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

2006－ARC－169 (8)
2006／8／1

ツインテール・アーキテクチャ

平井 遥[†] 入江 英嗣[††]
五島 正裕[†] 坂井 修一[†]

スーパスカラ・プロセッサの命令パイプラインにおいて、命令ウインドウより上流をフロントエンドと呼び、命令ウインドウおよびその下流をバックエンドと呼ぶ。従来のスーパスカラ・プロセッサでは演算器はバックエンドに配置され、命令の実行はバックエンドのみで行われる。これに対して我々はフロントエンド実行という手法を提案している。フロントエンド実行とはバックエンドに加えてフロントエンドにも演算器を配置し、実行可能な命令をフロントエンドでも実行することである。フロントエンド実行には従来のプロセッサに比べクリティカル・パス上の命令の実行間隔を狭める効果がある。本稿ではフロントエンド実行の考え方を押し進め、改良手法としてツインテール・アーキテクチャと呼ぶ手法を提案する。ツインテール・アーキテクチャはフロントエンド実行ステージを通常のパイプラインから独立させたものであり、これによってフロントエンド実行ステージによるパイプライン段数の増加はなくなる。この手法はフロントエンド実行において難点であった部分を改善してさらなる性能向上を図ることを目的とした手法である。

Twintail Architecture

Haruka Hirai,[†] Hidetsugu Irie,[††] Masahiro Goshima[†] and Shuichi Sakai[†]

The front end (or back end) of the pipeline in a superscalar processor refers to the pipeline before (or after) the instruction window. Traditionally, physical locations of ALUs, as well as instruction execution belong to the backend. We have proposed front end execution (FEE). In FEE, ALU allocation and instruction execution also take place in the front end. By early executing instructions with ready source operands, execution time of critical instructions can be reduced. This work proposes an enhanced FEE, called twintail architecture. In twintail architecture, the stages responsible for FEE are separated from the main pipeline. By which, the overheads previously caused by additional FEE stages can be removed.

1. はじめに

スーパスカラ・プロセッサの命令パイプラインにおいて，命令ウィンドウより上流を**フロントエンド**，下流を**バックエンド**と呼ぶ．通常のスーパスカラ・プロセッサでは，演算器はバックエンドに配置され，命令の実行はバックエンドのみで行われる．これに対して小西らは**フロントエンド実行** (frontend execution) という手法を提案している[5]．フロントエンド実行とは，バックエンドに加えてフロントエンドにも演算器を配置し，バックエンドに加えてフロントエンドでも命令の実行を行う手法である．図1に従来のプロセッサのパイプラインとフロントエンド実行を行う場合のパイプラインを示す．図のように，フロントエンド実行は，

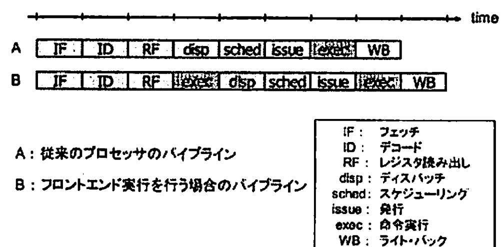

図1 フロントエンド実行

従来のパイプラインにおける，レジスタ読み出しと，ディスパッチの間で行われる．

詳しい理由は次章以降で説明するが，フロントエンド実行は通常のスーパスカラ・プロセッサでは（原理的に）最適にスケジューリングできない命令を早期に実行することで性能の向上を図る．依存関係にある命令をすべて back-to-back に実行することができれば，プログラムの実行時間は最短となる．しかし実際には，

† 東京大学
University of Tokyo
†† 東京大学/科学技術振興事業
University of Tokyo/Japan Science and Technology Agency

当然のことながら，そのようなことは不可能であり，依存先の命令が何サイクルにもわたってスケジューリングされないことがある．フロントエンド実行では，このような命令を早期に実行することにより，性能向上を狙うのである．文献 5) では，フロントエンド実行によって，従来のプロセッサに比べ 5～15%の性能向上が得られることが示されている．

さて，図 1 は，フロントエンド実行に 1 サイクルを割り当てた場合のものであるが，演算器を多段に配置し，2 サイクル以上を割り当てることも提案されている．その場合，より多くの命令をフロントエンドで実行することができる．しかし一方で，そのサイクル数分だけパイプライン段数が増え，分岐予測ミス・ペナルティが増加してしまう．すなわち，フロントエンド実行ステージ段数が，フロントエンド実行可能な命令数と分岐予測ミス・ペナルティの間のトレード・オフになっているのである．

そこで本稿では，フロントエンド実行を改良した**ツインテール・アーキテクチャ** (Twintail Architecture) を提案する．**図 4** 下に，ツインテール・アーキテクチャの概略図を示す．フロントエンド実行では，フロントエンド実行ステージとバックエンド実行ステージが直列に並べられている．それに対してツインテール・アーキテクチャでは，フロントエンド実行ステージとバックエンド実行ステージを並列に並べたものと解釈することができる．ツインテール・アーキテクチャでは，フロントエンド実行ステージに相当する部分を **in-order tail**，バックエンド実行ステージに相当する部分を **out-of-order tail** と呼ぶ．In-order tail では，その名のとおり，命令は in-order に実行される．そのため，out-of-order tail における out-of-order 実行のための，ディスパッチ，スケジューリング，および，発行のためのステージがなく，その分早期に命令を実行することができる．ツインテール・アーキテクチャでは，in-order tail と out-of-order tail を並列に並べることにより，フロントエンド実行可能な命令数と分岐予測ミス・ペナルティの間のトレード・オフを解消できるのである．

以下，本稿ではまず 2 章 でフロントエンド実行についてまとめた後，3 章 でツインテール・アーキテクチャについて詳しく述べる．そして，5 章 でまとめと今後の課題について述べる．

2. フロントエンド実行

本章では先行研究であるフロントエンド実行についてより詳しい説明を行う．ここでは，フロントエンド実行の仕組みについてその詳細を述べる．フロントエンド実行の効果については提案手法の効果と重なる部分が多いため，次章で提案手法の効果として述べることにする．

2.1 スーパスカラ・プロセッサの構成方式

既存の Out-of-order スーパスカラ・プロセッサを構成する方式には，**リザベーション・ステーション (RS) とリオーダ・バッファ(RB) を併用する方式**と，**物理レジスタに対するリネーミングを行う方式**が存在する．両者にはそれぞれいくつかの特徴があるが，特にそのレジスタ読み出しのタイミングに注目すると前者はレジスタ読み出しをフロントエンドで行う方式であり，後者はレジスタ読み出しをバックエンドで行う方式である．フロントエンド実行を行う際にはレジスタ読み出しをフロントエンドで行う必要があるので両者のうち前者を採用することが提案されている．

2.2 導入が必要なハードウェア

先行研究においてはフロントエンド実行にはそのステージとして 1～2 サイクルを割り当てることが提案されている．

フロントエンド実行に 1 サイクルを割り当てる際にはシングル・サイクルで実行を完了することができる ALU 演算やシフト演算のみを行う．これを行うためにはフェッチ幅と同数の演算器をフロントエンドに用意してやるだけでよい．

また，フロントエンド実行に 2 サイクルを割り当てる際には依存関係にある 2 つのシングル・サイクル演算を行うことに加えて，ロード命令を行うことも提案されている．この場合には 1 ステージ目でアドレスの計算を行い，2 ステージ目にキャッシュへのアクセスを行う．アクセスするキャッシュとしては 1 ステージでアクセスすることが可能な 1KB 程度の 0 次キャッシュをフロントエンドにも用意することが考えられている．[3]

このようにフロントエンド実行を行うためにはフェッチ幅と同数の演算器および小容量のキャッシュを用意すれば十分である．現存のスーパスカラ・プロセッサにとってはこれらは大きなハードウェアではなく，したがってフロントエンド実行に必要なハードウェア・コストは小さいと言ってよい．

2.3 フロントエンド実行機構

以上のことを踏まえた上で**図 2** に具体的なフロントエンド実行機構を示す．図 2 は 2 ウェイのスーパスカラ・プロセッサにおいて 2 段のフロントエンド実行を行う場合の実行機構である．

フェッチされた命令はまずフロントエンドでレジスタ読み出しを行い，この時点で演算可能ならば直下に

配置された演算器で直ちに演算を行う．この時点で演算可能でない命令はフロントエンド演算器の最後段まで進んだ後に従来のスーパスカラ・プロセッサと同様に命令ウィンドウにディスパッチされ，演算可能となったものから発行されてバックエンドの演算器で実行されることになる．

図2のようにフロントエンド実行を多段にする場合には，基本的には全対全の接続が必要となる．図2のフロントエンド実行ユニットでは自分自身やフロントエンド実行の2段目から1段目へのバイパスを設けることは行っていないが，これらのバイパスを設けることも考えられる．このようにすることで，フロントエンド実行できる命令数は増加すると考えられる．

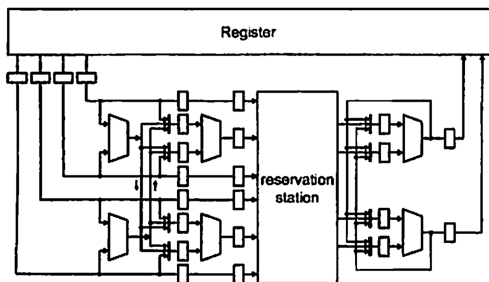

図2　フロントエンド実行機構

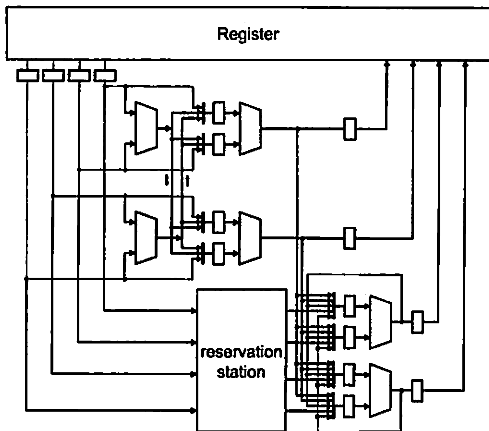

図3　ツインテール・アーキテクチャの実行機構

3. ツインテール・アーキテクチャ

3.1 ツインテール・アーキテクチャの提案

フロントエンド実行を行う場合にはその実行ステージの分だけ全体のパイプライン段数は増加してしま

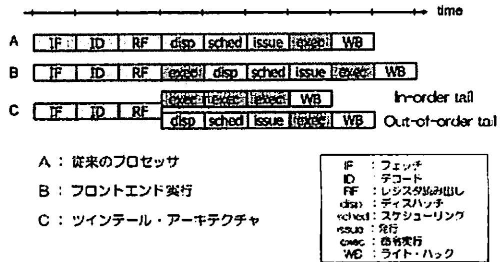

図4　ツインテール・アーキテクチャ

う．これは，図1で示したようにフロントエンド実行ステージが従来のプロセッサにおけるパイプラインのレジスタ読み出しとディスパッチの間に新たなステージとして組み込まれているからである．その結果，フロントエンド実行できなかった命令のバックエンド実行はフロントエンド実行ステージの分だけ遅れてしまうことになる．

本稿ではこれを改善する手法としてツインテール・アーキテクチャと呼ばれる手法を提案する．この手法では図4に示したようなパイプライン処理を行う．

まず，命令は従来のプロセッサと同様にフェッチされ，デコード，レジスタ読み出しの処理が行われる．その後，命令を複製し，これに対して in-order tail と out-of-order tail で異なる処理を行う．

in-order tail ではフロントエンド実行と同様の処理が行われる．in-order tail には演算器が多段に配置されており，そのステージにおいて命令が実行可能な場合にはその命令を実行する．実行可能でない命令に関してはそのまま次のステージへと送られる．本稿では in-order tail として3段程度の実行ステージを考えている．

また,out-of-order tail では従来のスーパスカラ・プロセッサと同様の処理を行う．out-of-order tail は命令ウィンドウや演算器など従来のプロセッサと同様な構成をとる．命令はスケジューリング処理の後，実行ステージで実行される．

3.2 提案手法の利点

ツインテール・アーキテクチャはフロントエンド実行ステージによってパイプライン段数が増加してしまうことを改善した手法である．

フロントエンド実行では従来のパイプラインにフロントエンド実行ステージを追加していたために全体のパイプライン段数が増加してしまっていた．それに対して，ツインテール・アーキテクチャでは新たに加える実行ステージを従来のパイプラインである out-of-order tail から切り離し，全く別のパイプラインである

in-order tail として独立させている.

したがって,in-order tail のパイプライン段数をいくら増加させたとしても out-of-order tail のパイプライン段数が増加することはない. ツインテール・アーキテクチャではパイプライン段数の増加を気にすることなく,in-order tail の実行ステージを多段化することが出来るのである.

3.3 提案手法詳細

3.3.1 バイパス・ネットワーク

ツインテール・アーキテクチャでは in-order tail と out-of-order tail の 2 つが並列にならぶ構成となる. また,in-order tail にはカスケード接続された演算器が多段に配置され,out-of-order tail には従来のプロセッサと同様に命令ウィンドウやバックエンド演算器が配置される. したがって, これはちょうど図 2 におけるフロントエンド演算器を命令ウィンドウと直列ではなく並列に並べた構成となる.

しかし, ツインテール・アーキテクチャではフロントエンド実行の場合と違い,in-order tail の最後段の実行ステージでの実行結果は次のサイクルに out-of-order tail で使用される場合もある. したがって,in-order tail の最後段の演算器から out-of-order tail の演算器へとバイパスを設ける必要がある.

以上より, ツインテール・アーキテクチャの実行機構は図 3 のようなものとなる.

3.3.2 ツインテール・アーキテクチャの振る舞い

次にツインテール・アーキテクチャの動作について述べる. フェッチされた命令はフロントエンド実行の時と同様にしてレジスタ読み出しをフロントエンドで行い, 以降では in-order tail と out-of-order tail でそれぞれ異なる処理をされることになる. まず, ツインテール・アーキテクチャでは命令のソース・オペランドとして読み出されたデータは in-order tail と out-of-order tail の双方にそれぞれ送られる.

in-order tail には先に述べたように多段に演算器が配置されている. これらの演算器は送られてきた命令がその時点で実行可能ならば演算を行い, そうでなければ演算を行わずに次の段の演算器へと命令を送る. このようにして in-order tail では命令は最後段まで送られた後に実行できた命令の実行結果のみを残して破棄される. なお, 下に示すように in-order tail で演算を行うことができなかった命令に関しても out-of-order tail で必ず演算が行われるため, どの命令に関してもその実行は保障される.

out-of-order tail は従来のスーパスカラ・プロセッサのバックエンドと同様の構成をとっている. こちら側に送られてきた命令は従来のスーパスカラ・プロセッサと同様にして命令ウィンドウにディスパッチされる. このとき,in-order tail で実行された命令は命令ウィンドウから取り除く. これにより,in-order tail と out-of-order tail で命令が冗長に実行されてしまうことを防ぐのである. そして命令はスケジューリングの後に発行されてからバックエンドに配置した演算器で実行される.

3.3.3 構成方式

ツインテール・アーキテクチャではフロントエンド実行と同様に 2.1 節で述べた二種類のスーパスカラ・プロセッサの構成方式のうちリザベーション・ステーション (RS) とリオーダ・バッファ(RB) を併用する方式を採用する. すでに述べたようにこの方式を採用するとフロントエンドでレジスタ読み出しを行うことになるので, その直下に配置した in-order tail の演算器で実行を開始することが可能になる. また, この方式ではレジスタ読み出しがフロントエンドにおいて行われるため in-order tail と out-of-order tail への読み出しを同時に行うことができる. したがって, この方式を採用すればレジスタ読み出しポートを新たに増やす必要はない.

3.4 ツインテール・アーキテクチャの効果

依存関係を持つ命令同士を矢印で結んでいったグラフをデータフロー・グラフと呼ぶ. データフロー・グラフ上のすべての依存関係にある命令をこのグラフの通りに back-to-back に実行することができれば, プログラムの実行時間は最短となる. しかし, 現実に存在するプロセッサには様々な制約があるため, 当然のことながらこれを実現することは不可能である. そのため, 実際には依存元の命令が実行されてから依存先の命令が実行されるまでに何サイクルもかかってしまう場合が多い.

ツインテール・アーキテクチャは従来のプロセッサでは back-to-back に実行を行うことができていない場面において, 依存先の命令を in-order tail で実行してしまうことによって, データフロー・グラフ上からこの命令を取り除くことができる. データフロー・グラフ上でクリティカリティの高い命令[7] が含まれるパス上から命令を取り除くことができれば, プログラムの実行時間は短縮される.

従来のプロセッサが依存関係にある命令を back-to-back に処理できない理由として,「演算器・スケジューリング能力の不足」「フェッチ幅の不足」「分岐予測ミス」が考えられる.

演算器・スケジューリング能力の不足

図 5 は演算器・スケジューリング能力の不足から実

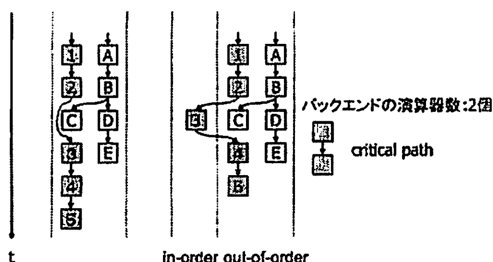

図5　ツインテール・アーキテクチャの効果

行時間に遅れが出てしまう場合の例である．縦軸 t は時間を表しており，この図は命令 A～E および 1～5 がどのタイミングで実行されたかを表している．図において従来のプロセッサでは本来は命令 3 を優先して実行しなければならないサイクルでスケジューリング能力の不足から C と D を優先して実行してしまっている．それにより，このサイクルで 3 を実行した場合に比べて実行時間が 1 サイクル遅れてしまうのである．これに対して，ツインテール・アーキテクチャで図のように 3 を in-order tail で実行することができたとすると,out-of-order tail では問題のサイクルの次のサイクルで 4 の実行を即座に開始することができる．これにより，実行時間の遅れが解消されていることが分かる．

フェッチ幅の不足

依存関係にある命令を back-to-back に実行できない他の理由としてフェッチ幅の不足が挙げられる．従来のプロセッサではフェッチ幅が十分でないためにプログラムに内在する並列性を十分に引き出すことができていないのである[2]．図 5 において仮に演算器が C と D に占有されていなかったとしてもそのサイクルに 3 がバックエンド実行ステージまで到達できなければ,2 と 3 を back-to-back に実行することはできない．つまり，依存関係にある命令を back-to-back に実行するためには依存元の命令がフェッチされた次のサイクルには依存先の命令がフェッチされていなければならないのである．依存先の命令のフェッチが遅れた場合，その分だけ命令の実行も遅れてしまう．これに対して，ツインテール・アーキテクチャではフェッチから in-order tail での実行までのステージ数が短いため，依存先の命令を in-order tail で実行することができれば，この遅れが軽減されるのである．

分岐予測ミス

分岐予測ミスも依存関係にある命令を back-to-back に実行できない理由のひとつである．分岐予測ミスが起こるとそれ以降の命令はフラッシュされ，その後に

A　IF | ID | Re | disp | sched | issue | exec | WB
B　IF | ID | Re | disp | sched | issue | exec | WB

従来のプロセッサでの実行の様子

A　IF | ID | Re | exec | exec | exec | WB
B　IF | ID | Re | disp | sched | issue | exec | WB

提案手法による実行の様子

図6　分岐予測ミス後の様子

正しい分岐先の命令のフェッチが開始される．このときの従来のプロセッサとツインテール・アーキテクチャでの振る舞いの例を図 6 に示す．図において命令 B は命令 A に依存しているものとする．図のように分岐予測ミス後の命令 A が in-order tail で実行できた場合には従来のプロセッサに比べて実行時間が短縮されることが分かる．

3.5　プリロードによるキャッシュ・アクセス・レイテンシの秘匿

フロントエンド実行や in-order tail でロード命令を行う場合にはそのロード命令がフロントエンドに用意した 0 次キャッシュにヒットしなかった場合，命令はキャンセルされ，バックエンドで再び実行される．これに対してフロントエンドのキャッシュにヒットしなかった場合にも引き続きバックエンドに存在する高次のキャッシュにアクセスを行う手法も考えられる．

この手法は，ロード命令に依存する後続の命令を投機的に発行することで性能向上を図る手法である．フロントエンド実行にこの手法を取り入れたときの効果を図 7 に示す．

A：ロード命令　　B：Aに依存する命令

A　IF | ID | Re | disp | sched | issue | exec | Mem | Mem | Mem
B　IF | ID | Re | disp | ——— | sched | issue | exec

従来のプロセッサでの実行の様子

A　IF | ID | Re | exec | Mem | Mem | Mem
B　IF | ID | Re | exec | disp | sched | issue | exec

フロントエンド実行を行ったときの実行の様子

図7　プリロードの効果

図のようにキャッシュのアクセスが後続の命令のスケジューリング処理と同時に行われるため，ロード命令のキャッシュ・アクセス・レイテンシを 0 サイクルであるかのように見せかけることができるのである．この手法を取り入れることによってフロントエンド実行の場合には平均約 11%の性能向上が示されている．[6]

4. 関連研究

4.1 RENO

Vlad ら[1] はリネーミングを工夫することで一部の命令をリネーミング・ステージで取り除く手法として RENO（RENaming Optimizer）を提案している. RENO はマッピング・テーブルの上手な利用と物理レジスタの共有構成を用いて命令を取り除き, 依存関係を構成し直す手法である. これはフロントエンドで命令実行と等価な処理を行うことで, その命令を取り除く手法であるという点で本研究と関連があるといえる. RENO は（1）取り除かれた命令をデータフロー・グラフから取り除ける（2）取り除かれた命令は以降でハードウェア資源を使用しない, といった利点を持っており, これによって 8〜13%の性能向上が見込めることが示されている.

4.2 スーパスカラ・プロセッサのための物理レジスタ 2 段階解放

山本らは物理レジスタをリネーム・ステージと命令ウインドウとの 2 段階で解放することによって, 物理レジスタの不足によるリネーム・ステージでのストールを解消することを提案している[4]. さらに, 命令ウインドウで 2 段階目の解放を待ち受けている命令が実行可能であるならば先行実行を行い, そのことによってロード・データのキャッシュへのプリフェッチを実現している. 先行実行された命令の結果をその命令の結果に依存する命令にフォワーディングすることで, 先行実行は連鎖して行うことができるため, 先行実行された命令の中にロード命令が存在すれば, そのロード命令によるプリフェッチが行われるのである. この手法はロード命令を先行実行することでキャッシュ・アクセス・レイテンシを減少させることができる点で本研究と関連がある研究であるといえる. 山本らは SPECfp2000 ベンチマークを用いて測量を行い, この手法によって物理レジスタ数が 64 個の場合に平均で 32 %もの大きな性能向上を達成できることを示している.

5. まとめと今後の課題

本稿では我々が提案しているフロントエンド実行という手法についてその説明を行い, さらにこれを改善する手法としてツインテール・アーキテクチャと呼ぶ手法を提案した. これはフロントエンド実行ステージによって全体のパイプライン段数が増加してしまうという点を改善することを目的とした手法である.

今後の課題としてまずはツインテール・アーキテクチャのより詳細な評価を行うことがあげられる. 本稿では In-order tail の実行ステージを 3 段としているが, 評価を行うことによってこの段数を変更することなども考えられる.

また, 他の課題としてさらなる改良があげられる. 具体的には冗長な命令実行を防ぐための wakeup 論理の実現方法を示すことが考えられる. IPC を低下させないためには wakeup 論理には select 論理と合わせて 1 サイクル内に収めなければならないという制約がある.[3] したがって, 複雑な wakeup 論理を実現する際にはこのことに十分な注意を払う必要がある. これに対して In-order tail に送る命令を選別するという方法も考えられる. このためにはレジスタ読み出しを行った時点での情報から判断してあらかじめどちらの tail に命令を送るかを決めることができればよいのである.

謝辞　本研究の一部は, 文部科学省科学研究費補助金 基盤研究 B(2) #16300013, CREST プロジェクト「ディペンダブル情報基盤」, 21 世紀 COE プログラム「情報科学技術戦略コア」の支援により行った.

参考文献

1) Petric, V., Sha, T. and Roth, A.: RENO: A Rename-Based Instruction Optimizer, *Department of Computer and Information Science, University of Pennsylvania* (2005).
2) Sankaralingam, K., Nagarajan, R., Kim, H. L. C., Huh, J., Burger, D., Keckler, S. W. and Moore, C. R.: Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture, *Computer Architecture and Technology Laboratory Department of Computer Sciences The University of Texas at Austin* (2003).
3) 五島正裕: Out-of-order ILP プロセッサにおける命令スケジューリングの高速化の研究, 京都大学 (2004).
4) 山本哲弘. 安藤秀樹. 島田俊夫: スーパスカラ・プロセッサのための物理レジスタ 2 段階解放, *2005-ARC-164* (2005).
5) 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, *2004-ARC-158*, pp. 13–17 (2004).
6) 小西将人. 福田匡則. 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によるプリロードの提案, *2004-ARC-159*, pp. 31–36 (2004).
7) 福田匡則. 小西将人. 五島正裕. 中島康彦. 森眞一郎. 富田眞治: グローバル分岐履歴を用いたスラック予測器, *2004-ARC-151* (2004).