システムＬＳＩ設計技術　107－10
(2002. 11. 28)

並列キュープロセッサの基本設計

Ben A. ABDERAZEK† 繁田 聡一† Kirilka NIKOLOVA† 吉永 努† 曽和 将容†

† 電気通信大学 大学院情報システム学研究科 〒182-8585 東京都 調布市 調布ヶ丘 1-5-1
E-mail: †{ben,shigeta,nikol,yosinaga,sowa}@is.uec.ac.jp

あらまし 我々の提案する並列キュープロセッサは，命令レベルの並列実行による性能向上と小さなプログラムコードサイズという利点を備えている．これは，キュー計算モデルの性質により，前後の命令間にデータの依存関係（レジスタ競合）が発生する可能性が低いことと，命令にレジスタ指定部が必要ないことに基づいている．本稿では，並列キュープロセッサの基本設計と基礎的な性能特性について報告する．

キーワード キュー計算モデル，プロセッサアーキテクチャ，命令レベル並列性

Fundamental Design of a Parallel Queue Processor

Ben A. ABDERAZEK†, Soichi SHIGETA†, Kirilka NIKOLOVA†, Tsutomu YOSHINAGA†, and Masahiro SOWA†

† Graduate School of Information Systems University of Electro-Communications.
1-5-1. Chofugaoka. Chofu-shi. Tokyo 182-8585. Japan
E-mail: †{ben,shigeta,nikol,yosinaga,sowa}@is.uec.ac.jp

Abstract We propose a Parallel Queue Processor architecture based on the queue computation model. The Parallel Queue Processor has advantages in instruction level parallelism and samll size of executable code. These characteristics come from the queue computation model's features: probability of occurence of data dependency (register conflict) between contiguous instructions are low and explicit specifications of registers are not required in executable code. A fundamental design of a Parallel Queue Processor and simulation results of preliminary evaluation are mentioned in this paper.

Key words Queue Computation Model, Processor Architecture, Instruction Level Parallelism

1. はじめに

スタックを使った計算方法はJavaなどでよく知られているが，スタックの代わりにキューを用いたキュー計算方法は，数々の良い特性を持っているにもかかわらず，シリアルプロセッサに関して少し研究されただけで[2]. [8]，まだ，ほとんど研究されていない．

キュー計算方法は中間結果格納用にFIFOの記憶装置を用いた計算方法で，この方法を用いたプロセッサでは，

（1） 問題が持っている並列性がすべて出せる
（2） プログラム長が短い
（3） 命令間の従属関係距離が長いのでストールしにくい
（4） 並列発見ハードウエアがスーパスカラコンピュータのように複雑でないので，ハードウエアが簡単
（5） レジスタ指定が暗黙的でキューなので，原則としてレジスタを介した従属性がなくレジスタリネーミングの必要がない．また，ハードウエアが簡単
（6） リアルなマルチスレッド処理に適している

など，数々の素晴らしい特性をそなえている．

我々は数年前から，並列処理を中心としたキュープロセッサの研究を行い設計を試みた[4]. [15]．ところが，このキュー計算方法はなかなか奥が深く，これらの経験を元に基礎原理から考え直し，その考察にそって研究を行っており[1]. [5]. [16]，これまでに多くの種類のキュープロセッサを提案している[13]. [14]．本稿では，そのうちの一番基本とされるキュープロセッサの基本原理とシミュレーションによる基本結果について述べる．

2. キュー計算モデル

キュー計算モデルとは，キューと呼ばれるFIFO（first in first out）のデータ構造を計算結果の格納場所として用いて計算を行う計算モデルである．Javaで採用されている，スタックと呼ばれるLIFO（last in first out）のデータ構造を計算結果の格

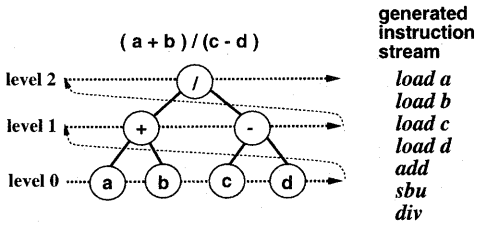

図1　構文木の幅優先探索による命令列生成

Fig. 1　Instruction stream generated by depth first traverse of a syntax tree.

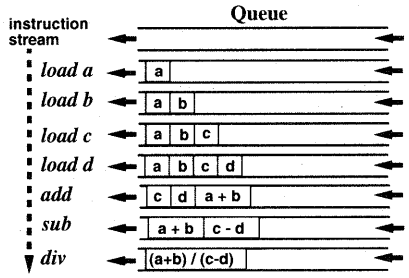

図2　キュー計算モデルによる命令実行の様子

Fig. 2　Instruction execution with queue computation moel.

納場所として用いる，スタック計算モデル [7], [9] と対称的な計算モデルである．

キュー計算モデルでは，キューの先頭からデータを取り出して演算を施し，その結果をキューの末尾に挿入する．データの取り出し位置と挿入位置は，キューの先頭と末尾に限定されている．そのため，ランダムアクセスのレジスタ計算モデルのように，どのレジスタからデータを取り出し，どのレジスタに結果を格納するのかという指定を各命令に対して明示的に行う必要がない．したがって，レジスタ指定部が不要な分だけ，プログラムコードのサイズが小さくなる．

図1は，$(a+b)/(c-d)$ という計算式の構文木を示している．各ノードが命令に対応し，ノード間のエッジは，データの依存関係を示している．(演算子が書かれたノードは演算命令に対応し，a や b という変数が書かれた命令はデータを準備する命令に対応する．) キュー計算モデルでは，図1の破線で示されているように，構文木を幅優先に探索して命列を生成する．

図2は，図1で生成された命令列を，キューを用いて計算する様子を示したものである．一番上のキューは，初期状態の空のキューである．最初の `load a` 命令で，キューの末尾から a というデータが挿入される．ここでは，キューの初期状態が空なので，このデータがキューの先頭に位置することになる．2番目の `load b` 命令では，同様にキューの末尾から b というデータがキューに挿入される．以下，4番目の *load d* 命令まで同様である．続いて，5番目の `add` 命令では，(2項演算子であるので) キューの先頭から2つのデータ a と b が取り出されて，加算が行われ，結果の $a+b$ がキューの末尾から挿入される．以下同様に `sub` 命令と `div` 命令が実行されると，$(a+b)/(c-d)$ がキューに残り，所望の計算式の結果が得られる．

3. 並列キュー計算モデルへの拡張

第2章で述べたように，キュー計算モデルでは，構文木を幅優先探索することで命令列を生成する．ここで，構文木に現れるノードの高さを「レベル」と呼ぶことにし，下から level 0, level 1, ... とレベル付けする (図1参照)．この時，同じレベルに属するノード間にはエッジが存在しない．これは，同じレベルに属する命令間にデータの依存関係が無いことを表している．したがって，キュー内に挿入されている先頭以外のデータに対しても先頭のデータと同じタイミングでアクセスすることを許せば，これらの命令は同時に実行することができる．このように，キュー内の複数のデータに対して，同時にアクセスできるように拡張したキューを「並列キュー」と呼ぶことにする．また，この並列キューを用いて計算を行う計算モデルを「並列キュー計算モデル」と呼ぶ．

図3は，並列キューを用いて，図1で生成された命令列を実行する様子を示している．同じレベルに属する命令を同時に実行している．この時，キューから取り出すデータの順序や計算結果をキューに挿入する順序に矛盾が生じないように制御する必要がある．そこで，命令が同時実行される前に，キュー内の何番目の位置のデータを取り出すのか，何番目の位置に結果を挿入するのかを各命令に対して計算している．例えば，`add` と `sub` を並列に実行するに際しては，`add` 命令が1番目と2番目の位置の a, b というデータを取り出して結果の $(a+b)$ を1番目の位置に挿入するように，`sub` 命令が3番目と4番目の位置の c, d というデータを取り出して結果の $(c-d)$ を2番目の位置に挿入するように，取出し位置と挿入位置の割当てを行う．並列に実行される n 個の命令のうちの $i+1$ 番目の命令に対する取出し位置 QHP (queue head point) と挿入位置 QTP (queue tail point) は，それぞれ以下の式により計算できる．ここで，$Consume(i)$ は i 番目の命令がキューから取出すデータの数を，$Produce(i)$ は i 番目の命令がキューに挿入するデータの数を表す．

$$QHP(i+1) = QHP(i) + Consume(i)$$

$$QTP(i+1) = QTP(i) + Produce(i)$$

この計算は，実行時に動的に行う．それゆえ，データの取出し／挿入位置の情報を各命令に対して静的に付けておく必要はなく，プログラムのコードサイズは従来のキュー計算モデルを用いた場合と全く変わらない．

4. 並列キュープロセッサの基本設計

4.1 命令フォーマット

図4に示されているように，並列キュープロセッサの命令フォーマットは1バイトと3バイトの可変長のフォーマットになっている．算術論理演算命令は全て1バイト長である．一方，メモリアドレスを伴う命令 (ロード／ストア命令，分岐命令)

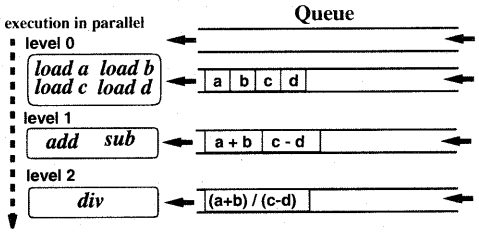

図3　並列キューを用いた命令の並列実行の様子
Fig. 3　Parallel execution by using a parallel queue.

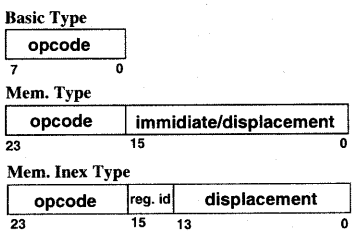

図4　並列キュープロセッサの命令フォーマット
Fig. 4　Instruction formats for a Parallel Queue Processor.

は全て3バイト長であるが，アドレッシングモードによって2種類のフォーマットがある．今回の基本設計では，インデックスレジスタとアドレスレジスタを4本ずつ搭載することを想定しているため，レジスタの識別に2ビットを割当てている．

4バイトの固定命令長のRISC（reduced instruction set architecture）命令と較べると，演算命令の長さが1/4，メモリアクセス命令の長さが3/4となっている．プログラムに含まれる演算命令とメモリアクセス命令の割合を考慮すると，並列キュープロセッサのプログラムコードサイズは，RISCプロセッサのプログラムコードサイズの1/2〜1/3の小ささになると考えられる．

4.2　基本ブロック構成

図5は，並列キュープロセッサの基本ブロック構成を示している．並列キュープロセッサは，次の5ステージで構成されるパイプライン処理を行う．

（1）　フェッチ: 毎サイクルnバイトを命令メモリからフェッチバッファにフェッチする．4.1節で述べたように，可変命令長の命令フォーマットを採用したため，3バイト長命令は最初の1バイト目もしくは2バイト目の部分で分断されてフェッチされる可能性がある．分断された3バイト長命令の検出と結合は，デコードステージで行う．

（2）　デコード: 命令のタイプと使用する機能ユニットの種類の判別を行う．分断されてフェッチされた3バイト長命令を検出した場合は，残りの部分との結合を行ってからデコードする．また，各命令に対して，データの取出し位置QHPと挿入位置QTPの計算を行う．

（3）　発行: 発行可能な命令が発行する．特に，OoO発行の場合は，命令ディスパッチバッファの中から発行可能な命令を探して発行する．

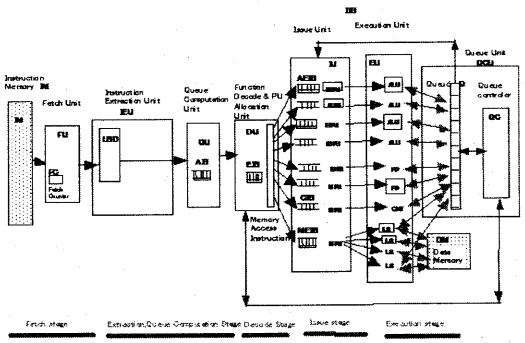

図5　並列キュープロセッサの基本ブロック構成
Fig. 5　Basic block diagram of a Parallel Queue Processor.

表1　測定に用いた4つの並列キュープログラムの性質
Table 1　The features of 4 parallel queue benchmark programs.

	命令レベルの最大並列度	データ依存関係がある命令間の距離
プログラムA	大	長
プログラムB	大	短
プログラムC	小	長
プログラムD	小	短

（4）　実行: 各種の機能ユニットで命令が実行する．

（5）　完了: 実行結果が，キューの挿入位置に書き込まれ，メインメモリの内容が更新される．

5.　シミュレーション

5.1　ベンチマークプログラム

プログラム実行時の命令レベルの並列度を測定するために，並列キュープロセッサ用の4つのベンチマークプログラムA, B, C, Dを用意した．Aは32点の高速フーリエ変換，Bは20×20の行列のLU分解，Cは次数8のプリフィックス計算，Dは次数4のニュートン多項式補間を行う計算にそれぞれ基づいて作成した．表1は，各々のベンチマークプログラムの性質を，静的な解析により求めた命令レベルの最大並列度とデータ依存関係がある命令間の平均距離の観点で分類したものである．

十分な機能ユニット数と命令フェッチ幅が提供されれば，最大の命令レベル並列度でプログラムが実行される．しかし，例えばプログラムAでは，それを達成するために64個のロードユニットと32個の算術演算ユニット，192バイトの命令フェッチ幅が必要となり，現実的ではない．そこで，機能ユニット数と命令フェッチ幅を変化させて，並列キュープロセッサでプログラムを実行した時の命令レベルの並列度に対する影響について調べた．

5.2　シミュレーション条件

並列キュープロセッサの基礎的な特性を調べるために，シミュレータを開発し，シミュレーションを行った．機能ユニット数と命令のフェッチ幅をパラメータとし，プログラム実行時の命令レベルの平均並列度（サイクル毎の命令レベルの並列度の平均値）を測定した．また，命令発行ポリシーの違いによる影響

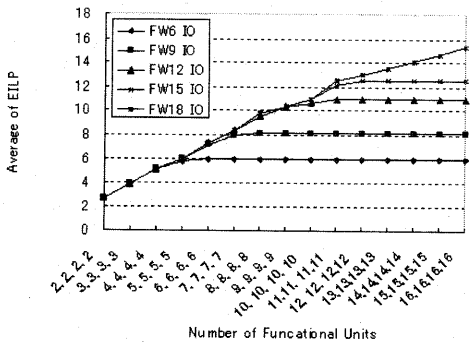

図6 A を IO 発行で実行した時の命令レベルの平均並列度
Fig. 6 Avarege of executed ILP for A with IO issue policy.

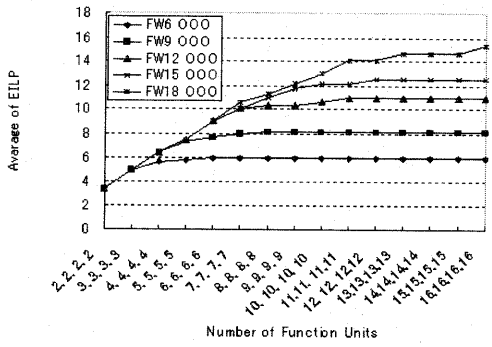

図7 A を OoO 発行で実行した時の命令レベルの平均並列度
Fig. 7 Average of executed ILP for A with OoO issue policy.

を調べるために，in order（IO）発行と out of order（OoO）発行の 2 つの命令発行ポリシーについてシミュレーションを行った．

今回のシミュレーションでは，以下の条件を仮定している．

- キューの長さは十分に長い
- 4 種類の機能ユニットを備えている
  - ロードユニット
  - ストアユニット
  - 整数算術論理演算ユニット（加減算）
  - 整数算術論理演算ユニット（乗除算）
- 5 段のパイプライン処理を行う
- 全ての命令の実行時間は等しい
- OoO 発行用の命令バッファのサイズは十分に大きい

5.3 シミュレーション結果

図 6 は，プログラム A を IO 発行で実行した時の，命令レベルの平均並列度の変化の様子を示している．横軸は 4 種類の機能ユニットの各々の個数を，縦軸は実行時の命令レベルの平均並列度である．例えば，ユニット数 (8.8.8.8) で構成される並列キュープロセッサでは，フェッチ幅をい 12 バイト以上にしても改善が見られないことが分かる．

図 7 は，プログラム A を OoO 発行で実行した時の，命令レベルの平均並列度の変化の様子を示している．例えば，ユニット数 (8.8.8.8) の並列キュープロセッサに対しては，15 バイトのフェッチ幅が最適であることが分かる．ここで，前述の IO 発行の場合と比較してみると，IO で約 10 命令，OoO で約 11 命令と，OoO 発行を適用しても命令レベルの平均並列度には大きな改善が見られないことが分かる．これは，静的な命令レベルの並列度が大きいという，プログラム A の性質のためであると考えられる．すなわち，構文木の同一レベルに属する命令数が多いので，フェッチした命令のほとんどが in order の順序で並列実行可能なためであると考えられる．

図 8 は，プログラム B を IO 発行で実行した時の，命令レベルの平均並列度の変化の様子を表している．A の場合と異なり，フェッチ幅の増加による，並列度の改善が見られない．これは，依存関係がある命令間の平均距離が短いという，プログラム B の性質のためであると考えられる．すなわち，フェッチ幅を大きくしたとしても，データ依存関係のために前の命令の完了を待たなければならないため，並列度が改善されないと考えられる．

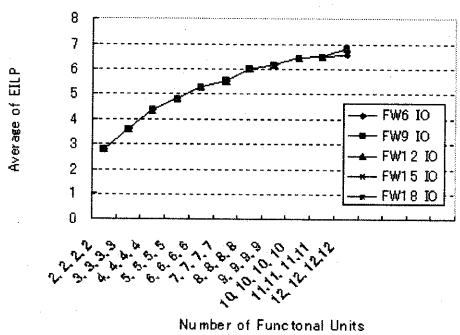

図8 B を IO 発行で実行した時の命令レベルの平均並列度
Fig. 8 Average of executed ILP for B with IO issue policy.

図 9 は，プログラム B を OoO 発行で実行した時の，命令レベルの平均並列度の変化の様子を表している．図 8 と比較すると，OoO 発行を適用することにより，実行時の命令レベルの平均並列度が効果的に改善されているのが分かる．例えば，フェッチ幅 12 バイト，機能ユニット数 (8.8.8.8) の場合で較べると，命令レベルの平均並列度が 6 命令/サイクルから 12 命令/サイクルと，100%の改善率になっている．さらに，フェッチ幅 18 バイト，機能ユニット数 (12.12.12.12) の場合では，7 命令/サイクルから 17 命令/サイクルと，143%も改善されている．このように，データ依存関係がある命令間の平均距離が短いプログラムでは，OoO 発行を適用することで，データ依存の解消を待っている命令よりも後ろの命令を先に実行することができるようになり，平均並列度が改善されることが分かる．

また，OoO 発行の適用によって，機能ユニットの不足によって引き起こされる平均並列度の低下を回避することができる．これは，IO 発行では，機能ユニットが使用可能になるのを待っている命令が後続命令の実行を妨げるのに対して，OoO 発行で

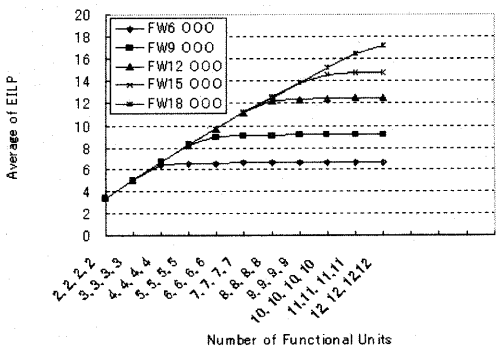

図9　B を OoO 発行で実行した時の命令レベルの平均並列度
Fig. 9　Average of executed ILP for B with OoO issue policy.

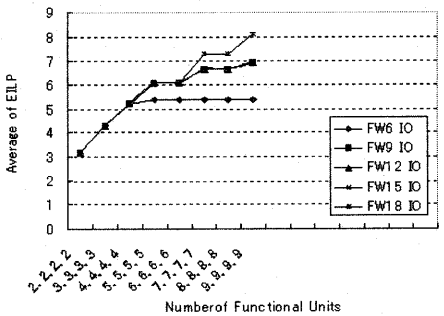

図10　C を IO 発行で実行した時の命令レベルの平均並列度
Fig. 10　Average of executed ILP for C with IO issue policy.

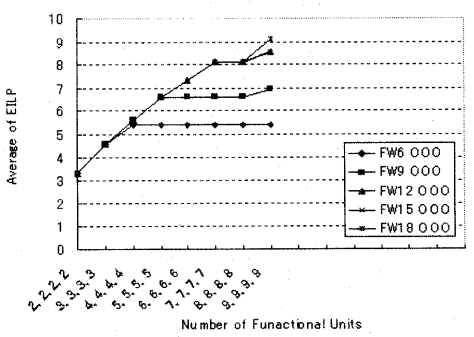

図11　C を OoO 発行で実行した時の命令レベルの平均並列度
Fig. 11　Average of executed ILP for C with OoO issue policy.

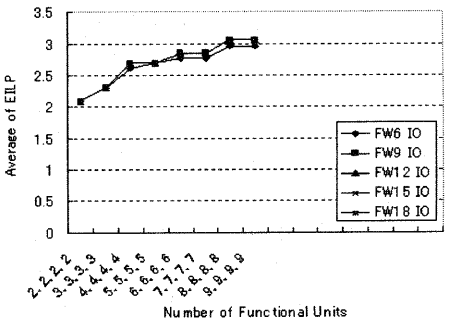

図12　D を IO 発行で実行した時の命令レベルの平均並列度
Fig. 12　Average of executed ILP for D with IO issue policy.

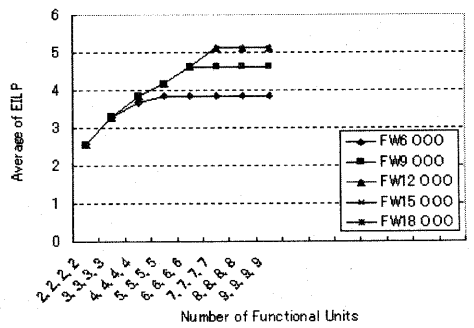

図13　D を OoO 発行で実行した時の命令レベルの平均並列度
Fig. 13　Average of executed ILP for D with OoO issue policy.

は後続命令を先に実行できるからである．例えば，図8と図9から，フェッチ幅6バイトで6命令/サイクルの平均並列度を達成するためには，IO 発行の場合は (8,8,8,8) のユニット数が必要なのに対して，OoO 発行の場合は半分の (4,4,4,4) のユニット数でよいことが分かる．

図10と図11はそれぞれ，プログラム C を IO 発行と OoO 発行で実行した時の，命令レベルの平均並列度の変化の様子を表している．これらは，プログラム A の場合と同様の傾向を示しているが，A の場合に較べて，速く飽和点に達している．これは，静的な命令レベルの並列性が小さいという，プログラム C の性質のためである．

図12と図13はそれぞれ，プログラム D を IO 発行と OoO 発行で実行した時の，命令レベルの平均並列度の変化の様子を表している．静的な命令レベルの並列度が小さいという D の性質のために平均並列度は大きくなりにくいが，プログラム B の場合と同様に，OoO 発行の適用によって平均並列度が効果的に改善されているのが分かる．

5.4 考　　察

並列キュープロセッサを実装する時に，どの程度の機能ユニットとフェッチ幅を実装すべきかを，シミュレーション結果から考察する．機能ユニット数は，並列キュープロセッサのハードウェア規模に大きな影響を与えるので，まず機能ユニット数について検討し，次にフェッチ幅について検討を行うことにする．

並列キュープロセッサのプログラムコードは，第1章で述べた構文木の幅優先探索によって，並列実行可能な命令が静的に並べられている．それゆえ，既存のスーパスカラプロセッサに較べて，並列実行可能な命令を発見するためのハードウェアが簡単化できると考えられる [6]. [10]〜[12]．また，実行時にレジスタリネーミングを行う必要がないので，その分だけハードウェア規模を抑えることができる．したがって，既存のスーパスカラプロセッサに実装されている機能ユニット数や今後の

表2　ユニット数(8,8,8,8)に対する最適フェッチ幅
Table 2　Optimum FW for (8,8,8,8) FUs.

	最適フェッチ幅（バイト）
プログラムA IO	12
OoO	15
プログラムB IO	6
OoO	12
プログラムC IO	15
OoO	12
プログラムD IO	9
OoO	12
平　均	11.6

表3　フェッチ幅12 byteに対するユニット数の飽和点
Table 3　Satulation point in number of FUs for 12Bytes FW.

	飽和点（ユニット数）
プログラムA IO	(10,10,10,10)
OoO	(7,7,7,7)
プログラムB IO	(8,8,8,8)
OoO	(8,8,8,8)
プログラムC IO	(5,5,5,5)
OoO	(7,7,7,7)
プログラムD IO	(4,4,4,4)
OoO	(7,7,7,7)
平　均	(7,7,7,7)

ハードウェア技術の進歩を考慮すると，(8,8,8,8)の機能ユニットを並列キュープロセッサに実装することは可能であると考えられる．

表2は，シミュレーションで用いた各ベンチマークプログラムを機能ユニット数(8,8,8,8)の並列キュープロセッサで実行する場合に最適なフェッチ幅をまとめたものである．この表から，機能ユニット数(8,8,8,8)の並列キュープロセッサでは，12バイトのフェッチ幅が適当であると判断できる．

そこで次に，フェッチ幅を12バイトとした場合に，どれだけの機能ユニット数が必要とされるのかを調べた．表3は，フェッチ幅を12バイトとした並列キュープロセッサでベンチマークプログラムを実行した場合の，機能ユニット数の飽和点をまとめたものである．この表から，機能ユニット数(8,8,8,8)の並列キュープロセッサを用いれば，プログラムAをIO発行した場合を除いて，フェッチ幅12バイトでの最大の性能を引き出せることが分かる．

6. ま と め

本稿では，キュー計算モデルに基づく並列キュープロセッサを提案し，その基本設計について述べた．また，シミュレーションにより，並列キュープロセッサの基礎的な特性を調べた．その結果，静的な命令ベルの並列度が大きく，依存関係がある命令間の平均距離も大きいプログラムでは，機能ユニット数の増加とフェッチ幅の増加にともなって，実行時の命令レベルの平均並列度が線形的に増していくことが確かめられた．これは，並列実行可能な命令が静的に並ぶように，構文木の幅優先探索によって並列キュープロセッサのプログラムコードが生成されるためである．また，依存関係がある命令間の平均距離が短いプログラムを実行する場合や，機能ユニット数が少ない並列キュープロセッサでは，命令発行ポリシーにout of order発行を適用することによって，実行時の平均並列度が効果的に改善されることが分かった．

現在，曽和研究室で提案された多くの種類のキュープロセッサのハードウェア設計が行われている．

文　献

[1] B. A. Abderazek, K. Nikolova, and M. Sowa, "FARM-Queue Mode: On a Practical Queue Execution model", *In Proceedings of the International Conference on Circuits and Systems, Computers and Communications*, pp.939-944 (2001).

[2] R. Bruno and V. Carla, "Data Flow on Queue Machines", *In Proceedings of 12th International IEEE Symposium on Computer Architecture*, pp.342-351 (1985).

[3] K. M. Michael and G. C. Harvey, "Processor Implementation Using Queue", *IEEE Micro*, pp.58-66 (1995).

[4] S. Okamoto, H. Suzuki, A. Maeda, and M. Sowa, "Design of a Superscalar Processor Based on Queue Machine Computation Model", *In Proceedings of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing* (PACRIM'99), pp.151-154 (1999).

[5] 奥村, 吉永, 曽和, "キューマシン用並列化Cコンパイラ", 情報処理学会研究報告 Vol.2002, No.81, pp.127-132 (2002).

[6] S. Palacharia, N. P Joupi, and J. E. Smith, "Complexity-Effective Superscalar Processor", *Ph.D. Dissertation*, Univ. of Wisconsin (1998).

[7] K. Philip, "Stack Computers", The New Wave, Mountain View Press (1989).

[8] B. R. Preiss, "Data Flow on a Queue Machine", *Doctoral Thesis*, Department of Electrical Engineering, University of Toronto (1987).

[9] R. Radhakrishnan, D. Talla, and L. K. John, "Allowing for ILP in an Embedded Java Processor", *In Proceedings of IEEE/ACM International Symposium on Computer Architecture*, pp.294-305 (2000).

[10] J. Silc, B. Robic, and T. Ungerer, "Processor Architecture: From Dataflow to Superscalar and Beyond", Springer-Verlag (1999).

[11] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors", *In Proceedings of the IEEE*, Vol. 83, No. 12, pp.609-1624 (1995).

[12] G. Sohi, "Instructions Issue Logic for High-performance, Interruptible, Multiple Functional Unit, Pipelined Computer", *IEEE Transactions on Computers*, Vol. 39, No. 2, pp.349-359 (1990).

[13] 曽和, "Parallel Queue Summary", 曽和研究室テクニカルレポート SLL00400 (2000).

[14] 曽和, "Serial Queue Summary", 曽和研究室テクニカルレポート SLL02300 (2002).

[15] H. Suzuki, S. Okamoto, A. Maeda, and M. Sowa, "Implementation and Evaluation of a Superscalar Processor Based on Queue Machine Computation Model", *In Proceedings of IPSJ SIG*, Vol. 99 No. 21, pp.91-96 (1999).

[16] L. Q. Wang, M. Sowa, and T. Yoshinaga, "High Parallelism Java Compiler with Queue Architecture", 情報科学技術フォーラム(FIT 2002), No.1, pp.135-136 (2002).