システムＬＳＩ設計技術　104－6
(2002. 1. 23)

COSMOSプロセッサにおける最適化の一実施例

森田　広† 山本 季之†† 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
†† 株式会社アルファシステムズ
††† 九州工業大学 マイクロ化総合技術センター

E-mail: †mori@mickey.ai.kyutech.ac.jp

あらまし COSMOSプロセッサは空間多重マルチスレッド (SMT: Simultaneous Multi-Threading) を用いて構成されており、命令レベルの並列性(ILP: Instruction Level Parallelism) とスレッドレベルの並列性(TLP: Thread Level Parallelism) に着目している。ILPが低い場合、複数のスレッドを同時に実行することでTLPを活用し、余った機能ユニットを用いてアプリケーションの実行時に動的最適化を行うオプティマイザを同時実行することで利用効率を向上させようと考えている。本稿では動的最適化を行った際に期待される性能向上について報告する。

キーワード 空間多重マルチスレッド, 命令レベル並列処理, 動的最適化

The KIT COSMOS Processor: A Case Study on Optimizing Hot Spots

Kou MORITA†, Toshiyuki YAMAMOTO††, Toshinori SATO†,†††, and Itsujiro ARITA†

† Department of Artificial Intelligence, Kyushu Institute of Technology
†† Alpha Systems Inc.
††† Center for Microelectronic Systems, Kyushu Institute of Technology

E-mail: †mori@mickey.ai.kyutech.ac.jp

Abstract This paper proposes a cooperation between dynamic optimization technique and simultaneous multi-threading (SMT) architecture. Recent studies on dynamic optimization reveal that it has large potential for improving processor performance. This is because every program running on the processor can be optimized using profile information gathered on-the-fly, which is unavailable to compilers. On the other hand, SMT processors are emerging in the near future. They maintain several contexts simultaneously and improve efficiency of their hardware resources. Thus, the secondly threads exploit idle hardware resources which the primary thread can not use. The dynamic optimization technique also benefits from the SMT, since any overheads caused by the optimization are mitigated. This paper introduces the combination of the dynamic optimization and the SMT, and shows preliminary evaluation results.

Key words simultaneous multi-threading, instruction level parallelism, dynamic optimization

1. はじめに

命令レベルの並列性(ILP:Instruction Level Parallelism)を獲得するために、現在のマイクロプロセッサは投機実行や複数パス実行などを行っている。これらの方式の欠点のひとつに、大きなILPの抽出に本来は不要な命令の実行を伴うことがあげられる。つまり、これらの方式は演算器が十分余っているという条件の下で意味があると言える。実際、現在迄のところこれらの方式はプロセッサ性能に大きな貢献を果してきた。しかし、命令の発行幅が広がるにつれて、深い投機実行は投入されるハードウエア資源の量にはみあわなくなりつつあると言われている。なぜなら、演算器に投入された命令の殆どが有効でなくなりつつあるからである。

このような状況を考慮した結果、われわれは有効な命令を引出す新しい方式を考案した。それは、マルチスレッド実行を利用して余っている演算器を有効利用するものである。そこでは、ユーザプログラムと最適化プログラムが同時に実行される。最適化プログラムがユーザプログラムを動的に最適化しパフォーマンスを向上させる。われわれはこの方式をCONDOR(CONcurrent Dynamic OptimizeR)と呼んでいる[2]。この方式は複数のスレッドトータルのILPを向上することを目指しているのではなく、あくまでもユーザスレッド(プログラム)だけのILP向上を目的としている。このようにCONDORの実行モデルは空間多重マルチスレッド(SMT:Simultaneous Multi-Threading)[27]と似ている。加えて、製品化は断念したもののCOMPAQは21464でSMTを採用している[14]し、IntelやSunもSMTの導入を公表している[17],[20]。そこで、われわれはCONDORを応用したマイクロプロセッサをSMTプロセッサとして実現することを考え、それをCOSMOSプロセッサと名付けている。

2. COSMOSプロセッサ

上述したようにCOSMOSはSMTに基いている。SMTは大規模スーパースカラプロセッサであるので、ハードウエアの複雑度が動作速度に与える影響を考慮する必要がある。この問題に対処するために、われわれはCOSMOSをクラスタ化マイクロプロセッサとして実現することを考えている。図1はCOSMOSのブロック図である。ひとつの命令供給部と二つの命令実行部(クラスタ)から構成されている。ただし、クラスタの数は二つである必要はない。詳細は文献[26]を参照されたい。

本稿では、CONDORにおける重要な構成要素である、ターボキャッシュ(Turbo$)に注目する。ターボキャッシュはオンチップSRAMであり、ユーザには不可視なメモリ領域にマッピングされている。最適化のための作業領域として利用するほか、最適化後のバイナリを保持するためのメモリである。つまり、ハードウエアで実現されるトレースキャッシュをソフトウエアで実現されるターボキャッシュで置き換えることができる。加えて、ユーザスレッドとオプティマイザスレッドの間での干渉によるキャッシュミスを削減できる。

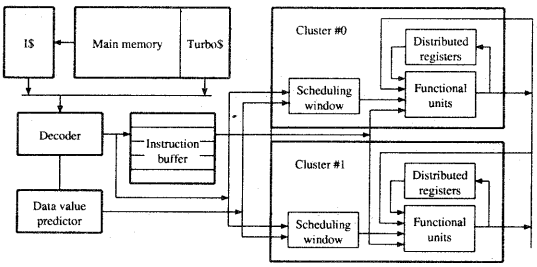

図1 COSMOSプロセッサ

3. 実行時動的最適化

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[6]。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DLL(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、両者はお互いに補完し合う関係にある。

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、ユーザプログラムのパフォーマンスへの貢献は大きい[4]。プロファイルを収集している間は、ユーザプログラム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。

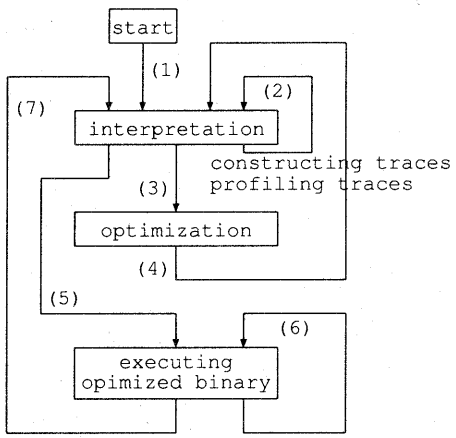

図2　実行時動的最適化

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) ユーザプログラム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、オプティマイザはユーザプログラムの実行を停止し、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、オプティマイザはユーザプログラムの実行を再開する。(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。

4. CONDOR 方式

本節でCONDOR を説明する。CONDOR はソフトウエアにより実現された履歴駆動プロセッサアーキテクチャ[25] である。CONDOR は実行時動的最適化とマルチスレッド機構を組み合わせ、ユーザスレッドとオプティマイザスレッドの同時実行を行うものである。

4.1 概　　要

CONDOR においては、実行時に利用されないで余っている演算器を用いて、動的最適化を行なう。動的最適化の結果、ユーザプログラムのILP が向上する。つまり、利用されていない演算器に、ユーザプログラムとは別の有効な命令を投入して、ユーザプログラムのパフォーマンスを向上させるわけである。ユーザプログラムに並列性が無ければ、十分演算器は余っているので、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。また、デスクトップ環境では利用出来るスレッドレベル(TLP:Thread Level Parallelism) の並列性は1.5と小さいことが知られており[15]、SMT プロセッサ上でCONDOR オプティマイザを実行するためのコンテキストには余裕があると言える。

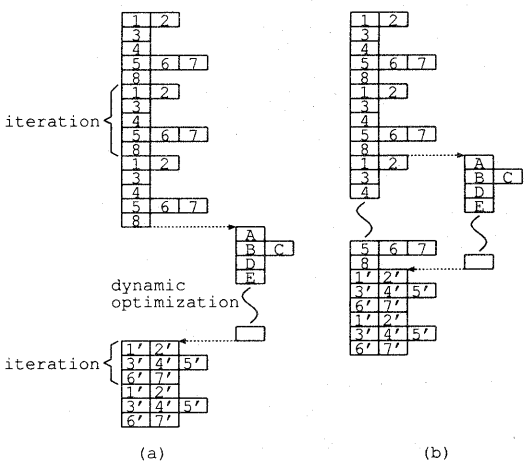

図3　CONDOR 方式

図3に、従来の動的最適化方式とCONDOR の違いを示す。図3(a) が従来の方式であり (b) がCONDOR 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... がユーザプログラムに属し、命令A,B,C... がオプティマイザに属す。ユーザプログラムは動的に最適化され、命令1',2'3'... となる。

(a) で説明されている従来の動的最適化では、あるトレース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。最適化の間はユーザ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、最適化後のユーザプログラムが再開される。最適化に必要なサイクルは、ユーザプログラム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では、動的最適化の実行中の間もユーザプログラムは実行されている。オプティマイザは遊んでいる演算器を利用して最適化を行なう。最適化が完了すると、ユーザプログラムの実行は最適化後の命令に切り替わる。すなわち、CONDOR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。複数回実行されるトレース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トレース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は従来の方式と比較して、最適化に要するオーバーヘッドを非常に小さくできる。

4.2 プロファイル情報の収集

上述したようにCONDORは動的に採取されたプロファイル情報を利用している。従来の動的最適化方式では、起動時にオプティマイザがユーザプログラムを制御下に置かれなければならない. つまり、ユーザプログラムの実行はオプティマイザによるインタプリタ実行に置き換わる。これを実現するには、

- カーネルのローダを変更する。
- ptraceを使ってユーザプログラムを繋ぐ。
- ユーザプログラムのコピーを作成し、テキストセグメントを拡張する。
- 専用のcrt0を使用する。

などの選択肢がある[6]。

対照的にCONDORではインタプリタ実行はされない。代わりに、従属補助スレッドがプロファイル情報を収集する。プロファイル対象の命令が発行されると、同時に従属補助スレッドが起動される。この従属補助スレッドはSMTプロセッサ上で実行され、対応する命令のプロファイル情報を収集する。プロファイル情報はターボキャッシュに保持される。従属補助スレッドの起動を、ハードウエアが透過的に行うべきか、あるいはコンパイラが静的にfork命令を挿入すべきかの判断は検討中であり将来の課題である。前者の場合はCONDORオプティマイザはユーザプログラムとは独立したプログラムあるいはライブラリとして実現され、後者の場合はCONDORオプティマイザはユーザプログラムの一部となる。

パフォーマンスカウンタを利用して動的最適化をサポートすることも可能である。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を知ることが出来る。例えば、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[16]も行なわれている。ユーザ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の効率が向上すると考えられる。パフォーマンスカウンタを利用したプロファイル収集についても現在検討中であり、将来の課題である。

4.3 CONDORにおける最適化

CONDORスレッドは起動されると、プロファイル情報を更新し、動的最適化を開始すべきかどうかを判断するために閾値との比較を行う。上述したように、この閾値は十分小さくすることができるので、動的最適化を開始に要するオーバヘッドは無視できる。

動的最適化は図4の様に実施される。図は分岐命令のプロファイルを採取している例である。各分岐命令が発行されると、例外ハンドラの形式で従属補助スレッドが起動され[30]、閾値にしたがって最適化が開始される。上述したように、最適化後のバイナリはターボキャッシュに保持される。

従属補助スレッドが起動された時、もしすでに最適化後バイナリがターボキャッシュ内にあれば、制御がターボキャッシュに移動する。同時に、元のバイナリの実行は停止され投機的状態は破棄される。従属補助スレッドが完了しない限り、該分岐命令に後続する命令はリタイアしないことに注意されたい。ターボキャッシュ内の命令を実行中に制御がターボキャッシュの外に移動した場合には、元の最適化前のバイナリ実行が再開される。最適化前後のバイナリ間でのコンテキストの交換は、SMTプロセッサの持つ同期機構[28]を用いて容易に実現できる。

4.4 CONDORの適用領域

本節でCONDORの適用例を紹介する。CONDOR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。CONDORを用いると、このような基本的な最適化に加えて、プロ

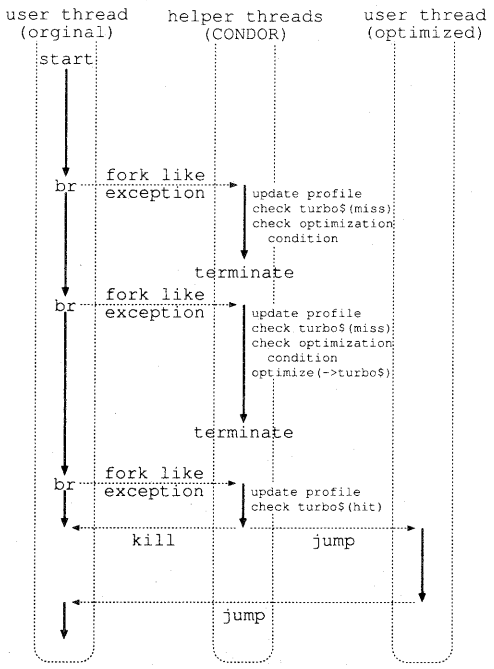

図4　CONDOR の実行フロー

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[29]: 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: 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: CONDOR はデータ投機実行にも有効である。遺物のような古いバイナリ上での値予測の効果は、現代のコンパイラ技術を用いた最適化に遠く及ばないことが知られている[3]。つまり、値予測と CONDOR との協調による最適化を行うことには意味がある。
- 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 は複数パス実行の効果を改善できる。CONDOR 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プロファイル情報を用いた精度の高い分割を、データ投機に基づいた投機的スレッド実行と組み合わせて実現出来る。これは、静的にオフラインでスレッド分割する方法[1] よりも効果が高いと期待できる。

上述した全ての最適化が常にユーザプログラムに適用されるわけではない。例えばコンパイラによる従属補助スレッド生成を採用した場合を考えると、コンパイラが有効だと考える動的最適化だけが選択され、ユーザプログラムに従属補助スレッドとして挿入されることになる。

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ユーザプログラムと最適化が施されるユーザプログラムが同一なので、非常に正確なプロファイル情報を利用できる。

5. シミュレーションによる評価

本節でシミュレーションに基く初期評価結果を紹介する。4.4 節で述べたように、分岐命令の分岐先の偏りは大きい。そのような分岐命令をジャンプ命令で置き換える、あるいは取り除くことを考える。この結果、容量に制限のある分岐予測表の利用効率改善や、基本ブロックのサイズ増大が期待できる。これらはILP の向上に大きく貢献できる。この最適化を、SPEC2000 ベンチマークから選んだ `175.vpr` と `300.twolf` に施した。SimpleScalar ツールセットを用い、その命令セットはAlpha 命令セットである。プロセッサの構成は表1 に示すとおりである。まず、図5 と図6 に示すようなトレースの採取を行い、ホット

表1　プロセッサ構成

フェッチ幅	4 命令
分岐予測	512 セット, 4 ウエイセットアソシアティブ BTB, 2048 エントリ bimodal 予測器, コミット時に更新, 8 エントリリターンアドレススタック, 3 サイクルミスペナルティ
命令ウインドウ	16 エントリ命令キュー, 8 エントリロードストアキュー
発火幅	4 命令
コミット幅	4 命令
演算器	4 iALU's, 1 iMUL/DIV, 2 Ld/St's, 4 fALU's, 1 fMUL/DIV
レイテンシ(全体/間隔)	iALU 1/1, iMUL 3/1, iDIV 20/19, Ld/St 2/1, fADD 2/1, fMUL 4/1, fDIV 12/12
レジスタファイル	32 本の 32 ビット整数レジスタ, 32 本の 32 ビット浮動小数点レジスタ
命令キャッシュ	16K, ダイレクトマップ, 32 バイトブロック, 6 サイクルペナルティ
データキャッシュ	16K, 4 ウエイセットアソシアティブ, 32 バイトブロック, 2 ポート, ライトバック, ノンブロッキング, 6 サイクルペナルティ
2 次キャッシュ	共用, 256K, 4 ウエイセットアソシアティブ, 64 バイトブロック, 48 サイクルペナルティ

スポットを特定した。図において、水平方向は実行された命令数を、垂直方向は命令アドレスを表している。続いて、このプロファイル情報を用いて、人手により最適化を行った。`175.vpr` においては43 命令、`300.twolf` においては146 命令の分岐が削除の対象として選択された。人手による最適化を行ったため、動的最適化に利用されるハードウエア資源には制約がないという条件下での評価となっていることに注意願いたい。最後に、表1 のアウトオブオーダー実行を行うスーパースカラプロセッサで性能を測定した。結果を表2 にまとめる。`175.vpr` においては4.9M 命令、`300.twolf` においては1.7M 命令の分岐が削除可能で、それぞれのIPC は6.87% および8.41%向上していることがわかる。以上により、CONDOR の有効性が確認できた。

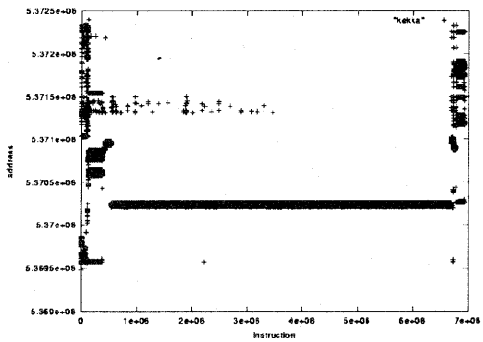

図5　`175.vpr` のトレース採取結果

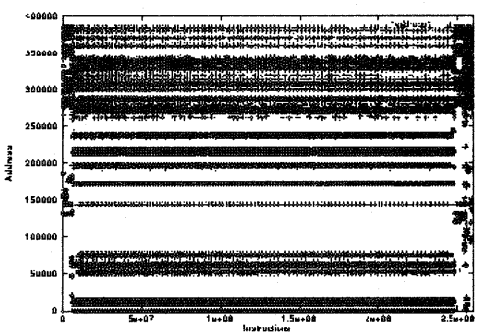

図6　`300.twolf` のトレース採取結果

	# of branches	# of cycles	ILP
175.vpr			
最適化前	82,831,324	514,250,281	1.35
最適化後	77,931,675	481,230,641	1.45
変化	-5.92%	-6.42%	+6.87%
300.twolf			
最適化前	32,975,978	192,312,206	1.35
最適化後	31,269,524	177,390,682	1.46
変化	-5.17%	-7.76%	+8.41%

表2　シミュレーション結果

6. 関連研究

Dynamo[6]、DAISY[12]、CodeMorphing[19] は、ソフトウエア指向の方式である。DAISY の目的はRISC 命令をVLIW 命令に変換することである。同様にCodeMorphing はx86 命令をVLIW 命令に変換する。一方、CONDOR 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 とCodeMorphing 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[13],[19]。この動的なプロファイル採集はCONDOR 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[13]。一方、CONDOR ではVLIW 命令への変換は行わないため、同様の問題は生じない。

Dynamo[6] 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 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 との間の相違点は、Dynamo がユーザプログラム実行とその最適化を逐次に行なうのに対して、CONDOR 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 では演算器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積極的な最適化が可能である。最後に、CONDOR ではSMT によるサポートを期待できるので、各スレッドの管理が容易になる。

Merten ら[22],[23] はハードウエアの支援の元に採取されたプロファイルを利用して動的な最適化を検討している。しかし、この場合もSMT を利用して最適化のペナルティを削減することは検討されていない。

CONDOR は Tullsen ら[27] 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ユーザプログラム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、ある特定のユーザプログラムひと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、ひと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。

Chappel ら[9] は、ユーザプログラム (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。CONDOR 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 オプティマイザはユーザプログラムと同様にメイン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、主スレッドであるユーザプログラム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 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。主スレッドを補助する第二のスレッドを利用する同様の方式には多くの報告がある[5],[7],[10],[11],[21],[24],[31]。

Keckler ら[18] と Zilles ら[30] は、マルチスレッド機構を例外ハンドラ実行の効率改善の目的で利用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、例外は稀にしか生じないものであるから、マルチスレッド機構の利用としては消極的であると言える。

7. 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 を提案した。CONDOR 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。ユーザ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、ユーザプログラム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演算器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。

謝　辞

本研究の一部は、科学研究費補助金奨励研究(A)(課題番号12780273) および基盤研究B(2) 展開(課題番号13558030)、福岡県産業・科学技術振興財団 テーマ探索・シーズ発掘事業(課題番号H12-1) の援助によるものです。

文　献

[1] 小野喬史, 大津金光, 横田隆史, 馬場敬信, "バイナリレベルにおけるマルチスレッド化コード生成手法とその初期評価," 情処研報 2001-ARC-144-32, 2001.

[2] 佐藤寿倫, 有田五次郎, "KIT COSMOS プロセッサ: 背景と着想," 信学技報 CPSY99-115, 2000.

[3] 濱野彰彦, 杉谷樹一, 佐藤寿倫, 有田五次郎, "コンパイラ最適化レベルのデータ投機実行に与える影響," 情処研報 2001-ARC-144-22, 2001.

[4] 山本季之, 佐藤寿倫, 有田五次郎, "COSMOS プロセッサにおける最適化の有効性," 信学技報 CPSY2000-61, 2000.

[5] M.Annavaram, J.M.Patel, and E.S.Davidson, "Data prefetching by dependence graph precomputation," 28th International Symposium on Computer Architecture, 2001.

[6] V.Bala, E.Duesterwald, and S.Banerjia, "Transparent dynamic optimization," Technical Report HPL-1999-77, HP Laboratory, 1999.

[7] R.Balasubramonian,S.Dwarkadas, D.H.Albonesi, "Dynamically allocating processor resources between nearby and distant ILP," 28th International Symposium on Computer Architecture, 2001.
[8] D.Burger and T.M.Austin: "The SimpleScalar tool set, version 2.0," ACM SIGARCH Computer Architecture News, vol.25, no.3, 1997.
[9] R.S. Chappel, J. Stark, S.P. Kim, S.K. Reinhardt, and Y.N. Patt, "Simultaneous subordinate microthreading (SSMT)," 26th International Symposium on Computer Architecture, 1999.
[10] J.D.Collins, H.Wang, D.M.Tullsen, C.Hughes, Y-F.Lee, D.Lavery, and J.P.Shen, "Speculative precomputation: large-range prefetching of delinquent loads," 28th International Symposium on Computer Architecture, 2001.
[11] M.Dubois and Y.H.Song, "Assisted execution," Technical Report CENG-98-25, University of Southern California, 1998.
[12] K.Ebcioglu and E.Altman, "DAISY: dynamic compilation for 100% architecture compatibility," 24th International Symposium on Computer Architecture, 1997.
[13] K. Ebcioglu, E.R. Altman, S. Sathaye, and M. Gschwind, "Execution-based scheduling for VLIW architectures," 5th International Euro-Par Conference, 1999.
[14] J.Emer, "Simultaneous multithreading: multiplying Alpha's performance," Microprocessor Forum, 1999.
[15] K.Flautner, R.Uhlig, S.Reinhardt, and T.Mudge, "Thread-level parallelism and interactive performance of desktop applications," 9th International Conference on Architectural Support for Programming Languages and Operating Systems, 2000.
[16] M. Horowitz, M. Martonosi, T.C. Mowry, and M.D. Smith, "Informing memory operations: memory performance feedback mechanisms and their applications," ACM Transactions on Computer Systems, vol.16, no.2, 1998.
[17] Intel Corporation, "Introduction to Hyper-Threading technology", White paper, September 2001.
[18] S.W. Keckler, W.J. Dally, A. Chang, W.S. Lee, and S. Chatterjee, "Concurrent event handling through multithreading," IEEE Transactions on Computers, vol.48, no.9, 1999.
[19] A.Klaiber, "The technology behind Crusoe processors," White Paper, Transmeta Corp., 2000.
[20] K.Krewell, "900MHz UltraSPARC III ready to ship," Microprocessor Report, vol.15, arc.9, 2001.
[21] C-K.Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," 28th International Symposium on Computer Architecture, 2001.
[22] M.C. Merten, A.R. Trick, C.N. George, J.C. Gyllenhaal, and W-m.W. Hwu, "A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization," 26th International Symposium on Computer Architecture, 1999.
[23] M.C. Merten, A.R. Trick, E.M. Nystrom, R.D. Barnes, and W-m.W. Hwu, "A hardware mechanism for dynamic extraction and relayout of program hot spots," 27th International Symposium on Computer Architecture, 2000.
[24] A.Roth and G.S.Sohi, "Speculative data-driven multi-threading," 7th International Symposium on High Performance Computer Architecture, 2001.
[25] T.Sato, "History directed processor architecture," Doctoral thesis, Kyoto University, 1998.
[26] T.Sato, T.Yamamoto, and I.Arita, "The KIT COSMOS processor: a low-complexity superscalar processor," International Journal of Computer & Information Science, vol.2, no.4, 2001.
[27] D.M.Tullsen, S.J.Eggers, and H.M.Levy, "Simultaneous multithreading: maximizing on-chip parallelism," 22nd International Symposium on Computer Architecture, 1995.
[28] D.M.Tullsen, J.L.Lo, S.J.Eggers, and H.M.Levy, "Supporting fine-grained synchronization on a simultaneous multithreading processor," 5th International Symposium on High-Performance Computer Architecture, 1999.
[29] T.Yamamoto, T.Sato, and I.Arita, "The KIT COSMOS processor: eliminating ineffectual branch instructions via concurrent dynamic optimization," 4th International COOL-Chips Conference, 2001.
[30] C.B.Zilles, J.S.Emer, and G.S.Sohi, "The use of multithreading for exception handling," 32nd International Symposium on Microarchitecture, 1999.
[31] C.B.Zilles and G.S.Sohi, "Execution-based prediction using speculative slices," 28th International Symposium on Computer Architecture, 2001.