学術情報センター電子図書館サービス　　© (社)情報処理学会

計算機アーキテクチャ　133－6
(1999. 5. 21)

VLDP アーキテクチャにおけるデータアクセスの軽減手法

高峰　信[†]　辻　秀典[†]　吉瀬　謙二[†]
田中洋介[†]　坂井修一[†]　田中英彦[†]

我々は、8以上の命令レベル並列性を取り出すことを目的として、大規模な投機的命令実行を行うVLDPプロセッサ・アーキテクチャを提案し、研究を行なっている。複数パスを扱うVLDPアーキテクチャでは、制御するパスの数に応じて、演算機構で生成される結果が大幅に増加する。実装の際には、膨大な演算結果のリタイア機構が高速化の妨げになることが予測される。本稿では、演算機構内で全てのデータの供給が完結しリタイアが不要となる変数を Closely-Accessed 変数と名付ける。静的な解析により Closely-Accessed 変数を検出した結果、プログラムによっては半数以上の演算結果のリタイアを削減できることを保証可能であるという結論を得た。

Data Acccess Eliminating Technique in VLDP Architecture

MAKOTO TAKAMINE,[†] HIDENORI TSUJI,[†] KENJI KISE,[†]
YOUSUKE TANAKA,[†] SHUICHI SAKAI[†] and HIDEHIKO TANAKA[†]

We have proposed, and have been studying a new architecture VLDP targetting to get more than 8 ILPs by the large scale speculative execution. In this architecture, a lot of values are created in ALUs by executing the several paths speculatively. They require a lot of retiring operations, which make it difficult to implement such an architecture efficiently. In this paper, we named the variables Closely-Accessed variables that finish all data supplies within ALUs ; such variables require no retiring operations. We counted the number of Closely-Accessed variables statically and showed that retires of more than 50% of calculated values can be eliminated.

1. はじめに

近年、マイクロ・プロセッサに対する高速動作の要求は高まるばかりであり、スーパースカラ方式、VLIW方式、SMTプロセッサに代表される複数命令実行を可能としたさまざまなアーキテクチャが考案されてきた。現在のマイクロ・プロセッサの主流であるスーパスカラ・アーキテクチャでは、並列実行可能な命令を動的に取り出すために、分岐予測を用いた投機的実行、レジスタ名前替えによるデータ依存関係の解消といった技術が利用されている。今後、より大きな命令レベル並列性を取り出そうとした時、従来のアーキテクチャの拡張だけではハードウェアが極度に複雑化する。そのため、命令レベル並列性を取り出す研究が進むにつれ、従来のアーキテクチャでの速度向上の限界もまた指摘されている。それに対して半導体デバイス技術の進歩は、Mooreの法則で言われる3年でトランジスタ数が4倍という進歩が現在もほぼ続いており、こういった膨大なハードウェア資源の利用方法についての研究が現在盛んに行なわれている。

このような背景により、大規模なハードウェア資源を積極的に利用した新しいアーキテクチャである大規模データパス方式のプロセッサ[1)] (VLDPプロセッサ)を提唱し、これについて研究を行っている。VLDPプロセッサでは、豊富なハードウェア資源を何段にも接続した多数の演算器 (ALU-NET) に割り当て、1チップ内に大規模なデータパスを構築する。さらに、複数パスの並列処理を行うことで、より大きな命令レベル並列性の利用が可能となる。

VLDPプロセッサでは、1サイクル当たりに作成される演算結果が多大なものとなり、それら値のリタイアが大きな問題となる。従来のプロセッサの機構を用いると、データ・アクセスはレジスタ・ファイルを介して集中的に制御されるため、演算の数が増加した場合データ・アクセスの集中が起こる。この問題はリオーダ・バッファ等の機構を備えた場合も同様である。VLDPアーキテクチャでは、ALU同士を結合する接続網により、ALU間で直接データの授受を行うことでデータ供給の分散化を図ることができる。しかし、ALUからのリタイアについては、全てのALUから

† 東京大学大学院 工学系研究科
Graduate school of Engineering, The University of Tokyo

© (社)情報処理学会

の値をレジスタ・ファイルへ書き戻す必要がある。これは作成された値について、いつ最後の参照が行われるのかが保証されないことに起因する。

本稿では、リタイアを必要とする演算結果が膨大な個数になるアーキテクチャを前提とした場合のデータ・アクセスの集中を削減することを目的とする。そのために、ALU-NET内でデータの参照が完結し、リタイアが不要となる変数をClosely-Accessed変数と定義し、そのClosely-Accessed変数を静的なデータ依存解析により検出する。そして、プログラム中に存在するClosely-Accessed変数の割合について評価する。

2. 大規模データパス方式

2.1 VLDPアーキテクチャの概要

近年のマイクロ・プロセッサでは、機能ユニットを複数備え、命令の同時実行を可能にする技術が用いられている。複数命令の同時実行を行なうマイクロ・プロセッサでは、命令レベル並列性を積極的に引き出すことにより、IPC (Instruction Per Cycle) を増加させ高性能を達成している。

VLDPアーキテクチャは、多数のALUとそれらを接続する接続網によって構成されるALU-NET[2]を用いて、大規模な複数パス投機実行を行うことにより、8以上の高い命令レベル並列性を取り出すことを目指した新しいアーキテクチャである。ALU-NETを効率良く利用するには、制御依存およびデータ依存を解消するための機構が必要となる。VLDPアーキテクチャでは、前者の制約を解消する機構をコントロールフロー先行展開、後者の制約を解消する機構をデータパス先行展開と呼ぶ。図1にVLDPアーキテクチャのブロック図を示し、以降に各機構の簡単な説明を行う。

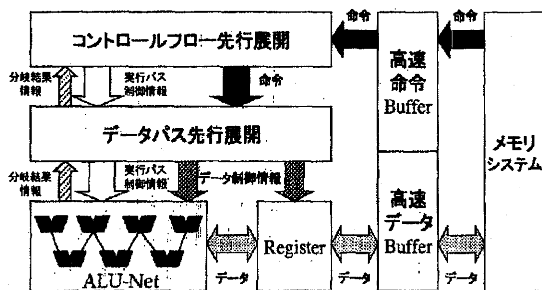

図1 VLDPアーキテクチャのブロック図

コントロールフロー先行展開[3)4)]では、投機的にフェッチした複数のパスについて制御依存関係に基づいた情報を付加する。データパス先行展開では、コントロールフロー先行展開で付加された制御依存情報をもとに、各制御パス毎に命令間のデータ依存関係を解析し、データパスを作成する。さらに、作成したデータパス情報に従って各命令の実行情報を得た後、ALU-NETへの命令の割り付けと実行パス制御を行なう。ALU-NETは数十〜数百のALUが接続網によって接続された演算機構である。1つのALUが作成した演算結果を、レジスタ・ファイルやフォワーディングパスの機構を介さずに、別のALUで利用することができるため、高速なデータ転送が可能となる。

2.2 ALU-NETにおけるデータ・アクセス

ALU-NETとは、1つのALUの出力を他のALUの入力として、必要最低限のスイッチとラッチを経由して接続したVLDPアーキテクチャの命令実行機構である。ALU間のネットワークにより、レジスタ・ファイル等を介さないデータの供給を行うことができるため、データ供給のオーバヘッドを抑えることができる。

現在、ALU-NET自体に拡張性を持たせることを考えており、デバイス技術の進歩と共にALUの個数を増加させることで、並列演算性能の向上を見込むことができる。ALUの個数を増加させるに従いALU-NETのデータ・アクセス量も増加し、関連するハードウェア機構のコストは膨大なものとなる。データ供給量の増大は接続網を介したデータの授受で削減できるが、リタイア量の増大を抑えることは困難である。そのため、膨大なリタイア制御を行うハードウェアが必要となり、全データを書き戻す場合には、VLDPは実装の現実性を欠いたアーキテクチャとなる。

3. データ・アクセスの削減

命令レベル並列性を向上させるために演算器の個数を増加させると、レジスタ・ファイルへのアクセスが集中し、プロセッサ全体の速度向上を妨げる。データ依存解析によって不必要なデータの移動を検出し、レジスタ・ファイルへのアクセスを削減する技術が要求される。

3.1 Short-Lived変数

スーパスカラ・プロセッサでは、一般にリオーダ・バッファを用いたレジスタ名前替えが行われ、並列実行可能な命令数を増やしている。リオーダ・バッファからレジスタ・ファイルに書き戻された値については、同じ番号のレジスタに次の値が上書きされるまで、レジスタ・ファイル内にその値が保持される。

ALUで作成された変数の値について、次の値が上書きされるまでに、最後に行われた値の参照を最後の参照とする。ここで変数とはプログラム・コード中で指定される論理レジスタの番号のこととする。変数値の作成後から最後の参照までの間に実行された命令数を変数の寿命とする。リオーダ・バッファのエントリのサイズよりも寿命が短い変数の値は、レジスタ・ファイルにリタイアされた後に値が読み出されることはない。実際のプログラム・コードによる例を図2に示す。寿命が短い変数が多数存在し、この検出を行うことができれば、参照されない値を書き戻すという無

© (社)情報処理学会

駄なデータの移動を削減することができる。これにより、レジスタ・ファイルを効率的に利用することが可能となり、資源競合による速度低下の影響も少なくする事ができる。

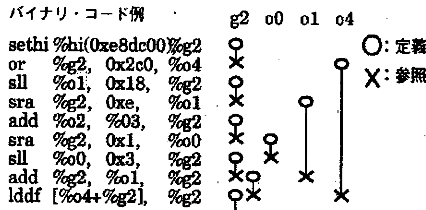

図 2　プログラム中の変数の寿命 (compress95)

文献[5]では、このような変数を Short-Lived 変数と定義し、値の書き戻しが不要な変数をコンパイラにより検出している。文献中で提案されている手法では、コンパイラが各変数について最後の参照を検出して寿命を求め、寿命の長さによって Short-Lived の判断をしている。Short-Lived 変数はシンボリック・レジスタという架空のレジスタに割り当てられ、無駄な物理レジスタ割り付けを削減している。

この文献中では、ベンチマークのトレースを入力としたシミュレーションを行い、レジスタへの書き戻しが不要になった値の割合が求められている。Short-Lived 変数の参照は、同じ基本ブロック中でのみの参照が許可されている。シミュレーションの結果によると、リオーダ・バッファのエントリ数を 32 としたとき、変数の約 85%〜99%が Short-Lived 変数であるという結果が示されている。

3.2　レジスタ・アクセス削減に関する関連研究

レジスタ・ファイルのアクセス削減については、他にもいくつかの研究が行なわれている。文献[6]によると、変数の参照は大抵 1 度だけであり、平均の参照回数は約 2 回である。これらの参照は 30〜40 命令内に行なわれており、Short-Lived 変数における検証結果と一致する。この文献では、スーパスカラ・プロセッサの実行機構内に命令をバッファリングすることで、50%のレジスタアクセスが削減できるという結果が示されている。

文献[7]では、無駄な書き込みが行なわれているレジスタをデッド・レジスタと呼び、独自に開発している SMT プロセッサにおいてデッド・レジスタを解放するようなコードと機構を備えた場合、2 倍程度の速度向上を得ることができると報告されている。

4.　VLDP アーキテクチャにおけるデータ・アクセスの削減

VLDP プロセッサでは大規模な複数パス投機実行を行なう必要があるため、パスの数に比例して増加する演算結果が生成されることになる。生成された全ての値をリタイアするためには、ALU-NET とリオーダ・バッファ間にスループットの大きなパスを構築し、全てのデータ移動を制御するハードウェアが必要となる。このような構成では、リタイアに関するコストが大幅に増大し、VLDP アーキテクチャの物理的な実現性が乏しくなる。

本章では、全てのデータ依存関係を ALU-NET 内で吸収できる変数について提案する。後続の命令に対する依存関係がないことを保証すれば、その命令が割り付けられた ALU についてリタイアのコストを削減することが可能である。

4.1　Closely-Accessed 変数

Short-Lived 変数の考え方では、レジスタ・ファイルへの書き込み量の削減はできるが、ALU-NET からリオーダ・バッファへの書き込みが依然存在する。

文献[8]では、ALU-NET のネットワークで吸収できるデータ・アクセスの割合が調査されている。理想的な ALU-NET で 80% 以上、現実的なモデルでも 60% 以上のデータの授受が ALU-NET 内で吸収可能であることが示されている。多くの変数の寿命が短いことを考えると、このような変数は演算機能ユニット間での近接したデータの授受にだけ用いられ、以降の機構への書き戻しが不要である。

本稿では、ALU-NET 外への無駄なデータ・アクセスの削減を目的として Closely-Accessed 変数を提案する。Closely-Accessed 変数とは、VLDP アーキテクチャにおける ALU-NET 内でデータの授受が完結し、外部の機構への書き戻しが不要となる変数である。

4.2　Closely-Accessed 変数の定義

この節では、本稿で用いている用語の定義を行う (図 3)。

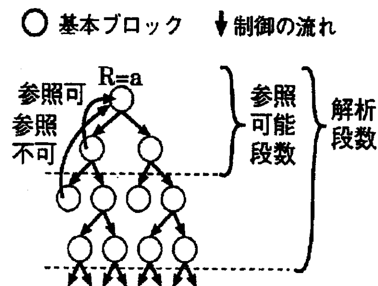

図 3　Closely-Accessed 変数解析における用語

VLDP アーキテクチャにおいては、大規模な投機的命令フェッチ機構の実現を前提としている。分岐命令は一定の割合で出現するため、フェッチされる命令の数は、投機的フェッチを行う際にまたいだ分岐命令数に比例している。そのため、命令間の距離としては命令数ではなく、またいだ分岐命令数をパラメータとした解析を行う。以降では、またいだ分岐命令数を命令間の距離と呼ぶ。

変数とは、コンパイラが割り付けるレジスタの番号

© (社)情報処理学会

を示す。本稿においてClosely-Accessed変数解析は、コンパイラを含めた静的な解析を前提としているため、レジスタ番号という言葉を避け、変数と呼ぶことにする。

解析段数 (al : Analyze Level) とは、ある命令を起点としてコントロールフロー・ツリーを解析した際、解析した制御パスの最大深さを表す。このとき、解析する制御パスは最大 2^{al} 本となる。

例えば、解析段数 0 の場合は、分岐命令を含まない 1 本の制御パスを解析しており、解析段数 3 の場合は、最大 8 の制御パスについて Closely-Accessed 解析を行っている。本稿での解析は静的に行っているため、$al=\infty$ とした全ての制御パスについて解析を行うことが望ましいが、解析にかかる時間の制約により $al=8$ を最大とした。

参照可能段数 (rl : Referable Level) とは、ある命令の出力先にあたる変数と依存関係のある命令が存在する時、ある命令を起点とした深さ rl までのコントロールフロー・ツリー中のパス (パス数は $2^{rl}-1$) が参照可能であることを示している。

Closely-Accessed パスとは、ある命令を起点とするコントロールフロー・グラフにおいて、その命令に対する参照可能段数を越えた参照が行われず、解析段数内で再定義が行われている制御パスがある場合の制御パスである。

本稿における Closely-Accessed 変数の定義としては次のようなものを用いた (図 4.2)。

「ある命令を起点とするコントロールフロー・ツリーにおいて、続く全てのパスがClosely-Accessed パスであると静的に保証された変数を Closely-Accessed 変数とする。」

また Closely-Accessed 命令とは、Closely-Accessed 変数を作成する命令とする。

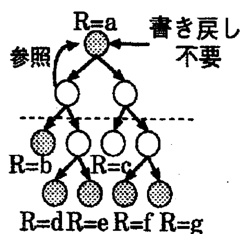

図4 Closely-Accessed となる変数

この定義によって検出された Closely-Accessed 変数は、動的に実行される全てのパスについて、Closely-Accessed であるという保障がなされる。ある命令の出力先にあたる変数の全てのデータ依存関係が ALU-NET で吸収できるとすれば、以後に続くどのパスが実行されるとしても、演算結果をリタイアする必要はない。

実際は、実行確率の高いパスだけについて値の破棄を保証したり、動的な解析による Closely-Accessed 予測等といった手法により、Closely-Accessed 変数の割合を増加させることも考えられ、これら手法によって検出された変数全てが Closely-Accessed であると言える。本稿における Closely-Accessed 変数とは、プログラムの性質によるものであり、評価に用いる機構の有無や規模によって Closely-Accessed 変数が制約をうけることはない。

4.3 Closely-Accessed 変数の検出

既存のコンパイラによって作成されたバイナリ・ファイルを静的に解析することによって、Closely-Accessed 変数の検出を行った。検出の際、Closely-Accessed 変数を参照可能な命令の個数は無限として理想化した。これは、寿命の短い変数は参照される回数が少ないこと、コンパイラに手を加えることによって参照回数の操作が可能であること、ALU-NET に対する命令割付機構の改善により、参照を可能とする命令の個数を増加させることが可能であること、との理由を考慮したためである。

5. Closely-Accessed 変数の検出結果と評価

本章では、汎用ベンチマーク・プログラムを解析し、実際に Closely-Accessed 変数を検出した結果について考察する。

5.1 解析プログラム

プログラムの解析は SPARC アーキテクチャ Version 7 の命令セットをターゲットとし、SPECint95 ベンチマーク・プログラムより go、cc1、compress、li、ijpeg、perl を対象として行った。コンパイルは Gnu C Compiler によって行い、オプションは -O のみを指定したバイナリ・コードを用いた。

リタイアが減少する割合として、Closely-Accessed 命令の実行回数をベンチマーク・プログラムのトレース・データより測定し算出する。トレース・データは表 1 に示す条件で作成し、先頭より 1 億命令を用いた。

表1 SPECint95 ベンチマークプログラム

Program	Description	Input Set	Static Inst
go	Plays the game of Go	50 21	84940
cc1	From GCC 2.7.1	2cp-decl.i	297206
compress	File compression	bigtest.in	28536
li	LISP interpreter	boyer.lsp	40663
ijpeg	Graphic compression	vogo.ppm	59381
perl	Manipulates strings	primes.in	89424

図5は、全実行命令数に対するリタイアが必要な命令数の割合である。図6は、Closely-Accessed 変数が参照された回数の平均値を示している。

5.2 Closely-Accessed 変数の検出結果

以下に Closely-Accessed 変数の検出結果を示す。図

学術情報センター電子図書館サービス
©(社)情報処理学会

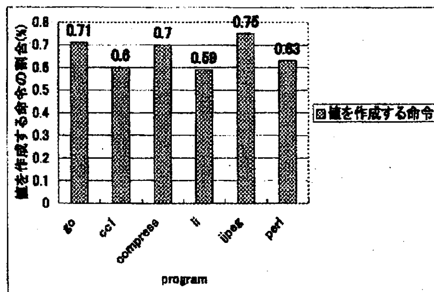

図5 値の書き戻しが必要な命令数の実行割合(1億命令中)

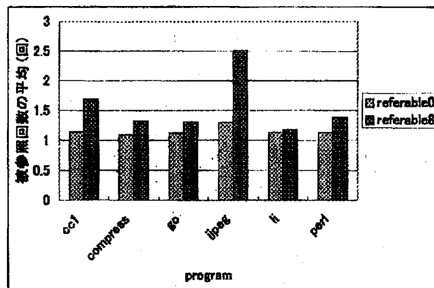

図6 Closely-Accessed 変数の被参照回数の割合(平均)

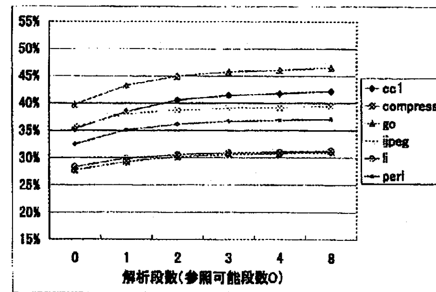

図7 Closely-Accessed 命令の静的割合(値作成命令中)(参照可能段数0)

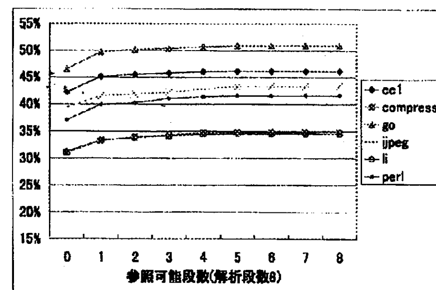

図8 Closely-Accessed 命令の静的割合(値作成命令中)(解析段数8)

7は、抽出したClosely-Accessed命令の静的な命令数の割合で、参照可能段数を固定し、解析段数を増加させた場合の変化である。横軸は解析段数を表し、縦軸はClosely-Accessed命令数の割合を表す。多くの分岐命令をまたいで解析を行うほど、より多くの命令がClosely-Accessed命令となっている。

図8は、図7同様、静的な割合で、解析段数を固定し、参照可能段数を増加させた場合の変化である。横軸は参照可能段数を表し、縦軸はClosely-Accessed命令数の割合を表す。分岐をまたいだ参照を可能とすれば、より多くの命令がClosely-Accessed命令となることがわかる。参照の多くは近くの命令から行われているため、参照可能段数をある程度まで大きくすると、Closely-Accessed変数の増加は飽和している。

図9は、実行トレースから算出したClosely-Accessed命令が実行された割合で、参照可能段数を固定し、解析段数を増加させた場合の変化である。横軸は解析段数を表し、縦軸はClosely-Accessed命令数の割合を表す。

図10は、図9と同様、実行割合で、解析段数を固定し、参照可能段数を増加させた場合の割合の変化である。横軸は参照可能段数を表し、縦軸はClosely-Accessed命令数の割合を表す。

測定した結果によると、解析段数を8とし基本ブロック内でのみ参照を許可した現実的な条件においては、静的な命令数の割合で21%~34%の命令についてClosely-Accessed命令であるとの検出ができた。動的なClosely-Accessed命令の実行割合を見ると、30% (perl) ~ 52% (ijpeg)のデータについてリタイアの削減が可能であることが分かった。解析段数を8として、参照可能段数を最大の8とした理想的な参照を可能とする場合には、41% (li) ~ 62% (ijpeg)のリタイアが削減可能となる。

5.3 Closely-Accessed変数の増加手法

基本ブロック内のみの参照を許可し、分岐をまたぐ参照はできないとした場合、書き戻しの削減量は30% (perl) ~ 52% (ijpeg)となる。分岐をまたぐ参照を可能とした場合、書き戻しの削減量は38% (perl) ~ 57% (ijpeg)となり、さらに約5~8%の削減を見込むことができる。連続して実行される複数パスの基本ブロック群を1つのフェッチ単位として静的にまとめておき、必ず同時にフェッチ・割り付けを行うというような、コンパイラサポートを考えることによって、分岐命令をまたいだ参照が可能となり、検出可能なClosely-Accessed変数が増加すると考えられる。

本稿では全てのパスでClosely-Accessedであるという条件でClosely-Accessed変数の定義を行なった。本提案手法は、実行時に起こる例外が最も少なくなるようにした安全側に立つ手法であると言える。言い換えると、リタイアを削減する技術を考えた場合に、演算結果の破棄の保証ができる最低値を示している。実際には、1度も実行されない、もしくは実行確率が非常に低いパスが存在する。このようなパスを含めて

学術情報センター電子図書館サービス　　© (社)情報処理学会

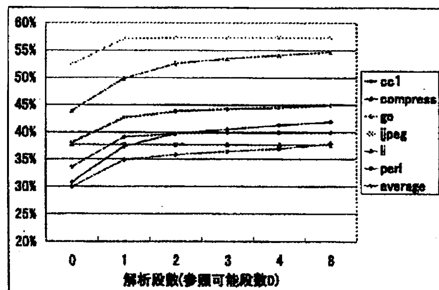

図9　削減可能なリタイア数の割合 (参照可能段数0)

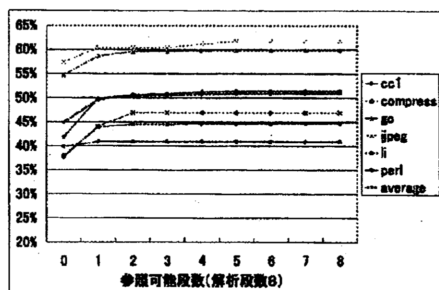

図10　削減可能なリタイア数の割合 (解析段数8)

Closely-Accessed を保証するのは、Closely-Accessed 変数の増加という視点では効率が悪い手法である。ソース・コードやトレース・データを解析時の参考にすれば、Closely-Accessed 変数を大幅に増加可能であると考えられる。その場合、どの程度の実行確率まで保証するか、Closely-Accessed が保証されていないパスが実行された場合のオペランドの供給について等、新たな問題を考慮する必要がある。

Closely-Accessed 変数を増加させる手法として、VLDP アーキテクチャ専用のコンパイラを用いることを考えれば、コンパイラによる Closely-Accessed 変数の情報付加が可能である。その場合、前述したパスの実行確率による Closely-Accessed の保証等もこれに含めることができる。また、最定義を検出することなく変数の寿命がわかるため、最後の参照が行われているにもかかわらず、解析範囲外で最定義されている変数についても Closely-Accessed 変数とすることができる。

6. 結　　論

本稿では、多くの命令レベル並列性を取り出す際に障害となるデータ・アクセスについて、データのリタイアを削減できる Closely-Accessed 変数を提案し、実際のプログラム・コードを解析して Closely-Accessed 変数の検出を行なった。その結果、静的な命令の割合で 21%～34% の命令が Closely-Accessed 命令であるとの検出結果を得た。また、動的に実行される Closely-Accessed 命令の割合をトレースデータを参照することで求めた結果、最も安全側に立った条件でもプログラムによっては半数以上のデータについてリタイアの削減が可能であると分かった。さらに、Closely-Accessed 変数を増加させるための手法について考察を行なった。今後の課題としては、コンパイラを利用した Closely-Accessed 変数の増加と、Closely-Accessed 変数を利用するためのハードウェア構成も含めた例外回復機構の考案が挙げられる。

参　考　文　献

1) 中村友洋, 吉瀬謙二, 辻秀典, 安島雄一郎, 田中英彦. 大規模データパスプロセッサの構想. 情報処理学会研究会 ARCH, Vol. 124, No. 3, pp. 13–18, June 1997.
2) 辻秀典, 中村友洋, 吉瀬謙二, 安島雄一郎, 高峰信, 坂井修一, 田中英彦. ALU-Net: VLDP アーキテクチャにおける命令実行機構. 情報処理学会 第57回全国大会, Vol. 1, No. 1Q-10, October 1998.
3) 中村友洋, 吉瀬謙二, 辻秀典, 安島雄一郎, 田中英彦. 分岐アドレス予測機構の比較検討. 情報処理学会 第55回全国大会, Vol. 1, No. 3F-5, pp. 20–21, September 1997.
4) 辻秀典, 中村友洋, 吉瀬謙二, 安島雄一郎, 田中英彦. 大規模データパスプロセッサにおけるフェッチ機構の検討. 情報処理学会研究会 ARCH, Vol. 126, No. 7, pp. 37–42, October 1997.
5) Luis A.Lozano C. and Guang R.Gao. Exploiting short-lived variables in superscalar processors. *In Proceedings of 28th MICRO*, pp. 292–302, November 1995.
6) M.Franklin and G.Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. *In Proceedings of 25th MICRO*, pp. 236–245, 1992.
7) Jack L.Lo, Sujay S.Parekh, Susan J.Eggers, Henry M.Levy, and Dean M.Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *University of Washington Technical Report*, December 1997.
8) 吉瀬謙二, 中村友洋, 辻秀典, 安島雄一郎, 田中英彦. Alu-net を用いることによるデータ移動の効率化. 情報処理学会 第56回全国大会, Vol. 1, No. 2N-1, pp. 109–110, March 1998.

学術情報センター電子図書館サービス　　© (社)情報処理学会

計算機アーキテクチャ　133－7
（1999．5．21）

複数パス投機実行のためのレジスタセット管理方式

安島 雄一郎, 坂井修一, 田中 英彦
東京大学大学院 工学系研究科

本論文では大規模な複数パス実行に適したレジスタセット管理方式として分散フューチャファイル・モデルを提案する。本方式ではレジスタ値出力に複数サイクルを要する場合があり、この遅延がプロセッサ性能に与える影響をシミュレーションによって評価した。結果として部分フューチャファイル数4〜8程度のクラスタ化によって性能低下を抑えられることを示した。

Register Management System for multi-path execution

Yuichiro AJIMA, Shuichi SAKAI, Hidehiko TANAKA
University of Tokyo, Graduate School of Engineering

We proposed the Distributed Future File Model; a register management system for multi-path execution. By using this model in multi-path execution processors, in particular case, it may spent multi-cycle delays before the value of register become available. We evaluated the decrease of performance caused by this penalty. The results showed that by clustering partial future files by number of 4 to 8, most amount of this penalty can be reduced.

1　はじめに

複数パスの投機的実行により、分岐予測失敗によるペナルティの削減と高い命令レベル並列性の取り出しが可能であることが指摘されている[2][7]。単一パスでは1つの実行状態のみを保持することで実行が可能であった。しかし、複数パス実行では分岐命令において実行状態を複製し、それぞれ別の実行パスに割り当てて並列に演算を進める必要がある。近年、これを可能にする複数パス実行機構に関する研究が盛んに行なわれている[4][5]。本研究では、複数実行状態のレジスタセットを部分フューチャファイル[8]によって管理する、分散フューチャファイル・モデルを提案する。また、部分フューチャファイルに分割することで生じるデータ転送に関する遅延が、プロセッサ性能に与える影響を評価する。

2　分散フューチャファイル

2.1　実行ステート

out-of-order実行を行なうプロセッサでは、例外回復機能を備えたレジスタ管理が要求される。例外回復を行なうためには、プロセッサは2つの実行ステートを保持する必要がある。1つは例外回復のためのインオーダ・ステートである。インオーダ・ステートは連続した完了命令だけによって作られたプロセッサの状態である。このステートは実行が確定した状態であるので、これ以前の状態を保存する必要はない。もう1つはオペランド供給を行なうアーキテクチャ・ステートである。このステートは命令の演算結果や、演算結果の格納先として予約されているエントリのタグを含んだ最新のステートである。

2.2　部分フューチャファイル

完全なアーキテクチャ・ステートを保持するレジスタファイル(フューチャファイル)を

© (社)情報処理学会

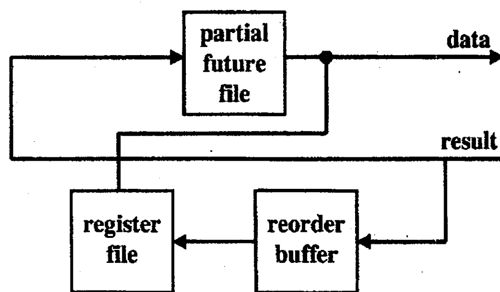

図 1: 部分フューチャファイル

用いた場合、回復動作においてインオーダ・ステートからアーキテクチャ・ステートへのレジスタファイルのコピーを必要とする。これを不要とするため、完全なアーキテクチャ・ステートを保持するのではなく、新しい演算結果のみを保持し、インオーダ・ステートの結合でアーキテクチャ・ステートを作る方式(図1)が提案されている[1]。この場合フューチャファイルからはインオーダ・ステートに含まれない演算結果や保留中のタグをオペランドとして供給し、レジスタファイルからはインオーダ・ステートからアーキテクチャ・ステートの間に更新されていないレジスタ値をオペランドとして供給する。以降ではこの新しい演算結果のみを保持するレジスタファイルを部分フューチャファイルと呼ぶ。

3 分散フューチャファイル・モデル

複数パス実行で高い IPC を取り出すためには、広範囲な命令から実行可能命令を取り出す大規模な投機的実行が必要になる。しかし、既存の実行機構では投機的実行の大規模化は難しい。これは各制御ユニットが単一であるため命令解析範囲の拡大に伴い、オペランド出力、フォワーディング、演算器制御が複雑になるためである。これを克服するにはオペランド供給、フォワーディング、演算器制御の分散化が必要となる。本章では部分フューチャファイル(Partial Future File: PFF)を複数使用することで、複数パス実行可能でかつ大規模投機的実行に適した新しい例外回復・オペランド供給機構、分散フューチャファイル・モデルを提案する。

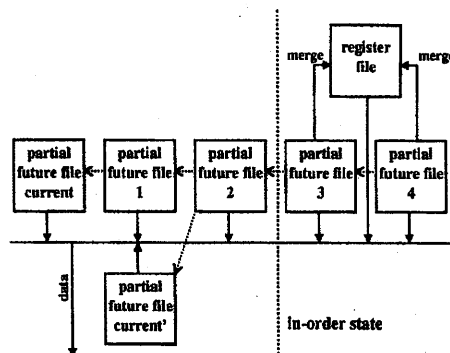

図 2: 分散フューチャファイルによるオペランド供給のモデル

分散フューチャファイルによるオペランド供給のモデルを図2に示す。ここで実線矢印はデータの流れを、破線矢印は命令実行順序の関係を表す。この機構ではオペランド供給は複数の部分フューチャファイルから行なわれる。図2の例では、部分フューチャファイル current の実行パス上の命令には、current 及び部分フューチャファイル 1 〜 4、レジスタファイルからオペランドが供給される。また、部分フューチャファイル current' の実行パス上の命令には、current' 及び部分フューチャファイル 2 〜 4、レジスタファイルからオペランドが供給される。

部分フューチャファイルは分岐命令が現われる毎に、その先の分岐パスに新たに割り当てられる。命令の演算結果はその分岐パスに割り当てられた部分フューチャファイルのみを更新する。

また、インオーダ・ステートはレジスタファイルといくつかの部分フューチャファイルで保持する。インオーダ・ステートに達した部分フューチャファイルは、その内容を出力してレジスタファイルに更新する。レジスタファイルを更新し終った部分フューチャファイルは捨てられる。部分フューチャファイルは分岐パス単位でレジスタファイル更新値を持つため、リオーダバッファのようにインオーダ・ステートを更新するために命令毎の演算結果を保存する必要はない。

例外回復は該当分岐パスまで実行を進めた後、その分岐パスの実行をやり直すことで実現される。分岐は新たな部分フューチャファイルの割当てで処理されるためやり直しが生じるのは割り込み処理や例外処理の場合のみ

 © (社)情報処理学会

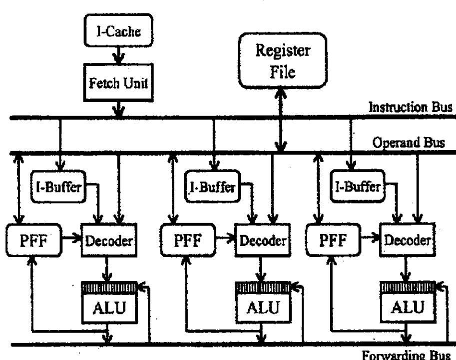

図 3: 分散フューチャファイルによるプロセッサの基本構成

である。このためやり直しによる回復が必要となる頻度は低く、やり直しのオーバーヘッドが性能に与える影響は小さいと考えられる。

このモデルによって大規模な投機的実行を行なうプロセッサを構成する利点を考える。単純に従来の機構を大規模化する場合、IPC向上に伴って大量に発生する演算結果の処理が問題となる。このモデルでは、演算結果は割り当てられた部分フューチャファイルのみを更新するため、投機的実行の規模を上げても部分フューチャファイルそれぞれの規模を上げる必要がない。また、分岐命令によって実行状態が複数に分かれる場合にアーキテクチャ・ステートのコピーを必要としない。一方、演算結果を命令の順序によって複数の部分フューチャファイルに保持するため、レジスタ値出力の制御に複数サイクルかかる場合がある。

3.1 プロセッサ構成

前節で述べたモデルに従った基本的なプロセッサ構成を図3に示す。フェッチ・ユニットは命令バスを通じて各命令バッファに命令を送り、デコーダは命令バッファから命令を取りだし、近傍の部分フューチャファイルからオペランドを得る。近傍の部分フューチャファイルからオペランドが得られない場合は、オペランド・バスを通じ遠隔の部分フューチャファイルまたはレジスタファイルからオペランドを得る。その後、デコードした命令を命令ウィンドウに送る。演算器は命令ウィンドウを監視し、データが揃った命令から演算を行なう。演算結果は近傍の部分フューチャファイルとフォワーディング・

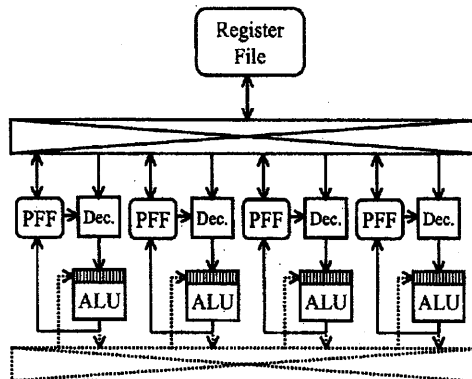

図 4: 分散フューチャファイルによるオペランド供給時のデータパス

バスに送られる。命令ウィンドウはフォワーディング・バスを監視し、演算結果をエントリに取り込む。

この構成では簡単のため各部分フューチャファイル命令バッファ、命令デコーダ、演算器を割り当てているが、PFF の結合(後述)などによりこれらのユニットは共有することが可能である。

図4に、分散フューチャファイルによるプロセッサ構成でのデータパスを示す。実線はオペランドのデータパス、破線はフォワーディングのデータパスである。なお、分散フューチャファイルではオペランドのデータパスを扱うため、フォワーディング・バスの構成にはこれとは別の機構を考慮する必要がある。図3では部分フューチャファイル間はバスで接続していたが、ここでは一般化してネットワーク接続としている。これは投機実行の規模が大きくなって実行中の分岐パス数が増えると、必要な部分フューチャファイルの数も増えるためバスが駆動できなくなるためである。大規模投機的実行においては部分フューチャファイル間のオペランド転送には階層化されたバスやクロスバ・スイッチなどの接続網が必要になると考えられる。

3.2 クラスタ化

分岐パスの平均の長さは predicate などによって分岐命令が除去されても数命令程度である [3][6]。このため、1つの分岐パス毎に部分フューチャファイルを割り当てた場合、頻繁に別の部分フューチャファイルからレジ

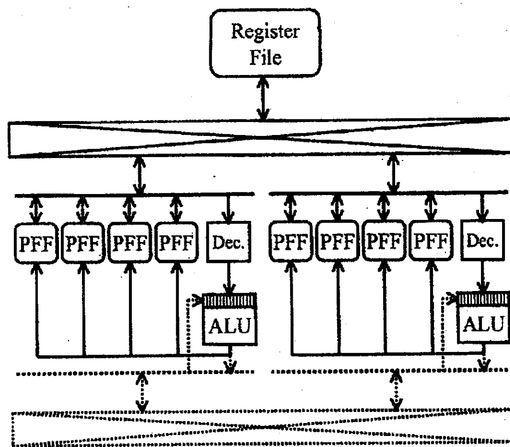

図 5: 4-PFF クラスタの例

スタ値を読み出すことになり、ペナルティが大きい。そこで部分フューチャファイルをクラスタ化し、同クラスタ内では1サイクルで処理を行なう。クラスタの規模を上げることによりレジスタ値読み出しの遅延を削減できるが、ハードウェアの複雑さは上がる。4つの部分フューチャファイルによるクラスタの例を図5に示す。クラスタ内部では演算器やパスを共有できる。部分フューチャファイルをマッピングテーブルと物理レジスタファイルに分けて実装する場合、物理レジスタファイルを共有することも可能である。またクラスタが小規模であれば、部分フューチャファイルにチェックポイント回復機能を追加することで、クラスタ内部の部分フューチャファイルを1つに統合することが可能になる。また、クラスタに連続した分岐パスを割り当てる場合、1つのクラスタは単一パス実行と同様の動作になるため、設計が容易になる。

4 評価

分散フューチャファイルでは部分フューチャファイルを分散配置する。このため遠隔の部分フューチャファイルからオペランドを供給する時に、レイテンシが複数サイクルかかることが考えられる。このレイテンシがプロセッサ性能に与える影響を評価する。評価はトレースベースのシミュレータを作成して行った。シミュレート対象は命令セットをSPARC Version8、OS を Solaris 2.5.1、実行プログラムはSPECint95 より compress, gcc, go, ijpeg, li, perl とし、それぞれ10,000,000命令実行した。また、シミュレートした CPU の設定は7ステージ・パイプライン、8命令同時デコード、分岐予測は2bit, 1024エントリ, 4way set associative、ストアバッファ8本で、メモリは1次キャッシュ 64KB, 256bit/line, 4way set associative、2次キャッシュ 1MB, 256bit/line, direct map, 5サイクル／アクセス、メインメモリ 50サイクル／アクセスであった。

シミュレータは static tree heuristic[2] による複数パス実行制御を行ない、命令フェッチ、フォワーディングパスに関しては大規模化に伴うペナルティはないものとした。シミュレーションは表1に示す5種類の投機的実行規模について行ない、オペランド供給のレイテンシを変えてIPCの変化を評価した。

オペランド供給のレイテンシは部分フューチャファイルのクラスタ内では0とする。また、部分フューチャファイルはプログラムの命令順に割り当てられるとし、レイテンシは演算結果を持つ部分フューチャファイルとオペランドを要求する部分フューチャファイル間のクラスタ距離に比例するとした。ここでは1クラスタにつき1サイクルとした。即ち演算結果を出力した命令からオペランドを要求する命令の分岐パス距離を部分フューチャファイル・クラスタの結合数で割ったものがレイテンシ・サイクル数となる。なお、同じ部分フューチャファイルで一度要求されたオペランドは部分フューチャファイルに保存され、2回目以降はレイテンシなしとする。

表 1: 投機実行の規模

主投機数	副投機数	PFF 数
7	0	8
15	1	17
31	3	38
47	7	76
63	15	184

4.1 結果

図6に compress でのシミュレーション結果を示す。図7,8,9,10,11はそれぞれ gcc, go, ijpeg, li, perl の結果である。横軸は部分フューチャファイル・クラスタの結合数である。最小の1では親の分岐パスでの出力を参照する場合も1サイクルのレイテンシがあり、10分岐パス離れた演算結果をオペラン

© (社)情報処理学会

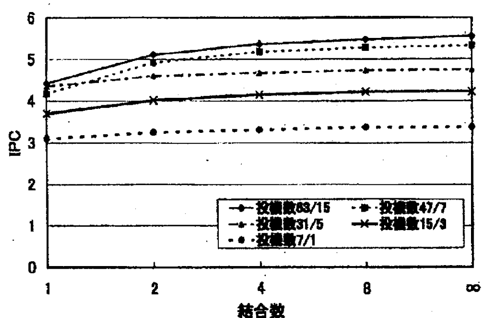

図 6: 結果: compress

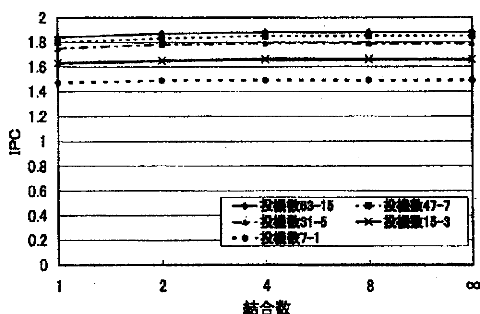

図 7: 結果: gcc

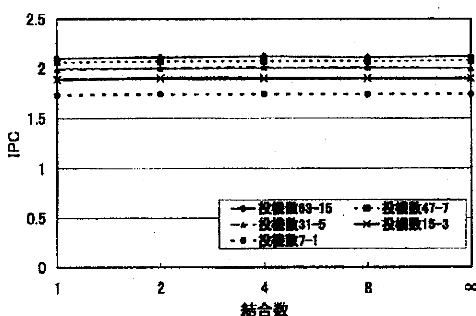

図 8: 結果: go

ドとする時は 10 サイクルのレイテンシが発生することになる。一方、無限大では他の部分フューチャファイルからオペランドを供給する時にもレイテンシが発生しない。これは即ち、全てのクラスタが 1 つのバスに接続されている等の構成である。

5 考察

compress, li, perl の大規模な投機的実行で性能が大きく向上するアプリケーションでは、部分フューチャファイル・クラスタ間のデータ転送レイテンシが大きい場合実行速度が低下することが分かる。特に結合数 1 では最大で compress の 20.1% と大幅に性能が

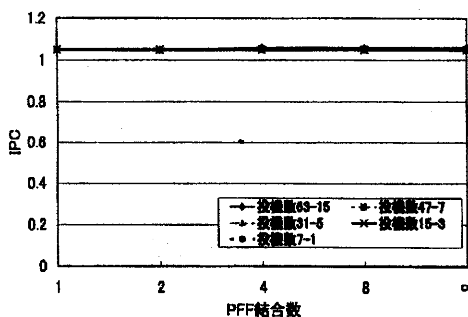

図 9: 結果: ijpeg

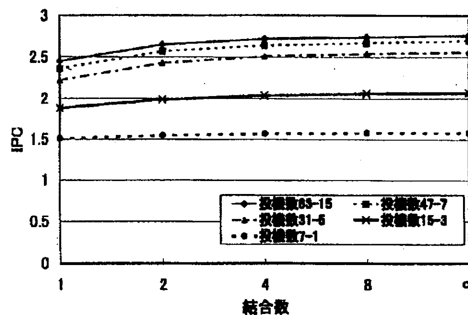

図 10: 結果: li

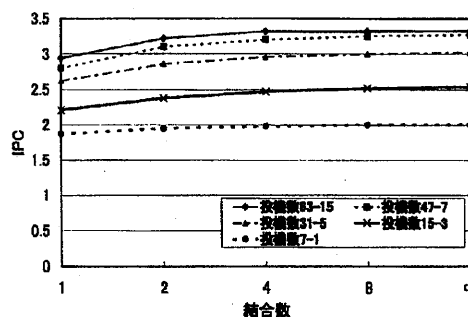

図 11: 結果: perl

© (社)情報処理学会

低下している。しかし、結合数 4 では最大 3.5%、平均 1.0% 程度、結合数 8 では最大 1.5%、平均 0.5% 程度の性能低下となり、結合数が大きくなると性能低下は著しく減る。一方、gcc, go, ijpeg といった大規模な投機的実行での性能向上が小さいアプリケーションでは、オペランド供給のレイテンシも性能に与える影響は少ない。

これより、大規模な投機的実行を行なう場合、部分フューチャファイル・クラスタの結合数を 4,8 程度にすればオペランド供給に関するレイテンシがプロセッサ性能を低下させる割合は低いことが分かった。結合数 4, 8 程度の部分フューチャファイル・クラスタは、大規模な投機的実行全体を制御する一つのユニットに比べて、非常に複雑さが小さい。また、投機実行の規模を上げる際もクラスタを増やすだけで良いので比較的容易である。評価結果より、結合数 4, 8 程度の部分フューチャファイル・クラスタは大規模投機的実行のオペランド供給機構として適していることが確認できた。

6 今後の課題

大規模に複数パスを実行するには本章で提案した分散フューチャファイル・モデルのみでは不十分であり、未解決の問題が残されている。1つは複数パス実行導入に伴って、レジスタ及びそのタグの生存時間を制御する方法である。もう1つは、大規模に投機的実行することに伴ってフォワーディングによるオペランド供給の制御が複雑になる問題である。オペランド供給は制御依存に従って分散化する機構が適していることを示したが、データ依存に従うフォワーディングについては異なる方式が必要になる。

VLDP プロジェクトでは多数の演算器をネットワーク的に接続した ALU-Net に関する研究を行なっている [7]。ALU-Net ではフォワーディングを行なう命令の対をハードウェア的に近い演算器に割当て、直接データ転送を行なう。これにより、演算結果出力時に共通バスの制御を行なう頻度が激的に減らせることが期待できる。今後の課題として、分散フューチャファイルによるオペランド供給機構と ALU-Net を組み合わせ、大規模に投機的実行を行なうプロセッサのデータパス構成手法を確立することが挙げられる。

7 まとめ

本論文では大規模な複数パス実行に適したレジスタセット管理方式として分散フューチャファイル・モデルを提案した。また、データ転送による遅延がプロセッサ性能に与える影響を評価し、部分フューチャファイル数 4 〜 8 程度のクラスタ化によって性能低下を抑えられることを示した。

本研究の一部は（株）半導体理工学研究センターからの補助による。

参考文献

[1] Mike Johnson: "*Superscalar Microprocessor Design*", Prentice-Hall (1991)

[2] Augustus K.Uht and Vijay Sindagi: "Disjoint Eager Execution: An Optimal Form of Speculative Execution", *MICRO-28*, pp.313-325 (1995)

[3] S. A. Mahlke and R. E. Hank and J. McCormick and D. I. August and W. W. Hwu: "A cmoparison of full and partial predicated execution support for ILP processors.", Proc. 22th Annual International Symposium on Computer Architecture, pp.138-150 (1995)

[4] Tien Fu Chen: "Supporting Highly Speculative Execution via Adapive Branch Trees.", The 4th International Symposium on High-Performance Computer Architecture, pp.185-194 (1998)

[5] Artur Klauser, Abhijit Paithankar, Dirk Grunwald: "Selective Eager Execution on the PolyPath Architecture.", Proc. 25th Annual International Symposium on Computer Architecture, pp.250-259 (1998)

[6] David I. August et al: "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture.", Proc. 25th Annual International Symposium on Computer Architecture, pp.227-237 (1998)

[7] 中村 友洋, 吉瀬 謙二, 辻 秀典, 安島 雄一郎, 田中 英彦: "大規模データパスプロセッサの構想", 情報処理学会研究報告 97-ARC-124, pp.13-18 (1997)

[8] 安島 雄一郎, 中村 友洋, 吉瀬 謙二, 辻 秀典, 田中 英彦: "例外回復可能な複数パス実行機構の提案", 情報処理学会研究報告 98-ARC-129 (1998)