計算機アーキテクチャ　144−16
（２００１．７．２６）

制御依存の緩和を考慮した並列性抽出手法

伊藤佑一[†]　古関聰[††]
小松秀昭[††]　深澤良彰[†]

制御依存の緩和により、従来手法に比べ命令レベルの並列性を高める手法を提案する。コントロールフローとデータフローの制約を緩和し、依存グラフのクリティカルパスを短縮することでプログラムの実行速度向上が可能である。現在データフローに関しては、投機実行や命令の複製等の制約緩和手法が提案されている。コントロールフローに関しては条件付き実行手法等が提案されているが、制約の緩和は充分ではない。そこで本手法ではコントロールフローに対し、条件分岐演算の投機実行、実行条件のブール最適化及び条件分岐演算の複製により、その依存グラフのクリティカルパスの短縮を図る。

A Parallelism-Extraction Method Considering the Relaxation of Control Dependence

Yuichi Ito ,[†] Akira Koseki ,[††] Hideaki Komatsu [††]
and Yoshiaki Fukazawa [†]

We propose a technique of extracting instruction level parallelism and critical path compression of the dependence graph by relaxing control dependence. The program execution time can be reduced by critical path compression of dependence graph. The reducing techniques of data flow, like speculation and duplication, are proposed. As for the reducing technique of control flow, predication is proposed, however the relaxation of control dependence is not enough. In this paper, we propose a technique of critical path compression of dependence graph by speculation of conditional operation, Boolean optimization of conditions and duplication of control flow.

1. はじめに

プログラムから抽出できる命令レベルの並列性 (ILP) には限界があり、プロセッサの並列度を極端に大きくしても、あまり効果が得られないことが知られている。そのため、高い並列度を持つプロセッサの能力を生かすためには、プログラムの実行順序に手を加えて ILP を引き出す必要がある。

その一つとして、制御依存を緩和する投機実行 (Speculation) がある。従来の投機実行では、ある分岐先の命令を、分岐命令の前に実行できる機構を用意することで命令を並列実行し、ILP を抽出していた。

ILP を有効に利用するために、IA-64 では「条件付き実行 (Predication)」という機能を導入している。これは、分岐予測が外れた場合のパイプラインストールによる遅延を回避するため、分岐の後続命令に対し実行条件を示すプレディケートレジスタというタグを付加し、その実行条件を参照させることで、条件分岐演算の結果待ちをすることなく分岐の後続命令を並列に実行させるというものである。

通常のレジスタ演算の場合、プログラム中の逆依存を回避するために施される SSA 変換[1] によって、実行条件に関わらず投機実行を行うことができる[2]。その場合、実際には実行されない演算結果のレジスタ値は無効化される。しかし、実行条件の付加された条件分岐演算はその実行条件次第で演算結果が異なるため、参照する実行条件が求まるまで演算を実行することができない。そのため、このような実行条件の付加された条件分岐演算は投機的に並列実行することができない。

上述の SSA 変換においては、ϕ ファンクション[3] が挿入される。これは、SSA 変換を施すと、異なる分岐上で同じ値を生成する場合、別の仮想レジスタが割り付けられるが、その値の内実行された方の値をその後続命令の値として使用させる命令である。ただし、実行条件を参照しなければ実行することができないため、制御依存により実行が遅くなってしまう。

† 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.

この依存を解消するため、ϕ ファンクション以後の命令を複製することで依存を回避する方法がとられてきた。しかし、条件分岐演算に関する ϕ ファンクションのあとの命令を複製してしまうと、条件分岐演算が複製されてしまい、制御構造が複雑になってしまうため、命令の複製を行っていなかった[4)5)]。その結果、命令を投機実行しても、コントロールフローによるクリティカルパス自体がボトルネックとなってしまい、効果的に投機実行することができていなかった。

このように投機実行を阻害しているコントロールフローによるクリティカルパスを短縮することができれば、より自由なスケジューリングが可能になり、プログラムを高速に実行できる。

これに対して、本手法ではコントロールフローの途中で実行条件を参照することになる ϕ ファンクションに対して、それ以降の命令列を複製して ϕ ファンクションを除去し、論理式化した実行条件の論理演算により条件分岐演算を投機的に並列実行することで、コントロールフローによるクリティカルパスの短縮を図る手法を提案する。

2. 関連研究

投機実行の関連研究としては、文献 4) が挙げられる。この手法は、拡張された PDG(Guarded PDG)[4)] を用いて依存グラフのクリティカルパスを短縮し、大域的にコードスケジューリングを行っている。この手法の中では、SSA 変換を施してレジスタの競合を避けることで、レジスタ演算ならば制御依存と関係なく投機実行することを可能にしている。さらに、後続の命令列を複製することで ϕ ファンクションを除去し、ϕ ファンクションに対してかかる制御依存を緩和している。しかし、この手法では、コントロールフロー中の ϕ ファンクションに対しては、実行条件自体が複雑化するため複製を行っていない。また、実行条件つきの実行条件演算も参照できる実行条件は一つという制約から投機実行を行っておらず、そのためコントロールフロー自体がボトルネックとなり、効率的なクリティカルパス短縮が行われていない。

このコントロールフローを短縮する手法としては、文献 2) が挙げられる。この手法は、実行条件の参照を必要とする命令に使用される実行条件を、条件分岐演算の結果を用いた論理式で表現し、それをハードウェアを用いて実装するというものである。ハードウェアを用いて論理演算を実装しているため、高速に論理演算を処理することができ、結果として高速な実行が行われている。しかしこの手法では、論理演算機構を実装するためのハードウェアコストが新たにかかる。また、命令の投機実行による副作用が考慮されておらず充分なクリティカルパス短縮が行われていなかった。

3. 本手法の特徴

データフローとコントロールフローを統合的に扱っている GPDG のようなグラフでは、グラフのクリティカルパス長が実行時間を左右する。このうちデータフローに関する部分には、投機実行や ϕ ファンクションの複製を用いて短縮が図られてきた。本手法の特徴は、ϕ ファンクションの複製と実行条件の論理演算によって、制御依存を緩和することで依存グラフを短縮し、ソフトウェアで論理演算を実装している点である。制御依存を緩和し、依存グラフのクリティカルパスの短縮を図ることができれば、たとえ命令数が増えたとしても、並列度の高いプロセッサでの高速な実行が可能である。そこで、コントロールフローに対して、以下のような手法を用いて制約の緩和を図る。

3.1 コントロールフロー中の ϕ ファンクション

制御構造の複雑化を回避するため、それ以後の命令を複製していない ϕ ファンクションでは、その制御依存がクリティカルパスの短縮を妨げている。

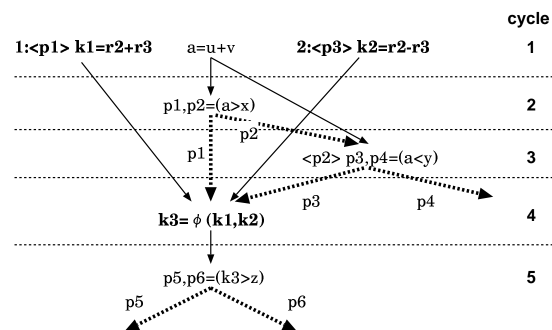

図１ ϕ ファンクションに対する制御依存

図 1 は、図 7 に示すサンプルコードのグラフの中から、ϕ ファンクションに関わる部分を抽出したものである。図中、実線の矢印は真依存を、点線の矢印は制御依存を表しており、点線の矢印に付加されているものは、その矢印の実行条件である。ϕ ファンクションの前のレジスタ演算命令 (図中の命令 1，２) は投機実行可能であるが、その値の選択をする ϕ ファンクションは、実行条件が求まるまで正しい値の選択ができないため、制御依存が発生し、コントロールフローが冗長になっている。

そこで、このコントロールフロー中の ϕ ファンクションに対しても、文献 4) で行っていたコードの複製を行

う。即ち、ϕ ファンクションの後続命令を複製して、ϕ ファンクションにより合流していたコントロールフローを分割する。ϕ ファンクションで得られた値を代入する部分には、それぞれのパスを通った場合に選択されるはずだった値を直接代入する。その際、複製される条件分岐演算から制御依存関係にある命令もコードの複製を行い、依存関係の分割を行う。このように命令の複製を行うことによって実行条件の参照を遅らせ、ϕ ファンクションを除去することができる。

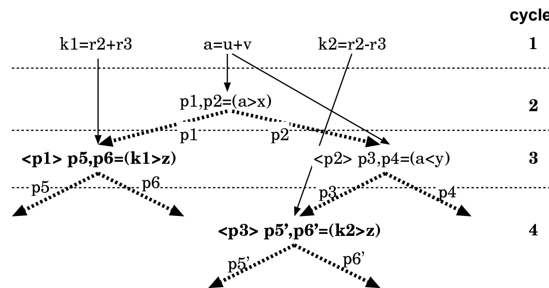

図 2 ϕ ファンクションの後続命令の複製

図 2 がコードの複製を行ったものであるが、複製前と比べると、少なくとも 5 サイクル必要だったものが 4 サイクルと 1 サイクル短縮されており、ILP の抽出に成功していることがわかる。

3.2 実行条件のブール最適化

IA-64 では、実行条件が格納されているプレディケートレジスタは 1 つの命令につき 1 つしか参照することができない。また、このレジスタの値は論理演算の引数とすることができないため、実行条件付き条件分岐演算は事前の条件分岐演算を越えて、または並列に実行することができない。図 3 は、図 1 と同様、図 7 の一部を抽出したものである。このように条件分岐演算が複数重なると、条件分岐演算の結果待ちにより、条件分岐演算命令を投機実行することができない。

そこで、実行条件の参照が必要な命令に対して、start ノードからその命令に至るまでの依存木を遡って条件分岐演算の結果を用いた論理式を算出する。これにより、実行条件付き条件分岐演算の実行条件参照は、条件分岐演算後の論理演算に置き換えられるため、実行条件付き条件分岐演算をその実行条件によらず、投機実行することができる。ただし、条件分岐演算の値をプレディケートレジスタに格納してしまうと論理演算に利用することができないため、この値は通常のレジスタに格納することにする。

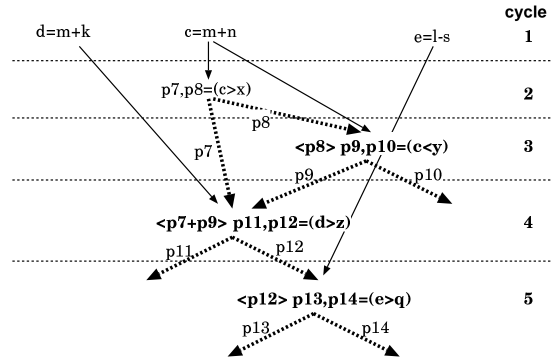

図 3 条件分岐演算に対する制御依存

図 4 は、図 3 のうち、p14 を求めるために、実行条件を論理演算に利用できる汎用レジスタに格納し、論理式化した実行条件の論理演算を組み込んだものである。他の実行条件もこれと同じかそれよりも短いサイクルで同様に求めることができ、少なくとも 5 サイクル必要だったものが 4 サイクルと 1 サイクル短くなることがわかる。

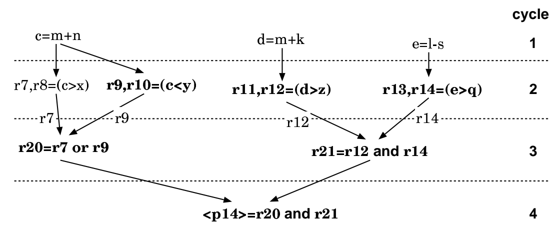

図 4 実行条件の論理演算

実行条件を論理式化するにあたって、求められた論理式をそのまま演算することは冗長な演算を含んでいることがある。そこで、この論理式に対してブール最適化を施して冗長な演算を除去する。例えば図 4 では p14 を求めているが、実際に依存木を遡って得られる論理式は $p14 = (p7 + p8 * p9) * p12 * p14$ である。このうち、$(p7 + p8 * p9)$ の部分に対しては、

$$
\begin{aligned}
p7 + p8 * p9 &= p7 * (p9 + p10) + p8 * p9 \\
&= p7 * p9 + p7 * p10 + p8 * p9 \\
&= p7 * (p9 + p10) + (p7 + p8) * p9 \\
&= p7 + p9
\end{aligned}
$$

のように変換することで冗長な演算を除去できる。これは、図 5 が示すように同値であることがわかる。

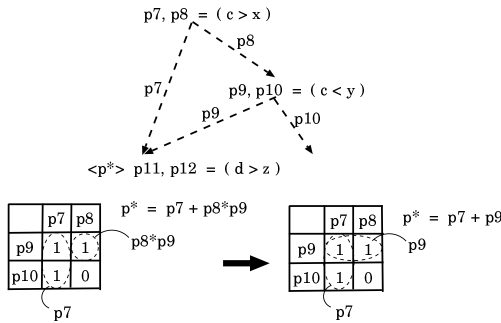

図５　２つの分岐条件によるブール最適化の例

4. 本手法の詳細

4.1 全体の流れ

全体の流れを、図6に示す。

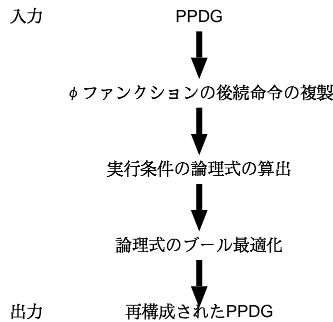

図６　本手法の流れ

本手法を適用するにあたって、入力にはSSA変換を施されたPPDG(Predicated PDG)を用いる。これは、PDGに実行条件という制御依存の情報を付加し、その依存関係を有向エッジを用いて表現することで、制御依存とデータ依存を同等に捉えることができるような拡張を加えたグラフである。

入力されたPPDGのグラフ中にあるφファンクションに対して、後続命令の複製を行ってグラフを変形する。変形したグラフから実行条件の参照が必要な命令を選択し、グラフを基に、その命令が参照する実行条件の論理式の算出をする。算出された論理式にブール最適化を施し、簡略化された論理式の論理演算を変形後のPPDGに組み込む。その後、不要となった制御依存のエッジを除去し、クリティカルパスの短縮を図る。

4.2 副作用の回避

本手法を適用するにあたって、本手法によって起こりうる副作用に注意を払う。従来の投機実行手法は、命令の順序変更によってレジスタ値が矛盾をきたさないように考慮をしてきたが、これはメモリの値にも当てはまる。この矛盾を引き起こさないために発生するのがメモリ依存と呼ばれる依存関係である。これは、store命令の後続命令としてload命令を演算する場合に発生する。それは、後続load命令がstore命令でstoreした値をloadするという可能性が否定できないためであり、これによりstore命令を越えて後続load命令を投機実行することはできない。IA-64では、このような命令に対しデータスペキュレーション[6]を行っている。これに対し、本手法では以下のようにソフトウェアからのアプローチを図る。

メモリ依存の解消のため、本手法ではload命令からの一連のパスに対して、φファンクションに適用したものと同様のコードの複製を行う。store命令と後続のload命令との依存関係をはっきりさせるため、load命令とstore命令のアドレス値を比較し、それを新たな実行条件として組み込む。これによってアドレス値が同じ時のパスと異なるときのパスを別に作成し、メモリ依存の解消を図る。そのアドレス値が異なる場合は、store命令とload命令の間に依存関係は存在しないということなので、load命令を投機的に実行することが可能である。グラフ中では、こちらを基本として投機実行を行い、その場合は実行条件を参照しない。アドレス値が同じ場合は、jumpコードを用いて投機実行した値を無効にして演算をやり直すコードへjumpする。

5. アルゴリズム

本手法では、以下のアルゴリズムに従ってPPDGの変換を行う。本手法を適用するためのサンプルコードとして図7のPPDGを変換して図示していく。

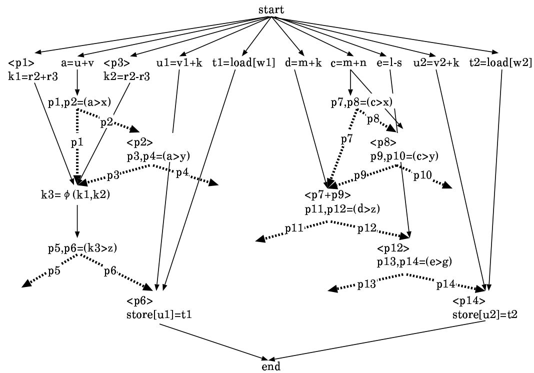

図７　サンプルコード

5.1 φファンクションのデュプリケーション

コントロールフロー中のφファンクションに対して、

本手法の特徴である ϕ ファンクションの後続命令の複製を行う。図 8 が変更後の図であるが、コードの複製によって、条件分岐演算 p5,p6(p5',p6') と store 命令が複製され、この部分のクリティカルパスが 1 サイクル短くなっていることがわかる。

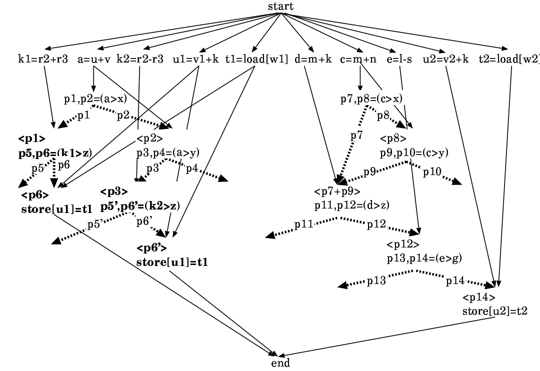

図 8　ϕ ファンクションの後続命令の複製

5.2 副作用の回避

ここでは、当該 load 命令に対しては投機実行を行い、アドレス値がそろった段階で store 命令と load 命令のアドレス値を比較する命令を挿入する。その比較が一致した場合には jump コードから演算をやり直すようなコードへ jump する。サンプルコードにおいて、$store = [u1]$ と $t2 = load[w2]$ の間にメモリ依存があったとすると、図 9 のように p15,p16、及び jump 命令が挿入される。

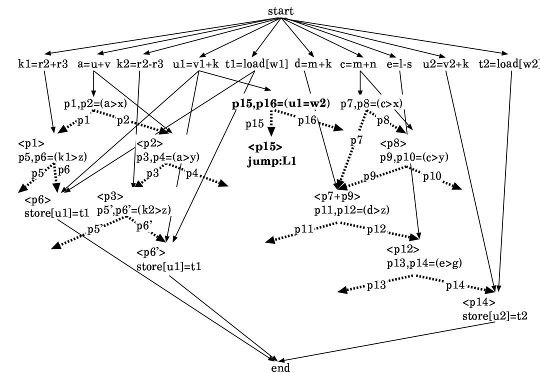

図 9　メモリ依存の回避

5.3 プレディケーションの参照が必要な命令の選択

命令中からプレディケーションを参照しなければならない命令を選別し直す。サンプルでは、後続命令を分割され除去された ϕ ファンクションと、論理演算で実装される実行条件演算の参照が必要なくなる。サンプルコードでは、 store 命令と jump コード以外の実行条件の参照が不要になる。

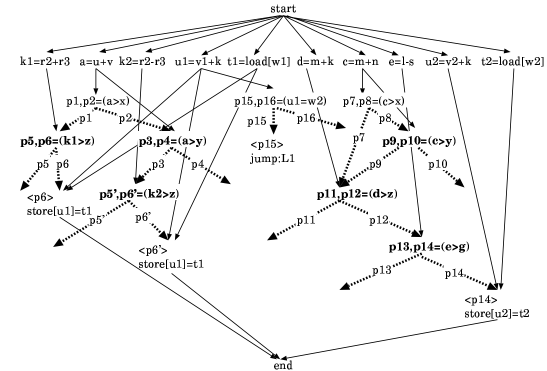

図 10　実行条件付き命令の選択

5.4 実行条件の論理式の算出

ここで、必要とされる実行条件の論理式を算出する。論理式はこれまで変換してきた PPDG の当該実行条件からコントロールフローを遡ることで、必要な実行条件とその論理式を得る。 jump コードはそのままなので、それを除いたサンプルコード中の実行条件は、

$$< p6 > = p1 * p6$$

$$< p6' > = p2 * p3 * p6'$$

$$< p14 > = (p7 + p8 * p9) * p12 * p14$$

という論理式で表すことができる。

5.5 論理式のブール最適化

上述により得られた論理式に対してブール最適化を施す。ブール最適化を施すことで論理演算を簡略化し、論理演算によるクリティカルパスの冗長な伸長を抑える。サンプルコードから得られた論理式の中で、

$$< p14 >= (p7 + p8 * p9) * p12 * p14$$

という論理式は、ブール最適化によって、

$$< p14 >= (p7 + p9) * p12 * p14$$

という形へと変形できる。

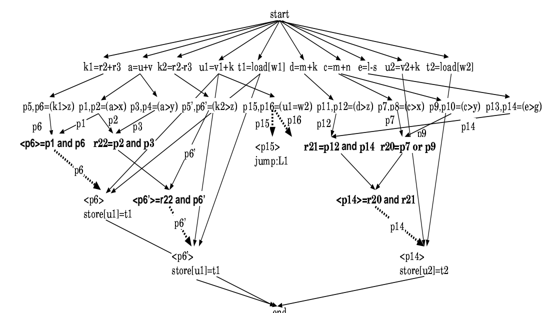

図 11　再構成された PPDG

5.6 PPDG の再構成

論理式の算出後、参照を必要としなくなった制御依存のエッジを消去し、クリティカルパスの短縮を図る。また、ブール最適化を施した論理式の演算を新たにPPDGに組み込む。図 11 がサンプルコードを再構成した図になる。

6. 評　　価

評価に用いたのは、 Stanford-Integer Benchmark から取り出した幾つかの最内ループである。ハードウェアは、 IA-64 に準拠したプロセッサとする。ただし、整数演算命令と store 命令は 1 サイクルを必要とし、 load 命令の遅延は、 1 サイクルとする。

評価値として用いたのは、 PPDG のクリティカルパス長である。プロセッサに十分な並列性がある場合、PPDG のクリティカルパス長は実行時間に比例するため、クリティカルパス長を比較することでその性能を比較することができる。

対抗手法として、文献 4) の GPDG を用いたコードスケジューリング手法を用い、我々の手法と比較した。表 1 にその結果を示す。

表 1　Stanford-Integer Benchmark のクリティカルパス長比較

benchmark	従来手法	本手法
quicksort	9	9
maxarray	13	12
permute	13	13
bubblesort	7	8
fit	20	14
remove	14	15

本手法が最も効果的にクリティカルパス長を短縮することができたのが、 fit である。このプログラムは、コントロールフローが極めて強くクリティカルパス長に影響を与えており、制御依存にとらわれない投機実行が効率よく行われた結果である。特に、ブール最適化を施した実行条件の値の論理演算が効果を発揮した。 maxarray では、クリティカルパスが 1 サイクル短縮されている。これは、従来行われていなかった制御演算に関する命令の複製による効果であり、本手法の特徴が良く反映されている。

反対に、 quicksort と permute ではクリティカルパス長に変化は無く、 bubblesort と remove では性能が僅か 1 サイクルではあるものの、劣る結果を示している。クリティカルパス長に変化のないプログラムに関しては、コントロールフローがクリティカルパスに影響していなかったためであると考えられる。この場合、コントロールフローに関して投機実行を行ってもクリティカルパス長に影響がなく、このような結果になる。また、クリティカルパス長が 1 サイクル伸びているプログラムは、実行条件の論理演算をソフトウェアで演算したことが原因である。ハードウェアによって論理演算を実装すれば従来手法と同様になる。現行のアーキテクチャで実装を行う場合は、コンパイラによる静的な解析により本手法を適用するか判別を行えば、これを回避できる。

7. 終わりに

本稿では制御依存を緩和することで、依存グラフのクリティカルパスを短縮する手法を提案した。本手法は、コントロールフローがクリティカルパスに影響を与えないようなプログラムの場合には効果が発揮できない。今後はコントロールフローを考慮し、本手法を適用する場合としない場合を選択するような機構を組み込んだ手法を研究していきたい。

参　考　文　献

1) R.Cyton, 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.
2) D.I.August, J.W.Sias, J.Puiatti, S.A.Mahlke, D.A.Connors, K.M.Crozier and W.W.Hwu: "The Program Decision Logic Approach to Predicated Execution", *International Symposium om Computer Architecture*, 1999, pp.208-219
3) K.Knobe and V.Sarkar: "Array SSA form and its use in Parallelization", *In 25th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, January 1998.
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.6, 1996, pp.1149-1161
5) 古関聰, 小松秀昭, 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リング技法とその評価", JSPP'94 論文集, 1994, pp.1-8
6) R.Zahir, D.morris, J.Ross and D.Hess: "OS and Compiler Considerations in the Design of the IA-64 Architecture", *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000, pp.212-221
7) 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
8) A.W.Appel: "Modern Compiler Implementation in C", *Cambridge University Press.*