2004－ARC－159　(14)
2004／7／30

# 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

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による並列プログラムを効率的に実行する。本稿では、Fuce プロセッサ上での実行を意識したスレッドプログラミングの手法を議論する。具体的には、スレッド分割の指針とスレッド間パイプライン並列実行の性能評価をする。

# Thread Programming Technique Based on Continuation Model

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 discusses a method of continuation-based thread programming with an aim to exploit parallelism on the Fuce processor. A guideline for dividing a program into threads is discussed, and the programming method for exploiting thread level pipeline parallelism is explained. Then, we evaluate the performance of the thread level pipeline-parallel execution on the Fuce processor by the software simulation tool.

## 1　はじめに

近年の代表的なプロセッサ技術として、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がある。スーパスカラプロセッサは、プロセッサの性能向上に対して多大な貢献をしてきた。しかし、単一プロセス実行または単一スレッド実行における命令レベルの並列性の抽出には限界があり、スーパスカラプロセッサを十分に活かしきれていない。一方、複数のプロセスまたは複数のスレッドを同時実行させて、スループットの向上をはかる SMT (Simultaneous Multi Threading) プロセッサ [3] が研究されている。また、近年の半導体技術の進歩によって、プログラムの実行をする複数のコアをチップに搭載した CMP (Chip Multi Processor)[4] の研究も進められている。CMP は、コアを複数搭載するので複数のプロセスまたは複数のスレッドを同時に実行させることが可能となる。

しかし、SMT プロセッサや CMP では、OS が管理するスレッドのスケジューリングにはオーバヘッドが存在する。このオーバヘッドのために、マルチスレッド実行を可能にしているプロセッサの性能を十分に活かせていない。さらに、プログラムは命令列

の制御関係に基づいて記述されているために、データの依存関係によって実行するデータフローモデルと比較すると、並列性を十分に抽出することが困難である。

そこで、筆者らは並列処理と親和性の高いデータフローモデルを基盤にして、スレッドの並列実行を追求した Fuce プロセッサ [2, 7] を開発してきた。Fuce プロセッサは複数のスレッド実行ユニット (Thread Execution Unit) を搭載した CMP である。Fuce プロセッサは、スレッド実行管理をハードウェアで行い、ソフトウェアのスレッド実行管理コストを軽減している。Fuce プロセッサは継続モデルに基づいたプログラミング・モデルを採用している。本稿では継続モデルに基づくスレッドプログラミング手法を提案する。

以下本稿では、Fuce アーキテクチャについて述べ、継続概念によるプログラミングの技法とその結果について議論する。

## 2 Fuce アーキテクチャ

筆者らは継続概念 [1] に基づく Fuce アーキテクチャの研究を進めてきた。Fuce アーキテクチャでは、以下のことを定義している。

・関数インスタンス

関数は複数のスレッドとしてプログラムされ、その関数の実行環境 (命令列とデータ) として関数インスタンスを定義する。

・スレッド

他からの干渉を受けずに中断することなく実行できる命令列をスレッドと定義する。スレッド中の演算は基本的に全てレジスタを用いて行われる。

・継続

スレッドとスレッドの間に実行順序が存在する。スレッドは計算結果をその計算を引き継ぐスレッドに渡し通知する。この通知を継続と定義する。スレッドは、全ての先行スレッドからの継続が終了した後に実行可能となる。

この章では、Fuce アーキテクチャを構成する Fuce プロセッサとプログラミング･モデルについて述べる。

### 2.1 Fuce プロセッサ

Fuce プロセッサは、スレッド実行管理を行う Thread Activation Controller (TAC)、複数の Thread Execution Unit (TEU)、メモリを搭載している。Fuce プロセッサはチップ上でこれら全てを搭載する。この結果として、Fuce プロセッサは各機構を高バンド幅のバスによって接続し、各機構間の大容量データ転送を可能とする。図 1 は Fuce プロセッサの概要図である。

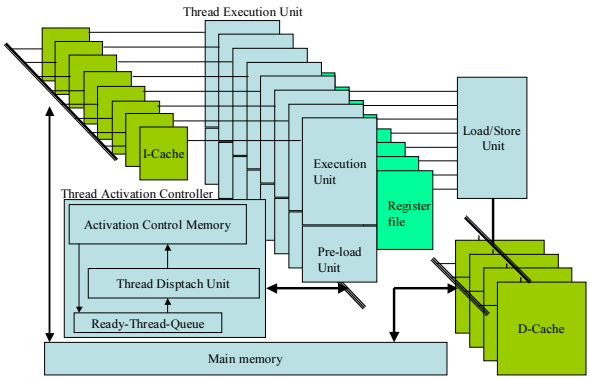


図 1: Fuce プロセッサ概要図

TEU 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。TEU 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。

ここでは、スレッド実行管理を行う TAC に関して述べる。

**Thread Activation Controller**

Thread Activation Controller (TAC) は、スレッド実行管理をハードウェアによって高速に行う。TAC は Activation Control Memory (ACM)、Thread Disptach Unit (TDU)、Ready-Thread-Queue (RTQ) で構成する。図 2 に ACM の概要を示す。

ACM は複数のページで構成する。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 で構成する。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 の初期値は fan-in である。lock-bit は排他制御に利用し、初期値は 0 である。仮に、lock-bit が 1 ならばそのスレッドは他のスレッドからの継続がロックされている。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。

TDU は、スレッド実行管理を行う命令 (継続命令) に従って ACM にアクセスし、実行可能となったスレッドを RTQ にエンキューする。また、空き状態の TEU が存在するときには RTQ のスレッドをデキューして、その TEU に渡す。

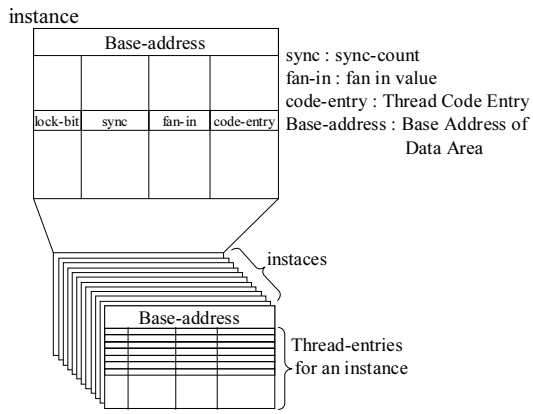


図 2: ACM 概要図

TAC は、TEU の発行する以下の継続命令によりスレッドの実行管理を行う。

・cont　thID

　thID で指定した継続スレッドの sync-count をディクリメントする。その際に、sync-count が 0 になれば、その継続スレッドを実行可能にする。

・rcont　thID

　thID で指定した継続スレッドの lock-bit を 0 にし、sync-count を (fan-in − 1) にする。その際に、sync-count が 0 になれば、その継続スレッドを実行可能にする。

・tsacm　Rd, thID

　thID で指定した継続スレッドの lock-bit が 1 ならば Rd に 1 を入れる。lock-bit が 0 ならば Rd に 0 を入れ、指定したスレッドの lock-bit を 1 にする。

### 2.2　プログラミングモデル : 排他制御の例

Fuce プロセッサでは、tsacm 命令を利用して排他制御を行う。

まず、継続スレッドのテスト&ロックを試みるために tsacm 命令を発行する。継続スレッドが他のスレッドからロックされている (lock-bit = 1) ときは、再度継続スレッドのテスト&ロックを試みる。継続スレッドがロックされていない (lock-bit = 0) ときはロックし、継続スレッドへ継続する。このとき、継続スレッドはロックされるので、このスレッドのロックが解かれるまで他のスレッドから継続されることはない。

## 3　スレッドプログラミング手法

Fuce アーキテクチャではプログラマはデータの流れに注目して、継続モデルに基づき自然にプログラムを記述することが可能となる。また、コンパイラはスレッド・レベルの並列性を抽出するためにデータの流れに注目して複数のスレッドに分割する。

本節では、プログラム中のデータ依存関係を抽出することにより複数のスレッドに分割する指針を示す。また、スレッド間パイプライン並列実行によってストリーム処理のようなプログラムから並列性を可能な限り抽出することができることを示す。

### 3.1　スレッド分割

Fuce プロセッサの TEU は、プリロードユニットを利用してメモリアクセスレイテンシを隠蔽する。プリロードユニットを効果的に利用するために、スレッドの先頭部分にメモリアクセスレイテンシを生じるロード命令を置き、スレッドの本体は残りの命令で構成するのが望ましい。

スレッド実行に必要なレジスタが不足する場合データのロード、ストアが必要となるが、スレッド本体の命令列中にロード命令があると、演算ユニットでメモリアクセスが生じる。この場合、メモリアクセスレイテンシを隠蔽するプリロードユニットを活用できずにストールを起こしスループットが低下する。そこで、プリロードユニットを活用するためにスレッドを分割する。

スレッド分割の指針は、算術命令列の途中で新たにメモリからデータを読み込む必要が生じるときは、そこを分割点にしてスレッドを二つのスレッドに分割することである。図 3 にスレッドを分割する例を

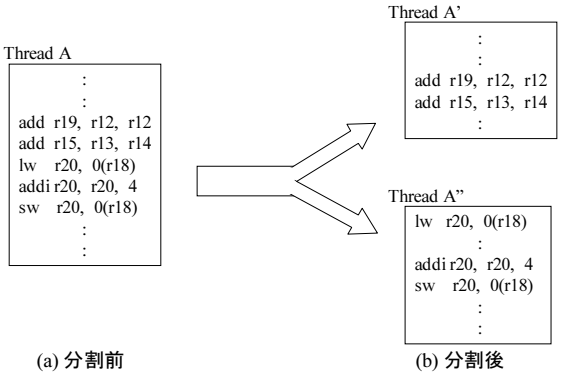


図 3: スレッドの分割

挙げる。スレッドを分割することにより新たなメモリアクセスをプリロードユニットで実行し、メモリアクセスレイテンシを隠蔽することができる。図 4 にスレッドを分割した際のプリロードユニットによる隠蔽の効果を示す。コンパイラはスレッド分割を

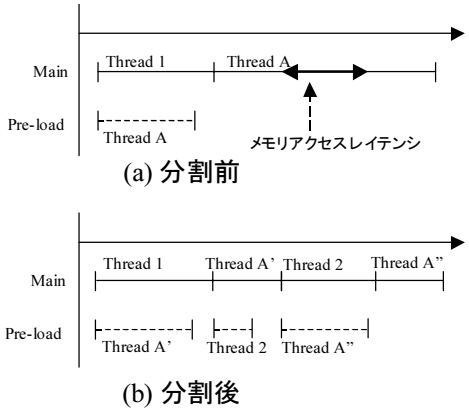


図 4: スレッドの分割による隠蔽効果の利用

繰り返して、プリロードユニットを活用できるようなスレッドプログラムコードを生成する。

また、他のスレッドからの結果待ちおよび外部からのイベント待ちの場合にもスレッドを分割する。この場合、結果待ちやイベント待ちの起こりうる箇所を分割点にしてスレッドを分割する。スレッドを分割することで、結果待ちやイベントの同期を取るための無駄な実行を減らすことができる。

### 3.2 スレッド間パイプライン並列実行

スレッド間パイプライン並列実行とは、各スレッドの実行がパイプライン動作するように制御することである。図 5 に、スレッド間パイプライン並列実

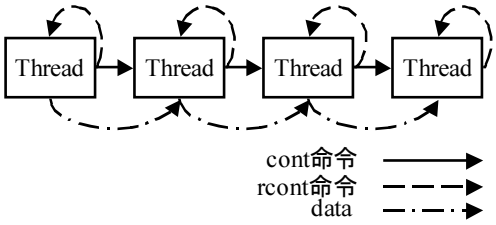


図 5: スレッド間パイプライン並列実行の様子

行の様子を示す。

スレッド間パイプライン並列実行は以下のように行われる。最初に、tsacm 命令によって継続スレッドのテスト&ロックを試みる。ロックされていないときは継続スレッドにデータを渡し、継続する。次に、自分自身に rcont 命令を発行し、ロックを解除して次のデータが渡されるのを待つ。仮に、継続スレッドが他のスレッドによってロックされているならば継続スレッドに継続できないので、再度継続スレッドのテスト&ロックを試みる。この動作を排他実行するスレッドへ継続する各スレッドが繰り返す。

たとえば、ストリーム処理を行うプログラムでは、図 5 のようにスレッド間パイプライン並列実行を行うとパイプライン並列性を抽出することができる。先行スレッドから引き渡されたデータを後続スレッドが処理している間に、実行可能となった先行スレッドは新たなデータを受け取り、そのデータの処理が行えるからである。

### 3.3 プログラム例:エラトステネスの篩い

プログラム例として、エラトステネスの篩い (以下 Sieve プログラムという) を対象にして行う。Sieve プログラムは、自然数列を生成する GENERATOR 関数と SIEVE(i) 関数で構成される。SIEVE(i) 関数は、i 番目の素数でデータを篩い落とす関数である。

従来モデルで Sieve プログラムを記述したときには以下の問題が生じる。

- バッファ
  ストリーム処理の場合には関数間にバッファを置く必要がある。
- 制御依存
  いつデータが到着しているのかを SIEVE 関数自身は知ることができないので、SIEVE 関数はバッファの中にデータが到着しているか否かを常に調べる必要がある。そのため、SIEVE(i) 関数と SIEVE(i+1) 関数の間でバッファへのアクセスを排他制御する必要がある。

一方、継続モデルで Sieve プログラムを記述した場合、従来モデルで生じた問題を以下の特徴を活かして解消できる。

- バッファレス
  継続モデルでは SIEVE 関数自身を排他制御し、データを直接後続の SIEVE 関数へ渡す。よって、SIEVE 関数同士の間には排他制御されるバッファを必要としない。
- データ依存
  後続の SIEVE 関数にデータを直接渡し継続することができる。よって、データが到着した段階で、SIEVE 関数はただちに実行可能となる。

## 4 継続モデルによるプログラムの評価

図 6 は、継続モデルに従って作成した Sieve プログラムの SIEVE 関数のコード例である。

この二つのモデルに関して、作成したプログラムのコード量、プログラムの実行終了までに要したクロック数、そのときの IPC を比較した。なお、各ス

```
SIEVE:
        lbq     r5, 0(r3)
        lbq     r9, 16(r3)
        lbq     r13, 0(r4)
        lbq     r17, 16(r4)
                #  プリロードユニット部分

        beq     r5, r6, プログラム終了部へ
        div     r14, r15
        mfhi    r5
                # r5 = 剰余
RETURN_WAIT_DATA:
        beq     r0, r5, 素数か倍数かの判定部へ
        beq     r0, r17, SIEVE関数生成部へ
TRY_NEXT:
        tsacm   r6, r19
        bne     r0, r6, TRY_AGAIN
                # ロック不可

        sw      r14, 0(r20)
                # SIEVE(i + 1) data area<= data
        cont    r19
                # SIEVE(i + 1)へ継続
END_ SIEVE :
        rcont   r18
                # SIEVE(i)へrcont命令発行 (ロック解除)
        end
```

図 6: コード例:SIEVE 関数

レッドのスケジューリングは Fuce プロセッサの TAC が自動的に行う。スレッド切り替えは TAC が 2 クロックで行うので、実行時間のほとんどがプログラムの実質的な実行時間である。ただし、現在 Fuce プロセッサで利用できるコンパイラをまだ開発していないので、Fuce アセンブラ語でプログラムを作成した。

### 4.1 プログラムコード

従来モデルで記述する場合、制御の流れを意識して記述しなければならない。また、Sieve プログラムにおいてはデータを渡すために各 SIEVE 関数の間に排他制御されるバッファが必要である。どうしても複雑な制御の流れを意識せねばならず、プログラムコードが複雑になりがちである。一方、継続モデルでは、SIEVE(i) 関数は SIEVE(i+1) 関数に直接データを渡し継続するので、図 6 のように簡潔にプログラムが記述できる。

従来モデルと継続モデルに従った場合のコード量を比較した結果を表 1 に示す。

| | 行数 |
|---|---|
| 従来モデル | 320 |
| 継続モデル | 142 |

表 1: アセンブリコード量の比較

継続モデルに従った場合、従来モデルに従った場合の半分以下のコードでプログラムを作成することができた。

### 4.2 プログラム実行

8 個の TEU を備える Fuce プロセッサを VHDL で実装し HDL シミュレータ (ModelSim) を用いて評価実験を行った。実験では、理想的なスレッド間パイプライン並列実行の場合に Fuce プロセッサの性能をどの程度引き出せるかを調べるために、Sieve プログラムに加えて単純なストリーム処理のプログラム (以下、Simple stream プログラムとよぶ) を実行させた。Simple stream プログラムは Sieve プログラムと同じ構造であるが、素数で篩い落とす代わりに先行関数から受け取った整数に 1 を加えてその結果を後続関数に引き渡す関数で構成されており、継続モデルに従って作られている。Sieve プログラムは 100 個の素数を生成させ、Simple stream プログラムは 100 個の整数を生成させる。Sieve プログラムのアルゴリズムは途中で素数でないデータが篩い落とされるので、並列性が顕著でない。一方、Simple stream プログラムは、整数が Sieve プログラムのように篩い落とされることがないので並列性を十分に抽出できる。

図 7 にメモリアクセスレイテンシを変化させた場合の各プログラムの IPC を示す。図 7 に Simple stream プログラムの実行結果を Simple stream、継続モデルの結果を Sieve Without Queue、従来モデルの結果を Sieve With Queue としてグラフに示す。

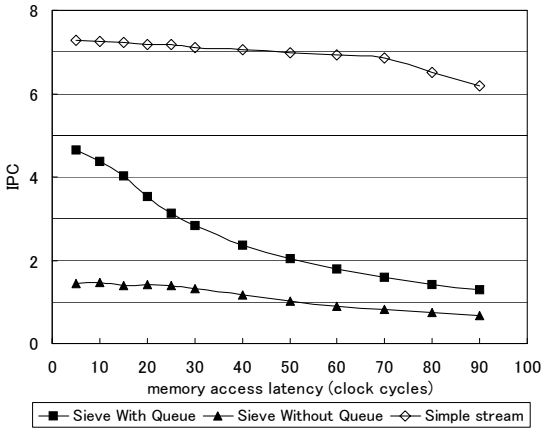


図 7: 実行評価 : IPC

また、図 8 に、メモリアクセスレイテンシを変化させた場合のプログラム実行終了までに要したクロック数を示す。

図 7 を見ると、Simple stream プログラムの IPC は 7 程度で非常に高い。一方、Sieve プログラムの IPC は低い。さらに、継続モデルに従った Sieve プ

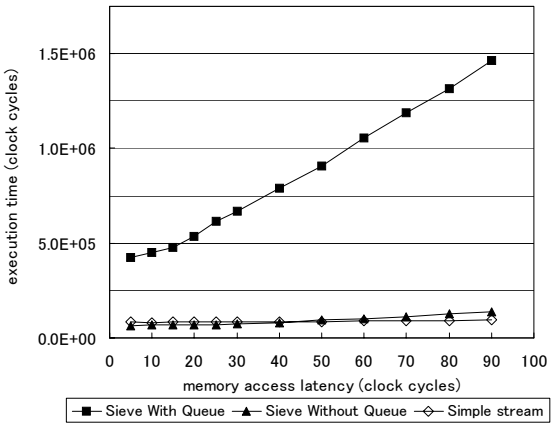


図 8: 実行評価 : クロック数

ログラムの IPC は、従来モデルに従ったプログラムの 52.9%以下である。しかし、図 8 に示すように、継続モデルに従った Sieve プログラムの実行終了までに要したクロック数は従来モデルの 9.2〜15.2%である。

従来モデルに従ったプログラムでは並列実行可能な多くのスレッドが常に存在する。全ての TEU でスレッドを実行しており、結果として、従来モデルに従ったプログラムにおいて IPC は高い。しかし、自然数の $\frac{2}{3}$ が SIEVE(1) 関数と SIEVE(2) 関数でシーブされ後続の SIEVE 関数へ届かないため、多くのスレッドはバッファチェックのために無駄な命令実行のクロックを費やしている。一方、継続モデルに従ったプログラムにおいて IPC は低いが実行終了までに要したクロック数は非常に小さく、Simple stream プログラムと同様に高速に処理されている。

さらに、TEU の台数効果を見るために 1 個の TEU を備えた Fuce プロセッサと 8 個の TEU を備えた Fuce プロセッサの両方でメモリアクセスレイテンシを 50 クロックとして、継続モデルの Sieve プログラムと Simple stream プログラムについて性能の向上を比較した。8 個の TEU を備えた Fuce プロセッサにおける Sieve プログラムの実行時間は 1 個の TEU を備えた Fuce プロセッサでの実行時間の 4.16 倍の速度向上をしており、Simple stream プログラムの実行時間では 6.87 倍の速度向上をしていた。

Simple stream プログラムは並列性を十分に抽出できるため TEU を増やした時の速度の向上は顕著に現れる。Sieve プログラムはデータ並列性が低いにもかかわらず、スレッド間パイプライン並列実行をすることによってパイプライン並列性が抽出されるため、TEU を増やした時に速度の向上が見られる。

## 5 おわりに

本稿では、継続モデルに基づくスレッドプログラミング手法としてスレッド分割の指針とスレッド間パイプライン並列実行の性能評価を示した。スレッド分割は、データ依存関係および外部のイベント待ちや他スレッドからの結果待ちの箇所を分割点にしてスレッドを分割する。また、スレッド間パイプライン並列実行では、データ並列性の小さいプログラムから可能な限りのパイプライン並列性を抽出する。

プログラム例としてエラトステネスの篩いを用い、従来モデルと継続モデルに基づいたプログラムのコード量と実行時の性能を比較した。その結果、データの依存関係に注目して記述する継続モデルのほうが記述量が少ない。また、従来モデルに従ったプログラムではデータが到着していないのに無駄な実行を繰り返していることが多いのに対して、継続モデルに従ったプログラムではデータが届いたときのみ実行をするので効率のよい動作を行うことができる。

今後は、FPGA エミュレータによるさらに詳細な性能評価、外部イベントの並列処理など OS カーネルプログラミング法の開拓、Fuce プロセッサ用コンパイラの開発を進めていく。

**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) 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] 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.

[4] Sun Microsystems Inc. Throughput computinghttp://www.sun.com/processors/throughput/.

[5] 雨宮 聡史, 松崎 隆哲, 雨宮 真人, “排他実行マルチスレッド実行モデルに基づくオンチップ・マルチプロセッサの設計”, 情報処理学会研究報告，2003-ARC-155，pp. 51-56，(2003).

[6] 雨宮 真人 他, 通信・放送機構 (TAO) 研究成果報告書, 「情報通信網の基盤技術に関する研究」, 平成 15 年 3 月.

[7] 松崎 隆哲, 雨宮 聡史, 泉 雅昭, 雨宮 真人, “排他実行マルチスレッド実行モデルに基づく Fuce プロセッサの評価”, 情報処理学会研究報告，2004-ARC-158，(2004).