社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report
2003－ARC－155　(7)
2003／11／28

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

雨宮 聡史　松崎 隆哲　雨宮 真人
九州大学 大学院 システム情報科学研究院

現在，マルチスレッド実行を意識したプロセッサの開発が主流になりつつある．しかし，大多数のプロセッサは命令レベルの並列性の抽出を追求しているものである．我々は命令レベルの並列性の追求をやめて，スレッドレベルの並列性のみに焦点を当て，データフローモデルを基盤とし，継続概念を核としたマルチスレッド実行モデルを提案する．また，このモデルを実現するオンチップ・マルチプロセッサの構成および命令セットアーキテクチャについて提案する．

On-chip Multi-Processor Design based on the Exclusive Multi-thread Execution Model

Satoshi Amamiya, Takanori Matsuzaki, and Makoto Amamiya
Faculty of Information Science and Electrical Engineering,
Kyushu University

Nowadays, development of processors which support concurrent multi-thread execution is becoming mainstream. However, most of the processors are aimed for exploiting instruction level parallelism. We are taking another approach, and developing the processor focused only on thread level parallelism. Our processor is named Fuce, and it is based on the continuation model which is a variant of data flow computing model. In this paper, we introduce the programming model for Fuce and the architecture of Fuce.

1　はじめに

今日の半導体技術の進歩にともなって，現代の代表的なプロセッサ技術であり，命令レベルの並列性を重視したスーパスカラ型プロセッサの活躍には目を見張るものがある．しかし一方で，単一のプロセスまたはスレッド実行における命令レベルの並列性抽出には限界があり，スーパスカラ実行を活かしきれていないという意見もしばしば聞かれるようになった．そのため，複数のプロセスまたはスレッドの同時実行を可能にして，スーパスカラ技術を最大限活用することで，スループットの向上を狙った SMT(Simultaneous Multi-Threading) プロセッサが提案 [3] され，実際に実用化 [4] もされている．さらには，複数の SMT コアを一つのチップ上に実装したチップマルチプロセッサの開発 [6] も実際に行われている．

我々の研究グループでは，命令レベルの並列性の抽出は限界であるとの立場からこの追求をやめて，元来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のみを追求したプロセッサ Fuce[2] を開発して来た．Fuce は複数のスレッド実行ユニットを搭載したチップマルチプロ

セッサの一種である．Fuce におけるスレッドは継続概念 [1] によるイベント駆動によってのみ制御され，排他的に実行され中断されることが無いという特徴を持っている．

ところで，データフローの観点から Fuce スレッドの並列実行モデルと SMT の並列実行モデルは対局を成していると考えられる．なぜなら，Fuce ではスレッド間の関係は完全にデータフロー型であり，スレッド（の命令列）の実行は完全にノイマン型である．一方，SMT ではスレッドの並列性は命令レベルの並列性に分解されノイマン型のように実行されるが，実は各命令は内部のリザベーションステーションでデータフロー的に取り上げられ並列に処理される．つまり，Fuce のスレッドは表面的にはデータフローであるが，内部ではスレッドの実際の処理はノイマン型であり，SMT はその逆となっている．これは興味深い違いである．

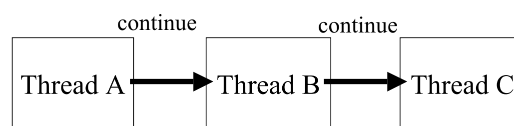
本稿では，Fuce の核となる概念である継続と，それを用いたプログラミングモデルを示し，そのモデルを実現するためのプロセッサアーキテクチャの構造を示す．

2　継続とスレッド

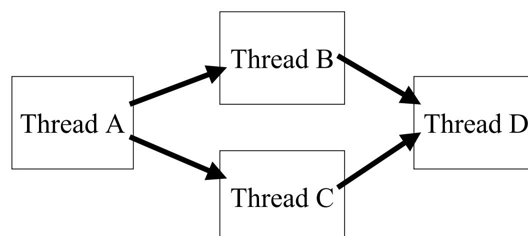
Fuce のスレッド並列実行モデルは，データフロー計算モデルに基づいた継続という概念を核として成り立っている．

継続とは，データフロー計算モデルでは 2 つの計算要素間の関係と定義され，Fuce のモデル上では，これはスレッド間の依存関係ということになる．図 1(a) は 3 つのスレッド A,B,C の依存関係を示している．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．これら 3 つのスレッドを実行するためには，A は計算結果とともに B に対して通知を送り，B は計算結果を C に通知しなければならない．我々はこの結果の通知というものを継続と呼び，A は B に継続し，B は C に継続すると言う．図 1(b) は継続するスレッドが複数の場合を示している．スレッド B と C は依存関係がないので並列走行が可能であり，スレッド D は B と C から継続されないと走行できない．

あるスレッドに対して継続される元のスレッドの数を fan-in 値，継続する先のスレッドの数を fan-out 値と呼ぶ．スレッド B の fan-out は 2 であり，スレッド D の fan-in は 2 である．



(a) Simple Continuation



(b) Multiple Continuations

図 1: スレッドと継続

Fuce のスレッド実行制御はすべて継続によって決まり，スレッドは継続されるたびに自 fan-in 値をデクリメントして，その値が 0 になった時に，実行可能となり発火・実行される．そして，そのスレッドは消滅するまでいかなる干渉も受けずに走行することができる．なお，概念上，スレッドは fan-out 値が 0 になるとき，つまり継続すべきスレッド全てに継続し終わったときに消滅する．

3　Fuce プログラミングモデル

Fuce では関数インスタンスを中心にしてプログラミングモデルを定義している．一般に，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がある．関数インスタンスの情報は Activation Control Memory (ACM) に登録される．ACM は図 2 に示すように，OS の仮想記憶システムと同様のページ構造となっている．ACM の各ページは 1 つの関数インスタンスの情報全てがテーブルに記録されている．記録される情報は，まず関数自体のデータ領域のアドレスである．関数内のスレッド情報は，スレッドの現在の fan-in 値，fan-in の初期値，そしてスレッドの先頭アドレスの 3 つの数値がスレッドの数だけ列挙されている．スレッドの ID は，基本的な仮想記憶システムと同様に，ACM のページ番号とページ内オフセットで表す．

なお，Fuce のスレッドは命令列としてプログラムされるが，その命令列は先頭部分は主にロード命

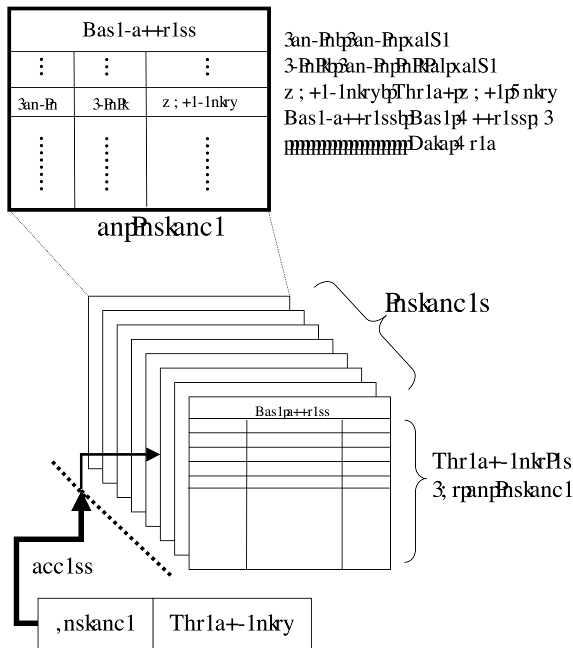


図 2: Activation Control Memory

令である．残りの部分は算術演算などとストア命令または継続命令である．この理由は，なるべくメモリアクセスせずに，レジスタ間のみで演算を行わせるためである．

Fuce プログラミングモデルを説明する上で必要となるマシン命令を定義する．

cont rD ID がレジスタ rD の値であるスレッドに継続する．対象スレッドの fan-in 値はデクリメントされる．対象スレッドの fan-in 値が 0 になったら，対象スレッドは発火される．

newins rD, func ACM に関数 func のエントリを確保し，その ID をレジスタ rD にセットする．

newda rD, func, size マクロ命令である．関数 func のデータ領域を size 分確保する．実際には動的メモリアロケータ関数を呼び出す．

delins rD ACM からレジスタ rD の値で示された ACM エントリを抹消する．

基本的には，これらの命令だけで，継続による関数コールと復帰，ループなどを記述することができる．

3.1 スレッド間のデータ渡し

継続とは単なる通知であり，スレッド間のデータの受け渡しは別途，メモリを介して行われる．図 3 はスレッド間でのデータの受け渡しの様子を示している．図 3 において，Thread1 と Thread2 は，それぞれ値をメモリに書き込んだ後，Thread3 に継続する．Thread3 は発火されると Thread1 と Thread2 が書き込んだ値をロードする．なお，これらのスレッドは同一インスタンスに属するので，それぞれのスレッドが使う変数のアドレスは，コンパイル時に解決することができる．

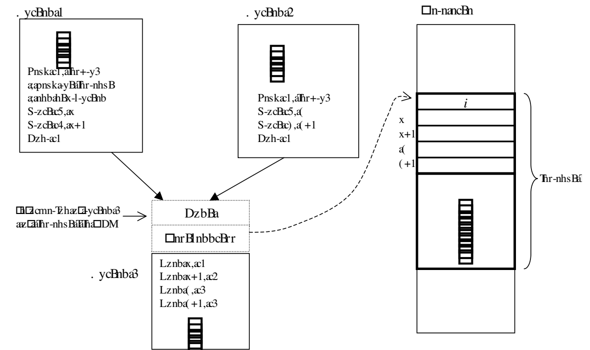


図 3: スレッド間のデータ受け渡し

3.2 関数コール

Fuce における関数は多数のスレッドによって実行されるので，従来の逐次実行型プロセッサのように，関数コールのためにスタックは使えない．そのため，図 4 に示すように，関数起動時には関数の

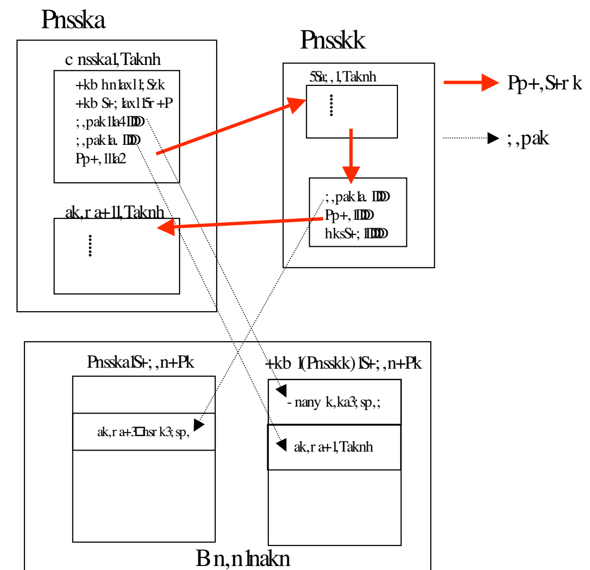


図 4: 継続による関数コール

データエリアが動的に確保される．データエリアは関数復帰時の継続先スレッド ID，パラメータ，戻り値，関数内局所データで構成される．

ところで，関数コールはスプリット・フェーズ方式で行われるので，関数呼び側は，他の関数を呼び出した後も計算を続けることができる．

関数コールは以下のように行われる．

1. 呼び側のスレッドは，呼び出す関数のためのデータエリアを newda 命令によって確保する．
2. 呼び側のスレッドは，呼び出す関数のための ACM エントリを newins 命令によって確保し，その ACM エントリに関数内のスレッドの情報をすべて登録する．
3. 呼び側のスレッドは，呼び出す関数のデータエリアのパラメータスロットに引数を書き込む．
4. 呼び側のスレッドは，呼び出す関数のデータエリアの復帰先スレッドスロットに関数復帰時の戻り先スレッドの ID を書き込む．
5. 呼び側のスレッドは，cont 命令によって呼び出す関数内の先頭スレッドに継続する．

3.3 関数からの復帰

関数からの復帰は以下の手順で行われる．

1. 呼び出された関数内のスレッドは，呼び出し側のデータエリアの戻り値スロットに関数の戻り値を書き込む．
2. 呼び出された関数内のスレッドは，cont 命令によって戻り先スレッドに継続する．
3. 呼び出された関数内のスレッドは，自関数のデータエリアを解放する．
4. 呼び出された関数内のスレッドは，delins 命令によって自関数の ACM エントリを ACM から抹消する．

3.4 ループ

Fuce プログラミングにおけるループの記述方法は主に２通りある．図５は２種類のループ実行の様子を示している．

一つはスレッド内で条件分岐を使う方法である．しかし，この方法では，一つのスレッドが長時間走行してしまい，プロセッサ資源を占有する可能性がある．Fuce プロセッサ上で多数のスレッドを並列実行させる上で障害となる可能性がある．

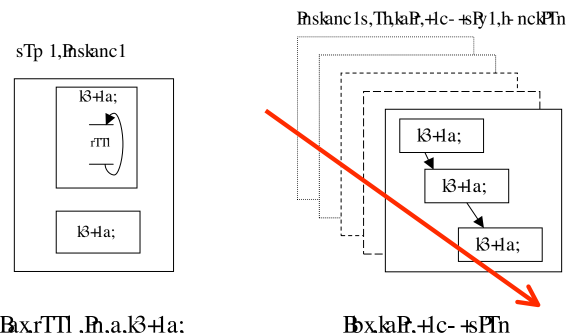


図 5: ２種類のループ

もう一つは末尾再起関数を使う方法である．末尾再起関数では，関数起動のたびに ACM のエントリと関数のデータエリアを確保する必要がなく，同一のものを使いまわすことができ，通常の関数起動に比べて高速である．また，この方法はスレッド内ループと異なり，スレッドの切り替わりが頻繁におこるので，スレッドがプロセッサ資源を長時間占有することはない．

3.5 排他制御と継続

OS の割り込み処理の一部やメモリアロケータなどのルーチンは，再入を防ぐために排他制御しなければならない．Fuce には排他制御のための継続命令が用意されている．

rcont rD ID がレジスタ rD の値であるスレッドに継続するが，まず対象スレッドの fan-in 値を f-init 値で初期化する．その後，対象スレッドの fan-in 値はデクリメントされる．

通常，cont 命令は fan-in 値が 0 でないスレッドに対して用いられるが，もし fan-in 値が 0 のスレッドに対して用いられた場合は，cont 命令はブロックされる．rcont 命令によって fan-in 値が初期化された後，ブロックされている cont 命令は直ちに実行される．

図 6 はメモリ割当などの典型的な排他制御ルーチンを示している．Thread E は rcont 命令によって自分自身に継続しており，それ以外のスレッドは cont 命令によって Thread E に継続している．Thread E

が rcont 命令を実行し終わるまで，他のスレッドの継続は待たされる．Thread E の fan-in 値が 2 の場合は，他のスレッドの継続は逐次的に実行される．

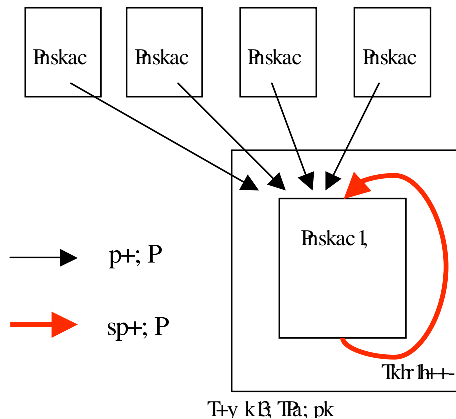


図 6: 排他制御と継続

4 Fuce プロセッサ

Fuce (Fusion of communication and execution) は，その名が示すように通信処理も通常の処理も同等に扱うという思想で設計されている．これは，すべてをスレッドとして統一的に処理するということを意味している．

Fuce におけるスレッドは他からの干渉を受けず，中断することなしに実行できる命令列として定義される．このような命令列の定義は TRIPS[5] でも使われているが，TRIPS の場合はスレッドではない．

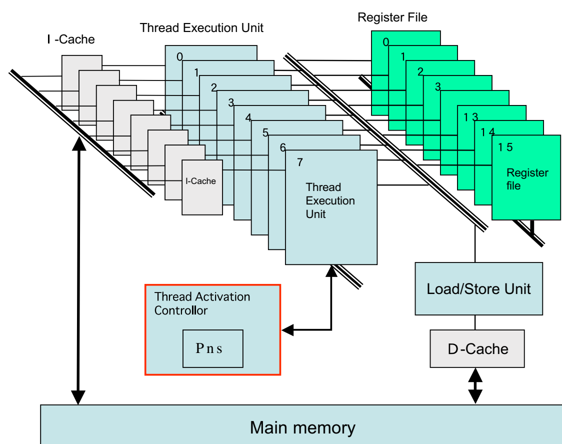


図 7: Fuce プロセッサ概要

図 7 に示すように，Fuce プロセッサは同一チップ上に複数のスレッド実行ユニットを搭載している．また，Fuce の通信処理は，マルチメディアデータのような容量の大きなデータの高速転送を主眼としているため，大容量メモリをプロセッサ本体に統合している．本稿で主張する点は排他実行スレッドの並列処理モデルであるため，以降メモリについては特に触れないので，Fuce のメモリ構造については参考文献 [7] を参照されたい．

並列処理の観点における Fuce の特徴は以下の 3 点である．

1. ８個のスレッド実行ユニット
2. １６個の大容量レジスタファイル
3. Thread Activation Controller

4.1 スレッド実行ユニット

スレッド実行ユニットはスレッド命令列を処理するユニットである．Fuce は 8 個のスレッド実行ユニットを搭載しているので，同時に 8 個のスレッドを実行することができる．スレッドを効率的に実行するために，スレッド実行ユニットはプリロードユニットと演算ユニットの二つで構成されている．演算ユニットは単純で古典的な 32 ビット RISC プロセッサであり，プリロードユニットは演算ユニットのサブセットとして，データのロードに関する命令のみサポートしている．

コンパイラの命令スケジュールによって，Fuce のスレッドをなす命令列は，最初の部分は主にロード命令であり，残りの部分はレジスタ操作の命令とストア命令となるようにコンパイルされることを前提としている．

プリロードユニットは，スレッド命令列の先頭部分を処理するものであり，演算ユニットは，残りの部分を処理するユニットである．これら 2 つのユニットは，レジスタファイルを介してパイプライン動作する．具体的には，演算ユニットであるスレッドを実行している間に，プリロードユニットは別のレジスタファイルを使って別のスレッドの処理を開始できる．このことによって，スレッド実行中のメモリアクセスレイテンシを隠蔽することが可能である．

4.2 レジスタファイル

スレッド実行ユニット内の 2 つのユニットを，できるだけストールさせずにパイプライン動作させ

るために，プロセッサ内部には，１６個のレジスタファイルが用意されている．そのため，走行中のスレッド実行ユニットが使用していないレジスタファイルを使って，プリロードユニットは，演算ユニットに先行してスレッドの処理を開始できる．また，各レジスタファイルは１２８個のレジスタで構成されているため，コンパイラによるスレッド命令列の命令スケジュールは容易である．

4.3 Thread Activation Controllor

Thread Activation Controllor (TAC) とはスレッドや継続に関係する機能ユニットであり，Fuce プロセッサの核心部分である．前述の ACM は TAC 内部にキャッシュメモリのような高速メモリとして実装されている．TAC はスレッド実行ユニットで発行された cont 命令や newins 命令などを処理し，それに応じて ACM を書き換える．

また，TAC は実行可能となったスレッドをスレッド実行ユニットに割り当てる処理も行う．そのため，TAC 内部にはキューが設けられており，実行可能になったスレッドはキューに入れられる．レジスタファイルに空きができると，キュー内のスレッドはスレッド実行ユニットに割り当てられ，その空きレジスタファイルを用いてプリロードが開始される．

5 おわりに

本稿では，Fuce プロセッサの動作原理の中心となる継続とスレッドについて述べ，それらを使ったプログラミングモデルを解説した．また，このモデルを実現するためのプロセッサアーキテクチャを紹介した．現在，このプロセッサはまだ，実装段階であるため，本稿にプロセッサのパフォーマンスデータ等を示すことはできなかったのは残念である．今後の課題は，まずプロセッサの実装を早期に終わらせることであるが，高速性を追求するためには，キャッシュメモリの効率的な使用法も検討する必要がある．なぜなら，Fuce のスレッドはデータ渡しにメモリを使うため，ここのメモリアクセスがボトルネックになる可能性がある．また，データフローのアプローチでは一般にデータの局所性を活かしにくいという欠点がある．これらを考慮したデータキャッシュの検討はパフォーマンスに大きな影響を与えると考えられる．

なお，本研究は科研費基盤研究 (A)「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」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] Marr, D. T. , Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A. and Upton, M. ”Hyper-threading technology architecture and microarchitecture: a hyperhtext history,” Intel Technology J. 6,1 2002 (online journal).

[5] Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Keckler, S. W. and Moore, C. R., ”Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture,” Proceedings of the 30th Annual International Symposium on Computer Architecture, pp.422-433, 2003.

[6] Throughput Computing, Sun Microsystems. http://www.sun.com/processors/throughput/index.html

[7] 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．