Vol.2011-SLDM-149 No.21
Vol.2011-EMB-20 No.21
2011/3/18

インクリメンタル高位合成に向けた設計記述間差分の計算手法

吉田浩章[†1,†2] 藤田昌宏[†1,†2]

ASIC の開発コスト増大と開発期間短縮に伴い，設計後の修正を行う Engineering Change (EC) 手法の重要性が増している．また一方で，ASIC 設計の生産性向上を目的として高位合成を利用した設計手法の導入が進んでいる．高位合成の普及とともに，従来のゲートレベルにおける EC 手法ではなく高位設計における EC 手法の重要性が高まってきており，近年インクリメンタル高位合成手法が提案されている[1),2)]．これらの手法では設計記述間の差分が合成結果に大きく影響を与えるため，できるだけ小さい差分を求めることが重要である．そこで本稿では，設計記述間の最小差分を求める手法を提案する．提案手法では内部表現の文字列表現の間の差分を求めることにより効率的に設計記述間の差分を求めることができる．実際にインクリメンタル高位合成システムに提案手法を実装し，例題に対して初期的な実験を行った．

Difference Computation of Design Descriptions For Incremental High-Level Synthesis

Hiroaki Yoshida [†1,†2] and Masahiro Fujita[†1,†2]

With the rising cost of SoC development and the shorter time-to-market, Engineering Change (EC) methodology has been becoming important. Recently, high-level synthesis methodology has been widely used to achieve a higher design productivity. To enable engineering change in high-level design methodology, inremental high-level synthesis methods have been proposed[1),2)]. In these methods, the difference of design descriptions impacts the synthesized results and hence it is desirable to minimize the difference. In this paper, we propose a method which computes the minimum difference of design descriptions. Since the proposed method is based on textual difference of sequences of instructions, the design difference can be computed efficiently. We have implemented the proposed method in our incremental high-level synthesis system, and conducted a preliminary experiment.

1. はじめに

特定用途向け集積回路 (ASIC) では製造後の修正がほぼ不可能なため，設計時に様々な要素を考慮した上での慎重な最適化が必要であり，また網羅的な検証も必要となるため，開発コストが必然的に高くなってしまう．また近年の製造プロセスの高度化に伴い，製造コストも莫大となっている．このため，再利用が不可能な投資に関するコストである NRE(Non-Recurring Engineering) コストが非常に高い．

そのため ASIC 開発においては，開発コストや開発期間の短縮を目的として設計後の修正を効率的に行う Engineering Change (EC) 手法が用いられてきた．EC 手法では，設計過程のある段階で設計誤りを発見した場合に設計全体をやり直すのではなく局所的な設計変更のみで対応することにより，設計および検証のコストを最小限にすることが可能である．現在広く行われている EC 手法では，あらかじめスペアセルと呼ばれる予備のセルをあらかじめ挿入しておき，設計変更が必要な場合には配線をスペアセルにつなぎかえることで機能修正を行う．これらの処理を自動化する技術として，インクリメンタル論理合成[3)–5)] やインクリメンタル自動配置配線[6)] が提案されている．これらの手法では基本的な手続きとして，まず設計の変更部分を特定し，次にその変更部分に対して変更がなるべく少なくなるように再合成を行う．これらの技術は商用ツールにも実装されており，広く利用されている．

一方で ASIC 開発の生産性向上を目的として，高位合成を利用した設計手法の導入が進んでいる．高位合成の普及とともに，従来のゲートレベルでの EC 手法ではなく，高位設計における EC 手法の重要性が高まってくるものと思われる．図 1 にオープンソースの Advanced Audio Coding (AAC) デコーダソフトウェアである FAAD2[7)] におけるバグ修正の例を示す．バージョン 1.33 からバージョン 1.34 にかけて，式の符号が修正されている．このようにバグ修正は設計全体に対して局所的かつ小規模であることが多いが，一般的な高位合成手法で再合成した場合には小規模な変更であっても合成結果が大幅に変わってしまう可能性があり，結果として論理・レイアウト設計の大部分をやり直さなくてはならなくなってしまう．

このような背景から，近年になりインクリメンタル高位合成手法が提案されている[1),2)]．Lavagno らの手法は，元の設計の合成時にスケジューリングやバインディングの情報を記憶しておき，変更後の設計を行う際に使用することでなるべく元の設計と近いハードウェア

†1 東京大学 大規模集積システム設計教育研究センター (VDEC)
†2 科学技術振興機構 戦略的創造研究推進事業 CREST

Vol.2011-SLDM-149 No.21
Vol.2011-EMB-20 No.21
2011/3/18

```
for(k = 0; k < N4; k++) {
                ...
     $out[          n] =  RE(x);
     $out[N2 - 1 - n] = -IM(x);
     $out[N2     + n] =  IM(x);
     $out[N  - 1 - n] = -RE(x);
}
```

(a) バージョン 1.33

```
for(k = 0; k < N4; k++) {
                ...
     $out[          n] = -RE(x);
     $out[N2 - 1 - n] =  IM(x);
     $out[N2     + n] = -IM(x);
     $out[N  - 1 - n] =  RE(x);
}
```

(b) バージョン 1.34

図 1 Engineering change の例: FAAD2 における mdct.c のバグ修正

を合成する．この手法では厳密な設計記述間の差分を計算しているのではなく，類似した設計であれば変数名なども類似しているという仮定の下で内部表現を文字列で表現し比較を行うことで差分の見積もりを行っている．そのため変数名の変更などを伴う設計変更に対してはロバストではない．小野らによる手法[2] では厳密な設計記述間の差分を入力として，変更部分のみをインクリメンタルに再合成することにより合成結果の変更量を最小にすることが可能である．また，変更部分のみを再合成するため，設計変更に対して比較的ロバストであると言える．また製造後機能修正を可能とするアクセラレータ[8] でも設計記述間の差分に基づいてコンパイルを行いパッチを生成している．これらの手法では設計記述間の差分が合成結果に大きく影響を与えるため，できるだけ小さい差分を求めることが重要である．

高位合成の内部表現として一般的に用いられるコントロールデータフローグラフ (CDFG) はグラフ表現であるため，グラフ間の差分を求めることによって設計記述の差分を計算することができる．2 つのグラフ間の最小差分を求める問題は，最大共通サブグラフ問題[9] と見なすことができる．最大共通サブグラフ問題のアルゴリズムは数多く提案されている[10] ものの，一般的に数十ノード程度の問題しか解けない．よって，一般的な高位設計における CDFG に適用することは難しい．

そこで本稿では設計記述を CDFG ではなく単純な命令の列で表現し，文字列の比較に基づいて差分を計算する手法を提案する．命令列はソフトウェアコンパイラ技術を用いることで生成することが可能であり，また容易に CDFG に変換可能である．EC は局所的かつ小規模な変更であることと入力記述は逐次的な記述であることの 2 つの特徴を利用して，まず大部分の命令間の等価性を効率の良いアルゴリズムで求めておき，差分となる可能性のある部分に関しては厳密な手法で計算する．まず第 2 節では本稿で使用する内部表現を説明し，問題の定式化を行う．次に第 3 節と第 4 節で提案手法について説明する．第 5 節では提案手法の実装および初期実験結果について述べる．

```
for(i = 2; i < 16; i++) {
    f[i] = f[i - 1] + f[i - 2];
}
```

(a) 変更前の設計記述

```
for(i = 3; i < 16; i++) {
    f[i] = f[i - 3] + f[i - 2] + f[i - 1];
}
```

(b) 変更後の設計記述

```
 1:   n1 = (lbl) label
 2:   n2 = (nil) jmp n3
 3:   n3 = (lbl) label
 4:   n4 = (i32) phi (i32) 2, n1, n14, n7
 5:   n5 =  (i1) lte n4, (i32) 16
 6:   n6 = (nil) br n5, n16, n7
 7:   n7 = (lbl) label
 8:   n8 = (i32) sub n4, (i32) 1
 9:   n9 = (i32) load n8
10:  n10 = (i32) sub n4, (i32) 2
11:  n11 = (i32) load n10
12:  n12 = (i32) add n9, n11
13:  n13 = (nil) store n12, n4
14:  n14 = (i32) add n4, (i32) 1
15:  n15 = (nil) jmp n3
16:  n16 = (lbl) label
17:  n17 = (nil) ret
```

(c) 変更前の設計記述の内部表現

```
 1:   n1 = (lbl) label
 2:   n2 = (nil) jmp n3
 3:   n3 = (lbl) label
 4:   n4 = (i32) phi (i32) 3, n1, n17, n7
 5:   n5 =  (i1) lte n4, (i32) 16
 6:   n6 = (nil) br n5, n19, n7
 7:   n7 = (lbl) label
 8:   n8 = (i32) sub n4, (i32) 3
 9:   n9 = (i32) load n8
10:  n10 = (i32) sub n4, (i32) 2
11:  n11 = (i32) load n10
12:  n12 = (i32) sub n4, (i32) 1
13:  n13 = (i32) load n12
14:  n14 = (i32) add n9, n11
15:  n15 = (i32) add n14, n13
16:  n16 = (nil) store n15, n4
17:  n17 = (i32) add n4, (i32) 1
18:  n18 = (nil) jmp n3
19:  n19 = (lbl) label
20:  n20 = (nil) ret
```

(d) 変更後の設計記述の内部表現

図 2 本稿で用いる設計変更の例題．下線は一致する命令がないことを示す．8 命令が一致している．

2. 問題定式化

高位合成では C 言語などといった高位言語で記述された動作記述をコントロールデータフローグラフ (CDFG) 等のグラフ表現に変換した後に処理を行う．この処理の過程においてソフトウェアコンパイラと同様にまず中間言語を用いた内部表現に一旦変換してからグラフを生成することが一般的である．本稿ではこの中間言語表現 (以下，内部表現) 間の差分を計算することを考える．

本稿で説明のために用いる設計変更例を図 2 に示す．本稿における中間言語としてはコンパイラの分野で広く利用されている LLVM[11] に似た表現を採用した．図 2(a) は C 言語で記述された元の設計記述であり，図 2(b) は設計変更後の C 言語記述である．また，図 2(c) と

Vol.2011-SLDM-149 No.21
Vol.2011-EMB-20 No.21
2011/3/18

(d) はそれぞれ図 2(a) と (b) を変換した内部表現である．まず，本稿で想定する中間言語について説明する．内部表現は逐次的に実行される命令列 $(i_1, ..., i_m)$ として表現される．ここで命令は以下のように定義される．命令 $i_k =< N(i_k), O(i_k), T(i_k), I_1(i_k), I_2(i_k), ... >$ は固有の名前 $N(i_k)$ を持ち，その名前を持つ変数への代入である．$O(i_k)$ は命令の種類を表しており，その種類としては加減算 (add, sub)， 乗算 (mul)，メモリ読み書き (load, store)，比較 (lte, eq,...) などがある．また制御構造も一貫した方法で表現するために，ラベル (label)，ジャンプ (jmp)，ブランチ (br) といった命令も存在する．上述の通り命令名が固有であるため，この表現は各変数への代入が高々一回となる静的単一代入 (SSA) 形式[12] となる．そのため，ϕ 関数を表現する phi 命令を持っている．T_k は命令の結果の型を表しており，32 ビット整数型 (i32)，ブール型 (i1)，ラベル型 (lbl) などがある．また，メモリ書き出しやジャンプ命令などの実際には結果に型がない命令の型は nil となる．$I_j(i_k)$ は命令の j 番目の引数を表す．

ここで図 2(c)(d) に示すような「N_k = (T_k) O_k I_{k_1}, I_{k_2}, ...」の形式の文字列を命令の文字列表現とする．各命令の左横にある数字はその命令の順番であり，i_k の k に相当する．また，命令列の文字列表現は各行が命令の文字列表現となるように命令の文字列表現を並べたものである．このとき，2 つの命令列の文字列表現を各行で比較することで差分を求めることができる．図 2(c)(d) において，下線が引かれた命令が差分に対応し，下線がない命令は一致する命令が存在する．このように本稿では命令の文字列表現によって命令間の等価性を定義する．以降特に指定がない限り，命令の文字列表現と命令列の文字列表現を単に命令と命令列と表記する．また，一致する命令が存在する命令，つまり下線がない命令をマッチ命令，一致する命令がない命令，つまり下線が引かれた命令を非マッチ命令と呼ぶ．設計変更前の非マッチ命令はインクリメンタル高位合成によって取り除かれ，設計変更後の非マッチ命令は設計に追加される．

図 2(c)(d) を見て容易にわかるように，いくつかの命令は命令名の不一致の理由のみによって非マッチ命令となっている．例えば，変更前の n16 と変更後の n19 は命令名を同じにするだけでマッチ命令となる．また，そのようにすることで変更前の n6 と変更後の n6 も引数が同じになりマッチ命令となることがわかる．このように命令名を適切に割り当てることで差分を最小化することができる．よって，本稿では命令列間の差分が最小となるような命令名の割り当てを求めることが目的である．この問題は第 4 節で説明する最大独立集合問題に帰着して解くことが可能であるが，この問題は NP 困難であることが知られており，大規模な命令列に適用することは難しい．EC では設計変更部分は局所的かつ小規模であるので，この部分のみを最大独立集合問題として解くことが望ましい．よって，次節では容易に等価性が判別可能な部分に関してマッチ命令を求める手法を提案する．残りの非マッチ命令に対して最大独立集合問題を解くことによって全体の差分を求めることが可能である．このようにすることで大規模な設計に対しても差分が小規模であれば差分を求めることが可能である．

3. 命令名の正規化に基づく命令マッチ発見手法

最長共通部分列 (Longest Common Sequence) とは，2 つの列の共通部分列の中で最も長いものである．提案手法は命令列の文字列表現の最長共通部分列を求める問題と見なすことができる．最長共通部分列問題は有限長の場合は動的計画法で多項式時間で求めることが可能であり[13]，UNIX の diff コマンドで用いられているアルゴリズムでもある．図 2(c)(d) において，下線のない命令の列が最長共通部分列である．この例題から容易にわかるように，多くの命令は命令名を入れ替えるだけで等価にすることができる．提案手法では，まず命令名に依存しない方法で最長共通部分列を求め，その後差分を最小にするように命令名を再度割り当てる．

以下，手法を説明する．まず最初に，各命令 T_k の名前を $\$T_k$ とする．つまり同じ型を持つ変数はすべて同じ名前を持つことになる．この変形を命令名の正規化と呼ぶ．図 2(c)(d) に対して命令名の正規化を行った内部表現を図 3(a)(b) に示す． このように命令名の正規化を行うことによって，楽観的な最長共通部分列を求めることができる．このとき，すべての命令が固有な名前を持っていないため，必ずしもマッチ命令が最終的なマッチ命令とならないことに注意されたい．次に，マッチした命令の各組に対して順番に命令名の再割り当てを行う．非マッチ命令に関してはそれぞれ別の名前を割り当てる．このようにして名前の再割り当てを行った内部表現を図 4(a)(b) に示す． この段階ではすべての命令が固有な名前を持っており，よって命令間のマッチ関係は正しいものとなっている．ここで，図 2 よりも差分が大きくなっていることが確認できる．しかしながら，本手法は非マッチ命令に対して異なる名前を割り当てているため，マッチ命令は引数に非マッチ命令を含んでいない．一方で図 2 の内部表現ではいくつかのマッチ命令の引数が非マッチ命令を含んでいるため，以降の処理で非マッチ命令の名前が変更された場合に非マッチ命令となってしまう可能性がある．よって，本手法で求めたマッチ命令はこの段階で確定され，以降の処理ではこれらのマッチ命令を無視することができる．大規模な設計で小規模な設計変更が行われた際には本手法で大部分のマッチ命令を求めることが可能である．

Vol.2011-SLDM-149 No.21
Vol.2011-EMB-20 No.21
2011/3/18

```
 1:  $lbl = (lbl) label
 2:  $nil = (nil) jmp $lbl
 3:  $lbl = (lbl) label
 4:  $i32 = (i32) phi (i32) 2,$lbl,$i32,$lbl
 5:   $i1 =  (i1) lte $i32, (i32) 16
 6:  $nil = (nil) br $i1, $lbl, $lbl
 7:  $lbl = (lbl) label
 8:  $i32 = (i32) sub $i32, (i32) 1
 9:  $i32 = (i32) load $i32
10:  $i32 = (i32) sub $i32, (i32) 2
11:  $i32 = (i32) load $i32
12:  $i32 = (i32) add $i32, $i32
13:  $nil = (nil) store $i32, $i32
14:  $i32 = (i32) add $i32, (i32) 1
15:  $nil = (nil) jmp $lbl
16:  $lbl = (lbl) label
17:  $nil = (nil) ret
```

(a) 設計変更前

```
 1:  $lbl = (lbl) label
 2:  $nil = (nil) jmp $lbl
 3:  $lbl = (lbl) label
 4:  $i32 = (i32) phi (i32) 3,$lbl,$i32,$lbl
 5:   $i1 =  (i1) lte $i32, (i32) 16
 6:  $nil = (nil) br $i1, $lbl, $lbl
 7:  $lbl = (lbl) label
 8:  $i32 = (i32) sub $i32, (i32) 3
 9:  $i32 = (i32) load $i32
10:  $i32 = (i32) sub $i32, (i32) 2
11:  $i32 = (i32) load $i32
12:  $i32 = (i32) sub $i32, (i32) 1
13:  $i32 = (i32) load $i32
14:  $i32 = (i32) add $i32, $i32
15:  $i32 = (i32) add $i32, $i32
16:  $nil = (nil) store $i32, $i32
17:  $i32 = (i32) add $i32, (i32) 1
18:  $nil = (nil) jmp $lbl
19:  $lbl = (lbl) label
20:  $nil = (nil) ret
```

(b) 設計変更後

図 3 命令名の正規化後の内部表現

```
 1:   n1 = (lbl) label
 2:   n2 = (nil) jmp n3
 3:   n3 = (lbl) label
 4:   n4 = (i32) phi (i32) 2, n1, n19, n9
 5:   n5 =  (i1) lte n4, (i32) 16
 6:   n8 = (nil) br n5, n27, n9
 7:   n9 = (lbl) label
 8:  n10 = (i32) sub n4, (i32) 1
 9:  n11 = (i32) load n10
10:  n12 = (i32) sub n4, (i32) 2
11:  n16 = (i32) load n12
12:  n17 = (i32) add n11, n16
13:  n18 = (nil) store n17, n4
14:  n19 = (i32) add n4, (i32) 1
15:  n26 = (nil) jmp n3
16:  n27 = (lbl) label
17:  n28 = (nil) ret
```

(a) 設計変更前

```
 1:   n1 = (lbl) label
 2:   n2 = (nil) jmp n3
 3:   n3 = (lbl) label
 4:   n6 = (i32) phi (i32) 3, n1, n25, n9
 5:   n7 =  (i1) lte n6, (i32) 16
 6:   n8 = (nil) br n7, n27, n9
 7:   n9 = (lbl) label
 8:  n13 = (i32) sub n6, (i32) 3
 9:  n14 = (i32) load n13
10:  n15 = (i32) sub n6, (i32) 2
11:  n16 = (i32) load n15
12:  n20 = (i32) sub n6, (i32) 1
13:  n21 = (i32) load n20
14:  n22 = (i32) add n14, n16
15:  n23 = (i32) add n22, n21
16:  n24 = (nil) store n23, n6
17:  n25 = (i32) add n6, (i32) 1
18:  n26 = (nil) jmp n3
19:  n27 = (lbl) label
20:  n28 = (nil) ret
```

(b) 設計変更後

図 4 命令名の割り当て後の内部表現．6 命令が一致している．

4. 最大独立集合問題に基づく最小差分計算手法

本節では前節で非マッチ命令となった命令列に対して名前を再度割り当て，最長共通部分列を求める手法を提案する．具体的には最長共通部分列問題を競合グラフ (conflict graph) の最大独立集合問題に帰着する．前節の手法を適用することなく，すべての命令名を正規化した命令列に適用することも可能であるが，前節で述べたように最大独立集合問題は NP 困難であるため大規模な問題への適用は難しい．

競合グラフ $G(V,E)$ は無向グラフであり，各節点 $v_{p,q} \in V$ は 2 命令 i_p, i_q のマッチに対応する．また，各辺 $e \in E$ はその辺が結ぶ 2 節点に対応する 2 つのマッチが同時に成立できないことを表す．最大独立集合とは，節点の部分集合 $V' \in V$ のうち V' の節点間に辺が存在しないもので $|V'|$ が最大になるようなものである．競合グラフの最大独立集合は命令列の最長共通部分列に対応する．

以下，競合グラフの作成手法を説明する．まず命令列内のすべての非マッチ命令を正規化する．図 4 の非マッチ命令の正規化後の内部表現を図 5 に示す．次に同じ文字列となる命令の組 (i_p, i_q) をすべて求め，各節点 $v_{p,q}$ とする．最後に，以下の条件を一つでも満たす場合に 2 節点 $(v_{p,q}, v_{s,t})$ の間に辺を作成する．

制約 *1* (命令順の競合) 変更前の命令 i_p と i_s の順序と変更後の命令 i_q と i_t の順序が異なる場合，つまり

$$((p > s) \wedge (q < t)) \vee ((p < s) \wedge (q > t)) \tag{1}$$

の場合にはどちらかの命令マッチが共通部分列に含まれない．よって両方のマッチを同時に満たすことはできない．

制約 *2* (命令一致の競合) ある命令は高々 *1* つの命令のみとマッチする．つまり

$$((p \equiv s) \wedge (q \neq t)) \vee ((p \neq s) \wedge (q \equiv t)) \tag{2}$$

の場合にはある命令が *2* つの命令とマッチする．よって両方のマッチを同時に満たすことはできない．

Vol.2011-SLDM-149 No.21
Vol.2011-EMB-20 No.21
2011/3/18

```
 1:    n1 = (lbl) label
 2:    n2 = (nil) jmp n3
 3:    n3 = (lbl) label
 4:  $i32 = (i32) phi (i32) 2, n1, $i32, n8
 5:   $i1 =  (i1) lte $i32, (i32) 16
 6:  $nil = (nil) br $i1, n21, n8
 7:    n8 = (lbl) label
 8:  $i32 = (i32) sub $i32, (i32) 1
 9:  $i32 = (i32) load $i32
10:  $i32 = (i32) sub $i32, (i32) 2
11:  $nil = (i32) load $i32



12:  $i32 = (i32) add $i32, $i32
13:  $nil = (nil) store $i32, $i32
14:  $i32 = (i32) add $i32, (i32) 1
15:   n20 = (nil) jmp n3
16:   n21 = (lbl) label
17:   n22 = (nil) ret
```

(a) 設計変更前

```
 1:    n1 = (lbl) label
 2:    n2 = (nil) jmp n3
 3:    n3 = (lbl) label
 4:  $i32 = (i32) phi (i32) 3, n1, $i32, n8
 5:   $i1 =  (i1) lte $i32, (i32) 16
 6:  $nil = (nil) br $i1, n21, n8
 7:    n8 = (lbl) label
 8:  $i32 = (i32) sub $i32, (i32) 3
 9:  $i32 = (i32) load $i32
10:  $i32 = (i32) sub $i32, (i32) 2
11:  $i32 = (i32) load $i32
12:  $i32 = (i32) sub $i32, (i32) 1
13:  $i32 = (i32) load $i32
14:  $i32 = (i32) add $i32, $i32
15:  $i32 = (i32) add $i32, $i32
16:  $nil = (nil) store $i32, $i32
17:  $i32 = (i32) add $i32, (i32) 1
18:   n20 = (nil) jmp n3
19:   n21 = (lbl) label
20:   n22 = (nil) ret
```

(b) 設計変更後

図 5 命令名の一部再正規化後の内部表現

```
 n1 = (lbl) label
 n2 = (nil) jmp n3
 n3 = (lbl) label
 n4 = (i32) phi (i32) 2, n1, n14, n7
 n5 =  (i1) lte n4, (i32) 16
 n6 = (nil) br n5, n16, n7
 n7 = (lbl) label
 n8 = (i32) sub n4, (i32) 1
 n9 = (i32) load n8
n10 = (i32) sub n4, (i32) 2
n11 = (i32) load n10



n12 = (i32) add n9, n11
n13 = (nil) store n12, n4
n14 = (i32) add n4, (i32) 1
n15 = (nil) jmp n3
n16 = (lbl) label
n17 = (nil) ret
```

(a) 設計変更前

```
 n1 = (lbl) label
 n2 = (nil) jmp n3
 n3 = (lbl) label
 n4 = (i32) phi (i32) 3, n1, n14, n7
 n5 =  (i1) lte n4, (i32) 16
 n6 = (nil) br n5, n16, n7
 n7 = (lbl) label
 n8 = (i32) sub n4, (i32) 3
 n9 = (i32) load n8
n10 = (i32) sub n4, (i32) 2
n11 = (i32) load n10
n18 = (i32) sub n4, (i32) 1
n19 = (i32) load n18
n20 = (i32) add n9, n11
n12 = (i32) add n20, n19
n13 = (nil) store n12, n4
n14 = (i32) add n4, (i32) 1
n15 = (nil) jmp n3
n16 = (lbl) label
n17 = (nil) ret
```

(b) 設計変更後

図 6 最大独立集合問題に基づく手法の適用後の内部表現．15 命令が一致している．

制約 *3* (引数の競合) ある *2* つの命令がマッチするとき，それらの引数もまたマッチしなくてはならない．このときマッチした引数と他のマッチが上記の競合を満たす場合には，そのため両方のマッチを同時に満たすことはできない．

最大独立集合問題は様々なアルゴリズムが知られているが，本稿では整数線形計画問題に帰着して解いている．制約式は以下のようにして生成する．各節点 $v_{p,q} \in V$ に対応する 2 値変数を $x_{p,q} \in \{0,1\}$ とする．

$$\begin{aligned} &\textbf{maximize } \textstyle\sum_{p,q} x_{p,q} \\ &\textbf{subject to} \\ &\qquad 0 \le x_{p,q} \le 1 \qquad \forall v_{p,q} \in V \\ &\qquad x_{p,q} + x_{s,t} \le 1 \qquad \forall (v_{p,q}, v_{s,t}) \in E \end{aligned} \tag{3}$$

本手法を適用した後の最終的な内部表現を図 6 に示す．図 2 ではマッチ命令数が 8 であったのが図 6 では 15 に増えており，差分が小さくなっていることが確認できる．

5. 初期実験結果

本稿で提案した手法を我々が開発している Cyneum 合成・最適化フレームワーク上に実装した．Cyneum にはすでにインクリメンタル高位合成手法[2] を実装しており，元の設計に対応する CDFG と設計変更後の CDFG の差分を入力することでインクリメンタルに合成を行うことが可能となっている．入力 C プログラムを解析し，命令列を生成する処理には LLVM コンパイラ・インフラストラクチャ[11] を用いている．設計記述および提案手法で求めた差分は内部表現から CDFG に変換されてインクリメンタル高位合成の入力となる．最長共通部分列を求めるアルゴリズムとしては，広く用いられている実用的な手法[14] を用いた．また，最大独立集合問題を解く際に使用する整数線形計画法のソルバとしては Gurobi Optimizer[15] を用いた．今回の実験では図 2(a)(b) に示す例題を用いて実際に最小差分を計算した後にインクリメンタル高位合成を行い所望の動作が得られることを検証した．例題が小規模であるため，計算時間も 1 秒以下であった．

Vol.2011-SLDM-149 No.21
Vol.2011-EMB-20 No.21
2011/3/18

6. まとめと今後の課題

本稿ではインクリメンタル高位合成に向けた設計記述間の最小差分を求める手法を提案した．高位合成の内部表現として一般的に用いられている CDFG 間の差分を求めるのではなく，逐次的な命令列を文字列として表現しその比較を行うことにより，効率的な差分計算が可能となっている．EC では一般的に設計記述の変更は局所的かつ小規模であることに着目し，まず単純な方法で大部分の等価性を求めておき，残りの部分を厳密な手法で求めることで更なる効率化を実現している．初期実験では実際の例題でインクリメンタル合成を行い，手法の正当性を確認した．

提案手法では最長共通部分列に基づいているため，命令の順序に従い命令間の対応を求めている．しかしながら，依存関係のない命令を入れ替えることで差分を更に小さくできる可能性がある．このような差分計算を実現するためには，第 4 節の手法において競合グラフの辺の作成手法を改良する必要があり，これは今後の課題とする．また今後は実用的な例題に対して様々な設計変更を適用し，提案手法の効率や有効性についても評価を行う予定である．

謝　　辞

本研究は文部科学省 科学研究費補助金 若手研究 (B) (課題番号: 22760245) の助成を受けたものである．

参 考 文 献

1) L. Lavagno, A. Kondratyev, Y. Watanabe, Q. Zhu, M. Fujii, M. Tatesawa, and N. Nakayama, “Incremental high-level synthesis,” in *Proc. IEEE Asia and South Pacific Design Automation Conf.*, 2010, pp. 701–706.
2) 小野 翔平, 吉田 浩章, 藤田 昌宏, “発見的手法に基づくスケーラブルなインクリメンタル高位合成,” 電子情報通信学会技術研究報告, vol. 110, no. 316, pp. 13-18, 2010 年 11 月.
3) S.C. Prasad, P.Anirudhan, and P.Bosshart, “A system for incremental synthesis to gate-level and reoptimization following RTL design changes,” in *Proc. ACM/IEEE Design Automation Conf.*, 1994, pp. 441–446.
4) D. Brand, A. Drumm, S. Kundu, and P. Narain, “Incremental synthesis,” in *Proc. IEEE Int. Conf. on Computer-Aided Design*, Nov. 1994, p.18.
5) G. Swamy, S. Rajamani, C. Lennard, and R. Brayton, “Minimal logic re-synthesis for engineering change,” in *Proc. IEEE Int. Symp. Circuits and Systems*, vol.3, Jun. 1997, pp. 1596–1599.
6) J.Cong and M.Sarrafzadeh, “Incremental physical design,” in *Proc. ACM Int. Symp. Physical Design*, 2000, pp. 84–92.
7) FAAD2, `http://www.audiocoding.com/faad2.html`.
8) 吉田 浩章, 藤田 昌宏, “製造後機能修正可能な高電力効率アクセラレータの高位設計手法,” 情報処理学会 DA シンポジウム 2010 論文集, pp. 45–50, 2010 年 9 月.
9) M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.
10) V. Kann, “On the approximability of the maximum common subgraph problem,” in *Proc. Annual Symposium on Theoretical Aspects of Computer Science*, 1992, pp. 377–388.
11) C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Proc. IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*, May 2004, p.75.
12) S.S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
13) E.Myers, “An O(ND) difference algorithm and its variations,” *Algorithmica*, vol.1, no.2, pp. 251–266, 1986.
14) J.W. Hunt and T.G. Szymanski, “A fast algorithm for computing longest common subsequences,” *Commun. ACM*, vol.20, pp. 350–353, May 1977.
15) *Gurobi Optimizer Reference Manual, Version 3.0*. Gurobi Optimization, Inc., 2010.