継続モデルにデータ要求概念を適用したスレッド駆動制御の提案

泉 雅昭† 雨宮 聡史† 松崎 隆哲‡ 雨宮 真人‡
†九州大学 大学院 システム情報科学府
‡九州大学 大学院 システム情報科学研究院
〒816-8580 福岡県 春日市 春日公園 6-1
{masaaki, roger, takanori, amamiya}@al.is.kyushu-u.ac.jp

継続モデルとは，他プログラムからの中断を受けずに排他的に走りきるプログラム片をスレッドと定義し，スレッド間の実行順序を継続概念によって制御するプログラミング・モデルである．継続モデルでは継続関係をデータ依存に沿って与えることにより，スレッド・レベルの並列実行が可能となる．Fuce プロセッサはこの継続モデルで実現する並列プログラムを効率的に実行する．本稿では新たにデータ要求概念を提案し，それを継続モデルに適用することで生産者・消費者問題に存在する排他制御のためのロック操作を排除できることを示す．さらに，ロック操作に比べデータ要求概念を用いることでスレッド・プログラム記述の観点から容易にプログラミングが可能であることを示す．

Thread Drive Control
Based on Continuation Model Applied Data Demand Concept to

Masaaki Izumi, Satoshi Amamiya, Takanori Matsuzaki, Makoto Amamiya
Graduate School of Information Science and Electrical Engineering, Kyushu University
6-1 Kasuga-koen, Kasuga, Fukuoka, Japan, 816-8580
{masaaki, roger, takanori, amamiya}@al.is.kyushu-u.ac.jp

Continuation model is a programming model in which programs are constructed with non-preemptive threads, and the execution order between threads is specified as the continuation of computation from a thread to one or more threads. By giving the continuation along data dependency, thread level parallellism will be exploited. The Fuce processor executes those continuation-based thread programs effciently. This paper proposes Data Demand Concept. The continuation model applied Data Demand Concept to excludes the lock operation in the exclusive control that exists in the producer and the consumer's issue. In addition, This paper shows easier programming from the viewpoint of the thread program description by the Data Demand Concept compared with the lock operation.

1 はじめに

命令レベルの並列性を利用するスーパスカラプロセッサは，ソフトウェアパイプライニングやループアンローリングなどのコンパイラ技術により性能改善を行ってきた．しかし，単一プロセスまたは単一スレッド実行における命令レベル並列性の利用には限界があり [3]，性能改善が困難になってきている．一方，スレッドレベル並列性の利用により性能改善する研究 [4] が行われている．そこではプログラムを複数スレッドへ分割することにより，I/O 処理とキャッシュミスによるメモリアクセスの遅延隠蔽や，本来プログラムが備えるデータ並列性の利用を図ることが主眼である．しかし，複数スレッドの並列実行はスレッド間の同期処理や複雑なスレッドスケジューリングのオーバヘッドが性能改善を妨げる問題がある．

そこで，筆者らは並列処理と親和性の高いデータフロー計算モデルを基盤とした継続モデルに基づく Fuce プロセッサ [2] を提案してきた．Fuce プロセッサは継続モデルをマルチスレッド実行モデルに採用することで，複数スレッドの並列実行性能を徹底的に追求する．継続モデルはスレッド間のデータ依存関係を継続概念 [1] として定義することで，スレッ

ドの駆動制御を行う．そのため，スレッドは起動時点でデータ依存関係を解消している．また，スレッドは中断なしに走行するので，スレッドの走行状態は割り込み等により休眠状態になることはない．以上の二点により従来のプロセッサで採用されているマルチスレッド実行モデルと比較して，継続モデルではスレッドスケジューリングを単純に行え，ハードウェアによるスレッドスケジューリングを実現できた．

しかし，生産者・消費者問題に存在する排他制御では資源を利用しようとするスレッドが資源獲得を失敗した際に，獲得するためのビジーウェイトを行うとデッドロックが発生する可能性があるため，資源獲得に失敗したスレッドは再起動を余儀なくされ，スレッドの再実行を繰り返すために演算資源が無駄に消費される．その無駄な消費により，他スレッドの起動に遅延が生じる可能性がある．本稿ではデータ要求概念によりこの問題を解決するスレッド駆動制御を提案する．さらに，スレッドプログラムの記述性を考察する．

以下本稿では継続モデル，Fuce アーキテクチャについて述べ，データ要求概念とその評価を議論する．

2 Fuce アーキテクチャ

2.1 プログラミングモデル

Fuce アーキテクチャは関数とスレッドを中心にしたプログラミングモデルを実現する．スレッドは他からの干渉を受けずに排他的に実行する命令列として定義され，スレッド間の計算結果の通知を継続と定義する．スレッドは先行するスレッドから継続を全て受け取ると実行可能になる．関数は複数のスレッドで構成され，その関数の実行環境 (命令列とデータ領域) として関数インスタンスを持つ．

2.2 Fuce プロセッサ

Fuce プロセッサは，スレッドの実行管理を行う Thread Activation Controller (TAC)，複数のスレッド実行ユニット (Thread Execution Unit)，メモリをチップ上に搭載している．図 1 は Fuce プロセッサの概要図である．ここでは，スレッド実行ユニットと TAC に関して述べる．

スレッド実行ユニット

スレッド実行ユニットは，演算を行う演算ユニットとスレッドコンテキストを整えるプリロードユニットを備える．スレッド実行ユニットが演算ユニットでスレッドの実行を開始する際には，既にプリロードユニットによってスレッドコンテキストが整って

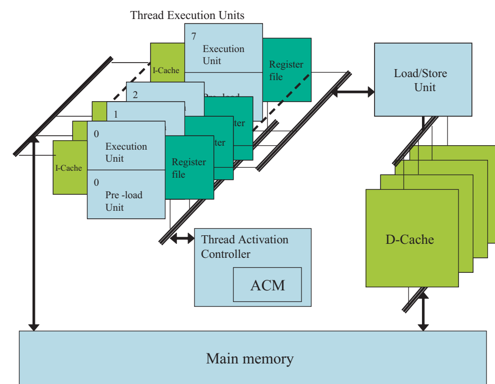

図 1: Fuce プロセッサ概要図

おり単純なパイプライン構造でもメモリアクセスに関するストールが起きにくい．

Thread Activation Controller

TAC はスレッド実行管理をハードウェアにより行う機能ユニットであり，Activation Control Memory (ACM) を用いてスレッド実行管理を実現する．図 2 に ACM の概要を示す．

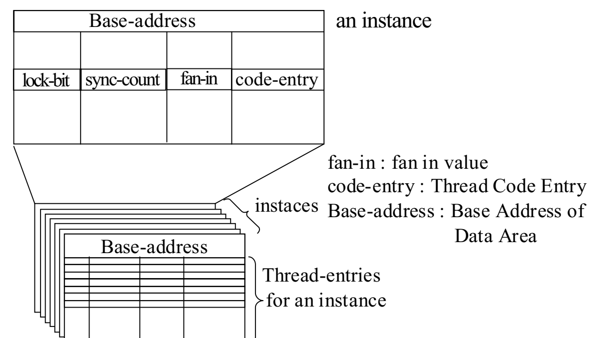

図 2: ACM の概要図

ACM は複数のページで構成される．ページは関数インスタンスに対応し，複数のスレッド管理情報と関数インスタンスのヒープ領域の先頭アドレス (Base-address) を保持する．スレッド管理情報は fan-in，sync-count，lock-bit，code-entry で構成される．fan-in はそのスレッドへ先行スレッドから継続される数である．sync-count は現在の継続される数であり，初期値は fan-in である．lock-bit は排他制御に利用し，初期値は 0 である．lock-bit が 1 の時にはそのスレッドへの継続は待たされる．code-entry はそのスレッドの命令コードの先頭アドレスである．

3 データ要求概念

筆者らは継続モデルに新たな概念として，データ要求概念を導入する．継続概念では先行スレッドに視点をおいてスレッド間の依存関係を継続として定義するが，データ要求概念では継続スレッドに視点をおいて先行スレッドへの継続を定義する．

図 3 はデータ要求概念を用いたスレッド A，B，C の実行の様子を示す．

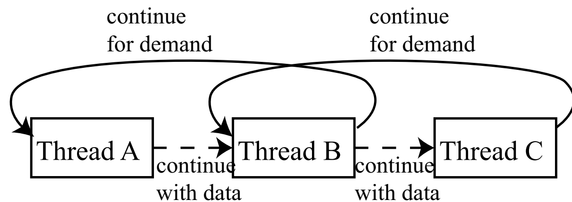

図 3: データ要求概念によるスレッド駆動制御

B は A の結果を必要とし，C は B の結果を必要としている．データ要求概念を用いてこれら 3 つのスレッドを実行するためには，C は B に対して計算結果の要求を送り，B は A に計算結果の要求を送る必要がある．その後計算結果の要求を受けた A は計算終了の後に B へ計算結果を送り継続する．同様に B は C へ計算結果を送り継続する．データ要求概念では計算結果の要求についても継続と定義する．さらに結果の要求と通知に関する継続を二つに分割することで，計算結果の要求のための継続を要求駆動の継続とし，計算結果の通知のための継続をデータ駆動の継続と定義する．スレッドはどちらからの継続も同様に fan-in 値をデクリメントする．

3.1 排他制御

Fuce アーキテクチャでは資源を扱うスレッドに対してロック操作を試みる手法を用いた場合と要求駆動を用いた場合の二つの手法で，排他制御を実現できる．

(1) ロック操作による排他制御

ここでは，資源を扱うスレッドへロックを試みる手法で排他制御を行う．二つの先行スレッドが，ある一つの継続スレッドに継続する場合を考える．ロックを試みる手法は，ACM 中のスレッド管理情報の lock-bit を利用することで排他制御を実現する．図 4 に排他制御の動作モデル例を示す．

1. 先行スレッドは継続スレッドへテスト＆ロック操作を行う．

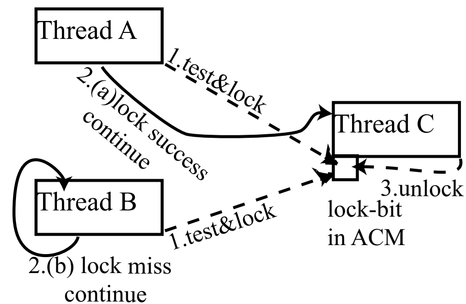

図 4: ロック操作による排他制御

2. **(a)** ロックに成功した場合

   継続スレッドへ継続する．

   (b) ロックに失敗した場合

   自スレッドを再起動し，再度ロックを試みる．

3. 継続スレッドはロックを解除し，終了する．

このように lock-bit を利用して排他制御を実現する手法をロック操作手法とよぶ．継続モデルではスレッドは終了するまで走り切るので，資源獲得を失敗した際に資源獲得のためのビジーウェイトを行うとデッドロックが発生する可能性があるので，スレッドは走行を一旦終了し再起動をする必要がある．そのために資源獲得を失敗したスレッドの再走行を繰り返すため演算資源が無駄に消費され，他スレッドの起動に遅延が生じる可能性がある．

(2) 要求駆動による排他制御

排他制御のロック操作で生じるスレッド再起動の排除のために，筆者らは要求駆動による排他制御を導入する．二つの先行スレッドが，ある一つの継続スレッドに継続する場合を考える．図 5 に要求駆動による排他制御の動作モデル例を示す．

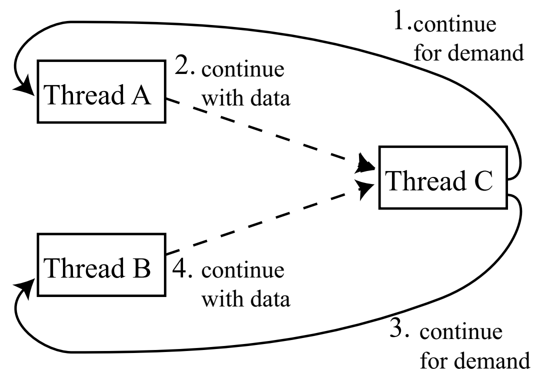

図 5: 要求駆動手法による排他制御

1. 継続スレッドは先行スレッドへ要求駆動の継続を行い，終了する．

2. 先行スレッドは継続スレッドへ計算結果を送り，データ駆動の継続を行う.
3. 継続スレッドは計算結果を用いて自身の処理を行う．その後，もう一方の先行スレッドへ要求駆動の継続を行い，終了する.
4. もう一方の先行スレッドは継続スレッドに計算結果を送り，データ駆動の継続を行う.

このように要求駆動による排他制御を行う手法を要求駆動手法とよぶ．この手法はロック操作を用いないため，ロック操作手法で可能性がある資源獲得を失敗したスレッドの再起動を排除する．しかしながら，要求駆動手法では継続スレッドから先行スレッドへ要求駆動の継続を行った後に，先行スレッドから継続スレッドへデータ駆動の継続を行うので，継続スレッドの起動に遅延が生じる.

4 要求駆動手法の評価

排他制御を含むスレッド・プログラム記述において要求駆動手法とロック操作手法を用いた際の記述の容易さを考察する．また，データ要求概念を適用した継続モデルに関し，排他制御のためのロック操作の削減によるスループット向上とスレッド起動の遅延について評価する.

4.1 スレッド・プログラム記述

図 6 にロック操作手法を排他制御に用いたスレッドの疑似ソースコードを示し，図 7 に要求駆動手法を排他制御に用いたスレッドの疑似ソースコードを示す．ここではどちらの疑似ソースコードも排他制御に着目し，理解しやすい表現にしている.

```
Thread
    test&lock(next_Thread);
    if(lock == failure)
      jump RETRY_AGAIN_PART ;
    continue(next_Thread);
    unlock(myself);
    end;

RETRY_AGAIN_PART:
    continue(myself);
    end;
```

図 6: ロック操作手法を用いた疑似コード

ロック操作手法では資源獲得を失敗した際のスレッドの再起動を考慮した記述が必要となる．一方，要求駆動手法ではスレッド起動時に先行スレッドとの同期が取れ資源を獲得できているので，ロック操作手法に比べてスレッドの再起動を考慮した記述は必要なく，自身の先行スレッドへの要求駆動の継続と継続スレッドへのデータ駆動の継続のみを行う.

```
Thread
    continue(next_Thread);
    SELLECT_precede_Thread_BLOCK
    continue(precede_Thread);
    end;
```

図 7: 要求駆動手法を用いた疑似コード

4.2 性能評価

性能評価は VHDL にて記述した Fuce プロセッサを ModelSim 上で動作させて行った．今回利用した Fuce プロセッサのシミュレーション環境を，表 1 に示す.

表 1: Fuce プロセッサ構成

スレッド実行ユニット	1～8 本
メモリアクセスレイテンシ	20，60 クロック
TAC スループット	1 クロック

ベンチマークプログラムとして，2048 個のデータ列を整列するマージソートプログラム (Merge) と 2048 要素を処理する基数 2 の一次元 FFT プログラム (FFT) を利用する．要求駆動手法によるスループット向上効果の評価のために，一般的なアルゴリズムに加えてスレッド間パイプライン並列実行方式 [5] を利用したプログラムを用いた．この方式は，ハードウェアのパイプライン構造のようにスレッドが動作することでパイプライン並列性を抽出する．その際，スレッド間の実行を排他制御によって制御し，プログラムの正しい動作を保障する.

マージソートプログラムでは開始時には多数の関数が並列に実行できるが，処理が進むにつれて並列に実行できる関数の数が減少する．結果として，プログラム全体から高い並列性を利用することが期待できない．また，マージソートプログラムの特性としてスレッド間パイプライン並列実行方式では関数は二つの先行する関数の一方からデータを受取り実行を開始するため，もう一方の関数はデータを送ることが可能になるまで待つ必要がある.

マージソートプログラムを一般的なアルゴリズムにしたがってマルチスレッド実行するプログラムを Standard_Merge とし，スレッド間パイプライン並列実行方式を適用し排他制御にロック操作手法を用い

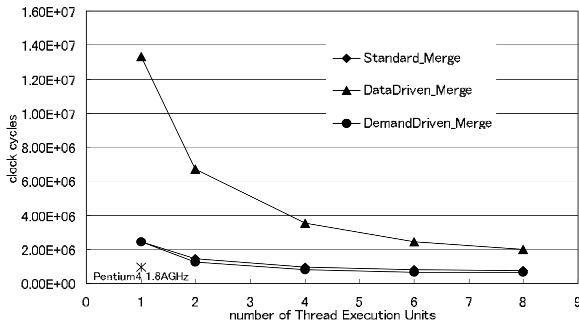

図 8: Merge における総実行クロック数：メモリアクセスレイテンシ 20 クロック

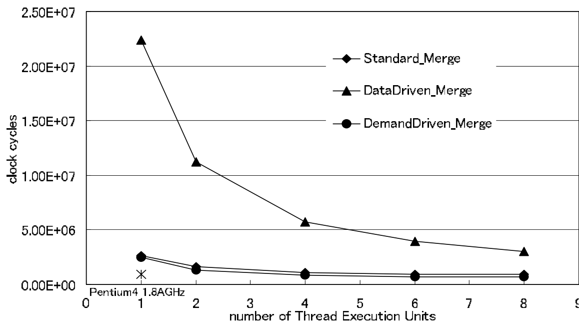

図 9: Merge における総クロック数：メモリアクセスレイテンシ 60 クロック

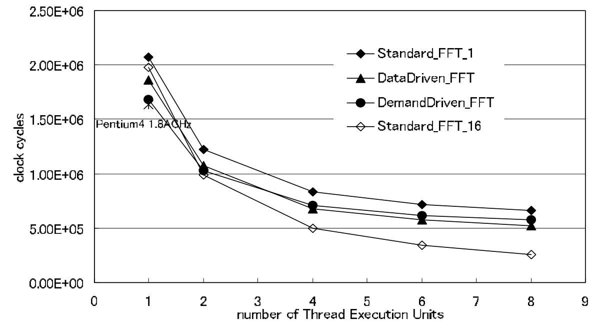

図 10: FFT における総実行クロック数：メモリアクセスレイテンシ 20 クロック

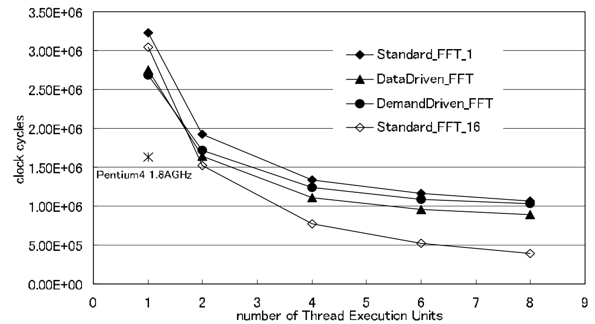

図 11: FFT における総実行クロック数：メモリアクセスレイテンシ 60 クロック

たプログラムを DataDriven_Merge とし，要求駆動手法を用いたプログラムを DemandDriven_Merge として作成した．

FFT プログラムは多数の関数が並列に実行でき，関数内に備わるデータ並列性を利用できる．FFT プログラムの特性としてスレッド間パイプライン並列実行方式では一つの関数は二つの関数に継続する．そのため，多数の関数が同時に並列実行でき，パイプライン並列性を十分に利用できる．

スレッド間パイプライン並列実行方式と比較するため，一般的なアルゴリズムを利用して関数内で 1 スレッドが走行し関数間の並列実行のみを利用した FFT プログラムを Standard_FFT_1 として作成した．さらに，関数間の並列実行に加えて，データ並列性を活かすために関数内で最大 16 スレッドを同時に並列実行するプログラム (Standard_FFT_16) を作成した．また，スレッド間パイプライン並列実行方式を適用し，排他制御にロック操作手法を用いた FFT プログラムを DataDriven_FFT とし，要求駆動手法を用いたプログラムを DemandDriven_FFT として作成した．

また，Fuce プロセッサの性能を従来の逐次処理プロセッサと比較するために，Pentium4 の 1.8AGHz を用いてシングルスレッド実行性能を測定した．

図 8，9 にメモリアクセスレイテンシ 20，60 クロック時の Merge における総実行クロック数を示し，図 10，11 にメモリアクセスレイテンシ 20，60 クロック時の FFT における総実行クロック数を示す．さらに，図 8，9，10，11 に Pentium4 の 1.8AGHz を用いて測定した実行クロック数を載せる．

DemandDriven_Merge は DataDriven_Merge に比べて最大 8.96 倍の性能向上を得，Standard_Merge に比べて最大 1.31 倍の性能向上を得た．DataDriven_Merge はロック操作において 98%以上の操作を失敗し，ロック操作に失敗したスレッドの再実行を繰り返すため演算資源をが無駄に消費され続け，実行性能を著しく悪化した．一方，DemandDriven_Merge は要求駆動手法により全てのロック操作を排除したので，ロック操作の失敗によるスレッドの実行が起きない．そのため，DemandDriven_Merge はスレッド起動の遅延が生じているにもかかわらず，ロック操作削減によるスループット向上により本来備えているパイプライン並列性を十分に利用できた．また，スレッド実行ユニットが 1 本の時と比較して，8 本

の時に Standard_Merg，DataDriven_Merge，DemandDriven_Merge はそれぞれ最大 3.37 倍，最大 7.39 倍，最大 3.93 倍の性能向上を得た．

DemandDriven_FFT は DataDriven_FFT に比べて最大 1.16 倍の性能悪化をし，Standard_FFT_1 に比べて最大 1.23 倍の性能向上を得た．DataDriven_FFT ではある関数が二つの関数に計算結果を送るとともにそれぞれにデータ駆動の継続を行い，再度起動して二つの関数に対して同様のことを行う．その後，二つの関数がそれぞれ 2 度の継続を受け終わって初めて実行可能になる．そのため，関数が二つの関数にロック操作をした際には，二つの関数は実行を既に終了しロックを解除している．その結果ロック操作において失敗せず，全てのロック操作に成功しパイプライン並列性を利用でき性能改善をした．一方，DemandDriven_FFT でも要求駆動手法により全てのロック操作を排除しているが，元々ロック操作で失敗しないためにスループット向上は得られず，スレッド起動で生じた遅延が実行性能を僅かに悪化させた．また，スレッド実行ユニットが 1 本の時と比較して，8 本の時に Standard_FFT_1，Standard_FFT_16，DataDriven_FFT，DemandDriven_FFT はそれぞれ最大 3.14 倍，最大 7.68 倍，最大 3.57 倍，最大 2.90 倍の性能向上を得た．

今回作成したスレッドプログラムではマージソートプログラムのスレッドの粒度が FFT プログラムのスレッドの粒度に比べて非常に大きい．そのため，メモリアクセスレイテンシを隠蔽できる効果が大きい．その結果として，マージソートプログラムが FFT プログラムに比べてメモリアクセスレイテンシが増加しても性能悪化を抑えることができた．

ちなみに，Pentium4 1.8AGHz と比較した場合，Pentium4 1.8AGHz においてはそれぞれのプログラムに必要なデータサイズがキャッシュに全て載ることができるサイズなのでキャッシュ効果が効いていることを考慮に入れても，Fuce プロセッサ (8 個のスレッド実行ユニット構成) はメモリアクセスレイテンシが 20 クロックの時にマージソートプログラムでは実行クロック数に換算して，最大 1.50 倍の速度を得，FFT プログラムでは最大 6.32 倍の速度を得た．メモリアクセスレイテンシが 60 クロックの時にマージソートプログラムでは最大 1.32 倍の速度を得，FFT プログラムでは最大 4.11 倍の速度を得た．

5　おわりに

我々は本稿で新たにデータ要求概念を提案し，それを用いることで排他制御におけるロック操作を排除できることを示した．排他制御が実行性能に大きな影響を及ぼしているときは，データ要求概念を用いた排他制御によりスループットが向上できることを示した．また，スループット向上を得ることが困難なプログラムではスレッド起動の遅延が生じて実行性能に影響したが，若干の性能悪化に抑えることができた．さらに，スレッド・プログラム記述の観点からロック操作に比べ容易にプログラミングできるのでデータ要求概念が有効なことがわかった．

データ要求概念の問題点は，継続スレッドにとって自身に継続する先行スレッドが特定されている場合に活用できるが，特定できない場合にデータ要求概念を利用することが困難なことである．そのため，ロック操作を用いた排他制御とデータ要求概念による要求駆動を用いた排他制御の両方をその特性に合わせて利用することが必要である．

今後はオペレーティングシステムにおいてサービスの要求時のプログラム起動にデータ要求概念を適用することを検討し，データ要求概念を適用した継続モデルによるオペレーティングシステム構成法を追求していく．

謝辞

本研究は科研費基盤研究 (A)(2)「細粒度マルチスレッド処理原理による並列分散処理カーネルウェアの研究」(課題番号:15200002) と九州大学における 21 世紀 COE プログラム「システム情報科学で社会基盤システム形成」の若手研究者のための研究助成の一環として行ったものである．

参考文献

[1] Amamiya, M.,”A New Parallel Graph Reduction Model and its Machine Architecture ”, Data Flow Computing: Theory and Practice, Ablex Publishing Corporation, pp.445- 467, (1991).

[2] Amamiya, M., Taniguchi, H. and Matsuzaki, T.,”An Architecture of Fusing Communication and Execution for Global Distributed Processing ”, Parallel Processing Letters, Vol.11, No.1, pp.7-24, (2001).

[3] D.W. Wall, “Limits of Instruction-Level Parallelism,” Proc. Fourth I nt'l Conf. Architectural Support for Programming Languages and Operating Systems, pp. 176-188, 8-11 Apr. 1991.

[4] Lo, J. L., Eggers, S. J. , Emer, J. S., Levy, H. M., Stamm, R. L. and Tullsen, D. M.,”Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading ”, ACM Transactions on Computer Systems, Vol.15, No.3, pp.322-354, 1997.

[5] 泉 雅昭, 雨宮 聡史, 松崎 隆哲, 雨宮 真人, “継続モデルに基づくスレッドプログラミング手法の提案”, 情報処理学会研究報告，2004-ARC-159，pp.79-84(2004).