計算機アーキテクチャ　107－11
（１９９４．　７．　２１）

スレッドレベル並列処理アーキテクチャの検討

井上　俊明　　本村　真人　　鳥居　淳*　　小長谷　明彦*

NEC マイクロエレクトロニクス研究所　*NEC Ｃ＆Ｃ研究所

本報告書では、スレッドレベル並列処理をサポートするプロセッサアーキテクチャの検討結果について報告する。スレッドの起動、切り替え、およびスレッド間の同期処理を命令セットとハードウエアでサポートするアーキテクチャを検討した。C++言語を用いてサイクルレベルのアーキテクチャシミュレータを開発し、検討したアーキテクチャの並列化構成に対して初期的な性能評価を行った。フィボナッチ数列の計算プログラムを用いて評価した結果、負荷分散を行わない場合でも、５台構成で単体プロセッサの4.02倍の性能向上が得られた。動的な負荷分散を行い、同期ミスなどによるアイドル時間を削減することによって、さらに台数効果を改善できる見通しを得た。

A Study on Thread-Level Parallel Architecture

Toshiaki INOUE, Masato MOTOMURA, Sunao TORII*, and Akihiko KONAGAYA*

NEC Microelectronics Res. Labs.　　*NEC C&C Res. Labs.

This paper describes a study on a thread-level parallel processor architecture which supports thread switching and thread synchronization in the instruction set level. A cycle-level simulator has been developed using C++ to evaluate the performance of various parallel architectures. A five-processor system is 4.02 times faster than a single processor without load balancing in the case of Fibonacci function by exploiting recursive parallelism. The performance can be improved by reducing idle times caused by synchronization faults etc., if dynamic load balancing using software and/or hardware methods are provided.

1　はじめに

近年、マイクロプロセッサはスーパースカラ化やVLIW化、あるいはスーパーパイプライン化などのアーキテクチャ技術によって大きな性能向上を果たしてきた。さらにこれら命令レベルの並列化に加え、スレッドレベルの並列処理を実現するプロセッサアーキテクチャの研究も盛んに行われている[1][2]。スレッドレベル並列処理アーキテクチャとは、比較的細粒度のスレッドの起動、実行スレッドの切り替え、およびスレッド間の同期処理などを、アーキテクチャレベル（命令セットおよびハードウエア）でサポートし、これらを高速化するものである。プログラムに内在するスレッドレベルの並列性を有効に利用できることや、レイテンシを隠蔽できるなどの点から、プロセッサの台数に応じてスケーラブルな性能向上を実現する超並列計算機システムの要素プロセッサのアーキテクチャとして注目されている。

スレッドレベル並列処理アーキテクチャは、数値計算用途主体の現在の超並列計算機の要素プロセッサとしての位置づけのみでなく、様々な応用形態をとるであろう次世代のマイクロプロセッサの鍵となる技術であると考えられる。これは、①命令レベルの並列処理技術と直交する次のプロセッサ高性能化技術である、②従来のデータ処理中心から割り込み処理などイベント駆動的な処理中心への利用形態の変化に対応しやすい、などの理由による。スレッドレベル並列処理アーキテクチャを、既存の命令レベル並列処理技術と組み合わせることで、相乗効果でプロセッサの性能が向上することを期待できる。また割り込み処理をスレッドとして捕らえ、アーキテクチャレベルでこれをサポートすることにより、従来のRISCアーキテクチャの低い割り込み処理性能の改善が期待できる[3]。

我々は様々なスレッドレベル並列処理アーキテクチャを検討している。これらのアーキテクチャの性能を定量的かつ高精度で比較評価するために、C++言語を用いてサイクルレベルのアーキテクチャシミュレータを開発した。簡単なスレッドレベル並列処理をサポートするアーキテクチャを検討し、並列プロセッサ構成による台数効果などの初期的な性能評価を行ったので、以下にその詳細を報告する。

以下、２章ではスレッドレベル並列処理を命令セットおよびハードウエアでサポートするプロセッサアーキテクチャの一検討を示し、そのプログラミングモデルと命令セット、およびハードウエアについて詳細に述べる。３章ではアーキテクチャシミュレータについて述べる。４章では検討したプロセッサの並列化構成と、シミュレータによる評価結果について報告する。５章では以上をまとめ、スレッド処理のアーキテクチャサポートに関する今後の展望について述べる。

2　スレッドレベル並列処理のサポート

スレッド処理に関連する機能を命令セットで定義し、アーキテクチャでサポートすることによって、これらを全てソフトウエアと割り込み機構で実現する現在のマイクロプロセッサと比較して、より高速な並列処理を期待することができる。スレッドレベル並列処理のアーキテクチャサポートには様々なレベルが考えられる。例えば文献[1]では、スレッドのキューイングや同期のためのハードウエアを持つなどの手厚いサポートを行っている。これに対して、我々はここでは、簡単なサポートのみを考慮したアーキテクチャを検討した。これは開発したアーキテクチャシミュレータ上での初期的なアーキテクチャ評価という位置づけに加え、①汎用マイクロプロセッサという枠組みで考える限り、複雑なハードウエアサポートによりプロセッサ状態を増加させることは避けたい、②スレッドレベルの制御をソフトウエア側に多く持たせることにより、スレッドレベルの並列処理を効率化できる可能性がある、などを考慮しての選択である。以下にアーキテクチャが対象とするプログラミングモデル、命令セット、およびハードウエアについて詳細に述べる。

2．1　プログラミングモデル

図１に２並列のプロセッサの場合の並列プログラミングモデルを示す。プロセッサ１があるスレッドを実行中にプロセッサ２に対して別のスレッドを起動する場合、スレッド起動命令を実行してメッセージ（Forkメッセージ）を作成し、プロセッサ２へ送信する。一方、プロセッサ２はスレッド切り替え命令を実行し、実行すべきスレッドが無ければ、パイプラインをストールしてアイドル状態に入る。アイドル状態でForkメッセージ

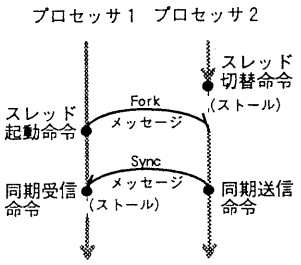

図1　並列プログラミングモデル

を受信すると、メッセージに含まれるスレッドの先頭命令のアドレスに制御を移してスレッドの実行を開始する。また、スレッド間で同期をとる場合、プロセッサ2は同期送信命令を実行してメッセージ（Syncメッセージ）を作成し、プロセッサ1へ送信する。プロセッサ1は同期受信命令を実行した時点でSyncメッセージが到着していれば、スレッドの実行を続けるが、到着していなければ、パイプラインをストールしてアイドル状態に入る。アイドル状態でSyncメッセージを受信すると、再びスレッドの実行を開始する。

Forkメッセージは、＜プロセッサ番号，スレッド先頭命令アドレス，データ＞からなるデータ列で構成される。また同期メッセージは同様に＜プロセッサ番号，同期受信命令アドレス，データ＞で構成される。データはユーザが自由にセマンティクスを定めて使用する。

この並列プログラミングモデルは、以下ような制限によりハードウエアを簡単化している。

(1)スレッドを実行するプロセッサをユーザ指定

Forkメッセージの送信先のプロセッサを、ユーザが明示的に指定するので、ハードウエアスコアボードなどによるプロセッサの資源管理が不要である。このため、ユーザはForkメッセージを受信したプロセッサが、スレッドを実行可能であることを保証する必要がある。

(2)スレッドの切り替えタイミングをユーザ指定

スレッドの切り替えは、ユーザがプログラムの中で明示的に記述したスレッド切り替え命令の位置のみで発生する。従ってプロセッサの空きに応じてスレッドの切り替えを行うための、ハードウエアキューなどは不要である。

このモデルに従うプロセッサのアーキテクチャは、従来のRISCアーキテクチャに、メッセージの作成および送受信と、同期メッセージのバッファリングを行うハードウエアを追加することで容易に実現でき、スレッドレベル並列処理に関して最低限のハードウエアサポートを行ったものになっている。

表1　本プロセッサの命令セット

ロード／ストア	LW, SW
算術演算(即値)	ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI, LUI
算術演算(3ｵﾍﾟﾗﾝﾄﾞ)	ADD, ADDU, SUB, SUBU, SLT, SLTU, AND, OR, XOR, NOR
シフト	SLL, SRL, SRA, SLLV, SRLV, SRAV
ジャンプ／ブランチ	J, JAL, JR, JALR, BEQ, BNE, BLEZ, BGTZ, BLTZ, BGEZ, BLTZAL, BGEZAL
スレッド制御命令	FL, FLR, STOP, SUSP
ﾚｼﾞｽﾀﾌﾚｰﾑ管理命令	FRL, FRS, FPA, FPR
同期命令	SSM, SSMR, RSM
特殊レジスタ転送命令	MVFREG, MVTREG

以上述べたように、このモデルではプロセッサのハードウエアは従来のRISCプロセッサ程度に簡単になる反面、同期待ちや実行すべきスレッドが無い場合に生じるアイドル時間によってプロセッサの利用率が低下することが考えられる。これを改善するためには、ユーザが意識してスレッドスケジューリングを行う必要があり、ソフトウエアに負担をかけたモデルであると考えられる。一方、動的にスレッドを切り替える機能をハードウエアとして組み込むことによって、ソフトウエアに負担をかけずに、より緻密なスレッドスケジューリングが可能であるが、スレッドキューや切り替えロジック、実行すべきスレッドの優先度を決定するハードウエアの実装など、ハードウエアに負担をかけることになる。

2．2　命令セット

表1に本プロセッサの命令セットを示す。MIPSのR2000/3000の命令セット[4]の一部（ロード／ストア、算術演算、シフト、およびジャンプ／ブランチ）に、新たにスレッド制御命令、レジスタフレーム管理命令、同期命令、特殊レジスタ転送命令を追加して構成される。これらの追加命令群の詳細を以下に示す。

(1)スレッド制御命令

プログラムカウンタ相対(FL)、またはレジスタオペランド(FLR)で指定されるアドレスをスレッドの先頭アドレスとして、指定されたプロセッサへForkメッセージを作成・送信し、スレッドを起動する。実行を完了したスレッドのコンテキストを破棄(STOP)または保存(SUSP)してスレッド切り替えを行う。

(2)レジスタフレーム管理命令

レジスタファイルとキャッシュの間で、8ワード（フレーム）単位のロード(FRL)またはストア(FRS)を行う。データキャッシュにヒットすれば、1サイクルで8ワードのデータが一括転送される[5]。レジスタファイルはフレーム単位でメモリへのポインタを持っており、メモリ空間にマッピングして使用することが可能である。FPAはメモリ空間の未使用領域にフレームをマッピングしてポインタを割り当て、FPRはポインタのリネーミングを行う。フレーム単位の一括転送はスレッド内の局所変数を効率よく扱うために設けた機能である。

(3)同期命令

プログラムカウンタ相対(SSM)、またはレジスタオペランド(SSMR)で指定されるアドレスをターゲットとして、指定されたプロセッサへSyncメッセージを送信する。RSMは同期受信待ちを行い、同期成立時に受信したデータをレジスタファイルに書き込む。

(4)特殊レジスタ転送命令

プロセッサ番号などを格納した特殊レジスタと、レジスタファイル間でデータの転送を行う。ソフトウエアで動的にスレッドスケジューリングを行う際に使用される。

2．3　ハードウエア

図2に本プロセッサのハードウエアとパイプラインステージを示す。命令・データキャッシュ、プログラムカウンタ、レジスタファイル、演算器など、従来のRISCプロセッサに含まれる機能ユニットと、スレッドレベル並列処理をサポートする外部制御ユニット、および同期ユニットで構成される。全体では従来のRISCプロセッサと同様の5段パイプラインステージ（命令フェッチ：IF、デコードおよびオペランドフェッチ：ID、実行およびアドレス計算：EX、メモリアクセス：MEM、およびレジスタ書き込み：WB）で動作する。また

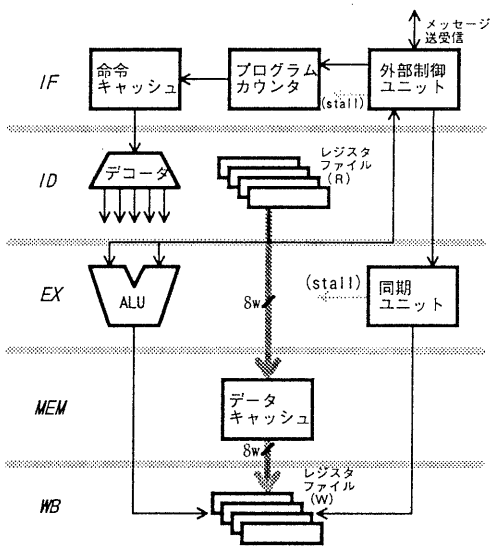

図2　本プロセッサのハードウエア

データキャッシュとレジスタファイルとの間には8ワード幅のデータを一度に転送するパスが設けられ、前節で述べたフレーム管理命令によって、1サイクルでロード／ストアを実行することが可能である。簡単のため、演算ユニットはALUによる単純な整数演算のみを仮定し、浮動小数点命令などの長レイテンシ命令を含まないが、3章で述べるように機能ユニットの追加は容易である。

本プロセッサではスレッドの起動やスレッド間の同期処理をそれぞれ外部制御ユニットと同期ユニットのハードウエアでサポートする。外部制御ユニットは、他のプロセッサへのForkメッセージやSyncメッセージの作成、送信、および受信を行い、IFステージとIDステージで動作する。スレッドの切り替えはIFステージにおいて、1サイクル以内に完了する。スレッド切り替え命令の実行時に切り替えるべきスレッドが存在しない（Forkメッセージが到着していない）場合には、新たなForkメッセージが到着するまでパイプラインはストールする。メッセージの作成および送信はIDステージで行われ、受信は任意のステージでパイプラインの動作とは非同期に行われる。

同期ユニットは内部に複数のエントリからなるバッファを持っている。一つのエントリにはアドレスとデータの対が格納される。外部制御ユ

ニットがSyncメッセージを受信すると、メッセージに含まれるアドレスとデータは同期ユニットに送られる。同期ユニットでは送られてきたアドレスを用いてバッファ内を検索する。アドレスに対応するエントリが見つかったら、同期メッセージの到着をパイプラインに知らせ、データをエントリに登録する。対応するエントリが存在しない場合は、未使用のエントリにアドレスとデータを登録する。プロセッサがRSM命令を実行すると、RSM命令のアドレス（同期ポイント）を用いてEXステージで同期ユニットのバッファを検索する。既に自分の同期ポイントに対するSyncメッセージが到着し、バッファに登録されている場合は、エントリに含まれるデータを読み出して、WBステージでこれをレジスタに書き込む。そうでない場合には同期ミスとしてパイプラインをストールさせ、RSM命令のアドレスをバッファに登録して同期メッセージの到着を待つ。

3　アーキテクチャシミュレータ

はじめに述べたように、我々はスレッドレベルの並列処理をサポートする様々なプロセッサアーキテクチャの検討を行っている。これらのプロセッサの性能を比較評価し、スレッド処理に関するハードウエア／ソフトウエアサポートのトレードオフを見い出す目的で、専用のアーキテクチャシミュレータを開発した。以下に本シミュレータの開発方針、構造、および動作を詳細に述べる。

3．1　開発方針

プロセッサの命令セットやハードウエア構成などのアーキテクチャ、またはキャッシュ容量やメモリアクセスレイテンシなどのシミュレーションパラメータを変えながら、並列プロセッサの性能を効率よく評価するためには、①アーキテクチャの拡張性、②変更の容易性、③シミュレーションの高速性、
などを考慮してシミュレータを開発する必要がある。現在、プロセッサの動作をサイクルレベルの精度でシミュレーションする専用のパイプラインシミュレータとして、DLXやSPIMが知られている[4]。これらはC言語で記述され、ターゲットとするアーキテクチャに特化して設計されており、高速である半面拡張性や汎用性に欠ける。そこで

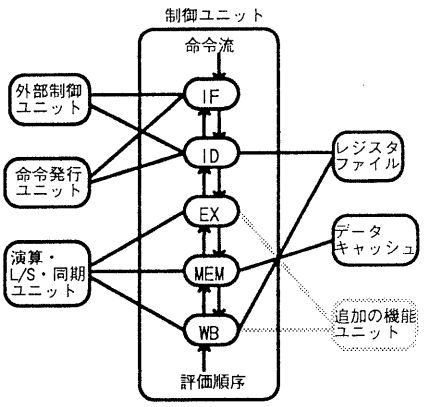

図3　本シミュレータのクラス構造

我々はC++言語とそのクラス関数記述を利用することで、上記の要求を満たすように配慮してシミュレータを開発した。

3．2　構造と動作

図3に、図2に対応するシミュレータの構造（クラス構造）を示す。外部制御ユニット、命令発行ユニット、演算ユニット、ロード／ストア(L/S)ユニット、同期ユニット、制御ユニット、レジスタファイル、データキャッシュ、および追加の機能ユニットで構成され、それぞれC++言語のクラス関数として記述される。各機能ユニットの動作を以下に述べる。

(1)命令発行ユニット

IFステージで命令キャッシュから命令をフェッチし、IDステージでデコードを行う。また演算命令の時、レジスタファイルからオペランドをフェッチする。条件分岐および無条件分岐命令でのターゲットアドレス計算、および条件分岐命令での条件比較もIDステージで行う。

(2)演算ユニット

命令ユニットから送られてきた命令とオペランドに従って、EXステージで演算を行い、WBステージで演算結果をレジスタファイルに書き込む。

(3)ロード／ストアユニット

ロード命令またはストア命令の時、命令ユニットから送られてきたオペランドに従って、EXステージでアドレス計算を行い、MEMステージでデータキャッシュをアクセスする。ロード命令の場合は、キャッシュから読み出したデータをWBス

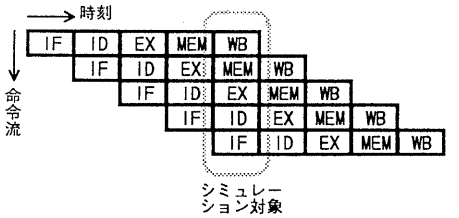

図4　本シミュレータのパイプライン処理

テージでレジスタファイルに書き込む。

(4)制御ユニット

パイプライン内の命令の流れを制御する。命令列がIFステージからWBステージに向かって流れるRISCのパイプライン処理をシミュレーションするために、WBステージから逆順に命令を評価していき、全てのステージにある命令の評価が完了するごとに、各ステージにある命令を次のステージへ移す。同時にハザードやキャッシュミスなどに起因するパイプラインストールの制御も行う。

本シミュレータの外部制御ユニットと同期ユニットの動作は、2.3節で述べたハードウエアの動作に相当する。シミュレータによるパイプラインステージの制御を簡単にするため、機能ユニットだけでなく、各パイプラインステージもまたクラス関数で記述されている。そして各機能ユニットはそれぞれ独立にパイプラインステージをオブジェクト（インスタンス）として所有している。一方、制御ユニットは、機能ユニットのパイプラインステージを参照し、あるパイプラインステージ内の情報（命令等）を別のパイプラインステージに移すことで、命令のパイプライン処理を制御している。

図4に本シミュレータ全体の動作を示す。ある時刻における全てのパイプラインステージを、制御ユニットが評価することで、1サイクルのシミュレーションが完了する。従ってサイクルレベルの精度でシミュレーション結果を得ることができる[4]。

以上のように、ハードウエアに対応した機能ユニットをクラス関数として記述することで、例えば浮動小数点ユニットなどを追加の機能ユニットとして加えたり、キャッシュの大きさを変更するなどの、アーキテクチャやシミュレーションパラメータの変更を容易にシミュレータに反映することができる。

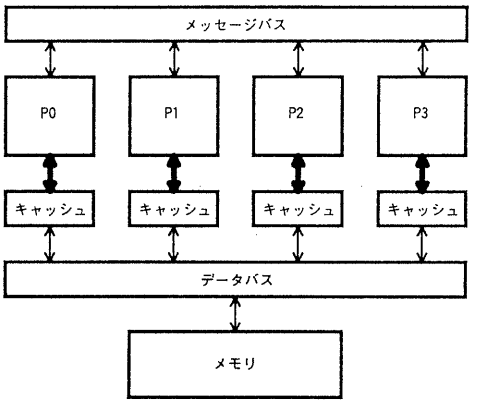

図5　並列プロセッサの構成

4　並列プロセッサ

2章で述べたプロセッサを用いて共有メモリ共有バス結合の並列プロセッサを構成し、台数効果など初期的な性能評価を行った。以下に並列プロセッサの構成と評価結果を示す。

4．1　並列プロセッサの構成

図5に4並列プロセッサの構成を示す。各プロセッサ(P0、P1、P3、およびP4)とそれぞれにつながるデータキャッシュ、メッセージバス、データバス、およびメモリで構成される。プロセッサとデータキャッシュの部分は図2のアーキテクチャに相当する。シミュレータの簡単のため、並列化の構成に関して以下の仮定を行った。

(1)共有バス共有メモリ構成
(2)SPMD(Single Program Multiple Data Stream)モデル
(3)ライトスルーによるコンシステンシ管理
(4)Invalidateプロトコルによるキャッシュヒーレンシ制御

この構成において、バスを使用するプロセッサに優先順位を定義してバスアクセスの衝突を調停することを仮定する。優先するプロセッサはラウンドロビン方式で切り替わる。またデータキャッシュをプロセッサごとに持たせることで、局所データをアクセスする際に広いバンド幅を提供している。

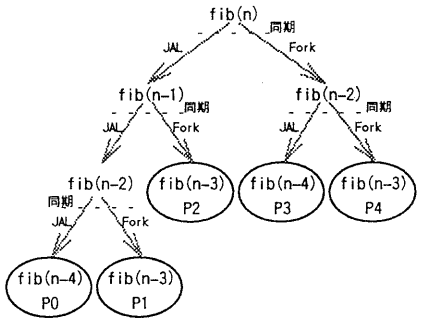

図6　フィボナッチ関数の並列化の例（5並列）

4．2　性能評価

フィボナッチ数列(fib)の計算を例として、前節で述べた並列プロセッサの初期的な性能評価を行った。本シミュレータで仮定したパラメータは以下のとおりである。

(1)命令キャッシュ：ヒット率100%
(2)データキャッシュ：8w×256ライン、ダイレクトマップ方式
(3)キャッシュミス遅延：1サイクル
(4)分岐遅延：1サイクル
(5)バスアクセス：1サイクル

フィボナッチ数列の計算は、データ並列ではないが、再起呼び出しを容易に並列化することが可能である。図6に5並列プロセッサ構成のために展開したfib(n)のツリー構成と、各プロセッサへのスレッドの割り当てを示す。JAL命令（逐次実行）とFork命令（並列実行）を繰り返し、ツリーの末端において各プロセッサ（P0、P1、P2、P3、P4、およびP5）がそれぞれの引き数でのfibの値を逐次的に計算する。Fork先のプロセッサからFork元のプロセッサへ計算結果を渡す際には同期を必要とする。このように、静的にプロセッサを割り当てる。

図7にfib(10)を実行した場合の台数効果の評価結果を示す。ほぼ線形に性能向上が得られている。4並列の結果はP3がP1からForkされたfib(n-4)を実行しており、他のプロセッサよりも早く終了するので、実行スレッド無しによるプロセッサのアイドル時間が大きく、負荷分散が適当でないことを示している。ハザードは従来の命令スケジューリングで削減することが可能である。またキャッシュミスもキャッシュサイズの増加または連想数の増加で削減可能である。従ってこれらの削減により、同期ミスやスレッド無しによるアイドル時間は相対的に増加するので、台数効果の改善のためには動的な負荷分散はより重要になる。

5　まとめと今後の展望

スレッドレベルの並列処理を実行するプロセッサのプログラミングモデル、命令セット、およびハードウエアを検討した。スレッドの起動、切り替え、およびスレッド間の同期処理を高速化するために、従来のRISCプロセッサのアーキテクチャに、メッセージ作成および送受信を行う機能と同期の待ち合わせを行う機能を新たなハードウエアとして追加し、これに関連する命令セットを拡張した。更に、C++言語を用いてサイクルレベルのアーキテクチャシミュレータを開発した。ハードウエア資源やパイプラインステージに対応したクラス関数を定義することで、様々なアーキテクチャを効率よく評価できるように工夫した。

また本プロセッサを用いて、共有バス共有メモリ結合による並列プロセッサを構成し、フィボナッチ数列の計算を例として、台数効果などの初期的な性能評価を行った。その結果、処理性能改善のためには、①ハザードやキャッシュミスのさらなる削減、②同期ミスや実行すべきスレッドが無いために生じるプロセッサのアイドル時間の削減、が必要不可欠であるという知見を得た。特に後者に関しては、ソフトウエアまたはハードウエアのサポートによる動的なスレッドの切り替え、適切な負荷分散によるオーバーヘッドの隠蔽が必要である。今後は上記の観点からスレッド処理をサポートする様々なアーキテクチャを、本シミュレータを用いて詳細に検討、評価する予定である。

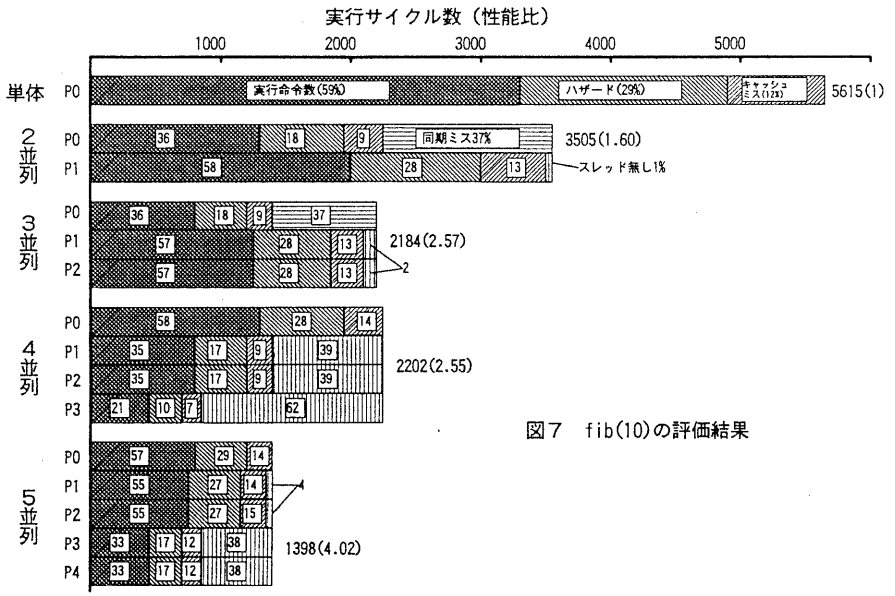

図7　fib(10)の評価結果

謝辞

本研究を遂行するにあたって有益な助言をして頂いた関係者各位に深く感謝します。またシミュレータを開発して頂いた日本電気情報システム㈱鈴木氏、加藤氏に深く感謝します。

参考文献

[1]坂井他，「超並列計算機RWC-1の基本構想」，並列処理シンポジウムJSPP'93論文集，pp.87-94，1993年5月.

[2]児玉他，「高並列計算機EM-Xのアーキテクチャ」，情報処理学会研究報告，93-ARC-101，pp.49-56，1993年８月.

[3]Thomas E. Anderson et. al, "The Interaction of Architecture and Operating System Design", Proceedings of Fourth International Conference on Architectual Support for Programming Languages and Operating Sysyems, pp.108-120, 1991.

[4]David A. Patterson and John L. Hennessy, "Computer Organization & Design The Hardware/Software Interface", Morgan Kaufmann Publishers, Inc., 1994.

[5]M.Motomura et. al., "Cache-Processor Coupling: A Fast & Wide On-Chip Data Cache Design", 1994 Symposium on VLSI Circuits.