計算機アーキテクチャ 134－20
（１９９９．８．３）

コンパイラのメディア命令対応とその評価

細井 聡＊　新井 正樹＊＊　小澤 年弘＊＊　木村 康則＊＊

＊富士通研究所　＊＊新情報富士通研

近年、メディア命令を持つプロセッサが増えてきたが、効率の良いメディアコードを自動生成することはかなり難しい。そこで、全てを自動生成する代わりに、メディア中間表現記述という記述を高級言語に追加し、それを用いてプログラマがソースを書き換え、書き換えたソースをコンパイラがコンパイルすることにより、メディアコードの生成を試みた。ある程度効率の良いコードが、アセンブラでプログラミングするよりかなり容易に得ることができた。

MIR-Description: Mechanism for efficient media code generation

Akira Hosoi*, Masaki Arai**, Toshihiro Ozawa**, Yasunori Kimura**

*FUJITSU Laboratories LTD.
**RWCP Multi-Processor Computing Fujitsu Laboratory

4-1-1. Kamikodanaka Nakahara-ku, Kawasaki 211-8588, Japan

email: hosoi@flab.fujitsu.co.jp

Many processors recently have media instructions. But it is difficult that today's compilers automatically generate efficient media code. So we propose 'Media Intermediate Representation Description'(MIR Description), by which programmer can specify which media instructions should be generated, and constraints for register allocation. We rewrote a small part of mpeg2 programs etc. with MIR Description. The rewritten programs were easily compiled into the efficient media code.

1 はじめに

近年、いわゆるメディア命令を持つプロセッサが増えてきた。しかし、効率の良いメディアコードを自動生成することは、現在のコンパイラ技術ではかなり難しいため、アセンブラを用いることが多い。しかし、アセンブラは開発効率や保守性、ポータビリティなどの点で難がある。

そこで我々は、全てを自動生成する代わりに、メディア中間表現記述（Media Intermediate Representation 記述、以下 MIR 記述と略す）という記述を高級言語（C 言語）に追加した。プログラマは MIR 記述を用いてソースを書き換え、書き換えたソースをコンパイラがコンパイルすることにより、効率の良いメディアコードを生成することを試みた。

本論文の構成は以下の通りである。2章では、代表的なメディア命令について紹介する。 3章では、メディア命令の自動生成が難しい理由について述べる。4.1章では、 MIR 記述による書き換えについて説明する。5章では、 MIR 記述の代わりに、アセンブラや新しい高級言語を用いた場合との比較を行なう。 6章では、 mpeg2 などに MIR 記述を適用した結果について述べる。最後にまとめを行なう。

2 メディア命令について

メディア命令とは、イメージ、グラフィックス、音声などのアプリケーションを高速に実行するために導入された命令群である。各プロセッサにより結構違いもあるが、おおよそ、次のような命令がある。

2.1 SIMD 命令

メディア処理では、部分的には並列度が大きく、かつ、32 ビットの演算に加えて 8 あるいは 16 ビットの演算が多い。そこで、1 命令で複数の 8/16/32 ビットの演算を行なう命令が用意されている。

SIMD 命令のペア・レジスタ規則について

16 ビットデータの加算を 4 個同時に実行する命令、

$$madd16x4 \quad f10, f20, f30$$

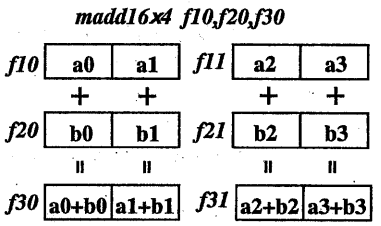

図 1: SIMD 命令

```
if (x > 255)
  x = 255;
else if (x < 0)
  x = 0;
```

(a) 飽和演算の定義

```
x = a[x];
```

(ただし、a[i] は i > 255 なら 255、i < 0 なら 0、それ以外は i に初期化)

(b) テーブル参照による実現

図 2: 飽和演算

は、入力レジスタとして、f10 と f11、および、f20 と f21、出力レジスタとして f30 と f31 を使う[1]（図 1）。SIMD 命令では、このように、必ず、偶数番から始まる連番のレジスタを使わなければならない[2]というペア・レジスタ規則があるのが普通である。もし、並列実行したいデータの一部がペア・レジスタに載っていない場合は、転送命令を用いて、ペア・レジスタに載せてから SIMD 命令を使う必要がある。しかしこれは一般に、命令数を増加させ、サイクル数を伸ばすことになる[3]。

2.2 飽和演算命令

飽和演算とは、図 2(a) のように、ある範囲外の値を最大値あるいは最小値とするような演

[1] レジスタ長が 32 ビットの場合

[2] 4 個のレジスタをペア・レジスタとして使う命令では、4 の倍数で始まる連続する 4 個のレジスタを使用しなければならない。

[3] floating レジスタを用いた転送命令は、3 サイクル程掛かることが多い。

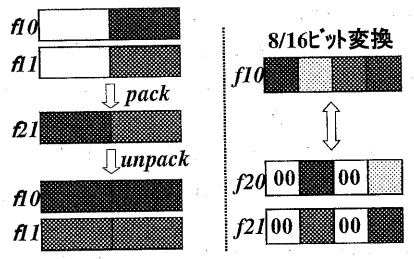

図 3: データ転送／変換命令

算で、イメージやグラフィックスの色計算などでよく行なわれる。

図 2(a) を一般命令で実現すれば、複数の分岐を含んだ命令列となってしまう。また、図 2(b) のようなテーブル参照で実現すれば、分岐命令なしで実現できるが、ロード命令での実現となるので、SIMD 命令程には並列度を上げにくい。

そこで、図 2(a) の処理を 1 命令で実現するような飽和演算命令が用意されることが多い。

2.3 データ転送命令

SIMD 命令を有効に使うために用意されている命令で、データのパック／アンパックや、8 ビットと 16 ビットとの変換、およびその逆などの命令がある（図 3）。

3 メディア命令の自動生成が難しい理由

自動生成が難しいのは、現状のコンパイラの能力による理由、メディア処理の特性による理由、および、双方による理由がある。

3.1 大局観の欠如

一般論として、コンパイラは、局所的な最適化が全体の最適化につながることをある程度前提としている。しかし、メディア処理では、この前提が当たらないことも多い。このことは、以下の理由の多くにも関係してくる。

3.2 演算精度の問題

RISC プロセッサでは、8/16 ビット整数の演算は、32 ビットに拡張されて行なわれるのが普通である。したがって、コンパイラの中間表現においても、演算が本来 8/16 ビットの演算をしていることをきちんと把握していない場合が多い。このため、8/16 ビット整数演算を抽出することは、一般には難しいのが現状である。また、40 ビットなど特殊なデータ長を扱うメディア命令の生成も、言語あるいはコンパイラを拡張しなければ生成は難しい。

3.3 メディア命令の選択の問題

floating のメディア命令ではあまり問題とならないが、integer のメディア命令の場合、どのメディア命令を使うか（メディア命令を使わないのも解の 1 つ）その選択に候補が複数あることが多い。たとえば、2 つの 16 ビットデータを 1 つのレジスタにパッキングする場合、i) 整数レジスタにロード、シフト & マスク、floating レジスタに転送、ii)floating レジスタにロード、パッキングの 2 通り考えられるが、どちらが良いかは、むしろその前後で何をしているかで決まる。たとえば、パッキング操作の前に既にデータを整数レジスタにロードしているならば、ii) よりも i) の方が全体としては効率が良いかもしれない。

3.4 SIMD 命令に関する問題

メディア処理プログラムは、局所的にはかなり並列性があり、どれとどれを並列に実行するか、選択肢がかなり多い場合がある。また、局所的には最適な並列実行の組合せでも、全体を考えると、他の組合せの方が良いこともある。同様に、ペア・レジスタ規則のために転送命令を出してまで SIMD 命令を生成すべきか否かは、その部分だけでは決まらないことも多い。

3.5 メディア命令のコストの問題

メディア命令の使用には、以下のコストが伴う。

- レジスタ間転送のコスト

多くのメディアプロセッサでは、integer のメディア命令は、floating レジスタを使用する。したがって、汎用レジスタから floating レジスタへの転送が必要な場合がある[4] [5]。

- メディア命令は一般命令よりレイテンシが長いことが多い

したがって、これらのコストに見合うだけの並列度がある場所のみにメディア命令を生成する必要がある。しかし、プログラマの書き方によっては、必ずしも充分な並列性が抽出できるとは限らない。

3.6 プログラムの意味理解の困難さ

たとえば、飽和演算を図 2(b) のようなテーブル参照で書かれてしまうと、これが飽和演算であることをコンパイラが自動で検出することは、容易ではない。

3.7 動的情報の必要性

普通コンパイラは、i) ループはよく回る、ii) 多重ループでは、最内ループが最も多く回る、ということを仮定する。しかし、メディア処理ではこの仮定がはずれることも多い。

また、実行時に取り得る値の範囲がわかれば、より少ない命令数で実現できるということがメディア処理にはよく見られる。

以上のような動的な情報の有無が性能に大きく効いてくることが少なくない。

4 MIR 記述

MIR 記述とは、プログラマにコンパイラの中間表現へのインタフェースを提供して、プログラマが指定したメディア命令を、入出力関係や同じくプログラマが指定したレジスタの制約条件などを満たしつつ生成させるための記述である。

[4] さらに、floating レジスタから汎用レジスタへ転送が必要な場合もある。

[5] 直接 floating レジスタにロードし、floating レジスタ上だけで演算を行ない、直接 floating レジスタからストアできる場合には、汎用レジスタへの転送は不要である。

```
foo (a,b,c,d,e,f)
int *a,*b,*c,*d,*e,*f;
{/*  *c = *a + *b; *f = *d + *e;  */
  int a0, b0, d0, e0;
  a0 = *a;  MIR_movgf (a0, "10");  (1)
  d0 = *d;  MIR_movgf (d0, "11");  (2)
  b0 = *b;  MIR_movgf (b0, "20");  (3)
  e0 = *e;  MIR_movgf (e0, "21");  (4)
  MIR_madd32x2 ("10","20","30");   (5)
  MIR_movfg ("30", a0);  *c = a0;  (6)
  MIR_movfg ("31", b0);  *f = b0;  (7)
}
```

図 4: MIR 記述の使用例

4.1 MIR 記述を用いた書き換え

まず、プログラマが

- 使用するメディア命令
- レジスタ割り付けの制約条件

を指定できる MIR 記述を用いて、ソースを書き換える。

コンパイラは、MIR 記述を用いて書き換えられたソースをコンパイルする。そして、C で記述された部分と MIR 記述とを合わせて、命令スケジューリング、レジスタ割り付け、その他、C コンパイラが行なう各種最適化を行なう。(一部の asm 文のように、最適化が制限されるようなことはない)。

図 4は、その注釈に示したような 2 つの加算をメディア命令で実現するために、MIR 記述を使用した例である。図 4において、MIR_*XXX* で記述された部分が MIR 記述である。*XXX* の部分が 1 つのメディア命令に対応していて、() 内の入出力関係を満たすように、メディア命令 *XXX* を生成させることを指示する。図 4の (1) では、アドレス a からロードした値を a0 に代入した後、仮想レジスタ"10" に転送する。(2) も同様。ここで、仮想レジスタ"10" と"11" のように連続する数字はペア・レジスタの指定であり、最終的に、コンパイラがダブルワードのレジスタに割り付ける。

図 5に、コンパイラの最適化ルーチンが管理する、中間表現のイメージ (一部) を示す。[ ]

```
[load ((use (reg SImode 1050)(reg ...
 (def (reg SImode 1056))) nil ...]
[movgf ((use (reg SImode 1056))
 (def (subreg SFmode (reg DFmode ...]
[load ((use (reg SImode 1051)(reg ...
 (def (reg SImode 1057))) nil ...]
[movgf ((use (reg SImode 1057))
 (def (subreg SFmode (reg DFmode ...]
                ...
[madd32x2
  ((use (reg DFmode 1066) (reg ...
   (def (reg DFmode 1068))) ...]
                ...
```

図 5: 図 4の中間表現のイメージ（一部）

```
ldf @(9,0),62; ldf @(12,0),63;
ldf @(8,0),60; ldf @(11,0),61;
madd32x2 60,62,58;
stf 58,@(10,0);
stf 59,@(13,0);
```

図 6: 図 4のアセンブラソース

内が1つ1つの命令に対するデータ構造（中間表現）である。movgfやmadd32x2がloadと同様な中間表現に展開されている。また、subregという表現は、ダブルワードレジスタの上位/下位ワードを表現している。MIR記述は、基本的に1つの
メディア命令を生成するが、コンパイラの最適化により、他のメディア命令と統合されたり、削除されることもある。

図6に、図4のアセンブラソースを示す。コンパイラの最適化により、movgfやmovfg命令が消え、floatingレジスタへ直接ロード、floatingレジスタから直接ストアするコードとなっている。

4.2 メディアマクロによるプログラミング

実際には、図4の(1)～(4)と(6)(7)は、図7のようなマクロを定義して、それを用いてプログラミングする（図8）。可読性、生産性を高めるためである。このように、MIR記述を

```
#define LOADF(ADDR,FREG) {\
  int _RO_ = *(int *)(ADDR);\
  MIR_movgf (_RO_, FREG); }
#define STOREF(FREG,ADDR) {\
  int _RO_;\
  MIR_movfg (FREG, _RO_);\
  *(int *)(ADDR) = _RO_; }
```

図 7: メディアマクロの例

```
foo (a,b,c,d,e,f)
int *a,*b,*c,*d,*e,*f;
{/*  *c = *a + *b; *f = *d + *e;  */
  int a0, b0, d0, e0;
  LOADF (a,"10");                    (1)
  LOADF (d,"11");                    (2)
  LOADF (b,"20");                    (3)
  LOADF (e,"21");                    (4)
  MIR_madd32x2 ("10","20","30");     (5)
  STOREF ("30", c);                  (6)
  STOREF ("31", f);                  (7)
}
```

図 8: メディアマクロの使用例

含んだマクロを特にメディアマクロと呼ぶ。

5 他方式との比較

5.1 アセンブラによるプログラミング

最高の性能が得られる可能性がある反面、開発効率が低い（4並列程度であっても、VLIWやスーパースカラのコードを並列動作を意識しながらアセンブラで書くのは大変）、ポータビリティ性が低い（並列度やレイテンシの異なるマシンに対しては、最適なコードにはならないので、書き直しが必要）などの短所がある。

一方、本方式では、命令スケジューリングやレジスタ割り付けはコンパイラが行なうので、開発効率やポータビリティ性は高い。

5.2 新高級言語によるプログラミング

メディア処理に向いた新しい高級言語を用いれば、C言語を越えた記述が可能となり、自動

表 1: mediabench の書き換え

プログラム	変更した割合	主な処理	Speedup
gsm.enc	50/6000	積和演算	1.80
gsm.dec	20/6000	積和演算 飽和演算	3.42
mpeg2.enc	100/8000	動き探索	2.50
mpeg2.dec	250/10000	動き補償	1.30

生成できる場合は確かに増える。しかし、効率を考えるとどうしても特定のマシン専用となりがちで、普及に疑問がある。また、メディア命令を生成させる必要があるのは全プログラムのごく一部であることも多いが、その割にコンパイラを修正／拡張する割合が大きくなる（本方式では、基本的には MIR 記述を解釈できるようにパーザを拡張するだけで良いのでコンパイラの早期提供が可能である。）。

また、充分効率の良いコードを生成するためには、（現在のコンパイル技術では）MIR 記述のように、ある程度レジスタを意識し、データをどうレジスタ上に載せるかを直接指定できるような low-level の記述がどうしても必要だと思われる。

6 実アプリへの適用

mediabench[1] ベンチマークのうち、mpeg2 と gsm（音声圧縮／復元）を MIR 記述を用いてソースの一部を書き換えた（表 1）。たとえば、gsm エンコードは、全ソース約 6000 行のうちの約 50 行を MIR 記述付きの C で書き換えた。書き換えた部分は積和演算が主であった。

そして、図 9に示すターゲット・アーキテクチャ上での性能をシミュレーションにより求めた。表 1の Speedup は、オリジナルのソースをコンパイルした（メディア命令を含まない）コードの性能に対する、MIR 記述を用いて書き換えたソースをコンパイルした（メディア命令を含んだ）コードの性能である。MIR 記述を用いて書き換えた部分は、ターゲット・アーキテクチャを考慮すると、充分効率の良いメディアコードに変換された。

- 4 命令発行の VLIW プロセッサ
  整数命令最大 2 個、floating とメディア命令合わせて最大 2 個、分岐命令最大 1 個
- 演算器　Load/Store 2、ALU 2、FADD 2、FMUL 2、Media 2、Branch 1
- 最大 floating 性能
  FADD2 個 +FMUL2 個 /cycle
- 最大メディア積和演算性能
  16 ビット整数積和演算 8 個 /cycle

図 9: ターゲット・アーキテクチャ

7 まとめ

C 言語に MIR 記述を追加したソースをコンパイルすることにより、ある程度効率の良いメディア命令コードを生成させることができた。

MIR 記述を用いた方法は、コンパイラによるメディア命令の自動生成と共存させることが可能である（実際、floating の SIMD 命令は一部、自動生成している）。したがって、メディア処理用のコンパイル技術の発展途上での現実的な解の 1 つであると考えている。

参考文献

[1] Chunho Lee, iodrag Potkonjak,and William H.Mangione-Smith, MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, MICRO-30,1997