計算機アーキテクチャ 115－21
設 計 自 動 化 78－21
（1995. 12. 15）

ハードウェア／ソフトウェア協調設計システム

雪下 充輝† 名古屋 彰† 伊藤 康史‡ 木村 晋二‡

†NTT コミュニケーション科学研究所
〒619-02 京都府相楽郡精華町光台 2
{yuki,nagoya}@cslab.kecl.ntt.jp

‡奈良先端科学技術大学院大学
情報科学研究科
〒630-01 奈良県生駒市高山町 8916-5
{yasufu-i,kimura}@is.aist-nara.ac.jp

あらまし 本報告では、アプリケーションの高速処理をハードウェア／ソフトウェア協調動作システム上で実現することを目的として、その協調設計手法について提案する。ハードウェア／ソフトウェア協調設計では、ハードウェアとソフトウェアを最適に分離することが最も重要であり、今までにも種々の手法が提案されている。ここでは、C 言語で記述されたアプリケーションから、ハードウェア化による性能向上率を予測してハードウェアを分離合成する手法について述べる。さらに、合成結果を汎用コプロセッサ GPCP-SS を有するシステムで実際に協調動作させ、分離手法の有効性を論ずる。

Hardware/Software Co-design system

Mitsuteru Yukishita†, Akira Nagoya†, Yasufumi Itoh‡, Shinji Kimura‡

† NTT Communication Science Laboratories
2 Hikaridai, Seika, Sohraku-gun,
Kyoto, 619-02 JAPAN
{yuki,nagoya}@cslab.kecl.ntt.jp

‡ Graduate School of Information Science,
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara, 630-01 JAPAN
{yasufu-i,kimura}@is.aist-nara.ac.jp

Abstract In this paper, we present a hardware/software co-design method for the fast execution of application programs on a co-operation system with reconfigurable hardware. From functional descriptions written in C language, we propose a new hardware/software separation method using the improvement rate of performance by hardware execution. We descuss the effectiveness of our method by executing the synthesized results on a hardware/software co-operation system GPCP-SS.

1 はじめに

各種のマルチメディア処理に代表される応用分野の拡大により計算機性能向上への飽くなき要求は留まることを知らない。常に、最も効率的にアプリケーションを処理できるシステムが要求されており、これに応えるためには、従来は人手で行ってきたシステム設計をハードウェア／ソフトウェア協調設計として自動化する必要性が高まってきた。本報告では、協調設計の中でも重要な課題であるソフトウェアとハードウェアの分離について、ハードウェア化有効率等を用いることによる自動化手法を提案する。

2 ハードウェア／ソフトウェアの協調設計

ハードウェア／ソフトウェア協調設計(hardware / software co-design)とは、実現しようとしているシステムに対し、ハードウェア化が有効な部分と、ソフトウェア化が有効な部分とを考慮し、両者のバランスを取りながら同時に設計を進めていく手法である[1]。実現しようするシステムの目標としては性能最大化、ハードウェアコスト最小化、消費電力最小化等がある[2]。さて、実際のシステムの構成法を考えると、

(1) 特定用途向き集積化プロセッサ(ASIP) による実現(PEAS-I[3], ASIA[4] など)、

(2) コアプロセッサと特定用途向け回路を組み込んだ集積回路による実現、

(3) (2)の特定回路部分のみをコプロセッサという形で実現し汎用 CPU の外部に付加する方法、

等に分類される。

(1)は対象となる応用プログラムに特化して命令をスリム化していることから、ハードウェアコストと消費電力の点では有利であり対象のプログラムに限定すれば速度の点でも不利にはならない。また、(2)(3)はリソースが増加することからハードウェアコストと消費電力の点では 不利であるが、特定の応用プログラムについては速度の点で大幅な向上が見込まれる。

しかしながら、三者とも特定用途向けに特化したハードウェアを用いることから、異なった特性を持つ他のプログラムを走行させたいという要求に対応することは困難である。

一方、ハードウェアの実現法として、従来のカスタム LSI ではなく FPGA(Field Programmable Gate Array)を用いてより柔軟で低コストなハードウェア部を実現する方法が最近になって注目され研究が進められている(RM-III[5], Splash[6] , PRISM[7] , Rasa Board[8] など) 。

そこで、我々は FPGA を用いることにより上記(3)の構成法におけるコプロセッサを柔軟に実現できる GPCP-SS[9]を提案してきた。以下では、この汎用コプロセッサ GPCP-SS を有する協調動作システムを対象として、その協調設計手法を述べる。

3 ハードウェアとソフトウェアの分離

3.1 協調設計の全体フロー

本研究では、C 言語で記述されたアプリケーションの高速処理をハードウェア／ソフトウェア協調動作で実現することを目標とし、そのための自動協調設計手法を明らかにする。協調設計では、まずアプリケーションをハードウェアで実行する部分とソフトウェアで実行する部分とに分離する必要がある。これを自動的に行なうために、ハードウェア化に適した部分の抽出をハードウェア化有効率の見積りを用いて行なう。ここで、ハードウェア化有効率と呼んでいるのは、ハードウェア化した時の実行時間とソフトウェアでの実行時間の比率である。

まず C 言語による記述を適当なブロックに分けて、上記の見積りを行なう。つぎにハードウェア化の効果が最大と見積られる部分を実際にハードウェア化することとし、ハードウェア記述言語へ変換し論理合成を行なう。合成された論理回路を FPGA で構成された汎用コプロセッサ GPCP-SS にマッピングして、アプリケーションの高速処理を行なう。

上記の協調設計では、C 言語による記述を書き換えると、生成されるハードウェアも変化する。ハードウェア化有効率の見積りの正当性は最終的な GPCP-SS を用いた実行時間で評価することができる。以下、C 言語からハードウェア化までの手法に

ついて詳しく述べる。

3.2 Control / Data Flow グラフの生成

C 言語の記述を解析するため図 1 に示すような Control / Data Flow グラフを生成する。Data Flow グラフは 2 入力、1 演算子、1 出力の演算を表す節点からなる有向グラフであり、条件判定などは含まない。一方、Control Flow グラフは条件判定節点と Data Block 節点からなる有向グラフである。Data Block 節点は、そこで行なわれる演算(の列)に対応する Data Flow グラフ群と結合される。

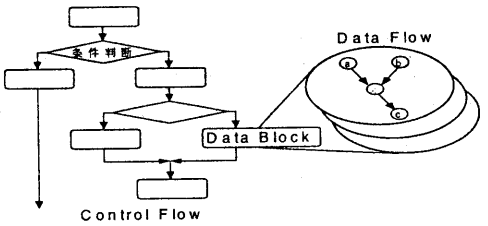

図 1　Control / Data flow

通常の C 言語の解析では、Control Flow グラフと Data Flow グラフとは各々独立に最適化されるが、ハードウェア化を考える場合には制御として記述されている部分をハードウェアの演算として実現することが有効である場合があり、Control Flow グラフの一部を Data Flow グラフへ変換したり、その逆の変換を行なうことを可能とすることとした。例えば、C 言語では条件により同じ変数へ異なる値を代入する場合 if 文を用いることが多いが、ハードウェアではセレクタを入れることで、データパスとして実現できる。このような制御とデータの相互変換のため、Control Flow グラフと Data Flow グラフを一元管理し、相互に最適化することを可能にする。

C 言語のソースからは、以下のようにして Control / Data Flow グラフを生成する。まず、変数の宣言についてはグラフの付加情報として変数名と属性の情報の表を作成する。これは、スコープルールなどを考慮して階層化する。つぎに、関数宣言については、関数名とそれが表す Control / Data Flow グラフの対応関係の表を作成する。

これらの C 言語の構文要素に関しては、基本的にテンプレートに従って処理する。代入文は Data Flow グラフへと変換する。if 文は、条件部分の演算を表す Data Flow グラフと条件判定節点へ変換する。for 文は、通常の手法で、条件判定とループからなるような Control Flow グラフへと変換する。

例として size 個の入力の XOR 演算を行なう関数を考える。C 言語による記述は図 2 の通りである。これをハードウェア化する場合、size が決まりさえすれば、XOR ゲートのバランスした木で実現できる。

```
int encoder(int size, int *indata)
{
  int temp = 0, i;
  for (i = 0; i < size; i++) {
    temp = temp ^ indata[i];
  }
  return(temp);
}
```

図 2　ループを持つ C 記述の例

3.3 ハードウェア化有効率の見積りとハードウェア部の分離

従来から、ソフトウェアのコンパイラもターゲットとなる CPU の命令を効率よく使用して高速に動作させるために様々な工夫をこらしてきた。これらのコンパイラはソースを静的に解析して得られた情報を用いて、高速化が期待できる部分を抽出するなどの処理を行っている。

本研究では、上述の Control / Data Flow グラフに基づき、以下のような評価パラメータを用いてハードウェア化の有効率の見積りを行い、ハードウェア化部分の決定に用いる。

まず、性能向上率を以下のように定義する。

$$\text{性能向上率}\delta = \frac{\text{ソフトウェア動作時の処理クロック数見積り}}{\text{ハードウェア化時の処理クロック見積り}}$$

ただし、ハードウェアの動作クロックと、ソフトウェア命令実行のクロックとは一致するとは限らないので、δの計算は正規化して行なう。また、汎用CPUとコプロセッサ間の通信によるオーバーヘッドが存在するが、その部分は別のパラメータで評価することとする。

次に、ハードウェア化した場合のコストをリソース量で評価する。

ハードウェア化リソース量Rh≒等価ゲート数

上記の評価は以下に示すように、まずData Blockのレベルで行い、次にボトムアップ的にControl Flowも含めた形で評価の単位を拡大しサブルーチンレベルまで評価を行い、最終的にサブルーチンレベルでハードウェアかソフトウェアかの判定を行う。

(1) Data Block レベルでの見積り

Data Block b_i をソフトウェアで処理する場合のクロック数 $Cl_S(b_i)$は、コンパイルされたマシンコードの実行マシンサイクルで求める。

一方、b_i をハードウェア化した場合の処理クロック数 $Cl_H(b_i)$では、各演算に必要と見込まれるクロック数の積算に加えて、変数の書き込み読み込み遅延も考慮する。具体的には、変数のライフタイム(書き込みが行われ次に読み込みが起こるまでの時間)を評価することにより、中間変数(V_T)か、レジスタ変数(V_R)か、メモリ変数(V_M)か、を判断する。後者2つの場合は書き込み読み込み遅延を考慮する。また、上記 V_T, V_R, V_M の総数、および、ライフタイム未確定情報をControl Flowレベルの処理まで保持する。

(2) Control Flow レベルでの見積り

1階層のifやswitch文の制御を受けるブロックのソフトウェア処理でのクロック数 $Cl_S(if\_block_j)$, $Cl_S(sw_block_j)$はそれぞれ内部に存在するData Block b_i の実行確率 $P(b_i)$、条件判断の平均クロック数 $Cl_S(condition_j)$、switchの分岐先数 m を用いて

$$Cl_S(if\_block_j) = Cl_S(if_j) + \sum_i Cl_S(b_i)\cdot P(b_i)$$
$$P(b_i) = \frac{1}{2}$$

$$Cl_S(sw\_block_j) = Cl_S(sw_j) + \sum_{i=1}^{m} Cl_S(b_i)\cdot P(case_i)$$
$$Cl_S(sw_j) = \frac{\sum_{i=1}^{m} Cl_S(case_i)}{m}\cdot\frac{(m+1)\cdot m}{2}\cdot P(case_i)$$
$$P(case_i) = \frac{1}{m}$$

と仮定する。

ループの場合は、ループ回数をN として、

$$Cl_S(loop\_block_j)=(Cl_S(if_j))+Cl_S(b_i))\cdot N$$

と表わし、これらをサブルーチンレベルまで積み上げて評価を行う。

一方、ハードウェア化した場合の Cl_H の評価を行う際には、Control Flowの解析では、Data Block内でライフタイムの確定できなかった部分を、複数のData BlockまでData Flowを追うことにより確定する。次に、変数種別によるアクセス遅延の違いを加味する。中間変数 V_T はクロック内だけ値を保持(後述のSFL[10]における端子に相当)、V_R, V_M は書き込むために1クロックが必要、V_M は読み込むためにもクロックが必要とする。そのため、Data Block内に変数 V_R, V_M が多数使用されるとハードウェア化するメリットが少なくなる。これらの変数の数で遅延の見積を行う。

$$Cl_H(b_i)=\text{演算クロック数}+\alpha V_R(b_i)+\beta V_M(b_i)$$

次に、Control Flowからハードウェアの特質である並列実行可能性をループ、switch文等でチェックする。可能並列度 $Par(condition_j)$を計算し、以下の式で各制御ブロックでのクロック数 Cl_H を見積り、最終的にサブルーチンレベルでのクロック数を見積もる。

$$Cl_H(sw\_block_j) = \frac{Cl_H(sw_j)}{Par(sw_j)} + \sum_{i=1}^{m} Cl_H(b_i)\cdot P(case_i)$$
$$Cl_H(sw_j) = \frac{\sum_{i=1}^{m} Cl_H(case_i)}{m}\cdot m\cdot P(case_i)$$
$$P(case_i) = \frac{1}{m}$$

$$Cl_H(loop\_block_j) = \frac{(Cl_H(if_j)+Cl_H(b_i))\cdot N}{Par(if)}$$

$$Cl_H(subroutine) = \sum_j (Cl_H(sw\_block_j) + \sum_j (Cl_H(loop\_block_j) + \sum\ldots$$

最終的にサブルーチンごとに、CPUとコプロセッサ間通信のオーバーヘッドを入出力変数の数 V_{IN}, V_{OUT} で見積もり、以下の式によって性能向上率を求め、ハードウェア化部分の抽出を行う。

$$\delta(subroutine) = \frac{Cl_S(subroutine)}{Cl_H(subroutine)+\gamma V_{IN}(subroutine)+\eta V_{OUT}(subroutine)}$$

(3) 性能向上率算定例

簡単な8ビットのデータのECCを計算する例[11]、ユークリッドの互除法を使用して5回の繰り返しで最大公約数を求める例[11]、8ビット16種の命令を有する簡単な2フェーズのノンパイプラインCPUの例、について性能向上率の見積り結果を表1に示す。

*:V_{IN}, V_{OUT} を考慮した値

	Cl_S	Cl_H	δ
ECC	1329	63*	21.1
GCD	65	28*	6.5
CPU	58	3	19.3

表1 性能向上率算定結果

3.4 ハードウェアモジュールのスケジューリングとアロケーション

基本的なアルゴリズムとして高位合成の研究で行われている、リソースに制約があるスケジューリング[12]、または、時間制約があるスケジューリング[12]等を参考にしたアルゴリズムを適用する。本システムでは、ターゲットが FPGA であり、処理の高速化を目的とすることから、リソースの制限は比較的緩い条件としている。

また、高位合成で行われているアルゴリズムに比較して、本システムでは、よりハードウェアを意識したスケジューリングを行う。具体的には、演算種別によるクロック数の違いの考慮、上述した変数種別による読み書きクロック数の考慮等である。

ところで、本システムではレジスタトランスファレベルのハードウェア記述言語 SFL[10]による記述を生成し、既存の論理合成システム PARTHENON[10]でハードウェアを合成する。ここでは、演算が1クロックで実行可能か、変数が記憶無しリソースである端子か、記憶有りのレジスタやメモリであるかの判定が重要になる。変数種別は性能向上率見積時に判定を行っているのでその結果を用いる。また、各種の演算については、予めライブラリの形で用意し、その情報として必要クロック数を与えておく。それらの情報と前段階の Control / Data Flow 解析時に得られた変数のライフタイム等を考慮し、与えられたリソースの許容重複度（加算器が n 個等）の範囲で、リソースのスケジューリングとアロケーションを進める。

4 協調設計例とその評価システム

本研究の現状では、サブルーチン単位での Control / Data Flow 生成、Control / Data Flow から変数のライフタイム算出による変数種判定部分が完成し Data Block レベルでのハードウェア化時の処理クロックが見積もれる段階にある。

以下、誤り訂正符号(ECC)の計算例[11] を用いて、協調設計の例を示す。ここで示す符号は、8 ビットの入力データに対し、1 ビットの冗長なデータビットを付加し、3×3 の行列と見て、縦、横のパリティをとり、さらに全体のパリティをとるものである。文献[11] では Hardware C の記述として図3 に示すような記述が与えられている。

図 3 で output_data[0], ..., output_data[7] は入力データのコピーであり、output_data[8] は冗長なデータビット、output_data[9], ...,output_data[15] が計算されたパリティである。output_data は整数の配列で、各整数は 0 または 1 の値をとる。なお、ビットのパッキングなどを行なおうとすると、ソフトウェアではかえって効率が悪くなる。

```
ENCODE
for (i = 0; i < 8; i++) {
  output_data[i] = (indata >> i) & 1;
}
output_data[8] = 0;
for i = 0 to 2 do
{
        output_data[i+9] = ENCODER (
                output_data[3*i] @
                output_data[3*i+1] @
                output_data[3*i+2]
              ) with (3);
        output_data[i+12] = ENCODER (
                output_data[i] @
                output_data[i+3] @
                output_data[i+6]
                ) with (3);
        output_data[15] = ENCODER (
                output_data[15] @
                output_data[3*i] @
                output_data[3*i+1] @
                output_data[3*i+2]
                )with (4);
}
ENCODER {
        temp = 0;
        for i = 0 to size-1 do
                temp = temp ^ indata[i];
        return_value = temp;
}
```

図 3　ECC の基本処理部

ハードウェア性能見積時には、まず、ENCODE というサブルーチン側で、for ループを解析する。内部で行なわれている演算が ビット間の xor で非常に簡単であることと、繰り返しの回数が固定であることから

temp = indata[0] xor indata[1] xor .. indata[size-1]

など同時に処理するハードウェア記述を生成することができる。

同様に、サブルーチンを呼び出す側でのループ内の処理も、Control / Data Flow グラフの解析により、生成される変数が独立であることを判断して、並列に動作可能なハードウェアを生成できる。変換結果の SFL 記述一部を図 4 に示す。

```
stage getdata {
   state_names  s_active_wait,s_active, s_ac
tive2;
   state_name   s_wait, s_wait2, s_wait3;
   first_state  s_active_wait;
        :
state s_wait3 par {
  out_data := (in_data<0> @ in_data<1> @
    in_data<2> @ in_data<3> @ in_data<4> @
    in_data<5> @ in_data<6> @ in_data<7> @
    in_data<8> ) ||
   (in_data<2> @ in_data<5> @ in_data<8> ) ||
   (in_data<1> @ in_data<4> @ in_data<7> ) ||
   (in_data<0> @ in_data<3> @ in_data<6> ) ||
   (in_data<6> @ in_data<7> @ in_data<8> ) ||
   (in_data<3> @ in_data<4> @ in_data<5> ) ||
   (in_data<0> @ in_data<1> @ in_data<2> )  ;
    generate putdata.run();
    goto s_active;
    }
  }
```

図 4 ECC の SFL 記述

実行時間の評価には、以下に述べる GPCP-SS を用いた。GPCP-SS は、4 個の FPGA により可変な回路を実現し、他に SUN WS との SBus インタフェース、1 メガバイトのキャッシュ／ローカルメモリとそのコントローラ、FPGA へ書き込む回路データを蓄積するコンフィギュレーションメモリとそのコントローラからなる。FPGA には ALTERA の EPF81188ARC240-2 を用い 1 チップあたり 12,000 ゲート相当の論理回路を実現できる。

ソフトウェアとのインタフェースに関しては、ソフトウェアからの SBus のアドレスを用いたレジスタ代入ができる他、ソフトウェアから渡されたアドレスをもとに、GPCP-SS から直接主記憶へアクセスすることもできる。

ECC 処理の実現では、indata の先頭アドレスをレジスタ代入で渡された状態で処理を開始し、indata を主記憶から読み込んで演算を行ない、さらに、結果を主記憶へ書き戻すまでのクロック数を計測した。クロック数自体も GPCP-SS の可変回路部でカウントし、カウンタの値を主記憶へ書き戻した。結果を表 2 に示する。

	Cl_S	GPCP-SS 実クロック	速度比
ECC	1329	46	28.9
GCD	65	49	1.3

表 2 GPCP-SS の実クロックと Cl_S との比較

このように、ハードウェア/ソフトウェア協調設計結果を GPCP-SS を用いて協調動作させることにより実クロック数の値で評価可能となる。

5 おわりに

C 言語レベルの機能記述から、ハードウェア化による性能向上率を予測してハードウェアを分離合成する手法について述べた。さらに、小規模の例について、合成結果を汎用コプロセッサ GPCP-SS で実際に協調動作させ、見積り手法の正当性を確認した。今後、システムの実装と共に、種々の例に適用して、パラメータの精練化を進める予定である。

参考文献

[1] Giovanni De Micheli.:Computer-Aided Hardware-Software Codesign, *IEEE Micro*, pp.10-16, (Aug. 1994)

[2] 今井正治.:ハードウェアの見積と生成, 情報処理学会誌, Vol.36, No.7, pp. 614-619, (July 1995)

[3] Jun Sato, Nobuyuki Hikichi, Akichika Shiomi and Masaharu Imai.:Effectiveness of a HW/SW Codesign System PEAS-I in the CPU Core Design, *APCHDL'94*, pp. 259-262 (1994)

[4] Ing-Jer Huang, Alvin Despain and Northwestern Univ., Bruce Holmer.:Generating Instruction Sets and Microarchitectures from Applications, *ICCAD'94*, pp.391-396 (1994)

[5] 沼昌宏,FPGA を利用したアーキテクチャとシステム設計, 情報処理学会誌, Vol.35, No.6, pp.551-518 (1994)

[6] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, and D Lopresti.:Building and Using a Highy Parallel Programmable Logic Array, *Computer*, Vol.24, No.3, pp.81-89 (Jan. 1991)

[7] P. Athanas and H. Silverman.:Processor Reconfiguration Though Instruction-Set Metamorphonsis, *Compuer*, Vol. 26, No.3, pp. 11-18,(1993)

[8] D.Thomas, J.Adams, and H. Schmit.:A Model and Methodology for Hardware-Software Codesign, *IEEE Design & Test of Computers*, Vol. 10, No.3, pp. 6-15, (1993)

[9] 伊藤康史,平尾誠,木村晋二,渡邊勝正.:汎用コプロセッサ GPCP-SS の実現と評価, 情処研報 95-DA-77, pp. 169-176, (Oct. 1995)

[10] 須山敬之,名古屋彰,小栗清,雪下充輝,関川浩: PARTHENON における最新の論理合成機能,信学技報 VLD93-103,ICD93-198, pp. 41-48 (Mar. 1993)

[11]Benchmarks for the 1991 High Level Synthesis Workshop (1991)

[12] Gajski, D.D., Dutt, N.D., Wu, A.C-H. and Lin, S. Y-L.: High-Level Synthethesis, Introduction to Chip and System Design, Kluwer Academic Publishers (1992)