドは実行可能状態であるが，フェッチされず待たされている状態になっている．これは優先度の低いタスクがスケジューリングされずに待たされている状態と等しく，優先度に応じたスケジューリングをハードウェアで行っていることになる．

優先度を付加し，8 スレッド実行した場合の各スレッドの IPC を**図 9** に示す．

図 9 で分かるように，優先度が最も高いスレッドが実行資源を十分に使用し，実行を行う．分岐予測ミス時等で最高優先度スレッドが実行されないときに次の優先度のスレッドが実行される．このとき，3 番目以降の優先度が設定されているスレッドはほとんど実行されず，レイテンシの隠蔽による性能向上の効果は低くなる．

そこで次に，スレッド並列実行によるレイテンシの隠蔽の評価を行う．評価環境は実行時間を測定したときと同様，1 スレッドが 1 プログラムを実行し，すべてのスレッドが同時刻に実行を開始する．IDCT，LU 分解（LU）を評価プログラムとして使用し，優先度を同様に段階的に設定し 8 スレッド合計の性能を評価する．

**図 10** に最高優先度スレッドの性能と，システム全体の性能を示す．図 10 中（1th）が最高優先度スレッド性能,（all）がシステム全体の性能である．

図 10 において，システム全体の性能に着目すると，IDCT，LU ともにスレッド数を 1 から 2 にした場合に最も性能が向上する．IDCT はスレッド数を 2 にすると，1 スレッド実行時の IPC1.41 と比較して IPC が 1.80 となり，0.39（27.8%）向上する．しかしスレッド数を 3 以上にしてもスレッド数が 2 の場合との性能差は 0.004 程度である．LU については，スレッド数を 2 にすると 1 スレッド実行時の IPC0.40 と比較して IPC が 0.58 となり，0.18（46.3%）向上する．スレッド数を 8 にすると単一スレッド実行時よりも IPC が 0.21（54.0%）向上する．

以上のように，優先度が 2 番目以降のスレッドはそれより上位のスレッドが実行できないときのみ実行されるために，3 番目以降のスレッドが実行される機会はかなり少ない．そのため，システム全体の性能が低くなり，画像処理のような演算量が多いソフトリアルタイムタスクが複数あるような場合には向いているとはいいがたい．

### 4.2 フェッチスレッド選択ポリシ

フェッチスレッド選択ポリシの評価を行う．評価環境は優先度特性の場合と同様のものを用い，最高優先度スレッドの性能の低下割合とシステム全体の性能の増加割合を評価する．

フェッチスレッド選択ポリシに用いた 5 つのパラメータのうち，1 種類のみを使用した場合について，最高優先度スレッドの実行時間と全スレッド合計の性能をそれぞれ単一スレッド実行時と比較したものを**図 11** に示す．図 11 において，棒グラフが全スレッド合計の IPC の増加割合，折れ線グラフが最高優先度スレッドの実行時間の増加割合を示している．全スレッド合計の IPC の増加割合が左側の軸，最高優先度スレッドの実行時間の増加割合が右側の軸を使用している．また，略称は IB が命令バッファ内命令数，P が分岐予測命令数，I がパイプライン中命令数，F がフェッチ制御パイプラインステージ数，R がリザベーション

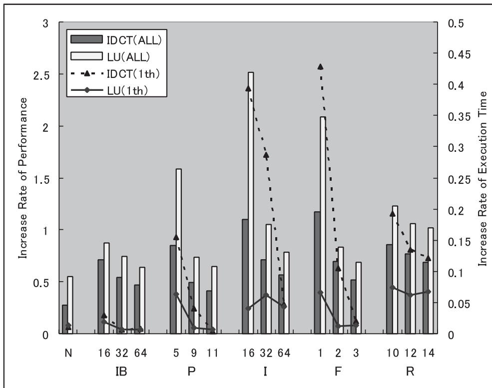


**図 11** フェッチスレッド選択ポリシ
Fig. 11 The policy of selecting the fetch thread.

ステーション内命令数である．比較として，ポリシを用いずに優先度を用いて 8 スレッド実行した場合を N として示している．

最高優先度スレッドの実行時間の増加割合に着目すると，図 11 の命令バッファ内命令数ポリシでは閾値が 64 のときに IDCT，LU ともに 0.6%，閾値を 32 の場合でも IDCT，LU ともに 0.7%しか実行時間は増加しない．また分岐予測命令数のポリシについても，閾値を 5 に設定すると実行時間の増加割合が大きくなるが，11 に設定すると IDCT で 0.5%，LU で 0.7%に実行時間の増加を抑えることができる．

命令バッファについては，前述したように 24 命令以上命令バッファに存在すれば，命令発行が滞らないので最高優先度スレッドの性能の低下を抑制できる．分岐予測命令数のポリシについては分岐予測ミスによるスロットの浪費を抑えることができる．つまりこれらのポリシは優先度の高いスレッドが浪費するスロットを優先度の低いスレッドが効果的に使用するので，最高優先度スレッドの性能を維持できる．両者のポリシの全体性能の性能改善は命令バッファの閾値が 32 のときに IDCT で 54.0%，LU で 74.0%，分岐命令の閾値が 11 のときに IDCT で 41.1%，LU で 64.7%であり，ポリシを用いない場合よりもいずれも 10%から 20%程度性能改善率が大きい．

以上より，有効な命令を命令実行機構に対して供給する割合を維持できる命令バッファ中の命令数やパイプライン中の分岐命令数に応じてフェッチを止めるポリシを実装することで，最高優先度スレッドの実行時間増加を 1%未満に抑え，システム全体の性能を 10%から 20%向上させることができた．このことは，時間制約が比較的厳しい複数のタスクをスケジューリングするのに効果的である．

IDCT と LU で最高優先度スレッドの性能増加割合が大きく異なる理由は以下のようになる．単一スレッド実行時の IPC の差が LU は IDCT と比較して 1.0 程度あり，データ依存関係により多くの命令がパイプライン中で待たされている．そのためフェッチに制限を加えて他のスレッドの命令を増加させても最高優先度スレッドに対する影響は低い．

次に全体性能の増加割合に着目すると，パイプライン中命令数やフェッチパイプライン占有ステージ数といったポリシの増加割合が大きいことが図 11 から分かる．パイプライン中命令数のポリシでは閾値を 16 に設定すると IDCT では 110%，LU で 252%性能が向上する．フェッチステージ数のポリシでは閾値を 1 にした場合で IDCT で 117%，LU で 209%性能が向上する．最高優先度スレッドの性能については，たとえば，パイプライン中命令数の閾値を 16 に設定すると IDCT では 39.2%実行時間が増加する．しかし，優先度を用いない場合の 8 スレッド実行時の増加割合は図 7 より 314%なので，その場合と比較するとそのタスクの時間粒度はかなり細かい．両者のポリシともにパイプライン中に多くのスレッドの命令を供給することになるので，レイテンシの隠蔽により性能が大きく改善されるが，優先度の高いスレッドの有効な命令スロットを減少させるので，実行時間が増加する．

以上より，システム全体の性能向上を重視したパイプライン中の命令数やフェッチステージ占有数によるフェッチを止める方式を実装することで，優先度の高いスレッドの実行時間が増加するが，システム全体の性能を大きく向上させることができた．優先度の高いスレッドの実行時間の増加割合は優先度を利用しない場合よりも小さいため，優先度の高いスレッドのデッドラインが比較的遠い場合や，特に演算量を必要とするタスク群をスケジューリングする際に効果的である．

リザベーションステーションの命令数による制御は最高優先度スレッドの増加割合に対する全スレッド合計性能の増加割合が他のポリシと比較して小さい．利用できるエントリが少ない分，投機実行できる命令数が少なくなるので性能が大きく減少してしまう．このポリシに関しては効果的とはいいがたい．

図 11 の結果より，最高優先度スレッドの実行時間増加割合の少ないポリシを組み合わせたものと，システム性能増加割合の大きいポリシを組み合わせたものの評価を行った．

評価したポリシを以下に示す．

(1) 分岐予測命令数 11，命令バッファ内命令数 32，フェッチステージ数 3

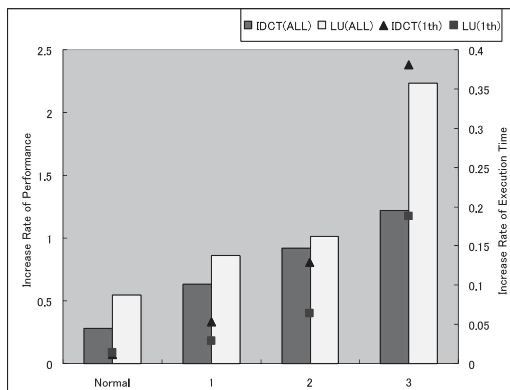


**図 12** フェッチスレッド選択ポリシ（複数パラメータ使用）
Fig. 12 The policy of selecting the fetch thread (use some thresholds).

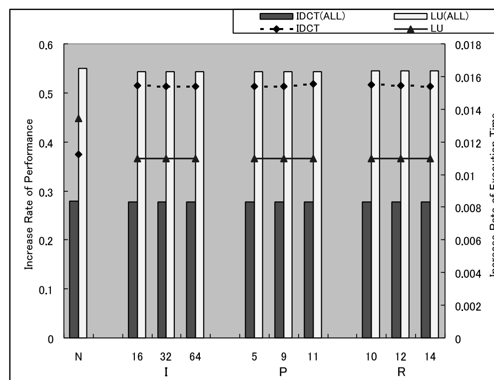


**図 13** 発行命令選択ポリシ
Fig. 13 The policy of selecting the instruction for issue.

（2） （1）にパイプライン中命令数 64 を追加
（3） 分岐予測命令数 9，命令バッファ内命令数 32，フェッチステージ数 2，パイプライン中命令数 32，リザベーションステーション内命令数 12

（1），（2）は最高優先度スレッドの性能を重視した場合，（3）はシステム全体の性能を重視した場合である．最高優先度スレッドの性能変化とシステム全体の変化割合を**図 12** に示す．最高優先度スレッドの実行時間変化割合を点で示している．他の特徴は図 11 と同様である．

（1）のポリシでは最高優先度スレッドの実行時間が IDCT で 5.3%，LU で 2.9%増加し，それに対しシステム全体性能が IDCT で 63.6%，LU で 86.2%向上する．ポリシ（2）では最高優先度スレッドの実行時間が IDCT で 12.9%，LU で 6.3%実行時間が向上するが，全体性能は IDCT で 91.7%，LU で 101.6%向上する．3 のポリシはシステム全体性能が IDCT で 122%，LU で 224%向上するが最高優先度スレッドの実行時間が IDCT で 38.1%，LU で 18.8%増加してしまう．

この結果を図 11 の場合と比較すると，たとえば LU でポリシ（1）の全体性能改善率 86.2%と同程度である，命令バッファ内命令数 16（87.1%）やフェッチパイプライン占有数 2（83.2%）では最高優先度スレッドの実行時間の増加はそれぞれ 1.9%，1.2%となり，いずれも増加割合がポリシ（1）の方が大きくなる．さらに，ポリシ（3）の場合では LU の全体性能改善率 224%より性能改善が可能なパイプライン中命令数 16（252%）の場合に最高優先度の実行時間の増加率は 4.0%とポリシ（3）の 18.8%よりも 1/4 以下に抑えることが可能である．IDCT の場合ではポリシ（3）の全体性能改善率 122%と同程度であるフェッチパイプライン占有数 1（117%）の場合では最高優先度スレッドの実行時間増加割合が 42.9%であり複数のポリシを組み合わせた方が効率が良い．

複数のポリシを組み合わせると，優先度の高いスレッドのフェッチが止まる可能性が高いために最高優先度スレッドの実行時間が増加することに加え，パイプライン中命令数を制限する場合と異なり，条件が複数重なるためにフェッチが止まる間隔が不定期になりやすくパイプライン中に各スレッドの命令をバランス良く供給できないために全体の性能の改善率もそれほど高くはならない．これは LU のようなパイプラインがストールしやすい場合に特に顕著となる．

### 4.3 発行命令選択ポリシ

発行命令選択に実装したポリシの評価を行う．フェッチスレッド選択ポリシの場合と同様の条件で評価を行う．

発行命令選択ポリシに用いた 3 つのパラメータのうち，1 種類のみを使用した場合について，最高優先度スレッドの実行時間と全スレッド合計の性能をそれぞれ単一スレッド実行時と比較したものを**図 13** に示す．図 13 において，棒グラフが全スレッド合計性能の増加割合，折れ線グラフが最高優先度スレッドの実行時間の増加割合を示している．また，略称は P が分岐予測命令数，I がパイプライン中命令数，R がリザベーションステーション内命令数である．

図 13 において，システム全体性能に着目するとポリシを用いない場合と比較して性能はわずかながら低下するが，その低下割合は 1 スレッド実行時の 0.1%未満である．図 13 に示されているようにパラメータを変更しても最高優先度スレッドの実行時間の増加割合についても 0.1%未満である．フェッチスレッド選択が優先度で行われるため，命令バッファ内に命令を格納

しているスレッド数が少ないので，これらのポリシの効果はほとんどない．

発行命令選択に実装したポリシは効果がほとんどなく，フェッチスレッド選択で選択されたスレッドの性能を落としてしまう．そこで，発行命令選択については通常の優先度制御のみを用いる．

### 4.4 共有資源制御

共有資源制御の評価を行う．共有資源制御の目的は最高優先度スレッドの性能低下をできるだけ抑えることである．前述したフェッチスレッドポリシをバッファの数を制限して実行した場合の最高優先度スレッドの実行時間の増加割合を評価する．

使用したポリシは図 12 の (1)，(3) である．最高優先度スレッドを重視した場合のポリシと，システム全体の性能を重視したポリシでどの程度最高優先度スレッドの性能に変化があるかを評価する．

各バッファの制限については，図 11 から命令バッファ数の閾値が 64 の場合に最高優先度スレッド性能の性能低下を抑えることを示しているので，命令バッファ数は最高優先度スレッドが 64 命令分使用する．リオーダバッファについては実行資源を最大限に使用できるように，最高優先度スレッドがすべてのバッファを使用できるように設定し，そのうちの 32 エントリを他のスレッドも使用できるように設定し，優先度の低いスレッドの優先度の高いスレッドへの性能への影響を抑える．以上より，以下のようにバッファ制限を設定した．

- 命令バッファ
  4 つのパーティション（64 命令）を最高優先度スレッドのみ，残り 4 つのパーティションを他 7 スレッドで共用
- リオーダバッファ
  12 個のパーティション（96 命令）を最高優先度スレッドのみ，残り 4 つのパーティションは 8 スレッドで共用

実行時間の変化割合を**図 14** に示す．図 14 中 (B) がバッファ制限した場合を示す．

ポリシ (1) の場合ではバッファ制限をしないときには最高優先度スレッドの実行時間増加割合が IDCT で 5.3%，LU で 2.9%であったのがバッファ制限を設けることで，IDCT で 2.8%，LU で 1.1%と抑えることができる．特に LU に関しては通常の優先度付き実行時の増加割合である 1.3%よりも軽減することができている．また，ポリシ (3) の場合ではバッファ制限をしない場合に最高優先度スレッドの実行時間増加割合が IDCT で 38.1%，LU で 18.8%であったのに対し，

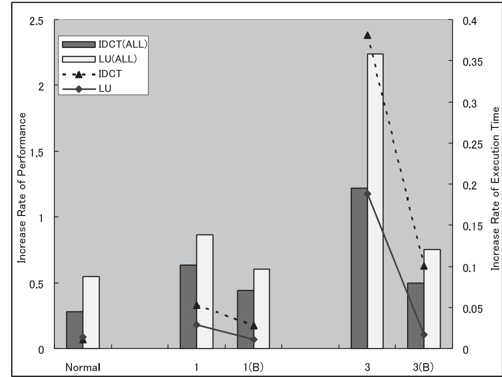


**図 14** バッファ制限の効果
Fig. 14 The effect of limiting number of buffer.

**表 2** 命令供給機構の面積
Table 2 The area of the instruction supply mechanism.

| | 面積 ($\mu m^2$) | 正方形換算時の 1 辺 (mm) |
|---|---|---|
| 従来の RMTPU | 8,143,219 | 2.853 |
| 本実装 | 8,152,298 | 2.855 |

バッファ制限を設けることで，IDCT で 10.0%，LU で 1.7%に抑えることができる．LU のように，データ依存等で，パイプライン内にとどまる命令数が多いプログラムの場合にバッファ制限は効果的である．

ただし，静的にバッファの使用条件を制限するためにシステム全体の性能はポリシ (1) の場合では IDCT で 63.6%から 43.9%に，LU で 86.0%から 60%に低下する．ポリシ (3) では IDCT で 122%から 49%に LU で 224%から 75%に低下する．バッファの割当てを静的に設定すると，バランス良く複数のスレッドから命令を実行できないので，システム全体の性能は低くなってしまう．

以上より，使用できるバッファ量を制限することで，スレッド間の性能への影響を抑えることができる．パイプラインがストールする率が高いようなタスクが最も優先度が高い場合に，優先度の低いスレッドが優先度の高いタスクに対する影響を抑えることができる．

### 4.5 ハードウェア量

評価に用いた命令供給機構の RTL を論理合成および配置し，従来の *RMT PU* との比較を行った．

TSMC 0.13 $\mu$m のプロセスで合成し配置した面積を**表 2** に示す．表 2 に示すように，本実装は従来の命令供給機構と比較して約 0.1%の面積増加に収まっている．本実装は命令供給機構の大部分を設計・実装しなおしているために，必ずしも本論文で述べた個所の実装による面積の増加分がこの値とはいえないが，全体の面積から比較すると微々たるものであるといえる．

## 5. 結 論

SMT 方式に優先度を導入してリアルタイム処理を行う *RMT PU* において，単純な優先度による制御だけでなく，パイプラインの状況によって低い優先度を持つスレッドを実行することで，高い優先度スレッドの性能低下を抑えながらプロセッサ使用率を向上させることを目的とした命令供給機構を設計・実装した．

最高優先度スレッドの性能低下をできるだけ抑えつつ，優先度の低いスレッドを実行するために，命令バッファ中の命令数や分岐命令数，パイプライン中の命令数やフェッチステージ内の命令数といった，パイプラインの状況に従って命令フェッチを止める機構を実装した．命令バッファ中の命令数や分岐命令数に従ってフェッチを止めるポリシでは最高優先度スレッドの性能低下を 1%未満に抑えつつ，全スレッド合計性能を 10%から 20%向上させることができた．また，パイプライン中の命令数や，フェッチステージ内の命令数に従うポリシでは，最高優先度スレッドの性能低下を約 40%程度に抑えつつ，全スレッド合計性能を約 100%向上させることができた．

前者のポリシを利用することで，従来の優先度のみを扱う手法と比較して最高優先度スレッドの実行を阻害せずにプロセッサ使用率を向上させることができるので，スケジューリングできるタスクの量が増加する．さらに，後者のポリシは演算量を多く必要とし，少しのデッドラインミスを許容できるソフトリアルタイムタスクを多く扱う際にも効果的である．

また，プロセッサ内の共有資源をスレッドごとに任意に割り振ることで，最高優先度スレッドの性能の低下を抑制した．

以上の機構の実装により，プロセッサの設定を変更することで性質の違うソフトリアルタイムタスクやハードリアルタイムタスクの両者のスケジューリングを支援できる．

現在，本実装の *RMT PU* を用いた新しいバージョンの *RMT Processor* のバックエンド設計（レイアウト，配置配線，DRC 等）を行っている．

**謝辞** 本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．

## 参 考 文 献

1) 内山真郷，伊藤　務，山崎信行，安西祐一郎：リアルタイム処理用マルチスレッドプロセッサの設計と実装，電子情報通信学会技術研究報告 CPSY99-122, pp.29–36 (2000).
2) 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).
3) Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L. and Stamm, R.L.: Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Processor, *Proc. 23rd Annual International Symposium on Computer Architecture*, pp.191–202 (1996).
4) 薄井弘之，内山真郷，伊藤　務，山崎信行：Responsive Multithreaded Processor における実時間処理用命令供給機構，情報処理学会研究報告，Vol.2004, No.33, pp.15–20 (2004).
5) Yamasaki, N.: Design Concept of Responsive Multithread Processor for Distributed Real-Time Control, *Journal of Robotics and Mechatronics*, Vol.16, No.2, pp.194–199 (2004).
6) Kreuzinger, J., Schulz, A., Preffer, M., Ungerer, T. and Brinkschutle, U.: Real-time scheduling on multithread processor, *Real Time Computing Systems and Applications* (*RTCSA*), pp.155–159 (2000).
7) Raasch, E.S. and Reinhardt, S.K.: Applications of Thread Prioritization in SMT Processors, *Workshop on Multithreaded Execution, Architecture and Compilation* (*MTEAC*) (1999).
8) Liu, J.W.: *REAL-TIME SYSTEMS*, Prentice Hall, pp.159–179 (2000).
9) Smith, E.J.: A Study of Branch Prediction Strategies, *Proc. 8th annual symposium on Computer Architecture*, pp.135–148 (1981).

(平成 16 年 1 月 31 日受付)
(平成 16 年 7 月 13 日採録)

**薄井　弘之**

1979 年生．2002 年慶應義塾大学理工学部情報工学科卒業．2004 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．同年株式会社東芝に入社．システム LSI 等の研究・開発に従事．

**内山　真郷**

1978 年生．2000 年慶應義塾大学理工学部情報工学科卒業．2002 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．同年株式会社東芝に入社．システム LSI 等の研究・開発に従事．

**伊藤　　務**（学生会員）

1977 年生．2000 年慶應義塾大学理工学部情報工学科卒業．2002 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，システム LSI 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．博士（工学）．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．