Vol.2015-ARC-216 No.3
2015/8/4

マルチバンク化と書込予測を用いた小面積レジスタファイルの提案

川島 弘晃[†1,a)]　佐々木 敬泰[†1]　深澤 祐樹[†1]　近藤 利夫[†1]

概要：
スーパースカラプロセッサの構成要素のうち，最も高コストなものの 1 つとしてレジスタファイルが挙げられる．特に，物理レジスタ方式のスーパースカラプロセッサではレジスタファイルの回路面積が非常に大きくなる．レジスタファイルの巨大化は，レジスタアクセス時間や消費電力の増加という問題を引き起こし，性能向上の障害となる．そこで，レジスタファイルの面積削減を目的として，マルチバンク化と書込予測を用いたレジスタファイルを提案する．本稿は，提案するレジスタファイルのためのマイクロアーキテクチャの構成について述べ，性能および消費電力の評価を行う．

1. はじめに

スーパースカラプロセッサの構成要素のうち，最も高コストなものの 1 つとしてレジスタファイルが挙げられる．特に，物理レジスタ方式 (Out-of-Order : OoO) のスーパースカラプロセッサでは命令レベル並列性（ILP）を活用するために，大容量かつ多ポートのレジスタファイルが必要とされる．また近年では，さらなる ILP 活用のために命令ウィンドウサイズや同時発行命令数を増加させる傾向にあり，SMT（Simultaneous Multi-Threading）などのマルチスレッディングを行うプロセッサでは，同時に実行されるスレッドのコンテキストを保持するため，スレッド数に応じた容量が必要となるなど，レジスタファイルのサイズ・ポート数は増加している．

レジスタファイルは多ポートの SRAM で構成されており，通常 1 命令あたり，2 つのリードポートと 1 つのライトポートが必要となる．よって，4 つの命令を同時に実行するスーパースカラプロセッサのレジスタファイルのポート数は合計 12 にもなる．SRAM の回路面積は，ポート数の 2 乗に比例するため，レジスタファイルは非常に大きなものとなり，SRAM の回路面積の増加は消費電力やアクセス時間の増加という問題も引き起こす．特に，FPGA や ASIC の設計フローでは，任意の多ポート SRAM を用意するのは困難である．そこで，これらの設計において 1R1W の SRAM の多重化によって多ポート SRAM を実現する手法 [1] が広く用いられている．本稿では 1R1W の SRAM の多重化を用いたレジスタファイルを想定している．多ポート SRAM の作成手法の問題点としてライトポートを多重化する際に，最新情報を持つバンクを記憶 (Most Recently Used : MRU) するフリップフロップや，最新のバンクからデータを引き出すためのセレクタといった追加のハードウェアが必要になることが挙げられ，ライトポートの多重化はコストが大きい．

そこで本稿では，レジスタファイルの面積を削減することを目的とし，マルチバンク化と書込予測を用いたレジスタファイル構成を提案する．マルチバンク化を行うことで例えば 8R4W の SRAM を 8R1W の 4 セットの SRAM に分割することができ，ライトポートの多重化を避けることができる．一方，マルチバンク化をすることで書込時にバンクコンフリクトが発生する可能性がある．そこで，本稿では同じサイクルでレジスタファイルに書込を行う可能性の高い命令を予測し，それらの命令の書込先を異なるバンクを割り当てることで，バンクコンフリクトを回避し，性能低下を抑えることを目指す．本稿では，1R1W の SRAM の多重化によって作成される多ポート SRAM の概要と，提案する書込予測機構について述べ，有効性を示すために性能評価を行う．

2. 1R1W の SRAM の多重化

一般的な FPGA や ASIC の設計フローにおいて利用されるメモリコンパイラは，3 ポート以上の SRAM の設計ができない．そのため，ポート数とエントリ数の組み合わせが異なる SRAM を全て手動で設計する必要があり非常に

[†1] 三重大学大学院工学研究科情報工学専攻
Graduate School of Information Engineering，Mie University
[a)] kawasima@arch.info.mie-u.ac.jp

時間がかかる．そこで 1R1W の SRAM の多重化を行い多ポート SRAM を短時間で設計することが出来る手法が用いられる．以下でリードポートの多重化，ライトポートの多重化手法について述べる．

- リードポートの多重化
  図 1 に，リードポートの多重化の例となるブロック図を示す．リードポートの多重化処理では，ライトポートへの入力信号は利用する 2 ポート SRAM で共有するように繋ぎ，リードポートはそれぞれで異なるように繋ぐことで多重化を行う．

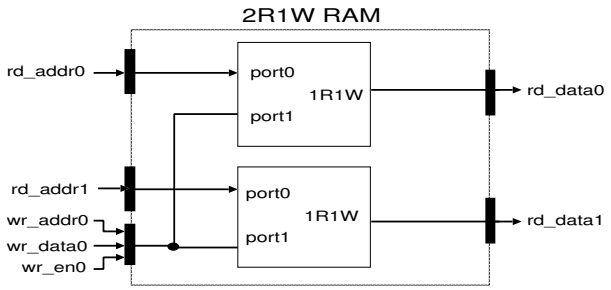

図 1 リードポートの多重化

- ライトポート多重化
  図 2 に，ライトポートの多重化の例として上記の 2R1W の SRAM を用いて 2R2W の SRAM を作成するブロック図を示す．ライトポートの多重化処理では，同容量の NR1W（N は自然数）を複数個用いる．また，各アドレスの最新の情報を持つバンクを識別するための MRU メモリ（図 2 中の ram_select_vector），および最新のデータを持つバンクからデータを引き出すためのセレクタが追加される．そのため，ライトポートの多重化はリードポートの多重化と比べてコストが高い．

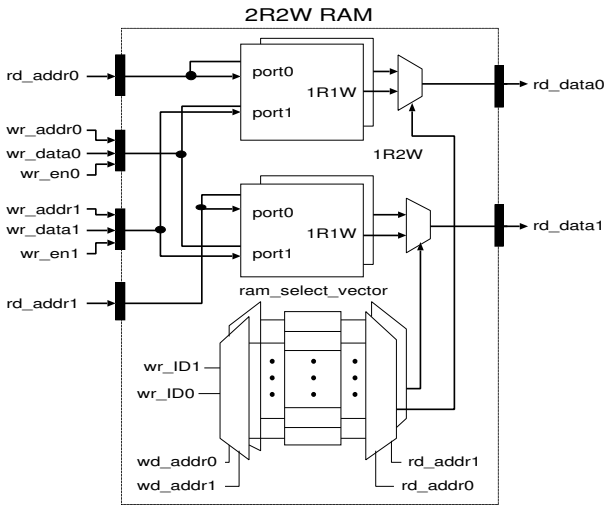

図 2 ライトポートの多重化

- バンク化した 2R2W-SRAM
  前述の通り，1R1W の SRAM を多重化して作成した多ポート SRAM ではライトポートの多重化に非常に大きなコストがかかる．そこで，本稿ではライトポートの多重化を行わず，マルチバンク化を行い，ライトポートの多重化時に発生するコストを削減する．図 3 にマルチバンク化を行った SRAM の例となるブロック図を示す．マルチバンク化を行う利点として，追加のハードウェアが不要になり，各 SRAM のエントリ数を削減出来る点が挙げられる．例えば 8R4W-80 エントリのレジスタファイルを作成する場合，初めに 8R1W-80 エントリの SRAM を作成する．その後作成した 8R1W-80 エントリの SRAM を 4 セット用いてライトポートを多重化することで，8R4W-80 エントリの SRAM が完成する．一方，4 バンクに分割すると 8R1W-20 エントリの SRAM を 4 セット作成するだけで 8R4W-80 エントリの SRAM が完成する．よってこの例では，使用する SRAM の個数は同じだが，トータルのメモリ容量を 4 分の 1 に削減することができ，追加のハードウェアも必要ない．しかし，バンクコンフリクトの発生が性能低下の要因となる．

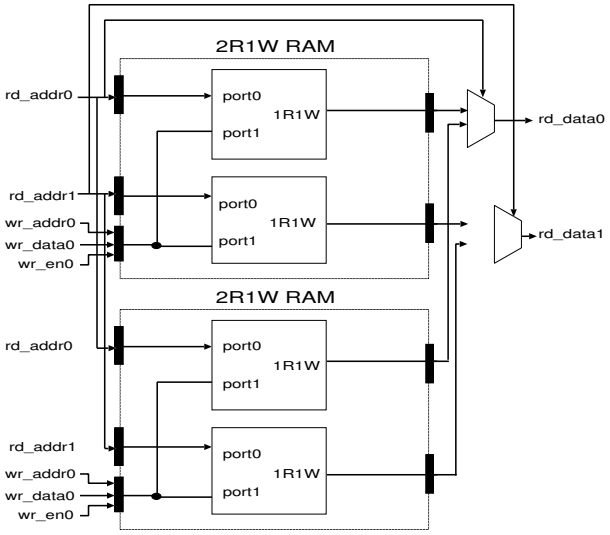

図 3 マルチバンク化を用いた SRAM の例

3. マイクロアーキテクチャ

本節では，提案するレジスタファイル構成を示すのに先立ち，まず想定しているマイクロアーキテクチャについて述べる．

3.1 従来型プロセッサのマイクロアーキテクチャ

本研究では，レジスタファイルのポート数の多い物理レジスタベースの OoO スーパースカラプロセッサを想定している．そこで，FabScalar[2] と呼ばれるフェッチ幅や，データ幅などのパラメータを与えることにより，任意の構成のスーパースカラプロセッサを自動生成できるツール

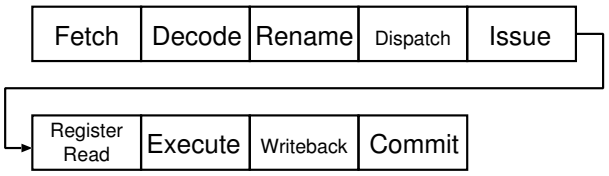

図 4　パイプライン構成

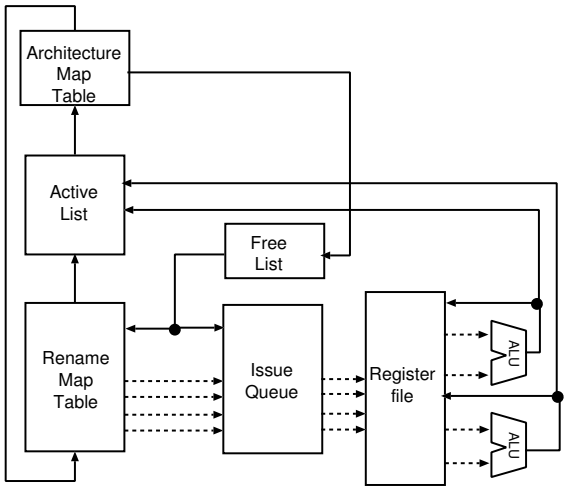

図 5　リネームステージ以降のブロック図

セットを用いて提案するレジスタファイル構成の実装を行うことにした．

図 4 は FabScalar のパイプライン構成を示している．本研究では Rename ステージ以降のマイクロアーキテクチャを変更の対象としており，図 5 はリネームステージ以降のブロック図を示している．以下，このプロセッサの動作について説明する．

Fetch，Decode ステージを経て Rename ステージに到達した命令のソースレジスタ番号は Rename Map Table（RMT）を参照し物理レジスタ番号に変換され，同時にその命令のデスティネーションレジスタとして新しい物理レジスタ番号が Free List から割り当てられる．Free List から割り当てられた物理レジスタ番号はデスティネーションレジスタ番号に対応する RMT に登録される．次に命令のコミットを命令順通りに行うため，Dispatch ステージでは全ての命令が Active List に命令順通り登録される．このとき，命令のデスティネーションレジスタ番号と，現在割り当てられている物理レジスタ番号が Active List に登録される．命令がコミットされる際には，その命令のデスティネーションレジスタ番号に対応した Architecture Map Table（AMT）のエントリに割り当てられた新しい物理レジスタ番号を登録し，その命令のデスティネーションレジスタ番号に以前に割り当てられていた物理レジスタ番号を解放し Free List に戻す．各命令は Rename ステージで得られたソース・デスティネーション物理レジスタ番号とともに，Dispatch ステージで Issue Queue に登録されソースオペランドがそろい次第発行（Issue）される．発行された命令は，レジスタ読み出し（Register Read）が行われ，ALU にて演算が実行（Execute）される．演算結果は Writeback ステージでレジスタに対して書き込まれる．分岐予測ミスや例外が発生した場合は，AMT から RMT にアーキテクチャステートをコピーすることで復帰することが出来る．

3.2　マイクロアーキテクチャの拡張と問題点

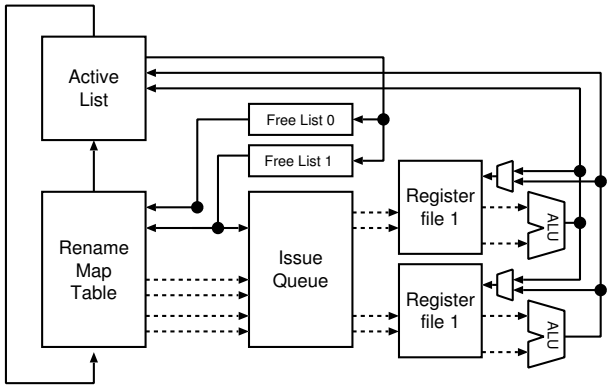

図 6　提案するレジスタファイル構成のブロック図

図 6 に提案するレジスタファイル構成を実現するための拡張を行った場合のブロック図を示す．図 6 は，従来のレジスタファイルを 2 バンクに分割した例である．なお，パイプライン構成は図 4 と同様である．

本拡張では，Rename ステージにおいて，各デスティネーションレジスタに対し割り当てる物理レジスタ番号を管理するフリーリストをレジスタファイルのバンク数分割する．従来型のプロセッサの場合，フリーリストのエントリ数はレジスタファイルのエントリ数と RMT のエントリ数の差である．しかし，本拡張において分割されたそれぞれのフリーリストのエントリ数を従来と同じように決めてまうと，フリーリストのエントリが破壊されてしまう．これを図 7 を用いて説明する．簡単のため，レジスタファイルの総エントリ数は 16，バンク数は 2，RMT のエントリ数は 4，偶数番号の物理レジスタ番号を Free List 0 が，奇数番号の物理レジスタ番号を Free List 1 が管理すると仮定する．

従来型のプロセッサのフリーリストのエントリ数の考え方を適用するとフリーリストの総エントリ数は $16-4=12$ エントリとなり，2 つに分割するのでさらに $12/2=6$ となるのでそれぞれのフリーリストのエントリ数は 6 となる．図 7 は現在の RMT と AMT の状態を示しており，デスティネーションレジスタ R1 を使用していた命令がコミットされ AMT の該当するエントリを更新し，古い物理レジスタ番号の P1 を解放し，Free List 1 に戻そうとしている場面である．

このときの Free List 1 の Head Pointer と Tail Pointer は同じ地点を指し示しており，この状態で P1 を書き込む

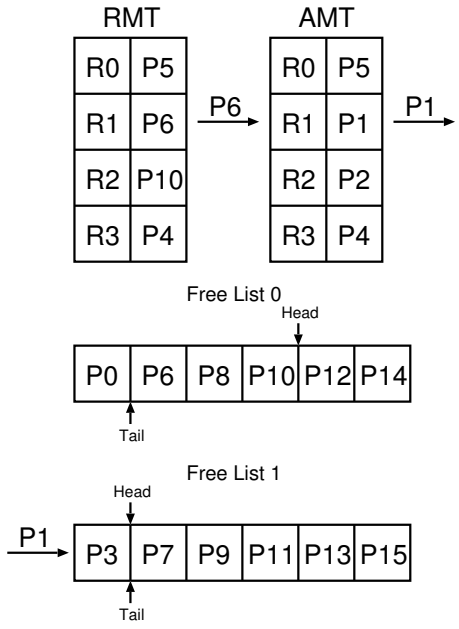

図 7　Free List の問題点

と Tail Pointer が Head Pointer を追い越してしまい P7 のエントリが破壊される．これは Free List 0 が管理する物理レジスタ番号によって Free List 1 が管理する物理レジスタ番号が解放されるため，Free List 1 のエントリ数が不足してしまうからである．これを防ぐためにフリーリストのエントリ数をバンク化したレジスタファイルのエントリ数と同じ数にまで拡張する．

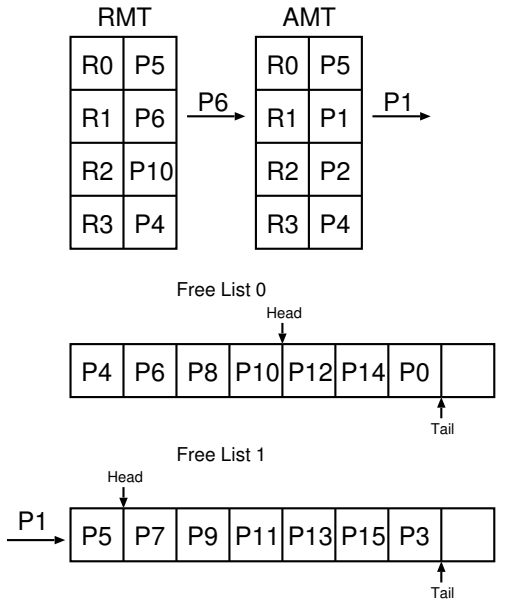

図 8　Free List の拡張

図 8 はフリーリストのエントリ数をバンク化したレジスタファイルのエントリ数と同じ 8 に拡張したフリーリストを示している．P1 の物理レジスタ番号を書き込むエントリが残されているため他のエントリのデータが破壊されることはなくなる．また，この拡張により Tail Pointer の初期位置が変化する．

従来型のプロセッサの場合，分岐予測ミスやオーバーフローなどの例外が発生した場合，アーキテクチャステートを復帰させるために AMT の内容を RMT にコピーし，同時にフリーリストの Head Pointer が指す位置を Tail Pointer が指す位置に合わせるだけでフリーリストの回復ができる．しかし，本実装では Tail Pointer の初期位置が変化したことにより，Head Pointer が戻るべき位置が特定できずアーキテクチャステートの復帰ができなくなってしまう．図 8 の AMT の内容を RMT にコピーした場合，HeadPointer が戻る位置を Tail Pointer が現在指している位置に合わせたとすると，次に読み出すデータが不定値となり，フリーリストの内容が破壊されてしまう．

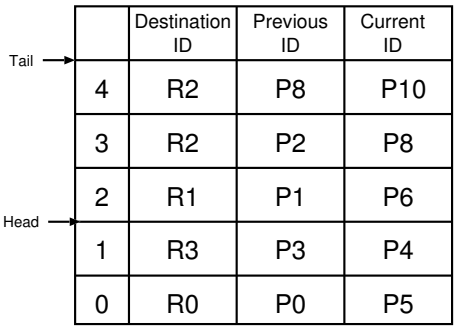

図 9　Active List

そこで，AMT を用いないアーキテクチャステートの復帰方法を採用する．図 9 のように Active List に命令のデスティネーションレジスタ番号，以前に割り当てられていた物理レジスタ番号，現在割り当てられている物理レジスタ番号を登録することにより，アーキテクチャステートを復帰させることが出来る．命令をコミットするときは Active List の Head Pointer から順番にデータを取り出し，以前割り当てられていた物理レジスタ番号を解放しフリーリストに戻す．アーキテクチャステートを復帰する場合は Tail Pointer からデータを読み出し，現在割り当てられている物理レジスタ番号をフリーリストに戻し Head Pointer を戻す．加えて，以前割り当てられていた物理レジスタ番号を RMT に登録する．

4. 書込先バンク予測機構の提案

マルチバンク化を用いることでバンクコンフリクトが発生し，性能低下の原因となる．そこで，各命令の書込先をそれぞれ異なるバンクに対して割り当てバンクコンフリクトの発生を防ぐ方法を提案する．提案する予測機構は同じサイクルでレジスタファイルに書込を行う可能性の高い命令をリネームステージ内で予測する．レジスタファイルに同時に書込を行う可能性の高い命令を検出するためには，命令の発効順序を予測する必要がある．命令の発効順序は

先行命令の実行順序や依存関係によって決まるため，高精度な予測をするためにはそれらの情報を保存しなければいけないが，そのような機構を実装するとハードウェア量が大幅に増大する危険性がある．そこで，本研究では，LW のような実行遅延サイクルが不確定で予測難度が高い命令は予測の対象外とし，ADD や SUB などの Simple 命令と JALR などのデスティネーションを持つ一部の分岐命令のみを予測の対象とすることで回路の複雑化を避ける．

予測機構を実装するプロセッサの前提として，毎サイクル必ず 4 命令同時にリネーミングが出来るスーパースカラプロセッサであり，レジスタファイルは 4 バンクに分割されているものとする．

Algorithm 1 予測機構のアルゴリズム

```
begin:initialization
last_assigned_bank ← −1
end
begin:Detect dependency
for i = 0 to DISPATCH_WIDTH − 1 do
  for j = 0 to i do
    if detectdependency then
      D ← i
      break
    end if
  end for
end for
end
begin:Assign physical register
for i = 0 to DISPATCH_WIDTH − 1 do
  last_assigned_bank ← last_assigned_bank + 1
  if last_assigned_bank ≥ 4 then
    last_assigned_bank ← 0
  end if
  phyDest[i] ← free_phys[last_assigned_bank]
  popfree_phys[last_assigned_bank]
  if i == D then
    depend_assigned_bank ← last_assigned_bank
  end if
end for
begin:Propagate
if detectdependency then
  last_assigned_bank ← depend_assigned_bank
end if
end
```

Algorithm1 は提案する予測機構のアルゴリズムを示しており，同図を用いて提案する予測機構の動作概要を説明する．

- Detect dependency
  リネームステージに到達した 4 命令の中から真の依存関係をもつ命令を検出する．
- Assign physical register
  保存したバンク番号+1 番のフリーリストから順番にデスティネーションレジスタに物理レジスタ番号を割り当てる．
- Propagate
  真の依存を検出した場合には，真の依存を持つ命令に割り当てたバンク番号を保存する．

0	lui	gp, 0x4b
1	addiu	gp, gp, 30336
2	lw	a0, -32744(gp)
3	lw	a1, 0(sp)
4	addiu	a2, sp, 4
5	li	at, -8
6	and	sp, sp, at
7	addiu	sp, sp, -32

図 10 命令列

次に，図 10 を用いて提案する予測機構の動作を具体的に説明する．保存されているバンク番号は 3 と仮定し，初めに 0〜3 番までの命令列のリネームを行う．保存されているバンク番号は 3 なので，0〜3 番の命令のデスティネーションには 0〜3 番のバンクを割り当て，命令間に真の依存関係がないかを調べる．1 番の命令は 0 番の命令と真の依存関係にあるため，1 番の命令に割り当てたバンク番号（1 番）を保存しておき次の 4 命令のリネームを行う．次の 4 命令では，先ほど保存したバンク番号+1 番から割り当てを開始する．よって，4 番の命令は 2 番のバンク，5 番の命令は 3 番のバンク，6 番の命令は 0 番のバンク，7 番の命令は 1 番のバンクがそれぞれ割り当てられる．6 番と 7 番の命令ははそれぞれ真の依存関係を持つが，保存するバンク番号は初めに検出した真の依存関係を持つ 6 番の命令に割り当てたバンク番号（0 番）のみである．以降，同様に予測とバンク番号の割り当てを行う．

5. 評価

5.1 評価環境

提案するレジスタファイル構成の性能を調べるため，FabScalar を用いた RTL のサイクルレベルシミュレーションにより評価を行う．

評価プログラムは，SPEC CPU2000 の整数ベンチマークを用いて，プログラムの最初から 1 億命令を実行し，評価を行った．なお表 1 に評価におけるプロセッサの構成を示す．

5.2 性能評価

図 11 に評価に用いたプログラム毎の実行サイクル数を示す．図中の Ideal は従来の理想的なマルチポートレジスタファイルを用いたプロセッサを表し，Banked(without prediction) はレジスタファイルのマルチバンク化のみを実

表 1　Processor configuration

Data path width	32 bit
Fetch, Dispatch, Issue, Commit width	4
Branch prediction	bimodal 16Ktable
BTB	1024sets
Issue Queue size	32
L1 I-Cache	32KB, 16B/line, 4way 1cycle latency
L1 D-Cache	32KB, 16B/line, 4way 2cycle latency

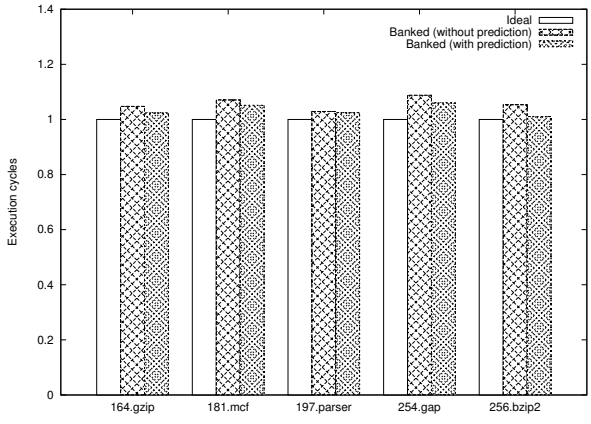

図 11　プログラム毎の実行サイクル数

装した場合を，Banked(with prediction) はマルチバンク化に加えて書込予測も実装した場合を表している．なお実行サイクル数は，各プログラムの Ideal の実行サイクル数を 1 として正規化している．評価に用いたレジスタファイルは 96 エントリのレジスタであり，マルチバンク化した場合は 4 バンクに分割を行った．よってマルチバンク化されたレジスタファイルのそれぞれのバンクのエントリ数は 24 である．

図 11 の結果より，本稿で提案する書込予測を用いたマルチバンク化レジスタファイルは書込予測を用いないマルチバンク化レジスタファイルより実行サイクル数を最大 4.1%，平均 2.2%短縮させることが出来た．書込予測によって，同時にレジスタファイルに書込を行う命令群の書込先がそれぞれ異なるバンクに割り当てることができたため，実行サイクル数が短縮したと推測できる．

5.3　ライトバッファの挿入による性能への影響

バンクコンフリクト発生によるペナルティを抑えるため，書込先のバンクが重複した場合には N 段のバッファ（N は自然数）にデータを保存しておき，パイプラインストールを回避する手法が考えられる．図 12 に本稿で提案するレジスタファイル構成に 1〜2 段のバッファを挿入した場合の各プログラムの実行サイクル数を示す．なお実行サイクル数は，各プログラムの Ideal の実行サイクル数を 1 として正規化している．

図 12 より 1 段もしくは 2 段のバッファを用いることで，

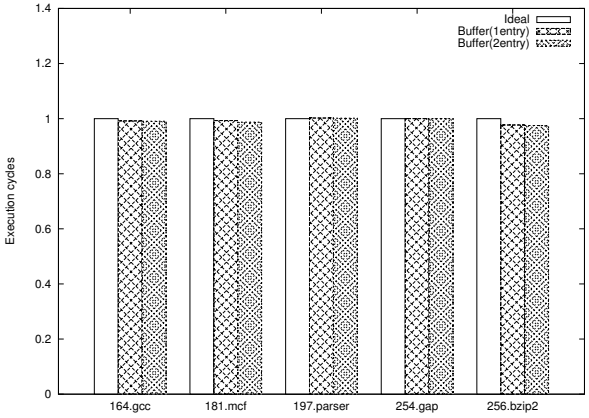

図 12　バッファ挿入後の実行サイクル数

バンクコンフリクトの発生をほとんど回避することができることが分かる．しかし，バッファ自体の回路面積やバッファからフォワーディングするためのパスの追加などハードウェアコストが増大してしまう危険がある．図 12 のプログラムの中にはバッファを挿入したレジスタファイル構成の方が従来の理想的なマルチポートレジスタファイル構成より実行サイクル数がわずかに少なくなっている場合があるが，これは現在の FabScalar の命令発効ステージにおいて依存があるため発行できずにキューに残っている命令 A と新しくキューに登録された命令 B が同時に発行可能になったとき，場合によっては B の命令の方が先に発行されてしまうなど，命令発効部分の最適化が進んでいないため，物理レジスタ番号の割り当てが変化した場合に一部命令の発効順序が変化したからではないかと考えられる．

5.4　面積評価

本稿で提案するバンク化されたレジスタファイル構成の面積と 1R1W の SRAM の多重化を用いて作成した理想的なマルチポートレジスタファイルの面積を図 2 に示す．評価には 8R4W-96 エントリのレジスタファイルを 4 バンクにマルチバンク化した場合と，マルチバンク化しない場合を用いて，Rohm CMOS 0.18μm プロセスのデータシートを元に面積を算出した．

表 2　面積評価

Ideal	2,764,611 μm^2
Banked	619,261 μm^2

4 バンクに分割を行った場合，各バンクのエントリ数は 24 エントリとなり，面積は 4 分の 1 になる．加えてライトポート多重化時に発生する追加のハードウェアを削減できるので，本稿で提案するレジスタファイル構成は多重化を用いたレジスタファイルと比べて，約 77.6%削減することが出来た．

5.5 電力評価

表 3 に従来構成のプロセッサ全体の NAND 換算のゲート数と，本稿で提案するレジスタファイル構成を実装したプロセッサ全体の NAND 換算のゲート数を示す．ただし RAM 部分は含まない．

表 3 プロセッサ全体の NAND 換算のゲート数

Conventional	152,236
Proposed	153,505

表 3 より，ゲート数の増加は従来構成のプロセッサと比較して約 0.83%であり，提案するレジスタファイル構成を実現するために必要な回路の消費電力は非常に少ないと考えられる．

6. 関連研究

従来より，レジスタファイルの大容量・多ポート化による回路面積・消費電力・アクセス時間の増加の問題への対処を目的として，様々な研究が行われている．

例えば，マルチバンク化 [3], [4], [5] や，レジスタ・キャッシュ [6], [7], [8]，クラスタ型マイクロアーキテクチャ [9], [10] などがよく知られる手法である．

文献 [3] は，ライトポートのみをマルチバンク化した例であり，バンクコンフリクトの発生を減らす工夫として，リネーミング時に直ちに物理レジスタ番号を割り当てず，依存性タグを割り当てておき，ライトバック時にバンクコンフリクトが発生しないように物理レジスタの割り当てを行う．このため，依存性タグと物理レジスタの対応をとるテーブルが必要となる．

文献 [4] の方式ではリード，ライトポートともにマルチバンク化が可能であるが物理レジスタへのアクセスにアクセス・キューが追加要素として必要である．

レジスタ・キャッシュは，ミス時のペナルティが大きいことが問題であったが，文献 [6] では物理レジスタ番号の割り当て順に着目しヒット率を向上させている．文献 [7] では，ミスを仮定したパイプライン構成を取ることで，IPC の低下を抑えつつ，面積，消費電力を削減することに成功している．しかし，アクセス時間の短縮は目的としていないので，レジスタアクセスには複数サイクルかかる．また，文献 [8] ではレジスタ・キャッシュとマルチバンク化を用いた手法を提案しているが，ライト・バッファの面積に占める割合が高すぎることが問題となっている．

本稿で提案するレジスタファイル構成は 1R1W の SRAM の多重化によって作成されるレジスタファイルを対象としている点で先行研究とは異なる．書込予測器は複雑な回路を必要とせず，バンクコンフリクトの発生を抑制することができ，性能低下を抑えることが可能である．マルチバンク化と書込予測を組み合わせることで，レジスタファイルの回路面積を大きく削減することができる．

7. まとめと今後の展望

本稿では，レジスタファイルの小面積化を目的として，マルチバンク化と書込予測を用いたレジスタファイル構成を提案した．書込予測は同時にレジスタファイルに書込を行う可能性の高い命令群を検出し，それぞれの命令の書込先を異なるバンクに振り分けることでバンクコンフリクトの発生を防ぎ，性能低下を抑える働きがある．

提案したレジスタファイル構成を評価した結果，書込予測を用いないマルチバンクレジスタファイルと比べて実行サイクル数を最大 4.1%，平均 2.2%短縮し，1R1W の SRAM の多重化を用いた理想的なマルチポートレジスタファイルと比べて面積を約 77.6%削減できた．今後は，書込予測のアルゴリズムをさらに発展させるなど，マイクロアーキテクチャ方式を工夫することでさらに性能低下を抑える方法を検討していく予定である．

謝辞 本研究は JSPS 科研費 24700047，15K00074 の助成を受けたものであり，東京大学大規模集積システム設計教育研究センターを通し，シノプシス株式会社，日本ケイデンス株式会社の協力で行われたものである．

参考文献

[1] D. Brandon, et. al.: *FPGA Modeling of Diverse Superscalar Processors*, (2011.11.02).
[2] N. K. Choudhary,et. al.: *FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template*, ISCA-38, pp. 11-22, June 2011.
[3] Il Park, Michael D. Powell, and T. N. Vijaykumar: *Reducing Register Ports for Higher Speed and Lower Energy*, Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35), 2002.
[4] Hironaka, T. Maeda, M., Tanigawa, K., Sueyoshi, T., Aoyama, K., Koide, T., Mattausch, H. and Saito, T.: *Superscalar processor with multi-bank register file*, Innovative Architecture for Future Generation High Performance Processors and Systems, 2005.
[5] J.-L. Cruz, et al.,: *Multiple-Banked Register File Architectures*, In Proc. the 30th ISCA, pp.62-71, June 2003.
[6] 小林 良太郎, 堀部 大介, 島田 俊夫: リネーミングされるレジスタ番号の整列によるレジスタ・キャッシュの高精度化手法, 情報処理学会研究報告, 2006.
[7] Shioya, R., Horio, K., Goshima, M. and Sakai, S.: *Register Cache System Not for Latency Reduction Purpose*, 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 301-312, 2010.
[8] 山田淳二, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: レジスタ・キャッシュ・システムにおけるレジスタ・ファイルのマルチバンク化, 情報処理学会研究報告, 2014.
[9] G.S. Shohi, et al.,: *Multi-scalar processors*, In Proc. the 22th ISCA, 1995.
[10] R.E. Kessler,: *The Alpha 21264 Microprocessor*, IEEE Micro, Vol.19, No.2, pp.24-36, Apr. 1999.