システムＬＳＩ設計技術　94－20
（2000. 1. 12）

KIT COSMOSプロセッサ：背景と着想

佐藤 寿倫　　有田 五次郎
九州工業大学 情報工学部 知能情報工学科
{tsato,arita}@mickey.ai.kyutech.ac.jp

あらまし　半導体技術の進歩によるトランジスタ集積度の向上を有効利用可能なマイクロプロセッサアーキテクチャを提案する．集積度の向上により１チップに搭載可能な演算器の数は増大し続けているが，従来のマイクロプロセッサ技術では，増大した演算器を有効利用できていない．本提案は空間多重マルチスレッド方式を応用し，有意義な命令の数を増やす事でこれらの演算器を有効利用する．アプリケーションプログラムと，それを実行時に最適化するオプティマイザを同時実行する事で，有意義な命令数を増やすと同時に，アプリケーションプログラムの実行性能を向上する．我々はこのConcurrent Dynamic Optimizer を利用したCOSMOSプロセッサを検討している．本稿では，この検討に至った背景と着想について説明する．
キーワード　命令レベル並列処理，動的最適化，空間多重マルチスレッド

The KIT COSMOS Processor: Background and Rationale

Toshinori Sato and Itsujiro Arita
Kyushu Institute of Technology
{tsato,arita}@mickey.ai.kyutech.ac.jp

Abstract　In this paper, we propose a microprocessor architecture which efficiently utilizes next-generation semiconductor technology. While the technology makes it possible to integrate a lot of functional units on a single chip, contemporary microprocessors can not exploit much instruction level parallelism so that the units are wasted. Our proposal based on Simultaneous Multi-Threading (SMT) increases the number of available instructions and thus the functional units work efficiently. As well as application programs, a binary code translator which dynamically optimize the applications on-the-fly is executed on the SMT. We call this mechanism CONcurrent Dynamic OptimizeR (CONDOR). Based on the CONDOR, performance of the applications are improved as well. We are currently studying the KIT COSMOS processor which utilizes the CONDOR. This paper describes background and some ideas behind the architecture.
key words　instruction level parallelism, dynamic optimization, simultanenous multithreading

1. まえがき

現在に至る半導体技術の発展は目覚しく，2010年には800Mトランジスタを集積するプロセッサの出現が予測されている[4]．そのような多量のトランジスタを有効に利用して性能に貢献させるためには，従来のマイクロプロセッサアーキテクチャでは不十分である．なぜなら，いくら演算器を追加して同時に実行可能な命令の数を増加させようとしても，プログラムが本来持っている命令レベル並列性 (ILP:Instruction Level Parallelism) が小さければ，同時に実行可能な演算は限られ，演算器は有効に使われないまま無駄になるだけだからである．

プロセッサが処理可能なILPを増やすために，従来は投機的実行や複数パス実行が行なわれている．分岐予測を用いた制御フローの投機的実行[20]は，将来実行されると予想されるパス上にある命令を実行するので，予測が間違っていた場合には，投機の開始点からの全命令の実行結果を破棄し，正しいパス上の命令を改めて実行しなければならない．すなわち，無駄な演算を追加することで同時に実行可能な命令を増やしていることになる．データ予測を用いたデータフローの投機的実行[16]は，データ依存関係に違反して命令を実行するので，予測が間違っていた場合には投機を開始した命令と依存関係にある全ての命令の実行結果を破棄し，それらの命令を再実行しなければならない．つまり，意味の無い演算を追加することで同時に実行可能な命令を増やしていることになる．さらに，再実行の対象になる命令の中に分岐命令が含まれている場合には分岐の方向が間違っている場合もあり，制御フローの投機失敗と同様な操作も必要となる．いずれにせよ，投機的実行は予測に基づいているため，予測が間違った場合には増加できた命令は無駄な演算だったということになる．

一方，複数パス実行[2),25]は，制御フローの分岐点で将来実行される可能性のある複数のパス☆を実行することで，分岐予測失敗時に制御フローの投機的実行が被るペナルティを削減することを目的としている．したがって，分岐方向の確定時に選択されなかったパスにある全ての命令の実行結果は破棄されることになる．分岐予測の精度や信頼度にしたがって意味の無い命令の実行を削減する検討がされているが[11),17]，方式の性質上無駄が無くなるわけではない．

以上のように，投機的実行と複数パス実行は無駄になる命令実行を覚悟の上で高いILPを抽出しようとする方式であり，演算器が使用されなくて余っているという仮定の元で正当化されてきた．実際，これまで二つの方式はマイクロプロセッサの性能向上に貢献しており，有効であることに疑いの余地は無い．しかし，投機の成功率が向上していくと使用されないで遊んでいる演算器は減り，より一層の恐らくは成功率の低い投機は好ましくない結果をもたらす可能性がある．複数パス実行も同様である．つまり，分岐予測方式やデータ予測方式が十分成熟した先では，無駄な命令実行を伴う投機的実行や複数パス実行は望ましくないことになる．

そこで，我々は無意味な命令ではなくて有意義な命令を増やすことでILPを向上させることを検討している．あるアプリケーションプログラムの実行時に余っている機能ユニットを利用して，そのアプリケーションのパフォーマンスを向上させる．すなわち，アプリケーションと，それを動的に最適化するオプティマイザとを同時実行させることを考えている．我々はこの方式をを CONcurrent Dynamic OptimizeR(CONDOR) 方式と呼んでいる．注意して欲しいのは，同時に複数のスレッドが実行されるので，一つのスレッドが実行される場合よりも同時に実行される命令が増えている，すなわち全てのスレッドのトータルでILPが改善されている，と主張しているわけではない．アプリケーションが動的に最適化されるために，そのパフォーマンスが向上する，すなわちアプリケーションプログラムだけを考慮した時のILPが向上できると主張している．上述した多重スレッドの同時実行は，空間多重マルチスレッド (SMT:Simultaneous Multi-Threading) 方式[10),24]が適している．そこで我々はSMTを元にして，CONDOR方式を利用するCOSMOSプロセッサ☆☆の検討を開始した．

以下，2節と3節でSMT方式と動的最適化技術を概観し，CONDOR方式を説明する．つづいて4節でCONDOR方式およびCOSMOSプロセッサを実現するための課題と対策を述べる．5節で関連する研究をまとめ，6節で結論とする．

2. 空間多重マルチスレッド

本節ではCONDOR方式がベースとしているSMT[24]の説明をする．SMTは2002年頃に市場に登場予定のコンパック社EV7プロセッサに採用されることが決定しており[10]，次世代マイクロプロセッサアーキテクチャの有力候補である．

SMTは複数のスレッド (あるいはアプリケーション) が同時にプロセッサ資源を共有することを可能にする．複数のアプリケーションが同時に実行されることで，プロセッサはILPとスレッドレベルの並列性 (TLP:Thread Level Parallelism) を使い分けることができる．プロセッサ資源を共有しているので，TLPがILPとして処理可能なわけである．例えば，ILPの小さなアプリケーションの実行中には複数のスレッドを実行して機能ユニットの利用効率を向上でき，逆にTLPが得られない時にはILPを十分に引き出せるように機能ユニットを利用する．

図1は，様々なアーキテクチャに基づいたプロセッサがどのように機能ユニットを活用できるかを表している[10]．縦方向は時間の経緯を，横方向の四角は命令発行スロットあるいは機能ユニットを表している．図1(a)～(d)は，それぞれスーパースカラ，マルチプロセッサ，細粒度マルチスレッド，SMTの場合に対応している．(a)のスーパースカラプロセッサは，アプリケーションに十分なILPが無い場合には機能ユニットを有効利用できない．(b)のマルチプロセッサでは，TLP処理のためにプロセッサ資

☆ 理想的には全てのパス

☆☆ 秋桜は飯塚の市の花である．

源を物理的に分割しているので，例えば並列プログラム内の逐次処理部分など，TLPが得られないような場合には機能ユニットを有効利用できない．(c) の細粒度マルチスレッドプロセッサでは，時間軸方向の空きスロットは埋めることができるが，各スレッドに十分なILPが無ければ横方向の空きスロットを埋めることはできない．これらと比較して (d) のSMTプロセッサでは，プロセッサ資源を物理的に分割すること無く複数のスレッド間で共有しているので，機能ユニットを非常に効率良く利用できている．

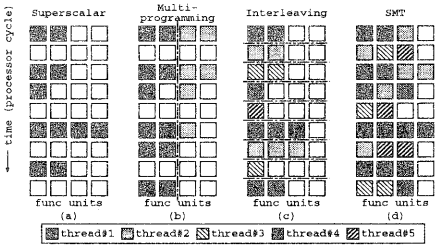

図 1 SMT の命令発行ポリシー
Fig. 1 Instruction issue policy

SMTのもう一つの特徴は実現が非常に容易なことである．今日のスーパースカラプロセッサに僅かなハードウエアを追加するだけで，SMTプロセッサに拡張することが可能である．追加すべき点は，

- 多重スレッド分のプログラムカウンタ
- スレッド数に見合った大きなレジスタファイル
- スレッド毎に命令をリタイア等させるための機構
- スレッド識別子を考慮した分岐予測機構

などである．逆に，機能ユニット，キャッシュ，アドレス変換機構などには全く手を加える必要が無い．

3. 動的最適化

動的最適化とは，コンパイラによる静的な最適化とは異なり，バイナリコードの生成時ではなく実行時に最適化を行なう技術である[3]．最適化の前後で命令セットに変更は無いので，基本的には最適化後の命令の実行に特別なハードウエアの支援を必要としない．動的最適化は，JITコンパイラとは異なり入力に変換が行なわれることはないし，またある種の動的コンパイルとも異なりプログラマがソースコード中にヒントを挿入する必要もない．つまり，動的最適化は最適化の1ステップに過ぎない．

動的最適化は静的には発見できない命令シーケンスに対して有効である．例えば，実行時に動的にリンクされるプロシジャのバウンダリを越えた最適化が可能になる．オブジェクト指向言語の趨勢やDDL(dynamic linked libraries) によるバイナリの流通などの傾向から，近年は静的最適化が困難な状況になりつつあるので動的最適化の活躍できる機会は多いと考えられる．すなわち，動的最適化は静的最適化を置き換えるものではなく，両者はお互いに補間し合う関係にある．

最適化には実行時に採取されたプロファイル情報が利用される．プロファイル情報に基づいて，実行頻度の高いトレースに対して最適化を施す．実行時間の大部分を占める少数のトレースが最適化されるだけでも，アプリケーションのパフォーマンスへの貢献は大きい．プロファイルを収集している間は，アプリケーションはインタプリタ実行されることになる．ここで，インタプリタ実行される命令はターゲットマシンの機械命令であることに注意されたい．最適化前のバイナリは変更されないで維持される．最適化後のバイナリをある記憶空間に保持しておき，そこに制御を移動することで最適化後のバイナリを実行する．

図2に示すように，動的最適化は以下のとおりに実行される．(1) アプリケーションはインタプリタ実行として開始される．(2) オプティマイザはインタプリタ実行の間，トレースの形成やプロファイル情報の採集を行なう．(3) プロファイル情報を基に，あるトレースの実行頻度が予め設定された閾値を越えたことが分かると，オプティマイザはアプリケーションの実行を停止し，そのトレースに対して最適化を開始する．最適化後のバイナリは，特定の記憶領域に保持される．(4) 最適化が終了すると，オプティマイザはアプリケーションの実行を再開する．(5) インタプリタ実行中に既に最適化されたトレースへの分岐が検出されると，インタプリタ実行を最適化後のバイナリの実行に切り替える．(6) 最適化バイナリを実行する．(7) 最適化バイナリの実行中に，未だ最適化されていないトレースに分岐すると，インタプリタ実行に戻り上記と同様の操作を継続する．

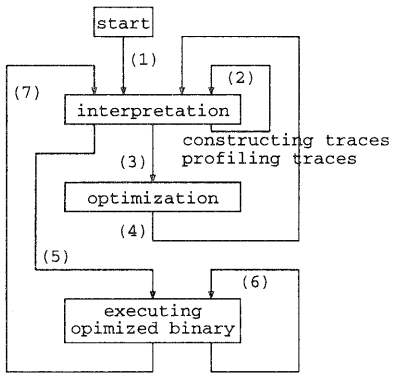

図 2 動的最適化
Fig. 2 Dynamic optimization

3.1 Concurrent Dynamic Optimizer

本節でCONDOR方式を説明する．簡単に言えば，CONDOR方式はSMT上でアプリケーションと動的最適化のためのオプティマイザを同時に実行する方式である．アプリケーション実行時に利用されないで遊んでいる機能ユニットを用いて，動的最適化を行なう．動的最適化の結果，アプリケーションのILPが向上する．つま

り，利用されていない命令スロットに，アプリケーションとは別の有効な命令を投入して，アプリケーションのパフォーマンスを向上させるわけである．アプリケーションに並列性が無ければ，十分機能ユニットは余っているので，オプティマイザを実行させる余裕がある．逆に，並列性が高く機能ユニットが余っていない時は，オプティマイザを実行する必要はない．

図 3 に，従来の動的最適化方式と CONDOR 方式の違いを示す．図 3(a) が従来の方式であり (b) が CONDOR 方式である．時刻は縦方向に上から下に向かっており，命令は四角で表されている．命令 `1,2,3...` がアプリケーションプログラムで，命令 `A,B,C...` がオプティマイザである．アプリケーションは動的に最適化され，命令 `1',2'3'...` となる．

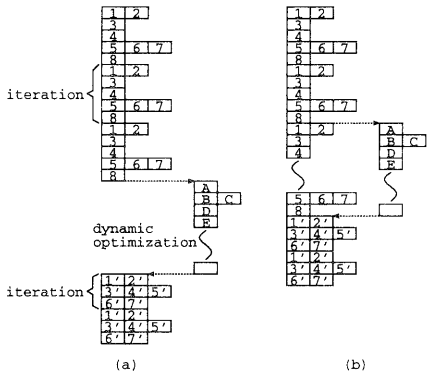

図 3 CONDOR 方式
Fig. 3 Concurrent dynamic optimizer

(a) で説明されている従来の動的最適化では，あるループの繰り返し回数が閾値を越えると，オプティマイザが起動され動的な最適化が開始される．最適化の間はアプリケーションプログラムの実行は停止している．オプティマイザの実行が完了すると，最適化後のアプリケーションが再開される．最適化に必要なサイクルは，アプリケーションの実行にとってはオーバーヘッドとなるので，十分繰り返しの多い場合を選んで最適化する必要がある．つまり，オプティマイザを起動するまでの閾値を高くする必要がある．

一方 (b) で説明される CONDOR 方式では，動的最適化の実行中の間もアプリケーションは実行されている．オプティマイザは遊んでいる機能ユニットを利用して最適化を行なう．最適化が完了すると，アプリケーションの実行は最適化後の命令に切り替わる．すなわち，CONDOR 方式では最適化に必要なオーバーヘッドを隠蔽可能である．さらに，オーバーヘッドが無いので最適化を開始するまでの閾値を低くできる．ループが検出されれば直ちにオプティマイザを起動することも可能であるし，あるいは最適化実行中にループの繰り返し回数が小さいことがわかればオプティマイザのスレッドを破棄することも可能である．

以上のように，CONDOR 方式は従来の動的最適化方式と比較して，最適化に要するオーバーヘッドを小さくできる．

3.2 応　　用

本節で CONDOR 方式の適用例を検討する．CONDOR 方式は一般的な最適化方式であるので，コンパイル時に可能な最適化の適用は容易である．これらの基本的な最適化は，元のバイナリコードの最適化が十分でない場合に特に有効であると考えられる．例えば，古くから利用し続けられているバイナリを考えると，最新のより高度な最適化の適用やプロセッサのハードウエア構成を考慮した最適化の適用などが可能である．

CONDOR 方式を用いると，上述したような基本的な最適化だけではなく，プロファイル情報を利用した高度な最適化も可能になる．そのような最適化には以下の項目が含まれる．

- 分岐予測：分岐命令には分岐方向の偏りが激しいものが存在する．このような分岐命令は静的に分岐方向の予測を行なうように最適化を施す．そうすることで分岐予測器のエントリが実質的に増加し，他の予測の困難な分岐命令の予測精度を向上できる．
- プリフェッチング：キャッシュミスのプロファイル情報に基づいて，プリフェッチの必要なロード命令だけを選択し，最適な場所にプリフェッチ命令を挿入することができる．
- ストア・ロード命令間の曖昧な依存性解消：静的コンパイル時には決定できなかったストア・ロード命令間の依存性を解析し，不要に命令発行を遅らせたり間違って命令発行してしまうことをなくす．
- データ予測：データ投機実行を有効に利用するためには，投機の対象となる命令をデータ予測容易なものに絞ることが重要である．また，予測精度を向上させるためには，複数の予測器を利用する混成型予測器が有効である．これらの目的にも CONDOR 方式は適用可能である．つまり，プロファイル情報に基づいて，予測対象の命令を指定したり，予測器の選択を静的に決定したりすることが可能になる．
- トレースキャッシュ[21]の置き換え：CONDOR 方式は実行時にトレーススケジューリングを行なっていることと同等である．したがって，トレースキャッシュをソフトウエアで実現していることになる．
- 命令のパッキング (ベクトル化)：SIMD 命令を仮定していないバイナリに対して，操作するデータ幅の小さい同じタイプの演算を SIMD 命令にパッキングする．SIMD 命令は短いベクトル命令であるので，命令のベクトル化を行なっているのと同等である．
- 複数パス実行：CONDOR 方式は複数パス実行を否定しているわけではない．CONDOR 方式によって予測が本質的に困難な分岐命令の検出が可能になるので，それらの分岐命令だけを動的にプレディケート実行すれば，無駄に実行される命令の増加を抑えつつ複数パス実行が可能になる．
- 実行時スレッド分割：情報不足のためにコンパイラ

には不可能な多重スレッドの分割を行なう．スレッド分割を考慮していないバイナリを静的にオフラインでスレッド分割する方法[14]を応用できると考えている．さらに，プロファイル情報を用いれば精度の向上が期待できるので，データ投機に基づいた投機的スレッド実行も実現性が増すと思われる．

- コヒーレンス制御：マルチプロセッサシステムでのコヒーレンス制御を投機的に行なうことが検討されている[15]が，動的なプロファイル情報を詳細に解析できるCONDOR方式は，この方面でも利用できると考えられる．

静的最適化でもプロファイルを利用することは可能であるが，以下の2点から動的最適化で利用できる場合の方がより効果的であると考える．

- プロファイルを採取するのは非常に手間のかかる作業であるので，実行時に採取されればプログラマの手間が省ける．
- プロファイルを採取したアプリケーションと最適化が施されるアプリケーションが同一なので，非常に正確なプロファイル情報を利用できる．

4. 課　　題

これまでCONDOR方式の有効性だけを述べてきた．しかしCONDOR方式を実現するためには，様々な問題を解決しなければならない．また，CONDOR方式をCOSMOSプロセッサ上に実装するためには，ハードウエア上の制約も検討する必要がある．本節では，これらの課題をまとめ，いくつかの解決策を検討する．

4.1 最適化バイナリを格納する記憶領域

まず，最適化後のバイナリをどのように保持するか検討する必要がある．動的最適化をソフトウエアのみで実現しようとすると，未使用のメモリマップ領域に割り付けたり[8]，ソフトウエア的に最適化バイナリのキャッシュをオプティマイザに用意させたり[3]する必要がある．いずれにせよ最適化後のバイナリは主メモリ上に置かれることになる．一方，ハードウエアキャッシュを用意する方法もある．つまり，ソフトウエアでコントロールされるトレースキャッシュである．このハードウエアキャッシュをターボキャッシュと呼ぶことにする．

以下の考察から，現在はターボキャッシュを実装する方が好ましいと考えている．

- 理想的には，アプリケーションは全てターボキャッシュに格納され，CONDORは従来の命令キャッシュに格納される．すなわち，命令キャッシュにおける二つのスレッド間の干渉を防ぐことができる．
- 最適化後のバイナリで，それらを保持するための記憶領域が一杯になった時には，入れ換え等を行なう必要がある．この操作をソフトウエアのみで実現すると，パフォーマンスへのペナルティが大きいと予想される．

4.2 CONDORの起動

CONDORがアプリケーションをインタプリタ実行してプロファイル情報を採集しているので，アプリケーションはCONDORの制御下に置かれなければならない．つまり，アプリケーションを起動すると，実際にはCONDORが起動されてその管理下でアプリケーションが実行される．このようにCONDORを起動するには，

- カーネルのローダを変更し，アプリケーションではなくCONDORを呼ぶようにする．
- ptraceを使って別プロセスとして起動されたCONDORとアプリケーションを繋ぐ．
- アプリケーションプログラムのコピーを作成し，CONDORを結合する．
- アプリケーションのロード時に，実行開始プロローグとしてCONDORをプリロードする．

などの選択肢がある[3]．

現在はどの方式を選択すべきか検討中である．ただし，元々SMT向けにOSを新たにする必要があるので，CONDORを組み込むためにOSを変更することには問題はないことを付け加えておく．

4.3 アプリケーションの実行法

上述したようにアプリケーションはCONDORの制御下に置かれなければならないので，正確にはアプリケーションとCONDORの二つのスレッドが実行されているわけではなく，アプリケーションをインタプリタ実行しているCONDORと動的最適化を行なっているCONDORの二つのスレッドが実行されることになる．インタプリタ実行を例えば仮想マシンとして実現する[8]と，モデルとしてはシンプルでわかりやすいが，実行のオーバーヘッドが心配である．特に，プロセッサが抽出するILPはユーザーアプリケーションのそれではなく仮想マシンのそれとなってしまい，CONDOR方式が狙っているものと異なってしまう．つまり期待しているCONDOR方式のproxy動作を実現できない．

オーバーヘッドを小さくするために，現在はバイナリ変換に基づいたインタプリタ実行[6),26]を検討している．この方式のインタプリタは，アプリケーションにインタプリタ実行するためのバイナリ変換を施して，変換後の命令をバッファにコピーした後，バッファされたプログラムを実行する．元々この方式はシミュレーションの高速化のために利用されているが，CONDOR方式では必ずしもシミュレーションに必要となる命令の変換は必要ないし，現存するマシン上で実行されるわけでもない．さらに，インタプリタ実行される命令はターゲットマシンの機械命令なので，バイナリ変換せずとも実行できる．したがって，命令をバッファリングしなくてもハードウエアによるサポートが可能である．アプリケーションの実行時に適切なタイミングでCONDORに制御が移動すれば良いので，例えば例外としてCONDORに割り込みをかける方法がある．基本ブロック単位でCONDORはプロファイル情報を更新する必要があるので，分岐命令が実行されると必ず割り込みを発生してCONDORに制御が移動するようにすれば良い．アプリケーションはCONDORの実行(例外ハンドラ相当)中に停止する必要は無く，並行して実行を続けていれば良い．また，分岐命令によって割り込みを発生するかどうかは，制御レジスタによってモードを選択できるようにする．

4.4 動的最適化の開始

割り込みによりCONDORに制御が移ると，まずプロファイル情報の更新を行ない，続いて動的最適化を起動する条件をチェックする．トレース実行状況が設定された閾値を越えていれば最適化を開始する．特筆すべきは，最適化によるオーバーヘッドを隠蔽可能なのでCONDORではこの閾値を小さくできることである．

以上をまとめるとCONDORの実行フローは図4のようになる．まずアプリケーションプログラムが起動されると，OS等のサポートによりCONDORに制御が移動する．このとき，プロファイル管理資源の割り当てなどのセットアップが行なわれる．セットアップの終了後，アプリケーションが実行される．分岐命令が出現すると割り込みが発生し，例外ハンドラとしてCONDORが起動される．CONDORはまずプロファイル情報を更新し，分岐先がターボキャッシュ内にあるかどうかをチェックする．ターボキャッシュ内に無い場合には最適化開始条件をチェックする．最適化の条件が満足されていなければ，CONDORのスレッドは終了する．条件を満足している場合は最適化を開始し，最適化後のバイナリをターボキャッシュに格納する．最適化が完了すればやはりCONDORのスレッドは終了する．

ターボキャッシュにヒットした場合には，最適化条件をチェックすること無くターボキャッシュに制御が移動する．同時に，実行中のオリジナルのバイナリのスレッドは破棄される．ここで，CONDORを起動した分岐命令より先の命令はまだコミットされていないことに注意されたい．CONDORは例外ハンドラ相当なので，CONDOR実行のための命令がコミットされなければ分岐命令より先の命令もコミットできない．したがって，ターボキャッシュヒット時には分岐命令より先の命令はまだコミットされておらず，容易に破棄することが出来る．ターボキャッシュ内に実行すべき命令が無くなると，オリジナルのバイナリに制御が移動する．したがって，オリジナルのバイナリとターボキャッシュ内のバイナリとの間でお互いに制御が移動するときには，コンテキストを受け渡す必要がある．このための機構は検討中である．

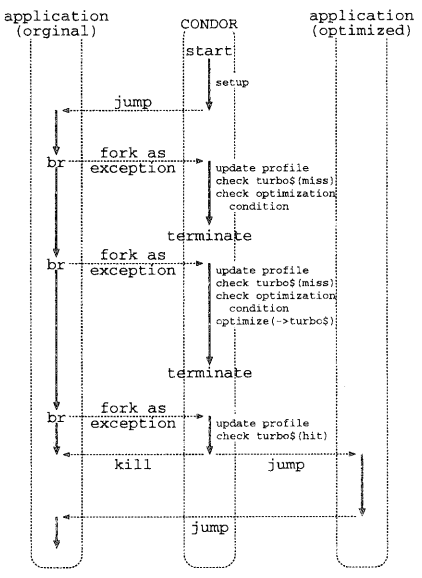

図4 CONDORの実行フロー
Fig. 4 Execution flow of CONDOR

4.5 子プロセスの扱い

アプリケーションが子プロセスを産んだ時の扱いは未検討である．子プロセスの起動時に，アプリケーション起動時と同様にCONDORを起動する方法が必要になると考えられる．

4.6 COSMOSプロセッサの実装

SMTは基本的には高並列なスーパースカラプロセッサなので，ハードウエア複雑度増大のサイクルタイムへのインパクトが最も大きな問題である．この問題を解決するためにはプロセッサを実行要素(PE:processing element)に分割するのが一般的であり，この方法には

- クラスタ化スーパースカラプロセッサ
- オンチップマルチプロセッサ

の選択肢が考えられる．前者はPE間で命令供給エンジンが共有され命令実行エンジンは分散されており，現在のスーパースカラプロセッサとの親和性が高い．後者は独立したプロセッサをPEとして扱うので，FIFOキューに基づく並列分岐実行[1]によるSMTの実現に適している．現在は前者での実現を検討しており，後者での実現は将来の検討課題である．

クラスタ化スーパースカラプロセッサでは，階層化の概念を応用し，ハードウエア規模の小さい中並列なスーパースカラプロセッサを並べて，高並列なスーパースカラプロセッサを実現する．クラスタ化プロセッサでは，クラスタ間でのデータフォワーディングにオーバーヘッドがあるため，命令発行のポリシーが重要である．すなわち，依存関係にある複数の命令ができるだけ同じクラスタに発行されることが好ましい．SMTでは同時に複数の独立なスレッドが流れているため，この発行ポリシーは容易に決定できる．つまり，スレッド識別子に基づいた発行ポリシーが利用可能である．クラスタ化に基づいたハードウエア構成とターボキャッシュを採用することを考慮すると，COSMOSプロセッサの実装にはトレースプロセッサ[22]との親和性が良いと考えられる．トレース形成のハードウエアを取り除き，ソフトウエアで制御可能なターボキャッシュに置き換える．各スレッドのコンテキストは，中並列スーパースカラ相当のPEに分散させる．分岐予測やデータ予測の機構も分散できる可能性がある．

4.7 パフォーマンスカウンタ

最近のマイクロプロセッサには，実行された命令の数などを計測できるパフォーマンスカウンタが備わっており，キャッシュミスの状況なども知ることが出来る．さらに，キャッシュミス時に例外を発生させてプロセッサハードウエアからソフトウエアに対して情報を提供する検討[12]も行なわれている．プログラムの実行時にこのようなハー

ドウエアからソフトウエアへの情報提供が出来ればプロファイル採集のオーバーヘッドを軽減でき，CONDOR方式の効率が向上すると考えられる．またキャッシュミスに限らず，分岐予測の情報やデータ予測の情報はそれらの機構の精度向上に役立つと考えられる．したがって，パフォーマンスカウンタを拡張した上記の機構の検討も必要であろう．

5. 関連研究

本研究はTullsenら[24]の提案しているSMTをベースにしている．SMTは元々同時に複数のアプリケーションを実行することでプロセッサのスループットを改善する目的で提案されていた．そのため，ある特定のアプリケーション一つだけを実行する場合には，それを複数のスレッドに分割するのが困難であると，SMTの効果は期待できない．Tullsenらの提案の後，以下で説明するような，一つのプログラムのパフォーマンスを改善する目的でSMTを応用する提案がなされている．

Wallaceら[25]は，複数パス実行機構をSMT上で実現している．分岐点で開始される複数のパスを，異なるスレッドとしてSMT上で実行する．選択されなかったパス(スレッド)は破棄される．この方式は，複数パスの開始時にレジスタファイルを複製するための巨大なネットワークを必要とする．

Chappelら[5]は，アプリケーション(主スレッド:primary thread)のパフォーマンスを改善するためにマイクロスレッド(microthread)を用いることを提案している．彼らはこの方式をSimultaneous Subordinate Microthreading(SSMT)と呼んでいる．マイクロスレッドは主スレッドと同時に実行され，プリフェッチングや分岐予測精度改善の目的で，主スレッドの実行をサポートする．我々のCONDOR方式は以下の点でSSMTと異なる．(1)SSMTではマイクロスレッドを保持するための特別なマイクロRAMを必要とする．一方，CONDOR方式のためのオプティマイザは他のアプリケーションと同様に主メモリ上に置かれる．(2)SSMTはハードウエア指向の方式であり，主スレッドであるアプリケーションのバイナリコードに対して最適化が施されるわけではない．また，SSMTには予めコンパイル時にプロファイル情報を利用したマイクロスレッド起動命令の挿入操作が必要となる．一方，CONDOR方式では既存のアプリケーションバイナリを最適化し，パフォーマンスの改善を図る．さらに，予め獲得されたプロファイル情報をソースコードのコンパイル時に利用するのではなく，実行時に動的にプロファイル情報を獲得し，それを用いてバイナリコードを最適化する．

Zillesら[27]は，SMT上で主プログラムと同時に例外ハンドラを実行することを提案している．同時実行されるだけでなく，不要な命令破棄操作を取り除くことが可能になり，主プログラムのパフォーマンスを改善できる．しかし，この方式は積極的にアプリケーションのパフォーマンスを改善しようとするものではなく，ソフトウエアTLBなどの極く限られた状況でのみ有効である．特に，ほとんど例外を生じないような場合では，有効性が疑わしい．

バイナリコードの動的な最適化には，ハードウエア指向の方式とソフトウエア指向の方式がある．DIF[19]，トレースキャッシュ[21]，NCB[23]はハードウエア指向方式である．アプリケーションの実行時に動的なトレースを採集してキャッシュに保持し，同時実行可能な命令の数を増加させる．一方，CONDOR方式はソフトウエア指向である．

Dynamo[3]，Conteらの方式[7]，DAISY[8],[9]は，ソフトウエア指向の方式である．Conteらの方式[7]は，VLIWマシン間での互換性を維持するための目的で動的にバイナリを変換するものであり，最適化を行なってはいない．DAISY[8]の目的はRISC命令をVLIW命令に変換することである．一方，CONDOR方式の目的は最適化であり，最適化前後で命令セットの変換は無い．DAISYでは動的に採取されたプロファイル情報を用いて，VLIW命令変換の対象を決定している[9]．この動的なプロファイル採集は，CONDOR方式でも採用されている方式である．DAISYの最大の問題は変換後のVLIW命令のキャッシュミス率である[9]．一方，CONDOR方式では理論的にはキャッシュミス率を改善するような最適化[13],[18]も可能である．Dynamo[3]は，動的に最適化を行なうオプティマイザであり，最適化の前後で命令セットの変更が無い点はCONDOR方式と同じである．また，動的に採取されたプロファイル情報を最適化に用いる点も同様である．DynamoとCONDOR方式との間の相違点は，Dynamoがアプリケーション実行とその最適化を逐次に行なうのに対して，CONDOR方式では同時に行なうことである．そのためDynamoでは，最適化に伴うオーバーヘッドを抑えるために，あるトレースが50回以上実行されなければ最適化を行なわないという閾値のヒューリスティックを設けている．一方，CONDOR方式では機能ユニットの空き状況に応じてこの閾値を変更可能であり，オーバーヘッドを抑えつつもっと積極的な最適化が可能である．

6. まとめ

本稿では，有効な命令を増加する事でILPを改善可能なアーキテクチャCONDOR方式を提案した．CONDOR方式はハードウエアとソフトウエアが協調動作することでプロセッサシステムの性能向上を図る．アプリケーションプログラムとその動的な最適化を行なうオプティマイザをSMTプロセッサ上で同時実行する事で，アプリケーションのパフォーマンスを改善することを狙っている．利用されないで遊んでいる機能ユニット上でオプティマイザを実行するので，動的最適化によって生じるオーバーヘッドを抑えることが可能である．我々は現在CONDOR方式に基づいたCOSMOSプロセッサを検討中である．本研究はまだ緒についたばかりであり，CONDOR方式の実現には解決しなければならない問題が山積みであることを指摘した．いくつかの課題に対しては解決策を提示した．今後，残っている問題や見落としている問題を検討しCONDORを開発しなければならない．さらに，COS-

MOSプロセッサのモデルを作成し，CONDOR方式の効果をシミュレーションにより評価する必要もある．

謝辞 本研究を行なうにあたり，御助言，御討論頂いた九州工業大学マイクロ化総合技術センター田中康一郎助手，鹿児島工専情報工学科堂込一秀助教授に感謝致します．

参考文献

1) 有田五次郎: FIFOキューを同期手段とする並列プログラムについて (I) —待ち無し並列プログラム—, 情報処理学会論文誌, vol.24, no.2 (1983).
2) August,D.I., Connors,D.A., Mahlke,S.A., Sias, J.W., Crozier,K.M., Cheng,B-C., Eaton,P.R., Olaniran,Q.B., and Hwu,W-m.W.: Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture, *Proc. of 25th Int'l Symp. on Computer Architecture* (1998).
3) Bala,V., Duesterwald,E., and Banerjia,S.: Transparent Dynamic Optimization, *Techinical Report HPL-1999-77*, HP Laboratories Cambridge (1999).
4) Burger,D. and Goodman,J.R.: Billion-Transistor Architectures, *IEEE Computer*, vol.30, no.9 (1997).
5) Chappel,R.S., Stark,J., Kim,S.P., Reinhardt,S.K., and Patt,Y.N.: Simultaneous Subordinate Microthreading (SSMT), *Proc. of 26th Int'l Symp. on Computer Architecture* (1999).
6) Cmelik,R.F. and Kepple,D.: Shade: A Fast Instruction-Set Simulator for Execution Profiling, *Technical Report UWCSE93-06-06*, Dept. of Computer Science and Engineering, Univ. of Wasington (1993).
7) Conte, T.M. and Sathaye, S.W.: Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures, *Proc. of 28th Int'l Symp. on Microarchitecture* (1995).
8) Ebcioglu,K. and Altman,E.: DAISY: Dynamic Compilation for 100% Architecture Compatibility, *Proc. of 24th Int'l Symp. on Computer Architecture* (1997).
9) Ebcioglu, K., Altman, E.R., Sathaye, S., and Gschwind,M.: Execution-based Scheduling for VLIW Architectures, *Proc. of Int'l Euro-Par'99 Conf.* (1999).
10) Emer,J.: Simultaneous Multithreading: Multiplying Alpha's Performance, *Proc. of Microprocessor Forum'99 Conference* (1999).
11) Grunwald, D., Klauser, A., Manne, S., and Pleszkun,A.: Confidence Estimation for Speculation Control, *Proc. of 25th Int'l Symp. on Computer Architecture* (1998).
12) Horowitz,M., Martonosi,M., Mowry,T.C., and Smith,M.D.: Informing Memory Operations: Memory Performance Feedback Mechanisms and thier Applications, *ACM Transactions on Computer Systems*, vol.16, no.2, (1998).
13) Hwu,W.W. and Chang,P.P.: Achieving High Instruction Cache Performance with an Optimization Compiler, *Proc. of 16th Int'l Symp. on Computer Architecture* (1989).
14) Krishnan, V. and Torrellas, J.: A Chip-Multiprocessor Architecture with Speculative Multithreading, *IEEE Trans. Comput.*, vol.48, no.9 (1999).
15) Lai,A-C. and Falsafi,B.: Memory Sharing Predictor: The Key to a Speculative Coherent DSM, *Proc. of 26th Int'l Symp. on Computer Architecture* (1999).
16) Lipasti,M.H., Wilkerson,C.B., and Shen,J.P.: Value Locality and Load Value Prediction, *Proc. of 7th Int'l Conf. on Architectural Support for Programming Languages and Operatings Systems* (1996).
17) Mahlke,S.A., Hank,R.E., Bringmann,R.A., Gyllenhaal,J.C., Gallagher,D.M., and Hwu,W-m.W.: Characterizing the Impact of Predicated Execution on Branch Prediction, *Proc. of 27th Int'l Symp. on Microarchitecture* (1994).
18) McFarling,S.: Program Optimization for Instruction Caches, *Proc. of Int'l Conf. on Architectural Support for Programming Languages and Operating Systems III* (1989).
19) Nair,R. and Hopkins,M.E.: Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups, *Proc. of 24th Int'l Symp. on Computer Architecture* (1997).
20) Patt,Y.N., Melvin,S.W., Hwu,W., and Shebanow,M.: Critical Issues Regarding HPS, A High Performance Microarchitecture, *Proc. of 18th Workshop on Microprogramming* (1985).
21) Rotenberg,E., Bennet,S., and Smith,J.: Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching, *Proc. of 29th Int'l Symp. on Microarchitecture* (1996).
22) Rotenberg,E., Jacobson,Q., Sazeidas,Y., and Smith,J.: Trace Processors, *Proc. of 30th Int'l Symp. on Microarchitecture* (1997).
23) Sato,T.: A Microprocessor Architecture Utilizing Histories of Dynamic Sequences Saved in Distributed Memories, *IEICE Trans. Electron.*, vol.E81-C, no.9, (1998).
24) Tullsen,D.M., Eggers,S.J., and Levy,H.M.: Simultaneous Multithreading: Maximizing On-Chip Parallelism, *Proc. of 22nd Int'l Symp. on Computer Architecture* (1995).
25) Wallace,S., Calder,B., and Tullsen,D.M.: Threaded Multiple Path Execution, *Proc. of 25th Int'l Symp. on Computer Architecture* (1998).
26) Witchel,E. and Rosenblum,M.: Embra: Fast and Flexible Machine Simulation, *Proc. of SIGMETRICS* (1996).
27) Zilles,C.B., Emer,J.S., and Sohi,G.S.: The Use of Multithreading for Exception Handling, *Proc. of 32nd Int'l Symp. on Microarchitecture* (1999).