4F－3

バイナリコンパチビリティを保ちながら loop を投機実行するアーキテクチャ

玉造潤史　松本尚　平木敬

東京大学大学院 理学系研究科 情報科学専攻

1 はじめに

現在、Chip 内の増え続ける計算資源を有効に利用するために VLIW やスーパースカラといったアーキテクチャが用いられている。今後とも増加するこの計算資源を高性能化に結び付けるためには従来の方法では限界があり、新たな MIMD アプローチが必要である。

- Chip 内では従来のプロセッサでは不可能な結合を作ることができ、この性質を利用して、プロセッサ間に跨る大きな依存の解決が可能である。

さらに、従来の命令レベルのレイテンシに対しての投機実行という小さな並列性だけでなく loop といった大きな構造に対する投機実行を実現する。これにより、完全に動的依存を排除出来ない loop 構造の並列実行が可能となる。さらに、この依存解決を依存の全く排除されていない逐次実行型バイナリで実現することでバイナリコンパチビリティが保つことが可能となる。

逐次実行のバイナリを並列実行するためには、loop 間の依存関係を保持しなければならない。loop 間依存関係のうち、逐次型 processor が loop 間での値引き渡しを行なう register の依存関係は instruction の履歴により解析できる。

本稿では、履歴によって解析できる register の依存は解消し、メモリアクセスや分岐の等の履歴からでは解消できない依存関係を動的に検出し投機実行することにより逐次 loop の並列実行を行なうアーキテクチャを提案する。

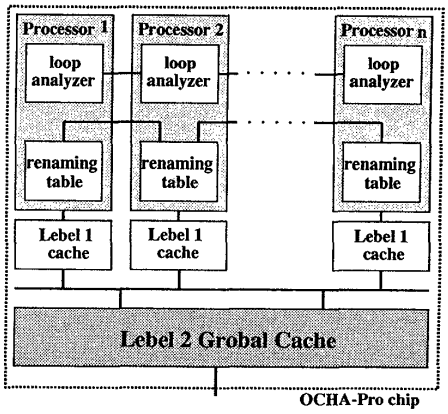
さらに、本機能を投機実行機構として付加した共有メモリ型マイクロプロセッサパイロットモデル OCHA-Pro(On-Chip MIMD Architecture Processor) [1] を述べる。

2 loop の動的依存解析

現在のマイクロプロセッサは、レジスタ、パイプラインなどの資源管理のために使用中の資源の履歴を取り、そこから資源利用の可否とデータ依存関係を解析し実行開始を判定している。したがって、ある register が現在利用可能な資源であるかどうかという依存関係だけの解析を動的に行なっている。そのため、コンパイラがプログラミング言語によって記述されているプログラム中の変数の依存関係を解析し、loop の並列化を行なわなければならない。しかし、プロセッサが動的な依存解析を備えることにより動的に変化する依存のために静的には並列化が行なえない loop も並列に実行できる。さらに、この機能により逐次のバイナリの動的な並列実行をサポートできる。これらのコンパイラが行なっているプログラム中の依存解析をプロセッサの命令流のレベルでの依存解析として行なわなければならない。バイナリコード上で依存関係として現れてくるのは、*(1).* register 上の data 依存 *(2).* memory 上の data 依存 *(3).* 実行時の制御依存により定まる data 依存である。これらの内、register 上の data 依存は命令流の解析によって見つけることができる。他の 2 点に関しては、実行時の状況に依存するので解析は行なわず、投機的な実行を行ない結果として依存を保つ。

この動的な依存解析は loop analyzer によって行なわれる。loop analyzer は、命令の履歴をとり、その履歴から loop の検出と register の依存を解析する。loop 間の依存となるレジスタは、Iteration を跨る依存であるので、解析の結果によって renaming 動作に隣接プロセッサとの overlap register に同時に書き込みを行なわせる。これによりレジスタ転送のレイテンシを隠蔽する。

3 OCHA-Pro の概要



OCHA-Pro は次に示す特徴を持つ On-Chip MIMD プロセッサである。

1. 各要素プロセッサが独立のスレッドを実行できる MIMD プロセッサ、要素プロセッサ数は当初 4 ～ 8 を予定。
2. 共有 2 次 cache を持つ。
3. loop の検出と register 依存を解析する loop analyzer
4. Elastic Barier[2] によって同期して更新される renaming table

An Architecture for speculative loop execution with Bianry compatibility

Junji TAMATSUKURI , Takashi MATSUMOTO , Kei HIRAKI
Department of Information Science,Faculty of Science, the University of Tokyo

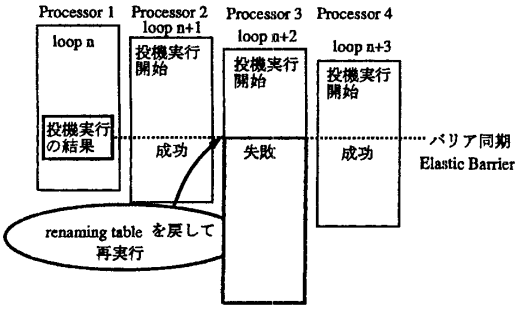
本方式による逐次ループの並列実行は各プロセッサの分散管理である。loop analyzer はその管理の中心でありプログラムの解析、renaming の生成、投機実行の管理を行なっている。

4 多重投機実行の実現

OCHA-Pro では、同時に複数の投機実行を register renaming table を用いて実現する。Renaming table には、現在のレジスタの renaming の状態を記録しておく。投機的な実行は renaming された register 上で行なわれる。投機実行の前に renaming table を更新する。

- 現在使用中のレジスタは書き込みを禁止して、destination として用いる時は renaming する。この動作によって context を保持する。
- 投機が失敗した場合、同期して renaming table を巻き戻す。この動作で投機実行前の状態に戻る。

したがって、この table の数までの多重投機実行が可能である。投機実行が成功し必要が無くなった renaming table は、flush (全てを消す) する。



5 投機に失敗しない並列動作

OCHA-Pro の並列動作時 Speed Up は

$$speedup = \frac{loopのclock数}{(準備 + loopのclock数)} \times processor数$$

となり、Speed Up は loop 回数に依存しない。

loop 間依存のない loop(tomcatv) では、要素プロセッサが整数 unit × 2、浮動小数点 unit× 1、ロードのレイテンシが 5 clock、整数 unit は 1 clock 1 命令を実行可能とした時、逐次 loop の 1 iteration が 338 clock、iteration の準備に 25 clock かかる。これを前式に当てはめると、93 % の効率で全プロセッサが並列実行できる。

6 if body の投機実行

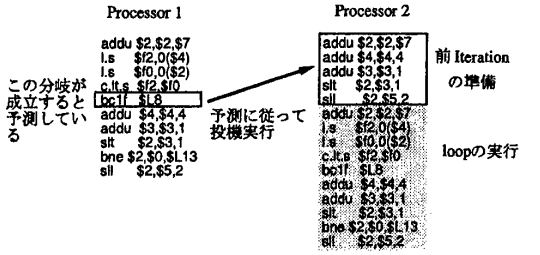
次例として、if 文を含む最内 loop を投機実行する。ここでは、livermore loop No.24 を例として用いる。

最内 loop 内にある if 文の成立は静的には予測不能である。プロセッサは動的に分岐に対して予測を行なう。この予測を利用してそれぞれの loop を予測が成功したとして投機実行を行なう。

branch 命令後の if body で生成される register はその後のその register を用いる実行に依存を作る。この解析は loop 内全てを 1 度解析するために回ることで行なうことができる。並列実行では解析結果にしたがって、if body 内で生成されるレジスタへの命令の実行は分岐の予測に従った投機実行を行ない if の結果予測と異なる依存となった場合生成されたレジスタは loop analyzer で検出され, その投機実行の開始時に状態を戻して実行する。

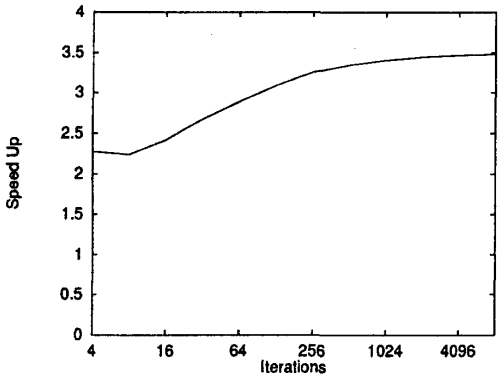
```
*  Kernel 24 -- find location of first minimum in array

for ( k=1 ; k<n ; k++ ) {
    if ( x[k] < x[m] ) m = k;
}
```



並列実行中、各要素プロセッサは、前イテレーションの状態の作成、loop body の実行を行なう。

livermore loop No.24 を前例と同じ要素プロセッサ 4 台で 2 bit の動的分岐予測を用いて投機実行すると下図の結果が得られる。



7 おわりに

今後は loop を持つ branch の並列化だけでなく branch を含む命令流の並列化に向けた一般化した投機実行機構としていく予定である。

References

[1] 玉造 潤史, 松本 尚, 平木 敬.,"Loop を並列実行するアーキテクチャ", 情報処理学会研究会報告 SWoPP 秋田 '96.

[2] 松本 尚.,"Elastic Barrier: 一般化されたバリア型同期機構", 情処学会論文誌 Vol32,No.7,pp.886-896,July 1991