翻訳

具体例によるIA-64アーキテクチャの紹介

Carole Dulong *Intel Corp.*
cdulong@mipos2.intel.com

翻訳：安藤　進
sando@twics.com

原文："The IA-64 Architecture at Work"
Computer, Vol.31, No.7, pp.24-32 (July 1998)

インテルIA-64アーキテクチャは，条件付実行制御（predication）と投機的実行制御（control speculation）という2つの制御機能をコンパイラに提供することで，命令レベルの並列処理（instruction-level parallelism）☆1を引き出すことを可能にした．ここでは，もともと逐次性が高いポインタ追跡問題および予測が困難な分岐を含むネスト化されたループのコード例を紹介しながら，コパイラがIA-64命令をどのように活用するかを具体的に解説する．

マイクロプロセッサの性能向上を目指す戦略はここ数年にわたり，命令レベルの並列処理を引き出すことに的が絞られてきた．簡単にいうと，同時に実行可能な命令を見つけ出すことだ．コンピュータ・アーキテクチャの設計者は，命令の実行に使用する機能ユニット（functional unit）を増やすことで性能を向上できると考えている．

しかし，次の2つの懸案がある．

- 制御依存のため時間がかかるという分岐命令の問題
- メモリからデータを取り出すのに時間がかかるというメモリ遅延の問題

従来のプログラミング言語の側では並列処理を明示できないので，並列処理を引き出す仕事はコンパイラの側でやらなければならない．インテル社の次世代64ビットアーキテクチャIA-64は，条件付実行制御と投機的実行制御を利用して並列処理を増やす戦略を採用しており，その中でコンパイラが非常に重要な役割を果たしている（次ページの囲み記事を参照）．

本稿は，コード例を2つ紹介しながら，条件付実行制御と投機的実行制御について説明する．どちらも実際のIA-64命令を使用した典型的な汎用整数コード（general-purpose integer code）であり，CADやデータベース・アプリケーションなどに利用されている．条件付実行制御と投機的実行制御を使った場合と使わなかった場合を比べることによって，この2つの機能を使った方が命令の実行に必要なサイクル数を減少させ性能を向上させることができることを示す．

条件付実行制御（predication）☆2

IA-64アーキテクチャで採用された完全条件付実行制御（full predication）モデルでは，コンパイラがすべての命令にプレディケート（predicate）を付加できる．プレディケートというのは，条件付きで命令の実行をプログラムに許可するタグ（フラグ）のことだ．条件付きというのは，プレディケート値に基づくということであり，これはさらに条件文の実行結果に基づいて決まることを意味する．プレディケート値がtrueの場合，命令は正常に実行される．プレディケート値がfalseの場合，命令がすでに発行されていても，その実行結果をレジスタやメモリへ書き込む操作は行われない．実験結果によると，分岐を除去し分岐の予測誤り[1]による余分な処理時間を減少させるうえで条件付実行制御が役立つことが分かった．それでは，予測困難な分岐を含む簡単なコード例を取り上げて，条件付実行制御により分岐を除去できることを示そう．

従来のCコードで記述したif-then-else文の例を図-1

☆1 instruction-level parallelism：著者Caroleから"I am using the word parallelism in the sense of parallel processing."という説明があった．この内容に基づいて「命令レベルの並列処理」と訳した．タスクやプロセスのレベルではなく，命令レベルでの並列処理である．「パラレリズム」と訳すと「～主義」というニュアンスが強くなりすぎるので避けた．ここで-ismという語尾は単に「動作」「状態」を示す．

☆2 predication：著者Caroleから"Predication would be best translated as conditional execution."という返事があった．この提案に基づいて「条件付実行制御」と訳した．言語の世界でpredicateは「主語に属性を付与する」という意味で使われる．ここでは，1ビットのレジスタ（predicate register）を使用してプレディケートと呼ばれるフラグを命令に付与する操作を表す．命令を先行処理した後，フラグ値をチェックし，trueであれば命令の実行結果をメモリやレジスタに書き込む．falseであれば廃棄する．スピード低下の要因である分岐処理を事実上なくす技術である．

IA-64の開発目的と特徴

IA-64はインテル社が初めて開発した64ビットアーキテクチャである．IA-64アーキテクチャに関する情報はこれまで断続的に公表されてきたが，ここで正式な情報をまとめておく．

IA-64アーキテクチャの開発目的は次のとおり．

- IA-32ソフトウェアとの完全なバイナリ互換を保証する．
- 将来の拡張にも余裕をもって対応できる広範なインプリメンテーションに対してスケーラビリティを保証する．
- 完全な64ビットコンピューティングを提供する．

IA-64はEPIC（Explicitly Parallel Instruction Computing）[1]という設計思想に基づいて開発された．EPICには次の3つの特徴がある．

- コンパイラが並列処理を洗い出す：EPIC準拠のアーキテクチャを採用すれば，マシンコードから並列処理の可能性を洗い出すことができる．ハードウェアがスケジューリングを行うウィンドウ（再順序付けバッファ）より，コンパイラがスケジューリングを行うスコープのほうが当然大きい．したがって，逐次性の強いマシンコードから命令レベルの並列処理を引き出す作業をハードウェアにさせるより，コンパイラにさせるほうが効果的である．
- コンパイラ支援機能を提供する：IA-64は並列処理を引き出すためのさまざまな機能をコンパイラに提供する．その中の2つの制御機能がこの論文の目玉である条件付実行制御と投機的実行制御である．
- ハードウェア資源を増やす：IA-64アーキテクチャでは，現在の商用プロセッサよりレジスタ数を増やしてある．64ビットの整数レジスタが128個，64ビットの浮動小数点レジスタが128個，1ビットのプレディケート（フラグ）用レジスタが64個もある．1台のプロセッサで複数の機能ユニットを効率的に使用するにはレジスタ数を増やす必要があるからだ．

コンパイラがコードを並列化し複数のハードウェア資源（機能ユニット）で処理する様子を図-A（1）に示す．

IA-64の命令バンドル（instruction bundle）を図-A（2）に示す．命令バンドルは，40ビットの命令が3個と8ビットのテンプレートで構成される．テンプレートは，命令依存を明示する手段として使われる．コンパイラはテンプレートをチェックして複数の命令バンドルにまたがる独立命令をグループにまとめることができる．

参考文献
1) Gwennap, L.: Intel, HP Make EPIC Disclosure, Microprocessor Report, pp.1, 6-9 (Oct. 27, 1997).

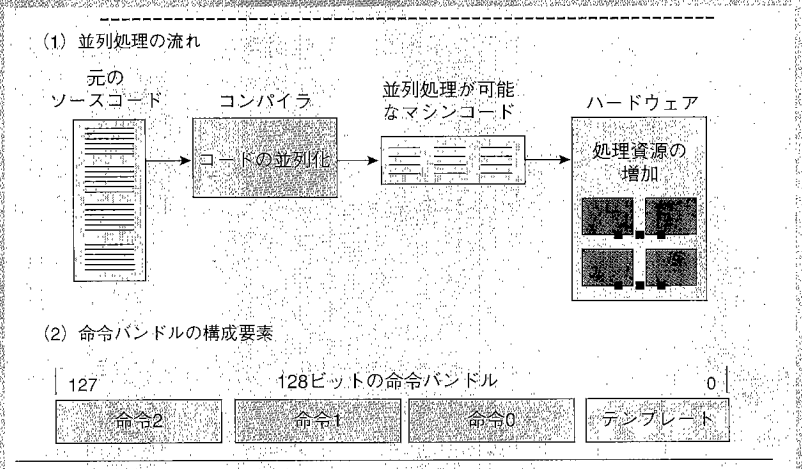


図-A コンパイラによる並列処理の流れ（1）と128ビットの命令バンドルの構成要素（2）

（a）に示す．従来のアーキテクチャでは，プロセッサがメモリからデータをロードし，a[i].ptrの値とゼロを比較し，その比較結果を条件分岐命令で使用する．条件分岐であるため，従来のコンパイラはこのコードを4つの基本ブロックに分けて処理する（図-1（b）を参照）．プロセッサは4つのブロックで構成される命令を逐次処理しなければならない．分岐命令があると並列処理ができないからだ．しかし条件付実行制御を利用すると，最初の基本ブロックにある予測困難な分岐を除去することができる．

IA-64アーキテクチャのcompare命令は次の2つのプレディケートを生成する（図-1（c）を参照）．

- 比較結果がtrueの場合に1に設定される真のプレディケート
- 比較結果がfalseの場合に1に設定される偽のプレディケート

比較結果がtrueであれば，thenパスを実行する．言い換えると，プレディケートp1はこのthenパスの実行を指示する．比較結果がfalseであれば，elseパスを実行する．言い換えると，プレディケートp2はこのelseパスの実行を指示する．

この処理の流れから分岐は発生しないので，thenパスとelseパスを同時に実行することができる．p1とp2は相互に排他的な関係にある（ある時点でtrueになるのはp1かp2のどちらか一方であり，同時に両方がtrueになることはないという意味）．したがって，store命令の実行結果を格納するアドレスが同じであっても，2つのstore命令を同時に実行することができる．store命令の実行結果をメモリに書き込むのは，プレディケート値によってどちらか一方に制限される．他方の実行結果は廃棄される．こうすれば命令レベルの並列処理が可能になる．条件付実行制御を利用すれば，分岐

☆3 control speculation：著者Caroleから"Yes, speculation is really a bet on the future. If you are correct, you win （in this case performance）, if you are wrong, you do not gain performance."という返事があった．この内容に基づいて「投機的実行制御」と訳した．ビジネスの世界でspeculateは「投機的な思惑から多少の危険を覚悟で実行する」という意味で使われる．成功すれば得する（性能が向上する）が，失敗すれば損する一種の賭けである．ここでは，load命令を先行処理することでメモリ遅延を減少させる技術を指している．predicationもcontrol speculationも，並列処理が可能な命令を洗い出し先行処理することで，全体として処理の高速化を図る技術である．

☆4 fixup code: 著者Caroleから"fix up code is not the OS routine. It is compiler generated code which re-does the instructions needed to be re-executed if the speculation has failed."という補足説明があった．適訳が思い浮かばず，取りあえず「フィックスアップコード」と訳した．OSの例外処理ルーチンではない．投機制御に失敗したときに命令を再実行するコードであり，コンパイラが生成する．

とそれに関連した予測誤りによるムダをなくすことができる．IA-64アーキテクチャを採用すれば，if-then-else形式の文を1つの基本ブロックとしたスケジューリングが可能になる（図-1（c）を参照）．

投機的実行制御（control speculation）☆3

コンパイラにload命令を先行処理させると，効率的にメモリ遅延の影響を低減し命令レベルの並列処理を増加させることができる[2]．IA-64アーキテクチャは，load命令の実行時期を分岐処理の前に移動するホイスティング（hoisting）という手段を提供する．プログラムがデータを必要とする前にロードしておくことでメモリ遅延を解消する手法である．この手法を使うと，プロセッサが使える実行ユニットを増やすことで，並列処理が可能な命令を増やすことができる．

投機的実行制御が適用できるコード例を図-2（a）に示す．ここで，配列[a]が大きくてキャッシュに入らないものと仮定する．その場合，コンパイラができるだけ早い時点でメモリから配列要素をロードしておかないと，メモリ遅延が発生してしまう．この例では，a[t1-t2]をロードする命令を，分岐が起きない地点まで繰り上げている．ただし，アドレスの有効性にかかわらず，アーキテクチャがこのような投機的ロード処理をサポートしていることが前提になる．たとえば，t1がt2より大きくなければ，a[t1-t2]のアドレスは無効になる．無効なアドレスからロードしようとすれば，例外が発生する．

IA-64アーキテクチャで投機的load命令が導入された．この命令ld.sは，メモリフェッチを実行し，例外も検出する．しかし例外は報告しない（OSの例外処理ルーチンを呼び出さないという意味）．check.sという名前の命令がload命令のホームブロックにあり，この命令check.sが例外を報告する．命令ld.sは，例外を検出すると，使用したレジスタにトークンビットを設定する．このレジスタに対して発行された命令check.sは，トークンビットが設定されていると，フィックスアップコード（fixup code）☆4へ制御を渡す．

投機的ロード機構を使うと，プログラムがデータアドレスの有効性を判定する前にload命令を実行することができる．このコード例に沿って説明しよう．まず，t1とt2を比較する命令より先に，a[t1-t2]をロードする命令を実行する．ここで，t1がt2に等しいかそれ以下であると，プログラムはレジスタr8のトークンビットを設定し，このパスをスキップする．ここで検出された例外は実行されない．t1がt2より大きい場合，load命令は正常に実行される．プロセッサは，load命令のホームブロックを実行する．命令check.s r8は実行されるが，レジスタr8にトークンビットが設定されていなければなにもしない．a[t1-t2]のアドレスは有効であるが，load命令の実行でページフォールトが発生した

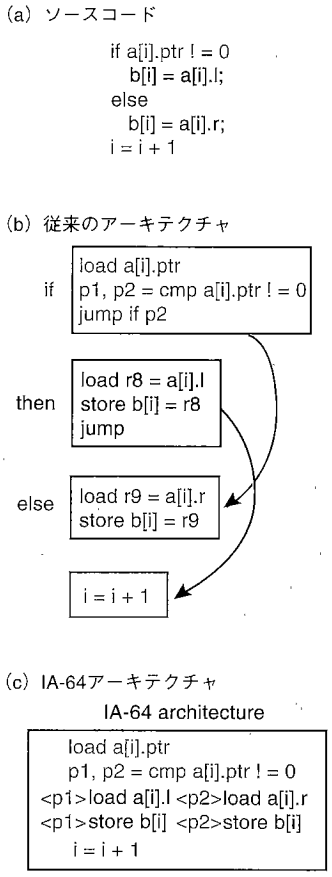


図-1 従来のアーキテクチャとIA-64アーキテクチャの比較
if-then-else文の例（a）．4つの基本ブロックで処理する従来のアーキテクチャ（b）．1つの基本ブロックで処理するIA-64アーキテクチャ（c）．

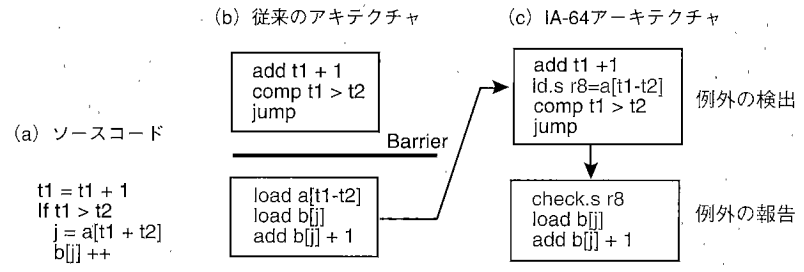


図-2 投機制御の適用例
ソースコード例（a）．従来のアーキテクチャ（b）．IA-64アーキテクチャ（c）．矢印はロード命令を先行処理することを示す．

図-3　xlxgetvalueのソースコード（a）と流れ図（b）

（a）ソースコード

```
for (fp = xlenv; fp; fp = cdr(fp))
  for (ep = car(fp); ep; ep=cdr(ep))
    if (sym == car(car(ep)))
      return (cdr(car(ep)));
```

（b）流れ図

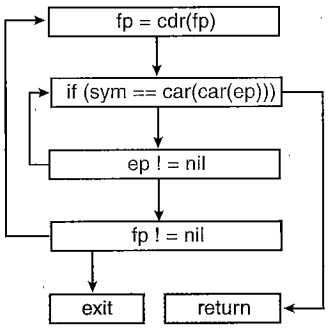
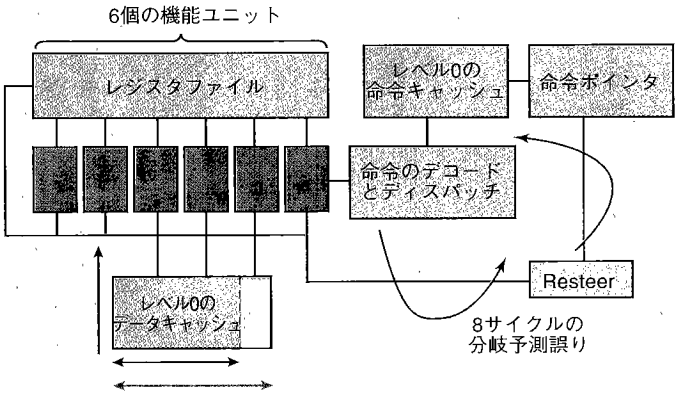


図-4　2種類のコードに対応した架空のマシン構成



場合，レジスタr8にトークンビットが設定される．命令check.s r8は，トークンビットが設定されていることを検出すると，プログラムの制御をフィックスアップコードへ渡す．このコードは，ページを読み込み，load命令を再実行する．こうして，check.s命令の後から処理が再開される．

IA-64アーキテクチャの適用例

条件付実行制御と投機的実行制御を実際に使用したコード例をそれぞれ1つずつ紹介する．どちらも典型的な汎用整数コードである．1つはxlxgetvalueと呼ばれるコードである．Lispインタプリタ用のliであるSPECintベンチマークから採用した．投機的実行制御を使用することで，メモリ遅延を解消し，データをロードしてからポインタの有効性を検証する．

もう1つはtreeinsという名前のコードである．これは木構造のデータに新しい要素を追加するコードである．条件付実行制御の使い方，さらにネスト構造になっているif-then-else文のすべてのパスを並列処理する手順を示す．どちらのコード例も，従来のアーキテクチャでは並列処理が困難なためほとんどが逐次処理されていた．

xlxgetvalue

このコードは，リンクされたリストを追いかけるポインタ追跡問題としてよく知られている．もともと逐次性の高いコードであるが，投機的実行制御を利用すると，そこから並列処理を引き出すことができる．

Cのコード例とそれに対応する流れ図を図-3に示す．まず，緑色で表示した内側のループに注目してもらいたい．導入変数epを2回使用（データへのポインタとして使用するという意味）し，値symと比較する．プロセッサがsymを検出すると，制御はループを抜ける．x=car(ep)とy=car(x)の2つのload命令がループのクリティカルパスにある．このクリティカルパスを短縮するには，load命令をできるだけ早く実行する必要がある．投機的実行制御を利用すると，コンパイラはepを検出した時点で，ポインタの有効性が不明であっても，load命令をスケジュールすることができる．

xlxgetvalueで使用するマシン構成を図-4に示す．なお，このマシン構成はあくまで架空のものであって実際のものではないことに注意．このモデルでは，機能ユニットが6個あり，どの機能ユニットでも，loadとstoreを除くすべての命令を実行できるものとする．レベル0のデータキャッシュにアクセスできる機能ユニットが2個あり，load命令とstore命令を発行できるのはこの2つだけに限られる．レベル0のキャッシュ遅延は1クロックだけとする．

なお，このコードはこのマシン専用に最適化してあるが，そのほかのマシン構成でも十分活用できる．IA-64アーキテクチャを採用すれば，機能ユニットの個数や機能ユニットごとのメモリ遅延に違いがあっても，互換性を維持することができるからだ．従来のVLIWアーキテクチャとの相違をまとめておく．

- IA-64はメモリ遅延を意識させない（ハードウェアがコンパイラに依存しないで遅延を考慮したスケジューリングをする）．
- IA-64はハードウェアの完全なインターロックを保証する（ハードウェアがスコアボード（scoreboard）を使って遅延を追跡する）．

これは従来のアーキテクチャについても言えることだが，EPICアーキテクチャでもプログラムをマシン専用にコンパイルすれば，それなりにかなり性能を発揮することができる．

ループの1回目の実行　コードxlxgetvalueを1回目に実行した状態を図-5に示す．この部分の前にあるC文は同じ色のマシンレベルの命令に対応する．各欄は6個の機能ユニットにそれぞれ対応している．各行は命令サイクルに対応し，同じサイクルで同時に実行で

for (ep = car(fp); ep; ep = cdr(ep))
if (sym == car(car(ep)))

Cycle	Unit 1	Unit 2	Unit 3	Unit4	Unit 5	Unit 6
1	Ld ep1					
2	Cond1 = ep1 == nil	Ld.s car(ep1)	Br nxt_fp if cond1			
3	Check.s	Ld x = car (car(ep1))				
4	Cond2 = sym == x	Br return if cond2	Br nxt_ep			

図-5 xlxgetvalueの1回目の実行

Cycle	Unit 1	Unit 2	Unit 3	Unit4	Unit 5	Unit 6
0	Ld ep1					
1	Ld.s car(ep1)	Cond1 = cmp ep == nil	Ld.s ep2 = cdr(ep1)	Br nxt_fp if cond1		
2	Check.s	Ld car car(ep1)	Ld.s car(ep2)	Cond3 = cmp ep2==nil		
3	Cond2 = cmp == sym	Ld.s car car(ep2)	Br return if cond2	Br nxt_fp if cond3		
4	Check.s	Ld nxt ep1 = cdr(ep2)	Cond4 = cmp == sym	Br return if cond4	Br nxt_ep	

図-6 xlxgetvalueの2回目の実行

きる命令を示す．たとえば，命令ld ep1はサイクル1で発行される．キャッシュの遅延が1クロックとすると，プロセッサはロード命令の実行結果をサイクル2で利用できることになる．そのため，cond1を生成する命令cmpがサイクル2で発行されているのである．

図-5のサイクル2で投機的load命令が発行されていることに注目してほしい．この命令ld.sは，プロセッサがep1の有効性を検証する前にcar(ep1)にアクセスする．ep1の有効性が確認できるのは，サイクル3で命令ld.s car(ep1)に対応するチェックが実行されるまで待たなければならない．プロセッサがサイクル2でcar(ep1)をロードできれば，car(car(ep1))をサイクル3でロードできる．そのため，プロセッサはサイクル4でこのロード結果をsymと比較することができる．ここで求めていた値が得られれば，ループは終了する．求めていた値が得られなかった場合は，制御が次のループnxt_epへ渡される．これで1回目の実行が4サイクルで処理できることが理解できたと思う．

ここで，サイクル2でcar(ep1)をロードする投機的load命令を実行したときにページフォールトが発生した場合について考えてみよう．ページフォールトが報告されるのは，サイクル3で命令check.sが実行されるまで待たなければならない．チェック結果でページフォールトが確認されると，フィックスアップコードが実行され，ページが読み込まれる．それからcar(ep1)をロードする投機的load命令が再実行される．特にこのような問題が発生しなければ，通常はチェック後の最初の命令，すなわちcar(car(ep1))をロードする投機的load命令から処理が再開される．そして，ep1がnilポインタであると，このループはサイクル2で処理を中断し，次の外側のループ（図-5のnxt_fp）へ制御が移る．このとき投機的load命令で例外が発生しても，プログラムはロード結果を必要としないので，例外はプログラムに報告されない．

なお，コンパイラがサイクル4で分岐命令を2つスケジュールしていることに注目してほしい．どちらか一方，たとえば，return if cond2が実行された場合は，もう一方の分岐命令は実行されない．return if cond2が実行されなかった場合は，もう一方の分岐命令が（この例では無条件で）実行される．

このループの1回目の実行でマシンの資源を全部使っているわけではない．未使用の機能ユニットが半分以上もある．そこで，このループを展開して並列処理を引き出し，マシンの実行ユニットを活用する方法を検討してみたい．

ループを展開する xlxgetvalueコードの内側のループを2回実行した状態を図-6に示す．1回目の実行に対応する命令は，図-5の場合と同じである．これらの命令は黒で表示してある．2回目の実行に対応するマシン命令は上右に示したCコードの色と同じにしてある．

このループの2回目の実行にも投機的実行制御が適用できる．プロセッサはサイクル1で，導入変数ep2の

図-7　従来のスケジューリング（投機的実行制御を使用しない場合）

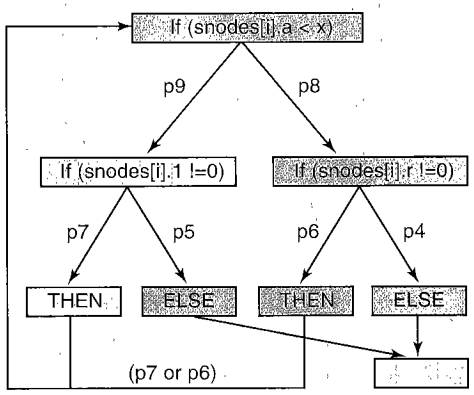
Cycle	Unit 1	Unit 2	Unit 3	Unit4	Unit 5	Unit 6
1		Cond1 = cmp ep == nil		Br nxt_fp if cond1		
2	Ld car(ep1)	Ld ep2 = cdr(ep1)				
3	Ld car car(ep1)		Cond3 = cmp ep2==nil			
4	Cond2 = cmp == sym	Ld car(ep2)	Br return if cond2	Br nxt_fp if cond3		
5		Ld car car(ep2)				
6	Ld nxt ep1 = cdr(ep2)	Cond4 = cmp == sym	Br return if cond4	Br nxt_ep		

図-8　treeinsのソースコード（a）と流れ図（b）

（a）ソースコード

```
L10: /* compare */
  if (s.nodes[i].a < x)
    if (s.nodes[i].l !=0) {
      i = s.nodes[i]. l;
      goto L10;}
    else {
      s.nodes[i].l = j;
      goto L20;}
  else{
    if (s.nodes[i].r != 0) {
      i = s.nodes[i].r;
      goto L10;}
    else{
      s.nodes[i].r = j;
      goto L20;}
  }
L20: /* insert */
      s.nodes[j].a = x;
      s.nodes[j].l = 0;
      s.nodes[j].r = 0;
      s.nxt_avail = j+1;
```

（b）流れ図



次の値を投機的にロードする．「投機的」というのは，1回目の実行で導入変数の有効性を確認する前に行うという意味である．この2つのload命令は，ep2だけではなく，サイクル2のcar(ep2)とサイクル3のcar(car(ep2))の投機的load命令の影響も受ける．ep2がnilポインタであれば，プログラムはサイクル3でループを抜ける（cond3がtrueであればnxt_fpへ移る）．したがって，その地点になるまで，ep2を使用する命令はすべて不確定なのである．

どの命令もすべてのレジスタに関連したトークンビット値を伝える．言い換えると，ある命令のオペランドのトークンビットに1が設定されていれば，使用したレジスタのトークンビットにも1が設定される．このコード例に即していうと，プロセッサはある投機的load命令で例外を検出すると，最後の投機的load命令で使用したレジスタのトークンビットを1に設定する．これは，関連したload命令が3つあった場合，プログラムで使用できる命令check.sは1つしかないという意味だ．つまり，一連のload命令で最後にロードしたレジスタのトークンビットをチェックして例外を検出したときに，check.sへ例外が報告されるのである．

フィックスアップコードは，関連した一連のload命令でロードされたすべてのレジスタをチェックして次の処理を行う．

- ページフォールトがあれば，不足したページを読み込む．
- 最初のパスで実行されなかったload命令があればすべて再実行する．

このマシン構成では2ポート形式のデータキャッシュを採用しているので，プロセッサはサイクル1とサイクル2で，2つのload命令を並列して実行する．2回目の実行は，図-5で示した1回目の実行と非常によく似ている．コンパイラが両方の実行を4クロックで並列にスケジュールする状態を図-6に示す．導入変数ep1を使用するload命令Load next ep1は，ループの最後サイクル4で行われる．サイクル0に示したep1のロードは，ループに入ったときに1回だけ行われるので，内側のループには属さない．

投機的実行制御を利用すればループを2回実行するのに4クロックですんだが，この機能を利用しない場合は6クロックかかる（図-7を参照）．導入変数ep1を使用するload命令を実行できるのは，ep1とnilポインタとの比較が行われた後になる（図-7で赤丸で囲んだ2つのload命令）．同じように，導入変数ep2を使用す

```
if (snodes[i].a < x)          else{
    if (snodes[i].l != 0) {       if (s.nodes[i].r != 0) {
      i = snodes[i].l;              i = s.nodes[i].r;
      goto L10;}                    goto L10;}
```

Cycle	Unit 1	Unit 2	Unit 3	Unit4	Unit 5	Unit 6
1	Shladd					
2	Shladd					
3	Add	Add				
4	Ld a	Ld l	Ld r			
5	Cmp p9, p8= a<x					
6	<p9>Cmp p7, p5= l|=0	<p8> Cmp p6, p4 r|=0	<p9> Mov i=l	<p8> Mov i=r	<p6> Br nxt_loop	<p7> Br nxt_loop

図-9 treeinsのthenパス部分の処理

るload命令を実行できるのは，プロセッサがep2とnilポインタとの比較を行った後になる．ep2がサイクル3で利用可能になっても，このload命令が実行されるのはサイクル4へ持ち越される．

投機的実行制御を利用すると，xlxgetvalueのループを2クロック間隔で実行できる．一方，投機的実行制御を利用しないと，このマシン構成の場合，ループの実行にそれぞれ3クロックかかる．わずか1サイクルの差にすぎないが，数百万回も実行するループに対してliベンチマークを実施すると，全体では非常に大きな差になる．

treeins

予測困難な分岐を除去し並列処理を引き出すために条件付実行制御を使用するコード例としてtreeinsを紹介する．Cコードと流れ図を図-8に示す．このコードの目的は，木構造のデータに値xを挿入することである．木構造の各ノードでプログラムは，既存のデータ値aとxを比較して，その木構造の右側（s.node.r）と左側（s.node.l）のどちらにxが属するのかを判定する．プログラムがノードをテストして新しい要素xを挿入するのは，木構造の葉に相当する場所である．

このモデルは，以前に紹介したモデルとほぼ同じマシン構成を前提にしている．ただし，両者にはデータキャッシュのポート数に違いがある．このモデルにはポートが3つあるので，各サイクルで最大3個の命令（loadまたはstore）が実行できる．

thenパス　このコードでは，コンパイラが最初にthenパスをコンパイルする構成になっている．図-8ではこれらのパスを強調表示してある．この図でパスの色は，以降の図に示すCコードとマシンコードの色に対応している．

各パスのサイクル数を増やさないで，2つのthenパスを並列処理した状態を図-9に示す．2つのthenパスのCコードも示してある．

プログラムは最初に，s.nodes[i].aのアドレスを計算しなければならない．s.nodesのデータ構造は，3つの整数要素（各要素は4バイト幅）を持つ．したがって，導入変数iに12を掛ける．プログラムはこの計算をするために2種類のshift-left-add命令を使用する．1つは左へ2桁シフトするshift-left-add by two命令（4倍）で，もう1つは左へ3桁シフトするshift-left-add by three命令（8倍）である．

データ構造の要素aのアドレスを得るには変位（displacement）を加算しなければならない．s.nodes[i].aの前にshladd命令が2つ，add命令が1つあるのはそのためである．このマシンはメモリポートが3つある構成なので，s.nodes[i]の3つの要素をロードする処理はサイクル4で並列に実行することができる．プロセッサはサイクル5でプレディケートを計算する．aがxより小さければ，p9がtrueになる．aがxに等しいかそれ以上であれば，p8がtrueになる．サイクル6でp9とp8は，rとlの値を比較するcompare命令で使用される．s.nodes[i].lの値を比較するというのは，2つの排他プレディケートp7とp5を計算することである．s.nodes[i].rの値を比較するというのは，2つの排他プレディケートp6とp4を計算することである．

compare命令で使用するプレディケート（この場合はp9またはp8）がfalseであれば，その影響を受けるプレディケート（p5とp7，またはp4とp6）もfalseになる．

あるサイクルで計算したプレディケートは，同じサイクルの分岐命令で使用することができる．p6とp7で処理した2つの分岐命令をサイクル6で実行できるのはそのためである．プレディケートが付けられたmove命令がサイクル6[☆5]に2つある．iを保持しているレジスタに左側（l）と右側（r）のどちらか一方がコピーされる．p9とp7は相互に排他的である．

elseパス　コンパイラは2つのthenパスを6クロックで処理する．次に，すでに処理したパスと並行してelseパスを処理する．treeinsコードの全体を図-10に示す．elseパスのCコードと対応するマシン命令は同じ色で表示してある．

プロセッサは，s.nodes[i]の場合と同じやり方で，s.nodes[j]のアドレスをサイクル1，2，3で計算する．

☆5 原文はcycle 5であるが，著者Caroleに確認の上「サイクル6」に訂正してある．

図-10　treeinsの全体の処理

☆6 著者Caroleに確認の上「s.nodes[i].l=j;」に訂正してある.

```
else{                    L20: /* insert */          else{
    s.nodes[i].l = j;☆6          s.nodes[j].a = x;        s.nodes[i].r = j;
    goto L20;}                   s.nodes[j].l = 0;        goto L20;}
                                 s.nodes[j].r = 0;
                                 s.nxt_avail = j+1;
```

Cycle	Unit 1	Unit 2	Unit 3	Unit4	Unit 5	Unit 6
1	Shladd		Shladd			
2	Shladd		Shladd			
3	Add	Add	Add	Add	Add	Add
4	Ld a	Ld l	Ld r			
5	Cmp p9, p8= a<x					
6	<p9>Cmp p7, p5= l!=0	<p8>Cmp p6, p4 r!=0	<p9> Mov i=l	<p8> Mov i=r	<p7> Br nxt_loop	<p6> Br nxt_loop
7	<p5>Store [i].l	<p4>Store [i].r	Store [i].a			
8	Store [j].l	Store [j].r	Store nxt_avail			

同じやり方とは，shift-left-add命令2つとadd命令1つを使うという意味だ.

else文に対応する2つのstore命令がサイクル7にある．この2つのstore命令は相互に排他的なp4とp5の制約を受けるので，実行結果をメモリに書き込めるのはどちらか一方だけに限られる.

コンパイラはサイクル7と8で，2つのelse文に共通のパスにある4つのstore命令をスケジュールする．これらのstore命令はすべて並列に処理できるが，キャッシュポートが3つしかないので，実際に並列処理できるのは3個に制限される．これらのstore命令は2つのelseパスで実行されるので，条件付実行制御の対象にはならない．プログラムがthenパスを処理しているときは，サイクル6にある分岐命令のどれか1つが実行されるので，サイクル7と8にある命令は実行されない.

全体の処理　全体の処理を図-10に示す．この図を見ると，thenパスが6クロック，elseパスが8クロックで処理できることが分かる．ループがelseパスを抜ける前にthenパスが何回も実行される確率が高い．したがって，平均すると1回の実行に7クロックまでかからない勘定になる．なお，コンパイラは4つのパス（thenパスが2つとelseパスが2つ）をすべて並列にスケジュールする．その結果，並列処理をかなり増やすことができた.

ここで紹介したコード例で条件付実行制御を利用しなければ，2つの分岐命令にかかわる次の問題が解決できない.

- 分岐予測が困難（2つの分岐命令で予測誤りの確率は実際上25%を超える）.
- 予測誤りによる処理に余計な時間がかかる（このマシン構成でいうと，1つの予測誤りに対して8クロック余計にかかる）.

上記をまとめると，1つの分岐につき最低2クロック，1回の実行につき4クロックも余計にかかることになる．これは7クロックシーケンスにとって非常に重いコストである．しかし，条件付実行制御を採用すれば，この問題を完全に解決できる.

条件付実行制御と投機的実行制御は，汎用整数コードに含まれる並列処理を増やし命令のサイクル数を減らす上でかなりの効果が期待できる．この2つの制御機能はすでに知られているものだが，商用アーキテクチャでこの両方を採用したのはIA-64が初めてである．今回は，コンピュータエンジニアリングの世界でよく知られているさまざまなアイディアを統合し実現したIA-64の一端を紹介させていただいた．今後の動向についてはインテル社が今後発表する最新情報を参照していただきたい.

※最新情報についてはインテル社のホームページを参照.
日本のサイト
http://www.intel.co.jp/jp/developer/design/processor/future/ia64.htm
米国のサイト
http://developer.intel.com/solutions/archive/issue6/stories/IA64.htm
※この論文の翻訳では，下訳段階から日立教育部の田口昭仁氏にいろいろ教えていただいた．また，著者のCaroleにもEメールで何回もお世話になった．お礼申し上げる.

参考文献
1) Mahlke, S., et al.: Characterizing the Impact of Predicated Execution on Branch Prediction, Proc. Micro 27, IEEE CS Press, Los Alamitos, Calif., pp.217-227 (1994).
2) Hwu, W., et al.: The Superblock: An Effective Technique for VLIW and Superscalar Compilation, J. Supercomputing, pp.229-249 (Jan. 1993).

参考資料
Colwell, R., et al.: A VLIW Architecture for a Trace Scheduling Compiler, Proc. 2nd Int'l Conf. Architectural Support for Programming Languages and Operating Systems, IEEE CS Press, Los Alamitos, Calif., pp.180-192 (1987).
Kathail, V., Schlansker, M. and Rau, B.R.: HPL PlayDoh Architecture Specification: Version 1.0, Hewlett Packard Computer System Lab., Tech. Report HPL-93-80 (Feb. 1994).
Rau, B.R., et al.: The Cydra 5 Departmental Supercomputer－Design Philosophies, Decisions, and Trade-Offs, Computer, pp.12-35 (Jan. 1989).
Schuette, M.A. and Shen, J.P.: Instruction-Level Experimental Evaluation of the Multiflow TRACE 14/300 VLIW Computer, J. Supercomputing, Kluwer Academic Publishers, Dordrecht, The Netherlands, pp.249-271 (1993).

（平成10年10月1日受付）

訂　正

本誌39巻12号（平成10年12月号）掲載の翻訳記事「具体例によるIA-64アーキテクチャの紹介」に対して，東京工業大学佐々政孝先生から翻訳上の問題点についてご指摘をいただきました．この誌面をお借りして，下記のとおり訂正させていただきます．ありがとうございました．

（訳者：安藤　進）

- p.1228 左側下から7行目：
  （誤）Lispインタプリタ用のliであるSPECintベンチマークから採用した．
  （正）SPECintベンチマークであるli（Lispインタプリタ）から採用した．

- p.1228右側12行目（ほか8カ所）：induction variableの訳語
  （誤）導入変数　（正）帰納変数

- p.1231右側下から8行目：（注）数字の転記ミス．
  （誤）p9とp7は相互に排他的である．
  （正）p9とp8は相互に排他的である．