

ソフトウェアを援用した投機的例外処理機構の実現と評価

古 関 聰[†] 影 井 慶 大[†] 仲 顕 照[†]
小 松 秀 昭^{††} 深 澤 良 彰[†]

計算機の高速化の重要な技術に、命令レベルでの並列処理が挙げられる。VLIWは主に数値計算を対象として研究開発されてきた命令レベル並列プロセッサであるが、最近では次世代の主力汎用プロセッサとして注目されている。汎用プロセッサとしてのVLIWの性能を引き出すためには、命令の投機的移動を主体とした大域的コードスケジューリング技法が重要である。しかしながら、命令の投機的移動は単純なVLIWでは制限があるため、ハードウェアによる支援が不可欠である。この制限を緩和するための支援機構として、これまでハードウェア的なアプローチ、また、ソフトウェア的なアプローチを採用する様々な方法が提案されてきた。しかしながら、前者には、ハードウェア量が多く、メカニズムが複雑になるという問題点、また、後者には、投機的移動の自由度が低い、あるいは、レジスタの使用効率が低いという問題点があった。我々は、ハードウェア量や複雑さに関する問題点を回避するため、後者のアプローチを探りながら独自の投機的命令支援機構を提案してきた。本論文では、本方式と後者のアプローチを採用した従来の方式を比較し、本方式において、投機的移動の自由度とレジスタの使用効率について改善がなされていることを示すことを試みる。

A Realization and Evaluation for the Software-Aided Speculative Exception Handling Mechanism

AKIRA KOSEKI,[†] YOSHIHIRO KAGEI,[†] AKITERU NAKA,[†]
HIDEAKI KOMATSU^{††} and YOSHIAKI FUKAZAWA[†]

Instruction-level parallel processing is one of the most important techniques to achieve a good performance on computing. The VLIW processor has been developed and researched as a special instruction-level processor for numerical programs, yet it has nowadays been expected to be the next mainstay of general purpose processors. To derive a high performance from a program for VLIW processors, the global code scheduler using speculative moves is really needed. However a simple VLIW processor does not allow schedulers to conduct speculative moves, so several speculative-move-supporting mechanisms have been proposed for relaxing this scheduling restriction, by using hardware-oriented or software-oriented approach. The former has problems regarding the amount and complexity of hardware, while the latter's concerns include the restriction of conducting speculative moves, and low usage of register resources. We have proposed a mechanism adopting the latter approach to avoid those hardware-related problems. In this paper, compared to existing method using software-oriented approach, we verify that our method improves the problems as to the restriction of speculative moves and the usage of register resources.

1. はじめに

計算機の高速化を実現する手段の一つとして、並列処理の研究が分散処理レベルから命令レベルまで様々な粒度で行われている。その中で我々は、命令レベル並列処理に着目し、研究を行っている。

命令レベル並列処理で高い並列性を引き出すために

は、命令スロットに絶えず命令を投入し、その充填率を高めることが重要である。そのためには、基本ブロック間での命令の移動、即ち条件分岐を超えて命令を移動する命令の投機的移動^{1)~5)}を用いたコードスケジューリングが必要になる。この命令の投機的移動を用いることにより、投機的移動を用いない場合に比べて並列度を2~8倍に増やすことが出来るという報告がされている⁶⁾。しかしながら、この命令の投機的移動を行うためのハードウェアあるいはソフトウェアのサポートがない場合は、ロード命令やストア命令などのメモリにアクセスする命令あるいは除算などの命令の投機的移動ができないという制限がある。この制限を緩和するために、

[†] 早稲田大学理工学部
School of Science & Engineering, Waseda University
^{††} 日本IBM(株)東京基礎研究所
Tokyo Research Laboratory, IBM Japan Ltd.

いくつかの投機的実行支援機構が提案されているが、これまでの方式には、ハードウェアの量及び複雑さが増大する、投機移動時に越えられる分岐の数及び方向に制限が生じる、または、プロセッサの状態を元に戻すために値を保持すべきレジスタの数が多くなってしまうという問題点があった。本方式では、これらの問題点をソフトウェアを援用しながら解決することを試みる。

2. 本研究の背景

命令を投機的に移動した場合に起きる大きな問題点は、投機的に移動した命令（以下、投機命令）がページフォールトや0除算などの例外を起こし、その復旧処理が行われたにも関わらず、後に他の命令がその命令の結果を参照しなかった場合、その処理にかかった時間が無駄になってしまうことにある。即ち、コンパイル時に予測された分岐の方向と実際の分岐の方向が異なった場合、投機命令が起こした例外の復旧処理にかかった時間が実行時間に多大な遅延を及ぼしてしまう、もしくはマシンを停止させてしまう事が問題である。この問題に対するサポートがない場合は、例外を起こす可能性のある命令を投機的に移動できないという制限が生じてしまい、スケジューリングの自由度を大きく低減させてしまうことになる。従って、この問題に対する何らかの支援が必要である。

この問題に対しては、例外を起こした投機命令の結果が実際に使われると確定する時点まで復旧処理を延期することで解決を図るが、その解決を図る従来の方式として主に以下の4つがある。

- ブースティング方式
- プレディケーティング方式
- non-excepting 命令方式
- センチネルスケジューリング方式

このうち、ブースティング方式⁷⁾とプレディケーティング方式⁸⁾はハードウェア的なアプローチであり、non-excepting 命令方式⁹⁾とセンチネルスケジューリング方式¹⁰⁾はソフトウェア的なアプローチであると言える。以下、各方式を説明する。

2.1 ブースティング方式

ブースティング方式では、TORCHアーキテクチャなどに見られるように、復旧処理の延期をシャドウレジスタファイルという構造を用いて実現している。シャドウレジスタファイルとは、各レジスタにデータを格納する部分を2個所保有しているレジスタの構造を言う。この2個のデータを格納する部分のうち片方をシャドウレジスタという。ブースティング方式では、投機的に移動した命令の実行結果はシャドウレジスタに書き込み、この投機命令が本当に実行されると確定した時点でシャドウレジスタから通常のレジスタ部分にデータのコピー（コミット）が行われ、実行されないと確定した時点でシャドウレジスタの値を破棄する。投機命令が実行され

るかしないかは、その投機命令が越えてきた分岐の分岐条件と実際に行われた分岐の分岐条件とが一致するかしないかで判断する。投機命令が例外を起こした際には、そのディスティネーションレジスタのエラーフラグを立てる。また、そのレジスタを投機命令が参照した場合には、同様にエラーフラグを立てる。この際に、再実行のためのリカバリコードというものを生成する。例外を起こした投機命令が本当に実行されると確定した時点でリカバリコードをもとに再実行を行なう。しかしながら、ブースティング方式では、コンパイラが予測した一意の制御パスからの投機的移動しか行なえないので、スケジューリングの自由度が低くなるという問題点がある。さらに、シャドウレジスタファイルを用いているので、ハードウェアの量が多くなるという問題点も合わせ持つ。

2.2 プレディケーティング方式

プレディケーティング方式では、ブースティング方式と同様にシャドウレジスタを用いて復旧処理の延期を実現している。ブースティング方式と異なるのは、投機命令が越えてきた分岐の分岐条件の論理式であるプレディケートをプレディケーティング方式では持っているという点である。このプレディケートを持っていることで複数制御パスからの命令の投機的移動が行なえるので、スケジューリングの自由度は非常に高くなる。

しかしながら、シャドウレジスタファイルを用いているので、ハードウェアの量が多くなるという問題点がある。また、プレディケートを持つことで、投機的命令が越えられる分岐の方向や数が限られ、広範囲での命令の投機的移動に制限が生じる可能性がある。

ブースティング方式とプレディケーティング方式では、以上のように投機命令が例外を生じた場合、シャドウレジスタファイル構造を持ちいており、データのコミットや無効化もハードウェア的に行なわれる。即ち、ハードウェア的なアプローチでその制御を行なっていると言える。

2.3 non-excepting 命令方式

non-excepting 命令方式では、投機命令が例外を起こしたその命令アドレスをディスティネーションレジスタに格納することで例外の復旧処理を延期する。具体的には、投機命令が例外を起こした場合、その投機命令の命令アドレスをディスティネーションレジスタに書き込み、書き込んだ命令が例外を起こしたことを示すビット（以下、例外ビット）を立てる。非投機命令が例外ビットが立っているレジスタを参照した時に、復旧処理を開始する。復旧処理は、ソースレジスタに格納されている命令アドレスをもとに行なう。non-excepting 命令方式では、投機命令に依存する命令の投機的移動が行なえないので、スケジューリングの自由度が格段に低くなるという問題がある。

2.4 センチネルスケジューリング方式

センチネルスケジューリング方式では、non-

excepting 命令方式と同様に、例外を起こした投機命令の命令アドレスをディスティネーションレジスタに格納することで復旧処理を延期する。non-excepting 命令方式と異なるのは、投機命令に依存する命令の投機的移動が可能になっている点である。センチネルスケジューリング方式では、例外を起こした投機命令の結果を投機命令が参照した場合、例外を起こした投機命令の命令アドレスを読み込んだ投機命令のディスティネーションレジスタに書き込み、例外ビットを立てる。さらに、投機命令が例外ビットが立っているレジスタを読み込んだ場合は、同様に復旧処理を延期し、最終的に非投機命令が例外ビットが立っているレジスタを読み込んだ時に復旧処理を開始する。復旧処理は、例外を起こした命令アドレスをもとに、その命令から例外を検出した命令までの全ての命令を再実行することで実現する。従って、センチネルスケジューリング方式では、その間に存在する全ての命令が参照するレジスタに再書き込みを行なってはならないという制限がある。従って、センチネルスケジューリング方式ではレジスタの使用効率が悪くなるという問題が生じる。

non-excepting 命令方式とブレディケーティング方式では、以上のように、投機命令が例外を起こした場合、命令アドレスを用いて復旧処理の延期と再開を行なっている。これは、プログラムの情報をもとに処理を行なっているため、ソフトウェア的なアプローチで制御を行なっていると言える。

3. 基本方針

本方式では、投機命令が例外を起こした場合、復旧処理の延期と再開をソフトウェア的なアプローチで行なう。具体的には、例外を起こした命令アドレスを用いて、例外を起こした投機命令の結果を非投機命令が参照するまで復旧処理を延期する方式をとる。

また、復旧処理を再開するときに、例外を起こした命令を特定するのに、例外を検出した命令からポインタをたどっていくことで行なう。これにより、例外を起こした投機命令に依存する命令だけを再実行すれば良いので、実行の高速化が図られ、レジスタの使用効率が低くなるということはない。また、ソフトウェアで制御を行なっているため、分岐を越える数および方向に制限がなく、シャドウレジスタファイル構造を用いていないので、ハードウェアの使用量が増加することもない。本方式では、この機能を例外が起こった時に起動される割り込みハンドラに処理法を記述することにより復旧処理を実現している。

4. ソフトウェアによる投機的例外処理

本方式を実現する上で、レジスタが保持している値が例外を起こした命令の結果かどうかを区別するビットを各レジスタに1ビット（以下、例外ビット）付加し、す

```
*i1 : r1=load[A]
*i2 : r2=load[B]
i3 : r3=load[C+r0]
i4 : r3=r3+1
i5 : r0=1
*i6 : r4=r1+r2
i7 : if(r10==0) goto .....
i8 : r5=r3+r4
```

図1 サンプルコード

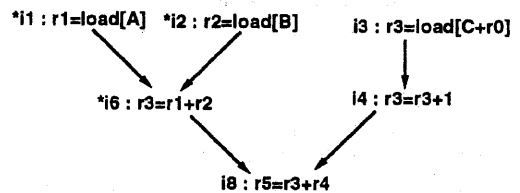


図2 依存グラフ

べての命令にその命令が投機命令か非投機命令かを区別するビットを1ビット付加する。以下に、投機命令が例外を起こした場合、その例外処理をどのように延期させるかを、また、処理の再開をどのように行うかを述べる。

4.1 復旧処理の延期

復旧処理の延期は、次のようにして実現する。

- (1) 投機命令が例外を起こした場合、ディスティネーションレジスタにその命令の命令アドレスを書き込み、さらにそのレジスタの例外ビットを立てる。
- (2) 例外ビットが立っているレジスタを投機命令が読み込んだ場合、その命令の命令アドレスをディスティネーションレジスタに書き込み、例外ビットを立てる。
- (3) 非投機命令が例外ビットの立っているレジスタの内容を読み込んだ場合、復旧処理を開始する。

図1のサンプルコードを用いて具体的な復旧処理の延期について説明していく。図中で*がついている命令はその命令が投機命令であることを示す。今、命令i1が例外を起こしたとする。この命令は投機命令なので、その場で復旧処理を行わず、この命令のディスティネーションレジスタr1に命令i1の命令アドレスを書き込み、さらにr1の例外ビットを立てる。命令i2、i3、i4、i5は通常の実行が行われる。命令i6は例外ビットが立っているレジスタを読み込んでいる投機命令なので、この場で復旧処理を行わず、ディスティネーションレジスタr4に命令i6の命令アドレスを書き込み、r4の例外ビットを立てる。命令i8が実行されると、この命令は例外ビットが立っているレジスタを読み込んでいる非投機命令なので、この時点で復旧処理の再開が行われ、割り込みハ

r1 : IA of i1
r2 : 2
r3 : 4
r4 : IA of i6
r5 : IA of i8

i1
i6
goto IA of i8

a) レジスタの状態

b) リカバリバッファ

IA : Instruction Address

図3 レジスタとメモリの状態

ンドラが起動される。

4.2 復旧処理の再開

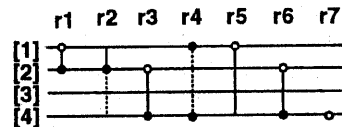
復旧処理の再開は、非投機命令が例外ビットの立っているレジスタを読み込んだ時点で行なわれる。本方式では、復旧処理を行う際に、例外を検出した命令に格納されているポインタから、再実行しなくてはならない命令を、例外を起こした命令まで順次たどる。その際、ポインタで指し示された命令をリカバリバッファと呼ばれるバッファに格納していく。例外を起こした命令までたどり終えたら、リカバリバッファに格納されている命令を順次実行する。本方式では、この例外の復旧処理をソフトウェアで制御する。これは、例外を検出した時点で起動される割込みハンドラに復旧処理プログラムを記述しておくというものである。具体的には、以下の機能が割込みハンドラに付されている。

- (1) 非投機命令が読み込んだソースレジスタに格納されているポインタが示す命令を読み込む。
- (2) この非投機命令をリカバリバッファに格納する。
- (3) ポインタが示す命令におけるソースレジスタに例外ビットが立っているレジスタがあればその中に格納されているポインタが指す命令を読み込む。
- (4) 読み込みもとの命令をメモリに順次書き込む。
- (5) すべてのポインタをたどり終えるまで (3) (4) の操作を繰り返す。

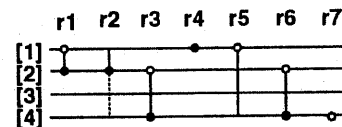
ここで、命令を積む順序は依存グラフで示されるツリーで深さ優先探索を行い、その途中にある命令を順次積んでいく。また、一番最後の番地には通常の処理に戻る復帰命令を格納しておく。図2および図3の例を用いて例外の復旧処理のプロセスを示す。ただし、この例では命令i1のみが例外を起こしたとする。この場合、命令i8が例外を検出するわけだが、まず、この命令のソースレジスタに格納されているポインタが指す命令i6を読み込む。これと同時に命令i8をリカバリバッファに格納する。次に、読み込まれた命令i6のソースレジスタに例外ビットが立っているレジスタr1があるので、その中に格納されているポインタが指す命令i1を読み込む。また、命令i6もリカバリバッファに格納する。次

```
[1] *i1 : r1=load[A]      i3 : r5=r4-1
[2] *i2 : r3=r1+r2        i4 : r6=load[B]
[3] i5 : if(r10==0) goto .....
[4] i6 : r7=r3+r6
```

中間コード



a) センチネルスケジューリング方式



b) 本方式

図4 レジスタの生存区間

に、命令i1のソースレジスタには例外ビットが立っているレジスタが無いので、この命令i1が例外を起こした命令であると判明する。この際、命令i1をリカバリバッファに格納する。これで、再実行すべき命令すべてをたどり終えたので、リカバリバッファに格納された命令を順次実行していく。これにより、プロセッサの状態を元に戻すことができる。

5. 他のソフトウェアアプローチとの比較

5.1 non-excepting 命令方式との比較

non-excepting 命令方式では、投機命令に依存する命令の投機的移動ができないという制限がある。従って、図1のように命令i1に依存する命令i6の投機的移動は行えない。

5.2 センチネルスケジューリング方式との比較

センチネルスケジューリング方式では、本方式に比べて再実行しなくてはならない命令が多いので、書き換えてはならないレジスタが多くなり、レジスタの使用効率が低い。例えば、図4の例を用いると、センチネルスケジューリング方式では、命令i1から命令i6までの全ての命令を再実行しなければならない。この結果、レジスタr4の生存区間が伸びてしまい（図4の点線部分）、レジスタアロケーションを行うときに必要な実レジスタの数は5個になる。一方本方式では、再実行する命令はi1、i2、i6だけで良いので、レジスタr4への書き込みが行える。この結果、r4の生存区間が伸びることはなく、レジスタアロケーションを行うときに必要な実レジスタの数は4個になる。

これにより、本方式とセンチネルスケジューリング方式

式とでは、本方式の方がレジスタの使用効率が高い。

6. 評価

本章では、まずスケジューリングの自由度を比較するため、表1に示した各評価モデルにおける実行性能を測定した。表1において、局所的コードスケジューリングはリストスケジューリング¹¹⁾、大域的スケジューリングは文献⁶⁾で提案された方式、また、non-excepting 命令方式は2章において述べられた方式、本方式は4章で説明した方式を意味するものとする。

具体的には、表1で示した各モデルについて並列度を1,2,4,6と変化させ、以下のプログラムについて、各最内ループを10～15回展開し、スカラプロセッサでの実行速度を1としたときの相対実行速度を測定した。なお、ここでいう並列度とは、同時発行可能な命令の数を表すものとし、算術演算、ロード/ストア、ジャンプユニットを並列度と回数持つものとした。

表1 評価対象

モデル番号	評価条件
1	VLIW+局所的コードスケジューリング
2	VLIW+大域的コードスケジューリング
3	VLIW+non-excepting 命令方式
4	VLIW+本方式

表2～8より、局所的コードスケジューリングでは、マシンの並列度を高くしても実行速度が向上していないことがわかる。これに比べ、モデル2では、並列度が高

表2 リストの最大値

モデル番号	並列度			
	1	2	4	6
1	1.000	1.000	1.000	1.000
2	1.308	1.700	1.700	1.700
3	1.308	2.125	2.125	2.125
4	1.308	2.429	2.833	2.833

表3 選択ソート

モデル番号	並列度			
	1	2	4	6
1	1.000	1.222	1.222	1.222
2	1.222	1.833	2.200	2.200
3	1.222	2.200	2.750	2.750
4	1.222	2.200	2.200	3.667

表4 素因数分解

モデル番号	並列度			
	1	2	4	6
1	1.000	1.000	1.000	1.000
2	1.000	1.286	1.286	1.286
3	1.000	1.286	1.286	1.286
4	1.000	1.286	1.500	1.500

くなるにつれ実行速度は向上しており、また、モデル1と比べて若干の速度向上が見られる。これは、大域的コードスケジューリングによる、命令の投機的移動の効果によるものである。しかしながら、2章で述べた通り、この移動は非常に制限されており、危険な投機的移動を行なうことはできない。モデル3では、この制限はある程度緩和されている。これにより、並列度が6の場合に、1.9倍の速度向上が得られている。本方式では投機的移動の制限を非常に少なくすることに成功している。この方式を採用することで、並列度が6の場合に、2.5倍の速度向上が達成されており、本方式がこのようなプログラムの並列実行に非常に有効であることがわかる。

次に、レジスタの仕様効率を比較するため、以下のプログラムに対し、プロセッサの並列度を無限大とした場合のセンチネルスケジューリング方式、本方式の使用レジスタ数を測定した。

表9から、必要となる実レジスタの数がセンチネルスケジューリング方式より本方式の方が少ないことがわかる。実際には、本方式の採用により、同方式における使用レジスタ数において、平均で88.3%、最大で65.4%

表5 二分木探索

モデル番号	並列度			
	1	2	4	6
1	1.000	1.000	1.000	1.000
2	1.077	1.556	1.556	1.556
3	1.077	1.556	1.556	1.556
4	1.077	1.750	2.000	2.000

表6 バブルソート

モデル番号	並列度			
	1	2	4	6
1	1.000	1.429	1.429	1.429
2	1.000	1.667	1.667	1.667
3	1.000	1.667	1.961	1.961
4	1.000	1.961	2.564	3.333

表7 ユークリッドの互除法

モデル番号	並列度			
	1	2	4	6
1	1.000	1.000	1.000	1.000
2	1.000	1.400	1.400	1.400
3	1.000	1.400	1.400	1.400
4	1.000	1.707	1.707	1.707

表8 リストの要素の過剰判定

モデル番号	並列度			
	1	2	4	6
1	1.000	1.000	1.000	1.000
2	1.100	1.623	1.623	1.623
3	1.100	1.650	1.980	1.980
4	1.100	1.687	2.415	2.415

の減少に成功している。この結果は、本方式がセンチネルスケジューリングよりも、投機的に命令を移動した際に保存しなければならないオペランドを少なくすることができることを示しており、プログラム最適化に欠かせない技術となっている大域的コード最適化をより自由に適用できることを証明するものである。

7. おわりに

命令レベル並列処理において高い並列性を引き出すためには、分岐条件を越えて命令を移動する命令の投機的移動を用いたコードスケジューリングが必要になる。命令の投機的移動を行なう際には、投機命令が例外を起こした際にその復旧処理を延期させる何らかの支援が必要である。本手法では、ソフトウェアでその制御を行なうことにより、ハードウェアの量および複雑さが増大するという点、あるいは越えられる分岐の数や方向に制限があるという問題点を解消した。また、センチネルスケジューリング方式に見られるレジスタの使用効率が低いという問題点を、必要な命令だけを再実行することで解消しており、命令レベル並列プロセッサを対象とした、有効な投機的処理機構を提案したと考える。

表9 使用レジスタ数

対象プログラム	使用レジスタ数	
	センチネル	本方式
リストの最大値	16	13
バブルソート	26	17
選択ソート	21	16
リストの奇偶判定	27	23
ラグランジュの補間	33	31
シンプソンの公式	31	30
ベッセンベルグ行列への変換	40	38
ニュートンの一般補間公式	43	41
LU 分解	26	23
QR 法	27	24

参 考 文 献

- 1) Fisher, J.A.: Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Trans. Comput.*, Vol.C-30, No. 7, pp.478-490, (1981).
- 2) Aien, A. and Nicolau, A.: A Development Environment for Horizontal Microcode, *IEEE Trans. Softw. Eng.*, Vol. 14, No. 5, pp. 584-594 (1988).
- 3) Ebicioğlu, K. and Nicolau, A.: A Global Resource-Constrained Parallelization Technique, *Proceedings of the 3rd International Conference on Supercomputing*, pp.154-163 (1989).
- 4) Hwu, W. W., Mahlke, S. A., Chen, W. Y., Chang, P. P., Walter, N. J., Bringmann, R. A., Ouellette, R. G., Hank, R. E., Kiyohara, T., Haab, G. E., Holm, J. G. and Lavery, D. M.: Hwu, W. W., et al.: The Superblock: An Effective Technique for VLIW and Superscalar Compilation, *Journal of Supercomputing*, Vol. 7, No. 1, pp.229-248 (1993).
- 5) Mahlke, S. A., Lin, D. C., Chen, W. Y., Hank, R. E. and Bringmann, R. A.: Effective Compiler Support for Predicated Execution Using the Hyperblock, *Proceedings of the 25th Annual International Symposium on Micro Architecture*, pp.45-54 (1992).
- 6) 小松, 古関, 深澤: 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, pp. 1149-1161 (1996).
- 7) Smith, M. D., Lam, M. S. and Horowitz, M. A.: Boosting Beyond Static Scheduling in a Superscalar Processor, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp.344-354 (1987).
- 8) 安藤, 中西, 原, 中屋: プレディケート付き状態バッファリングによる投機的実行, 並列処理シンポジウム JSPP'95 予稿集, pp. 107-114 (1995).
- 9) Colwell, R. P., Nix, R. P., O'Donnel, J. J., Papworth, D. B. and Rodman, P. K.: A VLIW Architecture for a Trace Scheduling Compiler, *IEEE Trans. Comput.*, Vol. C-37, No. 8, pp. 967-979 (1988).
- 10) Mahlke, S. A., Chen, W. Y., Hwu, W. W., Rau, B. R. and Schlansker, M. S.: Sentinel Scheduling for VLIW and Superscalar Processors, *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.238-247 (1992).
- 11) Coffman, E. G., Jr.: *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons (1976).