社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2005－ＳＬＤＭ－119 (23)
2005／3／18

非線形方程式と整数解の探索に基づく高位合成向けビット長最適化

土井 伸洋† 堀山 貴史†† 中西 正樹††† 木村 晋二†

† 早稲田大学大学院 情報生産システム研究科
†† 京都大学大学院 情報学研究科
††† 奈良先端科学技術大学院大学 情報科学研究科

あらまし ハードウェア設計においては浮動小数点演算の固定小数点演算化が面積や速度の点から重要であるが, 変換においては演算誤差を考慮したビット長最適化が必要であることから, 人手による変換は困難であった. そこで我々はビット長最適化問題を非線形問題へ帰着させて解く自動化手法の研究を行なっている. 一般的な非線形計画法では解が実数となるため, ビット長最適化においては解の切上げが必要であった. そこで本稿では, 実数解の制約のもとで, 探索により整数解を求める方法を示す.

キーワード 高位合成, ビット長最適化, 非線形計画法

Bit-length Optimization Method for High-level Synthesis based on Non-linear Programming and Searching Integer Solutions

Nobuhiro DOI†, Takashi HORIYAMA††, Masaki NAKANISHI†††, and Shinji KIMURA†

† Graduate School of Information, Production and Systems, Waseda University
†† Graduate School of Informatics, Kyoto University
††† Graduate School of Information Science, Nara Institute of Science and Technology

Abstract This paper presents bit-length optimization technique for high-level synthesis based on non-linear programming and searching integer solutions. The results of the bit-length optimization based on non-linear programming are real values, and these values are converted to integer with round-up for hardware implementation. In this paper, we show a method to search integer solutions under the constraints of the real solution. The experimental results shows the advantage of searching based method.

Key words High-level synthesis, Bit-length optimization, Non-linear programming

1. はじめに

回路の微細化技術が進歩すると同時に, 設計される回路の規模も増加の一途をたどっている. 回路設計では, ハードウェア記述言語 (HDL) を用い, レジスタトランスファーレベル (RTL) で回路を記述することが一般的であるが, 設計規模のさらなる増加と設計期間の短縮要求から RTL 記述を用いた方法では設計が不可能な状況になりつつある. このために, C 言語などのより抽象度の高い言語と高位合成系を用いた設計手法が急速に実用化されつつある. [1] しかし, 高位合成系を用いた設計手法にも以下に示すボトルネックが存在する.

高位合成おいてはコンパイラやハードウェア向け最適化技術が欠かせない. とくにレジスタやデータパスのビット長最適化は, 動作速度や面積に大きく影響するために非常に重要である. 現在使われている高位合成系では, 設計者がビット幅を指定する場合が多いが, この作業は設計上大変な負担となる上に, 長すぎたり短すぎたりといった指定誤りの可能性がある. さらに, 実数演算を含むアルゴリズムをハードウェアで実現する場合には, 誤差を考慮しつつ演算器の固定小数点演算化を行なう必要があり, より大きな負担となっている. 設計期間の半分以上が, この実数演算の固定小数点演算化に費やされているという報告もある [4].

これを解決するため, ビット長を自動的に推定する手法が数多く提案されている. [2] は C 言語を入力として整数変数や演算器のビット長を自動的に推定し, RTL 記述を生成するシステムについて述べている. 実数演算を含む場合は, 固定小数点演算化による誤差の正確な評価を行なう必要があり, さまざまなアプローチが存在する. FRIDGE Project [5] や Sung らの研究 [6] では固定小数点化のためのフレームワークとしてビット長を指定してシミュレーションを行なえる環境が提案されている. し

かし, 指定したビット長が目標としている精度を実現できているかをシミュレーションを用いて検証するため, 大規模な入力パターンを用意しなければならず, またシミュレーションにかかる時間も無視できない.

シミュレーションベースではなく解析的手法を用いた方法としては [7] などがある. ここでは C プログラム中の浮動小数点演算について誤差のモデル化を行ない, 誤差を解析的に見積もる. そして, この誤差モデルを利用し, ビット長最適化問題を非線形問題に帰着させ解いている. 非線形計画法を用いて得られる解は多くの場合実数であるため, 得られた解をそれぞれ切り上げることでビット長となるべき整数解を得ていた. 切り上げは解の正当性を保証するが, 切捨てた場合でも全体として制約を満たす場合があり, 全体をとおして最適とはいえなかった. そこで本稿では切り上げではなく整数解を探索することで最適なビット長を求める手法を提案する. また本手法を実現し, その結果について考察を行なう.

2. 固定小数点演算

動画や音声を処理するプログラムには多くの場合実数演算が必要であり, プログラム中では浮動小数点演算として記述される. だが浮動小数点演算器は動作速度やコストの観点から使用できない場合も多い. そのため, ハードウェア設計においては実数演算を固定小数点演算に変換し実装する場合が多い. 一般的な固定小数点フォーマットは以下に示す 3 つのパラメータで表される.

$sign$	符合ビット (2 の補数表現)
wl	語全体のビット長
iwl	整数部分のビット長

また, 小数部分のビット数は fwl ($fwl = wl - iwl$) と表されることが多い (図 1). 小数点の位置は仮想的に決められたものであるため, 演算には語長の等しい整数演算回路を用いることができる.

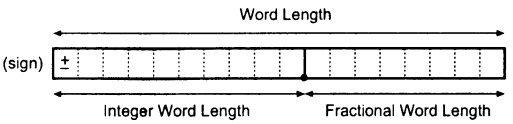

図 1 固定小数点の表現形式

ある固定小数点数 $\langle sign_1, wl_1, iwl_1 \rangle$ を別の固定小数点数 $\langle sign_2, wl_2, iwl_2 \rangle$ に変換する時に, 小数部を適切なビット長に丸めねばならないこともある. 本稿では丸めの方式として '切捨て' を標準とした. ハードウエアを設計したときに最もコストがかからない方式であり, もっとも広く使われているからである.

3. コンパイラの概要

図 2 はコンパイラの全体像である. 入力言語は SystemC 言語を選択した. SystemC 言語は C 言語の拡張言語であり, ビット長やインタフェースを記述するための型が定義されている.

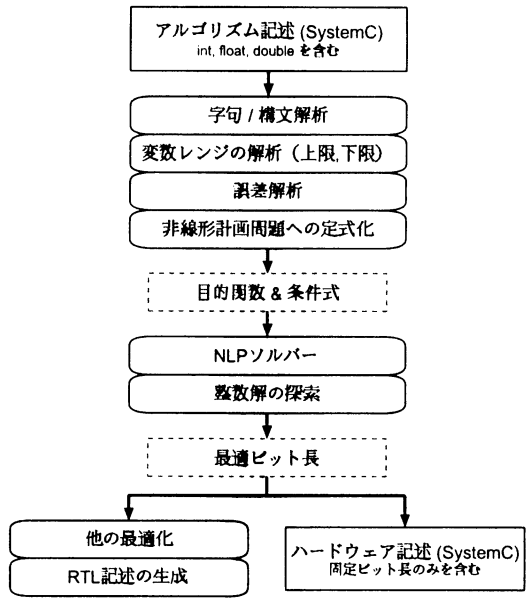

図 2 SystemC 言語からのハードウエア生成

固定小数点型についても sc_fixed 等の型を使うことで記述でき, 問題となるプログラムを記述するのに適している. コンパイラは入力として char や int にくわえ float や double といった型を使って記述されたプログラム記述を受け取り. 字句や構文の解析の後, 処理内容を表現するコントロール/データフローグラフを生成する. 次に, 得られたコントロール/データフローグラフを元にハードウェア向けの最適化を行なう. ビット長最適化処理はこの一部であり, 入出力の仕様 (例えば出力の精度) を満たすのに必要なレジスタや演算器のビット長を自動的に計算する. ここではビット数が明確に指定されなかった変数 (int, long) や実数変数 (float, double) を対象に解析を行なう. そして, 解析の結果得られた条件式を非線形計画法を用いて解き, さらに切り上げや探索といった後処理をすることでビット長を得る. 最終的に得られたコントロール/データフローグラフと変数や演算器のビット長に関する情報をもとに HDL 記述を生成する.

4. ビット長の最適化

ここでは浮動小数点演算の固定小数点演算化に伴う誤差を解析的に見積り, 最適なビット長を得る手法を示す. この手法ではプログラム解析を通じて見積もられた誤差と入出力ポートの定義より変数ビット長をパラメータとした条件式を導き出し, これを非線形方程式によって解くことで, 最適ビット長を得るものである. ここではすべての入力パターンに対し精度を保証することを解の必要条件とする.

説明にあたっては次の表色系変換を例とする.

$$Cr = -0.1684 * red - 0.3316 * green + 0.5 * blue$$

これを SystemC 言語で記述したものを図 3 に示す. このプログラムは red,green,blue の 3 つを入力として受け取り, 答とし

```
Header file
SC_MODULE(Cr){
  sc_in<sc_uint<8>> red,green,blue;
  sc_out<sc_ufixed<9,8>> Cr;
  double tmp0,tmp1,tmp2,tmp3;

  void Conv();
  SC_CTOR(C){
    SC_METHOD(Conv);
    sensitive <<red<<green<<blue;
  }
};
```

```
Source file
Conv(){
  tmp0 = 0.1684 * red.read();
  tmp1 = 0.3316 * green.read();
  tmp2 = 0.5    * blue.read();
  tmp3 = tmp0 + tmp1;
  Cr = tmp2 - tmp3;
}
```

図 3 ソースプログラム

て Cr を出力する. また, 演算のために 4 つの中間変数が定義されている. プログラム中には double 型として宣言された 3 つの実数定数が存在する. ハードウェア化のためには, Cr の精度を保証しながらできるだけ少ないビット長で中間変数および実数変数を表現する必要がある.

この変換式の実現には tmp0 等の中間変数をワイヤーとして実装する方法, 加算器を共有する方法, 演算順序を工夫する方法などさまざまな要素が関連してくる. しかし, 扱う問題が多きくなり過ぎるため, 本稿では与えられたプログラムの構成を変えずに, 最適ビットを推測するものとした.

コンパイラがこのプログラムを受け取った場合, 次の入出力仕様があるものとして解釈する.

- 変数 red,green,blue は 0-255 の整数.
- 出力 Cr は 0-255 の実数値をとり, その誤差は真値から ±0.5.

本章では, まず誤差見積もりの基礎となる誤差モデルと変数値の表現方法について説明する. 次に, 非線形計画法を利用した最適化について述べ, 得られた解に対する後処理について議論を進める.

4.1 誤差のモデル

変数に含まれる誤差は, 丸め誤差と伝搬誤差とでモデル化される. 演算の入力に含まれる誤差が演算結果に与える影響が誤差伝搬, それとは別に定数や演算結果を格納する変数のビット長が制限されることによって生成される誤差が丸め誤差である.

4.1.1 丸め誤差

丸め誤差は小数部を固定長で打ち切ったことにより発生する誤差である. 打ち切られた部分は先に述べたように '切捨て' によって処理する. ある固定小数点数 X の小数部ビット長を L_X と表すと, 丸め誤差 $E_R(X)$ は $0 \leq E_R(X) < 2^{-L_X}$ の範囲にあると考えられる. よって,

$$E_R(X) \quad = \quad [\ 0\ ,\ 2^{-L_X}\ ]$$

なる区間として表すことができる. ここで, 0 は負方向で誤差が最も大きくなる場合, 2^{-L_X} は正方向で誤差が最も大きくなる場合である.

4.1.2 伝搬誤差

伝搬誤差は, 誤差が演算により伝搬されることで発生する. 演算をすべて二項演算とし, 代入先のデスティネーション $X_{\rm dst}$, 演算の入力であるソースオペランド $X_{\rm src1}$, $X_{\rm src2}$, 演算の種類 $\circ$ で

$$X_{\rm dst} \leftarrow X_{\rm src1} \circ X_{\rm src2}$$

と表す. このとき $X_{\rm dst}$ への伝搬誤差 $E_P(X_{\rm dst})$ は, ソースオペランドの持つ誤差 $\Delta X_{\rm src1}$ および $\Delta X_{\rm src2}$ および演算の種類に応じて計算することができる. 例えば加算であれば

$$E_P(X_{\rm dst}) \quad = \quad \Delta X_{\rm src1} + \Delta X_{\rm src2}$$

となる. ここで, $X_{\rm src1}, X_{\rm src2}, X_{\rm dst}$ および $\Delta X_{\rm src1}, \Delta X_{\rm src2}, \Delta X_{\rm dst}$ はすべて区間であり, 伝搬式の演算はすべて区間に対する演算である.

丸め誤差と伝搬誤差の両方を含んだ $X_{\rm dst}$ の誤差 $\Delta X_{\rm dst}$ は

$$\begin{aligned}\Delta X_{\rm dst} &= E_R(X_{\rm dst}) + E_P(X_{\rm dst}) \\ &= E_R(X_{\rm dst}) \\ &\quad + Propagate(X_{\rm src1}, \Delta X_{\rm src1}, X_{\rm src2}, \Delta X_{\rm src2}, \circ)\end{aligned}$$

と表せる. $Propagate()$ は演算 $\circ$ に応じた伝搬誤差の関数を表し, 表 1 で定義される.

4.2 変数レンジの解析

誤差見積もりのためには, 誤差をモデル化するだけではなく, プログラム中の演算や制御を追跡し, おのおのの変数が取り得る値を求めることが必要である. しかし, これを集合として表現することは現実的ではない. そこで, 各変数のとりうる値を, その上界から下界までの区間として表現する区間演算 [2], [8] の手法を利用した.

区間演算においてはある変数 X の取り得る値を [X.min, X.max] のように表現する. ここで, X.min は変数 X の下界, X.max は上界である. また, 上下界を用いて表現された変数同士の演算についても区間演算を用いて定義することが可能である. 例題プログラムにおける tmp0 の計算は次のように表すことができる.

0.1684 = [0.1684 , 0.1684] (定数)
red = [0 , 255]
より
tmp0 = [0 , 0.1684 · 255]

プログラム中には単純な演算だけではなく, 分岐やループといった構造をもつ. 本手法では, 分岐構造に対しては if パートと else パートの両方について解析を行ない最悪ケースを採用する. またループ構造は展開し通常のコード列にすることで対応している.

表 1　伝搬誤差の見積り式 (Propagate)

演算	伝搬誤差の大きさ
$X_{\mathrm{src1}} + X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = \Delta X_{\mathrm{src1}} + \Delta X_{\mathrm{src2}}$
$X_{\mathrm{src1}} - X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = \Delta X_{\mathrm{src1}} - \Delta X_{\mathrm{src2}}$
$X_{\mathrm{src1}} \times X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = X_{\mathrm{src1}} \cdot \Delta X_{\mathrm{src2}} + X_{\mathrm{src2}} \cdot \Delta X_{\mathrm{src1}} + \Delta X_{\mathrm{src1}} \cdot \Delta X_{\mathrm{src2}}$
$X_{\mathrm{src1}} \div X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = \dfrac{\Delta X_{\mathrm{src1}}}{X_{\mathrm{src2}}}$ $\left(\text{除数が 0 をとりうる場合は } \begin{matrix} X_{\mathrm{src2.max}} = 2^{-L_{\mathrm{src2}}} \\ X_{\mathrm{src2.min}} = -2^{-L_{\mathrm{src2}}} \end{matrix}\right)$

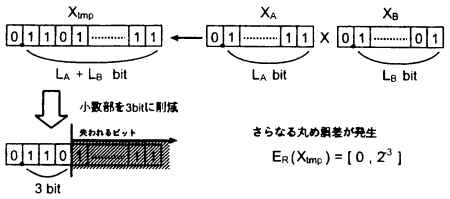

図 4　中間変数の丸め

4.3 誤差解析

入力変数や定数で発生した誤差は演算を繰り返すうちに変化し, 出力へと伝搬されてゆく. この伝搬されてゆく誤差の大きさを, 変数値の追跡によって得られた各変数の値を利用して解析的に導く. 伝搬においては, 常に誤差の最大値を伝搬することで, 最悪ケースを追跡する.

4.3.1 中間変数の丸め

固定小数点演算において乗算などの演算を繰り返すと, 小数部のビット長はしだいに長くなってしまう. そこで我々のアルゴリズムでは, 小数部のビット数が長くなり過ぎないように中間変数の丸めを行なう. これによりさらなる丸め誤差が発生してしまうが, レジスタやデータパスを削減するには非常に有効な戦略である.

図 4 に例を示す. 乗算 ($X_{\mathrm{tmp}} \leftarrow X_A * X_B$) を行なった直後では X_{tmp} のビット長は $L_A + L_B$ と長くなっている. 例ではこのビット数を減らすために 4 ビット目以下を打ち切っている. ただしこの打ち切りにより最大で 2^{-3} だけの丸め誤差が新たに発生する.

加減算では小数部ビット長が乗算のように増加することはないので, この戦略は適用されない.

4.4 非線形計画問題への定式化

4.2 節で述べた変数値の追跡と 4.3 節で述べた誤差の伝搬の解析作業が終了すると, 出力に含まれる誤差のとりうる範囲を $2^{-L_i}(i = 0, 1, \cdots, m-1)$ の多項式として得ることができる. (ただし L_i は変数の小数部ビット長). そこでこれを非線形計画問題として定式化し最適化することを考える.

今, 設計しているモジュールの出力を $O_j(j = 0, 1, \cdots, n-1)$, そして出力にそれぞれ付けられている精度に関する制約を $\mathrm{Lim}(O_j)$ とする. これは 3 章で述べたモジュールの入出力仕様によって与えられ, 「出力 O_j の誤差は $\mathrm{Lim}(O_j)$ 未満に収めなければならない」ということを規定している. 解析の結果, 出力に含まれる誤差は 2^{-L_i} の多項式として

$$\Delta O_0 = F_0(2^{-L_0}, 2^{-L_1}, \cdots, 2^{-L_{m-1}}),$$
$$\Delta O_1 = F_1(2^{-L_0}, 2^{-L_1}, \cdots, 2^{-L_{m-1}}),$$
$$\cdots$$

のように見積もられる. 問題では出力に関して精度の制約が与えられており, これを満たす必要がある. 故に出力に含まれる誤差は $[-\mathrm{Lim}(O_j), \mathrm{Lim}(O_j)]$ の範囲に収まらなければならない. ここで出力の誤差 $\Delta O_j(j = 0, 1, \cdots n-1)$ について, 正領域で誤差が最も大きくなる場合を $\Delta^+ O_j$, 負領域で最も大きくなる場合を $\Delta^- O_j$ とすると, 次の条件が成り立つ.

$$\mathrm{Lim}(O_0) > \Delta^+ O_0,$$
$$-\mathrm{Lim}(O_0) < \Delta^- O_0,$$
$$\mathrm{Lim}(O_1) > \Delta^+ O_1,$$
$$-\mathrm{Lim}(O_1) < \Delta^- O_1,$$
$$\cdots$$

これは非線形問題における制約条件と考えることができる. らにハードウエアの面積をできるだけ少なくしたいという観点から目的関数を次のように定めることができる.

$$\text{Minimize} \sum_{i=0}^{m-1} L_i$$

上記の問題を非線形計画問題の一般的なソルバーを用いて解くことで $L_i(i = 0, 1, \cdots, m-1)$ を得ることができる. 本稿では, 逐次 2 次計画法 (SQP 法) を用いた. これは, 目的関数と制約条件をそれぞれ偏微分して得られる傾きを利用して, 目的関数を最小化するラメータを求める手法である [9].

例題の場合次のようになる. 定数の丸め誤差はすべて区間として表現されるので

$$\Delta X_{0.1684} = E_R(X_{0.1684}) = [\,0\,,\, 2^{-L_0}],$$
$$\Delta X_{0.3316} = E_R(X_{0.3316}) = [\,0\,,\, 2^{-L_1}],$$
$$\Delta X_{0.5} = E_R(X_{0.5}) = [\,0\,,\, 2^{-L_2}].$$

定数 0.5 については固定小数点化しても誤差が発生しないので, $L_2 = 1$ であり誤差の範囲は [0 , 0] と置き換えることができる. tmp0,tmp1,tmp2 については, 表 1 より

$$\Delta X_{\mathrm{tmp0}} = [0, 255] \cdot \Delta X_{0.1684} + [\,0\,,\, 2^{-L_3}],$$
$$\Delta X_{\mathrm{tmp1}} = [0, 255] \cdot \Delta X_{0.3316} + [\,0\,,\, 2^{-L_4}],$$
$$\Delta X_{\mathrm{tmp2}} = [0, 255] \cdot \Delta X_{0.5} + [\,0\,,\, 2^{-L_5}].$$

tmp3 の誤差がとりうる区間については tmp0 と tmp1 の区間

表 2　非線形計画法によって得られた解

L_0	L_1	L_2	L_3	L_4	L_5	L_6
10.99	10.99	1	3.00	3.00	3.00	3.00

表 3　探索によって発見された解

L_0	L_1	L_2	L_3	L_4	L_5	L_6
11	11	1	3	3	3	3

から

$$\Delta X_{\mathrm{tmp3}} = [\,0\,,\ 255\cdot(\,2^{-L_0}+\,2^{-L_1})+2^{-L_3}+2^{-L_4}]$$

Cr は tmp2 と tmp3 の減算で求められる．この演算を経て正方向に最も誤差が大きくなる場合は，tmp2 と tmp3 の区間から

$$\Delta^{+}Cr = 2^{-L_5}, \qquad (1)$$

であると考えられる．同様に，負方向への誤差は次のようになる．

$$\Delta^{-}Cr = -255\cdot(2^{-L_0}+2^{-L_1})-2^{-L_3}-2^{-L_4}. \qquad (2)$$

故に，入出力仕様をもとに定められる $\Delta^{+}Cr$, $\Delta^{-}Cr$ の条件は

$$0.5 > \Delta^{+}Cr,$$
$$-0.5 < \Delta^{-}Cr,$$

と記述できる．このモデルから得られた各変数の小数部ビット長を表 2 に示す．

4.5　整数解の探索

非線形計画法によって得られる解は実数であるので，状況によっては後処理によって整数解を求める必要がある．これまではすべての数に対して切り上げることで整数化を行なってきた．本稿においては探索による整数解の算出について述べる．

ここで実数解を L_i とし，関連する整数解を L_i' とする．それぞれの変数について $L_i < L_i'$ $(i = 0, 1, ...)$ である時，制約が満たされ，正しい解である．よって切り上げによって得られた解は常に正しい．

以上を前提としてまず例題について考える．表 2 の結果を見ると，0.1684 および 0.3316 に相当する L_0, L_1 を除き，整数解へと収束している．また L_0, L_1 についても，それぞれ 11 が整数解であることが直観的にわかる．整数解を求める方法として切上げを採用した場合も同様に $L_0 = 11, L_1 = 11$ となる．そのためこの時の総ビット長は 35bit となる．だが例題においては $L_0 = 10, L_1 = 11$ としたり $L_0 = 10, L_1 = 10$ とする可能性もありうるので，これらについて探索を行なう必要がある．

例の場合は実数解である $\{L_0, L_1\}$ についてそれぞれ $\{\{\lfloor L_0\rfloor, \lfloor L_1\rfloor\}, \{\lfloor L_0\rfloor, \lceil L_1\rceil\}\{\lceil L_0\rceil, \lfloor L_1\rfloor\}\{\lceil L_0\rceil, \lceil L_1\rceil\}\}$ の場合をチェックする．これによりよりよい解を発見できる可能性がある．例題の場合，表 3 となる．

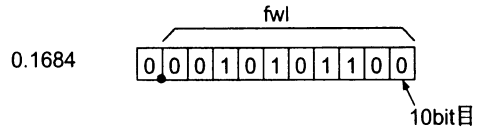

図 5　末尾のビットが 0 の定数

表 4　定数の性質を使い最適化を行なった解

L_0	L_1	L_2	L_3	L_4	L_5	L_6
10	10	1	3	3	3	3

4.6　定数の取り扱いについて

小数部のビット長がわからない状態では，$\Delta X_{0.1684} = [0, 2^{-L_0}]$ なる範囲として誤差を見積もっていた．しかし数によっては，誤差が 2^{-L_0} とならない場合もある．4.4 節で述べたように，定数"0.5"は，固定小数点化されても誤差 0 である．また 0.1684 という数値を小数部 10bit の固定小数点数で表した場合，その誤差は 0.0004312 程度つまり $0.0004312 < 2^{-11} < 2^{-10}$ ということになる．これは 0.1684 を二進数で表した場合，10bit 目が 0 となっているためである (図 5)．このような場合には，非線形計画法によって得られた解を下回っても，依然として精度が保証される場合もある．これを考慮することで，例題に対して表 4 に示す解がえられる．

変数の数が大きくなってきたり，制約条件が複雑である場合には，再度非線形計画法に持ち込む方法も考えられる．実数解のうちいくつかを整数解にまるめておき，その値を非線形計画問題の定数として参照するのである．例題の場合，$\{L_0, L_1\} = \{10, 10\}$ とし，条件式中 (式 (2)) でこれを参照する．

5.　実験および評価

入出力仕様で指定された条件を満たしつつ，変数ビット長を最適化するプログラムを C++言語，SystemC ライブラリ [10]，ANTLR パーサーライブラリ [11] を使い実装した．そして，例題で使用した表色系変換の他に FIR フィルタへ適用し，その評価を行なった．

FIR フィルタは音声処理などに頻繁に使用されており，ハードウェア化の要求も高いため例題としては適当である．対象とした FIR フィルタの仕様を以下に記す．この問題においては 11 個の係数を格納する配列を何 bit とするかが鍵となる．

- 段数は 11 段
- 入力は 16bit の整数 $(-2^{-15} \sim 2^{15})$
- 出力は 16bit の実数．許容誤差は ± 1.0

ここで，手作業による最適化においては，シミュレーションのための入力パターンを 100 万通りに制限した．これは全数検査が困難なためである．

実験結果は表 5 のようになった．係数を配列として宣言しているため，これを短くすることはできないが，「係数は 18bit」という情報を使うことで踏み込んだ解析ができる．そのため，係数以外の部分のビット長を減らすことができる．

表 5 FIR フィルタに対する最適化結果

	係数
切り上げ	18 bit
探索	18 bit
手作業	18 bit

	合計
切り上げ	206 bit
探索	204 bit
手作業	204 bit

6. まとめ

本稿では浮動小数点演算を含んだ SystemC 言語からハードウエアを自動生成する際に重要である浮動小数点演算の固定小数点演算化について, 出力精度を保証しながら必要なビット長を見積もる手法を提案した. 本手法では誤差を解析的に見積もり, 得られた解析結果を利用してビット長最適化問題を非線形問題として定式化した. そして, 非線形問題を解くことによって得られた解をもとに, 探索を併用し最適化ビット長を求めた. 結果, 実数定数が含まれる場合はそのビット長を束縛してやることで, 総ビット長を改善できることがわかった.

今後の課題として, より複雑で大規模なプログラム構造に対しても適用できるかを十分に検証することが必要である. また, 演算器の面積, 共有やスケジューリングを考慮する必要がある. 面積の大きい演算のビット長は 1bit の削減でも大きな効果が得られる. 演算器の共有を考慮すると演算の順序, 並列化, 動作速度などさまざまな要素を同時に考えねばならず, 問題のより詳細なモデル化が必要であると考えている.

謝辞

本研究を進めるにあたり日頃から有益なご助言, ご指導を頂いた早稲田大学大学院情報生産システム研究科の皆様に深く感謝します. また, 本研究は一部日本学術振興会科学研究費補助金, NEC, 文部科学省北九州知的クラスタープロジェクト研究費補助金による.

文献

[1] Kazutoshi WAKABAYASHI. "C-Based Synthesis Experiences with a Behavior Synthesizer "Cyber"". In *DATA '99*, pages 390–393, January 1999.

[2] Osamu Ogawa, Kazuyoshi Takagi, Yasufumi Itoh, Shinji Kimura, and Katsumasa Watanabe. "Hardware Synthesis from C Programs with Estimation of Bit Length of Variables". *IEICE Transaction*, E82-A(11):2338–2346, November 1999.

[3] D. Gordberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic". *ACM Computing Surveys*, 23(1):5–48, March 1991.

[4] Holger Keding and Markus Willems and Martin Coors and Heinrich Meyr. " FRIDGE: A Fixed-Point Design and Simulation Environment". In *Design Automation and Test in Europe*, pages 429–435, February 1998.

[5] Markus Willems, Volker Bursgens, Holger Keding, Thorsten Grotker, and Heinrich Meyer. "System Level Fixed-Point Design Based on an Interpolative Approach". In *Design Automation Conference*, pages 293–298, November 1997.

[6] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. "Fixed-Point optimization Utility for C and C++ Based Digital Signal Processing Programs". *IEEE Transaction on Circuits and Systems II*, 45(11):1455–1464, November 1998.

[7] 土井伸洋, 堀山貴史, 中西正樹, and 木村晋二. "浮動小数点演での誤差の増減を考慮した変数ビット長の最適化". DA シンポジウム 2004 論文集, pp.85–90, 2004 年 7 月.

[8] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. "Bitwidth Analysis with Application to Silicon Compilation". In *SIGPLAN'00 Conference*, pages 108–120, 2000.

[9] DONLP2 http://plato.la.asu.edu/donlp2.html

[10] The Open SystemC Initiative. http://www.systemc.org.

[11] ANTLR Parser Generator and Transrator. http://www.antlr.o