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

2007－ＳＬＤＭ－129／2007－ＥＭＢ－4（18）
2007／3／16

C言語プログラムにおけるループ最適化に対するループ展開を伴わない等価性検証手法

松本 剛史† 瀬戸 謙修†† 藤田 昌宏††

† 東京大学大学院工学系研究科電子工学専攻 〒113-8656 東京都文京区本郷 7-3-1
†† 東京大学大規模集積システム設計教育研究センター 〒113-0032 東京都文京区弥生 2-11-16
E-mail: matsumoto@cad.t.u-tokyo.ac.jp, seto@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

あらまし 組込みシステムのように設計制約の厳しいシステムの設計では、設計者の手による高度なループ最適化は重要な技術の一つである。本稿では、ループ最適化を対象とした等価性検証手法を提案する。従来の記号シミュレーションによる検証では、ループを展開するため、等価性は展開された回数までしか証明されず、また、展開回数が多い場合に検証時間が長期化するという問題点があった。そこで、提案手法では、比較するべき出力変数の記号式を記号シミュレーションによって求める前に、あらかじめプログラムを解析し、必要な文とそれが実行されるときのイテレータの記号値を求める。記号シミュレーションを選択された文に対してのみ行うことにより、ループ展開を行うことなく、等価性検証を行う。実際のループ最適化に対する実験を通して、提案手法が十分短い検証時間で、等価性を検証することができることを示す。

キーワード 等価性検証, ループ最適化, データ依存グラフ, 記号シミュレーション

Equivalence Checking of Loop Optimizations in C Programs without Loop Unrolling

Takeshi MATSUMOTO†, Kenshu SETO††, and Masahiro FUJITA††

† Dept. of Electronics Engineering, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113-8656 Japan
†† VLSI Design and Education Center, University of Tokyo
2-11-16 Yayoi, Bunkyo-ku, Tokyo, 113-0032 Japan
E-mail: matsumoto@cad.t.u-tokyo.ac.jp, seto@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

Abstract In designing systems such as embedded systems, where many restrictions must be satisfied, loop optimization plays an improtant role. In this paper, an equivalence checking of loop optimization is proposed. In the previous methods based on symbolic simulation, loops must be unrolled before verification. This results in that the equivalence is proved just for the unrolled programs and verification takes much time when the number of unrolling is large. In the proposed method, first, statements and the corresponding symbolic values of the iterators that are required to compute the output variables are identified. Then, symbolic simulation is applied only to the statements. Therefore, the equivalence can be checked without unrolling loops. The experimental results show that the proposed method can verify the real loop optimizations in short time.

Key words Equivalence checking, Loop optimization, Data dependence graph, Symbolic simulation

1. はじめに

厳しい設計制約の中で、与えられた動作性能を満たすことが求められる組込みシステムや SoC (System-on-a-Chip) の設計においては、性能向上のための様々な最適化が行われる。そのような最適化の中には、高位合成ツールやコンパイラによって自動的に実現することができるものも多い。しかし、複雑な最適化は人手で行われることも多く、また、それが製品の性能向上に大きく寄与している場合も少なくない。

そのような最適化技術の一つに、システムの動作設計におけ

るループ最適化が挙げられる。ハードウェアとソフトウェアの双方において、ループはシステム全体の性能を決定する部分である場合が多いため、設計者による高度な最適化が施される。そのような最適化は、例えば以下の目的を持つ。

- 配列変数へのアクセス回数を減らす
- 異なる繰り返し実行間の依存関係を取り除く
- 複数のループをまとめる

第1点については、ハードウェア実装におけるメモリアクセスの減少となり、実行時間の短縮が可能である。第2点は、複雑な依存関係を取り除くことによって、コンパイラで適用可能な最適化 (コード移動、共通部分式の抽出、投機実行、など) の候補の発見が可能になるという利点がある。また、第3点は、まとめられたループ内部に対する最適化や複数のループにまたがる並列化が可能となる。これらの最適化は、単純な場合を除いて、人手によって行われるため、その前後で動作の等価性を検証し、最適化の過程で設計誤りが発生していないことを確認することが重要である。

このようなループ最適化は、システムの動作を決定するシステムレベル設計において行われることが多く、設計記述はC言語やCベース設計言語[7],[8]が用いられることが多い。そこで本稿では、C言語プログラムにおけるループ最適化を対象とした等価性検証手法を提案する。また、最適化を必要とするループでは、信号処理におけるフィルタや算術変換のように配列間の演算を主に行うものが多い。そのため、提案手法では、そのような配列間の演算を主に行うループの最適化を対象としている。

文献[2],[3]で提案されている記号シミュレーションに基づく等価性検証手法では、設計を表す記号式を生成し、それを解析することで等価性を証明するため、[6]などのツールを用いることによって、複雑な式の等価性も証明できる。しかし、ループを検証する場合には、あらかじめ展開する必要があるため、展開した回数までしか等価性が保証されず、また、展開回数が多い場合には、検証時間が増大するという問題点がある。一方、既存のループ展開を伴わない等価性検証手法では、次節で述べる通り、扱える範囲に制約があり、複雑な式の等価性が証明できない、という問題があった。これらの問題点を解決するために、提案手法では、

- 全ての出力の記号式を生成するために必要な文 (statement) と対応する繰り返し回数を特定する。出力が配列の場合には、任意のインデックスに対する記号式が生成できるように文を選ぶ
- その選択された文と繰り返し回数に対してのみ記号シミュレーションを行う

ことによって、既存のループ展開を行わない検証手法に比べて、制約のより少ないループ記述を扱え、より複雑な式の等価性を検証できる。

本稿の構成は、以下のようである。第2節で、既存のループ最適化に対する等価性検証手法と記号シミュレーションを用いた等価性検証手法を紹介する。第3節で提案する検証手法を述べ、第4節で実際の最適化に対する実験結果により、提案手法の有効性を示す。最後に、第5節で結論と今後解決すべき課題を述べる。

```
#define N 1024
foo(int A[], int B[], int C[]) {
  int k, tmp[N], buf[2*N];
  for(k=0; k<N; k++)
s1:  tmp[k] = B[2*k] + B[k];
  for(k=N; k>=1; k--)
s2:  buf[2*k-2] = A[2*k-2] + A[k-1];
  for(k=0; k<N; k++)
s3:  C[k] = tmp[k] + buf[2*k];
}
```

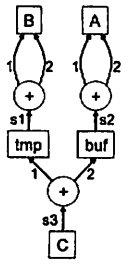

図1 ADDGの例[1]

2. 関連研究

2.1 ループ最適化に対する等価性検証手法

文献[1]では、配列間の演算を行うループ実行に対する等価性検証手法が提案されている。この手法では、検証する2つの配列データ依存グラフ (ADDG: Array Data Dependence Graph) を構築する。ADDGでは、ノードは配列か演算子を表し、エッジはデータ依存を表す。ADDGの例を図1に示す。この検証手法では、代入-使用 (define-use) の関係にある配列のインデックスをマップ (index map) として表現する。検証は、入力配列から各配列ノードの等価性を調べながら、出力配列までのADDG上の各パスをたどることによって行われる。入力と出力の間のindex mapと出力の記号式が同一である場合に、検証結果は等価となる。このとき、出力の記号式の等価性をuninterpretedに判定するため、算術的な動作の最適化などが同時に行われた場合、等価性を証明できないことがある。

この手法では、検証するプログラムは、ADDGとして表現するために、

- 変数は配列のみである
- 全ての配列要素は1回しか代入されない
- 制御フローがstaticである
- 配列インデックスがイテレータの線形式である
- ポインタ参照がない

という制約を満たす、もしくは、満たすように変換する必要がある。それに対して、本稿での提案手法では、配列変数以外の変数や同じ要素に対する複数回の代入、丸め込みなどのデータ依存する制御フローを扱うことができる。しかし、第4点、第5点については、提案手法においても、検証するプログラムが条件を満たしている必要がある。

2.2 記号シミュレーションに基づく等価性検証手法

記号シミュレーションとは、設計を記号的にシミュレートして、出力変数を表す記号式を生成する手法であり、ハードウェア設計の等価性検証を初めとする多くの検証手法に用いられており[2]、文献[3],[4]では、C言語で記述されたシステムレベル設計に対する等価性検証手法も提案されている。一般的に、記号シミュレーションでは、入力変数に対応する記号値をを入力として設計を記号的に実行し、出力変数に対応する記号式を

導出するため、テストパターンを必要としない形式的検証の実現が可能である。記号シミュレーションによる等価性検証では、記号実行中に検出された等価な記号式 (部分式も含む) をまとめていくことによって、等価性を証明していく。このとき、[5] などの論理式の真偽を判定する手法を併用することによって、複雑な記号式の等価性を証明することも可能である。

記号シミュレーションは、記号的にではあるが、設計をシミュレートするため、設計記述がループを含んでいる場合には、あらかじめループ展開を行う必要がある。このとき、証明される等価性は、展開した部分のみに対してのものであるため、展開回数が十分でない場合、証明されない実行が存在する。一方で、展開回数が多い場合には、生成される記号式が非常に大きくなり、検証時間が増大する。提案手法では、出力変数の記号式を生成するために必要なループ中の文 (statement) と対応するループの繰り返し回数を特定して、それらに対してのみ記号シミュレーションを適用する。そのため、記号シミュレーションされる文は、ループ展開を行う場合に比べて、非常に少なくなることが期待でき、検証時間の短縮が可能である。

3. 提案する検証手法

初めに、本稿で用いるイテレータとインデックスを次のように定義する。

- あるループのイテレータとは、そのループ中で実行を繰り返す度に、ある定数ずつ増加 (減少) する整数変数である。イテレータが決定されると、対応するループ実行が決定される。
- ある配列変数 a に対するアクセス $a[i]$ におけるインデックスとは、i である。

3.1 全体の流れ

提案する検証手法の全体の流れを図 2 に示す。検証手法に与える入力は以下の 3 つである。

- ループと配列を含む 2 つの C プログラム
- プログラムの入力変数・配列
- プログラムの出力変数・配列

配列については、そのインデックスが取り得る値についても指定する必要がある。検証は、与えられた入力変数・配列が等価であるときに、出力変数・配列が等価であるかどうかを調べることになる。

与えられたプログラム・入力・出力に対して、提案手法は、まず各配列アクセスのインデックスとイテレータの関係をプログラムから抽出する。そして、出力変数、または、任意のインデックスの出力配列の記号式を生成するために、シミュレートする必要がある文と対応するイテレータを特定する。最後に、それらの文を指定されたイテレータの時の実行を記号シミュレーションすることによって、出力変数・配列の記号式を生成し、等価性を判定する。

本稿では、2 つの配列変数が取り得るインデックスが同じであり、全てのインデックスに対応する要素が等価であるとき、その 2 つの配列変数は等価である、と定義する。そのため、以下のような 2 つの配列変数 a と b は等価でないと判定される。

- 取り得るインデックスが異なる場合。配列 $a[i](0 \leq i \leq 99)$、配列 $b[j](0 \leq j \leq 49)$ のとき、$a[k] = b[k](0 \leq k \leq 49)$ であっても、配列 a と b は等価ではない。
- インデックス間の対応関係の下で、全要素が等価である場合。配列 a と配列 b のインデックスがそれぞれ 0 以上 max 以下であり、$a[i] = b[max - i](0 \leq i \leq max)$ が成り立っているとしても、配列 a と b は等価ではない。

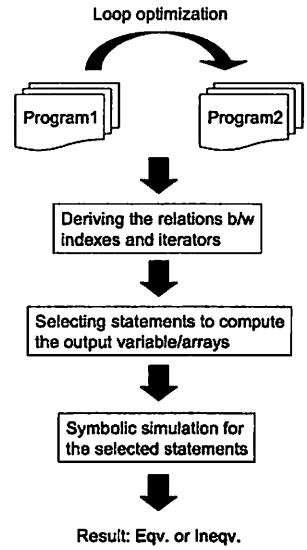

図 2 全体の検証の流れ

3.2 検証するプログラムの制約

提案手法で検証するプログラムが満たすべき制約は以下の通りである。

- 配列アクセスのインデックスはイテレータの線形式である
- ポインタ参照がない

文献 [1] に比べて、提案手法では、より制約の少ないプログラムを検証することが可能である。データ依存のある制御フローを扱えることは、データの丸め込み処理やショートカットパスを持つループの検証を可能にする。一方、同じ変数への複数回の代入が可能であるため、中間変数を含むループの検証も可能となっている。また、提案手法では、入れ子構造のループは検証の対象とせず、文献 [1] で提案されている手法と同じように、一つ、もしくは複数の一重ループのみを検証の対象とする。

3.3 インデックス - イテレータ関係の構築

初めに、検証する 2 つのプログラム中の各ループについて、イテレータの特性を次の形で表す。

$$ITER_{L_n} = (lower\_bound, upper\_bound, step)$$

ここで、L_n はループを識別するためのラベル、$lower\_bound, upper\_bound, step$ は、それぞれ、イテレータの上限、下限、増減幅である。

次に、各配列アクセス (書込、読出) について、インデックスとイテレータの関係を求める。インデックスが、イテレータの線形式として表現されている場合には、その式がインデックス

とイテレータの関係となる。インデックスが、イタレータ以外の変数によって表現されている場合には、その変数の依存関係を辿ることによって、イテレータによる表現に置き換える必要がある。この作業を行うことによって、各配列アクセスおけるインデックス - イテレータ関係は p, q を整数として、以下のように表すことができる。

$$IDX_{A,S\_m} = p * ITER_{L_n} + q$$

ここで、A はアクセスされている配列、S は A がアクセスされている文を表す。また、m は、S の中で何番目の配列アクセスであるか示す番号で、S の記述に表れる順番に応じて、左側から $1, 2, 3, \ldots$ と付ける。$ITER_{L_n}$ は、文 S を含むループである。

図 3 では、(a) に示した例題のプログラムに対する、各ループのイテレータと各配列アクセスのインデックス - イテレータ関係が (b) に示されている。

3.4 インデックス - イテレータ関係付きデータ依存グラフの解析による記号シミュレーションを行う文の抽出

提案手法では、出力変数を計算するために必要な文とその文を実行する際のイタレータをあらかじめ求め、それらに対してのみ記号シミュレーションを行って等価性を判定する。その際に、出力変数が配列である場合には、任意のインデックス i を計算するための記号式を導くことが必要である。一般的には、任意のインデックス i に対応する記号式は複数あり、最悪の場合、各インデックスについて、互いに異なる記号式が得られるため、i 通りの記号式を求める必要がある。しかし、通常のループ実行では、インデックスに依存する条件分岐は、それほど多くない場合がほとんどであるため、1 つの出力変数を計算するための記号式は 1 通り、または、数通りであることが期待できる。

出力変数を計算する記号式を求めるために、検証するプログラムのデータ依存グラフを用いる。この依存グラフでは、各エッジに、インデックス - イテレータ関係が付加されている。このインデックス - イテレータ関係付きデータ依存グラフは次のような特徴を持つ。

- ノードは、入力変数・配列、出力変数・配列、または、プログラム中の文を表す
- エッジは、データ依存を表す
- エッジは $n : (f(i), R_s) \rightarrow (g(i), R_d)$ をラベルとして持つ。このラベルの定義と作成方法については、後述する。

各エッジは、前述の通り、$n : (f(i), R_s) \rightarrow (g(i), R_d)$ をラベルとして持つ。このラベルは、読み出される配列のインデックスと書き込まれる配列のインデックスの関係を表す。ここで、ある依存エッジの根元に対応する変数を v_s、行き先に対応する変数を v_d とすると、ラベルの各要素は次のように決定される。

- n は、代入される式において v_s が出現する順番である
- R_s は、v_s が配列である場合に書き込まれるインデックスの範囲である。v_s が変数である場合は定義されない。
- R_d は、v_d が配列である場合に読み出されるインデックスの範囲である。v_d が変数である場合は定義されない。
- $f(i), g(i)$ は、v_s, v_d に対応しており、以下のように決定される。

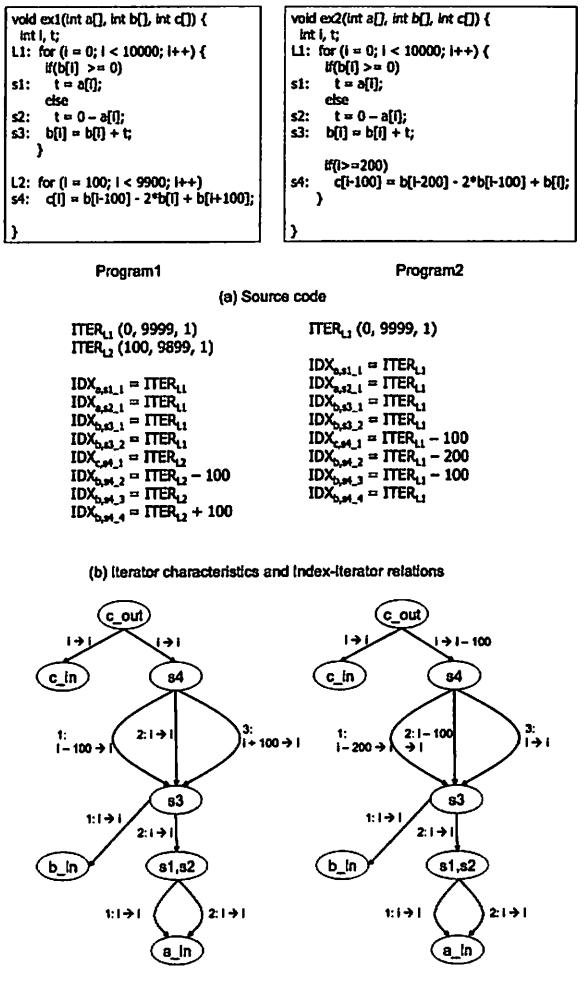

図 3 検 証 例

- n は、代入される式において v_s が出現する順番である
- R_s は、v_s が配列である場合に書き込まれるインデックスの範囲である。v_s が変数である場合は定義されない。
- R_d は、v_d が配列である場合に読み出されるインデックスの範囲である。v_d が変数である場合は定義されない。
- $f(i), g(i)$ は、v_s, v_d に対応しており、以下のように決定される。

  − 入力配列または出力配列の場合：i

  − その他の配列の場合：ループ内であればインデックス - イテレータ関係となる。ループ外であれば、アクセスされる 1 つのインデックス (定数) となる。

  − 変数の場合：ループ外であれば定義されない。ループ内であれば、v_s に対しては $f(i) = i$ とし、v_d に対しては依存がまたいでいる繰り返し回数 k に応じて、$g(i) = i + k$ とする。

同じ繰り返し実行内での依存の場合には、$g(i) = i$ となる。

また、条件分岐によって排他的に実行される代入文があったときに、同一の変数に対する代入の場合、または、同一の配列の同一のインデックスに対する代入の場合には、ノードをマージすることができる。

図 3(c) に、インデックス - イテレータ関係付きデータ依存グラフの例を示す。図中の *A_in*, *A_out* は、それぞれ、入力配列と出力配列は A であることを表している。なお、図では、簡単のため、R_s, R_d については省略している。$s1$ と $s2$ は前述の条件を満たすため、1 つのノードにマージされていることが分かる。

続いて、構築されたデータ依存グラフから、出力変数を計算するために実行しなければならない文と対応するイテレータの値を求める。図 3 の例では、任意のインデックス i に対応する配列要素 c[i] を求めようとすると、Program1 ではループ $L2$ の i 回目を、Program2 ではループ $L1$ の $i + 100$ 回目を実行する必要がある。これは、各依存エッジに付加されているラベルから、以下のアルゴリズムで求めることができる。記号シミュレーションにおいては、求めたイテレータの記号値における各文の実行のみをシミュレーションすればよい。

（1） 出力配列から出ている依存エッジの $f(i)$ と R_s に含まれる i を I で置換する

（2） 出力から入力に至る全てのパスについて、以下を行う。

- 既に f が I で表現されているエッジについて、$f(I) = g(i)$ を i について解き、その解を $C(I)$ とする
- エッジの先のノード N_d において $C(I)$ を記録する
- N_d から出ていく全てのエッジについて、$f(i)$ と R_s に含まれる i を、上で求めた $C(I)$ で置換する
- $C(I)$ で置換された R_s を I について解き、I で表現されたインデックスの取り得る領域を得る

インデックス - イテレータ関係の係数によっては、$C(I)$ の I の係数が分数になる場合がある。その場合には、全てのノードの係数が整数になるように、整数倍する。また、前述の作業は、出力から入力に至る全てのパスに対して行うため、1 つのノードに複数の $C(I)$ が記録されることがある。

図 3 の例を用いて、どのように $C(I)$ が求められるかを示す。Program1 では、まず、*c_out* から $s4$ への依存エッジについて、$I = i$ を i について解くため、$i = I$ を得る。そのため、$s4$ は $C(I) = I$ として記録される。これは、出力 c[I] を計算するためには、イテレータの値が I のときに $s4$ を実行する必要があることを示している。続いて、$s4$ から $s3$ への各エッジについて、$f(i)$ の i を $C(I) = I$ で置換して、同様に $C(I)$ を求めると、1 番目のエッジについては $I - 100$、2 番目のエッジについては I、3 番目のエッジについては $I + 100$ となる。これは、出力 c[I] を計算するためには、イテレータの値が $I - 100, I, I + 100$ のときの $s3$ を実行する必要があることを示す。同様の操作を繰り返すことによって、最終的に全てのノードがどのイテレータの値で実行される必要があるかが分かる。この作業を繰り返すと、図 3 (d) に示すような、文と対応するイテレータを得ることができる。

3.5 記号シミュレーションによる等価性検証

データ依存グラフの各ノードに記録された $C(I)$ は、出力配列の任意のインデックス I を計算するために、そのノードが実行されるべきイテレータの値を表している。そのため、記号シミュレーションでは、依存関係を保持したまま、各ノードをイテレータの値を $C(I)$ にしてシミュレーションすればよい。そこで、記号シミュレーションを行う必要がある文を制御フローに従って並べる必要がある。

また、記号シミュレーションでは、配列がアクセスされるインデックスの範囲を求める必要がある。これは、各エッジの R_s, R_d の共通部分を入力から出力まで取ることによって、各パスで計算された出力配列の記号式で有効なインデックスの範囲を得ることができる。

4. 実験結果

4.1 実験内容と例題

提案手法を実現するためには、以下の 4 つの部分が必要である。

- 与えられた C 言語プログラムからインデックス - イテレータ関係付きデータ依存グラフを構築する
- そのデータ依存グラフ上で、出力変数の記号式を生成するために、記号シミュレーションを行う必要がある文と対応するイテレータを算出する
- 算出された文とイテレータを実行順に並べ、記号シミュレーション可能な表現 (例えば、C 言語の関数) で表す
- 記号シミュレータで検証を行う

実験では、提案手法の正しさを確かめるために第 2 点を、検証時間の評価を行うために第 4 点を、それぞれ C 言語プログラムとして実装し、実際のループ最適化に対して検証を行った。実験は、2 GHz のプロセッサと 1 GB のメモリを有する PC によって行われた。上述の第 1 点については、人手により、グラフのデータ構造を作成した。また、第 3 点については、記号シミュレータが 1 組の C 言語プログラムを入力とするため、第 2 点の結果に基づいて記号シミュレーション用の C 言語の関数を人手で作成して、検証を行った。実装した記号シミュレータは、記号式の等価性を証明する際に、各演算を uninterpreted function として扱うと証明できない場合には、CVC [6] を用いて等価性を判定するようになっている。

実験に用いた例題を表 1 に示す。表中のノード数とパス数は、それぞれ、依存グラフ中のノードマージ後のノード数と、依存エッジをたどって出力配列から入力配列まで行くパス数を表している。*filter* 例題は、フィルタ演算をするプログラムに対して最適化を行ったものである。一方、*dwt* 例題は、離散ウェーブレット変換を行うプログラムに対して最適化を行ったものである。最適化前のプログラムでは、*filter*1 で 2 つ、*dwt*1 で 5 つのループを持っているが、ループフュージョンを行うことによって、それぞれ、1 つと 2 つに減少している。また、これらの例題は、文献 [1] の手法では直接扱うことのできない、ループ以外の代入文やループ内のスカラー変数への代入を含んでいる。

表 1 例題プログラムの特徴

	行数	ノード数	パス数	最適化の内容
filter1	16	5	28	Original program
filter2	19	5	28	Loop fusion
filter3	23	10	28	Loop fusion, Scalar replacement, Array duplication
filter4	22	9	28	Loop fusion, Scalar replacement
dwt1	25	7	18	Original program
dwt2	28	7	18	Loop fusion
dwt3	38	7	21	Loop fusion, Scalar replacement

表 2 実験結果

		検証結果	時間 (秒)	
			statement extraction	symbolic simulation
filter1	*filter2*	等価	< 0.1	0.50
filter2	*filter3*	等価	< 0.1	6.11
filter2	*filter4*	等価	< 0.1	6.30
filter1	$filter2_{bug}$	不等価	< 0.1	0.45
filter1	$filter4_{bug}$	不等価	< 0.1	4.96
dwt1	*dwt2*	等価	< 0.1	0.74
dwt2	*dwt3*	等価	< 0.1	1.79
dwt1	$dwt2_{bug}$	不等価	< 0.1	0.66

4.2 結　　果

実験結果を表 2 に示す。全ての実験で、正しい検証結果を得ることができた。検証時間については、第 3 節で提案したアルゴリズムでデータフローグラフを解析して、記号シミュレーションに必要な文と対応するイテレータを求める作業については、全ての例題で 0.1 秒以下となり、非常の高速に動作することが分かった。これは、検証した例題の規模がそれほど大きくなく、依存グラフ上で、出力配列から入力配列に至るパスの数が少なかったことが要因である。今回の例題では、パス数は 30 以内となっている。しかし、現実のループ最適化においても、今回の例題と同程度の規模のループに対して最適化が行われる場合も多いと想定されるため、提案手法で十分に検証可能であると考えられる。

また、記号シミュレーションによる等価性検証では、例題によっては数秒を要しているが、これは、条件分岐の判定と出力変数の等価性判定をより正確に行うために CVC を起動していることが影響している。一般的に、等価でない場合には、1 つの反例を発見した時点で、それ以上の検証を行う必要がないため、等価な場合に比べて、検証時間が短くなる傾向がある。

5. 結　　論

本稿では、ループ最適化前後のプログラムの等価性をループ展開を行わずに検証する手法を提案した。提案手法では、それぞれの出力配列に対して、任意のインデックス要素の計算を行うために必要な文と対応するループのイテレータの値をデータ依存グラフ上で求める。このとき、あらかじめ各配列アクセスに対するインデックス - イテレータ関係を求めておき、それをデータ依存グラフのエッジにラベル付けすることによって、依存グラフをたどりながら、必要な文と対応するイテレータ値を求めることができる手法を提案した。実際のループ最適化に対する検証実験により、提案手法によって、正しい検証結果を現実的な時間で得ることができることが確認できた。

現在、提案手法では、最終的な出力を計算するために必要なループの繰り返し回数が多い場合、データ依存グラフが大きくなるという問題点がある。特に、出力配列のあるインデックス要素を計算するために、ループの全ての繰り返しを実行する必要がある場合、データ依存グラフは、ループを全て展開したものと同等のものになってしまう。今後の課題として、このようなループに対する最適化の検証ができるように、帰納的な検証手法の導入を検討している。

文　　献

[1] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens, "Functional Equivalence Checking for Verification of Algebraic Transformations on Array-Intensive Source Code," *Proc. of Design, Automation and Test in Europe*, pp.1310–1315, Mar. 2005.

[2] G. Ritter, *Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation*, PhD thesis, Darmastadt University of Technology and Universite Joseph Fourier, 2000.

[3] H. Saito, T. Ogawa, T. Sakunkonchak, M. Fujita, and T. Nanya, "An Equivalence Checking Methodology for Hardware Oriented C-based Specifications," *Proc. of International Workshop on High Level Design Validation and Test*, pp.139–144, 2002.

[4] T. Matsumoto, H. Saito, and M. Fujita, "Equivalence Checking of C Programs by Locally Performing Symbolic Simulation on Dependence Graphs," *Proc. of 7th International Symposium on Quality Electronic Design*, pp.370–375, Mar. 2006.

[5] A. Stump, C. Barret, and D. Dill, "CVC: a Cooperating Validity Checker," *Proc. of International Conference on Computer-Aided Verification*, Jul. 2002.

[6] A Cooperating Validity Checker (CVC): http://verify.stanford.edu/CVC/

[7] D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publisher, Mar. 2000.

[8] SystemC: http://www.systemc.org/