計算機アーキテクチャ　130－13
（１９９８．　８．　５）

スケジューリング時におけるレジスタ不足の解消方式に関する研究

志水 浩司† 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:kshimizu,ishii@egg.ics.nitech.ac.jp
‡E-mail:liding@center.nitech.ac.jp

近年におけるハイパフォーマンスコンピューター技術において、レジスタは不足しがちな資源であり、レジスタの値を一時的に退避させるための最適化技術の必要性は高まっている。また、命令レベルの並列性を最大限に活用するために、レジスタをできるだけ利用したコードスケジューリングを行わなければならない。この論文ではスケジューリング時にスピルコードを使って必要なレジスタ数を減らすための方法について述べる。実験による結果では、スピルコードの使用によるプログラムへの影響を定量的に評価して使用する事で、プログラムの実行時間を縮める事に成功した。

An approach for code scheduling in presence of register constraints

Kouji Shimizu† Dingchao Li‡ Naohiro Ishii†

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

Registers are a scarce resource on all modern high-performance computer architectures, and their need is further increased by scheduling optimizations that introduce temporaries that have to reside in register. To best exploit available instruction-level parallelism, the code must be scheduled in the face of a bounded number of available registers. In this paper, we develop an efficient approach that helps compilers to make spilling decisions for reducing register requirements during the scheduling process. Our preliminary experimental results show that the algorithm can intelligently hide latency of the load/store instructions caused by register spilling from computation so that the program execution time is reduced as much as possible.

1 はじめに

逐次実行のプロセッサにおいて、レジスタ不足時にスピルコードを使用することは、プログラムの遅延の原因となるため極力避けられてきた。しかし、並列実行可能なプロセッサにおいては、スピルコードを他の命令と並列に実行し、プログラムに遅延を起こさないようにすることが可能である。

この論文では、先行制約と資源制約の両方を考慮した命令の実行タイミングの解析情報を利用し、プログラム内の命令の生存区間を予測する方法について述べる。その予測情報を基に、スケジューリング時にレジスタ不足が生じた時に、レジスタ再利用および、レジスタスピルによるプログラムへの影響を動的に計算し、レジスタ不足を解消する。レジスタスピルのプログラムへの影響を計算する時、どのレジスタにスピルコードを使うべきか、そして、いつメモリから値をリロードすべきかを判断する事ができる。これにより、コンパイラが選択的にスピルコードを使用し、プログラムのスケジューリング長を可能な限り縮める事ができる。

本論文は、以下のような構成となっている。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う。4章では、命令の実行タイミングの解析方法と、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章では今回提案するスケジューリングとレジスタ割当を同時に行う方法、およびレジスタ不足時のスピルコードの挿入条件について述べる。6章では結果を述べる。7章ではまとめを行う。

2 関連研究

レジスタ割当技法は、Chaitin[2] の方法がある。これは命令の依存関係をもとに変数の干渉グラフを作り、それをレジスタ数と同じ色数で色分けすることでレジスタ割当を行うというものであり、Briggs ら [3] の方法のように、多くのレジスタ割当技法は Chaitin の方法を応用したものである。

Norris と Pollock[4] は、Chaitin の方法を命令レベルでの並列実行が可能なアーキテクチャのための最適化コンパイラに応用し、SSG と呼ばれるレジスタ割当て方法を提案している。また、SSG を用いて、Norris と Pollock[5] はグローバルおよびローカルな命令スケジューリングとレジスタ割当を協調させる最適化技法を提案している。

SSG においてはどの変数にスピルコードを適用するかということをスピルコストという値を計算して判断する。しかし、SSG はスピルコストを逐次計算機でのレジスタ割当技法である Chaitin の方法と同様に計算しているため、並列計算機でのスピルコードが他の命令と同時に実行されて遅延が起きなくなることが可能であることを考慮していない。

そこで本稿では、Chaitin ベースのレジスタ割当て技法とは違い、レジスタ不足時にスピルコードがプログラム全体に与える影響を命令の実行タイミングの解析情報をもとに考慮したスピルコストの計算方法を提案する。それによって、プログラムの並列性を十分に生かした最適なスピルコードの使用が可能となる。

3 計算機及びプログラムモデル

3.1 計算機モデル

本研究で対象とするマシンモデルは VLIW アーキテクチャをベースとした、非均質な機能ユニットを有する命令レベルでの並列実行が可能な計算機であり、プログラム中の並列性の抽出は全て最適化コンパイラにおいて静的に行なう。機能ユニットには整数演算ユニット、浮動少数点演算ユニット、ロード/ストアユニットなどがあり、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。

ここで、計算機を P と表し、機能ユニットのタイプ数を s とする。また、k タイプの機能ユニットの数を m_k とする。機能ユニットを P_k^i は $\{P_k^i | 1 \leq k \leq s, 1 \leq i \leq m_k\}$ と表すことができる。

また、レジスタは全ての機能ユニットからアクセスでき、固定少数点数と浮動少数点数を共に扱うことのできる共有レジスタ $R = \{r_1, ..., r_l\}$ を採用している。l は実装されているレジスタ数を表す。レジスタは、定義点に変数をもつ命令の実行開始から変数に占有され、その変数を最後に使用する命令の終了と同時に解放されるものとする。

3.2 プログラムモデル

本研究では、通常のコンパイラによって作成されたスケジューリング及びレジスタ割当などの最適化を行う直前の中間プログラムを対象とする。その一連のプログラムにおける基本ブロックに対して本研究におけるコードスケジューリング及びレジスタ割当を行い、VLIW 命令コードを完成させる。

基本ブロック中に存在する出力依存関係及び逆依存関係は、変数に単一化代入を行うことによって

除去し、フロー依存関係のみからなるプログラムとする。また、プログラムの並列性をDAGグラフ $G=(\Gamma,A,\mu,\nu)$ で表現する。ここで Γ はプログラム中の命令 $I_i(i=1,...,n-1)$ の集合である。A は各命令の実行順序関係を表す。$\mu(I_i)$ は命令 I_i の実行時間(サイクル)を表す。$\nu(I_i)$ は命令 I_i が実行可能な機能ユニットの種類を表す。

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し、I_i で定義した変数を I_j で参照している。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。

DAGは必ず一つの入口ノード I_0 と一つの出口ノード I_n をもち、プログラム中の全ての命令は I_0 の後に実行され、かつ I_n の前に実行が終了する。

4 命令の実行タイミングの解析

コードスケジューリングやレジスタ割当などの最適化処理は、ヒューリスティックなアルゴリズムによって得られる近似解を利用しているため、必ずしも最適な命令の実行タイミングが得られるとは限らない。最適化処理のためのアルゴリズムは命令実行タイミングを近似解を求める際の評価基準としており、最適化処理の近似解を限りなく最適解に近づけるためには、命令の実行タイミングの下界値をより正確に求めるための命令実行タイミングの下界値の計算法が必要となる。

ここでは、最適化処理時において、命令の実行タイミングの下界値をより正確に求める解析手法として、[6]の方法を採用する。これは、プログラム実行時において非均質な機能ユニット数の不足から生じる、機能ユニットによる資源制約を考慮した命令の実行タイミングの解析手法である。この手法により、ある命令 I_i の実行が可能となる最も早い実行開始時刻 $\tau_{es}(I_i)$ および、I_i がプログラム全体の実行に遅延を起こさずに実行できる最も遅い実行開始時刻 $\tau_{ls}(I_i)$ が求められる。$\tau_{ef}(I_i)$ および $\tau_{lf}(I_i)$ は、それぞれ $\tau_{es}(I_i)$ および $\tau_{ls}(I_i)$ で I_i が実行開始した時の命令の実行終了時刻である。また、ある命令 $I_j\in Succ(I_i)$ において、I_i の実行終了から I_j の実行開始までに最低限必要な実行待ち時間を $d_r(I_i,I_j)$ であらわし、プログラム全体を実行するのに必要な時間を t_{cp} であらわす。

$\tau_{es}(I_i)$ および $\tau_{ls}(I_i)$ は、ヒューリスティックな値であり、スケジューリングの方法や、後にしめすレジスタ不足の解消方式によって遅延が起きた時には I_i は $\tau_{ls}(I_i)$ よりも遅いタイミングで実行開始するかもしれない。そのような時は命令の実行タイミングの下界値を再計算しなければならない。もし、I_i の実行開始が $\tau_{ls}(I_i)+t$ になるとき、命令の実行タイミングの下界を次のように再計算する。

$$
\begin{aligned}
\tau_{es}(I_i) &= \tau_{ls}(I_i)+t \\
\tau_{es}(I_j) &= \begin{cases} \max\{\tau_{es}(I_j),\tau_{es}(I_i)+d_r(I_i,I_j)\} & \\ \qquad\qquad \text{if } I_j\in Succ(I_i) & \\ \tau_{es}(I_j) \qquad \text{otherwise} & \end{cases} \\
\tau_{ls}(I_j) &= \tau_{ls}(I_j)+t
\end{aligned}
$$

5 スケジューリング方式

ここでは、前章までに説明した命令のタイミング解析情報等を利用した、スピルコードとレジスタ再利用を使用してレジスタ不足を解消することのできるレジスタ割当ての方法およびスケジューリング法を示す。

```
Procedure code_scheduling()
 Begin
  while I_unscheduling ≠ empty do
   cm := min_{P∈M} τ_f(P)
   For each type k where 1 ≤ k ≤ s do
    Compute P^k_idle(cm) and I^k_ready(cm)
    while P^k_idle(cm) ≠ empty
              and I^k_ready(cm) ≠ empty do
     select P from P^k_idle(cm) and I from I^k_ready(cm)
     if I needs register then                    if there is
no free register then
        Register_shortage_resolution()
        Register_assignment()
        I^k_ready(cm) − = {I}
      Else
        schedule(P, I)
        register_assignment()
        P^k_idle(cm) − = {P}, I^k_ready(cm) − = {I}
     Else
      schedule(P, I)
      P^k_idle(cm) − = {P}, I^k_ready(cm) − = {I}
 End.
```

図1: コードスケジューリングのアルゴリズム

5.1 アルゴリズム

図1はレジスタ不足の解消を考慮したコードスケジューリング法である。CP法などのリストスケジューリング法により、スケジューリングする命令を選択する。そのとき、命令に割り当て可能なレジスタが存在する場合は同時にレジスタを割り当て

る。割り当て可能なレジスタが存在しない時は次節以降で述べるレジスタ不足の解消法を用いて命令に割り当てるレジスタを命令に予約しておき、後にレジスタが使用可能になった時にスケジューリングを行う。

命令 I_i が現在スケジューリングの対象として選ばれたとする。もしスケジューリングする命令のためのレジスタが不足した場合、レジスタ不足を解消しなければならない。

本稿では、レジスタ不足の解消のために次の2つの方法を考慮する。1つ目の方法は、他の命令によって現在使用されているレジスタについて、レジスタを参照する命令が終了するまで命令 I_i の実行を待機させるという方法である。このときの命令の実行待機により必要なレジスタを確保する方法をレジスタ再利用と呼ぶ。2つめの方法は、他の命令によって使用されているレジスタを利用するために、レジスタの値をスピルコードを用いてメモリに退避することによって必要なレジスタを確保するという方法である。メモリに退避された値は、その値を再び使用する命令よりも前のタイミングでロード命令によって復帰されなければならない。

レジスタ不足を解消することにより、命令が $\tau_{lf}(I_i)$ までに実行完了できない場合、t_{cp} に対して遅延がおきてしまう。ここで、レジスタ再利用によっておきたプログラムの実行時間の増分を再利用コスト $Cost_{reuse}$ と定義し、スピルコードの使用によっておきたプログラムの実行時間の増分をスピルコスト $Cost_{spill}$ と定義する。$Cost_{spill}$ はストア命令挿入による実行遅延コスト $Cost_{spillout}$ とロード命令挿入による実行遅延コスト $Cost_{spillin}$ の和として表される。

スケジューリング時にレジスタ不足が起きた場合、各レジスタについて $Cost_{reuse}$ 及び $Cost_{spill}$ を以下に述べる方法を用いて見積り、$Cost_{reuse} \leq Cost_{spill}$ となるときはレジスタ再利用を行い、$Cost_{reuse} > Cost_{spill}$ となるときはスピルコードを使用する。

5.2 レジスタ再利用

ここでは、レジスタ再利用方式および再利用コストの見積り方法について述べる。スケジューリングの対象として I_i が選ばれたとする。I_i に割り当てるためのレジスタが不足して、レジスタ不足の解消のためにレジスタ再利用を行なうとする。

レジスタ再利用を行なうレジスタを r とし、現在の r の値を定義した命令を I_j とする。I_j で定義した値をを使用する命令は、プログラムの DAG グラフ

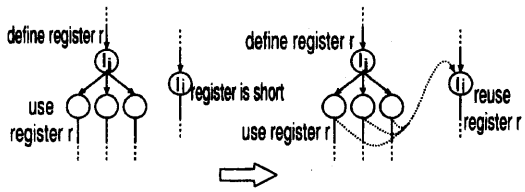

図 2: レジスタ再利用方式

において、I_j の直接後続ノードとなる命令である。図2のように、I_j の直接後続ノードの全てから、I_i へ新たなアークを加えれば、I_j の直接後続ノードの命令が全て終了した時点で、I_i が実行可能となり、同時に I_j で定義した r の値の使用が終了し、再利用が行なわれることになる。

ただし、I_i の後続ノードの集合 $Succ(I_i)$ に I_j の直接後続ノードのいずれかが含まれる場合、I_j の直接後続ノードの全てから I_i に新たなアークを加えて再利用しようとすることは、DAG グラフにループができてしまうことを意味する。この様な場合はプログラムは実行不可能である。ゆえにこのような命令 I_j によって定義されたレジスタについては再利用を行なわないことにする。

次に、レジスタ再利用コストの見積り法について述べる。レジスタ r を最後に参照する命令の終了タイミングを $\tau_f^R(r)$ とすると、もっとも早くレジスタが再利用可能となるタイミング τ_s^{reuse} は、$\tau_s^{reuse}(I_i) = \min_{r \in R}\{\tau_f^R(r)\}$ となり、レジスタ再利用による遅延コストは次のように見積もられる。

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

5.3 レジスタスピル

ここでは、スピルコードの挿入方法及び、スピルコストの計算方法について述べる。

スケジューリングの対象として、I_i が選ばれたとする。I_i に割り当てるためのレジスタが不足して、レジスタ不足の解消のためにスピルコードの挿入を行なうとする。スピルコードの挿入によって確保されるレジスタを r とし、現在の r の値を定義した命令を I_j とする。I_j で定義した値を使用する命令は、プログラムの DAG グラフにおいて、I_j の直接後続ノードとなる命令である。

図3のように、スピルコードの挿入を行なう。スピルコード挿入は、ストア命令の挿入とロード命令の挿入の2つの手順をとる。まず、r の値をメモリに退避させるためのストア命令を用意する。r を定

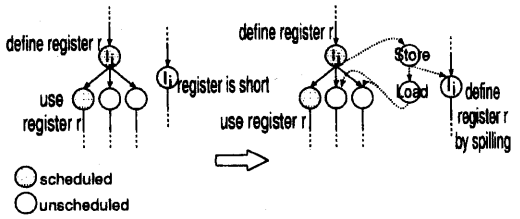

図 3: スピルコードの使用法

義する命令が完了しないと値をメモリに退避できないので I_j から用意したストア命令に新たなアークを伸ばす。ストアが完了した後 I_i が実行可能となるので、ストア命令から I_i にアークを伸ばす。つぎにメモリに退避させた値を復帰させるためのロード命令を用意する。ストアが完了しないと値をロードできないので、ストア命令からロード命令へアークを加える。ロード命令が完了した後、I_j のまだスケジューリングしていない直接後続命令は実行可能となるので、ロード命令から I_j の未スケジューリングの直接後続命令の全てにアークを加える。

また、リロードした値を使用する命令のシンボリックレジスタを、ロード命令で定義する新たな名前に改名しなければならない。

次に、スピルコストの見積り法について述べる。$Cost_{spill} = Cost_{spillout} + Cost_{spillin}$ としてスピルコストは見積もられる。まず $Cost_{spillout}$ の計算法について述べる。

現在、レジスタ r が他の命令によって最後に参照された実行タイミングを $\tau_f^{use}(r)$ とし、ストア命令の実行が可能な LD/ST ユニットの使用が完了したタイミングを $\tau_f(LD/ST)$ とすると、ストア命令の実行開始タイミングは $\max\{\tau_f(LD/ST), \tau_f^{use}(r)\}$ となりストア命令を使用してレジスタ r を確保し I_i が実行開始可能になる時刻は $\tau_s^{spill}(I_i) = \max\{\tau_f(LD/ST), \tau_f^{use}(r)\} + \mu(Store)$ となり、ストア命令挿入による遅延コストは次のように計算される。

$$Cost_{spillout}(r) = \tau_s^{spill}(I_i) - \tau_{is}(I_i)$$

つぎに、ロード命令の挿入による $Cost_{spillin}$ の計算方法について述べる。まず、ストア命令の挿入による遅延を考慮し、前述の命令の実行タイミングの再計算方式を用いて求められた命令の最早命令実行開始時刻を $\tau_{es}^{spill}(I_i)$ とし、最遅命令実行開始時刻を $\tau_{ls}^{spill}(I_i)$ とする。

スピルコードのストア命令によって退避した値を使用する命令を I_u とすると、$\tau_{ls}^{spill}(I_u)$ までにロード命令をつかって値を復帰させることができれば、I_u

```
Procedure _estimate_spillin_cost()
 Begin
  Cost_spillin(r) = ∞
  for τ_f from τ_ls^spill(I_v) to τ_s^spill(I_u) + μ(Load) Do
    τ_s = τ_f − μ(Load)
    I_LD/ST(τ_s, τ_f) = μ(Load)
    for each instruction I_i ∈ I_unscheduled Do
      if η(τ_s, τ_f, I_i) ≠ 0 and ν(I_i) = LD/ST then
        I_LD/ST(τ_s, τ_f)+ = η(τ_s, τ_f, I_i)
    δ_Load(τ_s, τ_f) = I_LD/ST(τ_s, τ_f)/m_LD/ST − μ(Load)
    if |V(τ_s, τ_f)| ≥ |R| then
      return Cost_spillin
    else
      if Cost_spillin > δ_Load(τ_s, τ_f)
        Cost_spillin(r) = δ_Load(τ_s, τ_f)
        τ_s(Load) = τ_s
        if δ_Load(τ_s, τ_f) ≤ 0 then
          Cost_spillin(r) = 0
          return Cost_spillin
 End.
```

図 4: $Cost_{spillin}$ 及びロード命令挿入時刻の計算

の実行の遅れによる遅延は起きない。ただし、ロード命令を実行するためには、ロード命令を実行するための機能ユニットと、ロード命令により復帰する値を保持するためのレジスタが必要となる。

ある時刻 θ_1 から θ_2 にかけて、命令 I_p が最低限実行しなければならない実行時間 $\eta(\theta_1, \theta_2, I_p)$ を

$$\begin{aligned}\eta(\theta_1,\theta_2,I_p) &= \max\{0, \min\{\tau_{ef}^{spill}(I_p) - \theta_1, \\ &\qquad \theta_2 - \tau_{ls}^{spill}(I_p),\ \ \theta_2 - \theta_1,\ \ \mu(I_p)\}\}\end{aligned}$$

として見積もる。$\nu(I_p) = Load/Store$ となる I_p について $\eta(\theta_1, \theta_2, I_p)0$ となる I_p の数がマシンに実装されている Load/Store ユニットの数 $m_{Load/Store}$ よりも小さい時、Load/Store ユニットの不足による実行遅延は起きない。

また、プログラムにおいて使われている変数の生存区間を予測することで必要なレジスタ数を見積もる。ある変数 v が命令 I_p で定義され、命令 I_q で最後に使用されるとする。v の生存区間を $[\tau_s(v), \tau_f(v)$ とすると、リストスケジューリングでは、スケジューリングする命令がレディー命令になった時に直ちにスケジューリングするため、変数の生存区間は、

$$\begin{aligned}\tau_s(v) &= \tau_{es}^{spill}(I_p) \\ \tau_f(v) &= \tau_{ef}^{spill}(I_q)\end{aligned}$$

とする。ただし、I_u およびその後続命令で使用される変数は、

$$\tau_f(v) = \tau_{lf}^{spill}(I_p)$$

とする。また、I_u およびその後続命令で定義される変数は考慮しない。変数の生存区間の情報により、ある区間 $[\theta_1, \theta_2]$ における必要なレジスタ数 $V(\theta_1, \theta_2)$ を求めることができる。

$\eta(\theta_1, \theta_2, I_p)$ および $V(\theta_1, \theta_2)$ の情報を基に $Cost_{spillin}$ および、ロード命令の最適実行開始時刻 $\tau_s(Load)$ を求める。I_u を実行するまでの区間において、レジスタ数が十分足りている時に、区間内で機能ユニット不足による影響が最も小さい時刻を $\tau_s(Load)$ としている。図4は、$Cost_{spillin}$ および $\tau_s(Load)$ を求めるアルゴリズムである。

表 1: ベンチマークによる評価結果

Prog.	reg.	A	B	Prog.	reg.	A	B
k01	11	17	17	k18a	21	40	40
k01	10	17	17	k18a	20	40	40
k01	8	23	28	k18a	18	40	41
k01	6	34	38	k18a	16	42	46
k02	10	13	13	k18b	22	50	50
k02	8	16	16	k18b	20	51	51
k02	6	21	24	k18b	18	55	69
k04	8	10	10	k18b	16	67	76
k04	6	17	17	k18c	10	12	12
k05	11	24	24	k18c	8	13	13
k05	10	24	24	k18c	6	23	25
k05	8	27	27	k19a	8	13	13
k05	6	35	36	k19a	6	17	17
k07	16	32	32	k19b	9	15	15
k07	14	35	37	k19b	8	18	16
k07	12	41	47	k19b	6	25	25
k07	10	56	60	k21a	7	12	12
k09	24	41	41	k21a	6	12	12
k09	22	44	45	k21b	15	23	23
k09	20	42	42	k21b	14	24	24
k09	18	52	52	k21b	12	35	38
k10	16	28	28	k21b	10	44	45
k10	14	28	28	k23	18	35	35
k10	13	30	36	k23	16	35	35
k11	8	12	12	k23	14	37	40
k11	6	26	27	k23	12	39	44

6 評価結果

ここでは、代表的なベンチマークプログラムである、Livemore kernel のプログラムについて本研究の方式を適用した場合のスケジューリング長を表1のAに示す。また、レジスタ不足の解消方式において再利用を優先して使い、スピリングは再利用が不可能な時にのみ行うという方式によるスケジューリング長も併せてBに示す。これは、スピルコードの使用をできるだけ避けるという従来の方式によるものである。

この実験においては、VLIW プロセッサは整数演算ユニットを2個、浮動少数点演算ユニットを2個、Load/Store ユニットを2個持つものとしている。また、Load 命令の実行時間は2、Store 命令の実行時間は2としている。

結果によると、本方式の方が従来の方式よりも優れている例が19例存在した。逆に、従来の方式の方が本方式よりも優れている例が1例存在した。

7 まとめ

この論文では、スケジューリング時にレジスタ数が不足した時にコンパイラが効果的にスピルコードを使用するための方式について述べた。実験結果では、従来の方式よりも優れたスケジューリング長でスケジューリングする事ができた。今後の課題としては本方式をより多くのプログラムについて実験し、その有効性を評価する事が挙げられる。

参考文献

[1] 小野田 亘利, 李 鼎超, 石井 直宏, 情報研報 Vol.96,No.81,pp.69-74,1996.

[2] G.J.Chaitin, Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction,pp. 98-105, 1982.

[3] P.Briggs, K.Cooper, K.Kennedy, and L.Torczon, Proc. of the ACM SIGPLAN'89 Conf. Programming Languages Design and Imple., pp. 275-284, 1989.

[4] C.Norris, and L.L.Pollock, Proc. Supercomputing'93, pp. 804-813,1993.

[5] C.Norris, and L.L.Pollock, IEEE Proc. of MICRO, pp. 169-179, 1995.

[6] 李 鼎超, 有田 隆也, 石井 直宏, 曽和 将容, 情報処理学会論文紙, Vol.34, No.11, pp.2378-2385, Nov. 1993.