

並列度の異なる VLIW 計算機ファミリでの 命令コード共有方式

鈴木 貢

渡邊 坦

gian@cs.uec.ac.jp tan@cs.uec.ac.jp

電気通信大学 情報工学科

〒182 東京都調布市調布ヶ丘1-5-1

本論文では、命令レベルの並列性を抽象化した「プライムブロック」という概念を提起し、それを用いて、同じアーキテクチャーを共有しながら、実行ユニットの構成が異なる VLIW プロセッサ間で同一のバイナリ (オブジェクト) コードを共有するための2つの方法を提案する。ひとつは、プログラムを主記憶にロードする際に命令のスケジューリングを行なう方法である。もうひとつは、プロセッサが実行時にスケジューリングを行なう方法である。両者とも、コンパイラによってバイナリコードないし命令に付加された情報によって、スケジューリングする。これらにより、従来は機種毎に必要であったバイナリコードを、同じファミリについては一本化することが可能となる。さらに、プロセッサの並列度を変えながら、提案する方法の有効性を検討する。

Binary code sharing among VLIW processors with different parallelism

Mitsugu Suzuki

Tan Watanabe

gian@cs.uec.ac.jp

tan@cs.uec.ac.jp

Department of Computer Science,

The University of Electro-Communications

1-5-1, Chofugaoka, Chofu-si, Tokyo, 182 Japan

A concept "*prime block*", representing an abstraction of instruction level parallelism, is presented, and methods sharing the same binary (object) code among VLIW processors with different construction of parallel execution are proposed. In one method, instructions are scheduled at loading time. In another method, they are scheduled at execution time by the processor. For both methods, compilers add information to the binary code to help scheduling. Presented methods enable the VLIW processors in a family to share the same binary code, even if the number of concurrently executable units differ with each other.

1 はじめに

プロセッサの構成技術とコンパイラの最適化技術の進歩等により、命令レベル並列実行型 (Instruction Level Parallelism, ILP) [2] プロセッサが広く用いられるようになった。その一種である VLIW (Very Long Instruction Word) プロセッサが、実用レベルの域にある [4]。VLIW プロセッサは、ILP の具体化として広く用いられている Super scalar プロセッサに比べて、命令の並列実行用の複雑な制御回路の大部分を省略できるという利点を有している。

しかし VLIW プロセッサの場合は、同じプロセッサファミリ (同一の命令語とユーザレベルレジスタ構成をもつプロセッサたち) でも、実行ユニットの構成 (発行命令数や組合せ可能な命令の種類) が異なると、プログラムの再コンパイルが必要となる。一般の商用アプリケーションプログラムでは、プログラムはソースコードの形でユーザに供給されないで、同じプロセッサファミリのより性能の高い計算機の購入とともに、蓄積してきたプログラムのバイナリコードを破棄しなければならない。またプログラムの供給者の立場からみると、同じプロセッサファミリの計算機を対象にしながらも、プロセッサの種類だけバイナリコードを用意しなければならない。

VLIW 計算機が汎用的になるには、「同じプロセッサファミリ内では、異なるプロセッサ間でも同一のバイナリコードを共有する」という問題を解決しなければならないといえる。本論文では、「性能が充分高く、与えられたプログラムに対して命令レベルの並列実行性を最大限に抽出可能であるコンパイラが用意されている」という仮定の下に、それが生成した同じバイナリコードを、同じプロセッサファミリの異なる構成のプロセッサで、それぞれのプロセッサでの並列度に合わせて実行することを可能にする 2 つの方式を提案する。

一つの方式は、プログラムを主記憶にロードする際に、命令語を VLIW 命令語に再編しな

がらメモリに固定し、それを実行する方式である。この方式は、プログラムの実行コードを RAM 上に置く計算機に適した方式であり、次に挙げる方式に比べて命令スケジューラのハードウェアが不要である。この方式を「LS (Load-time Scheduler) 法」と呼ぶことにする。

もう一つの方式は、プロセッサのハードウェアが実行時にプロセッサの並列度に応じて命令の再編を行なう方式である。この方式は、ROM に固定されたバイナリコードを直接プロセッサが実行するような場合に適した方式である。この方式を「ES (Execution-time Scheduler) 法」と呼ぶことにする。

提案する方式は、コンパイラで並列実行可能な命令を最大限に抽出し、それを実行するプロセッサの並列度で切り分けながら実行していくという方針を採っている。本論文ではコンパイラが抽出する並列度と実行するプロセッサの並列度を変えながら、プログラムの実行速度を検討することで、提案する方式が有効であることを示す。

2 方式説明

本論文で提案している方式ではコンパイラは、大域的なフロー解析、ソフトウェアパイプラインニング [3]、同型命令の発見 [7]、トレーススケジューリング [1] 等の技法を用いて、対象プロセッサファミリでの最大限¹の命令レベル並列度を抽出すると仮定する。

例えば、Fig.1(a) のようなプログラム (ベクトル a と b の内積) に対し、ループの展開をプロセッサファミリのレジスタ数が許す限り施し、(b) のようなプログラムに変換したものと等価なコード (Fig.2) を出力する。通常のアセンブラでは 1 行に 1 命令を記述するが、ここでは複数の命令が並列に実行可能であることを明示するために、それらを 1 行にまとめて記述する。

¹例えばループを展開する場合は、レジスタの個数等のアーキテクチャの制約で最適展開数が決まるだろう。

```

      (a)
float a[], b[], ip = 0;

for (i = 0; i < 1000; i++)
    ip = ip + a[i] * b[i];

      (b)
for (i = 0, i < 1000; i+=4) {
    ip0 += a[i + 0] * b[i + 0]
    ip1 += a[i + 1] * b[i + 1]
    ip2 += a[i + 2] * b[i + 2]
    ip3 += a[i + 3] * b[i + 3];
}
ip = ip0 + ip1 + ip2 + ip3;

```

Fig. 1: A code for inner product calculation

この例は Sparc 用のコードである。「|」が命令の区切りである。

この例のような並列実行可能な命令のかたまりをプライムブロックと呼ぶことにする。プライムブロックの性質は次の通りである。

1. プライムブロック内の命令は任意の順序で直列的に実行してもよい。(分岐命令を除く)
2. 分岐命令があるとすれば、1 プライムブロックに1つに限られ、必ずそのブロックの最後に置かれる。
3. プライムブロックへの飛び込み点があるとすれば、先頭に限られる。

第2,3項から、プライムブロックが通常のベシックブロックに含まれることに注意されたい。また、第1項が意味するのは、同じプライムブロック内の命令の間には依存関係があつてはならないということである。例えば、同じブロック内のある命令がレジスタ%15に結果を格納し、別の命令が%15の値を演算に用いるようなコードは許されない。なぜなら、それらの命令が並列に実行されるか、あるいはどんな順番で直列に実行されるかは不確定なので、実行の結果が一意に定まらなくなるからである。ここで提案する方式がコンパイラに期待しているのは、このプライムブロックを大きくすることである。ただし本論文では、コンパイラのこの事

項や関連技術については言及しない。

さて、提案する2つの方式の技術上の骨子は、プロセッサやローダにプライムブロックを認識させるための付加情報を、コンパイラがバイナリコードに付加し、LS方式の場合はプログラムのロード時に、ES方式の場合は実行時に、それぞれその情報を元にして対象プロセッサで並列実行性を引き出すことである。この付加情報は、論理的にはFig.3の左の図のように、1命令語当たり1ビットのAフィールドとして表現できる。コンパイラは、隣り合うプライムブロックのAフィールドの値を互い違いにしてプライムブロックを表現する。同じプライムブロック内の命令語のAフィールドは、同じ値となる。

3 ロード時並列化

ロード時に命令のスケジューリングを行なうLS方式では、AフィールドをバイナリコードのAフィールド用のエリアに固めて配置する。Aフィールドのために必要な記憶容量は、命令語が32ビットの場合は全命令語の約3%に過ぎない。ローダはVLIW命令語の構成の情報(一度に発行可能な命令の種別や個数とその並べ方)を持っていて、それに基づいてNOP命令を挿入したり飛び先アドレスの修正を行ないながら、プライムブロックを分割し、主記憶に対象プロセッサのVLIW命令語の形で展開する。

ローダが対象VLIW命令語を主記憶上に展開していく手順は、次の通りである。

1. ひとつのプライムブロックの命令語群を取って来る。
2. もし、そのブロックが飛び先であるなら、「飛び先表」に現在のVLIW命令のアドレスを記録する。
3. その中から、対象プロセッサで実行可能な命令群を取り出し、1つのVLIW命令の該当フィールドに埋め込み、それを主記憶に

```

mov 0,%o4 | sethi %hi(999),%o7 | sethi %hi(a),%o3 | sethi %hi(b),%g2
or %o7,%lo(999),%o7 | or %o3,%lo(a),%o3 | or %g2,%lo(b),%g2
.LL17:
                                ld [%o3],%f4 | ld [%g2],%f5
                                fmul %f4,%f5,%f4 | ld [4+%o3],%f6 | ld [4+%g2],%f7
fadds %f0,%f4,%f0 | fmul %f6,%f7,%f6 | ld [8+%o3],%f5 | ld [8+%g2],%f8
fadds %f1,%f6,%f1 | fmul %f5,%f8,%f5 | ld [12+%o3],%f7 | ld [12+%g2],%f9
fadds %f2,%f5,%f2 | fmul %f7,%f9,%f7 | add %o4,4,%o4 | add %o3,16,%o3
fadds %f3,%f7,%f3 | add %g2,16,%g2 | cmp %o4,%o7
ble .LL17
fadds %f0,%f1,%f0 | fadds %f2,%f3,%f2
fadds %f0,%f2,%f0

```

Fig. 2: An optimized assembly code for ILP processors

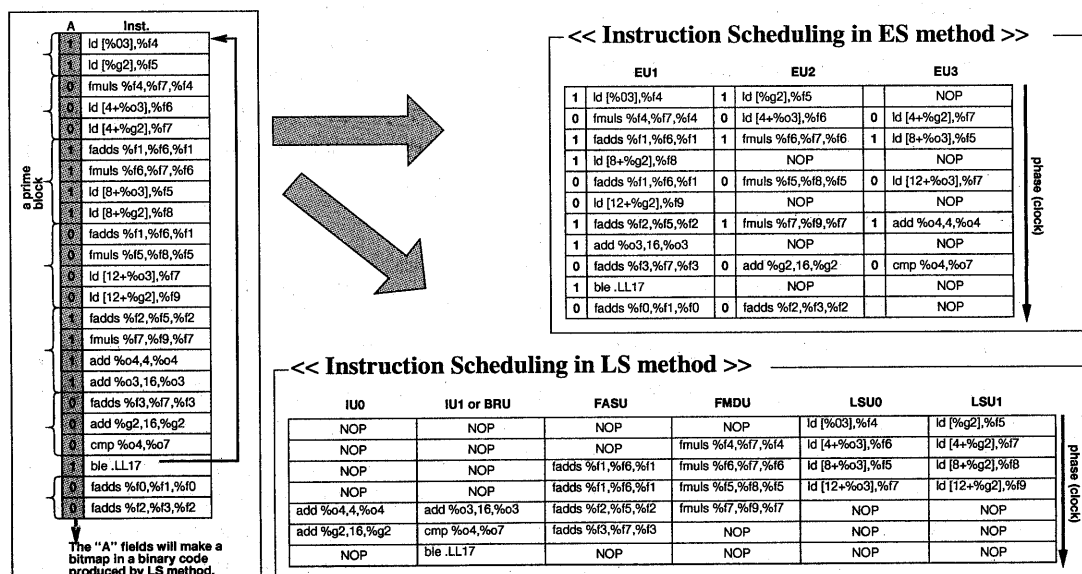


Fig. 3: Instruction arrangement by LS method and ES method

格納する。(埋めることができる命令が無ければNOPを挿入)

4. プライムブロックに実行すべき命令が無くなるまで3を繰り返す。終われば1へ。
5. 「飛び先表」を参照して、各ブランチ系命令や(C言語のswitch文等が生成する)ジャンプテーブルの飛び先アドレス(あるいはオフセット)を補正する。

例えば、対象VLIWプロセッサの実行ユニットの構成が整数系(IU1,IU2):2, 浮動小数乗除(FMDU):1, 浮動小数加減(FASU):1, ブランチ(BRU):1, ロード/ストア(LSU):2 であると

て、Fig.2のコードをVLIW命令に展開した結果はFig.3の右下の図のようになる。

4 実行時並列化

実行時に命令のスケジューリングを行なうES方式では、ひとつひとつの命令語がFig.3の左の図のように、Aフィールドを持っている。プロセッサは、これを見てプライムブロックを認識し、その中から実行可能な数の命令を切り出しながら実行していく。またこの方式では、プライムブロックの大きさがどうしても小

さくならざるを得ないようなプログラムでも、命令コードの密度を高くすることができる。通常の VLIW プロセッサでは、その場合は多くの命令語が NOP になってしまう。

プロセッサ内の命令発行ユニットは次の手順を踏んで、命令を実行ユニットに送り込む。

1. プライムブロックの先頭から発行命令数 (N とする) ずつ命令を切り出す。
2. 切り出したうちの 2 番目以降の命令語について、その A フィールドと先頭の語の A フィールドの値が異なっていれば、切り出した命令のうち、その命令より後の命令を全て無効化 (NOP への置き換え) する。
3. 無効化された命令があればその先頭が、そうでなければ切り出した命令の次の命令が、次の命令切り出しの開始点となる。

Fig.3 の右上の図は、Fig.2 のコードを 3 命令同時実行可能なプロセッサで実行する場合の例である。

5 評価

実験

Fig.1(a) のプログラムのループの部分を 2,4,8 回展開し、プライムブロックが最大になるように調整したコードを作った。ループの展開数に応じて命令レベルの並列度が大きくなり、プライムブロックも大きくなる。そして、ES 法を適用したプロセッサを使って実行する場合の、実行に要するクロック数をシミュレータによって求めた。その仕様は以下の通りである。

アーキテクチャ: レジスタ構成や命令の意味は、基本的に Sparc アーキテクチャ [5] に準ずる。

実行ユニットの個数: 2, 3, 4, 5, 6, 7, 8 個

レジスタ衝突: スコアボードを使って制御する。命令の発行の単純化のために、ひとつの実行ユニットが stall して命令を受け付けなくなると、他の実行ユニットも同様になる。

Table 1: Execution times

Operation	Latency (Clocks)
fadds	1
fnuls	3
fdivs	6
ALU	1
imul	4
idiv	18

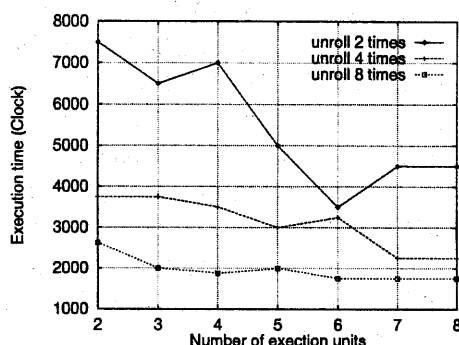


Fig. 4: Execution time for Fig. 1(a)

メモリ参照: 全てのロードは、必ずキャッシュメモリに当たり、2クロックの遅れ完了し、ストアも1クロックで完了する。

分岐: 遅延スロットを使わず、3クロックの遅れで次の命令の実行を開始する。(オリジナルの Sparc と異なる。)

演算結果の遅延: 演算命令の結果の遅延は Table1 の通りである。この数のクロックが経過する前に演算結果が書かれていないレジスタを参照すると、stallが発生する。この値は SuperSPARC での値 [6] を参考にした。

Fig.4 にループの展開数と実行ユニットの数を変えた時の、ループの実行時間を示す。

検討

Fig.1 のプログラムについてみれば、コンパイラがプライムブロックを充分大きくする努力を行なうほど実行時間は小さくなるが、実行ユ

ニットが4個ぐらいで性能が飽和してしまう。これは、実行ユニットの個数が多過ぎると、演算の結果を書く命令と読む命令の距離が近くなり、後者の実行が遅延するために、実行ユニットの数が少ない場合と同じ結果になってしまうからである。

LS法については、以下のような議論が成り立つ。ES法では、各々の実行ユニットがどんな種類の命令でも受け付けることができるのに対し、LS法で対象にするVLIW命令では、多くの場合実行ユニット毎に指定できる命令の種類が異なっている。例えばFig.1のプログラムに対してコンパイラが生成するコードが、ロードだけのプライムブロックと演算だけのプライムブロックになっている場合は、実行ユニットの個数ほどには性能が上がらない。そこで、コンパイラがプライムブロック内の命令の種類を均等化する。そうすれば、ES法と同様の性能が期待できる。

6 おわりに

命令レベル並列処理について、プライムブロックという概念を提起し、それを用いてVLIW計算機で性能を維持したままバイナリコードの共有を行なうための手法を2つ提案した。そして、次の事項を確認した。(1)コンパイラがプライムブロックを大きくする努力を行なうほど良い実行性能が得られること。(2)適当な実行ユニットの数とレジスタ数は、命令実行結果の遅延と関係があること。

今後の課題として以下の事項がある。(1)より多くのプログラムでの測定や評価。(2)ES方式での命令発行装置の改良。(3)プライムブロックに対するより好ましい制約条件の発見と、コンパイラによるその生成技術の開発。(4)ガードつき命令を持つVLIWプロセッサでの評価。(3)は、現在著者らの研究室のコンパイラプロジェクト[8]の一環として取り組んでいる最中である。

最後に、本研究についてのアドバイスを頂いた、電気通信大学情報工学科 中澤喜三郎教授、同 中川圭介助教授、ならびに、有益な情報を提供して頂いた(株)富士通研究所 清水 剛博士に感謝します。

参考文献

- [1] Ellis, J.: *Bulldog: A Compiler for VLIW Architectures*, MIT Press (1986).
- [2] Hennessy, J. and Patterson, D.: *Computer Architecture A Quantitative Approach, 2nd ed.*, Morgan Kaufmann Publishers Inc. (1996).
- [3] Lam, M.: Software pipelining: An effective scheduling technique for VLIW processors, *SIGPLAN Conf. on Programming Language Design and Implementation*, ACM, pp. 318-328 (1988).
- [4] Philips Electronics North America Co.: *TM1000 Preliminary Data Book*. <http://www.trimedia.philips.com>.
- [5] SPARC International: *The SPARC Architecture Manual Version 8*, Prentice-Hall (1992).
- [6] Sun Microsystems Computer Co.: *Super-SPARC White Paper* (1992).
- [7] 渡邊坦, 森教安, 菊池純男: SIMDマシンで並列実行させる同型命令列の認識方式, 情報処理学会プログラミング研究会 報告 No.8, pp. 19-24 (1996).
- [8] 渡邊坦, 鈴木貢: コンパイラ・エンジニアリング, 第38回プログラミング・シンポジウム報告集, 情報処理学会, pp. 135-142 (1997).