計算機アーキテクチャ 102－2
（1993．10．21）

スーパースカラ用コンパイラの評価

細井 聡　　新井 正樹　　木村 康則

富士通研究所

我々は、out-of-order 実行、先行実行を行なうスーパースカラーマシン上で、SPECint プログラムを実行する場合、コンパイラ によりさらに性能を上げるためには、どのような最適化が有効であるか検討した。Global Scheduling の 1 つである Percolation Scheduling は、命令レベルの並列度を上げるための最適化であるが、speculative なコード移動だけでは、さらに並列度を上げることは難しい。Loop Unrolling と 前方 Short Jump の削除は、分岐のオーバーヘッドを軽減する最適化である。Loop Unrolling はたいていのプログラムに対して有効であり、espresso や xlisp に対しては、マシンが BTB を持つ場合にも有効である。前方 Short Jump の削除 は、特に eqntott に対して有効である。

Evaluation of the compiler for superscalar processors

Akira Hosoi, Masaki Arai, Yasunori Kimura

FUJITSU Laboratories LTD.

1015. Kamikodanaka Nakahara-ku, Kawasaki 211, Japan

email: hosoi@flab.fujitsu.co.jp

What kind of compiler optimizations is effective for executing *SPECint* more efficiently on superscalar processors with out-of-order and speculative execution ? We tested Percolation Scheduling, Loop Unrolling, and *Deleting Forward Short Jump*: Percolation Scheduling, which is one of Global Scheduling, is to exploit more Instruction Level Parallelism(ILP).But without speculative code motion, Percolation cannot increase ILP on such a powerful machine. Loop Unrolling and *Deleting Forward Short Jump* are to decrease the branch penalty. Loop Unrolling is effective for almost programs, and also effective for *espresso* or *xlisp* even if processor has BTB. 'Deleteing Forward Short Jump' is very effective for *eqntott*.

1 はじめに

最近のプロセッサは、スーパースカラ方式をとるものが多く、Pentium のように BTB を持つものも現れた。また、後継 Pentium は out-of-order 実行や register renaming 機構を備えると言われている。

我々は、out-of-order 実行、分岐命令を越えての先行実行を行なうスーパースカラーマシン上で、SPECint のような非数値計算プログラムを実行する場合、コンパイラ によりさらにどのくらい性能を上げることができるか検討するため、いくつかのシミュレーションを行なった。スーパースカラの性能を上げるためには、まず、命令レベルの並列度を上げなければならない。また、SPECint は分岐の多いプログラムなので、分岐のオーバーヘッドを減らすことが重要であると考えられる。そこで、命令レベルの並列度を上げる最適化と、分岐のオーバーヘッドを減らす最適化について検討することにした。

まず、2節でターゲット アーキテクチャ、3節でベンチマークプログラムについて簡単に述べる。次に、4節で、ベースとするコンパイラについて説明する。 5節では、命令レベルの並列度を上げる最適化として、Percolation Scheduling の評価を行なう。また、6節では、分岐のオーバーヘッドを減らす最適化として、Loop Unrolling と前方 Short Jump の削除についての評価を行なう。最後に、まとめと今後の課題を述べる。

2 ターゲット アーキテクチャ

SPARC をベースに、表 1のようなアーキテクチャを想定する。そしてシミュレーションでは、HRR(Hardware Register Renaming) や BTB(Branch Target Buffer) の有無により、BASE (HRR, BTB 共になし)、HRR (HRR あり)、BTB (BTB あり)、HRR+BTB (HRR,BTB 共にあり) の 4 つの場合を考える。それぞれの場合に、コンパイラの最適化によりどのくらい speedup が得られたかを測定する。

3 ベンチマークプログラム

ベンチマークプログラムとして、SPECint92 の 6 つのプログラムを用いる。いずれも非数値計算プログラムで、分岐の多いプログラムである。表 2に各 ベンチマークプログラムとその実行条件を示す。

4 コンパイラ

4.1 ベース コンパイラ

ベースとするコンパイラ は、GNU C コンパイラ Ver2.1 である。Loop Unrolling と inline 展開以外の最適化 (分岐最適化、loop 最適化、共通項削除、局所 scheduling) は全て行なう。以下、この コンパイラ を GCC と記す。

今回の測定に当たっては、局所 scheduling は、GNU C コンパイラ の局所 scheduling をそのまま用い、特に、スーパースカラ 用に強化はしなかった。新たに強化しても、あまり効果はないと考えたからである。その理由は以下の通りである。

Int 系のプログラムにおいては、基本ブロック中の命令数は、想定アーキテクチャ の命令フェッチ数と同程度であり、しかもそれらが out-of-order に実行される。すなわち、局所的な scheduling は、動的な scheduling だけでも充分なされている可能性が高い。したがって、局所 scheduling の効果は、ハードウェアによる動的な scheduling に相殺されてしまう割合が大きいと考えられる。HRR や BTB を使用するなら、この傾向はなお一層強まると考えられる。

4.2 最適化の評価

各最適化が有効であるかどうかは、最適化を適用して生成したコードと適用しないで生成したコードの実行速度の比較を行ない、speedup が得られたかどうかで判断する。実行速度の比較は、Paratool (我々の研究室で開発した、実行 trace ベースの simulator) 上で実行し、総実行サイクル数を比較することにより行なう。

表 1: ターゲットアーキテクチャ

実行方式	in-order issue, out-of-order execution, out-of-order completion
	speculative execution（BTB がない場合は、全ての分岐命令は not taken であると予想して先行実行する）
命令フェッチ/命令発行	4 命令フェッチ/4 命令発行
Functional Units	load/store unit 1, ALU 2, shifter 1,branch unit 1
Reservation Station(RS)	集中型、エントリ数 32
Cache	I-cache, D-cache それぞれ 16KB (32bytes block, 128entry,4way) Miss Penalty 20 cycles
Store Buffer	8 エントリ
Branch Target Buffer (BTB)	16KB (32bytes block,128entry,4way)
Hardware Register Renaming(HRR)	integer register 32 個, icc 32 個, floating point register 32 個, fcc 32 個

表 2: ベンチマークプログラム

ベンチマークプログラム	実行条件
compress	compress 操作のみ。decompress は行なわない。
xlisp	シミュレーション時間の短縮のため、6queen で実行
gcc	入力 file は cexp.i のみを使用
sc	入力は short input を使用
eqntott	入力は int_pri_3.eqn を使用
espresso	入力 file は bca.i を使用

4.3 最適化項目

実行速度を向上させるためにコンパイラができることは、実行命令数を減らすこと、および、IPC(Instructions Per Cycle) を大きくすることである。実行命令数を減らすことは、conventional な最適化でもかなり行なわれているので、ここでは考えない。IPC を大きくするためにはおもに、命令レベルの並列度を上げることと、分岐のオーバーヘッドを小さくすることが考えられる。そこで、以下、命令レベルの並列度を上げる最適化と分岐のオーバーヘッドを減らす最適化について検討していく。

5 命令レベルの並列度を上げる最適化

命令レベルの並列度を上げる最適化として、Global Scheduling がある。特に Int 系のプログラムでは、基本ブロック内だけでは充分な並列度が得られないことが多い。Global Scheduling は、基本ブロック内だけでなく、複数のブロックにまたがって (分岐命令を越えて) 命令を移動することにより、パイプラインの空きサイクルを埋めたり、命令レベルの並列度を上げるようにする手法である。代表的なものに、Trace Scheduling[1] や Percolation Scheduling[2] がある。Int 系のプログラムは分岐が多く、しかも、必ずしも分岐に偏りがあるとは限らない。したがって、分岐に偏りがあることを仮定する Trace Scheduling はあまり有効ではないと考えられる。そこで我々は、 Percolation Scheduling を実装してその評価を行なうことにした。

5.1 Percolation Scheduling

Percolation Scheduling は、命令をプログラムの下方から上方へできるだけ上げることにより、命令レベルの並列度を上げることを狙う。Percolation Scheduling では、条件分岐命令を越えて命令を移動する仕方に、non speculative なコード移動と、speculative なコード移動の 2 つがある。

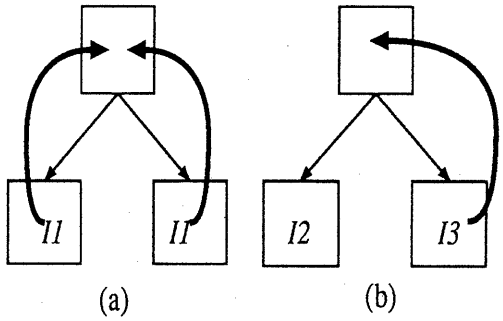

図 1: non speculative なコード移動と speculative なコード移動

non speculative なコード移動とは、図 1(a) のように、条件分岐命令の後続の 2 つのパスに同じ命令がある時にのみ、その命令を上方へ移動することを言う。speculative なコード移動とは、図 1(b) のように、条件分岐命令の後続の 2 つのパスのうち、片方にしかない命令を上方へ移動することを指す。図 1(b) の場合、I2 側へ分岐すれば、実行はかえって遅くなる可能性がある。

我々は non speculative なコード移動のみを行なった。これは以下の理由による。

- まず、non speculative なコード移動だけでどのくらい性能が良くなるかを調べる。
- ハードウェア自体も speculative 実行を行なっているので、少しばかり speculative なコード移動をしても効果が見えないと考えられる。また、そもそも分岐命令が多いので、複数の分岐命令を越えて命令を移動する必要があるが、そのためには、コンパイル時にかなり高い精度で分岐予測ができなければならない。しかしこれは、profiler を利用するなどしない限り、一般には難しい。BTB を使用する場合は、かえって性能が悪くなる可能性が高い。

表 3: Percolation Scheduling による IPC の変化

	BASE	HRR	BTB	HRR +BTB
compress	9.8%	6.5%	3.4%	0.8%
espresso	-0.8%	-0.4%	0.4%	1.5%
eqntott	12.0%	13.7%	-2.2%	-1.0%
xlisp	0.6%	0.0%	0.4%	-1.0%
gcc	-0.7%	-1.0%	-0.8%	-1.4%
sc	1.8%	1.6%	3.2%	3.6%

表 4: Percolation Scheduling による speedup

	BASE	HRR	BTB	HRR +BTB
compress	10.5%	4.0%	7.1%	1.5%
espresso	-0.8%	-0.4%	0.4%	1.6%
eqntott	19.5%	21.3%	4.3%	5.7%
xlisp	0.3%	0.3%	0.0%	-1.3%
gcc	-1.0%	-1.1%	-1.1%	-1.5%
sc	1.0%	0.8%	2.4%	2.9%

5.2 Percolation Scheduling の効果

表 3と表 4 に Percolation Scheduling による IPC の変化 と speedup を示す。たとえば、compress を HRR や BTB を使用しないで実行した場合、Percolation を行なったコードは行なわないコードに比べて、IPC は 9.8%増え、実行スピードは 10.5%向上している。また、HRR の場合には、IPC は 6.5%増え、スピードは 4.0%向上している。compress と eqntott 以外のプログラムに対しては IPC はほとんど変わらず、speedup もほとんどない。

compress と eqntott の場合の speedup の原因は以下の通りである。

- compress

  BASE の場合は、10%の性能向上がある。それは、ブロックのコピーにより無条件分岐命令が削除され、ブロック長が長くなったためである。

- eqntott

  BASE の場合は、20%近い性能向上がある。これは、ブロックのコピーにより、冗長な比較命令と分岐命令が削除されたためである。その分岐命令は、not taken であると予想すると大体外れる。したがって、BTB がなければ、その分岐命令が削除された効果は大きい。逆に、BTB があれば効果が小さくなる。HRR だけを使用しても両者の性能差にはほとんど影響しない。

結局両者とも、分岐のオーバーヘッドが減少したために IPC が増加している。Percolation を行なっても、命令レベルの並列度がさらに増加したわけではない。

Percolation Scheduling により命令レベルの並列度が上がらない理由としては以下のことが考えられる。

- speculative なコード移動を行なっていない。
- Int 系プログラムは元々並列度が小さい。
- Int 系プログラムでは、静的には解決できないデータ依存が多く存在する。そのため、あまり命令を広範囲に移動することができない。
- 狭い範囲での命令移動の効果は、 動的な scheduling や speculative execution の陰に隠れてしまっている。

6 分岐のオーバーヘッドを減らす最適化

ここでは、Loop Unrolling と前方 Short Jump の削除について検討する。Loop Unrolling は、分岐命令が taken する場合のオーバーヘッドを軽減する最適化であり、前方 Short Jump の削除は、分岐命令そのものを削除してしまう最適化である。

表 5: Loop Unrolling による speedup

	BASE	HRR	BTB	HRR +BTB
compress	3.3%	39.4%	-0.1%	0.0%
espresso	9.2%	8.4%	7.3%	13.2%
eqntott	26.5%	28.5%	6.1%	8.0%
xlisp	9.1%	10.3%	11.0%	11.3%
gcc	0.7%	0.6%	-1.6%	-1.6%
sc	1.0%	1.0%	1.7%	2.5%

6.1 Loop Unrolling

一部のコンパイラは、while 文を unrolling しないものがあるが、GCC は、loop body がある大きさを越えない限り、どんな loop も unrolling することができる。Int 系のプログラムに現れる loop は、実行してみて始めて繰り返し回数がわかったり、loop body が複雑な構造をしているものが多い。このような場合、異なる iteration をオーバーラップさせて実行することはできず、n 回 unrolling しても同じ loop body が n 個並ぶだけのことも多い。しかし、そのような場合でも、unrolling したコードの方が有効な命令をフェッチする確率が高くなるし、loop body 中の命令数が減る場合もある。また、BTB を用いないならば、実際に分岐するオーバーヘッドを減らすことができるので、効果があると考えられる。

6.1.1 Loop Unrolling の効果

表 5 より、gcc と sc 以外に対しては効果があることがわかる。

compress は特異的である。BTB があると、unrolling の効果は完全に相殺されてしまっている。HRR の場合に 40%近い speedup があるのは、renaming により異なる iteration をオーバーラップして実行できるようになったためである。

普通は、BTB を使用すると、Loop Unrolling の効果は小さくなってくる。しかし、espresso や xlisp では、BTB を使用した場合でも、unrolling の効果があまり小さくなっていない。この理由としては、以下のことが考えられる。

```
(変換前)  if (q < 0) a = 0;
(変換後)  a &= 0 - !(q < 0);
```

(a)C 言語による例

```
(変換前)  cmp %o1,0
          bl,a  L1
          mov 0, %o2
        L1:

(変換後)  sra %o1,31,%o3
          and %o2,%o3,%o2
        L1:
```

(b) アセンブラによる例

図 2: Superoptimizer の例

espresso のある while loop は、一度 loop に入ると、ほとんど 4 回回って loop から外へ出る。unrolling しない場合、loop の最後の条件分岐命令は、taken となる確率は約 80%、not taken となる確率は約 20%である。一方、5 回 unrolling すると、5 個の条件分岐命令が生じる。loop はほとんど 4 回回るので、最初の 4 つの分岐命令はほとんど not taken となる。また、5 番目の分岐命令は、そこで loop から脱出することが多いので、やはり not taken となることが多い。すなわち、unrolling しない場合の 1 個の分岐命令の予測の精度よりも、5 回 unrolling した 5 個の各々の分岐命令の予測の精度の方が高くなることがあり得る。

6.2 前方 Short Jump の削除

これは、Superoptimizer[3] による最適化を拡張したものである。

6.2.1 Superoptimizer

Superoptimizer による最適化とは、分岐を含んだ文を、分岐を含まない等価な文に変換する最適化である。C 言語およびアセンブラ による例を図 2に示す。この変換は以下の2つのステップからなる。

1. 比較の結果 (0 or -1) を汎用レジスタにセットする。いわゆる SCC 命令であるが、SPARC にはこのような命令はないので、一般に、複数の命令列でこれを実現することになる。

2. 条件分岐命令とラベルとの間にある命令列は、条件分岐が taken でも not taken でも意味が変わらないように変更する。

6.2.2 Superoptimizer の拡張

[3] では、図 2(b) のように、条件分岐命令とラベルとの間に $a=0$ や $a=1$ のような簡単な命令が存在する時に適用した例だけが載っている。これを任意の ALU 命令に対して行なうように拡張する。すなわち、$if(Q)a = X$ は、$a = a_0 + (X - a_0)\&(0-!Q)$ のように変換する。(ただし、a_0 は、if 文を実行する以前の a の値を表すものとする。) 現在は、条件分岐命令とラベルとの間に ALU 命令が 1 個だけ存在する場合にだけ適用している。

6.2.3 前方 Short Jump の削除 の効果

前方 Short Jump の削除 の効果を表 6に示す。compress と eqntott に対しては有効であることがわかる。

特に BTB がある場合の eqntott の speedup が大きい。これについてもう少し詳しく検討するために、HRR+BTB の場合の分岐予測の精度に着目する。表 7は、この最適化の有無により分岐予測の精度がどのように変化するかを示したものである。ここで、

- 「BTB ヒット率」とは、BTB にヒットした割合を表す。

表 6: 前方 Short Jump の削除 による speedup

	BASE	HRR	BTB	HRR +BTB
compress	5.7%	2.3%	9.1%	-0.4%
espresso	0.2%	0.1%	-0.4%	-0.2%
eqntott	7.8%	6.4%	30.8%	46.3%
xlisp	1.3%	1.4%	0.5%	-0.4%
gcc	1.1%	1.1%	0.2%	0.6%
sc	0.3%	0.5%	2.0%	2.2%

表 7: 分岐予測の精度 (eqntott)

	GCC	GCC+前方 Short Jump の削除
BTB ヒット率	99.998%	99.997%
BTB 予測正当率	81.964%	98.068%
分岐予測正当率	80.128%	94.178%

- 「BTB 予測正当率」とは、BTB にヒットした分岐命令のうち、分岐予測が正しく行なえた割合を表す。

- 「分岐予測正当率」とは、全分岐命令 (BTB にヒットしなかった分岐命令も含む) のうち、分岐予測が正しく行なえた割合を表す。

さて、「BTB ヒット率」は、この最適化の有無に関わらず、ほとんど100%である。しかし、「BTB 予測正当率」は、この最適化がないと、17%悪くなっている。その理由は以下の通りである。

すなわち、この最適化により削除された分岐命令は、実行頻度が非常に高いが、分岐の方向に偏りがそれほど多くはない。したがって、単純な 2-bit BTB では、予測が結構外れてしまう。これに対し、「前方 Short Jump の削除」を行なうと、そのような分岐命令を削除してしまうので、「BTB 予測正当率」が高くなり、最終的に、「分岐予測正当率」が 14%

改善される。

7 まとめ

out-of-order 実行、分岐命令を越えての先行実行を行なうスーパースカラマシン上で、SPECint のような非数値計算プログラムを実行する場合、コンパイラによりさらに性能を上げるためにはどのような最適化が有効であるか検討した。その結果、以下のような知見が得られた。

1. non speculative なコード移動だけを行なう場合、Percolation Scheduling を用いても、さらなる命令レベルの並列度の増加は見込めない。

   一部のプログラムで speedup につながっているのは、ブロックのコピーにより分岐命令が削除され、分岐のオーバーヘッドが軽減されたためである。命令レベルの並列度が増えたためではない。

2. Int 系のプログラムでも、loop は実行頻度が高いことが多い。したがって、Loop Unrolling は、たいていのプログラムに対して有効である。また、BTB がある場合でも、

   - 有効な命令をフェッチできる確率が高くなる。
   - loop body 中の命令数が減ることがある。
   - 平均の繰り返し回数の少ない loop に対しては、unrolling した方が分岐予測の精度が高くなり、性能が良くなることがある。(espresso などの場合)

   などの理由により性能アップにつながる。

3. 分岐命令そのものを削除してしまう Short Jump 命令の削除は、短い if 文を持つプログラムに対して効果がある。分岐命令自体を削除してしまうので、分岐に偏りが少く、かつ、BTB を使用している場合には、特に効果がある。(eqntott の場合)

8 今後の課題

命令レベルの並列度を上げるために、speculative なコード移動を含んだ Global Scheduling、loop の オーバーラップ (異なる iteration の並列実行) などを試す必要がある。また、ハードウェアをより単純にした場合 (たとえば、in-order 実行) には、どのような最適化が有効であるかなども検討していく予定である。

参考文献

[1] J. A. Fisher, Trace Scheduling: A Techinique for Global Microcode Compaction. IEEE Transactions on Computers, Vol.C-30 pp.478-490, July 1981

[2] A. Aiken and A. Nicolau, A Development Environment for Horizontal Microcode. IEEE Trans. Soft. Eng., vol.14, no.5, pp.584-594, May 1988

[3] T. Granlund and R. Kenner, Eliminates Branches using a Superoptimizer and the GNU C Compiler. Proc. of the SIGPLAN'92, pp.341-352