Vol.2013-ARC-206 No.5
2013/7/31

もしILPプロセッサのレジスタファイルが分散キーバリューストアになったら

入江 英嗣[1]　山中 崇弘[1]　佐保田 誠[1]　吉見 真聡[1]　吉永 努[1]

概要：プロセッサの性能向上の基本戦略は，2000 年頃からはマルチコア構成の拡張が主流となり，トランジスタ資源をコア数の増加に利用することで，効率的に TLP 性能を向上させてきた．しかしこのアプローチも，TLP の収穫逓減やダークシリコンの増加など，継続的な成長には限界が指摘されている．この限界を打ち破り，高性能なメニーコアプロセッサを実現するための課題の一つとして，一つ一つのコアの実行性能と電力効率の双方を高める実行アーキテクチャの開発が挙げられる．ここではピーク ILP 実行幅よりも，コンスタントな高性能と高効率が求められる．3 次元実装技術に代表されるように，パッケージ内トランジスタ数の増加は堅調であり，容量を用いて処理レイテンシと電力を削減するアーキテクチャへの転換が今後のプロセッサ成長の鍵と考えられる．本論文では，ライト・ワンス・マナーに基づいた大きな論理レジスタ空間を導入することで，レジスタリネーミング処理を取り除き，更にはバックエンド幅の増加なく実行性能を増加させる STRAIHGT アーキテクチャを提案し，実現のための技術と性能の見積もりを述べる．STRAIGHT アーキテクチャに見立てたパラメタを用いた初期評価では，同じワークロードに対するエネルギー消費を 12%削減しながら，同時に約 30%の IPC 向上が得られ，性能/パワー比を改善する新しい実行方式として有効であることが示された．

1. はじめに

情報機器の中核部品であるマイクロプロセッサは，半導体技術の堅調な進展を背景に指数的な性能向上を続けており，情報化社会を支えている．半導体微細化と共にパッケージ内で利用可能なトランジスタ数は増加を続け，アーキテクチャ技術はこのトランジスタ増をチップ全体の性能向上に結びつけてきた．

高性能 VLSI では，2000 年頃から配線遅延，電力・熱の問題が顕在化し [1]，トランジスタ資源によってパイプライン段数や発行・実行幅を拡張し，各種投機実行によってパイプラインスロットを埋めるそれまでの成長アプローチは消費電力の制約を受けることとなった．更に，パイプラインの幅を確保しても，並列に実行する命令が存在しないという，ILP(Instruction Level Parallelism) の収穫逓減の問題が設計の転換を促すこととなった．以後，増加するトランジスタ資源はプロセッサ実行部分の強化よりも，コア数やオンチップネットワークおよびメモリ帯域の増加にあてられるようになり，これらがプロセッサ性能向上を牽引している [2][3].

[1] 電気通信大学情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications

しかし，コア数増による成長戦略は，コア数の増加とともに，かつての ILP プロセッサ同様に TLP(Thread Level Parallelism) 収穫逓減と性能向上の限界が指摘されるようになってきた [4]．ワークロードが充分に並列化されていても，スレッド同期オーバヘッドやコア毎のスレッド長の偏りがあるため，コア数が増加するほどシングルスレッド実行能力が全体性能を頭打ちしてしまう．さらに，パッケージあたりの電力や熱の制限から，シリコン上に製造可能なトランジスタ数を同時には駆動できないという，いわゆるダークシリコン問題 [5] が，コア数増加アプローチの限界を早めている．この限界を越えるためには，メニーコアプロセッサを構成するコア一つ一つのアーキテクチャを転換しなければならない．従来よりも高性能なシングルスレッド実行能力を持ち，かつ同じ処理をより少ない消費電力で実行できる新しいアーキテクチャが求められている．

我々は，消費電力削減とシングルスレッド能力向上を同時に達成するアーキテクチャとして，新しいレジスタ管理方式を持つ STRAIGHT アーキテクチャを構想し，将来の実現可能な高性能メニーコアプロセッサを構成することを目指している [6]．STRAIGHT アーキテクチャはライト・ワンス・マナー制約に従う広大な論理レジスタ空間を持ち，従来のアウト・オブ・オーダ・プロセッサにおいて主要な電力オーバヘッドであったレジスタ・リネーミングとフ

リーレジスタ管理を必要としない．また，このライト・ワンス・マナーは，従来困難であった命令ウィンドウサイズやフロントエンド幅の軽量な拡張を可能とし，シングルスレッド性能を向上させる．さらに，このレジスタファイルに近年の分散キー・バリュー・ストア技術を適用することにより，効率的にハードウェアを構成することができる．

本論文では，これからのトランジスタ数の増加をプロセッサ成長に結びつけるための STRAIGHT アーキテクチャの設計について述べ，アーキテクチャ仕様を提案する．また，スーパスカラシミュレータおよび電力シミュレータを用いて，シングルスレッド性能，回路規模，消費電力について初期評価を行う．以降，本論文は次のように構成される．第 2 節では関連研究として，近年の TLP 収穫逓減に関する議論と，スーパスカラ以降の ILP アーキテクチャの研究を紹介する．第 3 節では STRAIGHT プロセッサの設計について議論し，第 4 節ではアーキテクチャ仕様を提案する．第 5 節では評価環境について述べる．第 6 節では得られた STRAIGHT プロセッサの性能評価見積もりを示す．第 7 節では更なる最適化のための技術を議論し，第 8 節でまとめを述べる．

2. 関連研究

マルチコア・プロセッサ設計の参照ベンチマークとして用いられている PARSEC について，Bhadauria ら [4] は実機のパフォーマンスカウンタによる傾向解析を，様々な規模の x86 プロセッサや Niagara を用いて行なっている．スレッド数の増加と性能向上の相関を見た評価では，PARSEC 中のいくつかのアプリケーションではスレッド数を増やしても性能は向上せず，さらに 8 スレッドを超えると多くのプログラムで性能が飽和するという結果が示されている．これは，十分な TLP があっても，逐次実行部分，同期オーバヘッド，負荷ばらつきの影響がスレッド数と共に大きくなるためである．また，PARSEC ワークロードでは計算能力の影響が相対的に多く，バンド幅やキャッシュ容量の影響を上回ったと報告されている．これらの評価結果ではシングルスレッド計算力の重要性が示唆されているが，一方で，現行のアウト・オブ・オーダ・アーキテクチャでは，ILP 性能向上のための資源投入はコストが大きく，ILP 性能に資源をかけるよりも SMT やチップ間通信に資源を使う方が効果的であると結論づけている．

さらに近年の評価として，Esmaeilzadeh ら [5] はデバイスの性能予測と並列ベンチマークを用いて，8nm 世代までのチップ性能予測を行っている．現行のアーキテクチャの性能/パワー比傾向や性能/エリア比傾向を外挿した性能見積もりが行われているが，彼らは，規模の異なる CPU や GPU コア，ヘテロ構成やダイナミック構成を含めて，マルチコア戦略のみではムーア則に添った成長が維持できないと予測している．これは，並列性の不足あるいは電力の不足のために電源を ON にできない「ダークシリコン」が生じ，ムーア則によって増加する資源の活用ができなくなるためである．22nm 世代ではプロセッサチップの 21%の領域がダークシリコンとなり，8nm 世代ではこの領域はチップの過半数を占めると見積もられている．彼らは，今後性能向上を維持するためには，各コアの実行方式において従来の性能/パワー比をブレイクスルーするアーキテクチャが必要と主張している．

このようにマルチコア設計の課題は，当初の (i) ワークロードが並列化されていない，という課題から，(ii) 並列化されていてもスケールしない，(iii) スケールしても電力が足りない，と年々厳しくなっている．周波数向上によるシングルスレッド能力向上は，消費電力の観点から有効でないため，課題 (ii) に対しては IPC(Instructions Per Cycle)，課題 (iii) に対しては IPC/電力比をそれぞれ向上させるアーキテクチャが必要となる．

シングルスレッド能力の加速に関して，Ipek ら [7] の Core Fusion や Boyer ら [8] の Core Federation では，シングルスレッド実行時に複数のコアを融合させて，大きなコアとして動作させ，シングルスレッド実行を加速するアーキテクチャを提案している．しかし，これらの手法は小さいコアを融合させて，通常規模のコアの性能に近づけるものであり，シングルスレッド実行について，最適なスーパスカラプロセッサよりも高い性能や電力効率が得られる手法ではない．

スーパスカラよりも高い IPC を得るためのアーキテクチャは，1990 年代に多くの研究が行われている．IPC を発行・実行幅の拡張によって増加させようとすると，スーパスカラ・アーキテクチャは比較器やパスが爆発的に増加してしまう課題を有している．この問題に対して，クラスタ化と局所性を利用することで，クリティカルパス長の増加を防ぎつつ，発行・実行幅を増加させる技術が提案されている [9][10]．またフロントエンド幅についても，投機スレッド実行を利用して，クリティカルパス長を伸ばさずに幅を増加させる技術が提案されている [11]．さらに，このようにして増やしたパイプラインスロットを埋めるための積極的な投機実行技術が多く提案された [12][13][14]．しかし，クラスタ化や積極的な投機実行によって広いパイプライン幅を稼働させるアプローチは，制御に必要な電力を増加させる一方で，ILP を抽出しやすいワークロード以外では，資源の多くが無駄になり，IPC/電力比を悪化させてしまう．

IPC/電力比を向上させる研究では，パワーゲーティングや動的サイズ切り替えによって，無駄な電力を削減するアプローチ [15][16] の他，実行に必要な電力を削減する研究が行なわれている．逆 Dualflow アーキテクチャ [17] では，レジスタリネーミングの結果をトレースキャッシュに保持し，ヒット時のレジスタリネーミングを省略する．この

アーキテクチャでは，ソースオペランドを “n 命令前の実行結果” という命令距離で指定する dualflow 形式を用い，リネーミング結果の再利用を可能としている．この方式では，実行パスによってソースを生成する命令との距離が変化するため，リネーム済トレースのバリエーションが増加し，トレースキャッシュに多くの容量を必要とする課題がある．これを低減する RTC[18] では，性能低下を 0.3%に抑えながら，リネーミング電力を 53%削減したと報告されている．

3. 新アーキテクチャの構想

前節に議論したように，プロセッサ成長を継続するためには IPC と IPC/電力比の両方を現在よりも向上させるアーキテクチャが必要であり，現存のアーキテクチャの組み合わせによるメニーコアプロセッサでは限界が指摘されている．一方，パッケージ内のトランジスタ数の増加は半導体 3 次元積層技術 [19] に代表されるように堅調であり，この資源を活用して将来のプロセッサ成長を持続可能とするアーキテクチャが必要とされている．

まず IPC に関して，発行・実行幅を増加させる技術は提案されているが，その幅を有効に使いきれるワークロードは少なく，逆に，使いきれるようなワークロードであれば，コード並列化によりマルチコア・プロセッサでより効率的に TLP 実行できる．求められる ILP 性能は，並列度の低いシーケンシャルな部分の低レイテンシ実行であり，コアあたりのピーク実行幅の拡大よりも，定常的に数命令を並列実行できる性能である．4way，6way といった通常規模のスーパスカラの発行・実行幅でも，機能ユニットの稼働率は低く，不足しているのは発行・実行幅よりも，命令ウィンドウサイズである．しかし，命令ウィンドウサイズの増加は物理レジスタ数を増加させ，リネーミングのための多ポート RAM である RMT の容量を増加させる．また，大きな命令ウィンドウの充填率を高めるためにはフロントエンド幅を増加させる必要があり，リネーミングをさらに複雑にする．

次に IPC/電力比に関して，消費電力制限の下で性能を高めるためには，無駄な消費電力を省くだけでなく，同じ命令をより少ない消費電力で処理できるアーキテクチャが必要である．命令処理のパイプライン中で，極力命令実行そのものの消費電力比率を高め，制御にかける消費電力を削減することが望ましい．このことから，イン・オーダ・アーキテクチャの選択もメニーコア構成の設計オプションに入っている．しかし，一般的なコード実行では浮動小数点演算やメモリ演算の依存においてアウト・オブ・オーダ実行の効果は大きく，IPC/電力比上も有利であることが多い．アウト・オブ・オーダ実行で複雑化する制御における消費電力では特にレジスタリネーミングが主要な負荷として知られている．以上のように，IPC と IPC/電力比双方の議論で鍵となっているのはレジスタ管理方法である．そこで我々は，トランジスタ増をレジスタ容量の増加にあて，代わりに制御を軽量化する STRAIGHT アーキテクチャを提案する．

(i) まず，充分に広い論理レジスタ空間をライトワンス・マナーで使用するようなプログラムコードを仮定する．すると，その実行には偽依存が発生せず，レジスタリネーミングの必要なく，コードに書かれた指定子に従って，そのままアウト・オブ・オーダ実行することができる．

(ii) 次に，充分に物理レジスタ数があれば，一つ一つのレジスタ解放タイミングをイーガーに管理する必要はなく，物理ディスティネーションレジスタ番号を命令のフェッチ順に割り振ることができる．レジスタリネーミングがなく，ディスティネーション番号がフェッチ順に定まるフロントエンド処理では，命令間の依存ループがなく，フェッチブロックをそのまま並列にデコード・ディスパッチすることができる．

(iii) さらに，管理が単純な大量のレジスタ容量と拡張の容易なフロントエンド幅は，命令ウィンドウサイズの拡張を容易とし，バックエンドパイプラインの稼働率を増加させる．また大きい論理レジスタ空間はレジスタプロモーションを進め，スピル減少や曖昧なメモリ依存の解消による IPC 向上に効果がある．

レジスタ大容量化を考慮する背景には，トランジスタ数の増加の他，大容量化しても適切なサブアレイ構成が取られていれば，アクセス電力は増加しないこと，リークや電源遮断に関する技術が進んでおり容量自体によって生じる電力は対策できることが挙げられる．一方で，レジスタリネーミング処理は，稼働率の高い多ポート RAM アクセスであり，一つの命令実行ごとに多くの消費電力を発生させる．また，レジスタリネーミングは主要なクリティカルループの一つである [10]．このように，STRAIGHT では，論旨・物理共にレジスタ数を増やすことにより，リネームを始めとする制御の消費電力を削減し，さらに，発行・実行幅を増やさずに IPC を向上させる設計を可能とする．次節では，このようなアプローチを実現するアーキテクチャ仕様を具体化する．

4. STRAIGHT アーキテクチャ

4.1 命令セットアーキテクチャ

STRAIGHT 命令は，レジスタがライトワンス・マナーに従う点を除けば，一般的な RISC 命令と同様の簡単なオペレーションを行い，演算，転送，制御，システムといった典型的な命令群を備える．処理は命令単位であり，バンドルは必要としない．典型的な 2 入力 1 出力の演算命令の例を**図 1** に示す．命令は固定長で，MSB 側からオペレーションコード，ソースオペランドレジスタ L，ソースオペランドレジスタ R の指定フィールドとなる．ソースオペラ

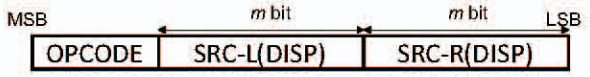

図 1　STRAIGHT 命令形式
Fig. 1　A Sample Instruction Format of STRAIGHT

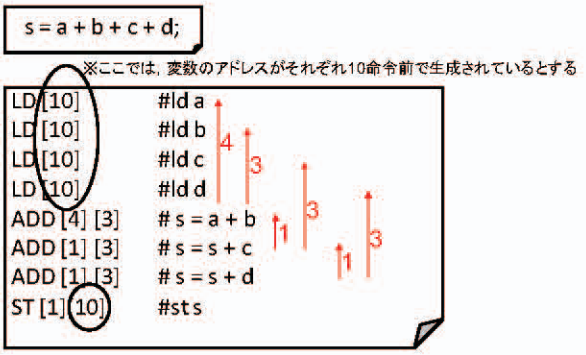

図 2　STRAIGHT コードサンプル
Fig. 2　A Sample Code of STRAIGHT

ンドは dualflow 形式のように命令位置の距離で指定される．例えばソースオペランドレジスタ L の値が 4 の場合，プログラム中で（正確には動的な有効フェッチ順で）4 つ前の命令の実行結果がオペランド L の値となる．ソースオペランド R は即値を取ることもでき，オペレーションコードによって区別する．ディスティネーションレジスタ番号は，命令のフェッチ順に定まるため，命令中で指定する必要はない．STRAIGHT 命令による簡単なアセンブリコードの例を**図 2** に示す．

ここで，オペランドフィールド長を m ビットとすると，STRAIGHT 命令は 2^m 個前までの命令の実行結果を参照できることになり，これが STRAIGHT アーキテクチャの汎用レジスタに相当する．この命令形式に従うことにより，ライトワンス性と，2^m 個後の命令実行終了後には参照されない，というレジスタ寿命が保証される．汎用レジスタの他，STRAIGHT アーキテクチャは上書き可能な特殊レジスタとして，スタックポインタとフレームポインタ，グローバルポインタを備えており，メモリを介すれば，ライトワンス性や寿命制約のない柔軟な参照記述が可能である．また，フェッチは通常の RISC アーキテクチャと同様に，上書き可能な PC レジスタによる．

4.2　コード生成

オペランドの指定法としてコンシューマからプロデューサへの命令距離を指定する方式では，制御の合流時に辿ってきたパスによって指定すべき距離が変化してしまう問題がある [17]．**図 3**(a) のように，分岐したパス A とパス B のそれぞれで更新される変数 x を合流後のパス C 中の命令 i が参照する場合，パス A 中の x の更新位置とパス B 中の x の更新位置が，それぞれ i から異なる距離に配置されていると，i のオペランド指示値が定まらなくなる．

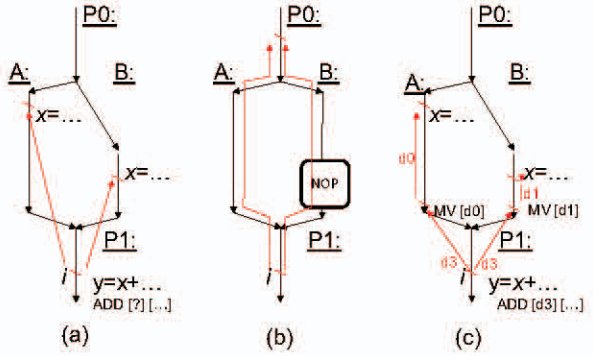

図 3　制御合流時の STRAIGHT コード生成手法
Fig. 3　Code Generation Method for Control Join of STRAIGHT

この問題を静的に解決するためには，パスの長さおよび変数更新の順番を調整するコンパイラアルゴリズムが必要である．まず，分岐前に定義した値を合流後に参照する場合について考える．このとき，分岐から合流までに取りうる全てのパスの長さが等しければ距離指定は整合する．そこで，パスの長さが静的に確定する場合は，全てのパスの長さが等しくなるように NOP 命令を挿入すれば良い (図 3(b))．関数呼び出しが入れ子になっている場合や，ループ回数が動的に定まる場合など，パスの長さが静的に解析できない場合は，分岐前の値が参照できなくなるため，合流後に引き続き参照するデータをスタックポインタを用いてメモリに待避する．このようなメモリアクセスの軽減が STRAIGHT コンパイラ最適化の課題である．

次に，分岐後の各パス中で更新される値を合流後に参照する場合を考えると図 3(c) に示すように，各パスについて，合流前の等距離の位置にレジスタ値移動命令をはさみ，位置調整を行えば良い．関数の呼び出し前後の参照も同様に，引数や返り値の距離を定めておくことにより，制御合流の問題を解決できる．

レジスタのライトワンス制約に関しては，多くのコンパイラでは，レジスタカラーリング前の中間言語は SSA(Static Single Assignment) 形式となっており，この出力を変形することで STRAIGHT コードを得ることができる．SSA 形式の中間言語をもつ LLVM では，制御の合流時に，通ってきたパスに従って値を選択する phi 命令が実装されている．この命令をレジスタ値移動命令に置き換えることで，図 3(c) の調整を記述することができる．**図 4** は，C 言語で記述した再帰フィボナッチ数のプログラムの LLVM 中間言語形式を STRAIGHT コードへ手動で変換した例である．

4.3　STRAIGHT マイクロアーキテクチャ

STRAIGHT の命令処理はフェッチ，デコード，ディスパッチ，イシュー，レジスタリード，エクセキューション，メモリ，レジスタライトの各ステージからなり，レジスタリネーミングがないことを除けば，ステージ構成はスー

```
1:      MAIN:   LD      gp              #a
2:              JAL     FIB             #dest(return address) <- PC+4
3:              MV      [5]             #get result value
4:              SYSTEM  PRINT           #output
5:              EXIT
6:
7:      FIB:    SPADDi  -24             #SUB $sp 24 $sp
8:              SLT     [3]     2       #set less than
9:              BNZ     [1]     L0:
10:
11:             ST      [5]     0($sp)  # save a -> 0(sp)
12:             ST      [5]     8($sp)  # save ra-> 8(sp)
13:             SUBi    [7]     1       #a - 1
14:             JAL     FIB
15:             ST      [5]     16($sp)
16:             LD      0($sp)          #restore a
17:             SUB     [-1]    2       #a - 2
18:             JAL     FIB
19:             LD      16($sp)
20:             ADD     [1]     [6]     #result value
21:             J       L1
22:
23:     L0:     MV      [5]             #result value
24:             NOP                     # v position
25:     L1:     SPADD   24
26:             LD      $sp(8)
27:             JR      [1]             #FIBOFFEST_V 5
```

図 4 フィボナッチ数計算の再帰コード
Fig. 4 STRAIGHT Code Sample of Recursive Fibonacci Algorithm

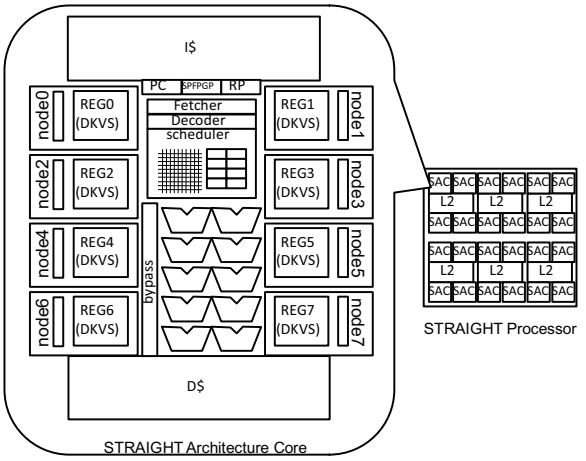

図 5 STRAIGHT アーキテクチャブロック図
Fig. 5 Block Diagram of STRAIGHT Architecture

パスカラプロセッサに類似している．**図 5** は STRAIGHT プロセッサのブロック図である．メニーコアプロセッサを構成する一つ一つの SAC(STRAIGHT Architecture Core) が並列実行パイプラインと分散レジスタファイルを備えている．

フロントエンドパイプラインでは，まず collapsing buffer[20] や trace cache[21] を用いて複数分岐を跨ぐ命令を同時にフェッチし，続くデコードステージでレジスタ番号を決定する．ここで RP(Register Pointer) と呼ぶ，命令 1 つごとにシーケンシャルに値の増加するアーキテクチャレジスタを導入する．各命令はデコード時の RP の値に従い，RP からオペランドフィールドで指定される距離を引くことでソースレジスタ番号を決定し，RP の値をディスティネーションレジスタ番号として用いる（**図 6**）．レジスタ番号は当該命令と RP のみによって決定するため命令同士に依存がなく，並列化が容易な処理となっている．RP の値は充分大きな数でターンアラウンドする．この数の設計については 4.4 で議論する．

命令はデコード後そのままディスパッチされる．レジスタ増による命令ウィンドウ幅の恩恵を得るためにはスケジューラサイズを大きくする必要がある．スケジューラでは，ディスパッチ後からコミットまでのインフライト命令を対象として管理する．命令情報を管理するペイロード RAM はサブアレイに分割することにより，スケジューラエントリ数に対してスケーラブルな実装が可能である．ウェイクアップ・セレクトロジックについて，発行幅は従来と同様の規模であるが，エントリ数が増加するため，従来よりも大規模なものとなる．このロジックはマトリックス方式 [22][23] を用いることにより軽量化する．

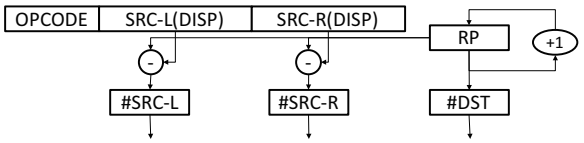

図 6 レジスタ番号の決定
Fig. 6 Determining the Register ID

命令が発行されると，ペイロード RAM の情報に従ってソースレジスタが読み出される．STRAIGHT のレジスタファイルは，レジスタ番号とレジスタ値を対とした大きなキー・バリュー・ストアを分散した実装となる．複数のテーブルで構成され，レジスタ番号にハッシュ演算を行うことで該当テーブルの ID を得て問い合わせる．簡単な実装として，存在しうる全てのレジスタ番号について，1 対 1 に対応する物理レジスタを備える場合，レジスタ番号がそのままテーブル ID およびエントリ ID となる（例えばレジスタ番号の上位 3 ビットがテーブル ID となり，残りの下位ビットが該当テーブル読み出しのインデックスとなる）．レジスタファイル全体では大容量となるため，ロングワイヤにより読み出しレイテンシは増加する可能性があるが，このレイテンシは性能への影響が小さいことが知られている [10]．また，アクティブ電力は該当するテーブルのみ発生するため，レジスタ読み出し 1 回あたりのアクティブ電力は大きくは増加しない．

STRAIGHT バックエンドは従来の RISC 同様の機能ユニットを備え，1 命令単位で実行する．発行幅や実行幅の増加は行なわず，バックエンドのクラスタ化なども行なわない．従来と同等のクリティカルパス長をもつ集中型データパスと，バイパスネットワークを備える．実行後，コミットはインオーダに行なわれるが，実行と例外の確認後に PC，GP, FP，SP，RP のコミットステートを更新するのみで良い．RP の更新により暗黙的にフリーレジスタが更新されるため，ほぼイシュー・アンド・フォーゲットと言える軽量コミットとなる．

4.4 レジスタ数と命令ウィンドウサイズの設計

4.1 節で述べたように，命令形式中の各オペランドフィールドの長さを m ビットとすると，各命令は 2^m 前までの

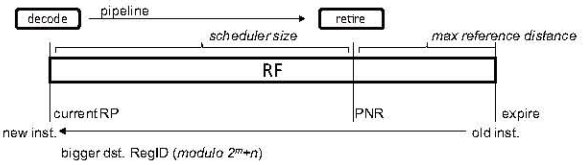

図 7 ディスティネーションレジスタ番号とパイプライン
Fig. 7 the Relationship between Dest. Register ID and Pipeline Stages

命令の実行結果を参照することができる．ここで，RP およびレジスタ番号がターンアラウンドする最大数について考える．ある命令がコミットするときに，その命令のディスティネーション番号から 2^m 離れたレジスタ番号はそれ以降アクセスされないことが保証され，新たに割り当て可能となる．n をコミットとデコードの間の最大命令数とすると，このとき，コミット中命令のレジスタ番号の後に最大で n 個の番号が割り当てられている．n はすなわち命令ウィンドウサイズであり，スケジューラのエントリ数にほぼ等しい．

このことから，RP は最低でも $2^m + n$ までの数値をとり，その後ラウンドして 0 に戻る．レジスタファイルは，先に述べた単純な実装であれば $2^m + n$ エントリのテーブルとなり，スケジューラマトリクスは $n \times n$ のサイズで構成される．**図 7** はこの関係のパイプラインとの対応を示したものである．

5. 評価環境

STRAIGHT アーキテクチャのアーキテクチャパラメタ，および IPC と IPC/電力比について初期見積もりを行う．まず，現在の RISC コードを対象に，レジスタ依存距離の分布を計測し，STRAIGHT 命令コーディングにおけるレジスタ参照距離制限の現実性を調べる．

次に，実行幅を増加させずに，命令ウィンドウ幅とフロントエンド幅の増加のみでどの程度の IPC 向上効果が得られるかを調べる．IPC の取得には STRAIGHT コードとパイプラインシミュレータが必要になるが，本論文では初期見積もりとして，RISC スーパスカラシミュレータのアーキテクチャパラメタを変更することによって STRAIGHT に見立てた評価を行った．STRAIGHT におけるレジスタプロモーションによる性能向上効果，逆にコード制約によるオーバヘッドは反映されないが，最新の分岐予測やメモリパラメタ下での ILP と命令ウィンドウサイズの相関は取得できる．パイプラインシミュレータには鬼斬 rev.5321[24] を用い，SPECCPU2006 ベンチマークの全プログラムについて先頭 10,000,000,000 命令をスキップし，続く 100,000,000 命令の動作を計測する．ベースラインプロセッサおよび STRAIGHT に見立てたモードのアーキテクチャパラメタを**表 1** に示す．STRAIGHT アーキテクチャではレジスタリネーミングが不要のため，フロントエンドレイテンシが縮まっている．

さらに，電力シミュレータを用いて IPC/電力比および面積の評価を行う．電力シミュレータとして McPAT[25] を用い，22nm プロセスを仮定した上で，上記パイプラインシミュレーションの構成パラメタおよび出力統計から入力を作成した．出力統計には上記 SPECCPU2006 ベンチマークの全プログラムの実行が反映されている．

6. 評価結果

6.1 レジスタ参照距離制限の現実性

図 8 は SPECCPU2006 の全ベンチマークについて，ソースレジスタが何命令前に依存しているかの分布を示した累積グラフである．図 8(a) が依存距離 1024 命令までの様子を示しており，(b) は 64 命令までの領域を拡大して示している．計測は通常の RISC コードについて行なっているため，分岐後にレジスタ値を持ち越せないケースのある STRAIGHT では依存距離が短くなる傾向にある可能性があるが，ここでは初期見積もりとして，この結果に基いて議論を行う．

図 8 では，過半数のソースオペランドは前方 10 命令以内で生成された値を用いている一方で，1000 命令を超える依存距離を持つオペランドも 7%強存在する．グラフからは，参照距離の制限を 64 命令とすると，全ソースオペランドの約 8 割，256 命令までとすると約 9 割が含まれることが分かる．この時，オペランド長はそれぞれ 6 ビット乃至 7 ビット，プロセッサから見える物理レジスタ数は 64 エントリ乃至 256 エントリ増加する．参照範囲を非常に長くしても含まれない一部のオペランドを除けば，数百命令の参照距離制限であれば，十分に従来のレジスタ依存を記述可能であることが分かる．なお，依存距離が参照可能範囲よりも長くなる場合，STRAIGHT ではレジスタムーブ命令を挿入して中継する．

6.2 IPC への影響

STRAIGHT アーキテクチャにおける命令ウィンドウとフロントエンド幅のそれぞれのスケールメリットを表 1 のパラメタによって評価した．表 1 に示すように，ベースラインパラメタは，レイテンシコア向けの，ラージコアと呼ばれる規模であり，従来のアーキテクチャではこの IPC を上回ることは難しい．**図 9** にフロントエンド幅とコミット幅をベースラインの等倍 (frontend4)，2 倍 (frontend8)，4 倍 (frontend16) と変化させ，同時に，命令ウィンドウ幅を等倍 (STRAIGHT1)，4 倍 (STRAIGHT4)，8 倍 (STRAIGHT8) としたときの SPECCPU ベンチマークの幾何平均 IPC 変化を 3 次元グラフで示している．x 軸と y 軸にそれぞれスケールパラメタをとり，z 軸にベースラインを 1 とした相対 IPC を示している．メモリおよび分岐による影響とポテンシャルを確かめるため，(a) メモリレイテンシが 200 サ

表 1　アーキテクチャパラメタ
Table 1　Architecture Parameters

	ベースライン	STRAIGHT
フロントエンド幅	4	4 / 8 / 16
リタイア幅	6	6 / 12 / 24
スケジューラサイズ	int32+fp16	int32+fp16 /int128+fp64 /int256+fp128
レジスタファイル	int128+fp128	int128+fp128 / int512+fp512 / int1024+fp1024
フロントエンドレイテンシ	7 cycle	5 cycle
発行幅		int2, fp2, mem2
D1 キャッシュ		64KB, 8way, 64Bline, 3cycle hit latency
I1 キャッシュ		64KB, 8way, 64Bline, 3cycle hit latency
L2 キャッシュ		4MB, 16way, 64Bline, 12cycle hit latency, with stream + stride prefetcher
メインメモリ		200cycle / 50cycle

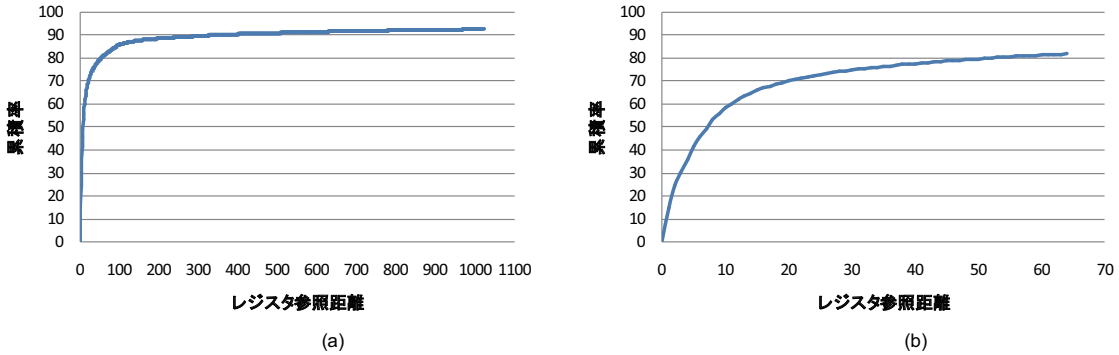

図 8　レジスタ参照距離の分布（累積グラフ）
Fig. 8　Distribution of Register References (Accumulated Graph)

イクルのとき，(b)3 次元積層化などの将来技術でレイテンシが縮まったとき (50 サイクル)，(c) 参考として理想分岐予測と理想キャッシュを仮定したときの傾向をそれぞれ取得し，図 9 では左から順に示している．(a)，(b)，(c) とも相対 IPC の基準はメモリレイテンシ 200 サイクルのベースラインモデルとしている．

3 次元グラフからはフロントエンド，命令ウィンドウ幅の増加とともに IPC の向上が見られ，メモリレイテンシが 200 サイクルの場合ベースラインから 30%の性能向上が見込まれ，今後の積層などによってこのポテンシャルはさらに約 10%向上することが分かる．STRAIGHT1 の系列に注目すると，命令ウィンドウサイズに対して大きすぎるフロントエンド幅はかえって性能を落としている．一方で，命令ウィンドウサイズの大きい STRAIGHT4 などの系列に注目すると，フロントエンド幅を大きくすることで更に性能向上が得られることが分かる．STRAIGHT は双方を軽量に拡張することができ，相乗効果で性能を向上させることができる．

図 10 は，frontend8，メモリレイテンシ 50 サイクルのときの命令ウィンドウサイズと相対 IPC の関係をベンチマーク毎に示したものである．ここでは，ベースラインモデルのメモリレイテンシも 50cycle としたときの相対 IPC を示している．ベースラインと命令ウィンドウサイズの変わらない STRAIGHT1 モデルに対し，命令ウィンドウが 4 倍となる STRAIGHT4 では調和平均にして 20%の性能向上を得ている．433.milc，444.namd などいくつかのベンチマークプログラムではさらに倍の命令ウィンドウサイズをもつ STRAIGHT8 モデルによってさらに性能が向上しているが，調和平均では STRAIGHT4 以降では飽和が見られている．

6.3　面積および電力評価

回路および電力評価では，スケールメリットの調査からパフォーマンス効率が良いと判断された frontend8/STRAIGHT4 のパラメタモデルについて見積もりを行った．まず，コアあたりの面積評価を**図 11** に示す．左からベースラインプロセッサ，STRAIGHT アーキテクチャ，さらに，もし同じ規模の命令ウィンドウサイズをスーパスカラアーキテクチャで構成した場合の面積を，各ユニットの大きさ毎に示している．

L2 キャッシュ，メモリコントローラ，命令キャッシュ，データキャッシュ，実行ユニットは規模が変わらないため，全てのモデルでほぼ同じ面積となっている．一方で，リネームロジックの面積について，STRAIGHT では不要となっている．リネームロジックの面積は，ベースラインモデルの命令ウィンドウサイズでは無視出来るほどである

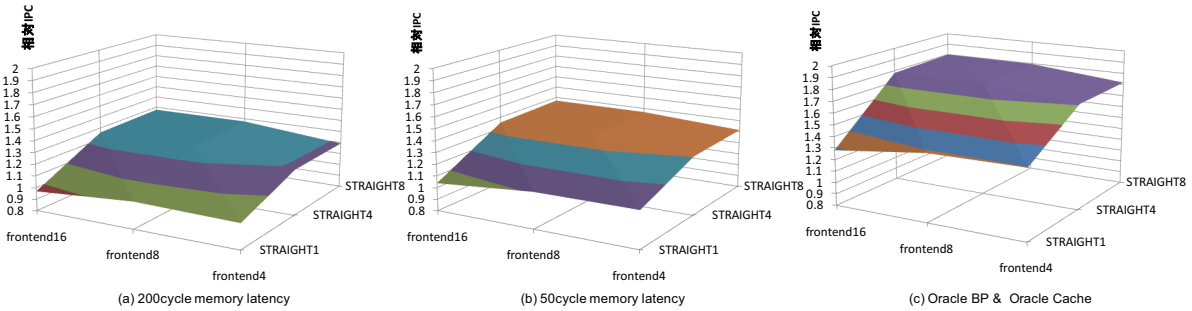

図 9 フロントエンド幅，命令ウィンドウサイズと IPC
Fig. 9 the Relationship between Frontend Width, Instruction Window size and IPC

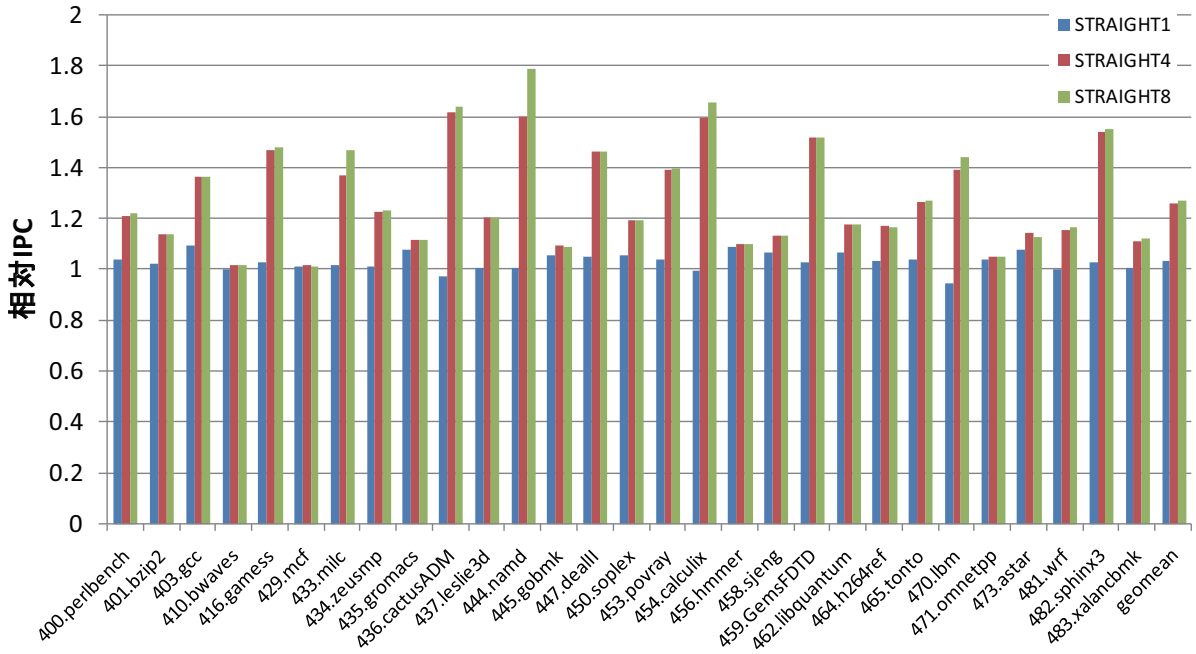

図 10 面積見積もり

が，STRAIGHT の規模のリネームロジックを従来アーキテクチャで設計すると，全コアの 1/3 近くを占める大きさになることが分かる．一方，レジスタとスケジューラの面積はベースラインに比べて STRAIGHT では増加しており，この増加分がコア全体の面積のオーバヘッドとなっている．コアあたりの面積オーバヘッドはプライベート L2 面積によって変化するが，今回の見積もりでは 15%程度となった．

次に電力・消費電力評価の結果を**図 12** に示す．まず，図 12(a) はキャッシュを除いた実行コア部分について，実行電力の内訳を示したものである．STRAIGHT アーキテクチャではリネームロジックの電力が必要ない一方で，レジスタのリーク電力やスケジューラ電力の増加により，トータルでは，ベースラインプロセッサの実行コア部分に対し 18%の電力増となった．次に，同じ処理への実行サイクル数を反映させ，消費電力の相対比較とした図が図 12(b) である．STRAIGHT では同じ処理に必要なエネルギー量を約 12%削減している．各項目に着目すると，リネーミングの排除とスケジューラおよびレジスタリークの影響がほぼ相殺する一方で，サイクル数短縮による実行機構のエネルギー減が貢献していることが分かる．

図 12(c) は，STRAIGHT と同じ規模の命令ウィンドウをスーパスカラアーキテクチャで設計した場合のエネルギー見積もりを加えたグラフとなっている．従来アーキテクチャの設計では，大きな命令ウィンドウを実現するためのレジスタファイル容量を得ようとすると，リネームロジックおよびレジスタアクセスに必要となるエネルギーが非現実的に増大することが分かる．

7. 議論：分散キー・バリュー・ストアの導入

STRAIGHT の性能・電力ブレイクスルーの鍵である大容量レジスタは，近年急速な進歩をとげている分散キー・

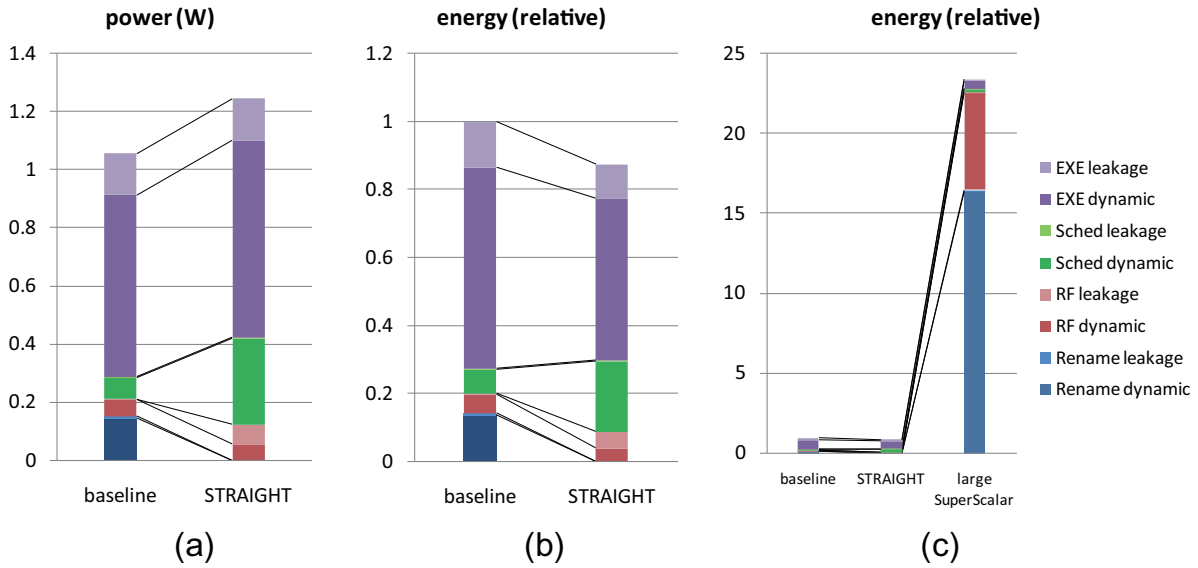

図 12 電力および電力/性能比の見積もり

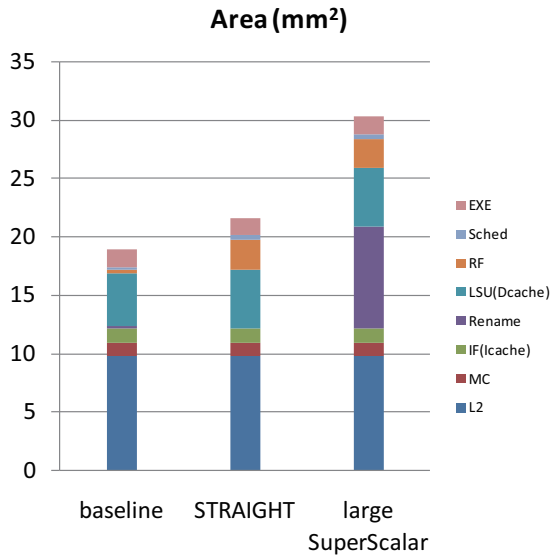

図 11 面積見積もり

バリュー・ストア技術をもちいることでハードウェアを効率化できる．大きなキー・バリュー・ストアを水平分散する技術は，コンシステント・ハッシング [26] やその発展アルゴリズム [27] など，通信量や制御を減らしながら，テーブルの分散管理について軽量で独立したコントローラ間で合意を得ることができる．第 4 節の図 5 に示したように，STRAIGHT アーキテクチャではレジスタファイルへのアクセスは，コア周縁部にある分散レジスタファイルのいずれかとの通信となる．しかし，多くのオペランド読み出しはバイパスネットワークから読み出されるため，レジスタファイルとのロングワイヤ通信を必要としない．さらに，32 エントリほどのレジスタキャッシュ [28] を導入すれば，ソースオペランド読み出しの大部分はレジスタファイルからの読み出しを必要とせず，集中したデータパス内に閉じた転送となる．また，寿命の短い値を生成する命令をコード中に明示できれば，分散レジスタファイルへの書き込みを必要とする命令の数を削減することができる．このような最適化を行ったとき，プロセッサから見える $2^m + n$ 個のレジスタは，全ての実体を備えている必要はなく，書き込み発生時に分散レジスタファイルの各マネージメントユニットがエントリを割り当てれば良い．$2^m + n$ 個の ID 空間のマッピングで複数マネジャが合意を得る方法や，プログラムフェーズによって増減する必要レジスタ数に従って分散テーブルの数を動的に変化させる方法など，既存の分散キー・バリュー・ストア手法を利用することによって，制御通信を増加させることなく，小容量化のための分散制御が可能となり，STRAIGHT アーキテクチャのための有力な最適化手法と考えられる．

8. まとめ

将来のメニーコアプロセッサの性能向上は，TLP 収穫逓減や消費電力の問題から，IPC と IPC/電力比の両方を同時にブレイク・スルーする実行アーキテクチャの実現が鍵となっている．本論文では，トランジスタ資源をレジスタ容量に用いて制御電力消費を軽量化する STRAIGHT アーキテクチャを提案した．STRAIGHT アーキテクチャレジスタ容量とライトワンスマナーによってレジスタリネーミングを排し，命令ウィンドウサイズを大きくすることによって，ピーク実行幅を拡張することなく IPC を増加させる．

コード仕様と生成手法，パイプラインアーキテクチャについて説明し，パイプラインおよび電力シミュレータによる初期見積もりを行った．SPECCPU2006 ベンチマークを対象とした評価では，従来のラージコアに対し，IPC を約 3 割増加させると同時に実行コア部分の消費電力を 12%削

減する結果となった．

STRAIGHT 命令コードの生成アルゴリズムの一般化と，スケジューラ，レジスタキャッシュなどのマイクロアーキテクチャ最適化が今後の課題である．

謝辞

本研究の一部は JSPS 科研費 25730028 の助成による．

参考文献

[1] Sylvester, D. and Keutzer, K.: Getting to the bottom of deep submicron , *Int. Conf. on Computer-Aided Design*, pp. 203 – 211 (1998).
[2] Lotfi-Kamran, P., Grot, B., Ferdman, M., Volos, S., Kocberber, O., Picorel, J., Adileh, A., Jevdjic, D., Idgunji, S., Ozer, E. and Falsafi, B.: Scale-Out Processors, *Int. Symp. on Computer Architecture* (2012).
[3] Duran, A. and Klemm, M.: The Intel Many Integrated Core Architecture, *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pp. 365 – 366 (2012).
[4] Bhadauria, M., Weaver, V. and McKee, S.: Understanding PARSEC Performance on Contemporary CMPs, *Int. Symp. on Workload Characterization*, pp. 98 – 107 (2009).
[5] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K. and Burger, D.: Dark Silicon and the End of Multicore Scaling, *Micro, IEEE*, Vol. 32, No. 3, pp. 122 – 134 (2012).
[6] IRIE, H., FUJIWARA, D., MAJIMA, K. and YOSHINAGA, T.: STRAIGHT: Realizing a Lightweight Large Instruction Window by using Eventually Consistent Distributed Registers, *Int. Workshop on Challenges on Massively Parallel Processors*, pp. 336 – 342 (2012).
[7] Ipek, E., Kirman, M., Kirman, N. and Martinez, J.: A Reconfigurable Chip Multiprocessor Architecture to Accommodate Software Diversity, pp. 1 – 6 (2007).
[8] Boyer, M., Tarjan, D. and Skadron, K.: Federation: Boosting per-thread performance of throughput-oriented manycore architecture, *Trans. on Architecture and Code Optimization*, Vol. 7, No. 4, pp. 19:1 – 19:38 (2010).
[9] Sohi, G., Breach, S. and Vijaykumar, T.: Multiscalar processors, *Int. Symp. on Computer Architecture*, pp. 414 – 425 (1995).
[10] Palacharla, S., Jouppi, N. P. and Smith, J. E.: Complexity-Effective Superscalar Processors, *24th Int. Symp. on Computer Architecture*, pp. 1–13 (1997).
[11] Oberoi, P. and Sohi, G.: Parallelism in the front-end, *Int. Symp. on Computer Architecture*, pp. 230 – 240 (2003).
[12] Krishnan, V. and Torrellas, J.: A chip-multiprocessor architecture with speculative multithreading, *Trans. on Computers*, Vol. 48, No. 9, pp. 866 – 880 (1999).
[13] Jiménez, D. A. and Lin, C.: Neural methods for dynamic branch prediction, *ACM Trans. Comput. Syst.*, Vol. 20, No. 4, pp. 369 – 397 (2002).
[14] Gabbay, F. and Mendelson, A.: Using value prediction to increase the power of speculative execution hardware, *ACM Trans. Comput. Syst.*, Vol. 16, No. 3, pp. 234 – 270 (1998).
[15] Albonesi, D., Balasubramonian, R., Dropsho, S., Dwarkadas, S., Friedman, E., Huang, M., Kursun, V., Magklis, G., Scott, M., Semeraro, G., Bose, P., Buyuktosunoglu, A., Cook, P. and Schuster, S.: Dynamically Tuning Processor Resources with Adaptive Processing, *IEEE Computer*, Vol. 36, No. 12, pp. 49–58 (2003).
[16] Usami, K., Shirai, T., Hashida, T., Masuda, H., Takeda, S., Nakata, M., Seki, N., Amano, H., Namiki, M., Imai, M., Kondo, M. and Nakamura, H.: Design and Implementation of Fine-Grain Power Gating with Ground Bounce Suppression, *Int. Conf. on VLSI Design*, pp. 381 – 386 (2009).
[17] 一林宏憲, 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一: 逆 Dualflow アーキテクチャ, 情報処理学会論文誌コンピューティングシステム, Vol. 1, No. 2, pp. 22 – 33 (2008).
[18] 塩谷亮太, 安藤秀樹: 一致経路長の短縮による Renamed Trace Cache のエネルギー効率向上, pp. 56 – 64 (2013).
[19] Hsieh, A. and Hwang, T.: TSV Redundancy: Architecture and Design Issues in 3-D IC, *Trans. on Very Large Scale Integration Systems*, Vol. 20, No. 4, pp. 711 – 722 (2012).
[20] Conte, T., Menezes, K., Mills, P. and Patel, B.: Optimization of instruction fetch mechanisms for high issue rates, *Int. Symp. on Computer Architecture*, pp. 333 – 344 (1995).
[21] Rotenberg, E., Bennett, S. and Smith, J.: Trace cache: a low latency approach to high bandwidth instruction fetching, *Int. Symp. on Microarchitecture*, pp. 24 – 34 (1996).
[22] Goshima, M., Nishino, K., Nakashima, Y., Mori, S., Kitamura, T. and Tomita, S.: A high-speed dynamic instruction scheduling scheme for superscalar processors, *34th Int. Symp. on Microarchitecture*, pp. 225–236 (2001).
[23] Sassone, P., J. Rupley, I., Brekelbaum, E., Loh, G. and Black, B.: Matrix scheduler reloaded, *Int. Symp. on Computer architecture*, pp. 335–346 (2007).
[24] 塩谷亮太, 五島正裕, 坂井修一: 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 2009 ポスター (2009).
[25] Li, S., Ahn, J.-H., Strong, R., Brockman, J., Tullsen, D. and Jouppi, N.: McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures, *Int. Symp. on Microarchitecture*, pp. 469 – 480 (2009).
[26] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. and Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web, *Symp. on Theory of computing*, pp. 654 – 663 (1997).
[27] Stoica, I., Morris, R., Karger, D., Kaashoek, M. and Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, *ACM SIGCOMM*, Vol. 31, No. 4, pp. 149–160 (2001).
[28] Yung, R. and Wilhelm, N.: Caching processor general registers, *Int. Conf. on Computer Design*, pp. 307 – 312 (1995).