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.

不定サイクル演算に対応した分散制御における動的演算バインディング

山下　真司†　　石浦菜岐佐†

† 関西学院大学 理工学部 〒669–1337 兵庫県三田市学園 2-1

あらまし　オペランドや実行時の状況によって実行サイクル数が変動する不定サイクル演算器を含む回路を効率的に制御するための分散制御方式において, 演算を実行するタイミングや順序だけでなく, 演算器も動的に決定できる方式を提案する. Del Barrio の分散制御方式は, 演算器毎に状態機械を設けて演算の実行制御を行うものであり, 回路規模の増加は少ないが, 同一演算器に割り当てられた演算の実行順序は変更できない. Pilato の分散制御方式は, 演算毎にその実行の可否を表わす状態変数を設けて演算の実行を制御するため, 演算の実行順序を動的に変更できるが, 各演算を実行する演算器は固定されていた. 本稿の制御方式は, Pilato の方式を拡張することにより, 各演算が実行可能になった時点で空いている演算器を使用できるようにするものである. 評価実験を行なったところ, 回路規模や遅延は増加するが, 実行サイクル数を約 6〜10% 削減できることが確認できた.

キーワード　高位合成, 分散制御, 動的演算バインディング

Dynamic Operation Binding in Distributed Controller for Supporting Functional Units with Variable Latency

Shinji YAMASHITA† and Nagisa ISHIURA†

† Kwansei Gakuin University, 2-1 Gakuen, Sanda, Hyogo, 669–1337, Japan

Abstract　This article presents a new distributed method for controlling circuits with variable latency units, which can dynamically assign operations to functional units as well as it adjusts timing and order of operation execution. The Del Barrio's method controls each functional unit with a dedicated state machine and adaptively changes the timing of operation execution, but the execution order of the operations for each unit is fixed. The Pilato's method allows the change of execution order by assigning a state variable to each operation to express its executability, but each operation is bound to a fixed functional unit. Our method extends the Pilato's method so that operations may be processed using either one of free units as soon as they are executable. A preliminary experiment shows that execution cycles are reduced by 6 to 10%, though there is significant increase in circuit size and delay.

Key words　High-Level Synthesis, Distributed Controller, Dynamic Operation Binding

1. はじめに

近年の半導体技術の発展に伴い, 益々大規模で複雑な集積回路が実装できるようになっている. その設計を効率化する手段として, プログラミング言語などの動作記述を入力としてハードウェアの設計記述を自動生成する高位合成技術 [1] の実用化が進められている.

従来の高位合成では, 全ての演算が固定のサイクル数で完了することを前提として, 演算を実行するタイミングや演算器を決定する. しかし, オペランドの値や利用状況に依存して実行時にサイクル数が変化する不定サイクル演算も存在する. 回路中にこのような演算器が含まれる場合, 固定されたタイミングで演算の実行を制御すると, どうしても無駄な待ちが生じてしまう.

これに対し, 演算器からの完了信号をもとに制御を実行時に変更する可変スケジューリング [2] が提案されている. 可変スケジューリングは, 可能な実行サイクル数の全組み合わせに対する制御パターンを網羅した状態機械で演算の実行を制御する方式であり, 各演算の実行タイミングを動的に変更する効率的な制御が可能になる. しかし, 状態数が著しく増加し, 合成されるハードウェアの回路規模や遅延が許容できないほど大きくなる恐れがある.

この問題を解決する手法として, 分散制御があげられる. 分散制御は, 独立に動く複数の状態機械で演算の制御を行うものであり, 制御回路の規模を抑えつつ, 不定サイクル演算の実行を効果的に制御することができる.

分散制御の方式としては, Del Barrio の方式 [4] と Pilato の方式 [5] が提案されている. Del Barrio の分散制御方式は, 演算器毎に状態機械を割り当て, その演算器に割り当てられた演算の実行を制御する方式である. 異なる演算器にバインディングされている演算は, 独立に動作タイミングを変更できるため, 集中制御方式のように制御回路が大きくなるのを防ぐことができる. しかし, 同一演算器に割り当てられている演算は, 設計時に決められた順序でしか実行ができないため, 後の演算が先に実行可能になっても前の演算が終わるまでそれを実行できない. Pilato の分散制御方式は, 各演算に対して実行の可否を表す 1 ビットの状態変数を割り当てて制御する方式であり, 同一演算器に割り当てられている演算の実行順序も動的に変更することができる. しかし, 設計時に決められた演算器以外で演算を実行することはできないため, 同一演算器に割り当てられた複数の演算が同時に実行可能になっても, 一方の演算は順番待ちをしなければならない.

そこで本研究では, 演算の実行タイミングと実行順序だけでなく, 実行を行う演算器も実行時に決定できる制御方式を提案する. 本制御手法は Pilato の分散制御手法を, 各演算を複数の演算器にバインディングできるように拡張したものである. 評価を行なったところ, 動的演算バインディングを適用した結果, Pilato の制御方式に比べ回路規模 (LUT 数) は 2〜26%増加し, 遅延は 17〜38%増加したが, 平均サイクル数は 6〜10%削減できた.

2. 不定サイクル演算に対応した分散制御

2.1 不定サイクル演算

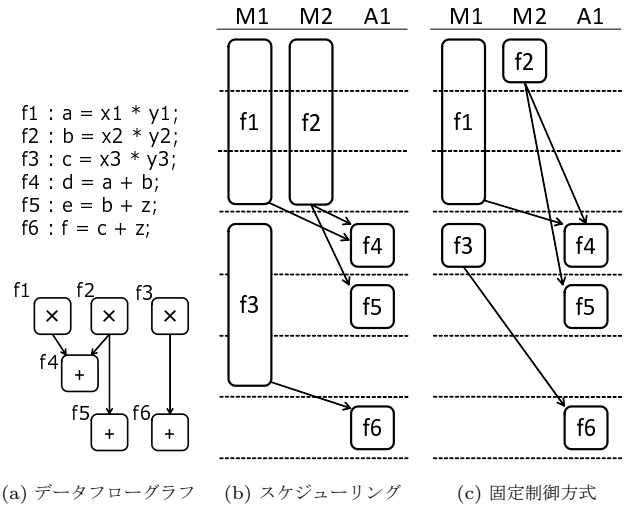
不定サイクル演算とは, オペランドの値や利用状況に依存して実行時にサイクル数が変化する演算である. 例えば, 乗除算では, オペランドの値によっては少ないサイクル数で計算を完了するものがある. キャッシュを持つメモリシステムへのアクセスでは, キャッシュにヒットするかどうかに依存してサイクル数が変化する. また, 回路の経年劣化に伴って遅延が増えることによりサイクル数が増加する場合もある.

図 1(a) のようなデータフローグラフが与えられたとき, 加算を 1 サイクルで行う加算器 A1 と乗算を 1〜3 サイクルで行う乗算器 M1, M2 でこれを実行する場合を考える. (b) は, 乗算に 3 サイクルを要するとしてスケジューリングした例である. ここで, 演算 f2, f3 が 1 サイクルで完了したとする. 従来の制御手法 (c) では, 演算の f2, f3 の後続の演算の実行を早めることはできない. また, 乗算器 M2 は 2 サイクル目以降空いているにもかかわらず, 演算 f3 を実行することはできない.

2.2 可変スケジューリング

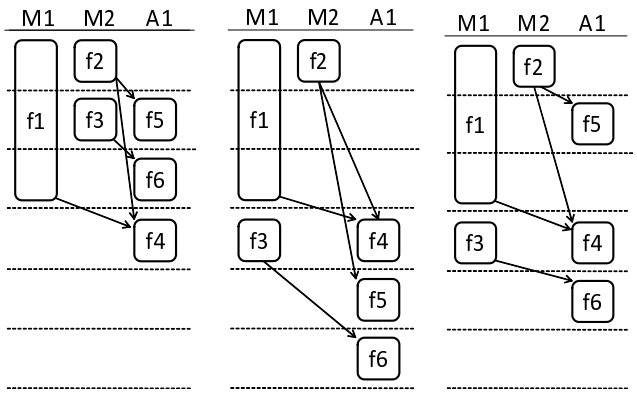
可変スケジューリング [2] は, 演算器からの完了信号を基に各演算の実行タイミングを動的に変更することにより演算を効率的に実行し, 全体の実行サイクル数を減少させるものである.

図 1 の例に対し, 可変スケジューリングによって制御を行なった例を図 2(a) に示す. この例は, 図 1(c) と同様に演算 f1 が 3 サイクル, 演算 f2, f3 が 1 サイクルで完了している. 演算 f5 は依存する演算 f2 が 1 サイクル目に完了するため 2 サイクル目に実行することができる. また, 演算 f3 は演算 f2 を完了した乗算器 M2 で 2 サイクル目に実行できるため, 演算 f6 は 3 サイク



(a) データフローグラフ (b) スケジューリング (c) 固定制御方式

図 1 不定サイクル演算を考慮しない制御



(a) 可変スケジューリング (b) Del Barrio の方式 (c) Pilato の方式

図 2 不定サイクル演算を考慮した制御

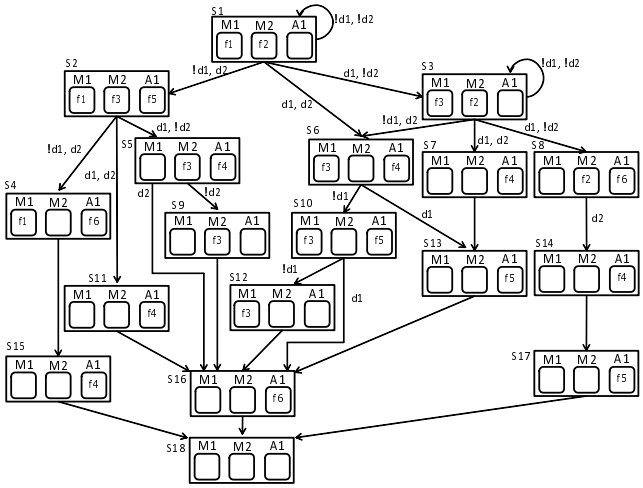


図 3 可変スケジューリングの状態機械の例

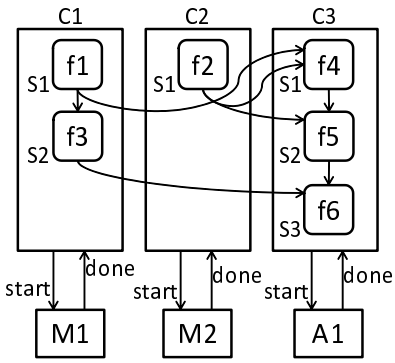


図 4　Del Barrio の分散制御方式

ル目に実行することができる. 結果, この例では合計 4 サイクルで実行を完了することができる.

このような制御は, あらかじめ全演算の可能な全ての実行サイクル数の組み合わせに対応する状態機械を構築することにより実現できる. 図 1 の例に対する可変スケジューリングの状態機械の例を図 3 に示す. この状態機械は乗算器 M1, M2 の完了信号 d1, d2 の真偽に基づいて状態を遷移させる. 例えば, 状態 S1 は乗算器 M1 で演算 f1 を, 乗算器 M2 で演算 f2 を実行し, d1 と d2 の真偽によって状態 S1, S2, S3, S6 のいずれかに遷移する.

しかし, 入力の動作記述によっては状態数が膨大になる可能性がある [3]. 状態数の増加は制御回路やデータパスのマルチプレクサの増大につながり, 面積と遅延が許容できないほど大きくなる恐れがある.

2.3　Del Barrio の分散制御方式

Del Barrio の分散制御方式は, 1 つの演算器に 1 つの状態機械を割り当て, それぞれの状態機械で演算器に割り当てられた演算の実行を制御するものである. 図 1 の例に対する, Del Barrio の分散制御の例を図 4 に示す. 状態機械 C1, C2, C3 はそれぞれ演算器 M1, M2, A1 で実行する演算に対応する状態を持っている. これらの状態機械は, 図 3 の状態機械を演算器毎に分解したものと見ることができる. これによって, 制御のための状態機械を大きくすることなく, 異なる演算器にバインディングされてた演算の動作タイミングを独立に変更することができる.

しかし, 同一演算器にバインディングされている演算は, 設計時に決めた順序でしか実行できない. 図 1 の例に対し, Del Barrio の方式で制御を行なった例を図 2(b) に示す. 図 1(c) に比べると演算 f6 の実行を 1 サイクル早めることができるため, 合計 6 サイクルで実行が完了する. しかし, 演算 f5 は 2 サイクル目に実行可能になっているにもかかわらず, 演算 f4 より先に実行することはできないため, 5 サイクル目まで実行できない.

2.4　Pilato の分散制御方式

Pilato の分散制御方式は, 各演算に対して実行の可否を表す 1 ビットの状態変数を割り当てて制御する方式である. その状態変数が 1 になり, 割り当てられた演算器が空いていれば演算の実行を行う. これにより, 異なる演算器にバインディングされている演算だけでなく, 同一演算器にバインディングされている演算同士でも実行順序を動的に決定することができる. 図 1 の例に対し, Pilato の分散制御方式で制御を行なった例を図

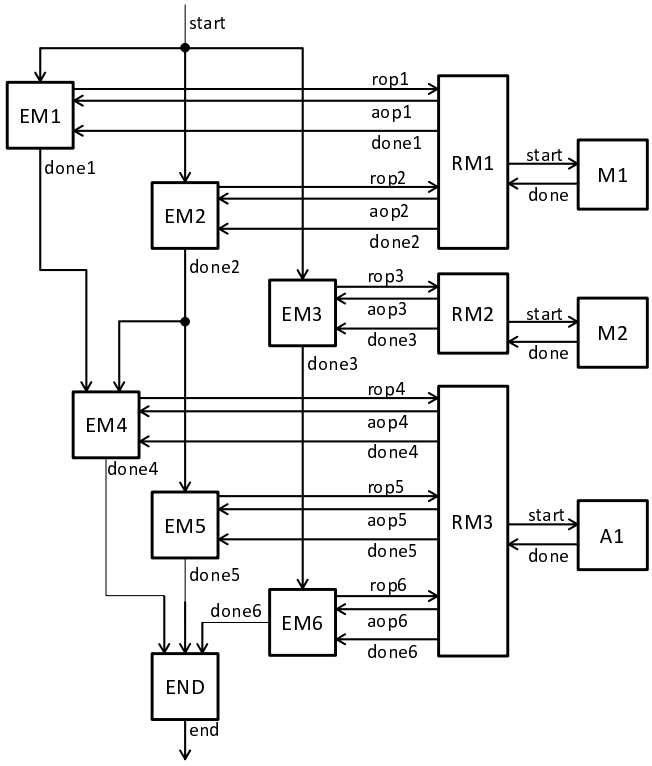


図 5　Pilato の分散制御方式における制御回路の構成

2(c) に示す. Del Barrio の方式 (b) では演算 f5 は f4 の後にしか実行できなかったが, Pilato の方式では演算 f4 を追い越して 2 サイクル目に演算 f5 を実行できるため, 合計 5 サイクルで実行を完了することができる.

Pilato の分散制御方式は, 演算毎に生成される Execution Manager (EM) と演算器毎に生成される Resource Manager (RM) によって制御を行う. 図 1 の例に対し, Pilato の分散制御方式の制御回路を構成した例を図 5 に示す. この例では, EM1〜EM6 は演算 f1〜f6 に対応しており, RM1, RM2, RM3 はそれぞれ乗算器 M1, M2, 加算器 A1 に対応している. この回路全体の開始信号 (start) が 1 になると, 演算 f1, f2, f3 に対応した EM1, EM2, EM3 が活性化し, 各演算の実行要求信号 (rop1, rop2, rop3) の送信を開始する. rop1 と rop2 を受け取った RM1 は, 優先順位に従って演算 f1 を乗算器 M1 で実行する. RM1 は演算 f1 の実行開始信号 (aop1) を EM1 とデータパスに送信し, aop1 を受け取った EM1 は rop1 の送信を終了する. M1 による演算 f1 の実行が完了すると RM1 は EM1 に演算 f1 の完了信号 (done1) を送信する. done1 を受け取った EM1 は演算 f1 に依存する演算 f4 に done1 を送信する. EM4 は done1 と done2 の両方を受け取ると活性化し, rop4 の送信を開始, 以下同様に行う. この回路全体の終了制御機械 (END) は done4, done5, done6 を全て受け取ると終了信号 (end) を出力する.

しかし, Pilato の方式では, 演算を実行する演算器を変更することはできない. 図 2(c) の例では, 演算 f3 が実行可能で, 2 サイクル目には乗算器 M2 が空いているにも関わらず, 演算 f3 は乗算器 M1 でしか実行できないため, 4 サイクル目まで実行が開始できない.

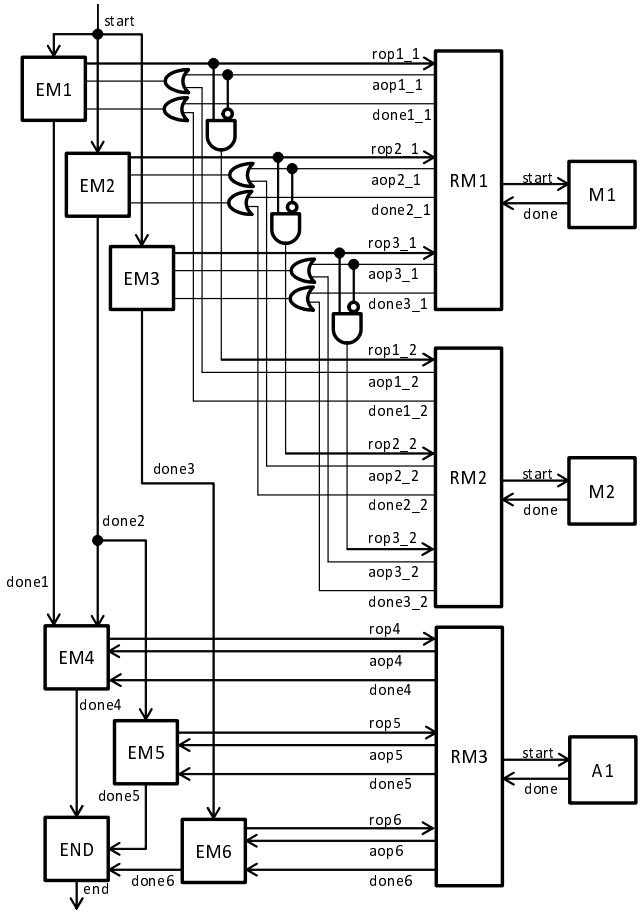


図 6　本稿の制御回路の構成

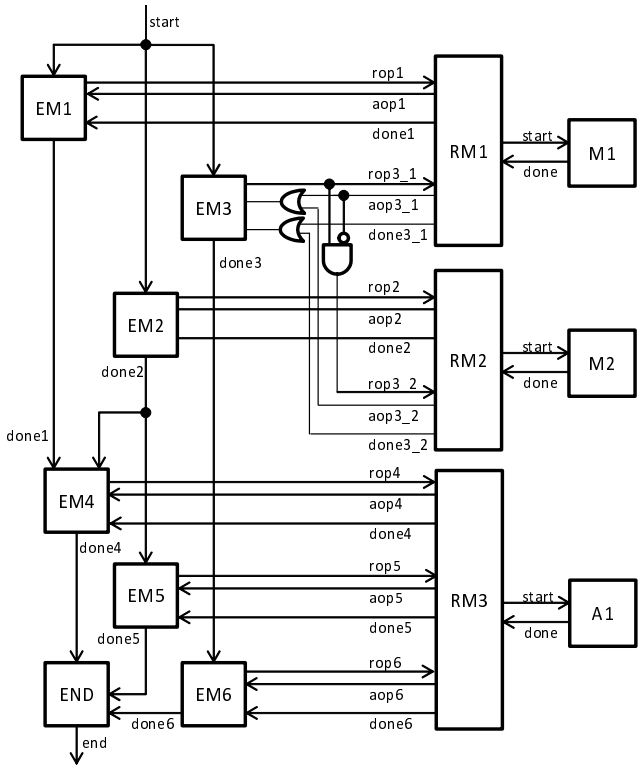


図 8　割り当てを限定した場合の制御回路の構成

3. 分散制御における動的演算バインディング

3.1 動的な演算バインディング

本稿では, 演算の実行順序だけでなく, 演算を行う演算器も動的に決定する分散制御方式を提案する. 本手法では, 各演算を単一の演算器ではなく演算器の集合にバインディングし, 実行時の状況に応じて演算器を選択できるようにする.

図 1 の例に提案手法を適用し, 演算 f3 を乗算器 M1, M2 のどちらでも実行できるように制御した場合, 可変スケジューリングによる制御例 (図 2(a)) と同様に, 演算 f3 が乗算器 M2 で実行でき, 合計 4 サイクルで実行を完了することができる.

3.2 制御回路の構成

動的演算バインディングの制御回路は Pilato の分散制御方式の RM を複数連結することにより実現できる.

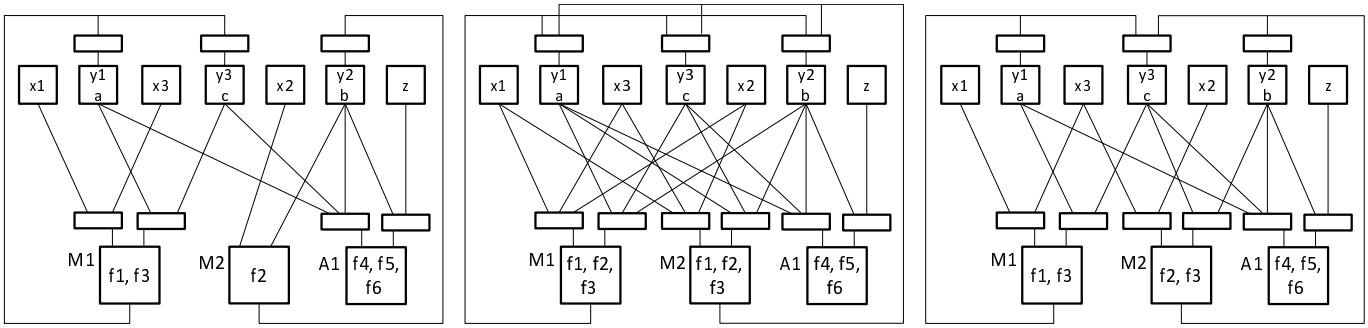
図 5 の制御回路を動的演算バインディングに拡張した構成例を図 6 に示す. ここでは, 乗算 f1, f2, f3 はいずれも乗算器 M1, M2 の両方で実行可能であるとする. EM1〜EM6 は Pilato の方式と同じである. また, RM1, RM2, RM3 は乗算器 M1, M2, 加算器 A1 に対応するものであり, 機能的には Pilato の方式と同じである. 本制御方式が Pilato の制御方式と異なる点は, f1 が M1 と M2 の両方で実行可能であるため, EM1 から RM1 と RM2 両方への接続があることである. まず, EM1 から RM1 への実行要求信号 (rop1_1) が 1 になり, 演算器 M1 が空いていれば実行開始信号 aop1_1 を 1 にして実行を開始する. 空いていなければ実行開始信号 aop1_1 は 0 になるので, RM2 への実行要求信号 (rop1_2) が 1 になる. 演算器 M2 が空いていれば実行開始信号 aop1_2 が 1 になり実行を開始する. 実行開始信号 aop1_1 か aop1_2 のどちらか一方が 1 になれば, EM1 への実行開始信号 (aop1) は 1 になる. また, 完了信号 done1_1 か done1_2 のどちらか一方が 1 になれば, EM1 への完了信号 (done1) は 1 になる. EM2〜EM6 も同様である.

本制御手法によるデータパスの構成を図 7 に示す. (a) は Pilato の方式 (図 5) のデータパス部分であり, (b) は本稿の方式 (図 6) のデータパス部分である. Pilato の方式では, 演算 f1, f3 が乗算器 M1 で, 演算 f2 が乗算器 M2 で実行されるため, レジスタから M1 への入力は 4 つ, レジスタから M2 への入力は 2 つである. 本稿の方式では, 演算 f1, f2, f3 が両方の乗算器で実行可能なため, M1, M2 共にレジスタからの入力が 6 つになり, 結線やマルチプレクサが増加する.

3.3 演算器割り当ての限定

1 つの演算を多くの演算器で実行できるようにすると, 制御回路およびデータパスの規模と遅延が著しく増加する可能性がある. これは, 各演算を実行可能な演算器集合を限定することにより, ある程度防ぐことができる. 例えば, 図 6 および図 7(b) の回路構成では, 演算 f1, f2, f3 全てを乗算器 M1 と M2 のどちらでも実行可能としていたが, f1 は M1 で, f2 は M2 で, f3 のみ M1 と M2 の両方で実行可能とすれば, 図 8 および図 7(c) のように, 実行サイクル数を増やすことなく制御回路およびデータパスを簡単化できる.

この例の場合には実行サイクル数は変化しないが, 一般には性能と回路規模のトレードオフになると考えられる.



(a) Pilato の方式　(b) 動的演算バインディング　(c) 割り当てを限定した動的演算バインディング

図 7　データパスの構成

表 1　実 験 結 果

(a) 2 次元正方行列の積 (乗算器 3 個, 加算器 2 個)

	平均サイクル数	LUT 数	遅延 (ns)
Pilato [4]	8.01 (100.0%)	5252 (100.0%)	21.39 (100.0%)
提案手法	7.24 (90.4%)	5912 (112.6%)	29.49 (137.9%)
提案手法 (限定)	7.45 (93.0%)	5359 (102.0%)	24.99 (116.8%)

(b) 8 個の乗算のツリー状加算 (乗算器 3 個, 加算器 2 個)

	平均サイクル数	LUT 数	遅延 (ns)
Pilato [4]	10.03 (100.0%)	5525 (100.0%)	21.39 (100.0%)
提案手法	9.26 (92.3%)	6398 (115.8%)	28.06 (131.2%)
提案手法 (限定)	9.39 (93.6%)	5996 (108.5%)	26.59 (124.3%)

(c) 16 個の乗算のツリー状加算 (乗算器 3 個, 加算器 3 個)

	平均サイクル数	LUT 数	遅延 (ns)
Pilato [4]	16.92 (100.0%)	6577 (100.0%)	25.31 (100.0%)
提案手法	15.58 (92.1%)	8291 (126.1%)	33.10 (130.8%)
提案手法 (限定)	15.71 (92.8%)	7018 (106.7%)	29.92 (118.2%)

4. 実 験 結 果

Pilato の分散制御手法と本稿で提案する制御手法について, 平均サイクル数, 回路規模, 遅延を評価する実験を行なった. データフローグラフは, (a) 2 次元正方行列の積, (b) 8 個の乗算のツリー状加算, (c) 16 個の乗算のツリー状加算である. 加算は 1 サイクル, 乗算は 1〜3 サイクルで完了するものとし, (a) と (b) は乗算器 3 個と加算器 2 個, (c) は乗算器 3 個と加算器 2 個を使用可能とした. 平均サイクル数の評価は, 乗算を 1 サイクルで完了する確率が 50%, 3 サイクルで完了する確率が 50%としてランダムに決定し, シミュレーションを 10000 回行なって平均を算出した. 回路規模 (LUT 数) と遅延は, Verilog HDL で実装した回路を Xilinx Spartan3E をターゲットとして ISE (Version 13.1) で論理合成することにより求めた. 結果を表 1 に示す. 「提案手法」は, 全ての演算を, 使用可能な演算器全ての集合に割り当てたものである. 「提案手法 (限定)」は, 演算器の割り当てを限定したものであり, 使用可能数よりも 1 つ少ない演算器のみで従来のバインディングを行い, 各演算をバインディング結果の演算器とバインディングに使わなかった演算器に割り当てた. 平均サイクル数は, 提案手法では Pilato に比べて 8%〜10%減小し, 演算器の割り当てを限定した場合でも 6%〜7%減少した. LUT 数は, 提案手法では割り当てを限定しないと 12%〜26%増加するが, 割り当てを限定すると 2%〜9%の増加に抑えられた. 遅延は, 提案手法で 31%〜38%, 提案手法 (限定) で 17%〜24%増加した.

5. む す び

本稿では, Pilato の分散制御方式を拡張することにより, 演算を行う演算器を動的に決定できる分散制御方式を提案した. シミュレータを用いて各手法の平均サイクル数を計測する実験を行なった結果, 平均サイクル数を 6〜10%削減することができた.

制御回路の構成法にはまだ改良の余地があるので, この点が今後の課題として挙げられる. また, 演算の演算器への割り当て方法によって, 実行サイクル数, 回路規模, 遅延が変化するので, 効率的な割り当てを求めることも重要な課題であると考える.

謝　辞

本研究を行う上で多くのご助言, ご協力頂いた京都高度技術研究所の神原弘之氏, 元立命館大学の中谷嵩之氏, 元京都大学の矢野正治氏に感謝します. また, 本研究に関してご協力, ご討議頂いた田村真平氏をはじめ, 関西学院大学石浦研究室の諸氏に感謝します.

文　献

[1] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[2] Yuki Toda, Nagisa Ishiura, and Kousuke Sone: "Static scheduling of dynamic execution for high-level synthesis," in *Proc. SASIMI 2009,* pp. 107–112 (March 2009).

[3] 曽根康介, 石浦菜岐佐: "高位合成における可変スケジューリングの近似手法," 電子情報通信学会技術研究報告, VLD2010-90, (Jan. 2011).

[4] Alberto A. Del Barrio, Seda Ogrenci Memik, María C. Molina, José M. Mendías, and Román Hermida: "Using speculative functional units in high level synthesis," in *Proc. DATE 2010,* pp. 1779–1784 (March 2010).

[5] Christian Pilato, Vito Giovanni Castellana, Silvia Lovergine, and Fabrizio Ferrandi: "A Runtime Adaptive Controller for Supporting Hardware Components with Variable Latency," in *Proc. 2011 NASAIESA Conf. on Adaptive Hardware and Systems (AHS-2011),* pp. 153–160 (June 2011).