クリティカルパス情報を用いた分散命令発行型マイクロプロセッサ向けステアリング方式

服部直也[†,☆] 高田正法[†] 岡部淳[†]
入江英嗣[†,☆☆] 坂井修一[†] 田中英彦[†,☆☆☆]

マイクロプロセッサの高クロック動作と高 IPC を両立させるために，小規模な演算資源の塊を間接的に接続したクラスタアーキテクチャが提案されている．このアーキテクチャでは，個々のクラスタは高速に動作する．しかしクラスタの命令発行幅が狭いため，命令を適度に分散しなければ高 IPC は得られない．その一方で，クラスタ間の信号転送遅延が小さくないため，データ依存のある命令を別クラスタに分散すれば IPC は低下してしまう．したがって，命令をクラスタに割り当てる，命令ステアリングには，データ依存と負荷分散の適切なバランスをとることが求められている．これまでの研究では，クラスタの負荷情報を指標としてバランスをとる方式が提案されていた．しかしこの方式は，データ依存に関係なく命令を分散させる点に非効率性が存在していた．そこで本論文では，クリティカルパスに属する重要な命令の分散を避け，重要でない命令を用いて予防的に負荷分散を行う方式を提案する．また，クリティカルパス情報だけでは判断が適切でない状況に着目し，これを回避するために，洗練された負荷指標を併用する方式を提案する．

Critical Path Based Steering Algorithms for Clustered Microprocessor Architectures with Distributed Issue Logic

Naoya Hattori,[†,☆] Masanori Takada,[†] Jun Okabe,[†]
Hidetsugu Irie,[†,☆☆] Shuichi Sakai[†] and Hidehiko Tanaka[†,☆☆☆]

To achieve both high clock rate and high IPC of microprocessors, clustered architecture has been proposed. In this architecture, small number of computation resources is clustered, and the clusters are interconnected each other. Each of them performs at high clock rate, but has only narrow throughput. Therefore, instruction workload of each cluster should be balanced for high IPC, On the other hand, dependent instructions should be steered into the same cluster, to avoid inter-cluster communication delay. To achieve high IPC, data dependence first strategy and load balance first strategy should be selected properly. Therefore, the steering algorithm to decide the strategy by workload information has been proposed. But this method has inefficiency because data dependence is always ignored while workload is unbalanced. In this paper, Criticality Steering is proposed. This steering algorithm prevents to ignore any critical data dependence, and uses all non-critical instructions for load balancing. Then we propose another steering algorithm with critical path and workload information, for the condition that critical path information alone does not work well.

1. はじめに

近年プロセッサの性能は目覚ましく向上しているが，高性能化への要求はとどまるところを知らない．近年のプロセッサは主に，パイプラインを深く設計する Deeper Pipeline 技術と，半導体技術の微細化による高クロック化によって性能を向上させてきた．プロセッサの性能は動作クロックと IPC（Instruction Per Cycle）の積で求められるため，性能向上のためには両者のバランス良い改善が望ましい．しかし実際には，動作クロックを向上させると IPC は低下する傾向にある．

パイプラインを深くすると各処理に必要なサイクル数が増加し，依存関係にある命令の発行間隔が大きく

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
☆ 現在，日立製作所中央研究所
Presently with Hitachi, Ltd., Central Research Laboratory
☆☆ 現在，科学技術振興機構
Presently with Japan Science and Technology Agency
☆☆☆ 現在，情報セキュリティ大学院大学
Presently with Institute of Information Security

なるために，IPC が低下する[5),14)]．また半導体技術の微細化に関しては，今後ゲート遅延に対して配線遅延が支配的になることが知られている．アーキテクチャ設計者は，処理遅延の増加を容認するか，演算資源を減らして遅延増加を抑えるかの選択を迫られるが，いずれにしても IPC が低下すると考えられている[1)]．

IPC の低下要因は，制御依存による発行間隔の拡大，レジスタデータ依存による発行間隔の拡大，メモリデータ依存による発行間隔の拡大，の 3 種に大別されるが，最も影響が大きいのはレジスタデータ依存である[5),14)]．そこでレジスタデータ依存に関係する機構として，演算器間のデータフォワーディング機構と命令発行機構（IQ: Issue Queue）に関する配線遅延が重要視されており，これまでに様々な研究がなされている[3),5),8)~10),13),15)]．本研究ではその中の，クラスタアーキテクチャに着目する．

Alpha 21264[6),7)] に代表されるクラスタアーキテクチャでは，演算資源をクラスタリングすることで IPC 低下問題を改善している．IPC に大きな影響を与える，演算器間のデータフォワーディング遅延に関しては，少数の演算器の塊（クラスタ）を形成することで対処する．クラスタ内の演算器は少数であるため，フォワーディングの距離を短くすることが可能であり，遅延を抑えられる．しかし，個々のクラスタのスループットは小さい．そこで複数のクラスタを用意して，それらを間接的に接続することで，プロセッサ全体のスループットを確保する．

命令発行時には，後続命令のオペランドが揃ったこと（Wakeup）の判断のために，全 IQ エントリに発行された命令のタグを送信するが，この命令タグ転送遅延も IPC に与える影響が大きい．そこで，命令発行機構に対してもデータフォワーディングと同様に，クラスタ化することが有用である．本研究ではそのような分散発行方式[3),8)] を仮定する．分散発行方式では，発行待ち命令を保持する IQ エントリをクラスタリングして，小規模な IQ を構成する．そしてその小規模 IQ を演算器クラスタと 1 対 1 に対応させる．この方式では IQ に対応する演算器数が減少するために，演算器に合わせて発行命令を選択する Select 処理も高速化される．

クラスタアーキテクチャでは命令をクラスタに割り当てる，命令ステアリングの質がその性能を左右する．

クラスタアーキテクチャでは，クラスタ内の信号転送は短い配線で高速に行われるのに対し，クラスタ間の信号転送は配線長が長く，遅延が大きい．そのため命令を，データ依存関係にある先行命令と同じクラスタへ割り当てることで，クラスタ間通信に起因する Wakeup 遅延を回避することができる．

また，個々のクラスタのスループットが小さいために，特定のクラスタに命令が集中すると命令が Wakeup していても Select されない，Select 遅延が発生する．この遅延は，命令を IQ 内の命令が少ないクラスタへ割り当てることで回避することができる．

このように命令をクラスタに割り当てる，命令ステアリングは，Wakeup 遅延と Select 遅延のバランスをとらなければならない，難しい問題である．

これまで，Wakeup 遅延回避/Select 遅延回避のいずれかに特化した命令ステアリング方式と，クラスタ間の負荷均衡状態を用いて両者を使い分ける折衷方式が提案されている．それらの中では折衷方式が IPC 的に最も優れているが，負荷集中が発生している場合にいっさいのデータ依存関係を無視する点に，非効率性が存在していた．

そこで本研究では，高負荷クラスタへのデータ依存だけを無視するための負荷指標を検討する．また別のアプローチとして，クリティカルパス[4),16)] に属する重要なデータ依存は負荷にかかわらずつねに最優先し，重要でない命令だけを用いて予防的に負荷分散を行う方式を提案する．最後に，クリティカルパス情報だけでは判断が適切でない状況に着目し，これを回避するために負荷指標を併用する方式を提案する．

本論文は以下のように構成される．2 章では命令ステアリングに関する関連研究を紹介し，3 章では本論文で想定するアーキテクチャと評価環境について説明する．4 章で関連ステアリングの性能を解析し，それを受けて 5 章ではより優れた負荷指標と，クリティカルパス情報を用いた方式を提案，評価する．最後に 6 章で全体をまとめる．

2. 関 連 研 究

本章ではこれまで研究されてきた，命令ステアリング方式を紹介する．

命令ステアリングには Wakeup 遅延回避と Select 遅延回避の 2 つの側面がある．クラスタアーキテクチャではクラスタ間に信号転送の遅延（通信遅延）が存在するため，データ依存関係のある命令を別のクラスタにステアリングすると Wakeup が遅れてしまう．このような Wakeup 遅延を避けるためには，データ使用命令を，データ生成命令と同じクラスタ（オペランドクラスタ）にステアリングすればよい．このようなステアリング戦略を本論文では，OP-戦略（OP は Operand Producer の略）と呼ぶ．また，クラスタ

アーキテクチャでは，各クラスタの命令発行幅が狭くなるため，同一クラスタに命令が集中すると Select が遅れてしまう．このような Select 遅延を避けるためには，命令を最も命令数の少ないクラスタにステアリングすればよい．このようなステアリング戦略を本論文では LW-戦略（LW は Low Workload の略）と呼ぶ．

Modulo-N Steering

Select 遅延の回避に特化した命令ステアリングとして，Modulo Steering が考案されている[2]．Modulo Steering では，命令を Round-Robin に各クラスタに割り当てることで，命令集中を回避する．この方式ではクラスタ数を C とすると，X 番目にフェッチされた命令は，[X mod C] 番のクラスタへステアリングされる．

ただし，これだけでは Wakeup 遅延が大きくなりやすいため，クラスタ間の通信遅延が大きい場合には不向きである．そこで，データ依存のある命令はフェッチ順的に比較的密集していることを利用して，連続した N 命令の塊を Round-Robin で割り当てる，Modulo-N Steering も考案されている．この方式では X 番目にフェッチされた命令は，[X/N mod C] 番のクラスタへステアリングされる．N の値はクラスタアーキテクチャの構成に応じて，発見的に選ばれる．

Dependence Based Steering

Wakeup 遅延の回避に特化した命令ステアリングとして，データ依存を重視した Dependence Based Steering が考案されている[2]．そのアルゴリズムを**図 1** に示す．このステアリング方式では，対象命令に未解決のオペランドが存在する場合はつねに OP-戦略を適用する．この際，未解決オペランドが複数存在する場合には，その中から任意のオペランドを選択する．また，未解決オペランドが存在しない場合は，Select 遅延を回避するために LW-戦略を適用し，最も命令数が少ないクラスタを選択する．このステアリングは OP-戦略を基本としているため，特定のクラスタに命令が集中しやすい．そのため，クラスタの発行幅が狭い場合には不向きである．

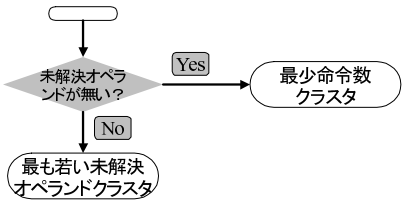


図 1 Dependence Based Steering のアルゴリズム
Fig. 1 Algorithm of Dependence Based Steering.

Focused Steering

Dependence Based Steering の改良方式として，クリティカルパス情報を用いた Focused Steering が提案されている[4),16)]．クリティカルパスとは，プログラムの実行時間を規定する，最も長いデータ依存のフローである．両文献では，プログラム中のクリティカルパスをヒューリスティックを用いて発見する方式が提案されているが，Focused Steering ではこの情報に基づいて，クリティカルパスの延長を避けるステアリングを行う．

Focused Steering のアルゴリズムを**図 2** に示す．この方式は基本的に Dependence Based Steering と同じ動作をするが，クリティカルパス上の命令の未解決オペランドが複数存在する場合には，クリティカルパスに属する重要なデータ依存を優先し，重要な Operand Producer と同じクラスタに割り当てる．この動作により，クリティカルパス中に通信遅延が追加されることを回避できる．

Parcerisa らの Steering

Wakeup 遅延と Select 遅延の双方の影響を避けるために，Parcerisa らはクラスタ間の負荷バランスに着目した折衷方式を提案している[11),12)]．そのアルゴリズムを**図 3** に示す．この方式では，負荷が均衡している場合はどのクラスタにも Select 遅延に差がないと判断して OP-戦略を採用し，負荷集中が発生している場合は Select 遅延を重視して LW-戦略を採用する．

彼らは負荷を把握するために，まず IQ 内の Wakeup 済み命令数である NREADY という指標を検討した．しかし，命令ステアリングの結果が NREADY に反映されるには時間を要するため，その間に NREADY が最小だったクラスタに命令が集中する可能性がある．これを避けるために，彼らは DCOUNT という近似

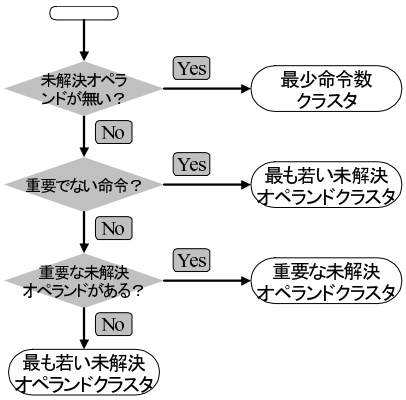


図 2 Focused Steering のアルゴリズム
Fig. 2 Algorithm of Focused Steering.

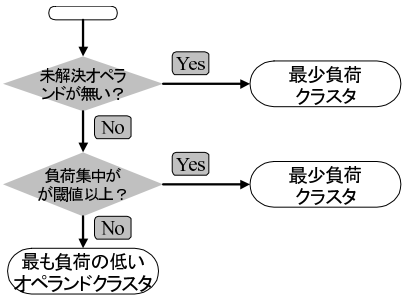


図 3 負荷情報を用いる命令ステアリングのアルゴリズム
Fig. 3 Steering algorithm using workload information.

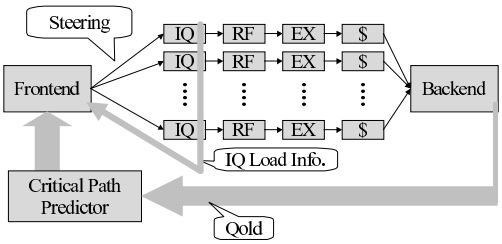


図 4 想定するアーキテクチャの構成
Fig. 4 Target architecture.

指標を採用した．彼らが用いた指標 DCOUNT は各クラスタの状態を示す符号付整数値で，命令がそのクラスタにステアリングされた場合に [クラスタ数 − 1] だけ増加し，他のクラスタにステアリングされた場合に 1 だけ減少する．負荷状態を判断する際には，最大の DCOUNT 値が閾値よりも大きければ負荷集中と見なし，小さければ負荷は均衡していると見なす．また，DCOUNT が最少のクラスタを最低負荷クラスタと見なす．

またこの方式では，複数オペランド命令に対するオペランドクラスタを選択する際にも，DCOUNT が最も小さいクラスタを選択する．

3. 本研究で想定するプロセッサ構成と評価環境

3.1 本研究で想定するプロセッサ構成

本研究で想定するアーキテクチャの構成を**図 4** に示す．想定するアーキテクチャは Frontend，分散クラスタ，Backend，クリティカルパス予測器から構成される．Frontend では命令のフェッチ，デコード，レジスタリネーミングを行い，分散した IQ への命令ステアリングを行う．IQ からキャッシュまではクラスタに含まれており，レジスタとキャッシュはそれぞれが完全な複製を保持する．Backend は主にリタイア処理を行い，リタイア命令の情報をクリティカルパス予測器に通知する．クリティカルパス予測器は Frontend に対して予測情報を与える．このほかにも Frontend へは，ステアリングの選択や IQ が不足した場合の stall 管理のために各 IQ の負荷情報として IQ 内の命令数が通知される．

3.2 評価環境

以下で評価に用いるシミュレータの設定を**表 1** に示す．1 並列のクラスタ 8 つからなる構成とし，キャッシュやメモリ依存予測は理想化した．また，各クラスタの演算器は完全にパイプライン化されていると仮定し，それぞれ Alpha 21264 のすべての種類の演算を実行できると仮定した．分岐ミスペナルティとなる総パイプライン段数は，全部で 20 段とし，データ依存のある命令の発行間隔（Issue-to-Wakeup 遅延）は最短で 1 サイクルとした．

表 1 シミュレータの設定
Table 1 Simulator configuration.

Total Pipeline Depth	20 stages
Issue-to-Wakeup	1 cycle
D1 Access	2 cycles
クラスタ間通信遅延	2 cycles
整数演算遅延	1 cycle
整数乗算遅延	15 cycles
浮動小数点演算遅延	4 cycles
分岐予測	Gshare（64 k エントリ）
メモリ依存予測	理想化
キャッシュ	100% hit
クラスタ数	8
各 IQ の発行幅	1 issue/cluster
各 IQ entry	32
フェッチ幅	8
リタイア幅	8
演算器の機能	全種類の命令を処理可能
命令セット	Alpha 21264
測定命令数	最大 16 M 命令

表 2 クリティカルパス予測器の設定
Table 2 Configuration of Critical Path Predictor.

推定法	Qold
indexing	PC base direct map
エントリ数	32 k エントリ
各エントリの内容	6 bit 飽和カウンタ
Critical と判定された場合	カウンタを MAX にする
Non-Critical と判定された場合	カウンタを −1 する
Critical と予測する閾値	1 以上

同様にクリティカルパス予測器の設定を**表 2** に示す．各命令がクリティカルであるか推定する方法として，文献 16) で提案されている Qold を用いた．これは，IQ 内の最古の命令がオペランド待ち状態である場合に，クリティカルな命令であると推定するヒューリスティックである．想定アーキテクチャでは IQ は分散しているが，すべての IQ の中で最も古い命令の

表 3 使用ベンチマーク
Table 3 Benchmark set.

MediaBench（計 16 種）	adpcm（rawcadio, rawdaudio），epic（epic, unepic），g721（encode, decode），gsm（toast, untoastv），jpeg（cjpeg, djpeg），mesa（mipmap, osdemo, texgen），mpeg2（encode, decode），rasta
SPECint2000 test（計 11 種）	bzip2, crafty, gap, gcc, gzip, mcf, parser, twolf, vortex, vpr（place, route）
SPECint95 train（計 10 種）	compress, gcc, go, ijpeg, li, m88ksim, perl（jumble, primes, scrrable），vortex

みを Qold の対象とした．

使用するアプリケーション全 37 種を**表 3** に示す．これらのアプリケーションを最大 16 M 命令動作させ，平均 CPI を求めた．

4. 従来命令ステアリングの性能解析

本章では既存の命令ステアリング方式の性能を確認し，さらなる性能向上への余地を検討する．

想定アーキテクチャに対して，2 章で述べた各種ステアリングの性能を評価した．ただし Parcerisa 方式に関しては，NREADY の近似として DCOUNT ではなく，クラスタ内の未発行命令数（NINST）を用いることも有効だと考え，合わせて評価した．その場合負荷均衡の判断としては，クラスタの未発行命令数の最大値と最小値の差を閾値と比較した．本論文ではこの判断指標を Global Balance と呼び，Global Balance を用いた命令ステアリングを Global Balance Steering と呼ぶ．この方式のアルゴリズムは図 3 と同様であり，負荷集中の判断に Global Balance を，最低負荷クラスタの判断に NINST を用いる．

また，命令ステアリングによる CPI の下限を把握するために，通信遅延が 0 で発行幅が 8 である 1 クラスタ構成の理想プロセッサ（Ideal）の CPI も測定した．理想プロセッサに関しては，動作クロックの低下等は考慮せず，IQ エントリが 8 倍の 1 クラスタ構成としてシミュレーションした．

性能評価の結果を**図 5** に示す．棒グラフの下端は想定アーキテクチャの CPI の下限値となる，理想プロセッサの CPI を表している．中央の灰色部までの高さは各クラスタの発行幅を 1 に制限した場合の CPI であり，Select 遅延の影響を受ける．最上端はさらにクラスタ間通信遅延を 2 サイクルに設定した場合の

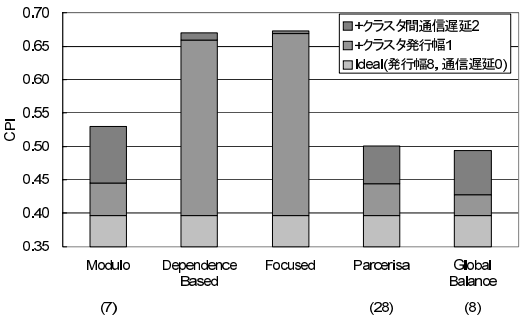


図 5 従来ステアリング方式の CPI
Fig. 5 CPI for each Prior Steering algorithm.

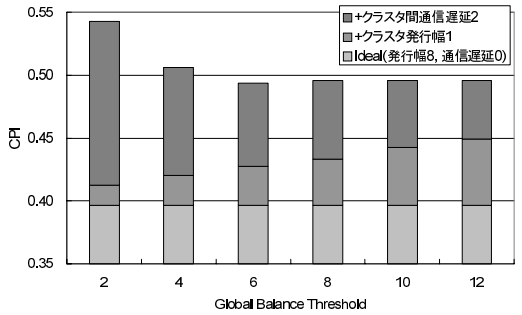


図 6 Global Balance Steering の CPI と閾値の関係
Fig. 6 CPI of Global Balance Steering for each threshold.

CPI であり，Wakeup 遅延の影響を受ける．最上端が各ステアリングを適用した場合の実際の性能を示している．なお，評価には括弧内に示した最適な閾値を用いた．

Focused は Dependence Based よりも通信遅延の影響を軽減できているが，すべてのデータ依存を重視するこれらの方式は Select 遅延の影響を強く受けており，他の方式よりも CPI 値が大きい．これはデータ依存関係にある命令が特定のクラスタに集中することが原因であり，未解決オペランドが存在する場合でも，必要に応じてデータ依存を無視し LW-戦略を採用することの必要性を示唆している．

また，命令分散機能に優れた残りの 3 方式に関しては，Modulo，Parcerisa 方式，Global Balance の順に CPI が低い．Modulo と Parcerisa 方式の比較からは，命令ステアリングの判断にデータ依存を用いることの有用性がうかがえる．Parcerisa 方式，Global Balance の比較からは，DCOUNT よりも Global Balance の方が命令ステアリングの判断指標としてより適していることが確認できた．

次に，最も CPI の低かった Global Balance Steering に対して，戦略判断に用いる閾値を変化させながら CPI の調査を行った．その結果を**図 6** に示す．

閾値を上げると LW-戦略の頻度が下がるために発行幅の影響が大きくなって Select 遅延が増加し，OP-戦略の頻度が上がるために通信遅延の影響が小さくなって Wakeup 遅延が減少する．閾値を下げると逆の変化が起こり，Wakeup 遅延と Select 遅延のバランスが最もよくとれている，閾値 6 の場合に最低の CPI である 0.494 を示している．閾値調整ですでに安定しているこの状態から，さらに CPI を削減するためには，以下の 2 つのいずれかを実現する必要がある．

(1) Select 遅延の増加を抑えつつ Wakeup 遅延を削減する．
(2) Wakeup 遅延の増加を抑えつつ Select 遅延を削減する．

5. さらに高性能な命令ステアリング方式の検討とその評価

5.1 より適切な負荷指標の検討

まず我々は，Select 遅延の増加を抑えつつ Wakeup 遅延を削減するアプローチとして，無意味な LW-戦略の適用に着目した．たとえば**図 7** は，横軸にクラスタを，縦軸にクラスタ内の命令数を示しており，この状況は特定のクラスタに命令が集中している状態を表している．このような状況下では，Global Balance が閾値を超えているため，次の命令には必ず LW-戦略が適用され，命令数が最少であるクラスタ c へ割り当てられる．しかし，未解決オペランドが比較的負荷の低いクラスタ b で生成される場合には，クラスタ c と b の Select 遅延の差は小さいと考えられるため，Wakeup 遅延を軽減する OP-戦略の方が適切と考えられる．このように適切な判断が行えない原因は，Global Balance が個々のクラスタの負荷を反映していないことにある．

そこで我々は，個々のクラスタの負荷状態を反映させた指標として，OP-クラスタの IQ 内命令数と，IQ 内命令数の最小値の差の命令数を検討した．この指標を Local Balance と呼び，この指標を用いる命令ステアリング方式を Local Balance Steering と呼ぶ．そのアルゴリズムを**図 8** に示す．図 3 との違いは，戦略判断にオペランドクラスタに関する負荷情報を使用する点にある．

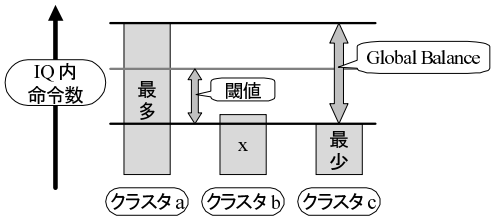


図 7 Global Balance によるステアリングが適切でない例
Fig. 7 An example of case where Global Balance is misleading.

Local Balance Steering を用いた場合の各閾値における CPI の内訳を，**図 9** に示す．判断に用いる指標は変わっているが，Global Balance Steering の同じ閾値と比べて CPI 中の発行幅に起因する部分に大きな変化は見られない．一方，通信遅延に起因する部分は，特に閾値が小さい場合に顕著に減少している．このことから，Local Balance を指標とすることで，Select 遅延を増やさずに Wakeup 遅延を削減できることが確認できた．また最適な閾値 4 の場合の CPI は 0.482 であった．

次に，閾値 CPI の関係に関して特徴的なアプリケーションのデータを**図 10** に掲載する．このグラフは最適な閾値 4 を，強調してある．右端の average37 が 37 アプリケーションの平均値であるが，多くのアプリケーションでは平均と同様に，閾値に対する CPI の変動は小さかった．しかしながら，adpcm では顕著な右下がり傾向が見られ，gap と mcf では顕著な右上がりの傾向が見られた．これは各アプリケーションの並列度に依存すると考えられる．CPI の大きい，並列度の低いアプリケーションでは命令を分散して発行幅を

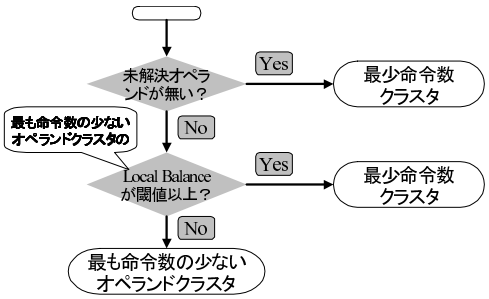


図 8 Local Balance Steering のアルゴリズム
Fig. 8 Algorithm of Local Balance Steering.

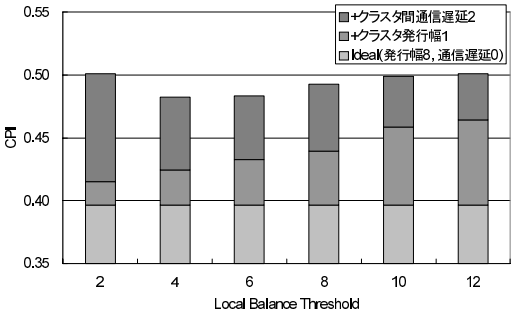


図 9 Local Balance Steering の CPI と閾値の関係
Fig. 9 CPI of Local Balance Steering for each threshold.

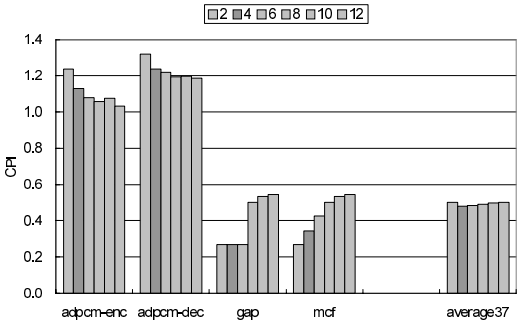


図 10 特徴的なアプリケーションに対する Local Balance Steering の CPI
Fig. 10 CPI of Local Balance Steering for some remarkable applications.

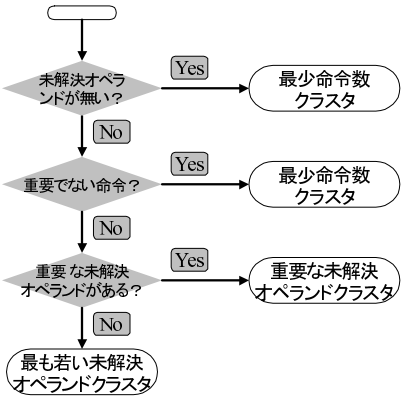


図 11 Criticality Steering のアルゴリズム
Fig. 11 Algorithm of Criticality Steering.

活用する意味が薄いため，閾値を大きくしてデータ依存を重視すべきである．それに対して CPI の小さい，並列度の高いアプリケーションでは，閾値を小さくして命令を分散させなければ，並列性を利用できない．

以上より，Local Balance Steering の性能は閾値設定に大きく依存することはないが，アプリケーションの並列性が極端な場合には閾値依存性が観測されることが分かった．

5.2 クリティカルパス情報を用いた戦略判断の提案

次に我々は，Wakeup 遅延の増加を抑えつつ Select 遅延を削減するアプローチとしてクリティカルパスに着目した．

4 章でも述べたように，データ依存に従ってステアリングを続けると，特定のクラスタに命令が集中して Select 遅延が発生する．そして，この遅延を避けるためには，データ依存を無視して LW-戦略を適用することが有効であった．これまでの方式では，負荷集中が発生してから LW-戦略を適用していたが，予防的に LW-戦略を用いておけば，すべてのクラスタを Select 遅延の少ない状態に保つことが可能であると考えられる．そこで我々は，重要でない命令に対して，予防的に LW-戦略を適用することで，重要な命令を速やかに発行できる可能性に着目した．

Focused Steering の項で述べたように，プログラムの実行時間を決めるのはクリティカルパスであり，その中に含まれる重要な命令に対する Wakeup 遅延や Select 遅延は，実行時間に大きく影響する．これとは対照的に，重要でない命令は多少処理が遅れても全体への影響は少ない．したがって，重要でない命令に対して予防的に LW-戦略を適用してもこれらの命令に対する Wakeup 遅延の影響は少ないと考えられる．一方で，重要な命令に対しては，どのクラスタも Select 遅延の小さい，良好な状態になっていると考えられる

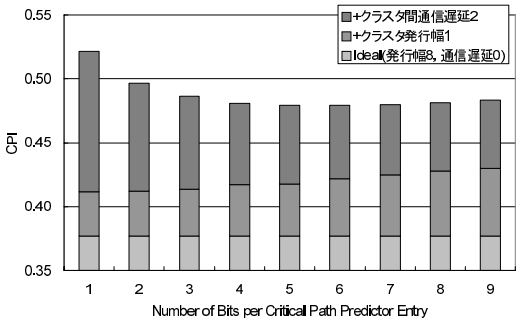


図 12 クリティカルパス予測テーブルエントリのビット数と Criticality Steering の CPI の関係
Fig. 12 CPI for Criticality Steering with different number of bits per Critical Path prediction table entry.

ため，OP-戦略を適用して Wakeup 遅延を軽減するべきだと考えられる．

そこで我々は，クリティカルパス情報を用いて OP-戦略/LW-戦略を判断する，Criticality Steering を提案する．負荷情報に基づいて戦略を判断するステアリングでは，複数オペランド命令に対するオペランド選択にも負荷情報を用いていたが，Criticality Steering ではオペランド選択にもクリティカルパス情報を用いる．そのアルゴリズムは**図 11** に示すように，分岐条件は Focused Steering と同じである．しかし，2 番目の分岐によって OP-戦略と LW-戦略が分かれる点に，発想の違いがある．

Criticality Steering を用いた場合の CPI を**図 12** に示す．Criticality Steering では，クリティカルパス予測の結果が直接ステアリング戦略になるため，予測された重要な命令の割合が性能に影響すると考えられる．そこでクリティカルパス予測器は表 2 で述べた構成のまま，予測テーブルエントリのビット数を 4〜9 に変化させることで，重要な命令の割合を変化させた．

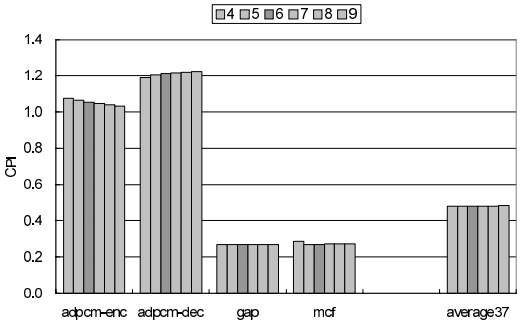


図 13 特徴的なアプリケーションごとの Criticality Steering の CPI
Fig. 13 CPI of Criticality Steering for some remarkable applications.

想定アーキテクチャに対しては，測定領域内で安定して低い CPI が得られている．また，6 ビットの場合が最も CPI が低く，0.479 であった．この数値は Global Balance Steering や Local Balance Steering よりも小さいことから，Criticality Steering の有効性を確認した．

次に図 10 で見た特徴的なアプリケーションの CPI をクリティカルパス予測器のビット数ごとにまとめたグラフを**図 13** に示す．このグラフも，最適なビット数である 6 を強調した．Criticality Steering では，グラフに示した 4 アプリケーションを含むすべてのアプリケーションで，予測器エントリのビット数と CPI の関係がおおむね平坦になった．このことから，Criticality Steering の CPI は予測器エントリのビット数に影響されにくいことが確認された．

5.3 クリティカルパス情報と負荷情報の併用

Criticality Steering では，重要な命令をステアリングする際にはつねに負荷バランスがとれていることを仮定して，つねに OP-戦略を適用していた．また，重要でない命令をステアリングする際にはつねに予防的負荷分散が有効であると仮定して LW-戦略を適用していた．しかし，クリティカルパス情報とクラスタの負荷情報は独立した情報であるため，必ずしもこの仮定は成り立たない．そこで本研究では，Criticality Steering で決定したステアリング先を，負荷情報を用いて修正することで，さらなる性能向上が得られる余地を検討した．その具体的な手順を**図 14**，**図 15** に示す．図 14 が処理の前半部であり，Criticality Steering のアルゴリズムを用いて，オペランドクラスタと閾値を選択する．続く後半部（図 15）では，決定した閾値を用いて Local Balance Steering を行い，最終的なステアリング先を決定する．この方式を Criticality + Local Balance Steering と呼ぶ．また，前半部で決定

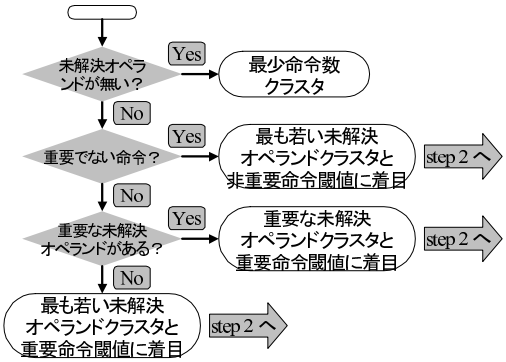


図 14 Criticality + Local Balance Steering のアルゴリズム（step1）
Fig. 14 Algorithm of Criticality + Local Balance Steering (step1).

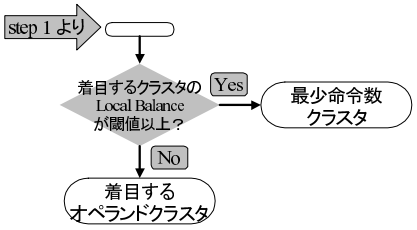


図 15 Criticality + Local Balance Steering のアルゴリズム（step2）
Fig. 15 Algorithm of Criticality + Local Balance Steering (step2).

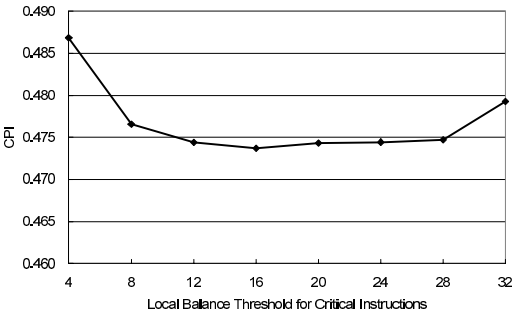


図 16 重要命令用閾値の変化と Criticality + Local Balance Steering の CPI（非重要命令用の閾値は 0）
Fig. 16 CPI of Criticality + Local Balance Steering for each critical threshold (NonCriticalThreshold = 0).

する 2 つの閾値をそれぞれ，重要命令用の閾値，非重要命令用の閾値と呼ぶ．

なお，Criticality Steering は重要命令用の閾値を IQ のエントリ数と同じ 32（すべて OP-戦略）に，非重要命令用の閾値を 0（すべて LW-戦略）に設定した，Criticality + Local Balance Steering と等価である．

非重要命令用の閾値を 0 に固定して，重要命令用の閾値を 32 から減少させた場合の CPI を**図 16** に示す．

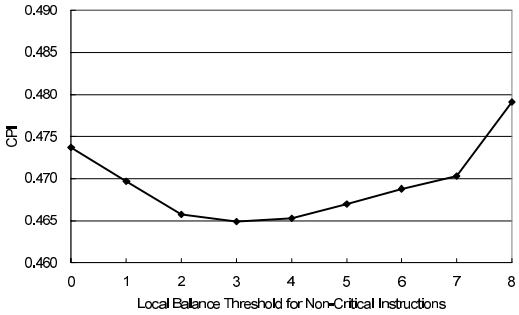


図 17 非重要命令用閾値の変化と Criticality + Local Balance Steering の CPI（重要命令用の閾値は 16）
Fig. 17 CPI of Criticality + Local Balance Steering for each non-critical threshold (CriticalThreshold = 16).

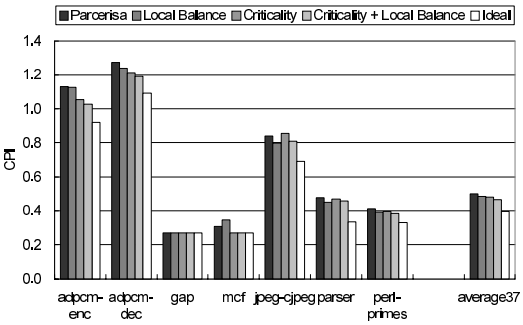


図 18 特徴的なアプリケーションに対する各方式の CPI
Fig. 18 CPI for each steering algorithm for some remarkable applications.

重要命令に対する最適な閾値 32 ではなく 16 であった．このことから，重要な命令であっても負荷集中時には LW-戦略を適用して Select 遅延を回避すべきであることが確認できた．

次に，重要命令用の閾値を 16 に固定し，非重要命令用の閾値を 0 から増加させた場合の CPI を**図 17** に示す．非重要命令に対する最適な閾値は 0 ではなく 3 であった．このことから，Criticality Steering では重要でない命令を負荷分散しすぎており，重要でない命令であっても負荷が十分に均衡している場合には OP-戦略を適用して Wakeup 遅延を回避すべきであることが確認できた．

以上から，Criticality Steering にクラスタの負荷情報を取り入れることで，さらに CPI を削減できることを確認した．また，重要命令用の閾値を 16，非重要命令用の閾値を 3 に設定するのが最適であり，その場合の CPI は Criticality Steering よりも 3%低い 0.465 であった．

5.4 各手法の性能比較

これまでに検討した方式と，提案方式の平均 CPI といくつかのアプリケーションにおける CPI を**図 18** にまとめる．adpcm 等，多くのアプリケーションでは平均と同様の傾向が見られたが，jpeg-cjpeg，parser，perl-primes 等のように Local Balance Steering と Criticality Steering の CPI が逆転するアプリケーションも少なからず存在した．これは，クリティカルパス情報がクラスタの負荷情報とは独立した指標であるため，それらを用いる命令ステアリングの優劣がアプリケーションに依存することを示唆している．しかしながら，両情報を組み合わせた Criticality + Local Balance 方式では，どのアプリケーションでも安定して低い CPI が得られている．

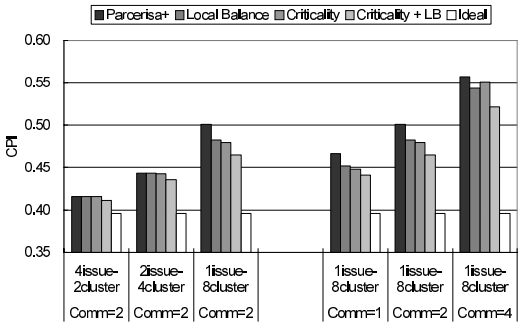


図 19 クラスタ数，通信遅延を変化させた場合の各方式の CPI
Fig. 19 CPI for each steering algorithm with different communication latency and number of clusters.

また，特殊なアプリケーションとしては，gap ですべての方式で同等の CPI が得られており，mcf で Local Balance 方式の CPI が他を上回っている．これらに関しては図 10 で述べたように，gap と mcf は並列度の高いアプリケーションであり，負荷分散がなされていれば理想に近い CPI が得られること，および Local Balance Steering の閾値依存性が高いことが原因である．

また，クラスタ数や通信遅延を変化させた場合の CPI を測定した結果を**図 19** に示す．全般的に，通信遅延が大きい場合やクラスタの発行幅が狭い場合といった，CPI 的に厳しい条件下では命令ステアリングの性能差が大きい．グラフは基本的に右下がりであるが，CPI 的な制限が緩い発行幅が 2 や 4 の場合には，各方式の性能差が小さく，場合によってはわずかに CPI が逆転している．また，最も厳しい条件である発行幅 1，通信遅延 4 という構成では，クリティカルパス情報のみを用いる Criticality Steering は，他の方式に比べて CPI の増加が著しい．これに関しては，Criticality Steering には重要でない命令を負荷分

散しすぎる性質があるために，通信遅延増加の影響を受けやすいことが原因であると考えている．しかしながら，Criticality Steering に負荷情報を取り入れることで，クラスタ構成によらず最も低い CPI が得られることを確認した．

6. まとめ

本論文では，次世代高クロック指向マイクロプロセッサアーキテクチャとして，分散 IQ 構成のクラスタ型アーキテクチャを想定し，クラスタの発行幅低下と通信遅延による CPI 増加を抑えるための，命令ステアリング方式に関して議論した．

関連研究で最も CPI の低かった Parcerisa 方式は，クラスタ負荷の近似指標として DCOUNT を用いていたが，未発行命令数を用いることでさらに性能が向上することを確認した．

次に，ステアリングの判断指標に，プロセッサ全体を代表する負荷情報ではなく，クラスタ個別の負荷均衡情報を用いる方式を検討した．この方式は Parcerisa らの方式と比べて 3.7% CPI 改善が得られた．

さらに我々は，プログラム中の重要でない命令に着目する，Criticality Steering を提案した．この方式では重要でない命令を用いて予防的に負荷を均衡させておくことで，重要な命令の Wakeup，Select 遅延を改善できる．この方式は Parcerisa らの方式と比べて 4.3% CPI 改善が得られた．

しかしながら Criticality Steering はクラスタの負荷情報に関して，重要な命令のステアリング時にはつねに負荷が均衡している，重要でない命令のステアリング時にはつねに予防負荷分散が必要である，という極端な仮定を前提としていた．そこで，Criticality Steering に負荷情報を取り入れる方式を検討したところ，CPI がさらに低下し，Parcerisa らの方式と比べて 7.1%の改善が得られた．

また，命令の重要性情報と負荷情報を単体で用いるステアリングの優劣は，アプリケーションやクラスタ数，通信遅延の大きさに依存していたが，両者を合わせた方式は環境によらず低い CPI が得られることを確認した．

今後の課題としては，提案方式のハードウェア的な複雑性の検討を考えている．

謝辞 本論文の研究は，一部 21 世紀 COE「情報技術戦略コア」による．

参考文献

1) Agarwal, V., Hrishikesh, M.S., Keckler, S.W. and Burger, D.: Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures, *ISCA 2000*, pp.248–259 (2000).
2) Baniasadi, A. and Moshovos, A.: Instruction Distribution Heuristics for Quad-Cluster Dynamically-Scheduled, Superscalar Processors, *MICRO 2000*, pp.337–347 (2000).
3) Canal, R., Parcerisa, J.-M. and González, A.: A Cost-Effective Clustered Architecture, *PACT 1999*, pp.160–168 (1999).
4) Fields, B., Rubin, S. and Bodik, R.: Focusing Processor Policies via Critical-Path Prediction, *ISCA 2001*, pp.74–85 (2001).
5) Hrishikesh, M.S., Jouppi, N.P., Farkas, K.I., Burger, D., Keckler, S.W. and Shivakumar, P.: The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays, *ISCA 2002*, pp.14–24 (2002).
6) Farkas, K.I., Chow, P., Jouppi, N.P. and Vranesic, Z.: The Multicluster Architecture: Reducing Cycle Time Through Partitioning, *MICRO 1997*, pp.149–159 (1997).
7) Kessler, R.E.: The Alpha 21264 Microprocessor, *IEEE Micro*, pp.25–36 (1999).
8) Kim, H.-S. and Smith, J.E.: An Instruciton Set and Microarchitecture for Instruction Level Distributed Processing, *ISCA 2002*, pp.71–86 (2002).
9) Kim, I. and Lipasti, M.H.: Half-Price Architecture, *ISCA 2003*, pp.28–38 (2003).
10) Palacharla, S., Jouppi, N.P. and Smith, J.E.: Complexity-Effective Superscalar Processors, *ISCA 1997*, pp.206–218 (1997).
11) Parcerisa, J.-M. and González, A.: Reducing Wire Delay Penalty through Value Prediction, *MICRO 2000*, pp.317–326 (2000).
12) Parcerisa, J.-M., Sahuquillo, J., González, A. and Duato, J.: Efficient Interconnects for Clustered Microarchitectures, *PACT 2002*, pp.291–300 (2002).
13) Raasch, S.E., Binkert, N.L. and Reinhardt, S.K.: A Scalable Instruction Queue Design Using Dependence Chains, *ISCA 2002*, pp.318–329 (2002).
14) Sprangle, E. and Carmean, D.: Increasing Processor Performance by Implementing Deeper Pipelines, *ISCA 2002*, pp.25–34 (2002).
15) Stark, J., Brown, M.D. and Patt, Y.N.: On Pipelining Dynamic Instruction Scheduling Logic, *MICRO 2000*, pp.57–66 (2000).
16) Tune, E., Liang, D., Tullsen, D.M. and Calder,

B.: Dynamic Prediction of Critical Path Instructions, *HPCA 2001*, pp.185–196 (2001).

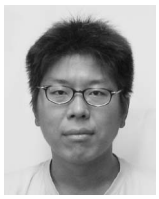
(平成 15 年 10 月 10 日受付)
(平成 16 年 2 月 28 日採録)

服部　直也

1976 年生．1999 年東京大学工学部電子情報工学科卒業．2004 年同大学院情報理工学系研究科電子情報学専攻博士課程修了．情報理工学博士．プロセッサアーキテクチャ等の研究に従事．

高田　正法（学生会員）

1979 年生．2003 年東京大学工学部電子情報工学科卒業．現在，同大学院情報理工学系研究科電子情報学専攻修士課程在学中．プロセッサアーキテクチャ等の研究に従事．



岡部　淳

1976 年生．1999 年早稲田大学理工学部電子通信学科卒業．2001 年東京大学大学院工学系研究科情報工学専攻修了．現在，同大学院情報理工学系研究科電子情報学専攻博士課程在学中．プロセッサアーキテクチャ等の研究に従事．

入江　英嗣（学生会員）

1975 年生．1999 年東京大学工学部電子情報工学科卒業．2004 年同大学院情報理工学系研究科電子情報学専攻博士課程修了．情報理工学博士．プロセッサアーキテクチャ等の研究に従事．

坂井　修一（正会員）

1981 年東京大学理学部情報科学科卒業．1986 年同大学院工学系研究科情報工学専門課程修了．工学博士．同年工業技術院電子技術総合研究所入所．1991 年〜1992 年，米国マサチューセッツ工科大学招聘研究員，1993 年〜1996 年 RWC 超並列アーキテクチャ研究室室長．1996 年筑波大学電子・情報工学系助教授．1998 年東京大学大学院工学系研究科助教授，2001 年同大学院情報理工学系研究科教授．計算機システム一般，特にアーキテクチャ，並列処理，スケジューリング問題，マルチメディア応用等の研究に従事．著書『論理回路入門』，『図説コンピュータアーキテクチャ』．電子情報通信学会，人工知能学会，IEEE，ACM 各会員．

田中　英彦（フェロー）

1965 年東京大学工学部電子工学科卒業．1970 年同大学院工学系研究科博士課程修了．工学博士．同年同大学工学部講師．1971 年同助教授．1987 年同教授．2001 年より同大学院情報理工学系研究科教授・研究科長．この間 1978 年〜1979 年米国ニューヨーク市立大学客員教授．計算機アーキテクチャ，並列処理，自然言語処理，メディア処理，分散処理，CAD 等の研究に興味を持っている．著書『非ノイマンコンピュータ』，『情報通信システム』，『Parallel Inference Engine—PIE』，共著書『計算機アーキテクチャ』，『VLSI コンピュータ I, II』，『ソフトウェア指向アーキテクチャ』．電子情報通信学会，人工知能学会，日本ソフトウェア科学会，IEEE，ACM 各会員．