プログラミング 14−21
（1997. 8. 22）

C 言語と reconfigurable computer との相性に関する一考察

齊藤 正伸† 三浦 誠‡ 多田 好克†

†電気通信大学 大学院 情報システム学研究科

〒182 東京都調布市調布ヶ丘 1-5-1

E-mail: {msaitoh,yoshi}@spa.is.uec.ac.jp

‡NTT データ通信株式会社

あらまし

C 言語で記述したプログラムをハードウェアに変換する場合の特徴について、その高速化の度合を主眼に置いて議論する。C 言語の演算子や条件分岐に関連する文はハードウェア化が容易である。ハードウェア化を行うと、MPU に比べて条件判断と分岐のオーバヘッドが生じにくくなる。このため、エラーなどの稀にしか起きない状態の条件判断を頻繁に行うプログラムをハードウェア化した場合の高速化の度合は大きい。C 言語は MPU を想定した言語であるため、MPU が持つ基本的な演算以外の演算は表現しにくいが、不要なラッチを極力設けずに適切な遅延の組合せ回路を構成することでこの欠点を緩和できる場合があることがわかった。

Suitability of C language for reconfigurable computer

Masanobu Saitoh† Makoto Miura‡ Yoshikatsu Tada†

†Graduate School of Information Systems, University of Electro-Communications

1-5-1 Chofugaoka Chofu-shi Tokyo 182 Japan

E-mail: {msaitoh,yoshi}@spa.is.uec.ac.jp

‡NTT DATA Corporation

Abstract

We have been researching a reconfigurable computer system and the compiler that compiles C source code into hardware descriptions directly.

We found some characteristics. The execution overhead of the condition check will be reduced in case the code rarely rsults true. Combining short delay operations reduces the cycle-time of FPGA execution, compared to the MPU operations.

1 はじめに

近年、FPGA (Field Programmable Gate Arrays) を用いてMPUよりも高速に計算を行うシステムに関する研究が盛んに行われている[3][5][6][7][8]。多様なFPGAの中でも、内部の回路が再構成可能なものを用いて、各アプリケーションごとにその実行に適した回路をFPGA上に実現し、高速に計算を行う方式をreconfigurable computingと呼ぶことが多い。

この方式のシステムの多くは、C言語などの高級言語をハードウェア化するというプログラミング方法を用いる。しかし、C言語プログラムをハードウェア化することを想定した場合のC言語プログラムの特性に関する研究は少ない。たとえば、同一アルゴリズムを異なった表現で実装したものが複数ある場合、これをハードウェアとして合成するとどのような違いが現れるかについて我々は興味がある。また、C言語はMPUを想定して設計されたものであるため、ALUがもつ機能以外の演算の表現力が弱いが、この欠点が克服できるかどうかについても興味がある。

本稿では、C言語をハードウェア化する場合の、言語が持つ特性について述べる。2章では、想定しているシステム構成と、ハードウェア化の基本戦略について述べる。3章ではC言語表現からハードウェアの変換方法の詳細について述べる。4章では、プログラム例をもとに、提案する変換方法によって高速化が可能なことを示す。5章はまとめと今後の課題を述べる。

2 想定するシステムとハードウェア化の基本戦略

本研究で想定しているreconfigurable computerの概略を図1に示す。本来MPU単体で行っていた処理をMPUとFPGAとに分割するため、その間で発生する通信遅延やバンド幅が、システム全体の動作速度に直接影響する。図1ではMPUとFPGAが個別のバスに接続されているが、同一のバスに接続されている場合も想定でき、さらにMPUの一部がFPGAで構成されている場合も考えられる。

C言語プログラムのハードウェア化においては、現状ではターゲットとなるFPGAの構造を特定はしない。これは、現在のFPGAの構造が各FPGAによって大きく違い、またFPGAの構造自体が発展途上にあるためである。以下にハードウェア化の基本戦略を示す。

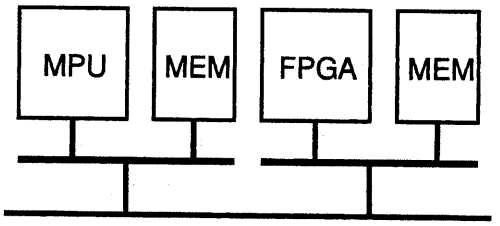

図1: ハードウェアシステム全体の概念図

データフロー構造を直接マッピングする

C言語の式 (expression) はデータフロー的に展開し、その構造を直接マッピングする。ここで直接とは、データフローグラフ内の各演算をそれぞれ個別の回路に置き換えるという意味である。これにより、演算用回路が利用可能になるまで待つということがなくなる。この方式をとると、データフローグラフの枝を実現する配線が増大し、FPGAによっては回路のマッピングが不可能になる可能性がある。本論文では、マッピング可能な規模の回路であると想定して議論をすすめる。

分岐構造はテンプレートを用いてマッピングする

ループ、ifやswitchなどの条件分岐は、あらかじめ用意したテンプレートにあわせてマッピングする。

変数のレジスタによる実現

MPUの場合、レジスタ数は固定であるため、レジスタ上にマッピングできない変数はメモリにマッピングされる。これに対して、FPGA上へマッピングする場合は、変数はできるだけレジスタとしてマッピングする。メモリ上に実現してしまうと、メモリへのアクセス時間がレジスタへのアクセス時間よりも長いことや、メモリの接続されたバスの競合が発生することなどにより、結果として計算時間が長くなる可能性が増す。

3 各C言語表現の変換方法

本節では、C言語の各表現のコンパイル方法について、高速化の度合に焦点を当てて議論する。

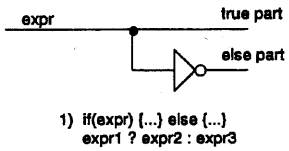

図 2: if 文と条件演算子のテンプレート

3.1 式の合成

C 言語の expression は、前節で述べた通りデータフロー化する。このとき、各演算結果はラッチを通す。但し、以下の演算子が連続している場合は、その間にはラッチを挟まずに、単一の組合せ回路として実現する。

- 定数でのシフト
- ビット単位での論理演算

これらは、ハードウェア化される場合は単なる配線かゲート 1 段の組合せ論理回路であり、四則演算等と比べて遅延が短い。これらが連続する場合は、統合して 1 つの組合せ論理回路を構成することにより、MPU ではレジスタを介して複数クロックかかる処理が、1 クロックで実行できるようになる。

通常、C 言語でプログラムを記述する場合、C 言語の演算子にない演算を行うために、シフト演算とビット単位での論理演算を組み合わせて実現することが多い。このようなプログラムにこの変換方法が有効である。

3.2 ループや分岐の合成

C 言語において分岐に関連する以下のもの

- if 文
- 条件演算子 (<expr1> ? <expr2> : <expr3>)
- switch 文
- for 文
- while 文
- do ··· while 文

について、そのコントロールフローを実現する回路のテンプレートを図 2から図 5に示す。

上記の各 statement において、条件判断式内にポインタによる参照がある場合、

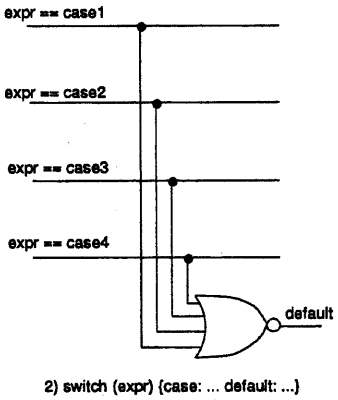

図 3: switch 文のテンプレート

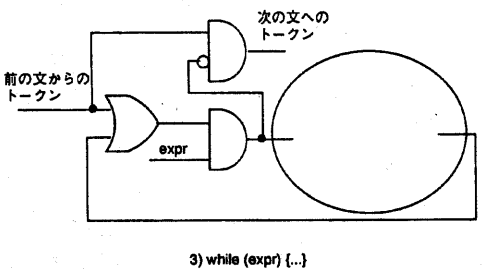

図 4: while 文のテンプレート

- それより前の straight forward なコード内にその読み書きがある
- 加えて、ポインタを介す場合はそれが volatile でない

を満たせば、その直前の参照の時点で、この条件判断内の参照に対するトークンを生成してよい。特に、while 文と do 文の条件式もデータフロー的に展開され、前の statement の完了を待たずに計算してもよい部分は前もって計算してよいため、ループのオーバヘッドが MPU に比べて小さい。

if 文は switch 文の特殊形とみなすことができる。switch 文内の各 case ラベルの式の値は全て異なっていることが保証されている。このため、switch 文の条件式が確定しだい全 case ラベルの比較を行い、一致したラベルに対応するトークンを生成することができる。これに対し、近年のマイクロプロ

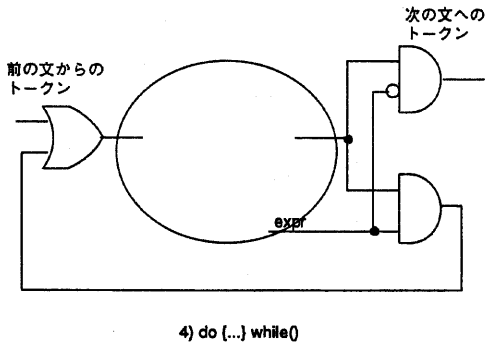

図 5: do ... while 文のテンプレート

```
char *
strchr(p, ch)
        register const char *p, ch;
{
        for (;; ++p) {
                if (*p == ch)
                        return((char *)p);
                if (!*p)
                        return((char *)NULL);
        }
        /* NOTREACHED */
}

size_t
strlen(str)
        const char *str;
{
        register const char *s;

        for (s = str; *s; ++s);
        return(s - str);
}
```

図 6: strchr() と strlen() の C 言語ソースプログラム

セッサはテーブルによるインデクスジャンプが遅いため、case の先頭から順に比較するコードを生成するコンパイラが多い。

また、if 文と switch 文では、条件判断部の遅延が短くかつ各条件に対応する statement の遅延も短い場合は、この 2 つを同一クロックで処理する回路として実現する。

再帰構造のハードウェア化は難しい。これは、FPGA の内部コンテキストを高速に保存する方法がない、もしくは、高速にコンテキストを保存できる FPGA が現在のところないためである。同様な理由により、setjmp() と longjmp() も高速に実現するのは難しい。

4 コンパイル例

本節では、いくつかの合成例を示し、C 言語プログラムのハードウェア化の有効性、欠点について述べる。

4.1 文字列検索

C 言語の標準ライブラリに用意されている strchr() と strlen() について、これらをハードウェア化した場合の動作の違いについて議論する。

図 6に、4.4BSD-Lite で用いられている strchr() と strlen() のソースコードを示す。strchr() と strlen() は似た仕事を行う関数であるが、strchr() は目的の文字かどうか調べながら文字列の終端であるかも同時に調べている。このため、多くの MPU では strlen() よりも strchr() の方が約 2 倍の計算時間を必要とする。これは、たまにしか起きないことを頻繁に調べるために速度低下が起こる例である。同じことが、strcmp() と strncmp() の関係でも成り立つ。近年は、エラーチェックの強化のため strcmp() のかわりに strncmp() の方が好まれることが多い。

これに対し、strchr() と strlen() をハードウェア化した場合は処理速度に違いはない。これは、文字列の終端であるか、一致する文字であるかを同時に調べてよいためである。

ハードウェア化した場合は、このようなオーバヘッドを削減することができる場合があることがわかる。

4.2 7bit ハミング符号のデコーダ

4bit のデータから生成された 7 bit ハミング符号をデコードするプログラムを用いて実験を行う。

通常利用される回路構造を図 8に示す。デコードするプログラムを C 言語で記述したものを図 7に示す。プログラム A は、通常用いられると思われる表現であり (文献 5 のものとほぼ同じ)、プログラム B は、図 8 の回路構造をそのまま再現した例である。この 2 つのプログラムを MPU で実行した場合は、演算、代入、分岐それぞれの数を数えれば、明らかにプログラム A の方が高速である。

プログラム A とプログラム B をそれぞれハードウェア記述言語に変換して論理合成すると、どちらもラッチのない 1 クロックで演算が完了する回路が合成され、遅延も小さい (論理合成のシステムには NTT 社の PARTHENON システム [9] を用い

た)。これに対し、マイクロプロセッサの場合はビット単位での操作が苦手であるため、図7のプログラム A をコンパイルすると非常に長い命令列となる。SPARC の場合は最短で 35 命令以上となる。

MPU 上の命令列と合成された回路を比較すると、C プログラム内にあるビットごとの AND 命令 (&) でマスクして定数シフトを行う式は、現在広く利用されている多くの MPU では 2 命令で実現されるが、ハードウェアで直接実現した場合にはゲートのない配線となる。また、MPU の場合、シンドロームの比較を行う switch 文は、たとえ default ラベルの確率が一番高くとも、インデクスジャンプを行わない限りそれを最初に実行することはできない。

プログラム A を実際に MPU 上のソフトウェアのみとして実行した場合と、ソフトウェアとハードウェアの組合せで実行した場合との違いを測定した。測定した環境は Sun Microsystems 社の SPARC Station 5 (SunOS4.1.4, CPU 速度 70MHz, SBus 速度 23.3MHz) に、我々が設計した無碍ボード [2] と呼ばれる FPGA を用いたボードを載せたものである。このボード上の FPGA を SBus の速度で動作させて測定した。

ソフトウェアとハードウェアの組合せで実行する場合は、デコードするハミング符号のデータは MPU からボードに対して 4 個ずつまとめて送信し、結果も 4 個まとめて受信する。

ソフトウェアのみの場合は、図 7をそのまま用いたプログラムと、入力データとデコード結果をあらかじめ計算しておいた表 (インデクス長 256 の 256 バイトの表) を用いた場合とで測定した。C 言語にない演算を MPU で高速に実現する場合には、テーブルを参照する方法が用いられることが多い。これは、FPGA の内部構造として用いられている lookup table と同じ動作であり、MPU のアクセス可能なメモリで疑似的に組合せ論理回路を実現しているとみなすことができる。

入力にランダムな 10MB のデータを用いた結果を表 1に示す。処理時間は、ユーザ時間とシステム時間の和を示してある。表の通り、忠実に計算するよりも FPGA 上で計算した場合の方が約 3 分の 1 の計算時間で終了する。また、テーブルを引く方法よりも高速になった。MPU と FPGA との動作周波数に違いがあり、さらにバスを介した転送を行うこのような方法であっても、高速なものを実現することが出来た。

```
inline unsigned char
hamm7(word)
   unsigned char word;
{
   unsigned char  b0, b1, b2, b3, b4, b5, b6;
   unsigned char  syn;

   b0 = (word & 0x01);
   b1 = (word & 0x02) >> 1;
   b2 = (word & 0x04) >> 2;
   b3 = (word & 0x08) >> 3;
   b4 = (word & 0x10) >> 4;
   b5 = (word & 0x20) >> 5;
   b6 = (word & 0x40) >> 6;

   syn = b0 ^ b1 ^ b3 ^ b6;
   syn <<= 1;
   syn |= b1 ^ b2 ^ b3 ^ b5;
   syn <<= 1;
   syn |= b0 ^ b1 ^ b2 ^ b4;

   switch (syn) {
      case 0x05:  word ^= 0x01;break;
      case 0x07:  word ^= 0x02;break;
      case 0x03:  word ^= 0x04;break;
      case 0x06:  word ^= 0x08;break;
   }
   return (word & 0x0f);
}
```

プログラム A

```
inline unsigned char
hamm7(word)
   unsigned char word;
{
   unsigned char      b0, b1, b2, b3, b4, b5, b6;
   unsigned char      syn0, syn1, syn2;
   unsigned char      f0, f1, f2, f3;

   b0 = (word & 0x1);
   b1 = (word & 0x2) >> 1;
   b2 = (word & 0x4) >> 2;
   b3 = (word & 0x8) >> 3;
   b4 = (word & 0x10) >> 4;
   b5 = (word & 0x20) >> 5;
   b6 = (word & 0x40) >> 6;

   syn0 = b0 ^ b1 ^ b3 ^ b6;
   syn1 = b1 ^ b2 ^ b3 ^ b5;
   syn2 = b0 ^ b1 ^ b2 ^ b4;

   f0 = ( (syn2 == 1) & (syn1 == 0) & (syn0 == 1)) ? 0x01 : 0x00;
   f1 = ( (syn2 == 1) & (syn1 == 1) & (syn0 == 1)) ? 0x01 : 0x00;
   f2 = ( (syn2 == 0) & (syn1 == 1) & (syn0 == 1)) ? 0x01 : 0x00;
   f3 = ( (syn2 == 1) & (syn1 == 1) & (syn0 == 0)) ? 0x01 : 0x00;

   word =   ((b3 ^ f3) << 3) | ((b2 ^ f2) << 2)
          | ((b1 ^ f1) << 1) | (b0 ^ f0);

   return (word);
}
```

プログラム B

図 7: 7bit ハミングデコーダの C 言語ソースプログラム

	計算時間 [s]
べた計算	9.45
テーブル	4.34
ハード	3.13

表 1: 7bit ハミングデコーダの実験結果

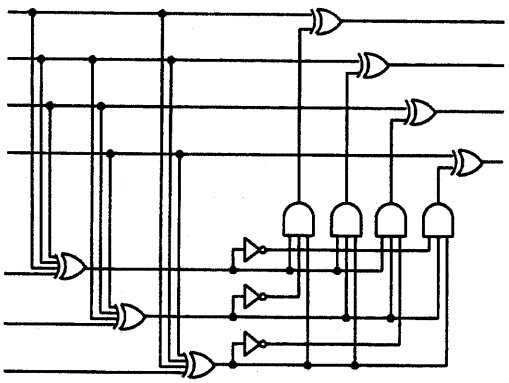

図 8: 7bit ハミングデコーダの回路

5 おわりに

C 言語からハードウェアへの変換法の一例を示し、実際にハードとソフトの協調動作をさせたうえで、この方法で高速化が可能なことを示した。データフローの技術を利用しているため、たまにしか起きない条件を頻繁に調べるプログラムに対して効果があることがわかった。また、演算のうち遅延の短いものが連続する場合はこれをまとめて 1 クロック内に収めることで、高速な回路を実現できることを示した。

今後は、C 言語をハードウェアに変換する場合の構造的な最適化技術をさらに追求することと、高速化しやすい、またはしにくいアルゴリズムの特徴について研究する予定である。

6 謝辞

本研究の推進にあたり、PARTHENON の利用においては (株)NTT および PARTHENON 研究会の皆様の御協力を頂いております。この場を借りて厚く御礼申し上げます。

7 参考文献

[1] 齊藤正伸, 多田好克, "ソフトウェアの部分的なハードウェア化による高速化技法の研究", 情報処理学会 第 36 回プログラミング ・シンポジウム報告集, pp. 127-134, Jan., 1995.

[2] 齊藤正伸, 多田好克, "計算資源としての FPGA とソフトウェアとのインタフェースに関する考察", 第 7 回パルテノン研究会資料集, pp. 27-36, Nov., 1995.

[3] 齊藤正伸, 多田好克, "高級言語から FPGA への部分コンパイルによる高速化技法の研究", 情報処理学会 第 48 回全国大会講演論文集 (6), pp. 117-118., 1994.

[4] P. M. Athan, and H. F. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis," IEEE computer, pp. 11-20, Mar., 1993.

[5] M.Wazlowskim L. Agawal, T. Lee, and et al, "PRISM-II Compiler and Architecture," Proceedings of the IEEE workshop on FCCM, pp. 9-16, 1993.

[6] D. E. Van Den Bout, J. N. Morris and et al. "AnyBoard: An FPGA-Based Reconfigurable System," IEEE Design & Test of Computers, pp. 21-30, Sep., 1992.

[7] R. W. Hartenstein, J. Becker and R. Kress, "Two-level Partitioning of Image Processing Algorithms for the Parallel Map-oriented Machine," IEEE Procesdings of Forth International Workshop on Hardwaer/Software Codesign, pp. 77-84, Mar., 1996.

[8] T. Yamauchi, S. Nakaya and N. Kajihara, "Compiler for An Adaptive Massively Parallel System," Proceeding of Real World Computing Symposium, pp. 479-486, 1997.

[9] 中村 行宏, 小栗 清, 野村 亮, "RTL 動作記述言語 SFL", 電子情報通信学会論文誌, Vol. J72-A, N0.10, pp. 1579-1593, Oct. 1989.