計算機アーキテクチャ 91－3
（1991. 11. 21）

スーパースカラプロセッサにおけるループ並列化の検討

白川 健治　井上 淳
東芝総合研究所

スーパースカラ・プロセッサにおける、ループ並列化の検討する。ループ並列化としてループ・アンローリングおよびソフトウェア・パイプライニングをとりあげる。スーパースカラ・プロセッサのモデルを設定し、コンパイラの生成するコードをシミュレートすることで、ループ並列化の効果を評価する。

評価の結果をもとに、ループ・アンローリングについては、ループの特性に応じて適当なアンローリング回数があることを示す。ソフトウェア・パイプライニングについては、あらかじめアンローリングによってループ内部の並列可能な命令を増やしたのちに行なうことが、良い実行性能をもたらすことをしめす。また、両ループ並列化によるコードサイズの増大に関わる問題について論ずる。

Evaluation of Loop Optimizations for a Superscalar Processor

Kenji Sirakawa, Atsushi Inoue
Research and Development Center, Toshiba Corporation
1 Komukai-Toshiba-cho Saiwai-ku Kawasaki, Kanagawa 210

Based on the observations of the simulated results, we show two important facts for loop optimizations. First, for loop unrolling, loops should be unrolled to an apropriate number which is determined based on the characteristics of the loop. Second, for software pipelining, the loop should be preprocessed to get a higher degree of instruction parallelism; for example using loop-unrolling to expand the loop before software-pipelinging is applied. Discussions are also made for the increase in size of the object code after these loop optimizations are applied.

1 はじめに

近年提唱されている高性能プロセッサの多くは命令レベルの並列性を重視したスーパースカラ・プロセッサである。本稿では、スーパースカラ・プロセッサ向きに作成したコンパイラの評価を行なう。

コンパイラの行なう最適化のひとつにループ並列化がある。ループ並列化の最適化手法から「ループ・アンローリング」、「ソフトウェア・パイプライニング」をとりあげ、これらのループ最適化をスーパースカラ・プロセッサに適用した場合の性能について論ずる。

スーパースカラ・プロセッサのモデルを設定し、そのモデルを実現するハードウェアシミュレータを用いて評価を行なう。まず、評価の対象となるスーパースカラ・プロセッサのモデルについて説明し、このプロセッサ向きのコンパイラの概要について述べる。次に、ここで評価するループ並列最適化手法について説明し、それぞれのスーパースカラ・プロセッサでの効果を評価し、問題点とその解決策を探る。

2 プロセッサのモデル

ループ並列化の評価を行なう対象であるスーパースカラ・プロセッサの構成について述べる。

スーパースカラ・プロセッサに特徴的なものは、その命令実行方式である。しかし、静的にスケジュールされているコードを生成する専用のコンパイラによる性能向上を考えるには、命令実行方式は簡単である方が評価には都合が良い。

スーパースカラ・プロセッサのモデルを次のように設定する。

◇ 命令発行方式は次の通り

- 4命令フェッチ
- in-order 発行、out-of-order 完了による4命令同時実行
- スコアボーディングによる依存、リソースコンフリクトの解消

◇ 演算器の次のような構成である

- 2つの整数 ALU
- 2つのロード / ストアユニット
- 1つの浮動少数点 adder
- 1つの浮動少数点 multiplier
- 1つのブランチユニット

◇ レジスタファイルは次のような構成である。

- 32bits × 32 の整数レジスタファイル
- 64bits × 32 の浮動少数点レジスタファイル

また、各演算器での命令実行に要するサイクル数を図1に示す。

図 1: 演算器構成

命令	命令発行	latency
整数 ALU(除 mul、div)	1	1
load (整数レジスタ)	1	2
load (FPU レジスタ)	1	3
fadd、fsub、fmul	1	2
fdiv (single)	7	9
fdiv (double)	14	16
fmov	1	2

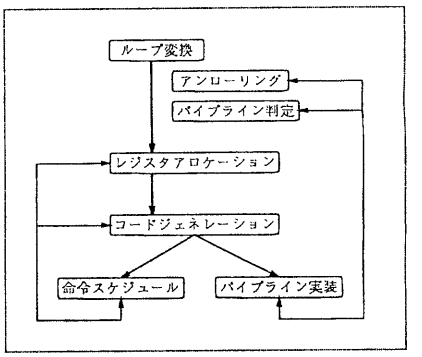


図 2: コンパイラ構成

3 コンパイラの構成

前節で述べたスーパースカラ・プロセッサのための専用コンパイラについて述べる。

専用コンパイラによる VLIW 的なコード生成は、スーパースカラとは馴染まない要素を含んではいるが、プロセッサの性能をどこまでコンパイラで引き出し得るかという問題を評価するには、演算器構成や演算器の性能に合わせてチューンアップしたコンパイラによるコード生成は不可欠である。

ここで用いるコンパイラ、とくにそのコード生成と命令スケジュールに関わる部分は、プロセッサのインプリメントに依存している要素が多い。

コンパイラの最適化部のうち、ここでの評価に関係の深い部分の構成を図2に示す。

◇ ループ変換部

- 大域的なループ構造の抽出、依存関係のリスト
- アンローリングの段数推定、実現
- パイプライン実現予測

◇ 命令スケジュール部

- リストスケジュールによる局所的命令スケジュール
- パイプラインスケジュールによる大域的命令スケジュール
- レジスタリネーミング
- 命令列のアラインメント処理
- 命令コンパクション

4 ループ並列化の評価

4.1 評価方法

各ループ並列化手法を紹介し、スーパースカラ・プロセッサでの性能を評価する。

プロセッサモデルを実現したハードウェアシミュレータ上でコンパイラの生成したコードを実行することで、最適化の効果を評価する。並列化の対象となるループは、Livermore Loops[1]から抜粋したものを利用した。

4.2 ループ・アンローリング

ループアンローリング[3]はループ内部を複数コピーして展開し、ループ反復を減らすプログラム変換である。広く行なわれるループ最適化であり、分岐およびメモリアドレス演算の減少により実行効率を向上をねらったものである。しかし、スーパースカラのような複数命令同時実行型のプロセッサでは、ループ内の独立した命令が増加することによる並列実行による実行効率の向上が主である。

図3を例にして、アンローリングの実際を説明する。このループはLivermore Loopsの12番のループである。図4は、このループにたいしてループ最適化を行なわなかったときの命令列である。各行はフェッチの単位である。便宜上、命令間の依存関係によるストールの発生するサイクルは '-' で印した。また、'.' で印された欄は依存関係により実行を止められている命令のスロットである。ロード命令および浮動少数点演算のレイテイシィによりほとんどのサイクルを、ストールで費やしている。ブランチペナルティを0とすると、ループ1回に7サイクル要する。

```
/* loop 12 */
for (i = 0; i ¡ n; i++)
    X[i] = Y[i+1] - Y[i]
```

図 3: アンローリングの対象となるループ

```
   r3 -- &Y[0]    r4 -- &X[0]    r2 -- &X[n]
loop:
   ld f0,8(r4)    .              .              .
   -
   -
   subd f4,f0,f2
   -
   addi r4,r4,8   movd f2,f0     .              .
   sd f4,0(r3)    addi r3,r3,8   bne r4,r2,loop.
```

図 4: アンローリング前のスケジュール

```
   r4 -- &Y[0]      r3 -- &X[0]      r2 = &X[n]
loop:
   ld f0,8(r4)      .                .                .
   ld f12,16(r4)    .                .                .
   ld f14,24(r4)    .                .                .
   subd f4,f0,f2    ld f16,32(r4)    .                .
   subd f6,f12,f0   ld f18,40(r4)    .                .
   subd f8,f14,f12  ld f1,48(r4)     .                .
   subd f10,f16,f14 ld f3,56(r4)     .                .
   subd f7,f18,f16  ld f5,64(r4)     .                .
   subd f9,f1,f18   .                .                .
   sd f4,0(r3)      subd f11,f3,f1   addi r4,r4,64    .
   sd f6,8(r3)      sd f8,16(r3)     subd f13,f5,f3   .
   sd f10,24(r3)    sd f7,32(r3)     .                .
   sd f9,40(r3)     sd f11,48(r3)    movd f2,f5       .
   sd f13,56(r3)    addi r3,r3,64    bne r4,r2,loop   .
```

図 5: アンロール後のスケジュール

図5は、このループに対して8回のアンロールを施したあとの命令列である。ループ1回に1.8サイクル要する。

この例のループの場合、ループ内部の文の間に、ループのインデックスにまつわる依存以外の依存関係がないため、十分な並列性が存在する。従って、依存関係によるストールはない。

このような、アンローリングを行なう場合、「何回のアンロールを施すのが良いか」が問題となる。この点に関しては、次の指針によりアンロール段数を決定する。

1、命令間の依存関係によってアンロールの効果が期待できない場合がある。

◇ ループの異なる反復間にデータ依存がある場合
異なる反復の命令の並列実行は期待できない。
しかし、ループ境界でのデータ依存や制御依存がなくなることによる並列性向上はある。

◇ ループの異なる反復間にデータ依存がない場合
アンロール段数が大きいほど並列性は向上する。
しかし、次の要因により並列性は押えられる。

- ループ内部でのデータ依存や制御依存
- ループ内部でのレジスタ使用状況

2、ループを展開することによるコードサイズの増大は、アンロールの効果を落す。これを防ぐためには、効果の期待できないアンロールを抑止する必要がある。ループ内部での演算器の使用頻度の分布がプロセッサの演算器構成からかけ離れる場合はアンロールを抑止する。

図6、図7は、Livermore Kernel のいくつかのループについて、アンローリング段数を変えたもとでのアンローリングの効果を示している。各ループについてのMflops値を実行性能として、アンローリングなしの場合との性能比で表している。図中、黒丸で示したアンローリング段数は、コンパイラが最適段数と推定したものである。また、いずれもデータキャッシュミスは存在しないものとしている。

図6は、反復間にデータ依存のないループを集めたものである。データ依存のない場合には、アンロールすることでループ内部の独立した命令を増やすことが可能であるため、高い並列実行度をしめす。しかし、アンロールの段数を増やすに従って、ループ帰納変数に関わる最適化などにより依存関係を生ずる場合、アンローリングのし過ぎはかえって実行効率を落す。

図7は、反復間にデータ依存のあるループを集めたものである。この場合、ループ内部の独立した命令を増やす効果はそれほど大きくないが、しかし、アンロールの回数を増やすに従って、ある程度性能の向上を見ることができる。

アンローリング段数を変えた場合の評価から、ループの性質・大きさと、プロセッサの演算器構成から決まる、最適なアンローリング段数があることが分かる。

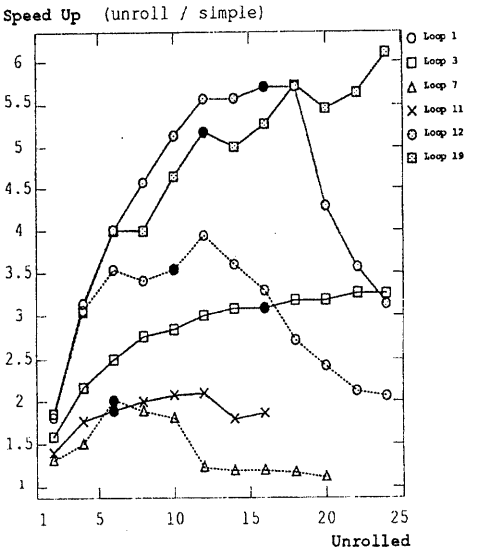


図 6: アンローリングの効果(データ依存なし)

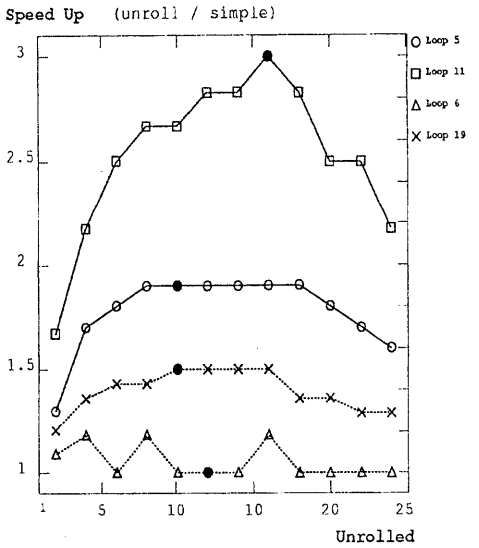


図 7: アンローリングの効果(データ依存等あり)

4.3 ソフトウェアパイプライニング

ソフトウェア・パイプライニング[2]は、ループアンローリングを無限に繰り返した状況を実現するプログラム変換である。

図8を例にしてパイプライニングの過程を模式的に説明する。ループ内部を、A、B、Cとするとき(これらは文ではなく、命令のブロックである)、ループのi回目が終了する前に、i+1、i+2回目を開始する。ループの3回目以降では定常状態となる。この定常部を実現するべくループを再構成するのが、パイプライン・スケジュールである。ループの最初と最後の2回は定常状態に含まれないため、ループの外にプロローグとエピローグとして実行する。

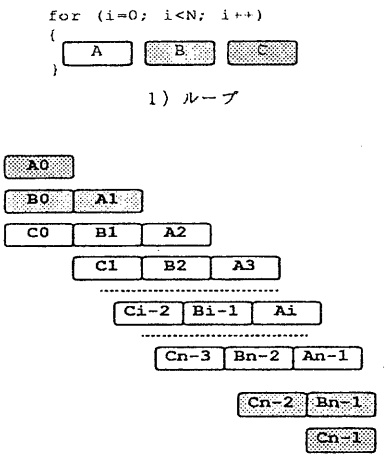


図 8: パイプライニングとは

```
   r4 -- &Y[0]    r3 -- &X[0]    r2 -- &X[n -1]
prologue:
   -
   ld f0,8(r4)    .              .              .
   -              .
   -
loop:
   addi r4,r4,8   subd f4,f0,f2  .              .
   movd f2,f0     ld f0,8(r4)    .              .
   -
   sd f4,0(r3)    addi r3,r3,8   bne r2,r4,loop
epilogue:
   addi r4,r4,8   subd f4,f0,f2  .              .
   mov f2,f0      .              .              .
   -
   sd f4,0(r3)    addi r3,r3,8   .              .
```

図 9: パイプライン・スケジュールの例

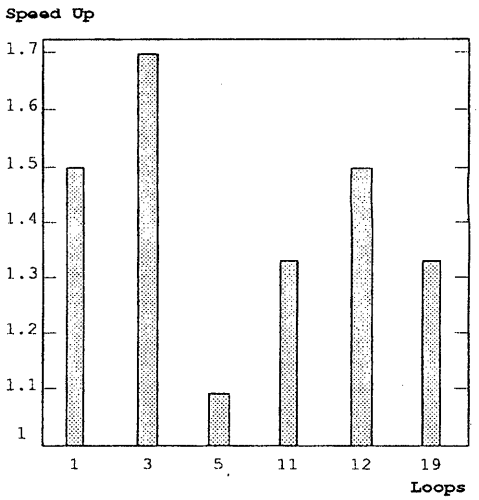


図 10: パイプラインスケジュールの実行効率

図9は、アンローリングの節で例としたループ(図3)をパイプラインスケジュールしたものである。この例では、ループ内部は2つのブロックに分割されている。ブランチペナルティを0とすると、ループ1回あたり4サイクルである。

パイプライン・スケジュールを実現するために重要なものは、

- ループの異なる反復の間の依存関係
- ループ内部に制御依存が存在するか否か、またその複雑さ
- ループの終了条件とその複雑さ

このうち、前者の2つはパイプライン後の実行効率上のメリットを決めることは、アンローリングの場合と同様である。

パイプライニングの実現には、パイプライニング判定部で、パイプライニング可否の判定および適当なパイプライン段数を決定し、命令スケジュール部で、パイプライン段数によりモジュロ・スケジューリング等によりパイプラインを実現するコードを生成する。

図10は、Livermore Kernelのいくつかのループについて、ソフトウェア・パイプライニングを適用した場合とパイプライニングを行なわなかった場合との実行性能比を示している。

4.4 アンローリングとパイプライニング

前節でアンローリングとソフトウェア・パイプライニングを適用した場合の性能向上を見た。単純なループに対するアンローリングの性能向上(図6、図7)と、パイプライニングによる性能向上(図10)を比較して分かるように、十分にアンローリングを施したほうが実行効率がよい。

これは、パイプライニングだけではまだプロセッサに並列実行の余裕が残っているためである。そこで、まずアンローリングを施してループ内部の並列実行可能な命令を増やした後にパイプライン・スケジュールを行なう。

図11は、アンローリングを行なった後にソフトウェア・パイプライニングを行なったの実行性能と、ソフトウェア・パイプライニングだけを行なった場合の実行性能の比を示している。また、図12は、アンローリング及びパイプライニングを行なった場合の実行性能を各ループについて見たものである。

loop 1、loop 12のような依存関係の少ないループに関しては、アンローリングによる前処理によって、ソフトウェア・パイプライニングの効果が3倍以上向上する。これら以外のループについても、アンローリングによる性能向上がのぞめる。

しかし、アンローリングによりループの異なる反復間に依存が持ち込まれると、パイプライニングにとって不利な状況となりうる。loop 1で5回以上アンローリングを行なった場合はこの例である。

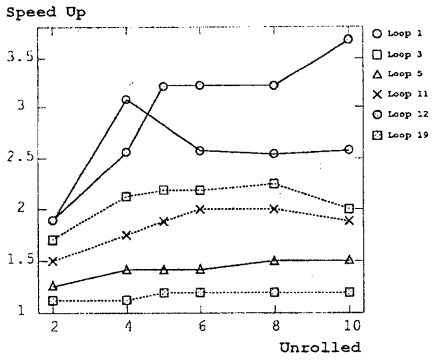


図 11: パイプラインスケジュールの実行効率(アンロール後)

4.5 命令キャッシュミスの影響

前節までは、キャッシュミスの影響を無視してきた。この節では命令キャッシュミスについて考える。アンローリング、パイプライニングはともにコードサイズの増大するプログラム変換である。アンローリングはコードを複製するためループ本体が大きくなる。ループの終了条件が繁雑である場合や、パイプライニングのプロローグ・エピローグ処理はループ外のコードが増える。

次のようなキャッシュの構成を考える。

- 命令キャッシュ
  16kbytes、2 way set associative、128bit block refill time 30cycles

キャッシュペナルティのもとで、ループがキャッシュ内に全て存在する状態と、キャッシュには存在しない状態から実行を開始した場合の実行効率を測定する。

図12で、×印はアンローリングのみ、○印はアンローリング・パイプライニング併用である。実線はキャッシュペナルティなし、破線はコールドスタート時の実行性能である。いずれのループも400(定数)回実行する。現実のキャッシュミスのもとでの実行性能はこの両者の中間に位置するであろう。

loop 3、loop 5、loop 19 のようにキャッシュペナルティのある場合、アンローリングを重ねても命令キャッシュミスのため性能向上が見られない例がある。また、アンローリング回数によっては、プロローグ・エピローグ処理が冗長になる場合、性能が低下する(アンローリング6回はこの場合、悪い数である)。

命令キャッシュミスによるペナルティを考慮して、コードサイズもアンローリング回数を決定する要素の一つとなる。

5 おわりに

スーパースカラ・プロセッサにおいて、アンローリングおよびソフトウェア・パイプライニングを施した場合の性能向上を評価した。

アンローリングに関しては、各ループの特性に応じたアンロール段数を決定するが重要である。

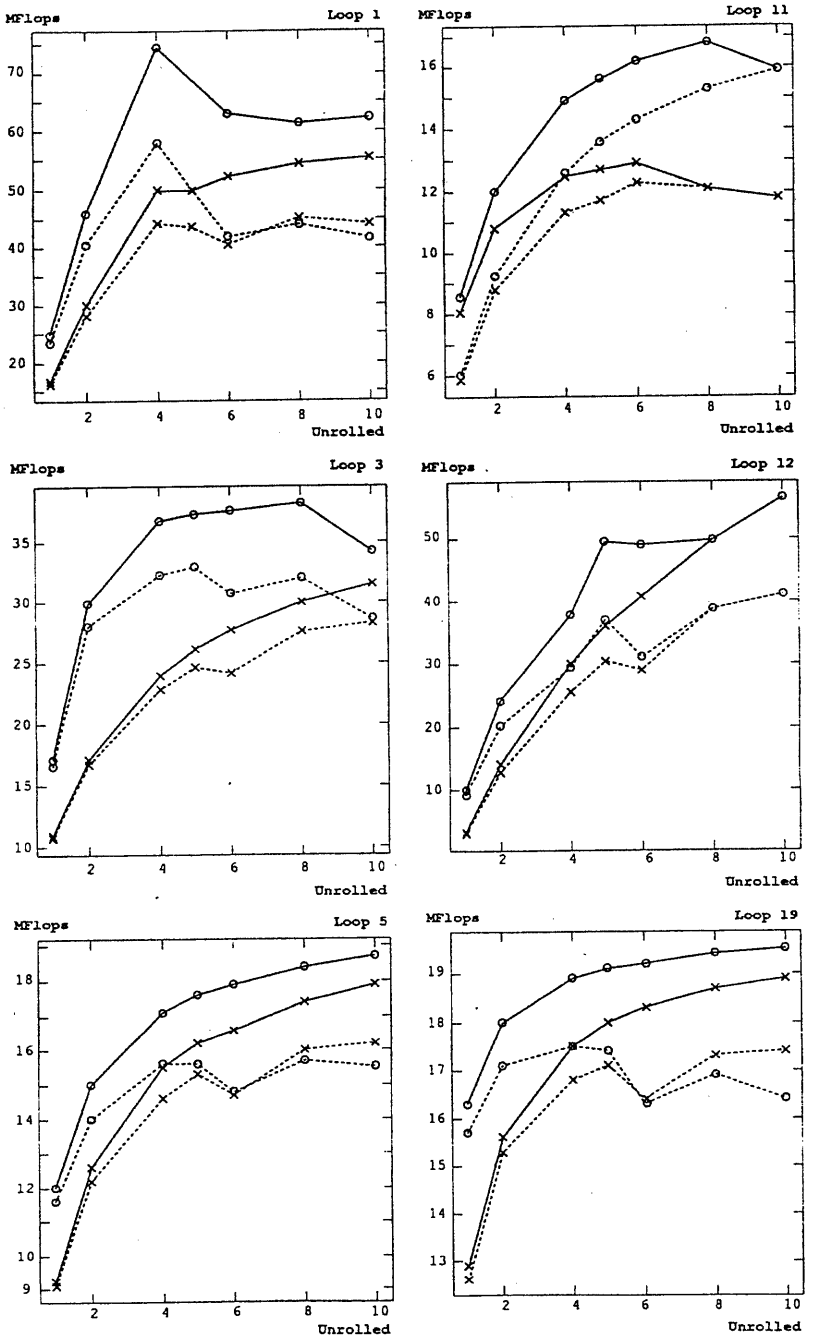
ソフトウェア・パイプライニングに関しては、ループ内部の並列度が少ない場合には、まずアンローリングによってループ内部の並列実行可能な命令を増やすことで高い並列性を引き出すことができる。

参考文献

[1] J. T. Feo. "an analysis of the computational and parallel complexity of the livermore loops". *Parallel Computing*, 7(2):163–185, June 1988.

[2] M. Lam. "software pipelining: An effective scheduling technique for vliw machines". In *Proc. SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328, June 1988.

[3] S.Weiss and J. E. Smith. "a study of scalar compilation techniques for pipelined supercomputers". In *Proc. ASPLOS II.*, pages 105–109, Oct. 1987.



×－アンローリングのみ、○－アンローリング・パイプライニング併用
実線－キャッシュベナルティなし、破線－キャッシュベナルティあり

図 12: アンローリング、パイプライニングの実行性能