社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

潜在的多様性を考慮したプログラマブルハードウェアの高位合成手法

吉田　浩章[†,††]　藤田　昌宏[†,††]

† 東京大学 システム設計教育研究センター (VDEC)
〒 113–8656 東京都文京区本郷 7–3–1
†† 科学技術振興機構 戦略的創造研究推進事業 CREST
E-mail: †hiroaki@cad.t.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

あらまし　SoC の開発コスト増大と開発期間短縮に伴い，製造故障や設計誤りの製造後修正を可能とする技術の重要性が増している．最近になり，制御部分をプログラマブルにすることにより製造後修正を可能とするプログラマブルハードウェアアクセラレータが注目されている．既存設計手法では初期設計に最適化されたデータパスとプログラマブル制御回路を合成するため，修正後の性能については考慮していない．本稿では，製造後に起こりうる潜在的な設計修正後の性能歩留まりを最大化するようなプログラマブルハードウェアの合成手法を提案する．例題を用いた計算機実験により，小さい面積オーバーヘッドで性能歩留まりを大幅に向上可能であることを示した．

キーワード　プログラマブルハードウェア，製造後修正，高位合成，性能歩留まり

High-Level Synthesis of Programmable Hardware Accelerators Considering Potential Varieties

Hiroaki YOSHIDA[†,††] and Masahiro FUJITA[†,††]

† VLSI Design and Education Center(VDEC), University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo, 113–8656, Japan
†† CREST, Japan Science and Technology Agency
E-mail: †hiroaki@cad.t.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

Abstract　Recently, programmable hardware accelerators have attracted more attention as an enabling solution for post-silicon engineering change, manufacturing defect tolerance, and efficient hardware reuse. Since existing techniques synthesize for a specific application, the synthesized hardware may not satisfy the performance goal under potential varieties such as engineering changes and manufacturing defects. This paper proposes a synthesis methodology of programmable hardware accelerators which maximizes the performance yield under potential varieties. Experimental results demonstrate that our methodology can improve the performance yield significantly with a small area increase.

Key words　Programmable hardware accelerators, engineering changes, high-level synthesis, performance yield

1　は じ め に

近年の SoC 開発コストの増大と開発期間の短縮に伴い，高位合成を利用した設計手法の導入が進んでいる．高位合成によって生成されたハードウェアアクセラレータは高性能と高効率の両立が求められる様々な分野において利用されている．一方で製造故障や設計誤りによる製造後修正がますます重要となっており，製造後修正を可能とするプログラマブルハードウェアアクセラレータが注目されている [1], [2].

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行う engineering change (EC) 手法が用いられてきた．EC は全体回路に対して十分小さいため，この目的には FPGA 等の高い柔軟性を提供するプログラマブル素子は面積オーバーヘッドも大きく向いていない．現在では EC は配線の入れ替えやスペアセルを用いた回路レベルの変更であるものの，高位合成手法の普及に進むにつれて高位における EC が重要になってくると思われる．また，プログラマビリティは製造故障に対する耐性向上にも有効である．EC の場合とは異なり，製造故障では一部が使用不能となるため，あらかじめ一定の冗長性を用意しておくことが必要となる．例えばプロセッサは高いプログラマビリティを提供しているが，必ずしも故障耐性が高いわけではない．さらに，プログラマビリティを持たせることで製品内の複数アプリケーション間や製品の複数世代間でのハードウェアの再利用が可能となる．このように，用途に

よっては必ずしも完全なプログラマビリティは不必要であり，用途に応じて適切なプログラマビリティや冗長性を持たせる手法の確立が重要である．

このような背景の下で，プログラマブルハードウェアアクセラレータの高位合成手法が提案されている．No-Instruction-Set Computer (NISC) [1] はマイクロコード方式のプログラマブル制御器とカスタムデータパスからなるプログラマブルハードウェアである．単一アプリケーションに対して最適なカスタムデータパスを生成する手法も提案されている [3]．この手法では，まず初期スケジューリング結果に基づいて十分な数の FU を割り当てる．次に，FU の使用頻度に応じて FU を段階的に削減することでカスタムデータパスを生成する．プログラマブルループアクセラレータ (PLA) [2] は NISC と類似しているものの，MOV 命令やグローバルバス，ポート交換などアーキテクチャ的にプログラマビリティを向上させる工夫をしている．これらの手法は与えられた単一アプリケーションの設計記述に最適化されたデータパスを生成する．したがって，合成されたハードウェアは設計変更後に必ずしも性能制約を満たすとは限らない．本稿では，製造後に起こりうる潜在的な設計変更に対して性能歩留まりを最大にするデータパスを合成する手法を提案する．設計の潜在的な多様性を表現するために多様性集合という概念を用いる．多様性集合はランダムサンプリングした事象の集合であり，各事象が確率を持つような確率空間である．また，実際に設計変更が行われた際にプログラマブルハードウェアの制御プログラムを生成するコンパイル手法も提案する．

2 提案手法

2.1 アクセラレータの基本構成

本稿で対象とするプログラマブルハードウェアアクセラレータ (以下，アクセラレータ) の基本構成を図 1 に示す．アクセラレータは機能ユニット (FU)，FU 間を接続する配線部分，およびプログラマブル制御器からなる．各 FU はあらかじめ決められた 1 つ以上の種類の演算を行うことが可能であり，制御信号によって演算の種類を決定する．典型的な FU としては，ALU や乗算器 (MUL)，比較器 (CMP)，バレルシフタ (SHFT) などがある．レジスタファイルやローカルストア，定数生成器などのメモリ素子では，メモリ素子の書き込みポートや読み込みポートをそれぞれ FU とみなす．このようにすることで，合成およびコンパイルの際にメモリ素子へのアクセスを算術演算と同様に扱うことが可能である．各 FU の入力は他の FU の出力もしくはマルチプレクサ (MUX) の出力と接続される．同様に各 FU の出力も他の FU の入力やマルチプレクサの入力に接続される．マルチプレクサの入力は FU の出力と接続されている．マルチプレクサの制御信号によって各 FU は入力信号を選択する．レジスタファイル (RF) はレジスタの集合であり，複数の書き込みポート (RFI) および読み込みポート (RFO) を持つ．レジスタは局所変数値の格納に使用される．各 RF ポートの制御信号によって，どのレジスタにアクセスするかを決定する．定数生成器 (CG) はあらかじめ決められた定数を出力することが可能であり，レジスタファイル同様，読み込みポート (CGO) への制御信号に基づいて定数を生成する．ローカルストア (LS) は主に配列やグローバル変数値を格納する RAM であり，また外部とのデータのやり取りにも使用される．ローカルストアも書き込みポート (LSI) と読み込みポート (LSO) を持つが，他のメモリ素子とは異なり，ポートはアドレスとデータの 2 つの信号線を

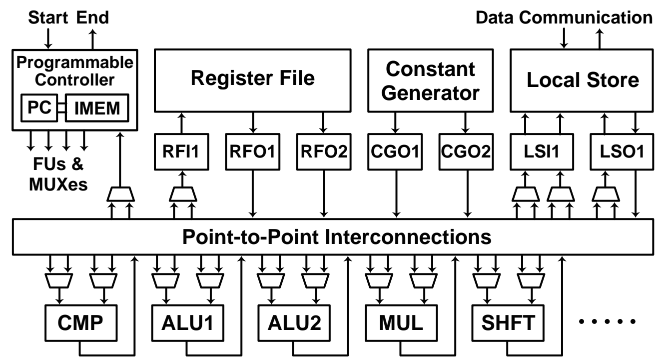


図 1: プログラマブルハードウェアアクセラレータの基本構成

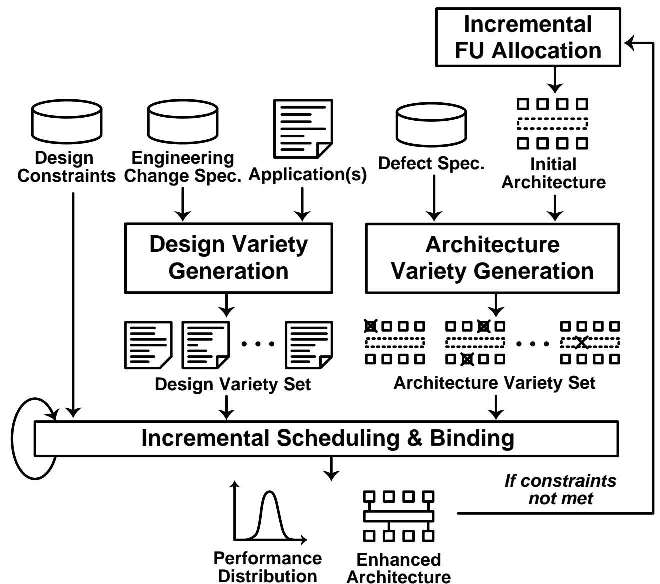


図 2: プログラマブルハードウェアアクセラレータの合成の流れ

持つ．ローカルストアの書き込みポートは書き込みイネーブルの制御入力を持つ．

プログラマブル制御器 (以下，制御器) は現在の状態に基づいて FU や MUX への制御信号を生成する．制御器はデータパスが生成した 1 ビットの制御信号に基づいて次状態を決定する．本稿では，制御器の実装方式として制御信号を格納する命令 RAM(IMEM) と現在の状態を表すプログラムカウンタ (PC) からなる水平型マイクロコード方式を採用した．RAM の各アドレスには対応する状態の制御信号および次状態の対が格納されている．一般的にマイクロコード方式の制御器は結線論理方式に比べ面積が大きくなる．この面積オーバーヘッドはコード圧縮手法 [4] などを利用することで，ある程度削減することが可能である．

2.2 潜在的多様性を考慮したデータパス合成手法

潜在的多様性を考慮したデータパス合成の流れを図 2 に示す．高位合成と同様に，C 言語などで記述された高位設計記述と設計制約を入力とする．また，潜在的な EC の度合いを指定する EC 仕様と故障率を指定する故障仕様も入力として与えることが可能である．

まず，設計記述を動作させるために必要な最小構成の FU をデータパスに割り当てる．この時点では各 FU 間は接続されていない．次に，EC 仕様に基づいて初期設計にランダムに変更を加えることで設計多様性集合を生成する．また，故障仕様に

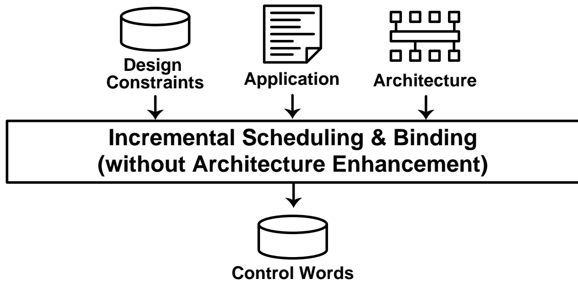


図 3: 制御プログラムのコンパイルの流れ

基づいて部分的に故障したデータパスを複数生成し，データパス多様性集合を得る．これらの多様性集合の生成手法については第 4 節 で詳しく説明する．設計多様性集合とデータパス多様性集合の各要素対に対して，第 3 節で説明するインクリメンタルスケジューリング・バインディングを繰り返し適用する．各繰り返しにおいて，データパス間の接続が必要に応じて追加されていく．その後，各要素対の合成結果が性能を満たしていたかどうかの割合から，性能歩留まりが求まる．次に，これまでの合成における FU の使用頻度に基づいて，データパスに新たな FU を割り当てる．この手続きを性能歩留まりがこれ以上向上しなくなるまで繰り返すことによって性能歩留まりを最大化する．

2.3 制御プログラムのコンパイル手法

実際に設計の変更が行われた際には，アクセラレータ内の制御器の制御プログラムを変更することによって修正を行う．ここでは設計変更が与えられた際に制御プログラムをコンパイルする手法を説明する．図 3 に示すように，高位設計記述と設計制約，対象とするデータパス構成を入力とする．また，初期設計のスケジューリングおよびバインディング結果が与えられた場合には，変更箇所のみを再スケジューリング・バインディングすることが可能である．これは，周辺回路との通信などの都合により初期回路のタイミングを変更したくない場合などに有用である．コンパイル手法は合成と同じインクリメンタルスケジューリング・バインディング手法を用いる．合成と異なる点は，スケジューリング・バインディング時に配線を追加しない点である．手法の詳細については次節で説明する．

3 インクリメンタルスケジューリング・バインディング手法

インクリメンタルスケジューリング・バインディング手法は第 2 節で提案した合成およびコンパイル手法の主要技術であるため，本節で手続きを詳細に説明する．まず，入力の高位設計記述を解析してコントロールデータフローグラフ (CDFG) を構築する．CDFG 内で表現される式はそれぞれ静的単一代入 (SSA) 形式になっているものとする．CDFG はコントロールフローグラフ (CFG) $G_C = (V_C, E_C)$ とデータフローグラフ (DFG) $G_D = (V_D, E_D)$ からなる．CFG は基本ブロックを表す制御ノードの集合 V_C と制御ノード間の制御フローを表す制御エッジの集合 E_C からなる．ここで，基本ブロックとは制御の変更を伴わない一連の演算を表す．DFG は演算ノードの集合 V_D と演算ノード間のデータ依存関係を表すデータエッジの集合 E_D からなる．スケジュール $S : V_D \to U$ は演算ノードの集合から制御ステップの集合への写像として定義される．データパス $A = (F, I)$

procedure Schedule-And-Bind(G_C, G_D, M, S, B)
 // $G_C = (V_C, E_C)$ is the input control flow graph
 // $G_D = (V_D, E_D)$ is the input data flow graph
 // $T \subseteq V_D$ is the set of the target operation nodes
 // $S[n]$ is the schedule which maps each operation node n to a step
 // $B[n]$ is the bind which maps each operation node n to an FU
1: **for all** basic block $BB \in V_C$ **do**
2: $BB \leftarrow (BB \cap T)$
3: SMS-Sort(BB)
4: **for all** operation node $n \in BB$, taken in sorted order **do**
5: $V \leftarrow$ Available-Slots(n)
6: $d \leftarrow$ Scan-Direction(n)
7: **for all** step $s \in V$, taken in order of d **do**
8: $S[n] \leftarrow s$
9: Bind(n)
10: **if** $B[n] \neq nil$ **then break**
11: **end for**
12: **if** $B[n] = nil$ **then**
13: $S[n] \leftarrow$ New-Step(n, d)
14: Bind(n)
15: **end if**
16: **end for**
17: **end for**
18: Assign-Registers(G_D, S)
19: Generate-Control-Words(S, G)

図 4: インクリメンタルスケジューリング・バインディングの手続き

は FU の集合 F と FU 間の配線の集合 I からなる．FU 割り当て $B : V_D \to F$ は演算ノードの集合から FU の集合への写像として定義される．インクリメンタルスケジューリング・バインディングの対象となる演算ノードの集合 $T \subseteq V_D$ を対象ノードと呼ぶ．ここで，残りの演算ノード $(V_D - T)$ のスケジュールと FU 割り当ては与えられているものとする．

提案手法はスケジューリングとバインディングを並行して行う．具体的には，各演算ノード $n \in V_D$ に対して，まず n をどの制御ステップで実行するかを決定 (スケジューリング) した後，どの FU で n を実行するかを決定 (バインディング) する．図 4 に示すスケジューリングの手続きは swing modulo スケジューリング [5] に基づいている．ここで，本手法は swing modulo スケジューリングに基づいているがソフトウェアパイプライン化を行うわけではないことに注意されたい．しかしながら，提案手法をソフトウェアパイプライン化に拡張することは容易である．各基本ブロック BB に対して，swing modulo スケジューリング手法を用いて演算ノードの集合 $(BB \cap T)$ のスケジューリング順を決定する (3 行目，手続き SMS-Sort())．スケジューリングの品質はスケジューリング順に大きく依存する．swing modulo スケジューリングはクリティカルパス上のノードを最優先とし，次に変数の寿命を最小にするようにスケジューリング順を決定する．決定されたスケジューリング順に各演算ノード n を選択し (4 行目)，以下の処理を繰り返す．手続き Available-Slots() によって n がスケジュール可能な制御ステップの集合 S を決定する (5 行目)．手続き Scan-Direction() によって決定 (6 行目) した順で S の各制御ステップを選択し，バインディングを試みる (9 行目)．割り当てが見つからない場合には，新しい制御ステップを挿入し (New-Step())，再びバインディングを行う (12-15 行目)．

```
procedure Bind(n)
    // n is the node to be bound
    // mode is either synthesis or compilation
    // A = (F, I) is the architecture
 1: G ← Available-FUs(F, s, n)
 2: Sort-FUs(G)
 3: for all functional unit f in G, taken in sorted order do
 4:     B[n] ← f
 5:     success ← true
 6:     for all neighboring node m of n do
 7:         g ← B[m]
 8:         if g = nil then continue
 9:         p ← Bind-Path(A, f, g)
10:         if p = nil and mode = synthesis then
11:             I ← I ∩ New-Interconnects(A, f, g)
12:             p ← Bind-Path(A, f, g)
13:         end if
14:         if p = nil then
15:             success ← false
16:             break
17:         end if
18:     end for
19:     if success then return // Binding found.
20:     else Undo-New-Interconnects(I)
21: end for
22: B[n] ← nil // No binding found.
```

図 5: バインディングの手続き

次にレジスタ割り当て Assign-Registers() を行い，各変数をレジスタファイル内のレジスタに割り当てる．本稿ではすべての局所変数は必ずレジスタに割り当てられる，つまりメモリスピルは行わない．提案手法では CDFG 内に表現される式が SSA 形式の場合に最適性を保証するレジスタ割り当てアルゴリズム [6] を採用した．最後に手続き Generate-Control-Words() によってスケジューリングおよびバインディング結果から制御プログラムを生成する．

図 5 にバインディングの手続きを示す．まず，演算ノード n に割り当て可能な FU の集合を手続き Available-FUs() で求める．その後，手続き Sort-FUs() によって FU の集合をバインディングのコストに基づいてソートする．演算ノード n を FU f に割り当てるコストは，割り当てた際に追加する必要のある配線のコストとする．ソートされた順に n を FU f に割り当て，f に隣接する演算ノード m に対応する FU g に接続する．ここで演算ノード m と n の間にデータエッジが存在している場合に 2 ノードが隣接していると呼ぶ．隣接ノード m がすでに FU g に割り当てられている場合には，手続き Bind-Path() によって f と g の間を接続する．このとき，f と g の間の接続には，配線，マルチプレクサ，レジスタポートを組み合わせる．もし m と n が異なる制御ステップにスケジュールされている場合には，演算結果をレジスタに格納するようにバインディングする．もし m と n の間にそのようなパスがない場合には，手続き New-Interconnects() によって新しい配線を導入してバインディングを再び行う．コンパイル時には New-Connection() は実行されない．この部分が唯一合成とコンパイルで異なる点である．それでもパスが見つからない場合には，直前の繰り返しで導入された配線をすべて除去し (Undo-New-Interconnects())，次の FU 候補 $f' \in G$ を n に割り当てる．

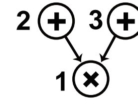
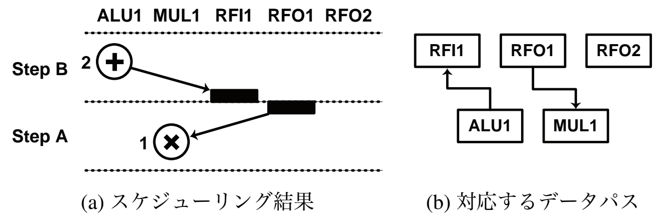
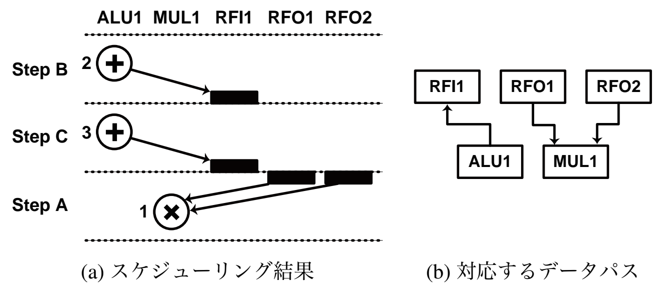


図 6: データフローグラフの例．ノードの横の数字はスケジューリング順を表す．



(a) スケジューリング結果 (b) 対応するデータパス

図 7: ノード 1 とノード 2 のスケジューリング・バインディング結果．黒い長方形はレジスタを表す．



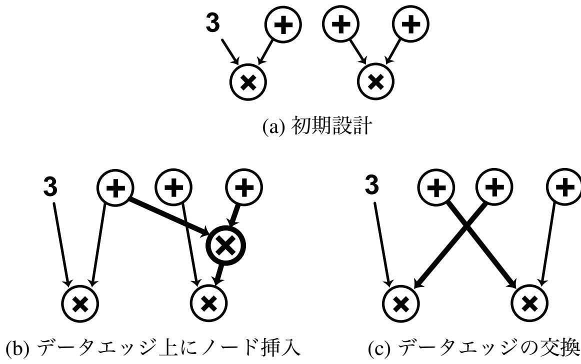
(a) スケジューリング結果 (b) 対応するデータパス

図 8: ノード 3 のスケジューリング・バインディング結果．制御ステップ C が新たに導入されており，RFO2 と MUL1 の間に新しい配線が追加されている．

図 6 に示す DFG を例として，提案手法を説明する．まず，3 つのノードを swing modulo スケジューリング手法を用いてソートする．図中の数字がスケジューリング順を表している．図 7 (a) に最初の 2 つのノードのスケジューリング・バインディング結果を示す．ノード 1 には MUL1 が割り当てられており，片方の入力はレジスタファイル読み出しポート RFO1 に接続されている．ノード 2 には ALU1 が割り当てられており，出力はレジスタファイル書き出しポート RFI1 に接続されている．対応するデータパス (図 7 (b)) は ALU1 と RFI1 の間と RFO1 と MUL1 の間に配線が存在する．次に，ノード 3 を制御ステップ B にバインディングする．制御ステップ B の ALU はすでに割り当てられているため，新しい制御ステップ C を挿入し，ALU1 をノード 3 に割り当てる．ノード 3 とノード 1 の間にはデータエッジが存在しており，かつ 2 ノードは異なる制御ステップにスケジュールされているので，各ノードはレジスタファイルのポートに接続する．ALU1 と RFI1 の間には配線があるが，RFO2 から MUL1 への配線は存在しないため，新たに配線を追加することで最終的なデータパスを生成する．

4 多様性集合の生成手法

多様性集合とは異なる事象の集合であり，各事象が確率を持つような確率空間である．ここで各事象を変異形と呼ぶ．本稿では設計多様性集合とデータパス多様性集合の 2 つの多様性集合を扱う．各事象の確率は性能歩留まりの計算に使用される．本節では，潜在的な engineeing change による設計多様性集合の生成手法と製造故障によるデータパス多様性集合の生成手法について説明する．



(a) 初期設計

(b) データエッジ上にノード挿入　(c) データエッジの交換

図 9:　Engineering change による設計変更のモデル

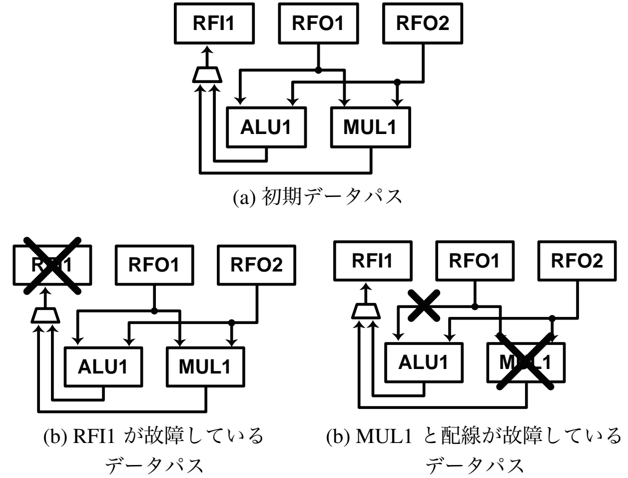
4.1　Engineering Change による設計多様性集合の生成

本稿では仕様変更や設計誤りによる設計変更を engineering change と呼ぶ．高位設計における EC では，初期設計記述の一部を変更する．一般的に初期設計の段階ではどのような EC が行われるかは不明であり，また可能な EC の方法は非常に多数であるため網羅的に求めることは現実的に不可能である．そのため，本稿ではランダムサンプリングによって潜在的な EC 後の設計記述の集合を求め，これを設計多様性集合とする．ここでは，初期設計の DFG に対して繰り返しグラフ変形を適用することで潜在的な EC をモデルする．文献 [7] では同様の方法で EC をモデルしているが，彼らは EC に対する耐性の評価のためにこれを用いている．また，本稿では CFG の変形は考慮しない．例えば `if` 文の `then` 節と `else` 節を交換するような CFG における変更は制御プログラムの単純な変更によって対応可能であるためである．

EC による設計変更のモデルとして，(i) データエッジ上にノード挿入，(ii) データエッジの交換，の 2 つのグラフ変形を考える．この 2 つの変形は高位記述においては，(i) ある式中に新しい演算を挿入する，(ii) 2 つの変数参照を交換する，という設計変更に対応する．最初の変形では，まず DFG 中からランダムにエッジを選択し，ランダムに選択した種類の演算ノードを挿入する．新しいノードが複数入力を持つ場合には，適当な数のノードをランダムに選択して入力とする．各変形は同確率で起こるものとする．図 9 に 2 つのグラフ変形の例を示す．初期設計の DFG に対して，2 つの変形をランダムに繰り返し適用することで EC 後の DFG が求まる．変形の適用回数は EC 仕様によって指定される．

4.2　製造故障によるデータパス多様性集合の生成

VLSI ダイ上で製造故障が起こる確率は単位面積当たり一定であり，この確率は故障仕様として与えられているとする．また，FU や配線が占有する領域内に 1 つでも故障が存在すれば，対応する FU や配線は使用不可能であるとする．つまり，FU や配線の故障確率はその面積に比例する．配線に関しては合成段階ではレイアウトが不明なため，ファンアウト数で見積もる．データパス多様性集合 D は異なる部分故障データパスの集合である．部分故障データパス $d \in D$ は無故障の初期データパス中の FU や配線の一部が故障しているデータパスである．部分故障データパスは初期データパスから決められた個数以下の FU や配線を故障させることで生成される．製造故障の箇所が 1〜2 箇所であれば，すべての部分故障データパスを生成することも可能である．各部分故障データパスは故障部分の面積に応じて確率が与えられる．図 10 にデータパス多様性集合の例を示す．



(a) 初期データパス

(b) RFI1 が故障しているデータパス　(b) MUL1 と配線が故障しているデータパス

図 10:　データパス多様性集合の例

表 1:　設計例題の概要

設計	演算ノード数	説明
`idct`	286	8x8 逆離散コサイン変換
`mpeg_pred`	369	MPEG-1 予測関数
`bdist2`	182	MPEG-2 `bdist2()` 関数
`bubble_sort`	55	バブルソート
`adpcm_decoder`	178	ADPCM デコーダ

5　計算機実験結果

本稿で提案した手法を我々が開発している SORA 合成・最適化フレームワーク上に実装した．入力 C プログラムを解析し，SSA 形式の CDFG を構築する処理には LLVM コンパイラ・インフラストラクチャ[8] を用いている．C プログラムが関数呼び出しを含む場合には，関数インライン化を用いて単一関数を生成している．また，SystemC 言語による記述を入力とすることも可能となっており，各モジュールに対して 1 つのアクセラレータを合成する．このとき，複数モジュール間はローカルストアを通じて通信する．合成されたアクセラレータの RTL 記述は Verilog HDL 言語で出力することが可能となっており，制御プログラムも様々な形式で出力可能である．

提案合成手法の有効性を示すため，以下の計算機実験を行った．例題設計として表 1 に示す 5 つの C 言語で記述された設計を用いた．比較対象として，多様性を考慮しないデータパスを合成した．これは一般的な高位合成ツールが生成するデータパスと同等である．FU やマルチプレクサ，メモリ素子，配線の面積は Rohm 0.18μm テクノロジを用いて見積もった．

まず，第 4.1 節で説明した手法によって engineering change による設計多様性集合を生成した．生成に際しては，演算ノードの増加が全体の 3%以下になるように制約を与え，また各設計に対して 100 個の異なる変異形を生成した．これを考慮することで，engineering change に対する耐性を持つデータパスを合成した．次に，多様性を考慮しないデータパスと考慮したデータパスのそれぞれに対して，上と同じ方法で生成した設計多様性集合を入力としてコンパイルを行い，実行ステップ数を 100 個求めた．比較結果を表 2 に示す．表中の「実行ステップ数」において,「初期」は初期設計の実行ステップ数,「平均」,「標準偏差」,「最悪」はそれぞれ設計多様性集合を入力として求まった 100 個の実行ステップ数の平均値，標準偏差，最悪値で

表 2: Engineering change に対する性能歩留まりの比較

回路	多様性考慮なし (既存手法)						多様性を考慮 (提案手法)					
	面積	実行ステップ数				性能	面積		実行ステップ数			性能
	合計 [mm^2]	初期	平均	標準偏差	最悪	歩留まり [%]	合計 [mm^2]	増加率 [%]	平均	標準偏差	最悪	歩留まり [%]
idct	2.159	74	77.9	2.3	88.0	31.0	2.348	8.8	74.7	2.8	85.0	77.0
mpeg_pred	3.071	160	166.3	2.3	174.0	39.0	3.116	1.5	161.6	2.4	167.0	93.0
bdist2	3.728	81	83.2	1.5	89.0	84.0	3.802	2.0	82.0	1.5	88.0	97.0
bubble_sort	0.772	29	31.0	0.2	33.0	0.0	0.773	0.1	30.0	0.0	30.0	100.0
adpcm_decoder	3.154	89	92.0	0.3	93.0	96.0	3.197	1.4	91.3	0.8	92.0	100.0
Average						**50.0**		**2.8**				**93.4**

表 3: 製造故障に対する性能歩留まりの比較

回路	多様性考慮なし (既存手法)						多様性を考慮 (提案手法)					
	面積	実行ステップ数				性能	面積		実行ステップ数			性能
	合計 [mm^2]	初期	平均	標準偏差	最悪	歩留まり [%]	合計 [mm^2]	増加率 [%]	平均	標準偏差	最悪	歩留まり [%]
idct	2.159	74	81.5	11.6	109.0	0.0	2.254	4.4	72.4	0.8	74.0	100.0
mpeg_pred	3.071	160	165.7	3.8	170.0	0.0	3.079	2.6	156.0	0.0	156.0	100.0
bdist2	3.728	81	81.9	0.3	82.0	7.9	3.794	1.8	81.0	0.0	81.0	100.0
bubble_sort	0.772	29	—	—	—	0.0	0.776	0.5	28.0	0.0	28.0	100.0
adpcm_decoder	3.154	89	—	—	—	0.0	3.159	0.2	89.0	0.0	89.0	100.0
Average						**1.6**		**1.9**				**100.0**

ある．性能歩留まりは 100 個の実行ステップ数のうち初期設計の実行ステップ数の 103%以内になった割合である．全体として，2.8%の面積オーバーヘッドで 43.4%の性能歩留まりの向上を確認した．

次に，第 4.2 節で説明した手法を用いて部分故障データパスの集合を生成した．ここでは，FU や配線の故障は高々1 箇所しか起こらないと仮定して，すべての可能な部分故障データパスを生成した．これを考慮することで，製造故障に対する耐性を持つデータパスを合成した．EC の場合と同様，多様性を考慮しないデータパスと考慮したデータパスのそれぞれに対して，上と同じ方法で生成した部分故障データパスを入力としてコンパイルを行って，実行ステップ数を求めた．比較結果を表 3 に示す．ここで，性能歩留まりは部分故障データパスの実行ステップが初期データパスの実行ステップ以内になる割合である．表中において，bubble_sort と adpcm_decoder の「—」は対応するデータパスが最小数の FU しか備えておらず故障時にアプリケーションの実行が不能であったため実行ステップが求まらなかったことを示している．この時の性能歩留まりは 0%としてある．全体として，1.9%の面積オーバーヘッドで 98.4%の性能歩留まりの向上を確認した．これらの結果によって，提案手法の有効性を確認した．

6 結　　論

本稿ではプログラマブルハードウェアアクセラレータの高位合成手法を提案した．既存手法とは異なり，提案手法は engineering change や製造故障修正などといった潜在的な多様性に対して耐性を持つデータパスを合成する．潜在的な多様性はランダムサンプリングに基づいて生成する多様性集合によって表現される．提案手法はこの多様性集合を入力として，性能歩留まりを最大化するデータパスを合成する．また，本稿では実際に設計変更が行われた際に制御プログラムを生成するコンパイル手法を提案した．例題を用いた計算機実験では，提案手法が小さい面積オーバーヘッドで性能歩留まりを大幅に向上可能であることを示した．

文　　献

[1] M. Reshadi and D. Gajski, “A cycle-accurate compilation algorithm for custom pipelined datapaths,” in *Proc. IEEE/ACM Int. Symp. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sep. 2005, pp. 21–16.

[2] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke, “Bridging the computation gap between programmable processors and hardwired accelerators,” in *Proc. Int. Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2009, pp. 313–322.

[3] J. Trajkovic and D. Gajski, “Automatic data path generation from C code for custom processors,” in *Proc. IFIP Int. Embedded Systems Symp. (IESS)*, May 2007, pp. 379–384.

[4] B. Gorjiara and D. Gajski, “FPGA-friendly code compression for horizontal microcoded custom IPs,” in *Proc. ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb. 2007, pp. 108–115.

[5] J. Llosa, “Swing modulo scheduling: A lifetime-sensitive approach,” in *Proc. IEEE Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, Oct. 1996, pp. 80–87.

[6] P. Brisk, F. Dabiri, R. Jafari, and M. Sarrafzadeh, “Optimal register sharing for high-level synthesis of SSA form programs,” *IEEE Trans. Computer-Aided Design*, vol. 25, no. 5, pp. 772–779, May 2006.

[7] K. Fan, H. Park, M. Kudlur, and S. Mahlke, “Modulo scheduling for highly customized datapaths to increase hardware reusability,” in *Proc. IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*, Apr. 2008, pp. 124–133.

[8] 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.