ハイパフォーマンス
コンピューティング 62－12
（1996. 8. 28）

資源制約を考慮した 命令の実行タイミングの解析とその応用

小野田 亘利† 李 鼎超‡ 石井 直宏†

†名古屋工業大学知能情報システム学科
‡名古屋工業大学情報処理教育センター
†E-mail : {onobu, ishii}@egg.ics.nitech.ac.jp
‡E-mail : liding@center.nitech.ac.jp

命令レベルでの並列実行が可能な計算機においては，計算機性能を最大限に引き出すためのコンパイラ技術が重要である．あらかじめプログラム中の各命令の実行タイミングを予測することが可能であれば，その解析情報を各種最適化において利用することによって，より効率の良いコードの生成が可能となる．本稿では，プログラムの中の命令の実行タイミングの解析手法について述べるとともに，その応用例としてプログラム解析情報をもとにした，コードスケジューリングとレジスタ割り付け時におけるスピルコードの挿入条件を示す．

Execution-Timing Analysis of Instructions under Resource Constraints and its Application

Nobutoshi Onoda† Dingchao Li‡ Naohiro Ishii†

†Department of Intelligence and Computer Science, Nagoya Institute of Technology
‡Educational Center for Information Processing, Nagoya Institute of Technology
†E-mail : {onobu, ishii}@egg.ics.nitech.ac.jp
‡E-mail : liding@center.nitech.ac.jp

Many of optimizations either reply on or can benefit from information about the program execution behavior. In this paper, we discuss how to obtain and use such information so as to predict the execution-timing of instructions. In particular, we present the condition for making the decision regarding which variables to spill, based on the use of the estimation technique developed so far.

1 はじめに

最適化コンパイラにおける重要な技術の一つとしてレジスタ割り付けがある．現代の計算機においても，実装可能なレジスタ数は様々な制約から制限されている．特に命令レベルでの並列実行が可能な計算機においては，プログラムの持つ限られた並列性を有効に利用するためにも，より多くのレジスタが必要とされている．

従来の最適化手法では，コードスケジューリングとレジスタ割り付けの最適化は別々に行なわれていたために，スケジューリングによって得られた並列性をレジスタ割り付けによって損なう場合が存在した．

本稿では，コードスケジューリングとレジスタ割り付けを同時に考慮するための手段として，プログラム実行時における命令の実行タイミングの解析手法について提案を行なう．この実行タイミングの解析情報から，プログラム実行時における各実行区間での命令の先行制約や機能ユニット不足による遅延コスト，使用レジスタ数などを予測することが可能となる．また，この解析情報を利用したレジスタ不足時におけるスピルコードの挿入条件を示す．これによって，スピルコードの最適な挿入タイミングの計算が可能となり，スピルコード挿入時におけるプログラムの実行遅延の最小化が可能となる．

以下，2章では関連研究について述べる．3章では今回対象とする計算機モデルとプログラムモデルについての定義を行なう．4章では命令の実行タイミングの解析手法とその情報をもとにした変数生存区間の解析手法について述べる．5章では実行タイミング情報をもとにしたスピルコードの挿入条件について述べる．6章ではスピルコードの挿入条件の計算例を挙げる．7章ではまとめを行なう．

2 関連研究

コードスケジューリングとレジスタ割り付けに関する最適化については，Goodman と Hsu[1] が行なっている．これはプログラム中のデータ依存解析結果をもとにレジスタ割り付けを行なうものである．

Natarajan と Schlansker[2] は，利用可能なレジスタ数の下界の情報をもとにした，スケジューリング長を最小化するヒューリスティックなスケジューリング手法を提案している．

また最近では，小松ら [3] はプログラムの持つ並列性を損なうことなくレジスタ割り付けを行なう技法を提案している．これはプログラムを GPDG と呼ばれるグラフ形式に変換したのちに，その上でクリティカルパスが伸びないようレジスタ割り付けを行なうものである．

これらの研究とは異なり，本稿では命令の実行タイミングの解析によって得られる各タイミングにおける利用可能な資源情報を利用した，スピルコードの挿入条件を示す．これによって，プログラムの実行遅延を最小に抑えるようなタイミングでのスピルコードの実行タイミングを算出することが可能となる．

3 計算機モデルとプログラムモデル

3.1 計算機モデル

本稿で対象とする計算機は，命令レベルでの並列実行が可能な非均質機能ユニットを有するものとする．各機能ユニットごとに実行可能な命令が制限されており，各命令ごとに実行サイクル数が異なるものとする．各機能ユニットには，浮動小数点演算ユニット (FU)，整数演算ユニット (IU)，ロード・ストアユニット (Load/Store) 等のタイプが存在するものとする．

ここで計算機を P とする．Pに実装されている k タイプの機能ユニットの番号を i とすると，P_k^iは，$\{P_k^i | 1 \leq k \leq s, 1 \leq i \leq m_k\}$ で表現される．ここで，s は機能ユニットのタイプの数を示し，m_kは P に実装されている kタイプの機能ユニットの数である．また，kタイプの機能ユニット上では，kタイプの命令しか実行できないものとする．

通常レジスタは，浮動小数点レジスタ，整数レジスタ，アドレスレジスタなどがあり，レジスタごとに格納可能なデータの種類が制限されている．ここでは，モデルを簡略化するために，P_k^iは各機能ユニットからアクセス可能な共有レジスタ $R = \{r_1, \cdots, r_l\}$ を有するものとする．lは実装されているレジスタ数である．

3.2 プログラムモデル

ここで対象とするプログラムは，トレーススケジューリング [4] などの手法を用いて，プログラム中から選択された一つのプログラムトレースとする．プログラムの並列性を阻害する逆依存や出力依存関係は，プログラムを SSA 形式 [5] に変換することによって除去する．したがって，各変数は 1 つの命令によって 1 回のみ定義され，他の命令より 1 回以上参照されるものとする．

SSA 形式に変換後，プログラムを有向無サイクルグラフ (DAG) 形式 $G = (\Gamma, A, \mu, \nu)$ で表現する．ここで G は，命令 I_i $(i = 1, 2, \cdots, n)$ の集合である．A は各命令間の実行順序関係を示し，$\mu(I_i)$ は命令 I_iの実行時間 (サイクル) を示す．$\nu(I_i)$ は各命令のタイプを示し，$\nu(I_i) = k$を満たすとき，kタイプの機能ユニット上でのみ I_iが実行されるものとする．

DAG において，(I_i, I_j) 間にアークが存在する時，I_jの実行に I_iの実行結果が必要であることを意味する．ここで，I_iの後続命令の集合を $Succ(I_i)$ とし，I_i の先行命令の集合を $Pred(I_i)$ とする．また DAG は必ず1つの入口ノード I_1と，1つの出口ノード I_nを有するものし，DAG に存在する全ての命令は，I_1の後に実行され，かつ I_nの前にすべて実行されるものとする．

4 実行タイミングの解析

DAG によって表現されたプログラムに対して，実行時における各命令の実行タイミングの解析を行なう手法について述べる．実行タイミングの解析時において，各タイミングにおける機能ユニット不足から生じる実行遅延を考慮することによって，より正確な実行タイミングの解析が可能となる．また，この解析結果を利用した変数生存区間の解析手法についても述べる．

4.1 先行制約を考慮した実行タイミング解析

DAG によって表現されたグラフ G 中の命令 I_iについて考える．I_iの実行タイミングが G 中に存在する先行制約によってのみ決定されるとする．このとき，I_iの計算に必要な値がすべて揃い，I_i が最も早く実行開始可能となる実行タイミングを$\tau_{es}(I_i)$ とする．また，G 中のクリティカルパス長を t_{cp}としたとき，t_{cp}を増加させることなく，I_iの実行を最も遅らせた時の実行開始タイミングを$\tau_{ls}(I_i)$ とする．これは，I_iが実行区間 $[\tau_{es}(I_i), \tau_{ls}(I_i)+\mu(I_i)]$ の間で実行できれば，t_{cp}の増加は生じないことを意味する．ここで，$\tau_{es}(I_i)$=$\tau_{ls}(I_i)$ を満たす I_iは，実行を遅らすことのできないクリティカルな命令である．

I_1から I_iに存在する k通りのパス上の命令の集合をπ_k，I_nから I_iに存在する k通りのパス上の命令の集合を$\bar{\pi}_k$とすると，$\tau_{es}(I_i)$，$\tau_{ls}(I_i)$ は，

$$\tau_{es}(I_i) = \max_{k} \sum_{I_j \in \pi_k} \mu(I_j)$$

$$\tau_{ls}(I_i) = \min_{k} \{t_{cp} - \sum_{I_j \in \bar{\pi}_k} \mu(I_j)\}$$

で求めることができる．また$\tau_{es}(I_i)$で I_iを実行したときの実行終了タイミングを$\tau_{ef}(I_i)$=$\tau_{es}(I_i)$ + $\mu(I_i)$，$\tau_{ls}(I_i)$ で I_iを実行したときの実行終了タイミングを$\tau_{lf}(I_i)$=$\tau_{ls}(I_i)$ + $\mu(I_i)$ とする．

4.2 資源制約を考慮した実行タイミング解析

より正確な実行タイミングを求めるには，機能ユニット不足から生じる実行遅延を考慮しなければならない．実行遅延とは，各依存関係と資源制約から生じる，命令間に存在する必要な実行待ち時間を指す．計算機の資源が無限にあるとするとき，I_iと I_j間に存在する実行遅延は，I_iと I_j間に存在するデータ依存関係によってのみ決定される．この時，I_jの実行に先行して I_iの実行が完了していなければならない時，実行遅延 $d_p(I_i, I_j)$ は，

$$d_p(I_i, I_j) = \tau_{es}(I_j) - \tau_{lf}(I_i)$$

と求められる．これは一つの実行遅延の下界値であるが，より正確な実行遅延の下界として，非均質機能ユニットから生じる実行遅延を考慮した，I_iと I_j間に存在する実行遅延 $d_r(I_i, I_j)$ の算出方法を示す．

まず $Succ(I_i)$ と $Pred(I_j)$ より，I_iの実行完了からI_jの実行開始までに実行しなければならない命令の集合Π_{ij}を求める．つぎに機能ユニットの非均質性を考慮するために，Π_{ij}を各命令タイプ k=$(1, 2, \cdots, s)$ ごとに部分集合Π_{ij}^k $(\Pi_{ij}^1, \Pi_{ij}^2, \cdots, \Pi_{ij}^s)$ に分割する．$let(\Pi_{ij}^k)$ をΠ_{ij}^kの実行に最低限必要な実行時間とすると，$d_r(I_i, I_j)$ は，

$$d_r(I_i, I_j) = \max_{1 \le k \le s} \{let(\Pi_{ij}^k)\}$$

で求めることができる．いま，Π_{ij}^kにおいて並列実行可能な命令が存在した時に，遅延 $d_p(I_i, I_j)$ に対し，kタイプの機能ユニットの不足によって生じる遅延の増分を $\lceil \delta_{ij}^k \rceil$ で示すとき，$let(\Pi_{ij}^k)$ は，

$$let(\Pi_{ij}^k) = d_p(I_i, I_j) + \lceil \delta_{ij}^k \rceil$$

で求めることができる．Π_{ij}^kの実行が可能な機能ユニットが m_k台あるとすると，$\lceil \delta_{ij}^k \rceil$ は文献 [6] より，

$$\delta_{ij}^k = \max_{[\theta_1, \theta_2]} [-(\theta_1 - \theta_2) + \frac{1}{m_k} \sum_{I_p \in \Pi_{ij}^k} \mu(\theta_1, \theta_2, I_p)]$$

で求めることができる．$\mu(\theta_1, \theta_2, I_p)$ は，区間 $[\theta_1, \theta_2]$ で実行しなければならない I_pの実行時間である．$I_p \subset \Pi_{ij}^k$を実行しなければならない区間は $[\tau_{es}(I_p), \tau_{ls}(I_p)]$ であるから，この区間についてのみδ_{ij}^kの計算を行う．

この $d_r(I_i, I_j)$ を利用して，より正確な実行タイミング$\tau_{es}(I_i)$，$\tau_{ls}(I_i)$ の再計算を行なう．図1は，非均質機能ユニット不足から生じる実行遅延を考慮した命令の実行タイミングを求めるアルゴリズムである．

4.3 変数生存区間の解析

命令の実行タイミングの解析情報を利用した変数の生存区間の解析手法について述べる．まず，変数 vが定義されるタイミングを$\tau_s(v)$，変数が最も最後に参照されるタイミングを$\tau_f(v)$ とする．このとき

```
Procedure Execution-Intervals
Begin
    /* Estimate the earlist starting and finishing times */
    Let τes(I1) = 0 and τef(I1) = μ(I1)
    For each task Ii from I1 to In-1 Do
        For each task Ij where Ij ∈ Succ(Ii) Do
            τes(Ij) = max{τef(Ii) + dr(Ii, Ij), τes(Ij)}
            τef(Ij) = τes(Ij) + μ(Ij)
        Endfor
    Endfor
    /* Estimate the latest starting and finishing times */
    Let τlf(In) = τef(In) and τls(In) = τlf(In) − μ(In)
    For each task Ii from In to I2 Do
        For each task Ij where Ij ∈ Pred(Ii) Do
            τlf(Ij) = min{τls(Ii) − dr(Ii, Ij), τlf(Ij)}
            τls(Ij) = τlf(Ij) − μ(Ij)
        Endfor
    Endfor
End
```

図 1: 実行タイミングを求めるアルゴリズム

変数 vの生存区間を $[\tau_s(v), \tau_f(v)]$ で示すとする．いま，vの値を定義する命令を I_iとし，vの値を最も最後に参照する命令を I_jとする．変数 vの生存区間は実行タイミングの解析情報を利用して，

$$\tau_{ef}(I_i) \leq \tau_s(v) \leq \tau_{lf}(I_i), \tau_{es}(I_j) \leq \tau_f(v) \leq \tau_{lf}(I_j)$$

と求めることができる．

各変数の生存区間を求めることによって，ある実行区間 $[\theta_1, \theta_2]$ 内で使用されている変数の数を求めることができる．$v(\theta_1, \theta_2, I_i)$ を区間 $[\theta_1, \theta_2]$ において I_iの実行に必要な変数の集合とする．区間 $[\theta_1, \theta_2]$ 内で生存する変数集合 $V(\theta_1, \theta_2)$ は，

$$V(\theta_1, \theta_2) = \bigcup_{i=1}^{n} v(\theta_1, \theta_2, I_i)$$

で求めることができる．このとき，I_iは $\mu(\theta_1, \theta_2, I_i) \neq 0$ を満たすものとする．この条件を満たさない I_iは，区間 $[\theta_1, \theta_2]$ において実行する必要のない命令である．

$|V(\theta_1, \theta_2)|$ を区間 $[\theta_1, \theta_2]$ 内で生存している変数の数としたとき，$|V(\theta_1, \theta_2)|$ は区間 $[\theta_1, \theta_2]$ 内で必要なレジスタ数を示す下界値である．本稿では，この値をスピルコードの挿入条件の一つとして利用する．

5 スピルコードの挿入条件

命令レベルでの並列実行機能を有しない計算機では，スピルコードの挿入によるレジスタの再利用は，プログラム実行時間の増大を招くため，極力さけるものとされてきた．しかしながら，命令レベルでの並列実行が可能な計算機においては，スピルコードを並列に実行することが可能である．

ここでは，命令の実行タイミングの解析情報を利用したスピルコードの挿入による遅延コストとスピルコードの挿入タイミングについて述べるとともに，スピルコードの挿入条件を示す．

5.1 レジスタ再利用による遅延コスト計算

命令 I_u がコードスケジューリングの対象として選択されたとする．このとき，I_u に割り当てるレジスタが不足したときに，再利用可能なレジスタが生じるまでスケジューリングを行なわない (レジスタ再利用) とする．$\tau_f(r)$ をレジスタ rの使用が完了するタイミングとすると，レジスタ再利用による I_uの実行開始タイミング $\tau_s^{reuse}(I_u)$ は，

$$\tau_s^{reuse}(I_u) = \min_{r \in R}\{\tau_f(r)\}$$

となり，t_{cp} に対する実行遅延コスト $Cost_{reuse}$は，

$$Cost_{reuse} = \tau_s^{reuse}(I_u) - \tau_{ls}(I_u)$$

となる．

5.2 スピルコードによる遅延コスト計算

I_uのレジスタ割り付けをスピルコードの挿入によって行なうものとする．スピルコードは，レジスタの値を退避させるためのストア命令と退避した値を再びレジスタに戻すためのロード命令の 2 命令から構成される．スピルコストの計算は $r \in R$を満たす，すべての rについて行なう．以下，ストア，ロード命令挿入時の遅延コストについて述べる．

5.2.1 ストア命令挿入による遅延コスト計算

レジスタ r が最後に参照されたタイミングを $\tau_f^{use}(r)$ とし，Load/Store ユニットの使用が完了したタイミングを $\tau_f(Load/Store)$ とする．ストア命令挿入による I_uの実行開始タイミング $\tau_s^{spill}(I_u)$ は，

$$\tau_s^{spill}(I_u) = \max\{\tau_f(Load/Store), \tau_f^{use}(r)\} + \mu(Store)$$

となり，t_{cp}に対する遅延コスト $Cost_{spillout}(r)$ は，

$$Cost_{spillout}(r) = \tau_s^{spill}(I_u) - \tau_{ls}(I_u)$$

となる (図 2)．以上より，スピルコードの挿入によって抑えることのできる t_{cp}に対する増加コストは，$Spilling\_gain = \tau_s^{spill}(I_u) - \tau_s^{reuse}(I_u)$ で求めることができる．

これは，$Spilling\_gain \leq 0$ ならばレジスタ再利用をすべきであり，$Spilling\_gain > 0$ ならばスピルコードの挿入を検討することを意味する．このときに限り，ロード命令挿入による遅延コストの計算を行なう必要が生じる．

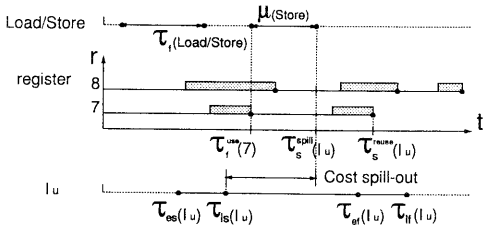

図 2: $Cost_{spillout}(r)$ の計算

5.2.2 ロード命令挿入による遅延コスト計算

まず，スケジューリングされていないすべての命令 I_i に対して，ストア命令挿入によるプログラムの実行タイミングの再計算を行なう．ストア命令挿入時の I_i の実行開始タイミングをそれぞれ，$\tau_{es}^{spill}(I_i), \tau_{ls}^{spill}(I_i)$ とすると，

$$\tau_{es}^{spill}(I_i) = \begin{cases} \tau_{es}(I_i) + (\tau_s^{spill}(I_u) - \tau_{es}(I_u)) \\ \qquad\qquad \text{if } I_i \in Succ(I_u) \\ \tau_{es}(I_i) \\ \qquad\qquad \text{otherwise} \end{cases}$$

$$\tau_{ls}^{spill}(I_i) = \tau_{ls}(I_i) + Cost_{spillout}(r)$$

と再計算できる．

ストア命令によって退避したレジスタ r の値を必要とする命令を I_v とする．スピルコードの挿入による実行遅延を隠蔽するためには，ロード命令を $\tau_s^{spill}(I_u)$ から $\tau_{ls}^{spill}(I_v)$ の間で実行しなければならない．そこで，実行区間 $[\tau_s^{spill}(I_u), \tau_{ls}^{spill}(I_v)]$ における，ロード命令挿入による t_{cp} に対する遅延コスト $Cost_{spillin}(r)$ の計算を行なう．これによって遅延コストだけでなく，遅延コストが最小となるようなロード命令の実行タイミング $\tau_s(Load)$,$\tau_f(Load)$ を算出することが可能となる．今回はロード命令挿入時において空きレジスタが存在するときのみ，スピルコードの挿入を考慮するものとする．

図 3 は，$Cost_{spillin}(r)$ と $\tau_s(Load)$ の実行タイミングを求めるアルゴリズムである．

まず $\tau_s = \tau_{ls}^{spill}(I_v) - \mu(Load)$ とし，τ_s を最も遅いロード命令の挿入タイミングとする．プログラムの実行タイミングの解析情報を利用して，ロード命令の実行区間 $[\tau_s, \tau_f]$ 内における必要なレジスタ数 $|V(\tau_s, \tau_f)|$ と，機能ユニット不足から生じる Load/Store ユニットの t_{cp} に対する遅延コスト $\delta_{Load}(\tau_s, \tau_f)$ を計算する．$\tau_s(Load)$ の計算は，区間 $[\tau_s, \tau_f]$ 内において，(1) 空きレジスタが存在しない ($|V(\tau_s, \tau_f)| \geq |R|$)，(2) 使用可能なレジスタと機能ユニットが存在 ($|V(\tau_s, \tau_f)| < |R|$ and $\delta_{Load}(\tau_s, \tau_f) \leq 0$) のいずれかの条件を満たすまで行なう (図 4)．$Cost_{spillin}(r)$ は，区間 $[\tau_s^{spill}(I_u), \tau_{ls}^{spill}(I_v)]$ において，実行遅延コストが最小となるタイミングで，ロード 命令を実行したときの遅延コスト $\delta_{Load}(\tau_s, \tau_f)$ とする．したがって，(2) の条件を満たすとき，$Cost_{spillin}(r)$=0 となる．

以上より，

$$Cost_{spillout}(r) + Cost_{spillin}(r) < Cost_{reuse}$$

を満たすならば，スピルコードの挿入を行なうべきである．

```
Procedure Reload-Cost-Estimate
Begin
Cost_spillin(r) = ∞
For τ_f from τ_ls^spill(I_v) Do
    τ_s = τ_f − μ(Load)
    δ_Load(τ_s, τ_f) = μ(Load)
    For each task I_i ∈ I_unscheduled Do
        If μ(τ_s, τ_f, I_i) ≠ 0 Then
            V(τ_s, τ_f) = V(τ_s, τ_f) + {υ(θ1, θ2, I_i)}
            If ν(I_i)=Load/Store Then
            I_Load/Store(τ_s, τ_f) =
                    I_Load/Store(τ_s, τ_f) + μ(τ_s, τ_f, I_i)
            Endif
        Endif
    Endfor
    δ_Load(τ_s, τ_f) =
            I_Load/Store(τ_s, τ_f)/m_Load/Store − μ(Load)
    If |V(τ_s, τ_f)| ≥ |R|
        Exit
    Else
        If δ_Load(τ_s, τ_f) ≤ 0 Then Exit
        Cost_spillin(r) = min{Cost_spillin(r), δ_Load(τ_s, τ_f)}
 Endfor
End
```

図 3: ロード命令の遅延コストを求めるアルゴリズム

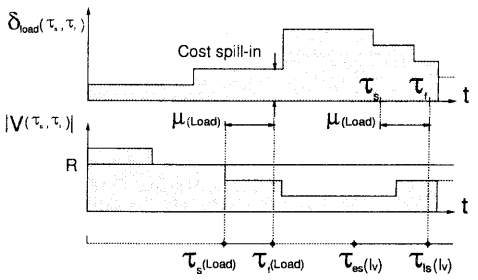

図 4: $Cost_{spillin}(r)$ の計算

6 スピルコストの計算例

図5は，あるプログラムトレース中の命令の実行タイミングの解析例である．この解析結果をもとに，

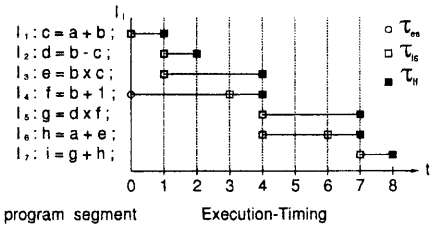

図 5: 実行タイミングの解析例

機能ユニット IU,FU,Load/Store をぞれぞれ1台ずつ有する計算機へのスケジューリングを行なう．

I_1, I_2, I_4, I_6, I_7は IU で実行可能な命令とし，実行には1サイクル必要とする．また，命令の実行開始条件として実行結果を保存するレジスタが確保されていることを前提とする．I_3, I_5は FU で実行可能な命令であり，実行には3サイクル必要とする．また，Load/Store の命令には2サイクル必要とする．また，この区間における利用可能なレジスタ数を4個と仮定する．

いま，I_1, I_2, I_3までのスケジューリングが完了し，タイミング2においてI_4のスケジューリング行なうとする．このとき，I_4の実行結果 fを保存するためのレジスタが不足する．

このときのレジスタ再利用時の遅延コスト $Cost_{reuse}$ は，$\tau_s^{reuse}(I_4) = 4$ より，$Cost_{reuse} = \tau_s^{reuse}(I_4) - \tau_{ls}(I_4) = 4 - 3 = 1$ である．また，レジスタ r_1 をスピルアウトした時の遅延コストは，$\tau_s^{spill}(I_4) = max\{0,1\} + 2 = 3$ より，$Cost_{spillout}(r_1) = \tau_s^{spill}(I_4) - \tau_{ls}(I_4) = 3 - 3 = 0$ となる．スピルイン時における遅延コストは，図4のアルゴリズムより，ロード命令の実行タイミング $[\tau_s(Load) = 4, \tau_f(Load) = 6]$ において，$|V(4,6)| < 0, \delta_{Load}(4,6) = 0$ となり，$Cost_{spillin}(r_1) = 0$ が得られる．よって，

$$Cost_{spillout}(r_1) + Cost_{spillin}(r_1) < Cost_{reuse}$$

を満たすので，スピルコードの挿入によるレジスタ割り付けを行なう (図6)．

7 まとめ

本稿では，命令の実行タイミングの解析手法を示すとともに，解析情報を利用することによって，スピルコードの挿入条件を示した．これによって，最適なスピルコードの挿入タイミングを決定することが可能となった．

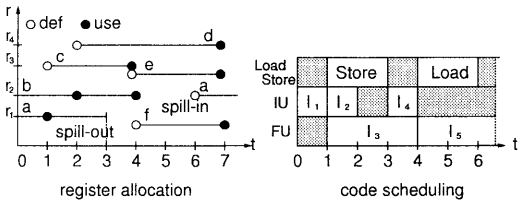

図 6: スピルコードの挿入

今後の課題としては，条件命令による実行遅延を考慮した実行タイミングの解析手法への拡張を行なうとともに，ベンチマークプログラムを利用した評価を行なっていくことが挙げられる．

参考文献

[1] J.R.Goodman, and W-C Hsu, " Code Scheduling and Register Allocation in Large Basic Blocks", *Proc. Intel. Conf. on Supercomputing*, pp. 442-452, 1988.

[2] B.Natarajan, and M.Schlansker, "Spill-Free Scheduling of Basic Blocks", *Proc. the 28th International Symposium on Microarchitecture*, pp. 119-124, 1995.

[3] 小松 秀昭, 神力 哲夫, 古関 聰, 深澤 良彰, "命令レベル並列アーキテクチャのためのレジスタ割付け技法", 情報処理学会論文誌, Vol.36, No.12, pp.2819-2830, Dec. 1995.

[4] J.R.Ellis, "Bulldog: A Compiler for VLIW Architectures", The MIT Press, (1985).

[5] Cytron R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K., "An Efficient Method of Computing Static Single Assignment Form", *Conference Record of the Sixteenth ACM Symposium on the Principles of Programming Languages*, pp.25-35, Jan. 1989.

[6] 李 鼎超, 有田 隆也, 石井直宏, 曽和 将容, "非均質並列プロセッサ用プログラムの実行時間の下界", 情報処理学会論文誌, Vol.34, No.11, pp.2378-2385, Nov. 1993.