プロセッサ仕様記述を用いたコードスケジューラの実装

Generation of code schedulers from processor description

加藤 吉之介　KATO Yoshinosuke
脇田 建　WAKITA Ken
佐々 政孝　SASSA Masataka

東京工業大学 大学院 情報理工学研究科
Graduate School of Information Science and Engineering,
Tokyo Institute of Technology

概 要

近年のプロセッサで高い実行性能を得るには、命令のスケジューリングが欠かせない。特定のプロセッサに特化したスケジューラは、生成されるコードの性能は高いが、移植が困難である。一方、プロセッサ仕様記述を用いることで移植を容易にする手法がある。しかし、生成されるコードの性能は高くなく、仕様記述が複雑であるという欠点がある。そこで本研究は、命令間の依存関係と資源競合に着目し、プロセッサ仕様記述を簡略化した。この仕様記述を用いることで高い移植性を持ち、かつ、実行効率の良いコードを生成するコードスケジューラの実装を目指した。実験により、gcc と比較して最大 14%の速度向上が得られた。

1　はじめに

近年のプロセッサは命令レベル並列と呼ばれる、複数の命令をプロセッサ内部で同時に実行可能なアーキテクチャとなっている。命令レベルの並列性を得るための主な技術には、パイプライン処理やスーパースカラがある。パイプライン処理は命令の実行を複数のステージに分割し、各ステージでの処理を同時に実行することで並列性を得る。スーパースカラでは、プロセッサの中に複数の独立に実行可能な演算ユニットを持たせ、同じステージの中でそれらを同時に実行することによって並列性を得る。

これらの技術を阻害する要因として以下の三種がある。まず、命令間の依存関係があるときには、ある命令の影響を被る命令の実行は前者の実行が完了するまで、実行を開始できない。次に、条件つき分岐命令の実行では、分岐条件が定まるまでは次に実行すべき命令を確定できない。最後に、複数の命令が同じハードウェア資源を要求するときには、それらを並列に実行できない。これを資源競合と呼ぶ。

これらの阻害要因のうち、命令間の依存関係、および、演算ユニットに関する資源競合によって生じる命令実行の遅延を、命令列の並び変えによって解消するのがスケジューリングである。通常、スケジューリングはコンパイラの最適化フェイズで行われる。

商用のコンパイラは、特定のプロセッサに特化することによって効果的なスケジューリングを行い、非常に実行効率の良いコードを生成する。しかし、対象が特化されているため他のプロセッサに移植するのは困難である。一方、gcc のようなコンパイラは、プロセッサ仕様記述を用いることで多くのプロセッサへ移植が容易である。だが、各プロセッサの特徴を考慮した最適化は十分ではない。また、そのような最適化を行うには複雑な仕様記述を書かなければならない。

本研究では、プロセッサ仕様記述を、スケジューリングに必要な命令間の依存関係と資源競合に関する記述のみにすることで簡略化した。この仕様記述を用いて、高い移植性を持ち、かつ、実行効率の良いコードを生成できるコードスケジューラの実装を目指した。

2　スケジューリングの定式化

コードスケジューリングとは、命令列の実行順序を変えることで、より実行効率の良い同等な命令列を得る技術である。コードスケジューラは、第 1 節のさまざまな阻害要因を避けるような実行順序を見つけることによって、より効率の良い命令列に並びかえる。

2.1　命令間の依存関係

スケジューリングによって命令列の処理の意味を変えないためには、スケジュールされた命令列が元の命令列の依存関係を保存することが重要である。Johnson は、命令間の依存関係は以下の 3 種に分類

できるとしている [1]。これらの依存関係は、先に実行される命令 op と後に実行される命令 op' がそれぞれ定義する資源 (def)、使用する資源 (use) を利用するときに、以下のように記述できる。

データ依存関係 (WR)　先行命令が値を書き込んだレジスタを、後続命令が使用すること。

$$def(op) \cap use(op') \neq \emptyset$$

逆依存関係 (RW)　先行命令が使用するレジスタの値を、後続命令が更新すること。

$$use(op) \cap def(op') \neq \emptyset$$

出力依存関係 (WW)　先行命令が値を書き込むレジスタに、後続命令も値を書き込むこと。

$$def(op) \cap def(op') \neq \emptyset$$

依存関係にある二つの命令は、先行命令の実行が完了するまで後続命令の実行を開始できない。命令の実行が完了するとは、その命令の演算結果をそれ以降に実行される命令で利用可能となることである。メモリから値を読むロード命令などは、多くのプロセッサで、実行開始から演算結果が利用可能となるまで複数のプロセッササイクルを必要とする。このサイクル数をレイテンシと呼ぶ。

本研究では、プロセッサの仕様記述の中で、命令セットのそれぞれの命令に対して source レジスタと destination レジスタを定義することによって、def と use を計算している。そして、その情報をスケジューリングに利用している。この定式化によってプロセッサに依存しないスケジューラの記述が可能になった。

2.2　資源競合

資源競合のために同時に実行できない二つの命令を近接して配置すると、一方の命令の実行が遅延される。このために、プロセッサの潜在的な並列性を活かしきることができない。スケジューリングでは、このような場合を考慮し、プロセッサの並列性を活かす命令列に並び変える。

2.3　プロセッサ仕様記述

プロセッサ仕様記述とは、各プロセッサごとに異なる、プロセッサの演算ユニットに関する情報と命令セットに関する情報を、コードスケジューラに与えるものである。演算ユニットの情報はスケジューリングに、命令セットの情報は命令間の依存関係解析とスケジューリングに必要となる。

プロセッサの演算ユニットに関する情報とは、プロセッサに存在する演算ユニットの種類と数である。

命令セットに関する情報とは、命令セットの各命令に対する次の三つである。まず、その命令の source レジスタ、destination レジスタがオペランドの何番目であるかである。次に、その命令が使用する演算ユニットである。最後に、その命令のレイテンシである。

例として UltraSPARC I の演算ユニット仕様記述と命令セット仕様記述をそれぞれ図 1、図 2 に示す。記述にあたっては [2, 3] を参考にした。ALU や FPU の記述を個数ではなく 1、2 と記述するのは、各演算ユニットによって実行できる命令のクラスが異り、個々の演算ユニットを判別する必要があるからである。addcc 命令の%xcc や%icc のように、オペランドに明示的に現れないフラグに暗にアクセスする命令は、それらのフラグを仕様記述内に明示した。

ALU	1,2
FPU	1,2
LD/ST	1

図 1　UltraSPARC I の演算ユニット仕様記述

命令	src	dest	use resource	レイテンシ
add	op1, op2	op3	ALU_1 or ALU_2	1
addcc	op1, op2	op3, %icc, %xcc	ALU_2	1
mov	op1	op2	ALU_1 or ALU_2	1
load	op1, op2, mem	op3	LD/ST	2
⋮				

図 2　UltraSPARC I の命令セット仕様記述 (一部)

3　実装

本研究で実装したコードスケジューラは、アセンブリコードに対して三つのステップによってスケジューリングをおこなう。そのステップとは、字句・構文解析、命令間の依存関係解析、スケジューリングである。なお、現在の実装では依存解析の簡略化のため、スケジューリング範囲を基本ブロック内に限定している。

3.1　命令間の依存関係解析

依存関係解析の結果からデータフローグラフを作成する。スケジューリングの範囲を基本ブロックに限定することによって、第 2 節で述べた依存関係の解析を、各命令の def と use を比較することで 1 パスで行えるようにした。

3.2 スケジューリング

依存関係解析により作成されるデータフローグラフに対して、適当なスケジューリングアルゴリズムによりスケジューリングを行う。現在の実装では、リストスケジューリング [4] を行っている。リストスケジューリングの大まかなアルゴリズムを解説する。

まずデータフローグラフの各パスについて、命令のレイテンシを元にパス全体の実行に要するサイクル数を計算する。このサイクル数に従って、各命令のスケジューリング優先度を定める。実行に時間のかかるパス上の命令は優先度を高く、短時間で実行できるパス上の命令は優先度を低く定める。

次に、1 サイクルごとに実行可能な命令の集合を求め、そのうち優先度の高いものからスケジューリングしていく。

以下の行列の積を求める C 言語プログラムを例に、本研究のスケジューリング効果について説明する。

```
for (i=0; i<300; i++)
  for (k=0; k<300; k++)
    for (j=0; j<300; j++)
      c[i][j] += a[i][k]*b[k][j];
```

これを UltraSPARC I 上で gcc 2.8.1 を用いて-O3 -mv8[1] オプションでコンパイルしたアセンブリコードのうち、最内ループを以下に示す。各命令の横に付加した WR、RW、WW はそれぞれ、その命令が依存している命令とその依存関係を表している。例えば 2 行の smul 命令は、1 行目の ld 命令にデータ依存していることを表す。

```
    .LL33:
1       ld [%o7+%i2],%g3
2       smul %g4,%g3,%g3    WR1
3       ld [%o0+%i0],%g2
4       add %g2,%g3,%g2     WR2, WR3, WW3
5       st %g2,[%o0+%i0]    WR4, RW3
6       add %i2,4,%i2       RW1
7       add %i1,1,%i1
8       cmp %i1,299         WR7
9       ble .LL33           WR8
10[2]   add %i0,4,%i0       RW3, RW5
```

図 3 に、このコードが UltraSPARC I で実行される様子を示す。ALU1 と 2、LD/ST は、命令が使用している演算ユニットを表している。W はその番号の ld 命令の結果が、どのサイクルで利用可能になるかを表している。左端の数字は実行サイクルである。各命令の配置は、どのサイクルで命令が実行開始されるかを表している。各命令のレイテンシは、ld 命令が 2、その他の命令は 1 である。各演算ユニットで実行できる命令は以下の通りである。

ALU1 : smul、ble、add

ALU2 : cmp、add

LD/ST : ld、st

また各命令が演算ユニットを占有するサイクル数は、smul 命令が 4 サイクル、その他の命令は 1 サイクルである。表の • は、演算ユニットが占有されており、他の命令を実行できないことを表す。

	ALU1	ALU2	LD/ST	W
1			ld_1	
2				
3	$smul_2$		ld_3	1
4	•			
5	•			3
6	•			
7[3]	add_4	add_6	st_5	
8	add_7			
9[4]	ble_9	cmp_8		
10	add_{10}			

図 3　gcc によるコードの UltraSPARC I での実行の様子

この gcc の出力コードを、本研究で実装したシステムを用いてスケジューリングする。そのコードを実行すると、図 4 のように実行サイクルが 10 から 8 へと減少する。これは、他の命令に依存しない add_7 命令と、その add_7 命令のみに依存している cmp_8 命令の実行を早めることによって、元のコードでは利用されていなかった 1、2 サイクル目の ALU を利用できるからである。

	ALU1	ALU2	LD/ST	W
1	add_7		ld_1	
2	add_6	cmp_8	ld_3	
3	$smul_2$			1
4	•			3
5	•			
6	•			
7	add_4		st_5	
8	ble_9	add_{10}		

図 4　本システムでスケジューリングしなおしたコードの UltraSPARC I での実行の様子

参考のために、以下に gcc で UltraSPARC I 向けに最適化した[5]場合の結果を示す。これを見ると、このコードに対する gcc のスケジューリングは最適ではないことがわかる。

	ALU1	ALU2	LD/ST	W
1			ld_1	
2	add_7		ld_3	
3	$smul_2$			1
4	•			3
5	•			
6	•			
7	add_4	add_6	st_5	
8	ble_9	cmp_8		
9	add_{10}			

図 5　gcc で UltraSPARC 向けに最適化したコードの UltraSPARC I での実行の様子

[1] SPARC v8 アーキテクチャコード生成オプション
[2] delay slot の扱いについての説明は省略する
[3] add の実行結果を書き込む st 命令は、その add 命令と同時に実行可能
[4] icc フラグをセットする命令と、それを使用する条件分岐命令は同時に実行可能
[5] -O3 -mcpu=ultrasparc -mtune=ultrasparc

3.3 実装の制限

現在の実装は、以下の仮定にもとづいて行っている。まず、メモリ操作命令のアクセスするメモリアドレスに関しては、アドレスにかかわらず仮想的なmemという一つのレジスタへの操作であるとする。これは、alias解析を行う手間を省くためである。次に、メモリアクセスを行う命令のレイテンシに関しては、全て1次キャッシュにヒットした場合のレイテンシを考える。これは実行時でないとデータがキャッシュに存在するかメモリに存在するか判別できないためである。キャッシュミスペナルティを考慮していないため、実際の実行時にキャッシュミスが起こる場合にはスケジューリングされた命令列は効果的であるとは言えない。しかし、多くのアプリケーションでは時間的・空間的局所性から、データの多くはキャッシュに乗ることが期待できるので、それほど悪いスケジューリング結果にはならないと思われる。

4 実験

本研究で実装したシステムを用いて、gccの生成するアセンブリコードを対象に、UltraSPARC IとAlpha 21164PC上で実験を行った。実験に用いたコードは、SPEC CINT95のcompressとliを-S -O3オプションでコンパイルしたアセンブリコードである。これを本研究で実装したシステムで再スケジュールしたものと、gccで-mcpuオプションを用い各プロセッサ向けに最適化したものの実行時間を比較した。

実行環境
CPU: UltraSPARC I 143MHz OS: Solaris 2.6 Memory: 288MByte gcc: 2.8.1

	gcc(1)	gcc(2)	本システム
compress	424.8	424.6	412.7
li	11.6	9.9	10.0

gcc(1)：gcc -O3 -mv8
gcc(2)：gcc -O3 -mcpu=ultrasparc -mtune=ultrasparc
本システム：gcc -O3 -mv8 -S の出力を本システムで再スケジューリング

表 1 UltraSPARC I での実行時間の比較 (単位:秒)

実行環境
CPU: Alpha 21164PC 533MHz OS: Linux 2.2.14
Memory: 128MByte gcc: 2.95.3

	gcc(1)	gcc(2)	本システム
compress	276.0	253.7	267.9
li	4.7	5.1	4.8

gcc(1)：gcc -O3
gcc(2)：gcc -O3 -mcpu=21164pc
本システム：gcc -O3 -S の出力を本システムで再スケジューリング

表 2 Alpha 21164PC での実行時間の比較 (単位:秒)

UltraSPARC Iでの実験では、gcc(1)のコードと比較して、compressで3%、liで14%の速度向上が得られた。また、UltraSPARC向けに最適化されたコードであるgcc(2)と比較しても、compressで3%高速、liでも1%程度劣っているだけである。

Alpha 21164PC での実験では、compress ではgcc(1)のコードと比較して3%程度の速度向上が得られたが、gcc(2)のコードよりは6%ほど劣っている。また、liでは最適化するほど実行速度が遅くなっている。この原因は現在調査中である。

5 まとめと今後の課題

本論文では、簡単なプロセッサ仕様記述を用いて、アセンブリコードを特定のプロセッサに特化したコードに変換するコードスケジューラを提案した。実験により、gccで特定のプロセッサ向けに最適化したコードと同程度の実行性能が得られることを示した。

関連研究としては、目的とするプロセッサの仕様記述から、そのプロセッサに特化した最適化を行うシステムであるPO[5]やTOAST[6]などがある。これらのシステムは、詳細なプロセッサ仕様記述を用いて最適化を行うので、本研究のような単純なスケジューリング以外にも複数の命令をプロセッサ独自の複合命令に変換するような最適化等が行える。そのため、本研究と比較して生成されるコードの実行効率は向上するが、ソースコードの解析に時間がかかる。また、[7]は本研究と同様にプロセッサ仕様記述からコードスケジューリングを行うシステムであるが、スケジューラの実装が特定のプロセッサに特化したものとなっている。

今後の課題としては、まずスケジューリングアルゴリズムの工夫があげられる。しかし、基本ブロックを越えたスケジューリングを行うアルゴリズムを実装しようとすると、現在のような単純な依存解析では不十分となり、基本ブロック間のデータフロー解析を行わなければならなくなるので、現在の手法では難しいと思われる。

また、このシステムをコンパイラのバックエンドへ容易に組み込めるようにすることで、コンパイラの最適化フェーズの作成が容易になると期待できる。

参考文献

[1] W. Johnson: *Superscalar Microprocesser Design,* PTR Prentice Hall, 1991.

[2] D. Weaver and T. Germond: *The SPARC Architecture Manual Version 9,* PTR Prentice Hall, 1994.

[3] Sun microsystems Inc.: *UltraSPARC User's Manual,* 1997.

[4] T. Adam, K. Chandy, and J. Dickson: *"A Comparison of List Schedules for Parallel Processing Systems,"* Communications of the ACM, Vol. 17 (December 1974), pp. 685-690.

[5] J. Davidson and C. Fraser: *Automatic Generation of Peephole Optimizations,* in Proc. of Symp. on Compiler Construction, pp. 111-116, June 1984.

[6] R. Hoover and K. Zadeck: *Generating Machine Specific Optimizing Compilers,* in Proc. of Principles of Programming Languages '96, pp. 219-229, January 1996.

[7] 長谷川 勇, 鈴木 正人, 今泉 貴史: プロセッサ仕様の形式的記述からの最適化コンパイラの自動生成, 日本ソフトウェア科学会第16回大会論文集, pp. 449-452, 1999.