計算機アーキテクチャ 149−14
（２００２．８．２２）

# レジスタ生存グラフを用いたレジスタ割付け及びコードスケジューリング技法

片岡正樹[†] 古関聰[††]
小松秀昭[††] 深澤良彰[†]

これまで、命令レベル並列性（ILP）を抽出するレジスタ割付け及びコードスケジューリングの統合資源割付け手法として、レジスタ生存グラフを用いたレジスタ割付け手法を提案してきた。しかし、この手法ではレジスタ数制約を満たす操作で ALU 数が考慮されていなかった。そのためコード発行時にレジスタが不足する可能性があり、コードスケジューラによりスピルコードが挿入され性能が想定に満たない場合があるという欠点があった。そこで本論文では、レジスタ数制約を満たす操作後に仮想的にコード発行を行ってレジスタ不足を回避することで、コードスケジューラによる余分なスピルコードの挿入を無くし、さらなる ILP 抽出手法を提案する。また、いくつかのプログラムを用いてその評価を行う。

# An Integrated Method of Register Allocation and Code Scheduling Based on Register Existence Graph

Masaki Kataoka,[†] Akira Koseki,[††] Hideaki Komatsu[††]
and Yoshiaki Fukazawa[†]

We have suggested an integrated algorithm of register allocation and code scheduling using Register Existence Graph in order to extract ILP. However, the number of ALUs hasn't been considered during dissolving register number restriction in this method. Therefore, it has a possibility of register shortage at the code generation stage, and doesn't give the assumed performance because of spill code inserted by the code scheduler. In this paper, we introduce a method avoiding register shortage using a virtual code generation after the stage that dissolves register number restriction in order to extract more ILP.

## 1. はじめに

現在主流となっている命令レベル並列プロセッサで効率良い実行を行うためには、通常演算に必要なデータがすべてレジスタに格納されている必要がある。そこで、レジスタ割付けによってスピルコードを少なくする研究[1] が進められ、レジスタが有効活用されるようになった。

一方、高い ILP(Instruction Level Parallelism) を抽出するためには、命令の入れ替えが必須となるが、並列性の考慮がされていないレジスタ割付けの後では逆依存が発生する可能性がある。つまり、レジスタ割付けが高い ILP の抽出を阻害してしまい、命令レベル並列プロセッサの性能を充分に引き出すことができていなかった。

そこで、命令レベル並列実行においては、スピルコードの最少化だけでなく、命令の並列実行を阻害しないように考慮されたレジスタ割付け手法[2)〜4)] が必要となってくる。この２つを同時に実現するためには、レジスタアロケータとコードスケジューラの相互作用問題を解決する必要がある。

レジスタアロケータとコードスケジューラの相互作用問題とは、コードスケジューリングを先に行うと (プリパス型[6])、コードスケジューラによって抽出された高い並列性を、レジスタアロケータが下げてしまい、逆に、レジスタ割付けを先に行うと (ポストパス型[6])、逆依存の発生や現実には必要のない余分なスピルコードの挿入が行われてしまい、並列実行に影響が出てしまうという問題である。つまり、コード最適化には欠かすことのできないレジスタ割付けとコードスケジューリングのどちらを先に行っても、高い ILP を抽出できないということである。今までに、これらの相互作用問題について、

† 早稲田大学理工学部
School of Science and Engineering, Waseda Univ.
†† 日本 IBM（株）東京基礎研究所
Tokyo Research Laboratory, IBM Japan, Ltd.

ヒューリスティクスに基づいたアルゴリズムがいくつか挙げられているにも関わらず、すべてのプログラムについて最適に近い解を求めることは未だにできてない。

この問題を解決し高い ILP を抽出するためには、使用レジスタ数を減らし、結果として並列度を下げる方向の動作を行うレジスタアロケータと、並列度を上げて使用レジスタ数を増やす方向の動作を行うコードスケジューラとの、協調動作を考慮することが重要となってくる。

そこで本論文では、この問題をレジスタ生存グラフを用いたレジスタ割付けと、プレスケジューリングを用いることで解決する手法を提案する。

## 2. 関 連 研 究

命令レベル並列プロセッサ向けレジスタ割付け手法としては、並列化レジスタ干渉グラフを用いたレジスタ彩色法[2)3)] が挙げられる。この手法は、並列性を保持するエッジを加えたレジスタ干渉グラフに対して、彩色問題を解くことでレジスタ割付けを行っている。

しかし、エッジは中間コードの並び順に依存し、また並列実行に対して安全な見積もりを行っているため、現実には干渉しないノード間にも、エッジが張られたままになることが多く、無駄なスピルコードが挿入されてしまうという問題があった。

この問題を解決する手法として、レジスタ生存グラフを用いたレジスタ割付け手法[4)5)] が提案されている。レジスタ生存グラフは、プログラム依存グラフ (PDG)[8)] に実行条件を示すガード情報を付加した、ガード付プログラム依存グラフ (GPDG)[4)] から生成され、仮想レジスタをノードで、その値の生産と使用をエッジで表し、同時刻に生存している仮想レジスタを表すノードを等時刻線と呼ばれる線が横切るグラフである。このため、中間言語の命令順序はレジスタ生存グラフには影響せず、並列化レジスタ干渉グラフでは切ることが困難であった、現実には干渉しないノード間のエッジを切ることができるという特徴を持っている。このレジスタ生存グラフを用いた手法では、この特徴を用いて無駄なスピルコードが挿入されることを防いでいる。また、あるタイミングに同時に生存している仮想レジスタ数が、利用可能な実レジスタ数以下になるように、あらかじめ命令の実行順序に制約を加えておくことで、コードスケジューラとの統合アルゴリズムを実現している。

しかし、この制約によって保証されることは、各サイクルに存在する命令がすべて並列に実行された場合には、レジスタ数制約が満たされるということである。しかし、実際に並列実行できる命令数は ALU 数までである。この手法では、レジスタ数制約を満たすための操作での ALU 数の考慮がされていないため、入力プログラムの構造によっては、コード発行時にレジスタが足りなくなる恐れがあった。この場合には、コードスケジューラによるスピルコードの挿入が行われてしまい、想定していた性能が出ないことがあるという欠点があった。

そこで、本論文では、この欠点を補うために仮想的にコード発行を行うことでレジスタ不足を前もって検出し、これを回避するように時間的な依存を加えてあらかじめ命令の入れ替え範囲を狭めておくことで、実際のコード発行時におけるレジスタ不足の可能性を低減する手法を提案する。

## 3. 本手法の概要

本手法は、大きく分けて２つのステージに分けることができる。１つは最大干渉度低減操作、もう１つはプレスケジューリングとコード発行である。それぞれについて以下に記す。

### 3.1 最大干渉度低減操作

主に、レジスタ数制約を満たす操作を行う。その流れを図 1に示す。

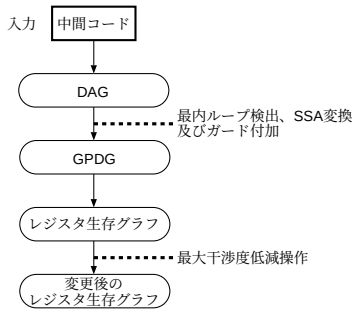


**図 1** 本手法までの流れ

プログラムの最内ループに対して、 SSA 変換[7)] を施し、ガード情報を付加して、 GPDG を作成する。GPDG は、プログラム中の最内ループのデータ依存と制御依存を同等に表すのに用いられ、最内ループの入口と出口を表す START ノードと END ノードを持ち、命令をノードで表現している。また、各ノードには、ガード情報が付加され、データと制御の依存関係を有向エッジで表現する。 SSA 変換後のサンプルコードとそれから作成した GPDG を図 2と図 3に示す。

次に、この GPDG からレジスタ生存グラフを生成する。レジスタ生存グラフでは、同時刻に存在する仮想レジスタ数、つまり、等時刻線で結ばれたノード数を干渉度、また、干渉度の最大値を最大干渉度と呼ぶ。ある時刻での干渉度が、実レジスタ数を超えている場合、レジ

```
      cc1=r3==0
(cc1)  r4=r1+r2
(cc1)  r5=r1-r2
(cc1)  r6=r2+r3
(cc1)  r7=r4<<1
(cc1)  r8=r5<<1
(cc1)  r9=r6+r1
```

**図 2**　サンプルコード

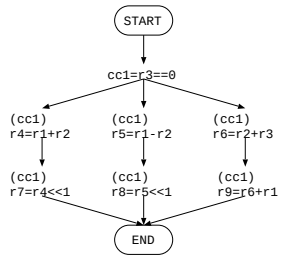


**図 3**　GPDG の例

スタが不足することを意味するため、制限を加えることで最大干渉度を実レジスタ数以下に低減する必要がある。このレジスタ数制約を満たすための操作を、最大干渉度低減操作と呼び、すべての時刻での干渉度、つまり最大干渉度が実レジスタ数以下になるようにする。

最大干渉度低減操作のアルゴリズムを以下に記す。

(1)　サイクル毎に干渉度を計算し、実レジスタ数を超えているサイクルを検出する。

(2)　干渉度を下げたいサイクルの次のサイクルに合流点が存在し、かつ現在より前のサイクルにその合流点を支配していない分岐点が存在する場合には、分岐点が現在のサイクルにくるように生存区間を伸ばしてみる。

(3)　（2）の操作が適用できない場合、もしくは、操作後も干渉度が実レジスタ数を超えている場合には、以下の操作を行う。

　(a)　作業用リストに、レジスタ生存グラフ上で現在のサイクルのノードをすべて入れる。

　(b)　作業用リスト内のノードを自由度の低い順に並べ替える。ここでいう自由度とは、クリティカルパスに影響せず、どれほど遅らせることが可能かという指標である。

　(c)　実レジスタ数を超えない分、作業用リストの先頭からノードを取り除く。

　(d)　作業用リストの残りのノードすべてに対して、スピルコードを生成する。さらに、スピルインノードを、レジスタ生存グラフ上で現在のサイクルに1増やした場所に付け加える。

ここで、（2）の操作は、演算の実行順序を操作することによって、レジスタ数制約を満たしている。それでも不足してしまう場合には、最も使用されるのが遅いレジスタ上の値を、メモリにスピルすることでレジスタ数制約を満たしている。スピルした値をレジスタに戻すタイミングは、現在のサイクル以降で、干渉度に余裕がある最初のサイクルになっている。

図 4に、図 3から生成されたレジスタ生存グラフの例を示す。簡単のため、今後はレジスタ数4、 ALU 数2というハードウェア上の場合で説明する。また、図 4のレジスタ生存グラフでは、すでにレジスタ数制約を満たしているため、最大干渉度低減操作を施す必要がない。

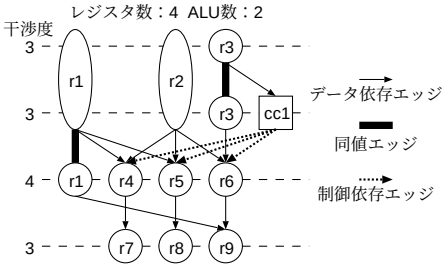


**図 4**　レジスタ生存グラフの例

### 3.2　プレスケジューリングとコード発行

最大干渉度低減操作で加えた制約の補完と保証、最後に資源割当てを行う。この流れを図 5に示す。

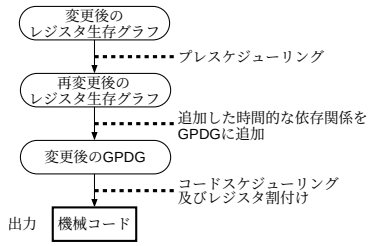


**図 5**　本手法の流れ

変更後のレジスタ生存グラフに対して、 ALU 数を考慮して、仮想的なコード発行を行う。この操作中にレジスタ不足を検出した場合、バックトラックを用いて、優先的に発行した方が良い命令を探索する。この探索方法を用いていくつか試行した結果、 2つの仮想レジスタを用いる演算の中で、参照が最後の仮想レジスタを1つ以上用いているものは、優先度の高い演算になり、定数との演算やシフト演算などは、優先度の低い演算になる傾向がある。ここで選択されなかったすべての命令に対して、新たに時間的な依存という制約を加え、その制約を満たすようにレジスタ生存グラフを変形する。こうすると、コード発行順序を限定でき、レジスタが不足する可能性を低減することができる。もし、どの命令を選択したとしても、レジスタが不足してしまう場合には、スピルコードの挿入を行い、レジスタ生存グラフを変形する。この変形操作を、すべてのサイクルについて、レジスタ不足が起こらなくなるまで繰り返す。この操作を施した後のレジスタ生存グラフに、最大干渉度低減操作と

プレスケジューリングによって適用されたすべての制約を、 GPDG にも反映させ、レジスタ割付けとコードスケジューリングを行い、中間コードの発行を行う。

図 4のレジスタ生存グラフは、コード発行時にレジスタが足りなくなる可能性がある例の１つである。この例では、２サイクル目に r4、 r5、 r6 を生成する３つの演算が存在する。 ALU 数が２であるため、３つの演算を並列実行することができず、これらの中から２つ選ぶことになる。ここでは、この３つの演算の自由度が等しいので、リストスケジューリングでは発行の優先度が等しい命令となる。もし、優先度が異なる場合には、優先度の高いものから発行される。

演算の選び方としては、 r4 と r5 を生成する演算組、 r4 と r6 を生成する演算組、 r5 と r6 を生成する演算組の３つが考えられる。しかし、もし、コードスケジューラが r4 と r5 を生成する演算組を選んで発行した場合、演算後に r1、 r2、 r3、 r4、 r5 の５つの値を格納するためのレジスタが必要になるが、レジスタ数が４であるため、すべてはレジスタに乗り切らない。そのため、コードスケジューラによって ALU の２サイクル目で空きスロットが作られ、レジスタ上の値は図 6のレジスタ生存グラフに示すような振る舞いをし、すべての演算を完了するために５サイクルかかることになる。ただし、他の演算組を選んだ場合には、レジスタが不足することはなく、すべての演算を完了するのに４サイクルで済むので、これが最悪時の性能ということになる。

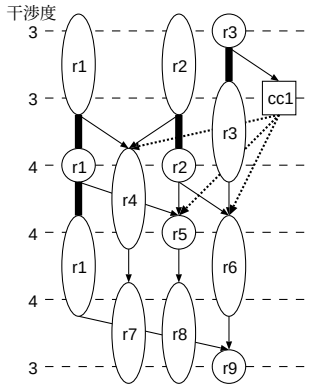


**図 6**　コード発行時における最悪時の性能

このようなコード発行時における性能の低下を避けるために、 r4 と r5 を生成する演算組が２サイクル目で選ばれないように、図 4のレジスタ生存グラフを、図 7のように変形する。 r4 と r5 を生成する演算組が選ばれないようにするには、 r4 を生成する演算を遅らせる方法と、 r5 を生成する演算を遅らせる方法の２種類が考えられる。この場合、どちらを選んでも性能は変わらないので、アルゴリズム上はどちらでも選べるようになっているが、ここでは説明の便宜上 r5 を生成する演算を遅らせることにする。

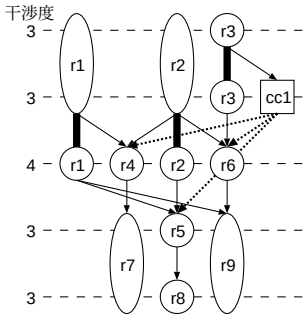


**図 7**　プレスケジューリング後のレジスタ生存グラフ

また、図 7のレジスタ生存グラフの３サイクル目で３つの演算が存在しているが、自由度の最も低い r5 を生成する演算は、発行の優先度が高いため必ず選ばれ、 r7 と r9 を生成する２つの演算の中から、１つがコードスケジューラによって選択されることになる。この場合、どちらが選択されたとしても、演算後にレジスタが不足することはないため、このまま変形せずにしておく。これは、プレスケジューリングによって完全に発行順序を決めてしまうのではなく、後に動くコードスケジューラに、レジスタが不足しない程度に命令選択の余地を残しておくためである。

図 7のプレスケジューリングを施したレジスタ生存グラフを基に、図 3の GPDG に、さらに時間的な依存を付け加えて変更した GPDG を、図 8に示す。

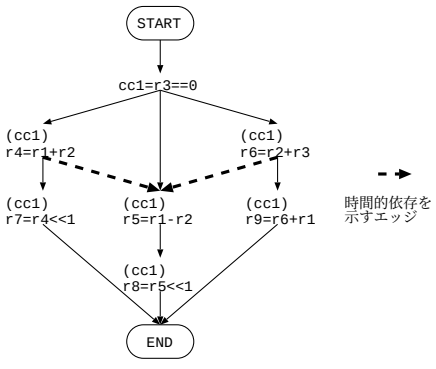


**図 8**　変更後の GPDG

r5 を生成する演算に対して、 r4 と r6 を生成する各演算から時間的依存を示すエッジを加えることで、 r5 を生成する演算と、 r4 と r6 を生成する各演算とが並列実

行されないようにしている。この GPDG からレジスタ割付けとコードスケジューリングを行った結果が図 9 である。

| REG1 | REG2 | REG3 | REG4 |
|---|---|---|---|
| r1 | | r2 | r3 |
| r1 | | r2 | r3 |
| r1 | r4 | r2 | r6 |
| r1 | r7 | r5 | r9 |
| r1 | r7 | r8 | r9 |

| ALU1 | ALU2 |
|---|---|
| cc1=r3==0 | |
| (cc1)r4=r1+r2 | (cc1)r6=r2+r3 |
| (cc1)r5=r1-r2 | (cc1)r7=r4<<1 |
| (cc1)r8=r5<<1 | (cc1)r4=r6+r1 |

**図 9** レジスタ割付けとコードスケジューリング結果

## 4. アルゴリズム

本手法で用いている、プレスケジューリングと資源割付けの具体的なアルゴリズムを以下に記す。

### 4.1 プレスケジューリング

最大干渉度低減操作を施した後のレジスタ生存グラフを入力として、以下のようにプレスケジューリングを行う。

(1) 作業用リストに、現在のサイクルで発行可能な命令をすべて入れる。
(2) 作業用リスト内の命令を、自由度の低い順に並べ替える。
(3) 作業用リストから、与えられた ALU 数で考えられる演算の組み合わせの中で、自由度の合計が最も低くなるものをすべて作る。
(4) すべての組み合わせに対して、命令実行後の干渉度を計算する。
(5) 干渉度が実レジスタ数を超えているものがない場合は、以下の操作を行う。
  (a) 組み合わせの中に入っているすべての命令を、作業用リストから除く。
  (b) 作業用リスト中の命令の自由度を 1 減らし、発行可能な命令がなくなるまで（1）へ戻る。
(6) すべて干渉度が実レジスタ数を超えている場合は、以下の操作を順に試す。
  (a) すでに作った組み合わせ以外の演算の組み合わせを作り、（4）に戻る。
  (b) 組み合わせる数を減らした演算の組み合わせや、スピルアウト命令との組み合わせを作り、（4）に戻る。
  (c) 1つも命令が発行できない場合には、スピルアウト命令を発行し、レジスタ生存グラフの再構築、命令の自由度の再計算をして、（1）に戻る。
(7) 干渉度が実レジスタ数を超えているものが 1 つ以上ある場合は、以下の操作を行う。
  (a) 命令実行後の干渉度が実レジスタ数を超えている組み合わせが、すべて起きないようにレジスタ生存グラフを変形する。
  (b) 変形の際に、生存区間が変化していない仮想レジスタを生成する命令を、作業用リストから除く。
  (c) レジスタ生存グラフの整合性がなくなるため、レジスタ生存グラフの再構築をする。
  (d) レジスタ生存グラフを再構築したことによって、命令の自由度も変わるので、自由度を再計算して、（1）へ戻る。

### 4.2 レジスタ割付け及びコードスケジューリング

最大干渉度低減操作とプレスケジューリングによって、レジスタ生存グラフに加えられた制約を満たすように、 GPDG に仮想的な依存を加え、レジスタ割付け及びコードスケジューリングを行う。ここでは、あらかじめレジスタ数制約を満たすことで、レジスタアロケータとコードスケジューラの統合アルゴリズムを用いることを可能としている。

以下に本手法で用いている、レジスタ割付け及びコードスケジューリングの統合アルゴリズムを記す。

(1) マシンサイクルを 1 とする。
(2) 作業用リストに GPDG 上での深さが 1 のノードをすべて入れる。
(3) GPDG のスタートノードにおいて、既に生存している仮想レジスタを実レジスタに割付ける。
(4) 以下の操作を、作業用リストが空になるまで繰り返す。
  (a) 作業用リストのノードを自由度の低い順に並べ替える。
  (b) 作業用リストの先頭から、 ALU 数を越えない数のノードに対応する命令を、現在のマシンサイクルの命令スロットに割付け、それらを作業用リストから除く。
  (c) 割付けた命令が定義する仮想レジスタを実レジスタに割付ける。
  (d) 作業用リストにあるノードの自由度を 1 減らす。
  (e) 命令スロットに割付けられた命令の結果により、依存関係が満たされたノードをリストに加える。
  (f) マシンサイクルを 1 増やす。

この結果から、機械コードを生成する。

## 5. 評　　価

本アルゴリズムを用いた、レジスタアロケータ及びコードスケジューラの評価を行う。ハードウェアは、IA-64に準拠したプロセッサとし、整数演算命令とStore命令には１クロック、Load命令には２クロックかかるものとした。レジスタ数が８、ALU数が２、４の各場合について評価をとった。評価プログラムとして、Stanford-Integer Benchmarkから取り出した、いくつかのプログラムの最内ループを用いた。

評価値として用いたのは、GPDGのコードスケジューリング後の実行サイクル数である。実行サイクル数は実行時間に比例するため、実行サイクルを比較することで、その性能を比較することができる。

従来手法として、文献4)で挙げられている手法でクリティカルパス長を求め、我々の手法を適用したときの性能と比較した。ALU数が２の場合での結果を表1に、ALU数が４の場合での結果を表2に示す。

**表１**　ALU数２でのクリティカルパス長比較

| *benchmark* | 従来手法 | 本手法 |
|---|---|---|
| *Bubble* | 23 | 23 |
| *Initarr* | 42 | 42 |
| *Maxarray* | 27 | 22 |
| *Permute* | 19 | 19 |
| *Try* | 64 | 64 |

**表２**　ALU数４でのクリティカルパス長比較

| *benchmark* | 従来手法 | 本手法 |
|---|---|---|
| *Bubble* | 15 | 15 |
| *Initarr* | 32 | 32 |
| *Maxarray* | 18 | 18 |
| *Permute* | 15 | 15 |
| *Try* | 49 | 49 |

本手法の効果が確認できたのは、ALU数が２の場合のMaxarrayのみだった。これは、従来手法では、コード発行時にレジスタが不足し、スピルアウト命令とスピルイン命令によって、ALUを占有されたのに対し、本手法では、レジスタ不足が極力起きないよう命令を選択し、並び替えたことによって、スピルアウト命令とスピルイン命令を無くすことに成功し、ALUを有効利用できた結果である。

ALU数が４の場合のMaxarrayでは、本手法によって、スピルアウト命令とスピルイン命令を無くすことに成功したものの、従来手法でも、ALUが空いているサイクルがあるため、スピルアウト命令やスピルイン命令が他の命令と並列実行できて、遅延が隠蔽されたため、結果として同じ性能になってしまった。

その他のプログラムでは、ALU数が２、４どちらの場合においても、コード発行時にレジスタ不足が起きず、プレスケジューリングによって追加された制約が無かったため、すべて同じ性能になっている。

## 6. 終わりに

本稿では、命令レベル並列プロセッサ向けにコードを最適化するレジスタ割付け及びコードスケジューリング技法を提案した。プレスケジューリングを用いることで、コード発行時にレジスタが不足する可能性の低減を図った。しかし、本手法は命令レベル並列性の低いプログラムや並列度の高いハードウェア上では、コードスケジューラによって挿入されたスピルコードと、他のコードとが並列実行が可能であり、本手法を適用した場合と性能が変わらない。今後はこの欠点を解消する手法を研究していく予定である。

## 参　考　文　献


1) G.J.Chaitin, M.A.Auslander, A.K.Chandro, J.Cocke, M.E.Hopkins, and P.W.Markstein: “Register Allocation via Coloring”, *Computer Languages Vol.6*, 1981, pp.47-57.

2) C.Norris and L.L.Pollock: “A Scheduler-Sensitive Global Register Allocation”, *Proc. of the ACM SIGPLAN '93 Conf. on Supercomputing*, 1993, pp.804-813.

3) S.S.Pinter: “Register Allocation with Instruction Scheduling: a New Approach”, *Proc. of the ACM SIGPLAN '93 Conf. on Programming Languages Design and Implementation*, 1993, pp.248-257.

4) 小松秀昭, 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大域的コードスケジューリング技法”, 情報処理学会論文誌, Vol.37, No.6,1996, pp.1149-1161

5) 古関聰, 小松秀昭, 百瀬浩之, 深澤良彰: “命令レベル並列アーキテクチャのためのコードスケジューラおよびレジスタアロケータの協調技法”, 情報処理学会論文誌, Vol.38, No.3,1997, pp.584-594

6) Monica Lam: “Software Pipelining:An effective scheduling technique for VLIW machines”,*In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1988, pp.318-328.

7) R.Cytron, J.Ferrante, B.Rosen, M.Wegman and K.Zadeck: “An Efficient Method of Computing Static Single Assignment Form”, *Conf. Record of the 16th ACM Symposium on the Principles of Programming Languages*, 1989, pp.25-35.

8) J.Ferrante, K.J.Ottenstein and J.D.Warren: “The Program Dependence Graph and Its Use in Optimization”, *ACM Trans. Prog. Lang. Syst. Vol.9, No.3*, 1987, pp.319-349.