Simultaneous Multithread（SMT）アーキテクチャの実現方式

河原章二[†] Mark Yankelevsky[††]
中條拓伯[†] Constantine Polychronopoulos[††]

我々はSMTアーキテクチャをベースにしたプロセッサ α-Coralの実現を試み，効率の良いスレッド実行環境の構築を目指す．α-Coralでは，コンパイラによりプログラム内の並列実行可能なスレッドを抽出し，それらの同時実行や高速切替えなどのサポートをハードウェアにより行う．α-Coralはコンパイラの生成するさまざまな粒度のスレッドを効率良く実行するためのスレッド制御命令を有し，一部のレジスタを共有することにより，少ない遅延でのスレッド間通信を実現している．本論文ではα-Coralのアーキテクチャについて述べ，その性能をシミュレーションにより評価した結果，シングルスレッドのプログラムと比べ，同一環境下でもプログラムにマルチスレッド化を施した場合は速度向上がみられることが確認できた．

A Study on an Implementation of a Simultaneous Multithreaded Architecture

Shoji Kawahara,[†] Mark Yankelevsky,[††] Hironori Nakajo[†] and Constantine Polychronopoulos[††]

We try to implement an SMT architecture based processor α-Coral which supports efficient execution of threads in a single chip processor. Multiple threads which are parallelized and optimized by a compiler are executed in parallel and fast switched by hardware. By using dedicated instructions for thread control, α-Coral handles multi-grained threads, and enables them to communicate with each other efficiently via shared registers. In this paper, the architecture of α-Coral is described and performance evaluation by simulation is shown. From the results of simulation, speed up of execution with multithreaded programs is confirmed compared with a single-threaded one in the same environment. Currently we have been designing α-Coral by Hardware Description Language (HDL) with refining and improving of functions which are implemented in the α-Coral simulator.

1. はじめに

ここ数年，半導体技術は急速な進歩をとげ，1チップに数千万の単位でトランジスタを集積することが可能になった．その結果，1つのマイクロプロセッサ内に，複数の演算ユニットや高度な分岐予測機構を実装し性能向上が図られることとなった．スーパスカラプロセッサがその顕著な例であり，大量の演算ユニットや巨大な命令ウィンドウを投入することにより，その性能を向上させてきた．

しかしながら，現在スーパスカラアーキテクチャは，ハードウェアを投入するコストに見合った性能改善がみられなくなってきている．その理由として，単一スレッド（命令流）から命令レベル並列性（ILP）のみを利用して実行速度を上げるスーパスカラアーキテクチャは，現在実行している命令流内のある一部分において，前後の命令間に依存関係があったとき，その近傍では逐次的に命令を実行しなければならず，全体の性能の改善が得られない場合があるからである．命令ウィンドウ（リザベーションステーション）を増強することによりプログラムに対する先見性が増し，この問題を解決をすることができるが，命令ウィンドウの構造上，そのサイズをむやみに大きくすることは現実的ではない[1)]．

そこで，旧来のスーパスカラアーキテクチャにとって代わるアーキテクチャとして，オンチップマルチスレッドアーキテクチャが有望視されている．オンチップマルチスレッドアーキテクチャとは，1チップ上に

† 東京農工大学工学部情報コミュニケーション工学科
Department of Computer, Information and Communication Sciences, Tokyo University of Agriculture and Technology
†† Center for Supercomputing Research and Development, University of Illinois at Urbana Champaign

複数のスレッドを内包させて実行するものであり，その中で Simultaneous Multithreading（SMT）アーキテクチャは，複数のスレッドでレジスタやパイプラインなどのチップ内のハードウェア資源の多くを共有することにより，効率の良いスレッド実行をサポートするものである．スーパスカラアーキテクチャは単体スレッドしか扱わないのに対して，SMT アーキテクチャは従来の ILP に加えてスレッドレベル並列性（TLP）をも利用することにより，プログラムの実行速度を向上させるオンチップマルチスレッドアーキテクチャである．

図 1 にプログラムの実行速度向上を目指した高性能プロセッサアーキテクチャの概念を示す．

図では，横軸がプロセッサの同時命令発行数，縦軸が実行サイクルを示し，上から下へと実行サイクルが進む．図 1 (a) は従来のスーパスカラプロセッサ（または VLIW プロセッサ）の命令発行の様子を示している．スーパスカラプロセッサは前述のとおり，全サイクルを通して実行するスレッドは 1 つである．何らかの要因（たとえばキャッシュミス）で当該スレッドがストールすると，命令をまったく発行できないサイクルが生じてしまう．このサイクル数の冗長を，Vertical Waste（VW）と呼ぶ．命令を実行できたサイクルにおいても，ILP が限られていることにより 1 命令や 2 命令程度しか命令を実行できてない場合もある．この命令発行の無駄を，Horizontal Waste（HW）という．

(b) は Tera の MTA [5] や MIT の Alewife [6] などに代表されるマルチスレッドアーキテクチャでのプログラムの実行の様子を示している．これらのアーキテクチャは，複数のスレッドを切り替え，各サイクルにおいてそれぞれのスレッドの命令を実行する．これにより VW は解決しているが，依然として HW の問題が残っている．これは，文献 5), 6) のアーキテクチャは実行サイクルだけに注目しており，1 サイクル中の命令発行数はスーパスカラと同様に ILP にのみ頼っているからである．

最後に，図中の (c) は SMT アーキテクチャにおけるプログラムの実行の様子を示している．注目すべき点は，1 サイクル中に複数のスレッドの命令を同時に実行していることである．これにより，VW のみならず HW をも解決することが可能である．

我々はこの SMT アーキテクチャをベースにしたプロセッサ α-Coral [2),3)] の実現を試み，コンパイラのサポートにより効率の良いスレッド実行環境の構築を目指す．α-Coral はイリノイ大学スーパコンピューティング研究開発センタ（CSRD）で提案されているオンチップマルチスレッドプロセッサアーキテクチャであり，コンパイラとしては，同じく CSRD で開発されている PROMIS [4] を用いる．α-Coral は柔軟なスレッド制御命令，レジスタのセグメンテーションを実装することにより，ハードウェアリソースをより効率良く使用しつつ，プログラムの実行速度の向上を図る．

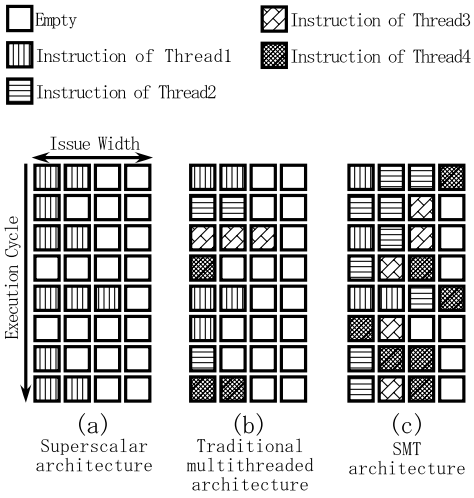

図 1 高性能プロセッサアーキテクチャにおける命令発行の概要
Fig. 1 Instruction issues in high performance processor architectures.

本論文において，2 章では，関連研究をあげ，3 章で α-Coral のアーキテクチャを具体的に説明する．4 章において，α-Coral で用いるコンパイラ PROMIS について触れた後，5 章でシミュレーション結果とそれらに対する考察を行う．そして，6 章でハードウェアコストの見積りを示し，最後に 7 章で本論文のまとめを行う．

2. 関 連 研 究

SMT とは異なるオンチップマルチスレッドアーキテクチャとして，オンチップマルチプロセッサ（OCMP）があり，これはすでにいくつかの提案がなされている [7)~11)]．

SMT と OCMP の相違点は，制御を集中的に行うか，分散して行うかという点にあり，基本的な SMT アーキテクチャは，単一のスーパスカラパイプラインに複数スレッドの命令を流し，実行ユニットも複数スレッドで共有する点から，それらを集中的に制御する機構が必要となる．それに対し OCMP は，制御を同一チップ内の各要素プロセッサ（PE）に分散させ，1 つの PE 上にそれぞれレジスタファイル，命令ウィンドウ，実行ユニットを実装する．そして 1 つのスレッドを 1 つの PE に割り当てることから，基本的には個々

のPEにのみ制御機構を実現すればよい．しかしながら，従来のマルチプロセッサアーキテクチャの課題である，各プロセッサに対する負荷分散の問題がOCMPには残り，これがうまく解決できない場合，OCMPはそのハードウェアコストに見合った性能が出せない．なぜならば，実行ユニットなどは各PEにそれぞれ割り振られており，そのPEに割り当てられたスレッド以外はそのハードウェアリソースを利用できないためである．

一方SMTの場合，すべてのスレッドにおいてハードウェアリソースを共有するので，多数のスレッドが生成できない場合においても，実行ユニットなどの無駄はOCMPほど生じない．しかし，SMTはその集中的な制御のため，ハードウェアが複雑化するという問題点がある．

SMTアーキテクチャの提案としては，TullsenらのSMT Project[12)]があげられる．元々SMTアーキテクチャは複数のプログラムを同時に実行するアーキテクチャとして提案されていた．現在，擬似的にマルチプロセッサとして稼動し，複数のタスクやプログラムを同時に実行するSMTアーキテクチャとしてIntel社のHyperThreading[14)]が，一般的なPCの市場で商用化されつつある．この手法においては，チップに投入するハードウェア量を増やし，それとともにプロセッサ内で稼動させるスレッド（タスク，プログラム）を増やすことで比較的容易に性能利得を得ることが可能である．一方，SMTプロセッサで単一プログラムを静的，もしくは動的に並列化し実行する手法はいまだ実用化には至っていない．

Loらはシングルプログラムを SMTプロセッサで動作させたときのシミュレーション結果を報告している[13)]．その結果，5つのSPLASH-2ベンチマークプログラムを用いた場合，SMTプロセッサのInstructions Per Cycle（ILP）は平均5.91，最大6.83となっている．

また別のSMTアーキテクチャの手法として，Wallaceらは分岐時に新たなスレッドを生成し，複数のパスを同時に実行する手法を提案している[15)]．このアーキテクチャでは，分岐で採用されたほうのスレッドの実行を進め，採用されなかったほうのスレッドは捨てられる．

そして，Chappellらや佐藤らは，主スレッドに対してサブスレッドが最適化を行うことにより，主スレッドの実行速度を改善する手法を提案している[16),23)]．このサブスレッドは，分岐予測改善，プリフェッチ，キャッシュの操作などを行うことにより，主スレッドの実行をサポートする．通常のシングルスレッドプロセッサの場合，サブスレッドを呼び出すときには主スレッドを一時停止させる必要がある．そのため，主スレッドを一時停止してもその分の利得を得られるように，サブスレッドを呼び出すための閾値を十分高く設定しなければならない．結果として，シングルスレッドプロセッサでは，ごく限られた状況でしかサブスレッドを呼び出すことはできない．しかしSMTアーキテクチャは，主スレッドとサブスレッドを同時に実行できるので，サブスレッドを呼び出すタイミングである閾値を低く設定することができる．

SMTアーキテクチャは上でも述べたとおり，ハードウェアの複雑化が問題点としてあげられている．Hilyらは十分にスレッドが生成できるのであれば，Out Of Order（OOO）実行を行う必要がないと主張している[17)]．また，同じくHilyらは文献18)において，主にL2-CacheのSMTプロセッサにおける性能を綿密に調査している．ここでSMTプロセッサの性能を考慮するとき，L2-Cacheの連想数，ブロックサイズなどの決定を注意深く行う必要があると結んでいる．SMTアーキテクチャの実現可能性を考慮し，ハードウェアの複雑さの緩和も視野に入れた場合にはIn Order実行の機構の選択，検討も行う必要がある．

3. α-Coral アーキテクチャ

3.1 α-Coral におけるスレッド

ここではα-Coralにおけるスレッドの説明をする．スレッドは，ユーザ（コンパイラ）がスレッド生成命令を用いて明示的に生成する．そして各スレッドにはプログラムカウンタ（PC），ローカルレジスタセグメントが割り当てられる．また，スレッドには状態レジスタが割り当てられる．以下に，スレッドの実行状態を示す．

`free` ：実行可能な状態．

`miss stall` ：フェッチサイクルでの命令キャッシュミスによるストール状態．

`flow interruption stall` ：スレッド制御命令をフェッチしてきたときに，その制御命令の実行終了までスレッドが停止している状態．

`blocked` ：`BLOCK`命令による一時停止状態．

`failed to allocate resource` ：リソース確保失敗によるストール状態．

3.2 α-Coral の基本ユニット

図2にα-Coralの概要を示す．

● PC Queue（PCQ）

生成されたスレッドには，このPCQというPCの

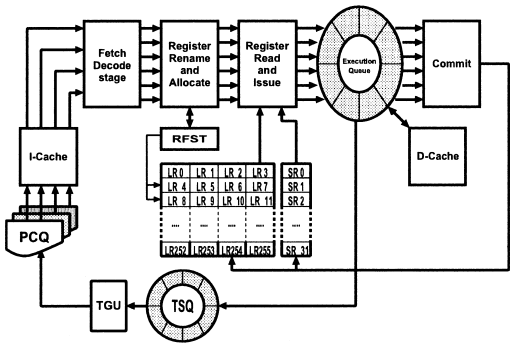

図 2 α-Coral の概観図
Fig. 2 Overview of α-Coral.

待ち行列の最後尾にある PC が割り当てられる．PC 数はすなわちプロセッサに内包するスレッド数を意味し，その数・制御方法は，後述するレジスタファイルの割当ての問題も含めて注意深く検討しなければならない．

●レジスタファイル

Tullsen らの提案[12]では各スレッドに割り当てるリソースはすべて均一であり，たとえばレジスタファイルのサイズはすべてのスレッドに対し 32 個で固定されている．この場合，8 個のスレッドをプロセッサ内に内包するのにレジスタファイルのサイズは 256 個になり，これ以上のスレッドを保持しようとすればさらにレジスタファイルは巨大化する．我々は，各スレッドに必ずしも 32 個ものレジスタが毎回必要であるとは限らないことから，1 つのスレッド内で使用するレジスタの数を抑えることで，レジスタファイルのサイズを縮小し，またチップ内に取り込み可能なスレッドの最大数を増やすことが可能であると考える．この機能を実現するため，我々は α-Coral にレジスタセグメントという，1 つのレジスタファイルをいくつかのセグメントに分割し，複数のスレッドがそれを使用する機構を取り入れた．

α-Coral はプロセッサ内部に，各スレッド固有に割り当てるための整数型・浮動小数点型ローカルレジスタ（以後それぞれ ILR，FLR と表記），そしてすべてのスレッドが参照・使用できる整数型共有レジスタ（SR）が存在する．α-Coral ではスレッドが生成された時点では，レジスタは割り当てられず，レジスタが割り当てられるタイミングは，そのスレッド内で初めて LR を参照・使用する命令が現れたときである．割り当てる LR 数はその命令のオペランドから算出する．具体的には，今新しいスレッドが生成され，そのスレッドから以下の命令をフェッチしてきたとする．

```
ADD LR15 LR0 LR1
```

このとき初めて α-Coral はレジスタの確保を開始する．確保すべき LR の数は，書き込み LR の添え字 +1 である．上記の例では，書き込み先の LR の添え字は 15 なので，α-Coral はレジスタファイルの中から 16 個のレジスタを，当該スレッドへのレジスタセグメントとして確保しようとする．ここでもし確保に失敗した場合は，α-Coral は当該スレッドをストールさせる．また，パイプライン内に存在する当該スレッドの命令はすべてフラッシュされる．そして PC の状態を `failed to allocate resource` とし，以後プロセッサ内で稼動しているスレッドが停止し，リソースを解放（deallocate）したときに確保に失敗した当該スレッドに対して再確保しようとする．レジスタ確保に成功すれば，レジスタファイルの中で割り当てたセグメントの開始番号と終了番号を，後述する Register File Segment Table（RFST）に書き込む．一方，SR は主にスレッド間の同期・通信に用いられる．以上のように，確保するレジスタ数はコンパイラがプログラムに合わせて決定し，もし確保した数以上に必要になったとしても α-Coral はレジスタの再割当てなどは行わない．

● Register File Segment Table（RFST）

各スレッドのローカルレジスタの割当て情報を保持しているテーブルマップである．

● Execution Queue（EQ）

EQ は基本的に，スーパスカラプロセッサの命令ウィンドウと同じ働きをするものである．しかし，今回提案している α-Coral は現在一般的なスーパスカラプロセッサに用いられているリネーミングロジックを実装していないため，レジスタの名前替えは行われない．よって，EQ でのソースレジスタの待合せは，物理レジスタ番号をそのまま使ったものになる．これが一般的な命令ウィンドウと異なる点である．また SMT アーキテクチャを実現するために，投機実行ミスによる各スレッドごとのフラッシュ機能を EQ に実装している．

● Thread Starters Queue（TSQ）

スレッド生成命令が一時格納される Queue で，すべてのスレッド生成命令はいったんここに格納される．格納された命令は，後述する Thread Generating Unit（TGU）により実行されるまで TSQ に待機する．この TSQ は，スレッド生成命令専用の命令ウィンドウといえる．

● Thread Generating Unit（TGU）

このユニットは TSQ に登録されているスレッド生成命令を取り出し，スレッドに PC を割り当てる．こ

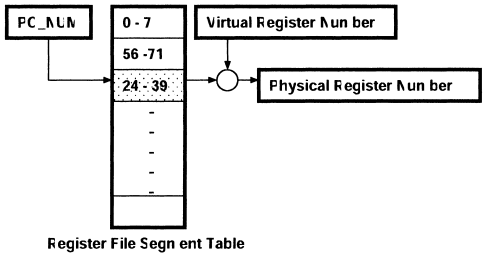

図 3 RFST を用いたレジスタリネーミング
Fig. 3 Register renaming with RFST.

のとき，PCQ 内のすべての PC がスレッドに割り当てられていた場合，もしくはローカルレジスタファイルが不足している場合，スレッド生成は一時的に止められる．そして，今まで稼動していたスレッドの停止が起こったときにスレッド生成を再開する．

3.3 スレッド実行方式

基本ユニットの説明においてスレッドの実行方式の流れを述べたが，α-Coral には Register Rename Stage という特殊なパイプラインステージが設けられている．通常のプロセッサのパイプラインの場合は命令フェッチ（IF）→ 命令デコード（ID）→ 命令発行（II）→ ··· と進むが，α-Coral では ID ステージと II ステージの間に，前述したレジスタセグメント方式の制御機構上生じたパイプラインステージが存在する．α-Coral では，フェッチしてきた命令のオペランドで示されるレジスタ番号が，ただちにはレジスタ読み出しに使用できない．フェッチされてきた命令に示されているレジスタ番号は仮想的な番号であり，実際には α-Coral プロセッサ内部で，物理レジスタ番号への変換を行わなければならず，RFST がこの変換の際に用いられる（**図 3**）．この機構はスーパスカラプロセッサの実現方式における，リオーダバッファやリネームバッファなどを用いたレジスタリネーミングではないのに注意されたい．

PC の番号がこのテーブルのポインタとして使われ，目的のオフセット情報を引き出す．変換後の物理番号は次のパイプラインレジスタに渡され，命令発行ステージに移る．

また，Register Rename Statge は LR の確保も請け負っている．もしまだスレッドに LR が確保されていないとき（つまりは新しく生成されたスレッドだった場合），α-Coral はローカルレジスタファイル内から要求された個数だけのレジスタをセグメントとして確保しようとする．ここで確保に失敗した場合は，前述した `failed to allocate resource` ステートを PC の状態レジスタに書き込む．

表 1 スレッド制御命令
Table 1 Instructions of thread control.

命令	形式
TFORK	TFORK <TRGT PC> <DIE>
PTFORK	PTFORK <LR 0> <LR 1> <TRGT PC> <DIE>
CTFORK	CTFORK <SR> <TRGT PC> <DIE>
DOALL	DOALL <LR 0> <LR 1> <TRGT PC> <DOUBLE> <DIE>
ADDS	ADDS <DEST SR> <SRC SR> <SRC LR|SRC C>
SUBS	SUBS <DEST SR> <SRC SR> <SRC LR|SRC C>
BLOCK	BLOCK <SR> <TRGT V>
STORES	STORES <normal store parameters> <DIE>
HALT	HALT

3.4 スレッド制御命令

●命令セット

α-Coral の命令セットは，現在 MIPS RISC アーキテクチャをベースに設計されている．ここでは現時点において，実装を試み，またシミュレータ上で実現しているスレッド制御命令を**表 1** にあげ，個々の命令の説明を示す．

`TFORK` スレッドを生成する命令である．`DIE` という 1 ビットのオペランドをとり，これがアサートされている場合はスレッドを新たに生成してから自スレッドを終了する．

`PTFORK` 新しくスレッドを生成する命令だが，このとき自スレッドの `LR0` および `LR1` レジスタの値を新しく生成するスレッドに渡す．

`CTFORK` `SR` の値が 0 だったときに新しくスレッドを生成する．それ以外の値のときは NOP になる．この命令は，たとえばループから通常のシーケンスへの再開のときなどに使用できる．ループする回数を `SR` に設定し，ループをまわすたびにそれをデクリメントすることにより，最後のイタレーションで，通常シーケンスのスレッドを生成することができる．

`DOALL` まず `DOUBLE` というオペランドに注目する．`DOUBLE` が 0 のとき，DOALL 命令は一重ループ（通常のループ）のイタレーション分解に使われる．具体的には，`LR0` の値の分だけスレッドが生成される．以下に示すようなループがあった場合の使用例を示す．

```
for( i = 0; i < 10000; i++ ){
  ...
}
```

このようなループがあった場合，DOALL 命令の `LR0` に 10000 をセットしておくことにより 10000 個のスレッドを生成することができる．このとき生成されるスレッド群には，上記のループのイタ

レーション番号と DOALL のオペランド LR1 の値が渡される．

DOUBLE ビットが 1 の場合，以下のようなループに適用することができる．

```
for( i = 0; i < 10000; i++ ){
  for( j = 0; j < 20000; j++ ){
    ...
  }
}
```

この場合，DOALL 命令の LR0 に 10000，LR1 に 20000 をセットしておくことにより 10000×20000 個のイタレーションをスレッドとして生成できる．生成されるスレッドには，内外の for ループの各イタレーションの番号が渡される．

ADDS，SUBS これらの命令は SR の演算に用いるものであり，主に下で述べる BLOCK 命令と一緒に同期処理に使われる．SRC LR はソースに LR をとり，SRC C は即値をとることを意味する．

BLOCK BLOCK 命令はオペランド SR と TARGET V が等しくなるまで自スレッドを一時停止する命令であり，スレッドの同期などに用いる．オペランド TARGET V は LR，または即値を指定する．

STORES メモリにデータをストアしたあと，DIE ビットが 1 だった場合は自スレッドを停止する．その他の機能は，通常のストア命令とは変わりはない．

HALT 自スレッドを停止する．

4. PROMIS

ここでは α-Coral を効率良く実行させるための，CSRD で開発されているコンパイラ PROMIS について触れる．

PROMIS は複数の言語をサポートする自動並列化コンパイラである．また PROMIS は，Unified and universal Internal Representation（UIR）という内部表現を用いてフロントエンドとバックエンドを統合することにより，フロントエンドからバックエンドへ詳細な依存情報や，最適化に有用な情報を劣化させることなく伝えることができる．そして PROMIS は対象とするアーキテクチャの情報をもとに，タスクレベル，ループレベル，命令レベルでの並列性を抽出する．コンパイラの核の部分である UIR は，Hierarchical Task Gragh（HTG）[19]を用い，対象とするアーキテクチャに適したフロントエンド，バックエンドの解析および最適化が行われる．そしてまた Symbolic Analysis[20]という記号解析の手法が，プログラムの制御流のより正確な解析をコンパイル時に実現している．

現在 PROMIS は言語 C，C++，FORTRAN，Java バイトコードをサポートしている．また CISC，RISC，DSP などのさまざまなプロセッサを対象にできるように設計されている．

4.1 スレッド生成

フロントエンドでの解析が終了した時点で，バックエンドでは IR を Low-level UIR（LUIR）に変換する．そして PROMIS を α-Coral に適用させた場合は，次にスレッド生成が**スレッド分割**，**レジスタ確保**，**アセンブリコード生成**の 3 つのフェーズに分かれて行われる．以下に簡単に解説する．

4.1.1 スレッド分割

スレッドはブロック内の並列性およびブロック間の並列性をもとに生成される．ブロック内の並列性は主に，ループイタレーション，または独立した複数のステートメントの集合から構成される．他に用いられているスレッドパッケージング手法[21]と同様に，スレッドを生成すべきコードをファンクションコードとして扱う．

まず当該コードをファンクションに置換し，この新たに作成したファンクションのポインタを用いることでファンクションコールに置き換えられる．α-Coral の場合は，このファンクションコールは DOALL や TFORK などの前述したスレッド生成命令に置換される．このアプローチでレジスタの一時退避などのオーバヘッドを回避することができる．また，同期のためのファンクションコールは BLOCK 命令に置き換えられる．そしてループは，スレッド制御命令の説明で示したように，各スレッドがイタレーションに割り当てられるように分割される．

実際の変換としては，まずループはファンクションの中に取り込まれる．そしてこのファンクションコールはコード生成の段階で DOALL 命令に置き換えられる．また，基本ブロックの中のデータフロー独立のものは 2 個以上のスレッドに分割される．この場合，スレッドの 1 つはプライマリスレッドとして扱われ，制御フローの中心に置かれる．たとえば，2 つのデータフロー独立の基本ブロックがあるとすれば，片方をプライマリスレッド，もう一方をサブスレッドとして定義する．両者の処理が進むにつれ，これら 2 つの基本ブロックが結合する地点でプライマリスレッドは，もしサブスレッドの処理が完了していなければ，BLOCK 命令によってウェイトがかけられて同期がとられる．一方でサブスレッドは処理が完了した時点で，プライ

マリスレッドに渡すべき処理結果を共有レジスタ，もしくはメモリにストアした後に終了（HALT 命令）する．その後はプライマリスレッドとして定義されたスレッドは，サブスレッドの処理結果を受け取り処理が再開される．このとき，コード内ではサブスレッドは TFORK ファンクションコールとして置換され，独立したファンクション内に置かれていく．

次にブロック間の並列性について論じる．連続した HTG ブロックがデータ独立のとき，ブロック間並列が生じる．これは基本ブロックの並列性と同様に扱われる．2 つ以上の連続ブロックがそれぞれスレッドとして扱われるとき，そのうちの 1 つはプライマリスレッドとして扱われ，その他のスレッドはファンクションコールとして置換される．以上で取り上げた状況では，バリア同期が必要な場合がある．そのようなときはプライマリスレッドの終端に BLOCK ファンクションコールを置いておくことにより，バリア同期を実装することが可能となる．

4.1.2 レジスタ確保

スレッド分割フェーズでは単にスレッドであることを示すタグを付けるだけである．このタグ付けされたスレッドに対してレジスタの確保数を決定するのがこの第 2 フェーズである．第 2 フェーズは 2 つのステージが存在する．

第 1 ステージ 各スレッド内の仮想レジスタに，スレッド内の変数を割り当てる．このときリダクション変数などはマーキングされ，SR に割り当てるようにする．

第 2 ステージ スレッド内の解析が完了した時点で，実際に必要なレジスタ数をカウントする．また，ループのイタレーションが多い場合は，可能な限りレジスタの割当て数を小さく抑えようとする．また，リダクション変数は SR に割り当てられる．

4.1.3 アセンブリコード生成

このフェーズで PROMIS のステートメントは MIPS R4000 と α-Coral の命令に置換されていく．拡張命令の説明で示したように，SR に対して特別に用意されている命令が存在する．よって，レジスタ確保のフェーズで SR とマーキングされていたレジスタを操作する場合には，専用の命令を生成する．具体的な例として，式 `SR1 = SR1 + 1` の場合は，自動的に `ADDS SR1 SR1 1` に変換される．

5. シミュレーションによる性能評価

今回ベンチマークプログラム実行の評価には CSRD で設計された α-Coral シミュレータ[2),3)]を用いた．プロセッサアーキテクチャとしては，Out-Of-Order（OOO）をサポートしているが，リオーダバッファなどのリネーミングロジックは現在のところサポートしていない．OOO 実行の実現方法としてはむしろスコアボーディング方式に近く，命令間の RAW，WAR，WAW のすべての依存関係がすべて解決されたところで初めて命令は実行ユニットに発行される．また α-Coral シミュレータでは，フェッチスレッドの選択方法をいくつか選択できる．これは毎サイクル，フェッチポートを使用するスレッドをどのように決定するかというものであるが，これに関して今回のシミュレーションでは PCQ の先頭から順に検索し，各スレッドの状態レジスタを参照してフェッチできる状態のスレッドがあればそれに対してフェッチポートの使用権を与えるという方法を選択した．たとえば，PCQ のエントリが 0 番から 31 番までの計 32 エントリだとしたら，毎サイクルエントリ 0 番のスレッドの状態から順に実行状態にあるか調べていく，という方法である．その他の選択方法に関しては文献 2)，3) にシミュレーションデータ，そしてそれらの考察が議論されている．この α-Coral のシミュレータを用いて，同時実行可能なスレッド数やレジスタ数など，設計上の重要なパラメータの調査に利用し，現状におけるα-Coral の性能予測を行っている．シミュレータは PCQ サイズ，各種レジスタファイルサイズ，各スレッドの命令発行数のなど，各種データの数値を設定することが可能である．

5.1 シミュレーションに用いたプロセッサモデル

ここでは，**表 2** に示すプロセッサモデルおよびメモリシステムのもとでシミュレーションを行った．表 2 で示した Low，Medium，High，Super とはプロセッサの規模を示し，いずれも α-Coral のアーキテクチャがベースである．これらのプロセッサモデルの違いは，主に同時命令発行数，それにともなう実行ユニットの数，レジスタファイルのサイズである．メモリシステムは α-Coral のスケーラビリティを評価するために，プロセッサモデルに関係なく一定にしてある．

5.2 評価プログラム

シミュレーションに使用したベンチマークプログラムは行列の乗算，Fibonacci 数列計算，SPECint95 の中から Compress，SPECfp95 の中から Swim，以上の 4 つである．プログラムはそれぞれシングルスレッドのものとマルチスレッドの 2 種類を用意している．現段階では，PROMIS の α-Coral への適用が完了しておらず，各プログラムのマルチスレッド化は手作業で行っている．今回のベンチマークでは，ループで 249

表 2 プロセッサモデルおよびメモリモデル
Table 2 Processor models and a memory model.

	Low	Medium	High	Super	
	Instructions per Cycle or Quantity or Size				Delay/Latency
Fetch/Decode/Issue Width	2	4	6	8	
Commit Width	2	4	6	8	
Units					
- ALU w/shifter	1	1	2	2	1 cycle
- ALU w/comparator	1	2	4	4	1 cycle
- Integer multiplier/divider	1	2	2	3	3 stage, 1 cycle
- Branch Unit (Int/FP)	2/1	3/1	4/1	4/1	1 cycle
- FPU (add/sub)	1	2	4	4	2 stage, 2 cycles
- FPU (multiply)	1	2	2	2	4 stage, 2 cycles
- FPU (divide/sqrt)	1	1	2	2	4 stage, 3 cycles
- Load Unit	1	3	6	6	1 stage, 1 cycle if hit
- Store Unit	1	1	2	3	1 stage, 1 cycle
- System Unit	1	2	4	4	2 cycles
Thread Creations/Cycle	1	1	1	1	
Program Counter/Queue	16	32	32	32	
Register Files (Int/FP)	64/64	128/128	256/128	256/128	
Segments	16	32	32	32	
Execution Queue	32	32	32	32	
Branch Target Buffer	64	256	256	256	
Memory System	Size and Associativity			R/W Ports	Delay/Latency
- L1 I-Cache	4096 words 4 way-assoc			4/2	1 cycle if hit
- L1 D-Cache	4096 words 2 way-assoc			2/2	1 cycle if hit read/write
- L2 Cache	32768 words 4 way-assoc			4/2	10 cycles if hit
- Main Memory	6000000 words			2/2	50 cycles if hit

個の Fibonacci 数を求め，このループを計 5 回繰り返した．Fibonacci 数列計算は計算結果を求めるのに前回と前々回の結果を使用するため，このプログラムはループのイタレーション間に依存関係のある典型的なものである．

5.3 各プロセッサモデルのシミュレーション結果

図 4 にシミュレーション結果を示す．
これらの結果は，各プロセッサモデルでシングルスレッドのプログラムおよびマルチスレッド化されたプログラムを動かしたときの実行サイクルを比較したときの性能向上を表している．たとえば High プロセッサモデルならば，High プロセッサモデルでシングルスレッドプログラムを動かしたときの結果と，マルチスレッドプログラムを動かしたときの結果を比較したときの性能向上を示している．次に，**表 3** に各ユニットの実動回数の測定結果を示す．図 4 より，行列の乗算および Swim はループを多く含んでおり，その結果として並列性を比較的多く抽出することが可能であることから，プロセッサモデルの強度を上げるにつれてそれに見合った速度向上が得られている．

一方で，Fibonacci 数列計算は先に述べたループの

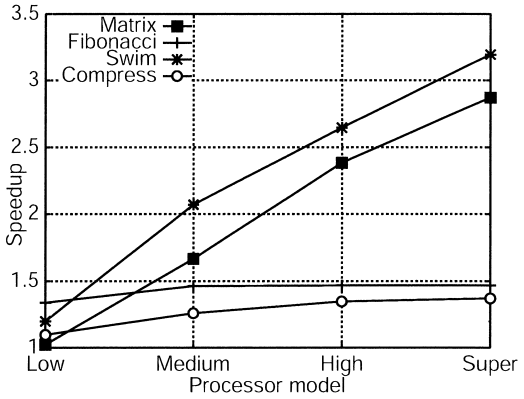

図 4 各プロセッサモデルでの実行速度向上率
Fig. 4 Performance improvement of each processor model.

イタレーション間の依存関係により，約 1.36 倍のところで頭打ちになっている．そして Compress では，基本的に並列性を抽出するのは難しく，主にプログラムに点在するループを DOALL 命令によって最適化を行った．その結果，最大で 1.49 倍の速度向上がみられた．また表 3 が示すように，LOAD・STORE 命

表 3 各ユニットの実動回数測定結果
Table 3 Number of working in each function unit.

		ALU	FPU	STORE	LOAD	JMP/BRANCH	THREAD	MISC	TOTAL
Matrix	Single	25761	0	400	9600	5255	1	0	41017
	Multi	25246	0	400	9600	4806	406	0	40458
	% (M / S)	98.0	n/a	100.0	100.0	91.5	40600.0	n/a	98.6
Fibonacci	Single	6191	0	2485	6150	1235	1	0	16062
	Multi	7505	0	1260	2480	6	2486	0	13737
	% (M / S)	121.2	n/a	50.7	40.3	0.5	248600	n/a	85.5
Swim	Single	5368025	408647	116804	1492727	24075	11	9998	7420287
	Multi	6359302	430231	92032	494999	100	24234	2742	7403640
	% (M / S)	118.5	105.3	78.8	33.2	0.4	220309.1	27.5	99.8
Compress	Single	162712	0	159239	194026	51425	1	19142	586545
	Multi	202217	0	143234	160745	42502	34158	17749	600605
	% (M / S)	124.3	n/a	89.9	82.8	82.6	3415800	92.7	102.4

令の削減もこの速度向上につながっている．マルチスレッド化を施したプログラムは，極力共有レジスタを使用して通信，同期を行うようにしている．それによりALUの使用頻度が上昇しているが，加算命令などはメモリアクセスなどのパイプラインを乱す外乱要因を持たない．L1 Cacheに2cycleの遅延を持たせた場合，Compressの速度向上は最大約1.45倍になった．つまりは，マルチスレッド化を施した場合はメモリアクセスが減少しており，結果として改善率が上がっている．

5.4 ハードウェアリソース変更の影響

5.4.1 PCQのサイズ，レジスタファイルサイズの変更

次にSuperプロセッサモデルを使用して，PCQのサイズ，レジスタファイルのサイズをそれぞれ変えてマルチスレッドプログラムを動かしたときのシミュレーション結果を，**図5**と**図6**に示す．性能比較の基準は，Superプロセッサモデルでシングルスレッドプログラムを動かしたときの実行サイクル数である．その際，PCQのサイズ，レジスタファイルのサイズはSuperプロセッサモデルのままの値を用いている．

PCQのサイズを変更したときの整数型レジスタファイルのサイズはすべて256個，浮動少数型レジスタファイルは128個と一定にしている．また，レジスタファイルを変更したシミュレーションでは，PCQのサイズはすべて32個で統一している．今回のシミュレーションでは，Fibonacci数列計算とCompressは行っていない．理由として，これらのプログラムは上記したとおり並列化が困難なことにより，PCQのサイズやレジスタファイルのサイズによらずに性能がほぼ一定になるためである．

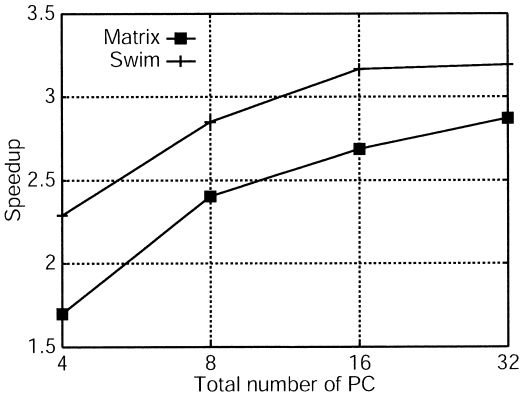

図5 Superプロセッサモデルで PCQのサイズを変化させたときの速度向上
Fig. 5 Performance improvement of Super processor model in changing total number of PCs.

まず行列乗算プログラムでは，PCQのサイズを増やしていくことにより性能は向上しているのが分かる．行列乗算の外側のループでは各イタレーション間には依存関係はないため，それを用いてより多くのスレッド生成を行うことができる．それによりSuperプロセッサモデルで与えられている命令発行数，実行ユニットを最大限に活用できるため，結果として性能が向上する．しかしながら，レジスタファイルサイズの変更においてはあまり影響が出ていない．これは行列乗算プログラムが多くのスレッドを生成できる一方で，各スレッドへのレジスタ割当て数は少なくてよく，その結果，レジスタ自体はさほど多くは必要ないという

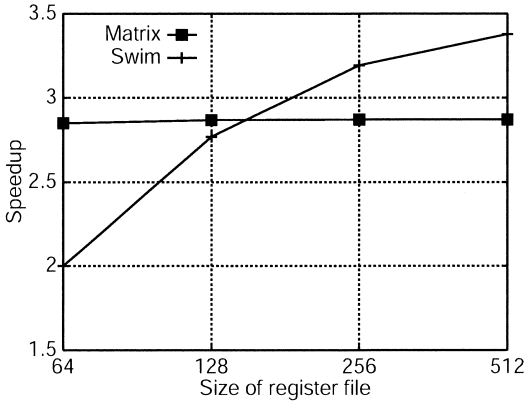

図 6 Super プロセッサモデルでレジスタファイルのサイズを変化させたときの速度向上
Fig. 6 Performance improvement of Super processor model in changing size of register file.

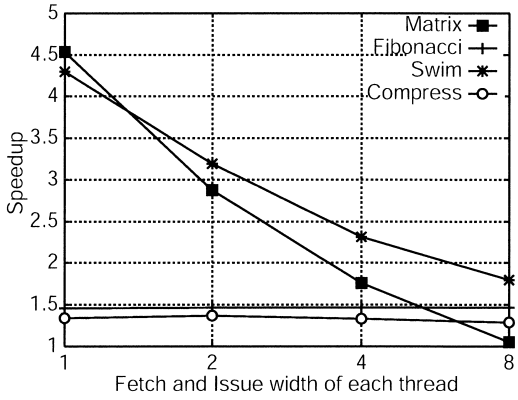

図 7 各スレッドの命令発行数を変更したときの性能
Fig. 7 Speedup in changing the instruction issue width of each thread.

ことがあげられる．今回のプログラムでは，行列乗算ループの各イタレーションの割り当てられたレジスタ数は 8 個である．つまり，プロセッサ全体のレジスタファイルサイズを 64 個としたときには，プロセッサ内に収まる最大スレッド数は 8 個であり，この状態においてすでに性能はほぼピークに達している．参考として，プロセッサのレジスタファイル数を 16 個，32 個にした場合の性能向上はそれぞれ，1.65，2.71 である．このことから，各スレッドのレジスタ消費数が少ない場合においては，レジスタセグメントの機構を適用することにより，従来の SMT プロセッサで用いられているレジスタファイルよりも小さなサイズで性能を向上させることが可能なのが分かる．

一方で Swim では PCQ サイズの変更，レジスタファイルサイズの変更，両方のシミュレーションにおいてそれぞれの数値を上げていくことにより，性能もそれに応じて高くなっていくのが分かる．PCQ サイズの変更は行列乗算のプログラムと同じ理由である．レジスタファイルサイズの変更での性能の変化は，行列乗算プログラムとは異なり，各スレッドに割り当てられるレジスタ数が多いためである（今回は整数型レジスタは 32 個，浮動少数型は 16 個）．これにより，サイズの大きいレジスタファイルを与えられていない場合，プロセッサ内に生成できるスレッド数が少なくなってしまうため，結果として性能向上が見込めなくなる．

5.4.2 各スレッドの同時命令発行数の変更

ここでは，各スレッドの同時命令発行数の変更にともなう性能の変化を示す．SMT アーキテクチャの理想の形態としては，スレッドが 1 つしか存在しないときには，そのスレッドがプロセッサ全体の命令発行数分だけ命令の発行が可能で，またスレッドが複数の場合は均等，もしくは何らかの優先順位のもとにプロセッサ全体の命令発行数を各スレッドに分割するのが望ましい．しかしながら，我々はハードウェア制御の複雑化を避けるために，各スレッドの同時命令発行数を固定している．つまりプロセッサ全体の命令発行数が 8 命令で，各スレッドの命令発行数が 2 命令と規定されていた場合，プロセッサ内にスレッドが 1 つしかなかった場合，理想としてはそのスレッドが 8 命令発行できればいいが，我々のシミュレーションではこのような状況においてもそのスレッドは 2 命令しか発行できないようにしている．そこで我々は，各スレッドの同時命令発行数を変更したとき，どのように性能に影響があるのかを評価した．

プロセッサモデルは Super を用い，各スレッドの同時命令発行数を 1，2，4，8 命令と変更した．表 2 でも示したとおり，プロセッサモデル Super におけるプロセッサ全体の同時命令発行数は 8 命令である．図 4 で示した Super プロセッサモデルの性能は，各スレッドの同時命令発行数は 2 であった．**図 7** に結果を示す．性能比較の基準は，スレッド命令発行数を 2 命令にした Super プロセッサモデルでシングルスレッドプログラムを動かしたときの実行サイクル数である．

図 7 から分かることは，性能の変化が顕著なのは行列の乗算と Swim である．この 2 つのプログラムは各スレッドの同時命令発行数を下げれば下げるほど，性能は向上している．この理由として，これらのプログラムは TLP が抽出しやすく，結果として多くのスレッドを生成することが可能である．そのためスレッドが大量に稼動している場合は，プロセッサ全体

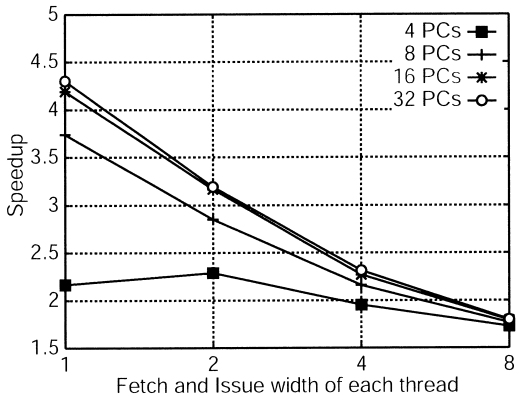

図 8 PCQ と各スレッドの命令発行数変更における性能
Fig. 8 Speedup in changing size of PCQ, and the issue width of each thread.

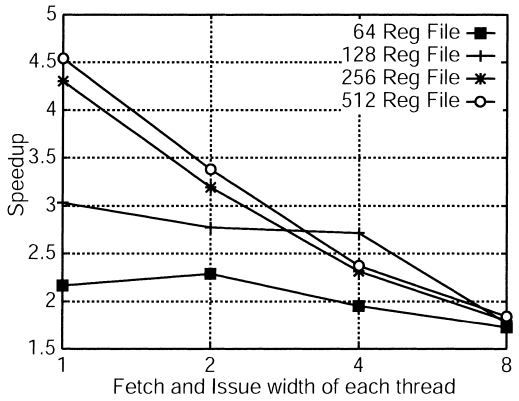

図 9 レジスタファイルサイズと各スレッドの命令発行数変更における性能
Fig. 9 Speedup in changing size of register file, and the issue width of each thread.

の同時命令発行数，機能ユニットを最大限に利用することができるので性能が改善する．Fibonacci および Compress は今述べた理由の逆があげられる．

ここで，スレッドの稼動数について触れたが，スレッド稼動数の議論は 5.4.1 項の，PCQ とレジスタファイルのサイズの変更による性能の変化のところでも触れていた．そこで，これらのパラメータをそれぞれ変えたシミュレーションの結果を**図 8**，**図 9** に示す．このシミュレーションは Swim についてのみ行った．その理由として，Swim は並列性を抽出しやすいが，各スレッドの使用レジスタ数が大きいので，プロセッサに非常に負荷のかかるプログラムためである．

図 8 において，PCQ のサイズが 4 のとき以外はすべて，スレッドの命令発行数が 1 のときが最も性能が向上している．PCQ サイズが 4 の場合，プロセッサ内に稼動できるスレッド数はたかだか 4 つで抑えられてしまうため，スレッドの命令発行数が 1 命令の場合，プロセッサ全体の命令発行数を最大限に利用することができないため，結果として性能が上がらなくなる．

また図 9 にレジスタファイルのサイズと，スレッドの命令発行数を変えたときの性能を示す．このとき，PCQ サイズは 32 個に固定している．図 8 と図 9 はおおよそ同じような性能の傾向を示している．レジスタファイルサイズが 64，128 のときには同時命令発行数を 1 にした場合においてもさほどの性能向上が得られていない．これはレジスタファイルのサイズにより，プロセッサ内に生成できるスレッドの個数が少なく抑えられてしまうためである．Swim ではレジスタを 32 個使うスレッドが多数存在し，たとえばプロセッサ全体のレジスタファイルサイズを 64 個としたときに生成できるスレッドは 2 つになる．このことからレジスタファイルを小さくすると，Super プロセッサモデルの発行命令数や実行ユニットを十分に生かしきれず，結果として性能が向上しなくなる．

図 8 および図 9 の両者から，PCQ サイズもしくはレジスタファイルのサイズによりスレッドの生成数が少なくなるときは，それに応じて性能が引き出せないということが分かる．またスレッド生成数が少なくする，つまり PCQ サイズを小さくするまたはレジスタファイルサイズを小さくする場合は，それに応じてプロセッサ全体での同時命令発行数を少なくすることでハードウェアの複雑化を避ける検討が必要になる．

5.4.3 Execution Queue サイズの変更

ここでは，Execution Queue（EQ）のサイズ変更にともなう性能への影響を調べる．**図 10** に，EQ のエントリ数を 8，16，32，64，128 命令にしたときのシミュレーションの結果を示す．性能の比較基準は EQ のエントリ数を 128 命令にした Super プロセッサモデルで，シングルスレッドプログラムを動かしたときの実行サイクル数である．

α-Coral はスーパスカラプロセッサをベースにしているので，EQ を拡大することにより，それに応じた性能向上を示している．Fibonacci 数列および行列の乗算の場合はエントリ数 32 で性能は飽和しているが，Swim に関しては，サイズを大きくするたびに性能が向上しているのが分かる．これは，Swim が浮動小数演算を多用しているためにである．基本的に浮動小数演算は結果を得るまでに数サイクルかかるので，どうしても EQ はそれらの命令に占有されてしまう．つまりは，Swim の場合は他のプログラムと比べてより EQ のエントリ数を必要とする．

EQ に関してもハードウェアの複雑さが問題になる．

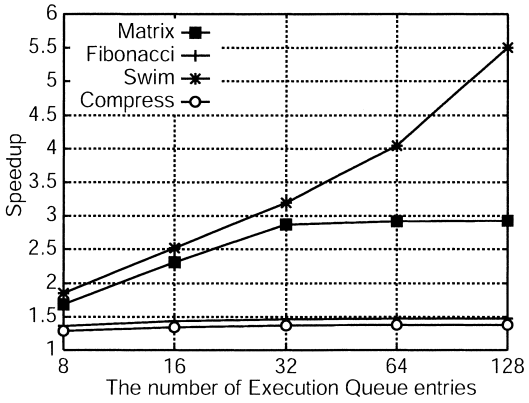

図 10 Execution Queue のエントリ数を変更したときの性能
Fig. 10 Speedup in changing the number of entries in Execution Queue.

スーパスカラプロセッサの問題点として，命令ウィンドウの連想検索機構があげられる．これについての詳細な議論は Palacharla らが行っている[1]．α-Coral も，新たな Out-Of-Order（OOO）機構（たとえば文献 22)，23)）を実装する，もしくは OOO を採用しない[17]といった対策をとらない場合は，この問題点をスーパスカラアーキテクチャからそのまま継承することになる．

5.5 シミュレーションのまとめ

今回のシミュレーションから，以下に示す知見が得られた．

(1) ベンチマークプログラム Swim において，Super プロセッサモデルで実行した場合最大 3.19 の性能向上が得られた．
(2) 並列化困難なプログラム Compress においては，最大 1.36，Fibonacci 数列演算においては最大 1.46 の性能向上が得られた．
(3) レジスタセグメンテーションにより，小規模のレジスタファイルでも性能向上が得られた．
(4) 各スレッドの同時命令発行数を 1 命令にすることにより，並列化が比較的容易に行えるプログラムにおいてはより高い性能向上が得られることが分かった．

今回のシミュレーションで行列乗算プログラムでは各スレッドの使用するレジスタ数が少なく，また Swim の場合は各スレッドが使用するレジスタ数が多かった．このため，行列乗算プログラムは 3 番目の知見がそのままあてはまるが，Swim に関しては各スレッドに割り当てるレジスタ数が多いため，プロセッサ全体のレジスタファイルが小さい場合は，ある程度のところで性能が抑えられてしまうのが分かっている（図 6）．今回はベンチマークプログラムに対して手作業でマルチスレッド化を行ったが，各スレッドに割り当てるレジスタ数を少なくしようとする場合はレジスタ割当ての問題が絡み，コンパイラとの連携した検討が必要である．またこれとは別に，ハードウェアによるレジスタのスワッピング機構の導入なども今後検討すべき点である．

また 4 番目の知見においては，今回の α-Coral シミュレータでリオーダバッファなどのレジスタリネーミングをサポートしていない場合での結果である．もしこのリネーミングロジックをサポートした場合は，必ずしもこの 4 番目の知見がそのままあてはまるとは限らなくなる．レジスタリネーミングは RAW 以外の依存関係を解決するので，今回の α-Coral で用いた Out Of Order（OOO）実行の手法に比べて，より良い性能が得られると考えられる．たとえば，今回のシミュレーションでは各スレッドの同時命令発行数を 4 命令にした Super プロセッサモデルでマルチスレッド化された行列乗算プログラムを動かした場合，命令発行の時点で WAW 依存関係が 20431 回検出された．表 3 から，行列乗算プログラムの総実行命令数は 40458 命令なので，この WAW 依存の回数は実行性能に影響していると考えられる．よって α-Coral で OOO でのリネーミングロジックを取り入れた場合，各スレッドの同時命令発行数を 2 命令から 8 命令に設定した場合，また違う傾向の結果が得られる可能性がある．シミュレータでのリネーミングロジックのサポート後の評価は，今後の重要な検討事項である．

6. ハードウェアコスト

具体的なハードウェアのコストについては現段階での詳細な見積りは行えないが，たとえば文献 24) では DLX アーキテクチャをベースとした SMT プロセッサのトランジスタ数の見積りを行っており，この見積りに使われた KSMS エスティメータ（Microsoft Excel のマクロデータ）が公開されており[26]，この KSMS もとに見積りを行った．この KSMS は SMT プロセッサの各種詳細なデータを入力することによりトランジスタ数の見積り，チップ面積の概算が行える．しかし本論文で論じている α-Coral はこの KSMS のデータ設定部分において，いくつかあてはまらないアーキテクチャ部分が存在する．具体的には以下のようなものである．

- α-Coral は PCQ，RFST などの他の SMT アーキテクチャに例をみないユニットを持つ．
- α-Coral はレジスタセグメンテーションの機構を

表 4 ハードウェアコストの見積り
Table 4 The estimation of the hardware.

	Transistor (Ktrans)	Area (Mlambda^2)	Area (mm^2)
Reg 256 (PCQ 32)	16617	22858	182.94
Reg 128 (PCQ 16)	12094	15459	125.22
Reg 64 (PCQ 8)	9830	11887	96.28

持つ．

- α-Coral は OOO 実行の実現方法であるリネームロジックを有していない．
- α-Coral の EQ は集中制御方式（命令ウィンドウ型）である（KSMS はリザベーション型，つまり分散制御方式）．

このほかにもデータパスが異なっているなど，あてはまらない部分は存在するが，それをふまえた上でいくつかの見積りの結果を**表 4** に示す．KSMS は各スレッドに割り当てるレジスタファイルが固定のため，プロセッサ全体のレジスタファイルサイズを変更するには 2 通りの方法を用いらなければならない．1 つは PC の数を変えることでプロセッサ全体のレジスタファイルサイズを変更する，もう 1 つの方法は各スレッドに割り当てるレジスタファイルサイズを変更する，というものである．ここで Super プロセッサモデルの場合，PCQ サイズが 32 でレジスタファイルサイズが 256 なので，すべての PC を使う場合に各スレッドに割り当てられるレジスタファイルサイズは 8 である．これを用いた見積りの際，各スレッドに割り当てるレジスタファイルサイズを 8 に固定し，PC の数を変更することでプロセッサ全体のレジスタファイルサイズを変更した．表 4 の中のチップエリアサイズ（mm^2）の数値は，0.18μm プロセスで計算したときの面積である．この表の数値は参考値ではあるが，レジスタファイルサイズを小さくしていくことで，格段にハードウェアコストが縮小されていくのが分かる．これは PC の数が少なくなるため，それだけ複雑さが緩和しているためと考えられる．参考として PC の数を 8，各スレッドに割り当てるレジスタファイルサイズを 32 とし，プロセッサ全体のレジスタファイルサイズを 256 にしたときのトランジスタ数は 10075K であった．レジスタファイルサイズでの比較を行うと，α-Coral は通常の SMT アーキテクチャに比べ劣っているのが分かるが，プロセッサ内に内包できるスレッドの数（PC の数）をもって比較すると，現状の見積りでは α-Coral が有利である．

より詳細なハードウェアコストの見積りは今後の課題である．

7. ま と め

本論文ではオンチップマルチスレッドアーキテクチャ α-Coral の性能を詳細に評価した．α-Coral は PROMIS コンパイラの生成するあらゆる粒度のスレッドを，スレッド制御命令で柔軟にサポートし，プロセッサ内部の共有レジスタを用いることで，スレッド間通信の遅延を最小限に抑える．また，PROMIS はプログラムから多粒度の並列性を生成し，α-Coral の性能を引き出すことが可能にする．

現在，α-Coral はシミュレータとして稼動しており，これを用いてシミュレーションを行った結果，シングルスレッドのプログラムと比べ，同一環境下でもプログラムにマルチスレッド化を施した場合は速度向上がみられることが確認できた．また現在，スレッドの生成，スレッドのスイッチングポリシ，デッドロック回避の検討も行っている．

今後はハードウェアとしての実測，評価を行い，現在シミュレータで実現している機能の改良，削減を行う．

参 考 文 献

1) Palacharla, S., Jouppi, N.P. and Smith, J.E.: Quantifying the Complexity of Superscalar Processors, Technical Report, Univ. of Winsconsin-Madison (1996).
2) Ynkelevsky, M.N.: The α-Coral Architecture for Hardware Multithreading: Concepts, Simulation, and Compilation, Master Thesis, CSRD (2000). http://www.csrd.uiuc.edu/acoral/
3) Ynkelevsky, M.N. and Polychronopoulos, C.: alpha-Coral: A Multigrain, Multithreaded Processor Architecture, *International Conference on Supercomputing* (*ICS 2001*), pp.358–367 (2001).
4) Saito, H., Stavrakos, N., Carroll, S., Polychronopoulos, C. and Nicolau, A.: The design of the PROMIS compiler, *Proc. Int'l Conf. on Compiler Construction* (*CC*) (1999). Also available in Lecture Notes in Computer Science No.1575 (Springer-Verlag) and as CSRD Technical Report No.1539 (rev.1) (1999).
5) Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A. and Smith, B.: The Tera computer system, *Proc. Int'l Conf. on Supercomputing* (*ICS '90*), pp.1–6 (1990).
6) Agarwal, A., Kubiatowicz, J., Kranz, D., Lim, B., Yeung, D., D'Souza, G. and Parkin, M.: Sparcle: An Evolutionary Processor Design

for Large-Scale Multiprocessors, *IEEE Micro*, pp.48–61 (1993).
7) Sohi, G.S., Breach, S.E. and Vijaykumar, T.N.: Multiscalar Processors, *Proc. 22nd Int'l Symp. on Computer Architecture* (*ISCA '95*), pp.414–425 (1995).
8) 鳥居 淳，近藤真己，本村真人，池野晃久，小長谷明彦，西 直樹：オンチップ制御並列プロセッサMUSCATの提案，情報処理学会論文誌，Vol.39, No.6, pp.1622–1631 (1998).
9) Tsai, J.-Y. and Yew, P.-C.: The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation, *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques* (*PACT '96*), pp.35–46 (1996).
10) 小林良太郎，岩田充晃，安藤秀樹，島田俊夫：非数値計算プログラムのスレッド間命令レベル並列を利用するプロセッサ・アーキテクチャSKY，並列処理シンポジウム（JSPP98）論文集，pp.87–94 (1998).
11) Hammond, L., Hubbert, B., Siu, M., Prabhu, M., Chen, M. and Olukotun, K.: The Stanford Hydra CMP, *IEEE MICRO Magazine* (2000), and presented at Hot Chips 11 (1999).
12) Tullsen, D.M., Eggers, S.J. and Levy, H.M.: Simultaneous multithreading: Maximizing on-chip parallelism, *Proc. Int'l Symp. on Computer Architecture* (*ISCA '95*), pp.392–403 (1995).
13) 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 Multi threading, *ACM Trans. Comput. Syst.*, pp.322–354 (1997).
14) http://www.intel.com/
15) Wallace, S., Calder, B. and Tullsen, D.M.: Threaded Multiple Path Execution, *Proc. Int'l Symp. on Computer Architecture* (*ISCA98*), pp.238–249 (1998).
16) Chappell, R.S., Stark, J., Kim, S.P., Reinhardt, S.K. and Patt, Y.N.: Simultaneous Subordinate Microthreading (SSMT), *Proc. Int'l Symp. on Computer Architecture* (*ISCA99*), pp.186–195 (1999).
17) Hily, S. and Seznec, A.: Out-Of-Order Execution May Not Be Cost-Effective on Processors Featuring Simultaneous Multithreading, *IRISA Report*, No.1179 (1998).
18) Hily, S. and Seznec, A.: Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading, Workshop on MultiThreaded Execution, Architecture and Compilation, Colorad State Univ. Technical Report, CS-98-102 (1998).
19) Girkar, M. and Polychronopoulos, C.D.: The Hierarchical Task Graph as a Universal Intermediate Representation, *Int'l Journal of Parallel Programming*, pp.519–551 (1994).
20) Stavrakos, N., Carroll, S., Saito, H., Polychronopoulos, C. and Nicolau, A.: Symbolic Analysis in the PROMIS Compiler, CSRD Technical Report, No.1564 (1999).
21) Girkar, M., Haghighat, M.R., Grey, P., Saito, H., Stavrakos, N.J. and Polychronopoulos, C.D.: Illinois-Intel Multithreading Library: Multithreading Support for Intel Architecture Based Multiprocessor Systems, *Intel Technology Journal 1998 Q1 issue* (1998).
22) 五島正裕，グェンハイハー，縣 亮慶，森眞一郎，富田眞治：Dualflowアーキテクチャの提案，並列処理シンポジウム（JSPP2000）論文集，pp.197–204 (2000).
23) 佐藤寿倫，中村佑介，有田五次郎：大規模スーパスカラプロセッサ向け命令発行機構，信学技報 ICD2000-144, pp.107–112 (2000).
24) Sigmund, U., Steinhaus, M. and Ungerer, T.: On Performance, Transistor Count and Chip Space Assessment of Multimedia-enhanced Simultaneous Multithreaded Processors, *Workshop on Multi-Threaded Execution, Architecture and Compilation* (*MTEAC-4*) (2000).
25) Steinhaus, M., Kolla, R., Ungerer, T., Larriba-Pey, J.L. and Valero, M.: Transistor Count and Chip-Space Estimation of Simulated, CEPBA, Technical Reports, UPC-CEPBA-2001-11 (2001).
26) Steinhaus, M. and Ungerer, T.: Hardware Complexity of Processors (2001). http://goethe.ira.uka.de/people/ungerer/complexity.html

(平成13年9月 4 日受付)
(平成14年2月13日採録)

河原 章二（学生会員）
昭和53年生まれ．平成13年東京農工大学工学部電子情報工学科卒業．現在，同大学大学院修士課程に所属．プロセッサアーキテクチャ，並列処理システムに興味を持つ．

Mark Yankelevsky

He received his M.Sc. from University of Illinois at Urbana-Champaign in 2000, B.Sc. degree in computer engineering from the University of Illinois at Urbana-Champaign in 1996. Currently, he is a Ph.D. candidate in the Department of Electrical and Computer Engineering, and a research assistant in the Center for Supercomputing Research and Development at the University of Illinois. His research interests include simultaneously multithreaded architectures, parallelizing compilers and multithreaded code generation.

中條　拓伯（正会員）

1961 年生まれ．1985 年神戸大学工学部電気工学科卒業．1987 年同大学大学院工学研究科修了電子工学専攻．1989 年神戸大学工学部システム工学科（後に情報知能工学科）助手を経て，現在，東京農工大学工学部情報コミュニケーション工学科助教授．1998 年より 1 年間 Illinois 大学 Urbana-Champaign 校 Center for Supercomputing Research and Development（CSRD）にて，Visiting Research Assistant Professor．プロセッサアーキテクチャ，分散共有メモリ，クラスタコンピューティングに興味を持つ．電子情報通信学会，IEEE CS 各会員．博士（工学）．

Constantine Polychronopoulos

He is a Professor in the Department of Electrical and Computer Engineering and the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. He received his Ph.D. from the University of Illinois at Urbana-Champaign in 1986, his M.Sc. from Vanderbilt University in 1982, and his B.Sc. from the University of Athens in 1980. His research interests are on compilers and architectures for high-performance computer systems, multithreading, and multiprocessor operating systems. He was the recipient of a 1989 NSF Presidential Young Investigator award, and is a Fulbright Scholar.