Vol.2013-ARC-206 No.7
2013/7/31

フロントエンドで命令を実行するプロセッサにおける エネルギー効率の評価

鷹見　怜[1]　塩谷 亮太[1]　安藤 秀樹[1]

概要：近年では高性能化への要求から，スマートフォンやタブレット端末などの携帯機器においても out-of-order スーパスカラ・プロセッサが広く採用されている．しかし，out-of-order スーパスカラ・プロセッサの消費エネルギーは大きく，この点が大きな問題となっている．これに対し，我々は高性能と低消費エネルギーを両立することを目的として，in-order と out-of-order の 2 つの実行系をもつフロントエンド実行方式を提案してきた．2 つの実行系のうち，in-order 実行系は演算器とバイパス・ネットワークのみからなる実行系であり, プロセッサのフロントエンドにおいて命令を実行する．また，out-of-order 実行系は通常のスーパスカラ・プロセッサの実行コアと同じものである. フロントエンド実行方式では，単純な in-order 実行系で多くの命令を実行することにより，out-of-order 実行系の消費エネルギーを減らしつつ，さらに性能を上げることができる．これまでに，このフロントエンド実行方式の性能は評価されているものの，消費エネルギーについては詳細なモデルの検討や評価が行われてこなかった．そこで本研究では，フロントエンド実行方式の消費エネルギーのモデルを構築し，シミュレーションにより評価した．評価の結果，最大でプロセッサ・コアの消費エネルギーを 34% 削減しながら，同時に性能を 23% 向上させることができることを確認した.

1. はじめに

マルチコアの時代においてもコアのシングルスレッド性能は依然として重要である．急速に普及が進んでいるスマートホンやタブレット端末においてもシングルスレッド性能は重視され，out-of-order スーパスカラ・プロセッサの採用が進んでいる．たとえば iPhone や iPad，主要な Android 端末に搭載されている ARM Cortex-A9 [1] やその後継 [2,6] は全て out-of-order スーパスカラ・プロセッサである．これらの端末ではユーザー要求の高まりからアプリケーションが高度化しており，従来よりも非常に高い性能が要求されている．また，これらの端末で動くアプリケーションは HTML5（javascript）や仮想マシン上に実装されることが多いが，これらは通常ネイティブ・バイナリよりも重たい上，javascript エンジンなどの処理系の並列化も容易ではない．このため，これらの端末ではシングルスレッド性能の高い out-of-order スーパスカラ・プロセッサを採用しているのである.

しかし，out-of-order スーパスカラ・プロセッサは高性能である半面，in-order プロセッサと比べて消費エネルギーが非常に大きい．これは，動的命令スケジューリングを行うハードウェアの消費エネルギーが非常に大きいためである．このハードウェアとは，具体的には発行キューやリオーダ・バッファ（Reorder Buffer: ROB）であり，その実体は多ポートのメモリである．多ポートのメモリでは 1 アクセスあたりの消費エネルギーがその容量とポート数に比例して増加する [8,12]．また，そのアクセス回数も同時発行幅に応じて増えるため，結果としてこれらの消費エネルギーは非常に大きなものとなる．これらの in-order プロセッサにはないハードウェアを持つため，out-of-order スーパスカラ・プロセッサは大型で消費エネルギーが大きくなる.

これに対し，我々は in-order 実行系と out-of-order 実行系の 2 つの実行系をもつ**フロントエンド実行方式** [15–17] を提案してきた. この手法では 2 つの実行系を組み合わせることにより，out-of-order スーパスカラ・プロセッサ以上の高性能と低消費エネルギー化を同時に実現する. 2 つの実行系のうち out-of-order 実行系は通常の out-of-order スーパスカラ・プロセッサの実行コアそのものであり，リザベーション・ステーション (Resevation Station: RS) や演算器などを含む．これに対し，in-order 実行系は主に演算器とバイパス・ネットワークからなる実行系であり，プロセッサのフロントエンドに位置する．フロントエンドでソース・オペランドを読み出し，その際レディであった命令はこの in-order 実行系で実行され，out-of-order 実行系へはディス

[1] 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University

ⓒ 2013 Information Processing Society of Japan

パッチしない．

この in-order 実行系では，ソース・オペランド読み出しの時点でレディであった命令に加え，in-order 実行系内で依存が解決された命令も実行できるため，非常に多くの命令を実行できる．また，演算器を複数段にわたって配置することにより，さらに実行可能な命令を増やしている（3 節）．実際に，5 節の評価では平均で全命令の半分程度を実行できることを示す．

この in-order 実行系で多くの命令を実行し，out-of-order 実行系へのディスパッチをフィルタすることで，フロントエンド実行方式は out-of-order 実行系の消費エネルギーを大きく削減できる．

これまで，フロントエンド実行方式の性能については評価されているが，消費エネルギーについては詳細なモデルの検討や評価をしていなかった．これに対し，本論文では，フロントエンド実行方式の消費エネルギーについて詳細な検討を行う．評価の結果，プロセッサ全体の消費エネルギーを最大 34%削減しながら，性能を 23%向上させることができることを確認した．

本論文の以降の構成は以下の通りである．2 節では，フロントエンド実行方式の背景として RS ベースの out-of-order スーパスカラ・プロセッサについて説明する．3 節ではそのアーキテクチャの基本的な方式について説明し，4 節では消費エネルギーについて述べる．5 節ではシミュレーションにより定量的な評価を行い，6 節でまとめる．

2. RS ベースのアーキテクチャ

本節では，フロントエンド実行方式の背景として RS ベースの out-of-order スーパスカラ・プロセッサを説明する．Out-of-order スーパスカラ・プロセッサのアーキテクチャには，大きく分けて RS ベース [4,9] と物理レジスタ・ファイル・ベース [5,10] の 2 つがある．これらのうち，フロントエンド実行方式は RS ベースのアーキテクチャをベースとしている．以降では**通常のスーパスカラ・プロセッサ**とした場合，この RS ベースのアーキテクチャを指すものとする．

フロントエンド実行方式ではフロントエンドで命令を実行することに特徴があるため，本節ではフロントエンドに焦点を当てて説明する．RS ベースのアーキテクチャのパイプラインとブロック図を図 1 に示す．命令はパイプラインの各ステージにおいて以下のように処理される：

(1) **fetch**: 命令キャッシュから命令をフェッチする．

(2) **rename**: 命令のオペランドはリネーム・ロジックによりリネームされる．具体的には論理レジスタ番号を用いて Register Alias Table (RAT) と呼ぶテーブルを引くことにより，ソース・オペランドに対応する ROB または論理レジスタ・ファイル（Logical Register File: LRF）のエントリ番号を得る[*1]．なお，ROB は古典的には CAM で構成されるが，本論文では ROB を RAM で構成し，RAT を介して読み出す方式 [4] をベースとする．

(3) **register read**: rename ステージで得たエントリ番号を用いて ROB または LRF を参照し，ソース・オペランドを得る．プロデューサ命令がまだ実行されておらず，ソース・オペランドが得られない場合は，後述するように RS にてオペランドを待ち合わせる．

(4) **dispatch**: register read ステージで読み出したソース・オペランド，op コードなどを RS に書き込む．RS では，まだ得られていないソース・オペランドを待ち合わせる．

(5) **issue**: ソース・オペランドが揃った命令とそのソース・オペランドが RS から読み出され，演算器に対して発行される．

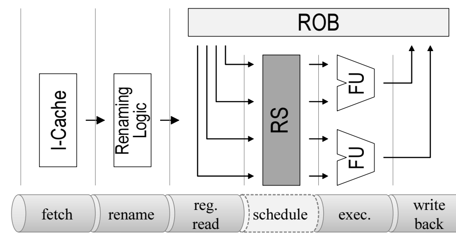

図 1　RS ベースのアーキテクチャ

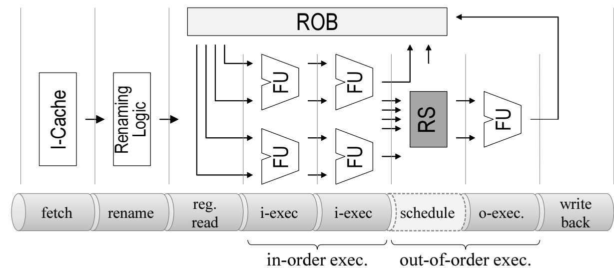

図 2　フロントエンド実行方式

その後，発行された命令は実行され，実行結果を ROB と RS に書き込む．なお，これらのステージのうち dispatch と issue は，図 1 では schedule と書かれている部分に含まれる．

3. in-order 実行系をもつアーキテクチャ

フロントエンド実行方式は，RS ベースのスーパスカラ・プロセッサをベースにしている．フロントエンド実行方式では，フロントエンドでソース・オペランドの読み出しが終了した後，レディな命令を in-order 実行系で実行する．以下では提案手法のハードウェア構成について説明した後，その基本的な動作について述べる．

*1　図 1 では, LRF を省略している．

3.1 ハードウェアの構成

フロントエンド実行方式のブロック図とそのパイプラインを図 2 に示す．このアーキテクチャは以下の二つの実行系をもつ．

(1) **out-of-order 実行系**: 通常のスーパスカラ・プロセッサの実行コアと同様のものである．図 2 の schedule 以降のステージがこれにあたり，主に演算器やバイパス・ネットワーク，動的命令スケジューリングを行うための RS などからなる．この実行系では，命令は通常の out-of-order スーパスカラ・プロセッサと同様に，RS でソース・オペランドの待ち合わせを行い，依存が解決したものから順に発行される．

(2) **in-order 実行系**: 主に演算器とそれらを接続するバイパス・ネットワークからなる．この実行系は，フロントエンドの register read ステージと dispatch ステージの間に配置される．register read ステージで ROB または LRF から読み出された値は，この in-order 実行系に入力され，in-order に実行する．

in-order 実行系はこの他に frontend load buffer (FLB) と呼ぶごく小容量のバッファ（数エントリから 10 数エントリ程度）を持つ．この FLB は L1 データ・キャッシュに対するキャッシュとして働き，ロード命令に値を供給する．

3.2 命令パイプラインの基本的な動作

in-order 実行系は，out-of-order 実行系に対するフィルタとして働く．すなわち，in-order 実行系で実行された命令は out-of-order 実行系で実行せず，命令パイプラインから取り除かれる．以下ではこのことを通常のスーパスカラ・プロセッサと比較しながら説明する．なお，ここでは命令は 1 サイクルで実行できる整数演算命令であるとして説明する．それ以外の命令については後の 3.4 節で詳しく説明する．

2 節で述べた通常のスーパスカラ・プロセッサの動作に対し，フロントエンド実行方式ではソース・オペランドを読み出すまでは同じものの，その後の処理が異なる．以下は in-order 実行系による命令処理のおおまかな流れである：

(1) フロントエンドでソース・オペランドを ROB または LRF から読み出す．

(2) 命令がレディであるかどうかを判定する．レディである条件は，ソース・オペランドが全て得られていることである．ソース・オペランドは以下の 2 通りの経路によって得られ，これらによってソース・オペランドが揃うかどうかを判定する：

(a) ROB または LRF からの読み出し．

(b) In-order 実行系内の実行結果のバイパス．

(3) 命令がレディかどうかに応じて，in-order 実行系では以下のように処理する：

(a) レディであった命令は in-order 実行系で実行し，RS にディスパッチしない．また，その実行結果は in-order 実行系を出た後に ROB に書き戻される．

(b) レディではない命令は，in-order 実行系を NOP としてそのまま通過する．その後，命令は RS にディスパッチされ，実行される．

上記のうち，レディではない命令の処理が通常の in-order プロセッサの場合と異なることに注意されたい．通常の in-order プロセッサでは，レディではない命令のデコード時は，依存が解消されるまでパイプラインはストールさせる．これに対し本研究の in-order 実行系では，レディではない命令は NOP として通過し，パイプラインはストールされずに命令は流れ続ける．

フロントエンド実行方式では in-order 実行系で多くの命令が実行できるため，out-of-order 実行系へディスパッチされる命令は少なくなり，その結果 out-of-order 実行系の消費エネルギーを大きく削減できる．

2 つの実行系のうち，out-of-order 実行系は通常のスーパスカラ・プロセッサと同様の構成をとるため，新たに説明することはない．次節では，この in-order 実行系内の命令処理について，具体例をあげながら説明する．

3.3 in-order 実行系の動作の詳細

本節では，in-order 実行系内の構造や命令処理について説明する．in-order 実行系では，実行できる命令数を増やすために，演算器を 2∼3 段に直列に配置する．図 3 に in-order 実行系内のデータ・パスの例を示す．同図は 2 命令幅 × 2 段の演算器を持つ場合を示す．同図の $FU(y,x)$ は，x ステージ目の上から y 番目の演算器を示し，例えば $FU(0,1)$ は右上の演算器を示す．各演算器は，その実行結果を次のサイクルに使えるよう，バイパス・ネットワークで接続される．同図の左側から ROB または LRF から読み出したソース・オペランドが入力される．また，右側へは演算結果が出力され，ROB へ書き戻される．

この in-order 実行系上で図 5 のコードを実行した場合を例としてその動作を説明する．同図のコードは，プロデューサ I_p と，コンシューマ I_{c0}，I_{c1} を含むものである．これらの命令のソース・オペランドのうち C 以外は全て ROB または LRF から読み出されているものとする．以下では，その動作についてサイクル順に説明する．

0 サイクル目： 図 4(a) は，in-order 実行系の 1 ステージ目に I_p と I_{c0} の 2 命令があるサイクルを示している．この時，$FU(0,0)$ 上の I_p はソース・オペランドが全て揃っているため，ここで実行され C を出力する．この出力は，自身の前方にある演算器と，次のサイクルにそれを使用する演算器に対して送信される．たとえば $FU(0,0)$ の実行結果 C は，この時自分の前方にある $FU(0,1)$ と，次のサイクルに C を用いる $FU(0,0)$ と $FU(1,1)$ に対して送られる．一方，I_{c0} はソース・オ

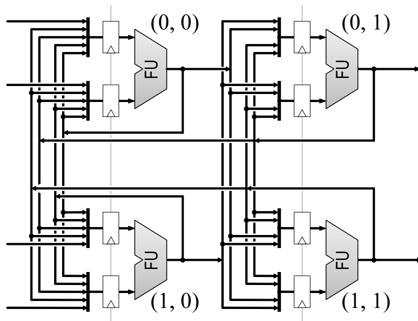

図 3　in-order 実行系のデータパス

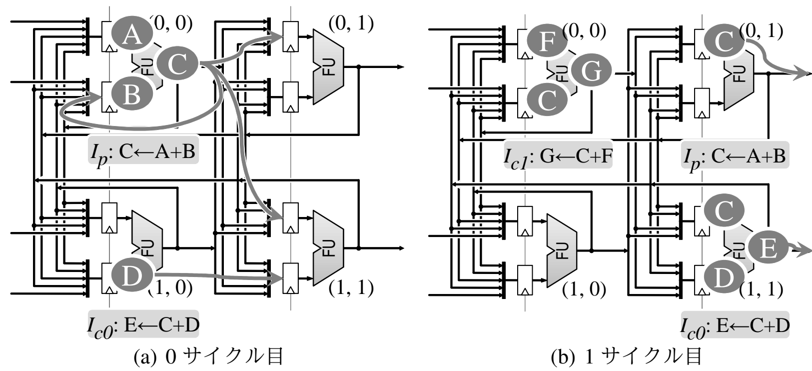

図 4　in-order 実行系内の命令の実行の様子

I_p :　C ← A + B
I_{c0} :　E ← C + D
I_{c1} :　G ← C + F

図 5　in-order 実行系上で実行されるコードの例

ペランドの C がまだ計算されていないため，$FU(1,0)$ では演算を行わず，もう 1 つのソース・オペランドである D を次段に送る．

1 サイクル目：　図 4(b) は図 4(a) の 1 サイクル後の様子である．同図では I_p と I_{c0} が 2 ステージ目に移動し，新たに I_{c1} が 1 ステージ目に送られる．$FU(1,1)$ 上の I_{c0} は前サイクルで実行された C がソース・ラッチに送られてきているため，これを使って演算する．$FU(0,0)$ 上の I_{c1} も同様であり，C を使って演算する．$FU(0,1)$ 上の I_p は既に 0 サイクル目で実行されているため，ここでは何もせず，実行結果 C をそのまま出力する．

このように多段に配置された演算器を使うことにより，in-order 実行系では I_p と I_{c0} のような互いに依存関係にある命令が並列に in-order 実行系に入った場合でも実行することができる．

in-order 実行系の制御，すなわち，どの演算器上で命令を実行するかやオペランド通信の制御は，オペランド・バイパスと同様にしてレジスタ番号を比較して決定する．この比較では論理レジスタ番号を用いるため，制御の決定はリネームやレジスタ読み出しと平行して行うことができる．in-order 実行系にはこの時生成した制御信号を命令とともに送り，演算器やバイパスを制御する．このため，制御の決定のためにクリティカル・パスが伸びることはない．

3.4　その他の命令の動作

ここまでは整数演算命令の実行について in-order 実行系の動作を説明してきた．これに対して，本節ではそれ以外の分岐命令，浮動小数点 (FloatingPoint: FP) 命令，ロード/ストア命令の実行について順に説明する．

3.4.1　分岐命令

整数演算命令と同様にして，分岐命令は in-order 実行系で実行される．in-order 実行系で分岐命令の実行結果と予測結果の比較を行い，分岐予測ミスを検出する．予測ミスが検出された場合，その時点でパイプラインをフラッシュし，リカバリする．このようにインオーダ実行系で分岐命令を実行することで，通常のスーパスカラ・プロセッサよりも上流のステージで予測ミスを検出し，分岐予測ミス・ペナルティを軽減することができる．

3.4.2　FP 命令

in-order 実行系には FP 演算器は搭載しない．これは，回路面積のオーバヘッドを抑えることと，パイプライン長が伸びることを防ぐためである．一般に，FP 演算のレイテンシは 3 サイクル程度であり，これを多段に配置した場合パイプライン長が大きく伸びてしまう．このため，in-order 実行系では FP 命令は実行せず，全て out-of-order 実行系で実行する．

3.4.3　ロード/ストア命令

ロード/ストア命令は，out-of-order 実行系側の資源と調停をしながら in-order 実行系で実行する．この時調停する資源とは，ロード・ストア・キュー（Load Store Queue: LSQ）や L1 データ・キャッシュである．この調停では，out-of-order 実行系側で実行される命令が優先される．in-order 実行系で命令が実行できない場合はそのまま out-of-order 実行系にディスパッチすればよいため，パイプラインを止める必要がなく，性能への影響が小さいためである．

in-order 実行系では，ロード命令に依存した命令の実行に制限がある．通常，ロード命令の実行レイテンシは，キャッシュ・ヒット時においても数サイクルかかる．このため，in-order 実行系で実行されたロード命令の結果はバイパス・ネットワークを介しても後続の命令に渡せず，ROB に書き込まれるまで in-order 実行系で使用できない．

この問題を緩和するのが，3.1 節で述べた FLB である．FLB は小容量であるため，1 サイクルでアクセスできる．このため，FLB にヒット時に得られた値は，in-order 実行系内で使うことができ，in-order 実行系で実行できる命令数を増やすことができる．

ⓒ 2013 Information Processing Society of Japan

4. フロントエンド実行方式の消費エネルギー

本節では，フロントエンド実行方式によるプロセッサの消費エネルギーについて説明する．フロントエンド実行方式では，通常のスーパスカラ・プロセッサと比較して消費エネルギーが削減される．以下ではこれを，
1) in-order 実行系の追加による消費エネルギーの増加，
2) out-of-order 実行系の縮小による消費エネルギーの削減，
に分けて説明する．

4.1 in-order 実行系の消費エネルギー

in-order 実行系の追加による消費エネルギーの増加は，プロセッサ・コア全体の消費エネルギーを大きく増やさない．3 節でも説明したように，in-order 実行系の実体は演算器とバイパス・ネットワークであり，同時発行数に応じて大きくなる多ポート・メモリを含まない．このため，その消費エネルギーは主に演算器とバイパスに関わる消費エネルギーによって決まる．以下ではこれらについて順に説明する．

4.1.1 演算器

演算器の消費エネルギーは個々の演算器の消費エネルギーと，そのアクセス回数によって決まる．個々の演算器は通常のスーパスカラ・プロセッサと提案手法の間で同じであるため，アクセス 1 回あたりの消費エネルギーは同じである．また，全ての命令は必ずどこかの演算器で 1 度実行されるため，両者の演算器への総アクセス回数も大きくは変わらない．これらのかけ算によって求まる演算器の総消費エネルギーは，結果として両者でほとんど変わらない．

4.1.2 バイパス

3.3 節で説明した in-order 実行系内のバイパス・ネットワーク は，一般的な RF のバイパス・ネットワーク（以下，通常のバイパス・ネットワーク）と同程度の複雑さであり，消費エネルギーも大きくは変わらない．以下ではそれぞれのバイパス・ネットワークを比較しながら説明する．

通常のバイパス・ネットワークは，レジスタ番号の比較器と，オペランドを選択するためのセレクタからなる．バイパス対象のオペランドは RF のレイテンシに発行幅をかけた数だけあるため[*2]，セレクタの入力数はこれに一致する．また，ソース・オペランドごとに必要な比較器の数もこれに一致する．これらのセレクタや比較器はソース・オペランドの総数分だけ存在する．以上をまとめると，RF のレイテンシを RL，発行幅を IW とした場合，セレクタの入力数 SI，セレクタの総数 SN，比較器の総数 CN はそれぞれ以下のように表される：

$$SI = RL \times IW \tag{1}$$

[*2] 実際には RF のレイテンシの 2 倍の期間バイパスを行う必要があるが，ここでは簡単のため 1 倍とした．

$$SN = 2 \times IW \tag{2}$$

$$CN = 2 \times RL \times IW^2 \tag{3}$$

in-order 実行系のバイパス・ネットワークもまたレジスタ番号の比較器とセレクタからなる（3.3 節）．in-order 実行系内の各演算器は最大で全ての演算器から実行結果を受け取る．このため，セレクタの入力数は演算器の総数に一致する，また，このセレクタはソース・オペランドの総数分だけ存在する．比較器については，in-order 実行系のバイパス・ネットワークでは演算器の各段に用意する必要はなく，演算器の幅の分だけあればよい．これはレジスタ番号の比較を 1 命令につき 1 回行えばオペランドの送信元がわかるためである．3.3 節でも述べたように，in-order 実行系ではこの比較を命令が in-order 実行系に入る前に行なっている．このため比較器の数は，演算器の総数に 1 段あたりのソース・オペランドの総数をかけたものとなる．これらより，in-order 実行系の段数を FS，幅を FW とした場合，セレクタの入力数 SI，セレクタの総数 SN，比較器の総数 CN はそれぞれ以下のように表される：

$$SI = FS \times FW \tag{4}$$

$$SN = 2 \times FS \times FW \tag{5}$$

$$CN = 2 \times FS \times FW^2 \tag{6}$$

以上を元に，5 節で用いたパラメータを使用して，それぞれの回路規模を比較する．レジスタ・ファイルのレイテンシを 2 サイクル，発行幅を 4 とした場合，通常のバイパス・ネットワークにおいては $SI = 8$，$SN = 8$，$CN = 64$ となる．一方，in-order 実行系のステージ数を 3，幅を 3 とした場合，そのバイパス・ネットワークにおいては $SI = 9$，$SN = 18$，$CN = 54$ となる．これらを比較した場合，両者の回路規模は大きくは変わらないことがわかる．なお，各命令はプログラム・オーダ上で自身の上流にいる命令に実行結果を送ることはないため，in-order 実行系の実際の回路規模はこれよりも小さい．また，通常のバイパス・ネットワークの比較器が物理レジスタ番号を比較しているのに対し，in-order 実行系では論理レジスタ番号を比較しているため，比較器の幅も小さくなる．

以上の議論より，in-order 実行系のバイパス・ネットワークは通常のバイパス・ネットワークと同程度の複雑さであるため，その消費エネルギーも大きくは変わらない．

4.2 RS

フロントエンド実行方式は，通常のスーパスカラ・プロセッサと比べて RS の消費エネルギーが大きく削減される．フロントエンド実行方式は，in-order 実行系によって多くの命令が実行されるため，性能を落とすことなく RS の容量と同時発行数を削減できる．この RS の実体は多ポートの

メモリであり，1 アクセスあたりの消費エネルギーは容量とポート数のそれぞれに比例する [8, 12]．このため，容量と同時発行数（ポート数）を削減することは，RS の 1 アクセスあたりの消費エネルギーを大きく削減させる．また，in-order 実行系で実行された命令は RS にディスパッチしないため，そのアクセス回数自体も大きく減る．5 節の評価では，RS の消費エネルギーが従来のスーパスカラ・プロセッサと比較して数分の 1 にまで縮小されることを示す．

4.3 ROB

ROB の消費エネルギーもまた通常のスーパスカラ・プロセッサと比べて削減される．以下では ROB の消費エネルギーをアクセス回数とアクセス 1 回あたりの消費エネルギーに分けて考える．

4.3.1 アクセス回数

ROB へのアクセス回数は，通常のスーパスカラ・プロセッサとフロントエンド実行方式では変わらない．フロントエンド実行方式と通常のスーパスカラ・プロセッサでは，ROB の読み出し方法は全く同じであるため，その回数も同じである．また，ROB の書き込み回数も変わらない．これは，全ての命令はその結果を in-order 実行系か out-of-order 実行系のどちらかで 1 度だけ ROB に書き込むためである．

4.3.2 アクセスあたりの消費エネルギー

アクセス 1 回あたりの 消費エネルギーはフロントエンド実行方式により削減される．ROB は多ポートの RAM からなるため（2 節)，ROB のアクセス 1 回あたりの消費エネルギーは容量とポート数のそれぞれに比例する．これらのうち，フロントエンド実行方式では書き込みのポート数が削減される．

フロントエンド実行方式では ROB の書き込みポート数は，

(a) out-of-order 実行系の書き込みポート数と，

(b) in-order 実行系 の書き込みポート数

を加えたものとなる．(a) はフロントエンド方式では通常のスーパスカラ・プロセッサと比べて削減される．これは RS の同時発行幅が削減されるためである．一方，(b) は in-order 実行系の追加によって増加する．この増加分は in-order 実行系の幅とは無関係に常に 1 ポートである．これは，in-order 実行系による ROB への書き込みは，プログラム・オーダに従って連続したエントリに対して行われるため，インターリーブすることができるためである[*3]．これらの結果，フロントエンド実行方式では ROB の書き込みのポートが若干削減される．たとえば 5 節の評価で用いた構成では，通常のスーパスカラ・プロセッサの書き込みポート (a) は 4 ポートである．これに対し，フロントエンド実行方式では，(a) は 4 ポートから 2 ポートに削減され，(b) として 1 ポートが追加される．このため，全体として書き込みポートが 4 から 3 に削減され，ROB の消費エネルギーが削減される．

*3 同様のインタリーブはコミット時の実行結果の読み出しポートでも行われている．たとえば実際に MIPS R10000 の ROB（Active List）ではコミット時の読み出し部分は 1 ポートである [13]

4.4 LSQ

フロントエンド実行方式では in-order 実行系で実行されるストアについては順序違反の検出を省略できるため，その消費エネルギーが削減される．フロントエンド実行方式では，ストア・セット予測器などの依存予測器にもとづいてロード/ストア命令を投機的に発行する方式 [3] を想定している．この方式では，ストア命令を実行した際に，プログラム・オーダ上で自身に後続するロードが既に同じアドレスを読みだしていた場合に，それを違反として検出する．in-order 実行系でストアを実行した場合は，自身より後続のロードが先に実行されていることはありえないため，これを省略することができる．

以上で説明したように，フロントエンド実行方式では通常のスーパスカラ・プロセッサと比べて RS と ROB，LSQ の消費エネルギーが削減され，演算器の消費エネルギーは変わらない．RS の消費エネルギーは非常に大きいため，その消費エネルギーの削減は，プロセッサ全体の大きな消費エネルギー削減につながる．また，フロントエンド実行方式は，性能向上によるプログラム実行時間の短縮により，プロセッサ全体の静的消費エネルギーを削減する．

5. 評価

シミュレーションにより，フロントエンド実行方式の性能と消費エネルギーを評価した．以降では評価環境について述べた後，評価結果について説明する．

5.1 評価環境

本節では，性能と消費エネルギーの評価環境について説明する．性能の評価には，プロセッサ・シミュレータ鬼斬 [14] を用いた．また，消費エネルギーの評価は McPAT [7] を用いて 22nm テクノロジを想定して行った．評価には SPECCPU2006 [11] に含まれる全ベンチマークを用いた．ベンチマークのコンパイルには gcc 4．5．3 を用い，コンパイル・オプションには “-O3” を指定した．入力セットには ref を使用し，プログラムの先頭 4G 命令をスキップした後の 100M 命令について測定した．

5.2 評価モデル

表 1 に評価モデルについて示す．ベースラインとなる構成は，フェッチ幅や発行幅，ROB サイズなどの主要なパラメータを ARM Cortex-A57 [2] のそれらと同一としている．このベースラインに対し，以下のモデルを実装して評

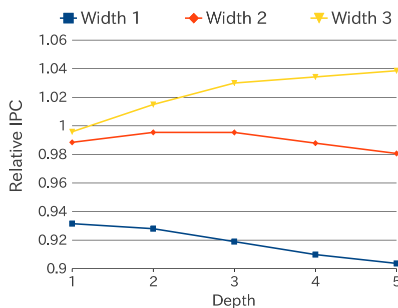

図 6　in-order 実行系の段数と幅に対する性能の変化

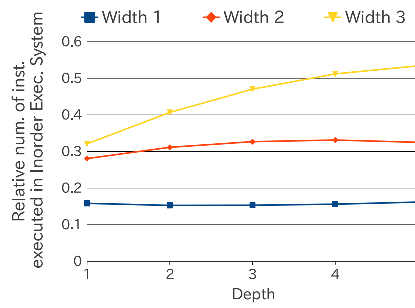

図 7　in-order 実行系で実行された命令の割合の変化

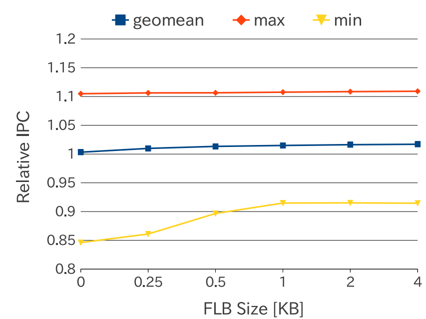

図 8　FLB サイズに対する性能の変化

表 1　評価モデルの構成

	Base	Narrow
Pipeline Stages	Fetch:3，Rename:1，Reg. Read:2，Dispatch:1，Issue:1	←
Fetch Width	3	←
Issue Width	4	2
RS	64 entries	32 entries
Functional Units	ALU:2，FPU:2，MUL:1，MEM:2	←
Store Queue	32 entries	←
Load Queue	32 entries	←
ROB	128 entries	←
L1 I-Cache	48KB，12-way，3 cycles，64 bytes/line	←
L1 D-Cache	32KB，8-way，4 cycles，64 bytes/line	←
L2 Cache	512KB，8-way，10 cycles，64 bytes/line	←
Main Memory	200 cycles	←

表 2　in-order 実行系のパラメータ

Width	1～3
Depth	1～4
Frontend Load Buffer	0～4 KB，4-way，1 cycle 64 bytes/line

価した：

Base :　ベースラインとなるモデル．

Narrow :　Base において，命令の発行幅と RS の容量を削減したモデル．パラメータを表 1 の右側に示す．

FrExec :　フロントエンド実行方式を実装したモデル．out-of-order 実行系の構成については Narrow モデルと同様である．in-order 実行系の構成を表 2 に示す．同表の Width は，in-order 実行系の各ステージにおける演算器の数を示す．また，Depth は in-order 実行系のステージ数を示す (3.3 節)．演算器の幅は最大でフェッチ幅であり，3 命令である．

5.3　in-order 実行系の構成の検討

フロントエンド実行方式には多数のパラメータがあり，それらが相互に影響する．このため，本節では演算器の構成などのパラメータを変化させて性能を評価し，以降で使用するパラメータを決定する．

5.3.1　ステージ数と幅の評価

本節では，in-order 実行系の演算器のステージ数と幅を変化させた際の IPC について評価する．図 6 に結果を示す．ここで，FLB のサイズは 1KB とした．同図の縦軸は Base で正規化した IPC を，横軸は in-order 実行系のステージ数を示す．同図の IPC は，全ベンチマークにおける相対 IPC の幾何平均を表し，それぞれの折れ線は演算器の幅ごとの IPC を示す．同図より，演算器の幅が 3 より狭い場合は Base よりも性能が低下してしまうことがわかる．また，演算器の幅が 3 の場合はステージ数が 3 段までは性能が向上しており，それ以降の性能向上はゆるやかである．したがって，以降では演算器の幅を 3，ステージ数を 3 として評価を行う．

なお，同図では幅が 1 と 2 の場合はステージ数の増加とともに性能が低下しているが，この性能低下は分岐予測ミス・ペナルティの増加によるものである．FrExec では分岐命令が out-of-order 実行系で実行された場合，パイプライン長の増加により予測ミス・ペナルティが増加する．このペナルティの増加により，ステージ数を増加させても in-order 実行系で実行される命令数が増えなかった場合は性能が下がってしまう．

図 7 に，同様にして演算器のステージ数と幅を変化させた際の in-order 実行系で実行された命令の割合を示す．同図の縦軸は全実行命令に対する in-order 実行系で実行された命令の割合を，横軸は in-order 実行系のステージ数を示す．同図の結果は全ベンチマークの結果の幾何平均であり，図内の各折れ線は演算器の幅ごとの結果を示す．同図より，幅が 3 の場合にはステージ数の増加とともに in-order 実行系で実行された命令が増えている．これに対し，演算器の幅がそれより狭い場合にはステージ数を増やしても実行された命令がほとんど増えていない．このような場合，前述したように分岐予測ミス・ペナルティのみが増えてしまい，性能低下につながる．

ⓒ 2013 Information Processing Society of Japan

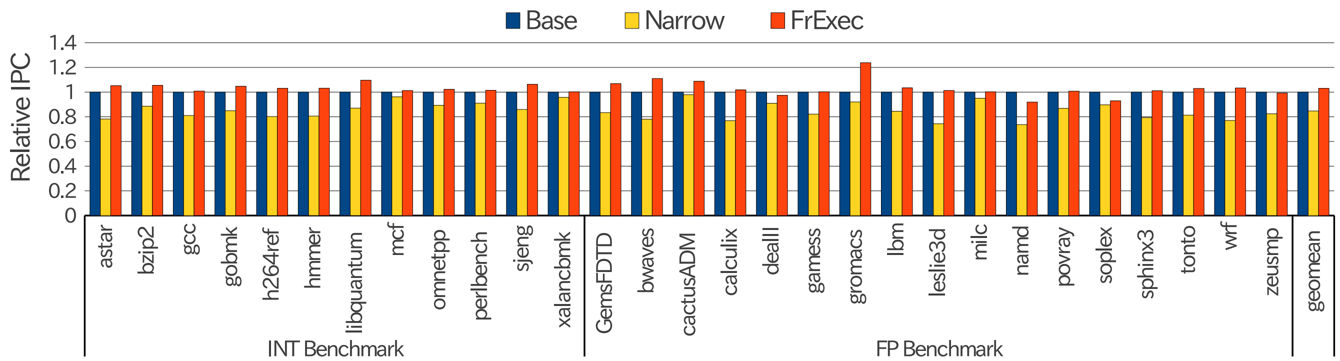

図 9 Base に対する IPC

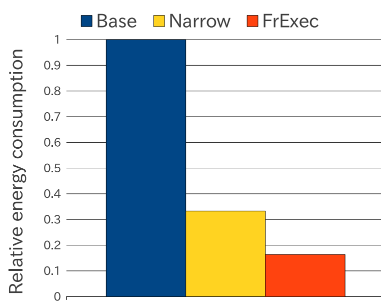

図 10 RS の消費エネルギー

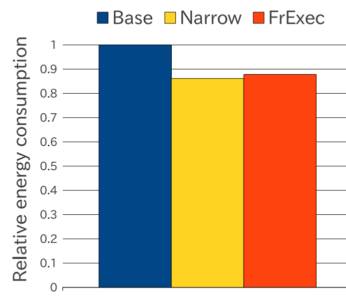

図 11 ROB の消費エネルギー

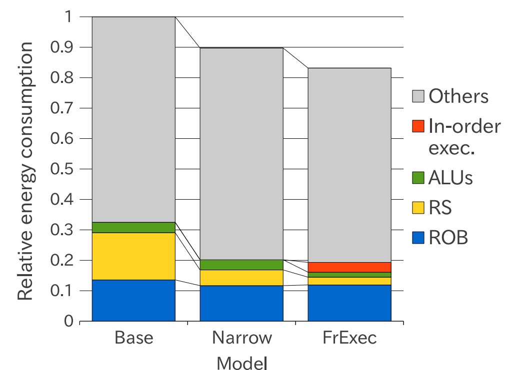

図 12 プロセッサ・コア全体の消費エネルギー

5.3.2 FLB サイズの評価

つぎに，FLB のサイズを変化させた際の IPC ついて評価する．図 8 に FLB サイズに対する IPC の変化を示す．同図の縦軸は Base で正規化した IPC を，横軸は FLB サイズを示す．それぞれの折れ線は全ベンチマーク中の最大，最小，幾何平均を示す．図 8 より，FLB サイズが 1KB 以上では IPC は大きく変化しない．このため，以降では FLB サイズは 1KB として評価する．

5.4 IPC

図 9 に，各モデルの Base に対する相対 IPC を示す．同図の縦軸は Base で正規化した IPC を，横軸はベンチマーク名を示す．図 9 より，Narrow は Base と比べて平均で 16%と大きく性能を低下させている．特に leslie3d や namd の性能低下は大きく，それぞれ 25%，26%性能が低下している．一方，FrExec は Base と比べて平均で 3%性能を向上させており，特に gromacs では 23%と性能が大きく向上している．一部のベンチマークでは性能が低下しているものの，低下率は小さく，最悪の場合でも soplex の 8%である．また，Narrow と比較した場合は FrExec は性能を 21%向上させており，in-order 実行系の追加により大きく性能が向上させられることがわかる．

5.5 消費エネルギー

本節では，各モデルの消費エネルギーについて評価する．まず，4 節で述べた RS，ROB の消費エネルギーについて評価した後，プロセッサ・コア全体の消費エネルギーについて評価する．

5.5.1 RS

図 10 に RS における消費エネルギーについて示す．同図の縦軸は，Base で正規化した消費エネルギーを示す．図内の 3 組のバーは全てのベンチマークの消費エネルギーの平均を示す．各バーは左から順に Base，Narrow，FrExec を示す．図 10 より，Narrow は RS の縮小により Base と比べて消費エネルギーを 74%削減している．FrExec では RS そのものの縮小に加えてアクセス回数も削減されるため (4.2 節)，Narrow よりもさらに消費エネルギーが小さく，Base と比べて 83%削減されている．

5.5.2 ROB

図 11 に ROB における消費エネルギーについて示す．同図の見方は図 10 と同様である．FrExec は Base と比べて消費エネルギーを 12%削減している．これは，Base と比較して ROB の書き込みポートが削減されているためである (4.3 節)．FrExec は Narrow と比較すると 2 %消費エネルギーを増加させているが，これは in-order 実行系で実行された命令の結果を書き込むために Narrow よりも書き込みポートが 1 つ増えているためである．ポート数の増加に対して消費エネルギーの増加が大きくないのは, 静的な消費エネルギーのためである.FrExec は Narrow よりも大幅に性能が高いため, 静的消費エネルギーが削減されており, ポー

ⓒ 2013 Information Processing Society of Japan

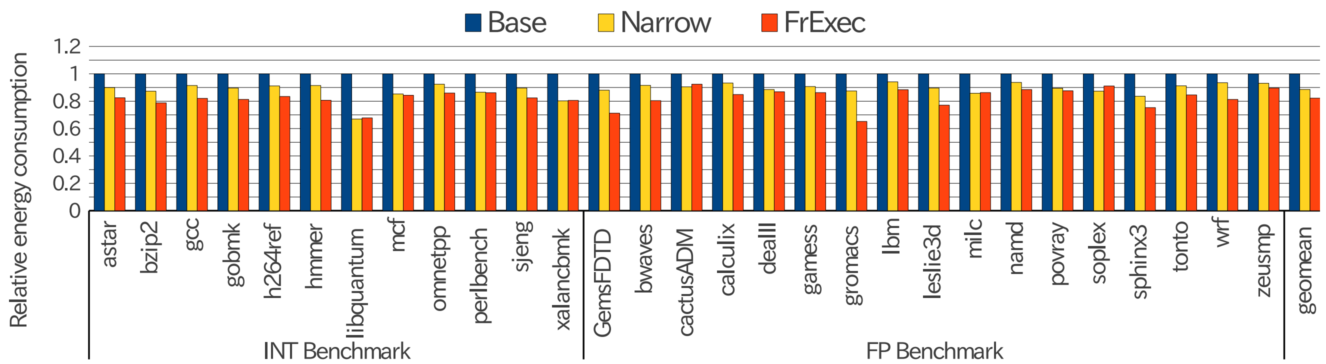

図 13　Base に対する消費エネルギー

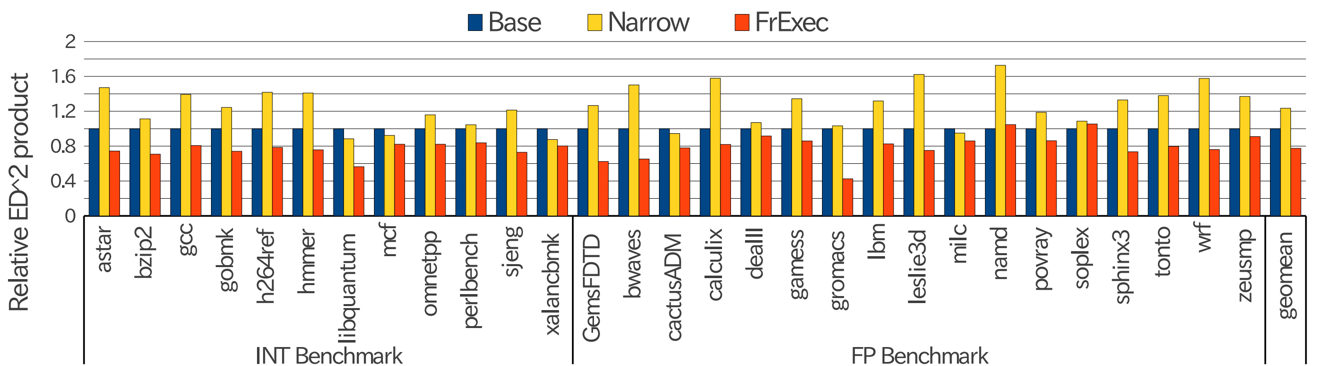

図 14　Base に対する ED^2 積

ト数の増加による消費エネルギーの増大を相殺している．

5.5.3　プロセッサ・コア全体の消費エネルギー

図 12 にプロセッサ・コア全体における in-order 実行系の消費エネルギーの割合を示す．同図の縦軸に Base で正規化した消費エネルギーを，横軸に各評価モデルを示す，同図の積み上げ棒グラフは全ベンチマークの平均を示し，棒グラフの各ラベルは下から順に ROB，RS，out-of-order 実行系の ALU，in-order 実行系，それ以外のモジュールの合計を示す．FrExec では out-of-order 実行系の ALU による消費エネルギーが削減される一方で,in-order 実行系による消費エネルギーが増加する. これらの結果，演算器に関する部分の消費エネルギーを比べると,FrExec は Base よりも 3%消費エネルギーを増加させている．プロセッサ・コア全体として見た場合は RS や ROB などの消費エネルギー削減の効果により，Base と比べて 17%と大きく消費エネルギーを減らしている．

5.6　エネルギー効率の評価

ここまでは各モデルの性能と消費エネルギーを個別に評価してきたが，本節ではこれらを合わせたものとして ED^2 積の評価を行う．図 14 に ED^2 積の評価結果を示す．同図は，縦軸に Base で正規化した ED^2 積を，横軸にベンチマーク名を示す同図では，数値が小さいほどエネルギー効率が良いことを示す．図 14 において Narrow は，ほとんどのベンチマークで ED^2 積が Base より悪化しており，平均で 23%，最大 で namd の場合に 73%悪化している．これは，Narrow では資源の縮小により消費エネルギーが削減されている一方，前述したように性能が大きく低下しているためである．これに対して FrExec では，ほぼすべてのベンチマークにおいて ED^2 積を改善しており，平均で 22%，最大 で gromacs の場合に 58%改善している． これは，前述したように，FrExec では in-order 実行系での命令の実行によって性能を向上させながら同時にエネルギーを削減しているためである．

6.　まとめ

近年では高性能化への要求から，スマートフォンやタブレット端末などの携帯機器においても out-of-order スーパスカラ・プロセッサが広く採用されている．しかし，out-of-order スーパスカラ・プロセッサの消費エネルギーは大きく，この点が大きな問題となっている．これに対し，我々は高性能と低消費エネルギーを両立することを目的として，in-order と out-of-order の 2 つの実行系をもつフロントエンド実行方式を提案してきた．本論文では，このフロントエンド実行方式の消費エネルギーについてモデル化し，シミュレーションによってそのエネルギー効率を評価した．評価の結果，フロントエンド実行方式は Cortex A-57

ⓒ 2013 Information Processing Society of Japan

をモデルとした通常のスーパスカラ・プロセッサと比較して消費エネルギーを平均で 17% 削減しながら IPC を 3%向上させられることを確認した．また，ED^2 積の評価については通常のスーパスカラ・プロセッサと比較して 22%改善することを確認した．

今後の課題として，in-order 実行系における実行命令数の増加があげられる．既存の方式では in-order 実行系で命令が実行できなかった場合，その命令の直近にあるコンシューマは全て in-order 実行系では実行不能となる．このような命令に対し，選択的に in-order 実行系をストールさせることによって依存が解消されるのを待ち，in-order 実行系で実行される命令を増やす方法を検討している．

謝辞

本研究の一部は，日本学術振興会 科学研究費補助金 若手研究 (A)（課題番号 24680005）による補助のもとで行われた．

参考文献

[1] ARM: The ARM Cortex-A9 Processors, *ARM White Paper* (2007).

[2] Bolaria, J.: Cortex-A57 Extends ARM's Reach:High-End 64-bit CPU Strives for Servers, *Microprocessor Report 5/11/3-1*, pp. 1–5 (2012).

[3] Chrysos, G. and Emer, J.: Memory Dependence Prediction Using Store Sets, *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 142–153 (1998).

[4] Intel: P6 Family of Processors Hardware Developer' s Manual, *Intel White Paper* (1998).

[5] Kessler, R.: The Alpha 21264 microprocessor, *IEEE Micro*, Vol. 19, No. 2, pp. 24–36 (1999).

[6] Krewell, K.: Cortex-A53 Is ARM's Next Little Thing:New CPU Core Brings 64 Bits to Big.Little, Mobile, *Microprocessor Report 5/11/3-2*, pp. 1–4 (2012).

[7] Li, S., Ahn, J. H., Brockman, J. B. and Jouppi, N. P.: McPAT 1.0: An Integrated Power, Area, and Timing Modeling Framework for Multicore Architecture, Technical report, HP Laboratories (2009).

[8] Rixner, S. et al.: Register Organization for Media Processing, *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 375–386 (2000).

[9] Singhal, R.: Inside Intel Next Generation Nehalem Microarchitecture, *Hot Chips*, Vol. 20 (2008).

[10] Sinharoy, B. et al.: IBM POWER7 Multicore Server Processor, *IBM J. Res. Dev.*, Vol. 55, No. 3, pp. 191–219 (online), DOI: 10.1147/JRD.2011.2127330 (2011).

[11] The Standard Performance Evaluation Corporation: *SPEC CPU2006 suite* `http://www.spec.org/cpu2006/`.

[12] Thoziyoor, S., Muralimanohar, N., Ahn, J. and Jouppi, N.: CACTI 5.1., Technical report, HP Laboratories (2008).

[13] Yeager, K.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, Vol. 16, No. 2, pp. 28–41 (1996).

[14] 塩谷亮太ほか：プロセッサ・シミュレータ「鬼斬弐」の設計と実装，先進的計算基盤システムシンポジウム SACSIS，pp. 120–121 (2009).

[15] 小西将人，五島正裕，中島康彦，森眞一郎，富田眞治：フロントエンド実行，情報処理学会研究報告 2004-ARC-158，pp. 13–17 (2004).

[16] 堀尾一生，塩谷亮太，五島正裕，坂井修一：ツインテール・アーキテクチャの評価，情報処理学会研究報告 2008-ARC-179，pp. 7–12 (2008).

[17] 堀尾一生，　平井遥，五島正裕，坂井修一：ツインテール・アーキテクチャ，先進的計算基盤システムシンポジウム SACSIS，pp. 265–274 (2007).