ループアンローリングの特徴抽出とそのモデル化

吉田映彦[†] 佐藤周行[††]

ループ最適化の 1 つにループアンローリングがある．プログラムにループアンローリングを適用することで，命令レベルでの並列度が上がり，レジスタの使用効率も良くなるという効果があることから，近年のスーパスカラプロセッサにおいて実行性能を高めることができる．コンパイラの中にはループアンローリングを自動的に適用するものも存在するが，残念ながら必ずしも最適なループアンローリングが行われているわけではない．従来のループアンローリングの研究では，ループアンローリングを適用することでプログラムの性能を向上させるという結論に達しているものの，定量的な評価が行われていないのが現状である．本研究の目標は，ループアンローリングをモデル化し，定量的な評価を行うための基盤を構築することである．本稿では最適なループアンローリング段数を得るための指針として，どのような要素がループアンローリングに影響を与えているかを調べ，その定量的な解析を試みた．その結果，演算の実行時間とキャッシュミスによるコストの 2 つの要素がループアンローリングに影響を与えていることが分かり，特にキャッシュミスに関しては，ロードと参照の距離を考慮することが重要であった．

Characteristics Extraction of Loop Unrolling and Its Modeling

Yoshida Teruhiko[†] and Sato Hiroyuki[††]

Loop unrolling is one of the useful methods to optimize loop execution. Loop unrolling works well for modern super-scalar processors because it improves instruction-level parallelism and register usage. Compilers can automatically unroll loops but their factors are not always optimal. Conventional research for the loop unrolling just concludes that applying loop unrolling can improve program performance. However, there is little discussion of quantitative evaluation. Our goal is the construction of a symbolic model to quantitatively evaluate loop unrolling. In this paper, we inquire which parameters affect loop unrolling perormance and have quantitative analysis. As a result, we study both calculation cost and cache miss cost affect loop unrolling. Particularly, it is important to consider the distance between load instructions and reference instructions.

1. はじめに

ループ最適化の 1 つにループアンローリングがある．プログラムにループアンローリングを適用することで，命令レベルでの並列度が上がり，レジスタの使用効率も良くなるという効果があることから，近年のスーパスカラプロセッサにおいて実行性能を高めることができる．しかし，アンローリング段数を的確に決定できなかった場合，性能が低下する場合も存在する．実際，アンローリング段数が小さすぎると命令レベルでの並列度が必ずしも上がらず，ループ内でのレジスタ使用効率が低下する恐れがある．また，アンローリング段数が大きすぎるとスピルコードを生成することになり，余計なロード，ストアが生じてしまう．近年のコンパイラにはループアンローリングを自動的に適用するものも存在するが，残念ながら必ずしも最適なループアンローリングが行われているわけではない．そのため，ユーザが手動でループアンローリングを記述するといったことも行われている．

従来のループアンローリングの研究では，ループアンローリングを適用することでプログラムの性能を向上させるという結論には達しているものの，定量的な評価が行われていないのが現状である．本研究の目標は，ループアンローリングをモデル化し，定量的な評価を行うための基盤を構築することである．モデル化を行うことによってループアンローリングの効果を定量的に解析できることから，コンパイラに対してループアンローリングを用いた最適化が行われる場合の指

† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, Graduate School of Science, the University of Tokyo
†† 東京大学情報基盤センタースーパーコンピューティング研究部門
Computer Centre Division, Information Technology Center, the University of Tokyo

針を提供することもできると考えられる．

本稿の構成は次のとおりである．まず，2 章でモデル化のための前提条件を述べる．次に 3 章でループアンローリングの効果を予測するためのモデルを構築する．4 章ではいくつかの例題を用いてモデルの検証を行う．5 章では関連研究について述べ，最後に 6 章でまとめと今後の課題について述べる．

2. モデル化のための準備

2.1 モデル化の前提条件

以下の議論で対象とするループは，内部に値のロード/ストアおよび浮動小数点演算を含むような数値計算プログラムとする．ループの反復回数はあらかじめ分かっていることとし，議論を簡単にするために反復回数はアンローリング段数の倍数としておく．すなわち，ループアンローリングを適用した場合に端数の処理が存在しない．また，ループアンローリングの適用は最内ループに限定するため，プログラムで指示されているメモリのアクセスパターンを変更しない．ループアンローリングの効果はプログラムが実行されるハードウェアやプログラムがコンパイルされるソフトウェアに大きく依存する．ここでは，モデル化を行ううえで前提としているハードウェア，ソフトウェアの条件を示す．

- ハードウェアモデル
  プロセッサとしては典型的なスーパスカラの RISC 系プロセッサを仮定する．これは，次に示すような性質を持つ．
  - 演算ユニット
    プロセッサは複数の整数演算ユニット，浮動小数点演算ユニットを持っており，1 サイクルあたりに複数の命令を同時に発行することが可能である．浮動小数点演算のオペランドはすべてレジスタである必要があり，計算を行うにあたってはあらかじめ値をレジスタにロードしておく必要がある．
  - ロード/ストアユニット
    プロセッサは複数のロード/ストアユニットを持つ．ただし，近年のプロセッサではロードあるいはストアのどちらか片方だけを 1 サイクルに 1 回実行できるようになっているものが多く，今後の議論でも特に断りがない限りは 1 サイクルあたりロードあるいはストアのどちらか 1 つだけを処理することができることにする．
  - 命令レイテンシ
    プロセッサで実行される命令セットには実行されてから結果が得られるまでに遅延（レイテンシ）が存在し，それは先行命令と後続命令によって定義される．これらはプログラムのスケジューリングを行うための重要な要素となる．
  - キャッシュ
    プロセッサは 1 次キャッシュと 2 次キャッシュを持っており，キャッシュミスが生じることによるペナルティが存在する．ループアンローリングを適用する対象を最内に限定したことでメモリへのアクセスパターンがループアンローリングによって変化しないことから，同一プログラムにおいてはキャッシュミスが生じる確率はアンローリング段数によらずつねに一定である．
- ソフトウェアモデル
  ここでいうソフトウェアモデルとは，アンローリングの対象となるプログラムがどのように生成されるかを意味しており，モデル化を行ううえでコンパイラに要求されるものである．ここでは，次に示すような性質を仮定する．
  - レジスタアロケーション
    レジスタアロケーションのアルゴリズムは，基本ブロック内での割当てを優先するものとする．こうすることでループ外に存在する値のためにレジスタがリザーブされるといったケースがなくなるからである．
  - 命令スケジューリング
    命令スケジューリングのアルゴリズムには単純なリストスケジューリングを使用する．スケジューリングのアルゴリズムが複雑になると，アンローリング段数を増加させた場合に生成されるコードの特徴が分かりにくくなるからである．

  今回はターゲットコンパイラとして GNU-C 2.95.2 を選択した．GNU-C のコンパイラは上に述べた 2 つの条件を満たしているからである．

2.2 表　　記

本稿ではアンローリング段数を ℓ で表し，アンローリングの対象となるループのサイズを N で表す．ループ内にはロード/ストア命令や浮動小数点演算命令が存在し，1 反復あたりのそれらの命令数はアンローリング段数によって変化する．

3. モデル化

3.1 Unrolling Shape

ループアンローリングのモデル化を行ううえで，本稿では任意のアンローリング段数においてロード，ストア，浮動小数点演算命令がどのようにスケジューリングされるかを示したアンローリングの一般形を定義する．

定義 1（Unrolling Shape） **N** を自然数の集合（具体的には実行サイクルを表す），**Unit** を演算ユニットの集合とし，$Ope(\ell)$ をアンローリング段数が $\ell-1$ 段から ℓ 段に変化した場合に 1 反復あたりで増加する命令列（ただし $Ope(1)$ はアンローリングを行わなかった場合の命令列）とした場合，ある制約条件の下で，

$$Ope(1), Ope(2), \cdots, Ope(\ell) \rightarrow \mathbf{N} \times \mathbf{Unit}$$

という対応付けを行ったものを，Unrolling Shape という．ただし制約条件は，

- 命令のスループット
- 実行から結果を得るまでの遅延（レイテンシ）

の 2 つである．

Unrolling Shape とは，アンローリング段数が ℓ 段の場合に，ループ 1 反復の命令が何サイクル目で実行されるかをスケジューリングしたものである（Unrolling Shape の具体例は 4 章で示す）．Unrolling Shape の定義から，スケジューリングを行うためには，プロセッサモデルで定義される演算ユニットや命令レイテンシといった情報が必要となる．Unrolling Shape を求めるためのアルゴリズムの概要を**図 1** に示す．基本的には，リストスケジューリングを用い，各アンローリング段数で増加する命令群をスケジュールするという作業を繰り返すことになる．

実際にスケジューリングの対象とするのはロード/ストア命令と浮動小数点演算命令だけに限定する．近年のスーパスカラプロセッサで整数演算ユニットを複数持ち 1 サイクルで 2 つ以上の命令を同時に実行できる機構を持っている場合には，変数のアドレッシングやループカウンタの処理，分岐判定は，ロード/ストアおよび浮動小数点演算と同時に実行されることにより，その計算のコストが表に出てこない場合が多いことが予想できるからである．実際ここで述べる方法で変数のアドレッシングやループオーバヘッドの命令も含めてスケジューリングを行った場合では，それらのほとんどがロード/ストアおよび浮動小数点演算命令と並列に実行されるようにスケジューリングされ，そのコストは無視できるという結果になった．

ℓ : アンローリング段数
$Ope(\ell)$: アンローリング段数が $\ell-1$ から ℓ になったときに増加する命令列
Q : スケジューリング待ち命令の集合

```
1.   for( x = 1, 2, ..., ℓ ){
2.     P = Ope(x)
3.     while( P ≠ ∅ ){
4.       Q ={p ∈ P | p has no dependency}
5.       P = P − Q
5.       for( q ∈ Q ){
6.         schedule q
7.         update dependency concerning q
8.         Q = Q − {q}
9.       }
10.    }
11.  }
```

図 1 Unrolling Shape を求めるアルゴリズム
Fig. 1 Algorithm to calculate Unrolling Shape.

cycle	LD/ST	Float
j	ld a_i, r_0	
$j+1$	ld b_i, r_1	
$j+2$		
$j+3$		add r_0, r_1, r_2
$j+4$		
$j+5$		
$j+6$	st r_2, c_i	

distance:
p(add r_0, r_1, r_2)−p(ld b_i, r_1)=2
p(st r_2, c_i)−p(add r_0, r_1, r_2)=3
(ただし，$p(x)$ は命令 x が実行されるサイクル)

図 2 Unrolling Shape の例
Fig. 2 Example of Unrolling Shape.

次にスケジューリングの例を示す．ここで，スケジューリングアルゴリズムには 2.1 節で述べたソフトウェアモデルを満たすコンパイラの 1 つである GCC に準ずるものを用いることにする．今，

- **Unit** ={ LD/ST　Float}
- $Ope(i)$ =
  {ld a_i, r_0　ld b_i, r_1　add r_0, r_1, r_2　st r_2, c_i}
- ld，st，add のレイテンシがそれぞれ 2，1，3

という条件が与えられているとすると，サイクル j を基準にスケジューリングを行った場合には Unrolling Shape は**図 2** に示すような形となる．まず，依存関係

を持たない 2 つの命令 ld a_i, r_0，ld b_i, r_1 が待ち行列 Q に入っているはずなので，これら 2 つの命令をスケジューリングする．次に，この 2 つのロード命令がスケジューリングされたことによって，add r_0, r_1, r_2 がスケジュール待ち集合 Q に追加されるため，この命令のスケジューリングを行う．ここで，ロードのレイテンシが 2 であることに注意すると，add 命令は $j+2$ 以前に配置されることはないことが分かる．最後に add 命令がスケジューリングされたことで st r_2, c_i をスケジューリングすることが可能となり，レイテンシに注意してこの命令の配置を完了すれば，図 2 を得る．

アンローリングの一般形である Unrolling Shape を定義することで極限をとるといった操作も可能となる．アンローリングの極限とは無限にアンローリングした状態を指し，この計算を行うことでアンローリングによる効果の上限を予測することができるから，最大でどの程度の効果が得られるかを知ることができる．

3.2 Parallelization Effect

プログラムの実行時間は，演算に要した時間にロード/ストアのストール等によるペナルティを加えたものである．ここでは，前者の値の予測について述べる．

定義 2（Parallelization Effect） アンローリング段数が ℓ 段の場合において 1 反復の実行に必要となるサイクル数を $cycle(\ell)$ で表したとき，演算に要するサイクル Parallelization Effect $pe(\ell, N)$ を，

$$pe(\ell, N) = cycle(\ell) \cdot \frac{N}{\ell} \quad (1)$$

で表す．

$pe(\ell, N)$ は，演算に必要となるサイクル数を表しているから，1 反復に要するサイクルに反復回数を乗じることで得ることができる．$cycle(\ell)$ は Unrolling Shape を計算することで求まることから，Parallelization Effect の値は Unrolling Shape を求めることができれば計算することが可能であることが分かる．

3.3 Memory Effect

実際の実行時間の内訳を考えた場合，キャッシュミスのコストは無視できるものではないため，その予測は重要である．

定義 3（Memory Effect） キャッシュミスが生じることによって実行時間に与える影響を Memory Effect と呼ぶ．今，x がプログラム中で初めてロードされる変数とし，$X(\ell)$ をその変数の集合，$c_{L1}(\ell, x)$，$c_{L2}(\ell, x)$ をそれぞれ変数 x のロードによって 1 次キャッシュ，2 次キャッシュミスが生じたときのコスト，$m_{L1}(x)$，$m_{L2}(x)$ をそれぞれ 1 次キャッシュミス，2 次キャッシュミスが生じる確率とした場合，Memory Effect の値 $me(\ell, N)$ は，

$$me(\ell, N) = \sum_{x \in X(\ell)} c_{L1}(\ell, x) \cdot m_{L1}(x) \cdot \frac{N}{\ell} + \sum_{x \in X(\ell)} c_{L2}(\ell, x) \cdot m_{L2}(x) \cdot \frac{N}{\ell} \quad (2)$$

である．

アンローリングの対象を最内ループに限定したことによってメモリのアクセスパターンがアンローリング段数によって変化しないため，キャッシュミスが発生する回数はループの反復数 N だけで決定することができ，アンローリング段数 ℓ の値にはよらないので，$m_{L1}(x)$，$m_{L2}(x)$ は ℓ にはよらない．しかし，キャッシュミスの回数が一定であっても，キャッシュミスのコストはアンローリング段数に依存する．キャッシュミスによってロードに必要となるサイクルを s としておくと，アンローリング段数を増加させることで，値をロードする命令と値の参照を行う命令の距離が s 以上に離れることがあり，キャッシュミスのコストを完全に隠蔽することが可能だからである．今，プログラム内で変数 x が p サイクル目にロードされることを $p_d(\ell, x)$，変数 x が初めて参照されるサイクルを $p_r(\ell, x)$ と記述することにし，1 次キャッシュ，2 次キャッシュミスが発生した場合のロードのコストをそれぞれ s_{L1}，s_{L2} とすると，変数 x をロードしてキャッシュミスが生じた場合のコストは，

$$c_{L1}(\ell, x) = \max(0, s_{L1} - (p_r(\ell, x) - p_d(\ell, x)))$$
$$c_{L2}(\ell, x) = \max(0, s_{L2} - (p_r(\ell, x) - p_d(\ell, x)))$$

と表すことができる．

Memory Effect に影響を与えるものは，キャッシュミス率（$m_{L1}(x), m_{L2}(x)$）と $c_{L1}(\ell, x)$，$c_{L2}(\ell, x)$ であり，これらを適切に与えれば精度の良い予測が可能となる．これらはプログラムに依存するものであるため，具体的には 4 章で述べることにする．

3.4 Execution Cycle

ここでは各アンローリング段数における性能予測について述べる．プログラム全体の実行に要する値として Execution Cycle を使用することにし，次のように定義する．

定義 4（Execution Cycle） Execution Cycle とはプログラム全体を実行するのに必要となるサイクル数であり，Pararell Effect と Memory Effect を使用して表される．今，Execution Cycle を $U(\ell, N)$ で表すことにすれば，

$$U(\ell, N) = pe(\ell, N) + me(\ell, N) + \epsilon(\ell, N) \quad (3)$$

である．ただし，$\epsilon(\ell, N)$ は補正項である．
$\epsilon(\ell, N)$ は Unrolling Shape と実際に生成されるコードの間にある差を埋めるためのものであり，ほとんどの場合では $\epsilon(\ell, N) = 0$ である．

3.5 Unrolling Effect

定義 5 アンローリングを行うことでどの程度の性能向上が見られるかを示す値として，アンローリング段数が 1 の場合とアンローリング段数が ℓ の場合の比をとったものを用いることにし，これを Unrolling Effect（UE）と呼ぶ．$UE(\ell, N)$ は次のような式で表される．

$$UE(\ell, N) = \frac{U(1, N)}{U(\ell, N)} \tag{4}$$

今までに CPI（Cycles Per Instruction）値を基にループアンローリングやソフトウェアパイプライニングの効果を予測するといった提案は存在したが，キャッシュに関しての考慮が欠落していることが多かった．本式の 1 つの特徴としてキャッシュミスを考慮に入れたことから，従来より精度の良い予測ができる．$UE(\ell, N)$ の値を求めることを考えていくと必要となる値を得るためには Unrolling Shape を計算する問題に帰着することから，Unrolling Shape を求めることができれば，ループアンローリングの効果を予測できるということになる．

4. 実験と考察

最適な Unrolling Shape を求めるのは整数計画法問題となるので，一般的な計算は容易ではない．ここでは具体的な問題についてスケジューラを固定したうえで Unrolling Shape を求め，それらを基に Parallelization Effect と Memory Effect を計算し Unrolling Effect を求める．また，Unrolling Effect の実測値と予測値を比較し，本モデルの正当性を検証する．

4.1 実験準備

モデル化を行う対象とするプロセッサには UltraSparc を選択した．UltraSparc の特徴を次に示す[10]．これは，Unrolling Shape を求めるための制約条件となっている．

- 2 つの整数演算ユニットと 2 つの浮動小数点演算ユニットを持つ．
- 浮動小数点演算ユニットは加減算ユニット FA と乗除算ユニット FM からなる．
- 1 サイクルにロードあるいはストアのどちらか片方が 1 回実行でき，それは整数演算ユニットを用いる（ただし，本稿ではロード/ストアユニットと記述することがある）．

```
1:  do i = 1, n
2:    tmp(i) = a(i)
3:  enddo
4:  do i = 1, n
5:    b(i) = a(i) − a(i + 1) − a(i − 1)
6:  enddo
```

図 3 キャッシュ上にデータをロードするプログラム
Fig. 3 A program to load data on cache.

- 浮動小数点レジスタ数は 32 個である．
- 命令のレイテンシは，ロード，ストア，浮動小数点演算それぞれ 2，1，3 である．
- 1 次キャッシュのラインサイズは 16 byte，キャッシュミスによるペナルティは 6 サイクルである（ロードのレイテンシと合わせると，キャッシュミス時にはロードに 8 サイクルかかる）．

実験の対象とするプログラムは，8 バイトの浮動小数点演算を取り扱う 1 次元差分法に関して 2 種類のものを用意している．プログラムで扱うデータの配列の大きさは 12600 としており，すべてのデータが 2 次キャッシュ内に納まるようになっているため，以下の議論では 1 次キャッシュミスだけを考え，2 次キャッシュミスは考慮にいれない．ループ内で参照される変数をキャッシュ上に置くため，ループ本体が実行される直前にそれらの変数に連続アクセスするループを用意し，すべての変数を参照するようにしている．たとえば配列 a のデータをメインループで使用する 1 次元差分法のプログラムは，**図 3** のようなプログラムとなる．4〜6 行目が実際に差分を計算するループであり，そこで使用される配列 a は 1〜3 行目であらかじめキャッシュ上にロードされている．

また，UltraSparc には Performance Control Register（PCR）と呼ばれる計測用のレジスタが存在し，サイクルをカウントすることが可能である．本稿ではループの先頭と終端に PCR のサイクルをカウントしているレジスタの値を読み込む命令を挿入し，その差を計算することでループの実行サイクルの計測を行っている．本稿に掲載されているサイクルの値は，誤差を減らすために同様のプログラムを 100 回実行し，その平均をとった値である．

4.2 1 次元差分法 (1)

アンローリング段数が 1 段の場合と ℓ 段の場合の Parallelization Effect，Memory Effect を計算するという手順を追うことで，アンローリングを行った場合にどの程度の効果が得られるかを予測する．まずは，Parallelization Effect から考える．**図 4** にアンローリ

```
1:  do i = 1, n
2:    b(i) = a(i) − a(i − 1) − a(i + 1)
3:  enddo
```

図 4 差分法 (1) のプログラム（アンローリング 1 段）
Fig. 4 Program of difference method (1) (unrolling factor: 1).

```
1:  do i = 1, n, 2
2:    b(i) = a(i) − a(i − 1) − a(i + 1)
3:    b(i + 1) = a(i + 1) − a(i) − a(i + 2)
4:  enddo
```

図 5 差分法 (1) のプログラム（アンローリング 2 段）
Fig. 5 Program of difference method (1) (unrolling factor: 2).

ングを適用していない 1 次元差分法のプログラム（アンローリング段数が 1）を示す．アンローリング段数が 1 段の場合の 1 次元差分法のプログラムには 3 つのロード，1 つのストア，2 つの浮動小数点演算が存在している．したがって，

$$Ope(1) = \{\ \text{ld}\ a_i \quad \text{ld}\ a_{i+1} \quad \text{ld}\ a_{i-1} \quad \text{st}\ b_i \quad \text{fsubd}\ a_i, a_i, t \quad \text{fsubd}\ t, a_i, b_i \quad \}$$

となる．まず，3.1 節で述べたアルゴリズムを利用してアンローリング段数が 1 段の場合のスケジューリングを行ったところ，$cycle(1) = 8.0$ という結果になるため，アンローリング段数が 1 段の場合の Parallelization Effect は，

$$pe(1, N) = 8.0 \cdot \frac{N}{1} = 8N$$

となることが分かる．次にアンローリング段数が 2 段の場合のプログラムを**図 5** に示す．アンローリング段数を 1 段増やすごとに，ロード命令が 1 つ，ストア命令が 1 つ，浮動小数点演算命令が 2 つずつ増加していくことが分かる（図の下線部）．したがって，アンローリング段数が 1 段増えることによって増加する命令 $Ope(\ell)$ は，

$$Ope(k) = \{\text{ld}\ a_{i+k} \quad \text{st}\ b_{i+k-1} \quad \text{fsubd}\ a_{i+k-1}, a_{i+k-2}, t \quad \text{fsubd}\ t, a_{i+k}, b_{i+k-1} \quad \}$$

となる（ただし，$k \geq 2$）．また，各命令と直積空間 $\mathbf{N} \times \mathbf{Unit}$ との対応関係を次に示す．

$x \in Ope(k)$	$\rightarrow$	$\mathbf{N} \times \mathbf{Unit}$
ld a_{i+k}	$\mapsto$	$(k+2, LD)$
st b_{i+k-1}	$\mapsto$	$(k+\ell+6, ST)$
fsubd a_{i+k-1}, a_{i+k-2}, t	$\mapsto$	$(k+3, FA)$
fsubd t, a_{i+k}, b_{i+k-1}	$\mapsto$	$(k+\ell+3, FA)$

これらの情報を基に Unrolling Shape を計算する過程

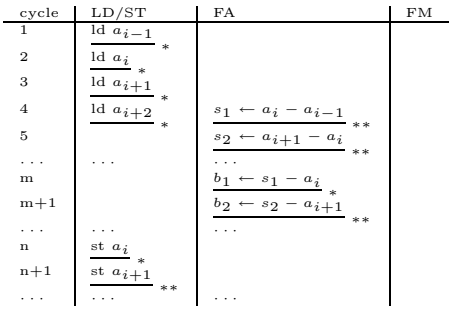

図 6 Unrolling Shape の計算過程
Fig. 6 Steps of calculating Unrolling Shape.

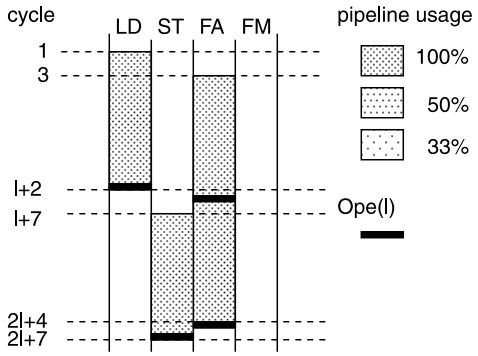

図 7 差分法 (1) における Unrolling Shape
Fig. 7 Unrolling Shape of difference method (1).

を**図 6** に示す．まず，$Ope(1)$ が図中 $*$ に示す部分にスケジューリングされる．$Ope(1)$ だけは他の $Ope(k)$ と比較して ld 命令が 2 つ多い．次に $Ope(2)$ が図中 $**$ に示す部分にスケジューリングされる．以降，$Ope(3), Ope(4), \ldots Ope(\ell)$ をスケジューリングしていくと，最終的には**図 7** のようになる．ここで，LD はロード，ST はストア，FA は浮動小数点の加減算，FM は積除算の各命令列を指す．□に囲まれた領域には命令が存在していることを示し，塗りつぶしの濃さが命令の飽和度を示している（この図の場合は，どの部分も飽和状態である）．また，■の部分はアンローリング段数が $\ell - 1$ 段から ℓ 段に増えた場合に増加する命令列 $Ope(\ell)$ が配置される場所を示している．これによって 1 反復にかかるサイクルは $cycle(\ell) = 2\ell + 7$ となることが分かるため，アンローリング段数が ℓ 段の場合の Parallelization Effect は，

$$pe(\ell, N) = (2\ell + 7) \cdot \frac{N}{\ell} = \left(2 + \frac{7}{\ell}\right) N$$

となることが分かる．

次に Memory Effect について考える．キャッシュのラインサイズが 16 byte，扱う浮動小数点は 8 byte であり，どの変数も連続アクセスであることから，キャッ

cycle	LD	FA	FM
1	ld a_{i-1}		
2	ld a_i		
3	ld a_{i+1}		
4	ld a_{i+2}	$s_1 \leftarrow a_i - a_{i-1}$	
5	ld a_{i+3}	$s_2 \leftarrow a_{i+1} - a_i$	
6	ld a_{i+4}	$s_3 \leftarrow a_{i+2} - a_{i+1}$	
...	...	...	
$k+1$	ld a_{i+k-1}	$s_{i+k-2} \leftarrow a_{i+k-3} - a_{i+k-4}$	
$k+2$	ld a_{i+k}	$s_{i+k-1} \leftarrow a_{i+k-2} - a_{i+k-3}$	
$k+3$	ld a_{i+k+1}	$s_{i+k} \leftarrow a_{i+k-1} - a_{i+k-2}$	
$k+4$	ld a_{i+k+2}	$s_{i+k+1} \leftarrow a_{i+k} - a_{i+k-1}$	
...	...	...	

distance:
$p(\text{ld } a_{i+k-1}) - p(s_{i+k+1} \leftarrow a_{i+k} - a_{i+k-1}) = 3$
$p(\text{ld } a_{i+k}) - p(s_{i+k+1} \leftarrow a_{i+k} - a_{i+k-1}) = 2$

図 8 差分法 (1) のロードと参照の距離
Fig. 8 Distance between load and reference in difference method (1).

シュミスが生じる確率は変数にはよらず一定となり，$m_{L1}(x)$ は，

$$m_{L1}(x) = \frac{8}{16} = 0.5$$

となる．また，値のロードとその値の参照の距離を考えるために Unrolling Shape の中身を見てみると，プログラムのロードと参照の関係は**図 8** に示すようになっている．最初の a_{i-1} と a_i は 1 つ前の反復でロードされている値であるから，ループの第 1 番目の反復以外では考慮する必要はない．また，一番最後にロードされる変数（$a(i+\ell+1)$）は値のロードと参照の距離が十分に離れることになる．その他の変数ではロードと参照の対応関係は同じパターンとなっており，図中の下線で示されている．これを見るとロードと参照の距離が 2 あるいは 3 で一定であることが分かるため，平均をとれば $c_{L1}(\ell, x) = 8 - 2.5 = 5.5$ となる（ここでは，この値はアンローリング段数によらない）．以上のことから Memory Effect はアンローリング段数によらず一定で，

$me(\ell, N) = (2+(N-1))\cdot 0.5 \cdot 5.5 = 2.75(N+1)$

となる．したがって，$UE(\ell, N)$ は，

$$UE(\ell, N) = \frac{8N + 2.75(N+1)}{(2+\frac{7}{\ell})N + 2.75(N+1)} = \frac{10.75N + 2.75}{(4.75+\frac{7}{\ell})N + 2.75}$$

という結果になる．N を十分大きいと仮定すれば Unrolling Effect は ℓ だけの関数となり，

$$UE(\ell) = \frac{10.75}{4.75 + \frac{7}{\ell}}$$

となる．この式に対して極限をとることで，アンローリングを適用することで最大どの程度の効果を得ることができるかを予測することが可能である．アンローリングの効果の上限 $UE_{\max}$ は，

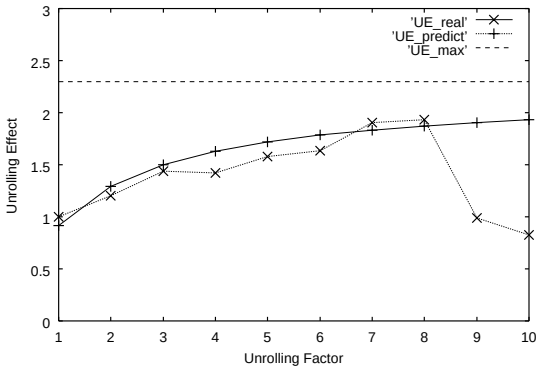

図 9 差分法 (1) の Unrolling Effect
Fig. 9 Unrolling Effect of difference method (1).

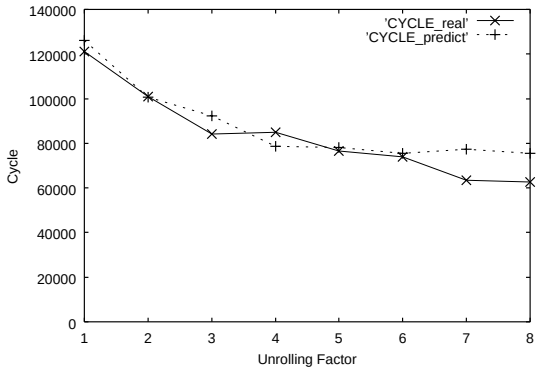

図 10 差分法 (1) の実行サイクル
Fig. 10 Execution cycle of difference method (1).

$$UE_{\max} = \lim_{\ell \to \infty} UE(\ell) = \frac{10.75}{4.75} \approx 2.3$$

となることから，アンローリングをしない場合と比較して最大で 2.3 倍高速になることが予測できる．Unrolling Effect の実測値と予測値のグラフを**図 9** に示す．グラフは横軸にアンローリング段数をとり縦軸に Unrolling Effect をとったもので，グラフ中の UE_real が実測値，UE_predict が予測値，UE_max がアンローリングの効果の上限を示している．実際の場合では，アンローリング段数が 8 段での 1.93 倍が最大でそれ以降は性能が低下している．これはアンローリング段数が 9 段以降の場合はレジスタ数が足りなくなってスピルコードが生成されてしまうからであり，Unrolling Effect の予測値が実際にあてはまるのはアンローリング段数が 8 段までとなる．実測値の 1.93 倍と予測値の最大 2.3 倍という値を比較すれば，精度良く予測できているといえる．

次に各アンローリング段数におけるサイクルの予測値と実測値を**図 10** に示す．この予測値は各アンローリング段数のアセンブラコードを基にスケジューリン

```
1:  do i = 1, n
2:    a(i) = a(i) − a(i − 1) − a(i + 1)
3:  enddo
```

図 11 1 次元差分法 (2) のプログラム（アンローリング 1 段）
Fig. 11 Program of difference method (2) (unrolling factor: 1)

グを行って得たものであり，図 9 で示したグラフとは振舞いが異なる．実測と予測の間の誤差の最大は 21%であり，振舞いを予測できているということができる．アンローリング段数が 7，8 段の場合には予測と実測に差があるが，これは先に述べた理由による．

4.3 1 次元差分法 (2)

この例題も 1 次元差分法であるが，4.2 節とは異なる性質のプログラムである．**図 11** にアンローリングを適用していない場合のソースを示す．ロード /ストアおよび浮動小数点演算の個数は 4.2 節に示した例と変わらないが，左辺と右辺が同じ配列のためループをまたぐ依存関係が生じている部分が異なっている．この制約によって演算の順序を入れ替えることができなくなるため，演算パイプラインを飽和状態にするのは難しい例題であることが分かる．まず，アンローリング段数が 1 段の場合のスケジューリングを行うと，$cycle(1) = 8.0$ という結果になったため，

$$pe(1, N) = 8.0 \cdot \frac{N}{1} = 8N$$

となる．また，ある反復でストアした $a(i)$ の値を次の反復ですぐロードし直すために read after write hazard が発生する．UltraSparc ではストア命令が実行されてから実際にデータがメモリに書き込まれるまで 5 サイクルを必要とする[10] ため，修正項としてコスト $\epsilon(1, N) = 5N$ を追加する．アンローリング段数を 1 段増やすごとに新たに増える命令は，ロード命令 1 つ，ストア命令 1 つ，浮動小数点演算 2 つと 4.2 節で示した例と変わらないため，$Ope(\ell)$ は，

$$\begin{aligned} Ope(k) = \{&\text{ld } a_{i+k} \quad \text{st } a_{i+k-1} \\ &\text{fsubd } a_{i+k-1}, a_{i+k-2}, tmp_k \\ &\text{fsubd } tmp_k, a_{i+k}, a_{i+k-1} \quad \} \end{aligned}$$

となる．このことから Unrolling Shape を求めてみると，**図 12** のようになる．図中の■は $Ope(\ell)$ が配置される場所であり，前の例題とパターンが違うことが分かる．この図において，ロードとストアの命令が配置されている領域での命令の飽和度は 100%であるが，浮動小数点演算の領域では 33%と，命令パイプラインにゆとりがある状態となる．この図より 1 反復にかかるサイクルは $(6\ell + 3)$ であるから，

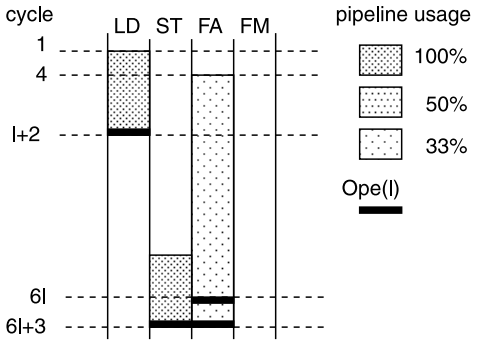

図 12 差分法 (2) における Unrolling Shape
Fig. 12 Unrolling Shape of difference method (2).

cycle	LD	FA	FM
1	ld a_{i-1}		
2	ld a_i		
3	ld a_{i+1}		
4	ld a_{i+2}	$s \leftarrow a_i - a_{i-1}$	
5	ld a_{i+3}		
6	ld a_{i+4}		
7	ld a_{i+5}	$t_1 \leftarrow s - a_{i+1}$	
8	ld a_{i+6}		
9	ld a_{i+7}		
10	ld a_{i+8}	$s \leftarrow a_{i+1} - t_1$	
11	ld a_{i+9}		
12	ld a_{i+10}		
13	ld a_{i+11}	$t_2 \leftarrow s - a_{i+2}$	
14	ld a_{i+12}		
15	ld a_{i+13}		
16	ld a_{i+14}	$s \leftarrow a_{i+2} - t_2$	
...	...	...	
$k+2$	ld a_{i+k}	...	
...	...	...	
$6k+1$		$t_k \leftarrow s - a_{i+k}$	
...	...	...	

distance:
$p(\text{ld } a_{i+k}) - p(t_k \leftarrow s - a_{i+k}) = (6k+1) - (k+2) = 5k - 1$

図 13 差分法 (2) のロードと参照の距離
Fig. 13 Distance between load and reference in difference method (2).

$$pe(\ell, N) = (6\ell + 3) \cdot \frac{N}{\ell} = \left(6 + \frac{3}{\ell}\right) N$$

となることが分かる．次にキャッシュミスによるコストを計算するために値のロードとその値の参照の距離を考える（**図 13**）．このプログラムでは浮動小数点演算のパイプラインが疎の状態になってしまうというデメリットが存在するが，それはロードと演算の距離を離すというメリットを生んでいる．$a(i-1)$ および $a(i)$ は 1 つ前の反復でロードされているからこれらは第 1 番目の反復でだけキャッシュミスのコストを考える必要があり，そのコストの平均は $c_{L1}(\ell, x) = 5.5$ である．その他の変数ではロードと参照の対応関係は同じパターンとなっており，図中の下線で示されている．この図より分かるとおり $a(i+k)$ がロードされてから初めて参照されるまでの距離は $5k - 1$ であることから，$k \geq 2$ の場合にはキャッシュミスのコストは完全に隠蔽されることとなり，$k = 1$ でのコストだけを

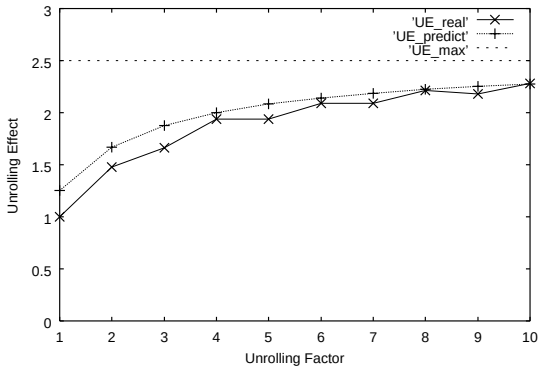

図 14 差分法 (2) の Unrolling Effect
Fig. 14 Unrolling Effect of difference method (2).

考えればよい．したがって，1 反復あたりのキャッシュミスのコストは $(8-4)\cdot 0.5$ となる．$m_{L1}(x)$ の値は前の例題と同じく 0.5 であるから，Memory Effect の値 $me(\ell, N)$ は，

$$me(\ell, N) = 2\cdot 0.5\cdot 5.5 + 0.5\cdot 4\cdot \frac{N}{\ell} = 5.5 + \frac{2}{\ell}N$$

となる．したがって，$UE(\ell, N)$ は，

$$UE(\ell, N) = \frac{8N + (5.5 + 2N) + 5N}{(6+\frac{3}{\ell})N + 5.5 + \frac{2N}{\ell}} = \frac{15N + 5.5}{(6+\frac{5}{\ell})N + 5.5}$$

という結果になる．N を十分大きいとすれば，Unrolling Effect は ℓ だけの関数となり，

$$UE(\ell) = \frac{15}{6+\frac{5}{\ell}}$$

となる．この式に対して極限をとることで，アンローリングを適用することで最大どの程度の効果を得ることができるかを予測することが可能である．よってアンローリングの効果の上限 $UE_{\max}$ は，

$$UE_{\max} = \lim_{\ell\to\infty} UE(\ell) = \frac{15}{6} = 2.5$$

となることから，アンローリングをしない場合と比較して最大 2.5 倍高速になることが予測できる．実際の性能向上比と予測値のグラフを**図 14** に示す．グラフをみて分かるとおり，実測値と予測値の結果がほぼ一致しており，精度の良い予測ができているといえる．これは，演算処理の時間（Parallelization Effect）だけでなくキャッシュミスによるコスト（Memory Effect）の予測がうまくいっていることを意味していると考えられる．この例から分かるとおり，キャッシュミスの予測は非常に重要である．

4.4 考　　察

考察を行ううえで，例題であげた 2 つの特徴を**表 1** に示す．表を見て分かるとおり，同じ差分法のプログラムであってもその特徴は対照的で，この 2 つを比較する意義は大きいことが分かる．Parallelization Effect については (1) の方が (2) に比べておよそ 1/3 で済んでいる．逆に Memory Effect を考えると，(2) はアンローリング段数が増えれば減少するのに対し，(1) はアンローリング段数が変化しても変わらない．

表 1 2 つの例題の比較
Table 1 Comparison of examples.

	1 次元差分法 (1)	1 次元差分法 (2)
$pe(\ell, N)$	$(2+\frac{7}{\ell})N$	$(6+\frac{3}{\ell})N$
$(\ell\to\infty)$	$2N$	$6N$
$me(\ell, N)$	$2.75 + 2.75N$（一定）	$5.5 + \frac{3}{\ell}N$（減少）
$(\ell\to\infty)$	$2.75 + 2.75N$	5.5

ループアンローリングの効果を予測するために実行サイクルを予測するが，その予測には演算の処理時間（Parallelization Effect）の予測とキャッシュミスによるコスト（Memory Effect）の予測の 2 つが必要で，どちらか 1 つが欠けても予測はうまくいかない．従来は命令の並列度だけに注目してソフトウェアパイプライニングやループアンローリングの効果を予測していたため，実際にそれらの手法を適用した場合に生じる現象を説明できない場合が多かった．キャッシュミスのコストを正確に予測することで，今まで説明できなかった現象を説明できるようにしている．キャッシュミスのコストの予測で重要なのは，単にロードの個数とキャッシュミスのレイテンシだけを考慮するのではなく，値がロードされてから参照されるまでの距離を考えることである．この距離が十分離れることでキャッシュミスのコストを隠蔽することが可能なことから，命令の並列度が十分に上がらない場合でもループアンローリングの効果が十分に得られるケースは多数存在する．

また，アンローリングの効果の極限を計算することで，アンローリングによる効果の上限を計算している．実際に本モデルを基に最適化を行う場合にはスケジューリングというコストのかかる作業が入ってくるため，アンローリングの極限を考えて上限を抑えるということは最適化を適用するかどうかを判断するための基準として利用することができる．Unrolling Effect を計算するうえでキャッシュミスの影響は大きいことから，1 次キャッシュミスに加えて 2 次キャッシュミスが頻発するようなプログラムでは $me(\ell, N)$ の値がさらに大きくなって Unrolling Effect の値を小さくしてしまい，場合によってはアンローリングの効果を打ち消してしまう可能性もある．

5. 関連研究

ループアンローリングを行うことで効果があることは知られている[3]ことから，アグレッシブにループアンローリングを行うという研究[2]が以前から行われている．ループアンローリングやソフトウェアパイプライニングといった最適化に影響を与えるものとして，レジスタアロケーションや命令スケジューリングのアルゴリズムと，それらにともなう命令並列度の増加やレジスタプレッシャーによる制約がある[4),6)~8),11)]．これらの議論を基に，アンローリングの効果を CPI 値に基づいて予測するという研究[1]が存在するが，本研究では，命令の並列度だけではなくメモリに関してキャッシュミスの予測が重要であることを述べた．メモリの最適化に関してはブロッキングによるアクセスパターンの最適化[5]や，プリフェッチによるロードのコストの隠蔽[9]があるが，これらのメモリ最適化は単独で議論するのではなく，先に述べたループアンローリングやソフトウェアパイプライニングとあわせて議論を行うことが重要である．

6. まとめと今後の課題

本稿では，まず 2 章でモデル化を行ううえでの前提条件について述べた．対象とするプロセッサは典型的な RISC 系のプロセッサで，レジスタアロケーションと命令スケジューリングを固定することが重要である．次に 3 章でモデル化について述べ，ループアンローリングの効果を議論するための基盤として Parallelization Effect，Memory Effect を定義し，それらの値を用いて Unrolling Effect を定義した．4 章では，2 種類の異なる性質を持つプログラムを使用して例を示し，精度の良い Unrolling Effect の予測ができていることを示した．演算に要する時間の予測（Parallelization Effect）とキャッシュミスによるコスト（Memory Effect）の予測という 2 つの要素をあわせて用いることで Unrolling Effect の予測を精度良く行うことができる．

今後の課題は，各アンローリング段数における性能向上比（UE）の精度を向上させることである．この予測がどの程度一致するかは Unrolling Shape を精度良く予測できるかどうかに依存するため，予測を行うための厳密なアルゴリズムを開発することが今後の研究課題である．

謝辞 この研究は一部日本学術振興会未来開拓学術研究推進事業「分散・並列スーパーコンピューティングのソフトウェアの研究」（JSPS-RFTF 00505）の補助を受けた．また日頃の研究にあたり有益な助言をいただいている東京大学金田康正教授に感謝します．

参考文献

1) Bose, P., Kim, S., O'Connell, F.P. and Ciarfella, W.A.: Bounds modelling and compiler optimizations for superscalar performance tuning, *Journal of Systems Architecture*, Vol.45, No.12-13, pp.1111–1137 (1999).
2) Davidson, J.W. and Jinturkar, S.: An Aggressive Approach to Loop Unrolling, Technical Report CS-95-26, Department of Computer Science, University of Virginia (1995).
3) Dongarra, J. and Hinds, A.: Unrolling Loops in FORTRAN, *Software Practice and Experience*, Vol.9, pp.219–229 (1979).
4) Hennessy, J.L. and Patterson, D.A.: *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc, San Mateo, CA (1990).
5) Lam, M.S., Rothberg, E.E. and Wolf, M.E.: The Cache Performance and Optimizations of Blocked Algorithms, *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.63–74 (1991).
6) Lee, R.L., Kwok, A.Y. and Briggs, F.A.: The Floating-Point Performance of a Superscalar SPARC Processor, *ACM SIGPLAN Notices*, Vol.26, No.4, pp.28–37 (1991).
7) Llosa, J., Valero, M. and Ayguade, E.: Register Requirements of Pipelined Loops and their Effect on Performance, *Proc. 2nd International Workshop on Massive Parallelism: Hardware, Software and Applications MP94*, pp.173–189 (1994).
8) Motwani, R., Palem, K.V., Sarkar, V. and Reyen, S.: Combining Register Allocation and Instruction Scheduling, Technical Note CS-TN-95-22, Stanford University, Department of Computer Science (1995).
9) Mowry, T.C., Lam, M.S. and Gupta, A.: Design and Evaluation of a Compiler Algorithm for Prefetching, *Proc. 5th International Conference on Architectual Support for Programming Language and Operating Sytems*, pp.62–75 (1992).
10) Sun microsystems: *UltraSPARC^{TM} User's Manual* (1997).
11) Weiss, S. and Smith, J.E.: A Study of Scalar Compilation Techniques for Pipelined Supercomputers, *Proc. 2nd International Conference on Architectural Support for Programming*

Languages and Operating Systems, pp.105–109 (1987).

(平成 12 年 10 月 25 日受付)
(平成 13 年 2 月 20 日採録)

吉田 映彦（学生会員）

1977 年生．1998 年沼津工業高等専門学校制御情報工学科卒業．2000 年東京農工大学工学部電子情報工学科卒業．現在，東京大学大学院理学系研究科情報科学専攻修士課程在学中．コンパイラ，特にループ最適化に関する研究に興味を持つ．

佐藤 周行（正会員）

1962 年生．1985 年東京大学理学部情報科学科卒業．1990 年同大学院博士課程修了．理学博士．現在，東京大学助教授（情報基盤センター）．プログラム意味論，並列化コンパイラの研究に従事．