Vol.2010-SLDM-146 No.4
2010/10/5

ループ融合を利用した複数の for ループからのパイプラインハードウェア合成

瀬戸 謙修[†1]　田中 輝明[†2]　佐々木 俊介[†3]
小松 聡[†4]　藤田 昌宏[†4]

短期間にハードウェアを設計するための技術として，C 言語からの高位合成が有効である．特に高位合成によるループのパイプライン化は，高性能なハードウェアを設計するのに役に立つ．しかしながら現状の高位合成ツールでは単純なパイプライン自動合成しかできないことが多い．本稿では，高位合成においてパイプライン化する前に複数の連続するループを融合するなど各種のソースレベル最適化を適用することで，高位合成ツールをそのまま使用した場合よりも高性能なパイプラインを合成する方法を提案する．また，ループ融合を適用することでパイプラインの性能が向上する可能性があるかどうか判断する式を導く．実験結果から，提案する最適化によって連続するループを積極的に融合することで，パイプラインの性能が大幅に向上することが示された．

Pipeline synthesis from multiple for-loops with loop fusion

Kenshu Seto,[†4] Teruaki Tanaka,[†2]
Shunsuke Sasaki,[†3] Satoshi Komatsu[†4]
and Masahiro Fujita [†4]

High-level synthesis from C is an effective technique for designing hardware in short time. In particular, loop pipelining using high-level synthesis helps design high-performance hardware. However, most existing high-level synthesis tools can only perform simple pipeline synthesis. In this paper, we propose a method for synthesizing high-performance pipeline by applying several source-level optimizations such as fusing successive loops before loop pipelining using high-level synthesis. We also present formulae that show whether loop fusion can improve the performance of synthesized hardware pipeline. Experimental results showed that the performance of pipeline can be greatly improved by fusing successive loops aggressively.

1. 背　景

ハードウェア設計期間短縮のため，C 言語による高位合成ツール[1)2)3)] が実設計に適用されつつある。高位合成とは，RTL(レジスタ転送レベル) に比べてより抽象度の高い言語（C 言語など）で書かれた設計記述から RTL を自動生成する技術であり，すでに多くの研究成果が報告されている．高位合成の主な処理には，演算のスケジューリング，演算器のアロケーション，演算の演算器へのバインディングがある．高位合成には，シミュレーション時間の短縮，ハードウェア・ソフトウェア協調設計の行いやすさ，設計記述の変更・修正の容易さなどをはじめとする利点がある．今後高位合成が幅広く利用されるようになるためには，与えられた C 記述からさらに高性能な RTL 記述が生成されることが重要である．

ところが現状の高位合成ツールの欠点として，与えられた C 記述の書き方によっては高性能な RTL が生成されない場合があることが挙げられる．高性能にならない理由は，C 記述によっては，並列化を妨げる依存関係や制約が存在するためである．このような依存関係や制約を取り除くため，並列化コンパイラ技術[4)] が長い間研究されてきており，すでに多くの研究成果が得られている．並列化コンパイラ技術にあるようなソースレベル最適化と高位合成をうまく組み合わせることにより，より高性能な RTL が生成できると期待できる．

そこで本稿では，高位合成ツールの前処理としてソースレベル最適化を行い，高位合成ツール単体を使用したときと比べて，生成される RTL の性能を向上させる方法を提案する．具体的には，C 記述中の連続する複数ループを対象とし，ループ融合をはじめとするソースレベル最適化を施した後に高位合成を用いることで高性能なパイプラインを合成する手法を提案する．画像処理のようなアプリケーションでは，性能上のボトルネック部分は連続する複数のループ処理で記述されていることが多い．提案する一連の最適化によってそのような部分を効率的にパイプライン化することで，パイプライン性能の大幅な向上が期待できる．ただし連続する複数ループを融合したままでは性能的に満足のいくパイプラインが得られるとは限らないため，ループ融合前後に適用すべき最適化についても述べる．例えば，ローカルメモリのメモリアクセスボトルネックを緩和するため，配列の複製などを行う．

†1 東京都市大学工学部 School of Engineering, Tokyo City University
†2 三菱電機 (株) Mitsubishi Electric Corporation
†3 東京大学, 現在東芝 Previously with The University of Tokyo, currently with Toshiba Corp.
†4 東京大学大規模集積システム設計教育研究センター VDEC, The University of Tokyo

Vol.2010-SLDM-146 No.4
2010/10/5

```
1  void filter(int a[], int c[]) {
2    int b[10000], i;
3    for (i = 1; i < 9999; i++)     // LOOP L1
4      b[i] = a[i-1] - 2*a[i] + a[i+1];
5    for (i = 100; i < 9899; i++)   // LOOP L2
6      c[i] = b[i-100] - 2*b[i] + b[i+100];
7  }
```

図 1 例題
Fig. 1 Example

```
1  void filter(int a[], int c[]) {
2    int b[10000], i;
3    for (i = 1; i < 9999; i++)     // LOOP L1
4      b[i] = a[i-1] - 2*a[i] + a[i+1];
5    for (i = 200; i < 9999; i++)   // LOOP L2
6      c[i-100] = b[i-200] - 2*b[i-100] + b[i];
7  }
```

図 2 ループアラインメント
Fig. 2 Loop alignment

図 1 に，現状の高位合成ツールでは高性能なパイプラインが自動合成できない例を示す．4 節で，この例を使用して提案手法を説明する．なおこの例は，画像のフィルタ処理を変更・簡単化した C 記述である．図 1 の例では，配列 a[], c[] はそれぞれ 1 ポートの外部メモリにマッピングされ，配列 b[] は 2 ポートのローカルメモリにマッピングされるものとする．この例で高性能なパイプラインが合成できないのは，以下の理由による．

- 外部メモリ a[] へのアクセスが 4 行目に 3 回あるが，ポートが 1 つしかないため，リソース競合が起こりスループットが遅くなる．
- 6 行目でローカルメモリ b[] へのアクセスが 3 回あるが，ポートが 2 つしかないため，リソース競合が起こりスループットが遅くなる．
- 3 行目のループ L1 が終わってからでないと 5 行目のループ L2 が実行できない．

提案手法を適用することで，ここに述べられた問題は解決され，処理全体のパイプライン化が可能になる．

2. 関 連 研 究

Ziegler らは，ソースレベル最適化を活用して，連続する複数ループから，各ループを非同期的に動作するパイプラインステージとしてパイプライン合成を行う手法を提案した[5]．Weinhardt ら[6] はループに対していくつかのループ変換を適用した上でパイプライン化を行った．ループ融合の利用にも言及しているが，スカラリプレースメント[4] を行なっていないため，I/O ポートの制約によりパイプラインのスループットが遅くなると述べられている．本稿の提案手法では，ループ融合後に，スカラリプレースメントを適用することで，このような問題を解決する．特に本稿では高性能なパイプラインを合成するためのソースレベル最適化としてループ融合に注目し，それ以外の最適化も合わせた最適化方法を提案する．

3. ループパイプライン化

本稿で提案する最適化後に，高位合成ツールでループパイプライン化を適用することを想定している．そのため，本節ではループパイプライン化に関する予備知識と，よく利用されているループパイプライン化技術であるモジュロスケジューリング[7] について解説する．ループパイプライン化とは，ループ処理に対し，複数の繰り返し処理 (イタレーション) を重ねて同時実行させることで，性能向上を図る技術である．パイプラインの開始間隔 II (Initiation Interval) とは，あるイタレーション i を実行開始してから，次のイタレーション i+1 を実行開始するまでのサイクル数である．レイテンシ L とは，一つのイタレーション i を開始してから終了するまでのサイクル数と定義する．

3.1 パイプラインの全実行時間

N をループ実行回数とするとき，パイプラインの全実行サイクル Cycle は，次式となる．

$$Cycle = II \times (N-1) + L \simeq II \times N \quad (1)$$

ただし，画像処理などのように，N が II や L に比べてはるかに大きい場合（本稿でも以降そのような前提を設ける）を想定し，近似を行った．開始間隔 II が短いほど，パイプラインのスループットが大きくなる．パイプラインの全実行時間は，(1) 式の全実行サイクルに，合成されたパイプラインの動作するクロック周期を乗じたものである．目標クロック周期は制約条件として高位合成ツールに入力され，高位合成ツールはそれをできるだけ満たすような RTL を生成する．本稿では，ソースレベル最適化の後に，高位合成ツールが目標クロック周期で動作するパイプラインを合成できるものと仮定して議論を進める．

3.2 開始間隔の下限値

モジュロスケジューリング[7] では，与えられたループをパイプライン化した場合の開始間隔の下限値 MII (Minimum Initiation Interval) を求め，開始間隔 II をまず MII に設定してリストスケジューリングを行う．スケジューリングに失敗した場合，開始間隔 II を 1 増やし，再びスケジューリングを行い，最終的にスケジューリングが見つかるまでこの過程を繰り返す．通常，開始間隔 II は下限値 MII と同じ値になることが多い．

開始間隔の下限値 MII は，リソース制約で決まる下限値 ResMII および，ループ中のイタレーションにまたがる依存関係で決まる下限値 RecMII のどちらか大きい方になる[7]．

$$MII = \max(ResMII, RecMII) \quad (2)$$

ResMII は次式で表される．

$$ResMII = \max_{r \in R} \left\lceil \frac{Use(r)}{Num(r)} \right\rceil \quad (3)$$

式 (3) において，R は異なる種類のリソース（例えば加算器，乗算器，メモリポートなど）からなる集合を表し，Use(r) はリソース r がループ記述中で使用されている回数，Num(r) はリソース r の利用可能な個数を表す．この式から，ループ記述中に利用可能な個数よりも多くのリソースが使用されている場合，ResMII は 1 より大きくなることがわかる．これはリソース競合が原因で，パイプラインストールが起こることを表している．

一方，RecMII は次式で表される．

$$RecMII = \max_{c \in C} \frac{Delay(c)}{Distance(c)} \quad (4)$$

式 (4) において，C はループ記述（ループ本体の基本ブロック）をデータ依存グラフ (Data Dependence Graph)[7] で表したときの，閉路の集合を表す．Delay(c) はデータ依存グラフ中の閉路 c を一周したときのエッジに割り当てられた遅延の合計，Distance(c) は依存距離の合計である．

3.3 高性能なパイプライン合成のためのソースレベル最適化

ループ実行回数が固定されており，しかもその回数が多い場合，高性能なパイプライン合成のためには，開始間隔 II をできるだけ小さく抑えることが重要である．これは，開始間隔の下限 MII をできるだけ小さくすることにほぼ相当する．ハードウェア合成では，面積制約に収まれば，演算器の数は，必要なだけ用意することが可能である．しかし，I/O やメモリのポート数は限られているので，メモリアクセスが多い場合に開始間隔の下限値を小さな値に抑えるには，ソースレベル最適化によりメモリアクセスを削減するか，メモリバンド幅を向上する必要があり，本稿ではそのための手法を 4 節の中で説明する．

4. 提 案 手 法

本節では，高位合成によるパイプライン化前に，ループ融合をはじめとするソースレベル最適化を適用することで，合成されたパイプラインの性能を向上させる手法を提案する．この目的のために並列化コンパイラ技術[4] を部分的に利用する．提案手法により，逐次的に実行されていた二つのループがループ融合によって同時に実行されるようになる．特にループ融合がパイプラインの性能に与える影響として，開始間隔の下限 MII がどのように変化するかが重要であるが，5 節で証明するようにループ融合後の開始間隔の下限 MII_{L1+L2} は，ループ融合前の開始間隔の下限 MII_{L1}, MII_{L2} の合計以下であるため，式 (1) で表される性能がループ融合後に悪化することはない．したがって提案手法のようにループ融合を積極的に行うことで，パイプラインの性能向上が期待できる．ただし，この議論では融合する複数のループの実行回数はどれもほぼ同じと仮定している．

4.1 提案手法のフロー

提案する一連の最適化において，入力はハードウェア化対象の C 記述，出力はパイプライン合成向けに最適化された C 記述である．出力の C 記述は，高位合成ツールでループパイプライン化する．

提案する一連の最適化では，以下のステップを実施する．

(1) 連続する複数ループの識別
(2) ループアラインメント
(3) イタレーションの条件付き実行
(4) ループ融合
(5) スカラーリプレースメント
(6) 配列の複製

なお対象とするループは一重ループと仮定し，他のループを含まないものとする．実行回数の少ないループが含まれている場合は，あらかじめループ展開しておくことで扱える．

提案するソースレベル最適化は，連続する複数のループからなる C 記述に対して主に以下を行い，高性能なパイプラインを合成することを目的とする．

- ループ融合を妨げる依存性の除去
- ループ融合による複数ループの同時実行
- 外部メモリアクセスの削減
- ローカルメモリ分割によるメモリバンド幅の向上

4.2 例を使用した説明

図 1 の C 記述を例にとり，提案手法で使用する最適化を説明する．

4.2.1 ループアラインメント[4]

ループ融合のように，文の実行順序を入れ換えるソースレベル最適化を行う場合，その前後ですべての依存関係（フロー依存，逆依存，出力依存）が保存されている必要がある．逆に，それらの依存関係が保存されていさえすれば，その入れ換え後のプログラムは，入れ換え前のプログラムと等価であることが知られている[4]．一般に，複数のループを融合する場合，そのままでは必ずしも依存関係が保存されるとは限らないため，前処理を行ってループ融合前後で依存関係が保存されるようにしておく必要がある．そのような時に利用される

```
1  void filter(int a[], int c[]) {
2    int b[10000], i;
3    for (i = 1; i < 9999; i++)  // LOOP L1
4      b[i] = a[i-1] - 2*a[i] + a[i+1];
5    for (i = 1; i < 9999; i++)  // LOOP L2
6      if (i >= 200)
7        c[i-100] = b[i-200] - 2*b[i-100] + b[i];
8  }
```

図 3 イタレーションの条件付き実行
Fig. 3 Guarded iterations

```
1  void filter(int a[], int c[]) {
2    int b[10000], i;
3    for (i = 1; i < 9999; i++) {   // LOOP L1+L2
4      b[i] = a[i-1] - 2*a[i] + a[i+1];
5      if (i>=200)
6        c[i-100] = b[i-200] - 2*b[i-100] + b[i];
7    }
8  }
```

図 4 ループ融合
Fig. 4 Loop fusion

ソースレベル最適化がループアラインメント (loop alignment) である．

図 2 に，図 1 の C 記述をループアラインメントした後の C 記述を示す．図 1 の 2 つのループ L1, L2 をそのままループ融合すると，4 行目の配列への書き込み b[i] と 6 行目の配列参照 b[i+100] において依存関係の逆転が起こる．つまり，6 行目の配列参照 b[i+100] が，4 行目の配列への書き込み b[i] が行われる前に参照されてしまう．ループアラインメント後の C 記述では，ループ L2 のループインデックス i を 100 ずらし（ループの初期値と終了条件もそれに合わせて変更），ループ融合しても依存関係の逆転が起こらないようにする．提案手法では，ループアラインメントによって可能な限りループ融合できるようにする．

4.2.2 イタレーションの条件付き実行

二つのループを融合する場合，ループの初期値および終了条件が一致している必要がある．ループアラインメントを行った後には必ずしもこの条件が成立しなくなるため，何らかのソースレベル変換を行いこの条件を満足させる必要がある．

そのような変換の一つとして，イタレーションの条件付き実行がある．図 2 の C 記述中で，ループ L1 のインデックスは [1, 9999)，ループ L2 のインデックスは [200, 9999) という異なる範囲を動く．ループ L1,L2 のインデックスの範囲をそろえるため，イタレーションの条件付き実行を利用して変形した例を図 3 に示す．図 2 のループ L2 の初期値および終了条件をループ L1 のものと一致させ，そのかわり図 3 の 6 行目の条件文を付加した．

4.2.3 ループ融合[4)]

この例では，ループアラインメントおよびイタレーションの条件付き実行という前処理の後，ループ融合が可能になる．図 3 の C 記述をループ融合した結果を，図 4 に示す．一般にループ融合すると，必要なレジスタ数が増える傾向にある．しかしながら，レジスタ数の決まっているプロセッサと異なり，ハードウェア合成では比較的多くのレジスタを用意可能である．したがって提案手法では，パイプライン高性能化のため，可能な限り最大限ループ融合を行う方針をとる．

```
1  void filter(int a[], int c[]) {
2    int b[10000], i, tb1, ta1, ta2, ta3;
3    ta1 = a[0];
4    ta2 = a[1];
5    for (i = 1; i < 9999; i++) { // LOOP L1+L2
6      ta3 = a[i+1];
7      tb1 = ta1 - 2*ta2 + ta3;
8      b[i] = tb1;
9      if (i>=200)
10       c[i-100] = b[i-200] - 2*b[i-100] + tb1;
11     ta1 = ta2;
12     ta2 = ta3;
13   }
14 }
```

図 5 スカラリプレースメント
Fig. 5 Scalar replacement

```
1  void filter(int a[], int c[]) {
2    int b1[10000], b2[10000], i, tb1, ta1, ta2, ta3;
3    ta1 = a[0];
4    ta2 = a[1];
5    for (i = 1; i < 9999; i++) { // LOOP L1+L2
6      ta3 = a[i+1];
7      tb1 = ta1 - 2*ta2 + ta3;
8      b1[i] = tb1;
9      b2[i] = tb1;
10     if (i>=200)
11       c[i-100] = b1[i-200] - 2*b2[i-100] + tb1;
12     ta1 = ta2;
13     ta2 = ta3;
14   }
15 }
```

図 6 配列の複製
Fig. 6 Array duplication

4.2.4 スカラリプレースメント[4)]

本稿では配列はメモリにマッピングされると仮定する．スカラリプレースメントは，メモリアクセスで読み込んだデータが後に再びアクセスされる場合，最初のアクセスで読んだ値を一時的にスカラ変数 (レジスタ) に入れておき，二度目のアクセス以降ではそのスカラ変数を参照するようにすることで，メモリアクセス数を削減するソースレベル最適化である．

図 4 のループ融合された C 記述に対して，スカラリプレースメントを適用した結果を図 5 に示す．図 4 の 4 行目における配列 a[] の参照と 6 行目における配列 b[] の参照の一部が，図 5 ではスカラ変数 ta1, ta2, ta3, tb1 の参照に置き換わっている．この変換により，ループ L1+L2 における外部メモリ a[] へのアクセス数が 3 から 1 に，内部メモリ b[] へのアクセス数も 4 から 3 に削減されている．なお，図 5 の 10 行目の配列 b[] の参照 b[i-200], b[i-100] も同様にスカラリプレースメント可能であるが，スカラ変数（すなわちレジスタ）が大量に必要になるため行わない．その代わり，次に説明する配列の複製で対応する．

4.2.5 配列の複製

図 5 のループ L1+L2 中において，1 ポートの外部メモリにマッピングされる配列 a[], c[] へのアクセスはそれぞれ 1 回だけなのに対し，2 ポートの内部メモリにマッピングされる配列 b[] へのアクセスは 3 回あるため，ローカルメモリへのアクセスがボトルネックとなってパイプラインの開始間隔の下限 MII を 1 にすることができない．

図 6 に示すようにローカル配列 b[] を二つのローカル配列 b1[] および b2[] （したがって二つのローカルメモリ）に複製することで，ローカルメモリのメモリバンド幅を向上させ，開始間隔の下限 MII を 1 にすることができるようになる．これはローカルメモリの構成方法が自由なハードウェア合成において有効なソースレベル最適化である．

5. ループ融合のパイプライン化への影響

前節では，高性能なパイプライン合成を行うため，ループ融合可能な連続するループを積極的に融合する手法を示した．しかし，このようなループ融合によって性能が向上するかどうかは必ずしも自明ではない．そこで本節ではループ融合が，合成されたパイプラインの性能へ与える影響を考察する．例えば画像処理フィルタのようにループ実行回数が大きい場合，パイプラインの性能はほぼ開始間隔の下限 MII で決まるため，ループ融合の MII への影響を調べれば，パイプラインの性能がどうなるか近似的に分かる．

そこで，二つの一重ループ L1, L2 を融合する場合について，MII がどのように変化するかを調べる．MII は，ResMII および RecMII のどちらか大きい方で定まるため，まずそれらの変化をそれぞれ調べる．まず式 (3) で表される ResMII へのループ融合の影響を調べる．ループ L1，ループ L2，およびそれらを融合したループ L1+L2 の C 記述中で，リソース r が使用されている回数を，$Use_{L1}(r), Use_{L2}(r), Use_{L1+L2}(r)$ と表す．このとき，ループ融合前後の ResMII は次式のようになる．

$$ResMII_{L1} = \max_{r \in R} \left\lceil \frac{Use_{L1}(r)}{Num(r)} \right\rceil \tag{5}$$

$$ResMII_{L2} = \max_{r \in R} \left\lceil \frac{Use_{L2}(r)}{Num(r)} \right\rceil \tag{6}$$

$$ResMII_{L1+L2} = \max_{r \in R} \left\lceil \frac{Use_{L1+L2}(r)}{Num(r)} \right\rceil \tag{7}$$

ループ L1+L2 の C 記述中のリソース r の使用回数は，ループ L1 および L2 の C 記述中のリソース r の使用回数の和よりも大きくなることはない．スカラリプレースメントや共通部分式削除などの最適化を利用すれば逆に減ることもありうる．したがって次式が得られる．

$$Use_{L1+L2}(r) \leq Use_{L1}(r) + Use_{L2}(r) \tag{8}$$

式 (7) および式 (8) を合わせると，

$$ResMII_{L1+L2} \leq \max_{r \in R} \left\lceil \frac{Use_{L1}(r) + Use_{L2}(r)}{Num(r)} \right\rceil \tag{9}$$

が成り立つ．また，式 (5)，(6)，(9) および

$$\max_{r \in R} \Big\lceil f(r) + g(r) \Big\rceil \leq \max_{r \in R} \Big\lceil f(r) \Big\rceil + \max_{r \in R} \Big\lceil g(r) \Big\rceil \tag{10}$$

という関係式から，次の式を得る．

$$ResMII_{L1+L2} \leq ResMII_{L1} + ResMII_{L2} \tag{11}$$

次に式 (4) で表される RecMII へのループ融合の影響を調べる．ループ L1 および L2 を融合してループ L1+L2 になったときに新たにサイクルが生成されることはない．したがってループ L1+L2 中のサイクルの集合 C_{L1+L2} は，ループ L1 および L2 中のサイクルの集合 C_{L1}, C_{L2} の和集合になる．このことから，ループ L1+L2 の RecMII は次式で表される．

$$\begin{aligned} RecMII_{L1+L2} &= \max_{c \in C_{L1+L2}} \frac{Delay(c)}{Distance(c)} \\ &\leq \max_{c \in C_{L1} \cup C_{L2}} \frac{Delay(c)}{Distance(c)} \\ &= \max(RecMII_{L1}, RecMII_{L2}) \end{aligned} \tag{12}$$

上式と式 (11) を合わせると，ループ L1+L2 の開始間隔の下限 MII_{L1+L2} は次式となる．

$$\begin{aligned} MII_{L1+L2} &= \max(ResMII_{L1+L2}, RecMII_{L1+L2}) \\ &\leq \max(ResMII_{L1} + ResMII_{L2}, RecMII_{L1}, RecMII_{L2}) \end{aligned} \tag{13}$$

一方，ループ L1，L2 の開始間隔の下限 MII_{L1}，MII_{L2} はそれぞれ以下の式で表される．

$$MII_{L1} = \max(ResMII_{L1}, RecMII_{L1}) \tag{14}$$

$$MII_{L2} = \max(ResMII_{L2}, RecMII_{L2}) \tag{15}$$

以上の式より，次の式が成立する．

$$MII_{L1+L2} \leq MII_{L1} + MII_{L2} \tag{16}$$

最後に，ループ融合後の性能 (実行サイクル数) を表す式を求める．イタレーションの条件付き実行を行う前のループ L1 および L2 のループ変数の上限と下限を，それぞれ l_{L1}, u_{L1}, l_{L2}, u_{L2} とする．この場合，ループ融合前の性能は以下のように表される．

$$(u_{L1} - l_{L1} + 1) \times MII_{L1} + (u_{L2} - l_{L2} + 1) \times MII_{L2} \tag{17}$$

ループ融合後の性能は，以下のように表される．

$$\{max(u_{L1}, u_{L2}) - min(l_{L1}, l_{L2}) + 1\} \times MII_{L1+L2} \tag{18}$$

式 (16)，(17) および (18) を利用することにより，ループ融合後に性能が向上しうるかどうか判断可能である．例えば，上限 u_{L1}，u_{L2} および下限 l_{L1}，l_{L2} が互いに近い値のとき，ループ融合後の性能は，最悪の場合でもループ融合の前とほぼ等しいといえる．したがって，そのような場合には積極的にループ融合を行うことにより，パイプラインの性能向上が期待できる．

6. 実　　験

提案手法が，合成されたパイプラインの性能にどの程度効果があるかを調べるため，3 つの例題を使用して実験を行った．実験では高位合成ツールとして Synopsys 社の CoCentric SystemC Compiler[8] を利用してループパイプライン化を行い，出力レポートから，パイプラインの開始間隔 II，レイテンシ L，ゲート規模のデータを得た．なお，32 ビット加算器

Vol.2010-SLDM-146 No.4
2010/10/5

が 1 クロックで動作するように，高位合成の目標動作クロック周期として 80MHz を与えた．使用したライブラリのプロセスは 0.18μm である．なお，ソースレベル最適化は人手で行った．

使用した例題は，filter.c が本稿で使用したフィルタの例題，wavelet.c がウェーブレット変換のカーネル処理の一部，image.c が動画像処理である．それぞれ 2 個，4 個，6 個の融合可能な連続するループからなる．例題では，すべてのループを融合した．

各々の結果を，表 1 から 3 に示す．各例題について，高位合成のみを適用した場合と，提案手法も適用した場合のそれぞれについて，開始間隔 II，レイテンシ L，ゲート数を示した．高位合成のみの場合には，各ループを個別に最大限パイプライン化した．高位合成のみの場合の開始間隔 II の合計は，連続するループを実行するのに要した合計時間の目安となる．

例えば filter.c の例では，表 1 の開始間隔 II において，提案手法は II=1 であるのに対し，高位合成のみの場合は合計 II=5 となっており，5/1=5.0 倍の高速化が達成されている．ただしゲート規模は 2.4 倍に増大している．wavelet.c の例題では，表 2 に示されているように，6 倍の性能向上がなされているだけでなく，ゲート数もわずかに減少している．これは 12 個あった配列変数へのアクセスが，スカラリプレースメントによって 4 個に減少したため，メモリアクセス部分の回路が削減されたためと推測される．image.c の例題は，動画の X 方向のライン処理を表す連続するループである．要求仕様として，ビデオレート (30 フレーム/秒) である．動画のサイズを 1280x960 ピクセルとすると，高位合成のみの設計では，X 方向の 1 ライン分の処理に，およそ 11 × 1280 サイクル要する．したがって，一枚の画像を処理するのに，11 × 1280 × 960 サイクル要する．動作周波数が 80MHz のため，

$$\frac{80 \times 10^6}{11 \times 1280 \times 960} = 5.9 frame/s$$

となり，要求仕様が満足されない．提案手法により 5.5 倍の速度向上が可能なため，要求仕様を満たすことが可能となる．

7. 結　　論

従来の高位合成ツールは並列化のためのソースレベル最適化を自動で行っておらず，最適なパイプライン回路を合成できない場合がある．本稿では，ループ融合を中心としたソースレベル最適化と高位合成によるループパイプライン化とを組み合わせることによりパイプラインの性能を大幅に向上可能であることを示した．実験から，高位合成のみを使用した場合に 5.9 フレーム/秒しか性能が出なかった動画処理が，提案手法の適用によりビデオレートで動作するようになることがわかった．今後の課題は，今回適用した一連の最適化の自動化および一般化である．

表 1　実験結果 (filter.c)
Table 1　Experimental results (filter.c)

	ループ	II	L	ゲート数
高位合成のみ	L1	3	6	-
	L2	2	4	-
	合計	5	-	1.3K
提案手法	-	1	4	3.2K

表 2　実験結果 (wavelet.c)
Table 2　Experimental results (wavelet.c)

	ループ	II	L	ゲート数
高位合成のみ	L1	5	10	-
	L2	5	10	-
	L3	1	3	-
	L4	1	3	-
	合計	12	-	784
提案手法	-	2	6	770

表 3　実験結果 (image.c)
Table 3　Experimental results (image.c)

	ループ	II	L	ゲート数
高位合成のみ	L1	2	8	-
	L2	1	3	-
	L3	3	15	-
	L4	1	3	-
	L5	3	15	-
	L6	1	15	-
	合計	11	-	32K
提案手法	-	2	20	101K

参 考 文 献

1) NEC Corporation: CyberWorkBench.
2) Mentor Graphics Corp: Catapult C.
3) Forte Design Systems: Cynthesizer.
4) Allen, R. and Kennedy, K.: *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers (2002).
5) Ziegler, H., So, B., Hall, M. and Diniz, P.C.: Coarse-Grain Pipelining on Multiple FPGA Architectures, *Proceedings of FCCM'02*, p.77 (2002).
6) Weinhardt, M. and Luk, W.: Pipeline Vectorization, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.20, No.2, pp.234–248 (2001).
7) Rau, B.R.: Iterative modulo scheduling: an algorithm for software pipelining loops, *Proceedings of MICRO 27*, pp.63–74 (1994).
8) Synopsys, Inc: CoCentric SystemC Compiler.