2004－SLDM－117 (32)
2004／12／2

Force-Directed Schedulingアルゴリズムを用いた非同期式データパス回路合成と効率化の検討

齋藤　寛[†]　米田　友洋[††]

† 会津大学コンピュータ理工学部　〒965-8580 福島県会津若松市一箕町鶴賀
†† 国立情報学研究所　〒101-8430 東京都千代田区一ツ橋 2-1-2
E-mail: †hiroshis@u-aizu.ac.jp, ††yoneda@nii.ac.jp

あらまし　本稿では，非同期式データパス回路のスケジューリングを効率よく行うために，非同期式回路の性質を利用した Force-Directed Scheduling アルゴリズムについて報告する．

キーワード　Force-directed scheduling algorithm，data flow graph，演算の終了時間，trigger ノード，コントロールステップ

Asynchronous Data-Path Circuit Synthesis by Using Force-Directed Scheduling Algorithm and Consideration to Improve Efficiency

Hiroshi SAITO[†] and Tomohiro YONEDA[††]

† Faculty of Computer Science and Engineering, The University of Aizu　Tsuruga, Ikki-machi, Aizu-Wakamatsu-shi, Fukushima, 965–8580 Japan
†† National Institute of Informatics　2–1–2 Hitotsubashi, Chiyoda-ku, Tokyo, 101–8430 Japan
E-mail: †hiroshis@u-aizu.ac.jp, ††yoneda@nii.ac.jp

Abstract　To improve scheduling of asynchronous data-path circuits, we describe the Force-Directed Scheduling algorithm based on the property of asynchronous circuits.

Key words　Force-directed scheduling algorithm, data flow graph, completion time of operation, trigger node, control step

1. はじめに

本稿では，非同期式回路のデータパス回路部で実行される演算のスケジューリングについて報告する．非同期式回路は，クロック信号を用いて回路内部のデータ処理やデータ転送を行う同期式回路とは異なり，事象 (イベント) 駆動原理に基づいて動作する回路である．データパス回路部の実装にはさまざまな手段が存在するが，ここでは，束データ方式に基づいた非同期式回路をスケジューリングの対象とする．束データ方式に基づいた非同期式回路では，1 ビットのデータは 1 本の配線で実現される．そのため，同期式回路のデータパス回路部で利用されているものと同じものが利用できる．違いは，データパスリソースの制御タイミングのとりかたが，クロック信号ではなく，演算にかかる最大遅延時間に相当した遅延素子を付加した要求信号によってとられているということにある．データパス回路部で実行される全ての演算のなかで，最も時間がかかる演算の最大遅延時間によってクロック信号が制約される同期式回路とは異なり，束データ方式に基づいた非同期式回路では，各演算はそれぞれ固有の最大遅延時間で実行される．

非同期式回路のスケジューリングとしては，一般的に，1) 既存の同期式回路設計で利用されている手法 [1] [2] を適用する，2) 既存の手法を改良したものを適用する，3) 非同期式回路特有の手法を適用する，のいずれかが考えられる．既存の同期式回路設計で利用されている手法のほとんどは，クロックサイクルを意図したコントロールステップを基にスケジューリングが行われる．しかし，非同期式回路の場合，前の演算の実行が終了した時点で次の演算の実行が開始されるという性質があるので，クロックサイクルのような一定の間隔でコントロールステップを決めてしまうと，無駄な計算やタイミング調整といった手間が生じてくる．非同期式回路に特有な手法としては，Bachman らの手法 [3] があげられる．この手法は，演算の間に依存関係を追加していくことによってスケジューリングを行う．したがって，演算に要する遅延時間がデータに依存するような回路を対象とした場合特に有効である．しかし，束データ方式に基づい

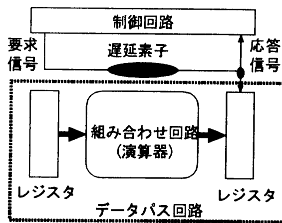

図 1　束データ方式に基づいた非同期式回路

た非同期式回路のように，各演算が最大遅延で動作するというのがあらかじめわかっている場合，逆に Bachman らの手法の利用は好ましくない．なぜなら，計算時間が演算の数に対して指数的に増加するからである．

以上のような状況を踏まえて，本稿では，既存の手法を改良するといった手段を検討する．改良の対象として，Paulin らによって提案された Force Directed Scheduling (FDS) アルゴリズム [4] を用いる．FDS アルゴリズムは，与えられた時間制約の下，リソースの利用が最適になるようスケジューリングを行う時間制約アルゴリズムの一種である．計算量がほかのものと比べて少なくてすむということより，幅広い用途で用いられている．改良における最も重要なポイントは，非同期式回路の性質を維持したまま効率よくスケジューリングできるよう，コントロールステップを演算の終了時間を基に計算することである．以下では，演算の終了時間を基にしたコントロールステップの計算法と、非同期式回路向けに改良した FDS アルゴリズムを中心に報告する．

2. 準　　備

2.1 束データ方式に基づいた非同期式回路

本稿では，スケジューリングの対象として，束データ方式に基づいた非同期式回路を想定している．束データ方式基づいた非同期式回路は，データパス回路部と制御回路部の 2 つから構成される (図 1)．データパス回路における 1 ビットのデータは，1 本の配線で実現される．そのため，データパス回路自体は同期式で利用されているものと違いはない．本稿では，データパス回路はあらかじめ準備されたリソースを用いて実現することを想定している．これらのリソースは，最大遅延時間や面積でパラメータ化されているものと仮定する．

一方制御回路は，データパスリソースを制御する要求信号を生成する．レジスタにデータを書き込むタイミングは，要求信号にデータパス回路で処理される演算の最大遅延時間に相当した遅延素子を付加することで保証される．レジスタにデータを書き込んだあと，要求信号は応答信号として制御回路に演算の終了を知らせる．

2.2 Data Flow Graph

Data Flow Graph(DFG) は，データパス回路で処理される演算の流れをグラフ化したものであり，$G < N, E >$ で定義される．ノード集合 N における各ノード $n_i (i = 1 \ldots k)$ は，データパス回路で処理される 1 つの演算を表す．一方，エッジ集合 E における各エッジは，ノード間のデータ依存を表す．図 2 は，

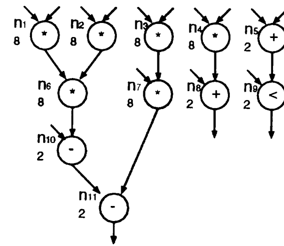

図 2　DFG の一例

DFG の一例である．

提案手法を用いてスケジューリングを行う前に，各ノード n_i の演算時間がどのくらいかかるかをあらかじめ計算しておく．これは，パラメータ化されたリソースを，各ノードに割り当てることによって算出することができる．図 2 の DFG における各ノード n_i に割り当てられた数字は，そのノードを実行するためにかかる最大遅延時間である．

3. 従来手法の問題点

従来のスケジューリング手法は，与えられた時間制約を基にコントロールステップ (以下ステップとよぶ) を割り出し，どのステップに演算をスケジュールするかということを考える．したがって，ステップの決定はスケジューリング時間に大きな影響を与える．

従来の手法では，ステップは，後々のクロックサイクルを想定して一定の時間間隔をもつように決定される．例えば，図 3(a) の DFG では，最大遅延時間の異なる 2 つの演算 (乗算と加算) が含まれる．このとき，乗算の最大遅延時間 (5ns) を基にステップの間隔を決めてしまうと，ステップ $cstep_2$ で 2 つの加算器が必要となる (図 3(b))．一方，乗算と加算における最大遅延時間の最大公約数である 1 をもとにステップの間隔を決めると，加算器は全体で 1 つあれば十分だということがわかる (図 3(c))．しかし，ステップの間隔を細かくすればするほどスケジューリングにかかる時間も増加してしまう．

また，仮にステップを細かく設定したと仮定しても，非同期式回路を対象とした場合，非同期式回路の性質を考慮しないでステップを設定してしまうと，計算時間の浪費やタイミング調整の必要性に直面する．例えば，図 3 の例で，ノード n_4 をステップ $cstep_4$ にスケジュールするかどうかであるが，既存手法はコントロールステップが設定されているので，スケジューリングの可能性を探ろうとする．しかしながら，非同期式回路を対象とした場合，必ずしもこのステップでのスケジューリングを考慮する必要はない．なぜなら，非同期式回路における演算は，前の演算の終了が引き金となって開始されるが，この場合，前の演算の終了から一ステップ分の時間の空きがある．つまり，n_4 の開始の引き金となる演算が前のステップに存在しない．従って，演算器の個数を減らすことにつながらない限り，このステップでの計算は無駄になってしまう．また，仮にこのステップでの開始を強いる場合，開始タイミングを保証する遅延素子を実装の段階で導入する必要がある．

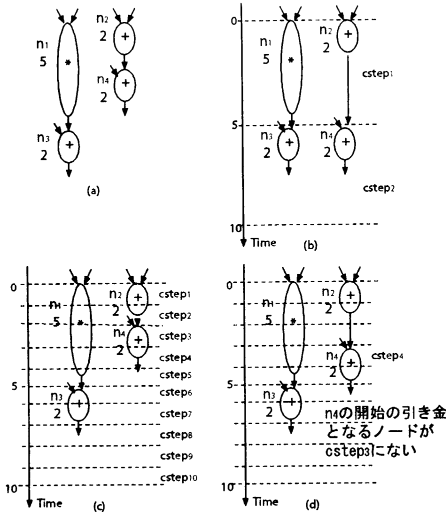

図 3 (a)DFG, (b) ステップ間隔が 5ns のときのスケジュール例, (c) ステップ間隔が 1ns のときのスケジュール例, (d) ステップ $cstep_4$ にノード n_4 をスケジュール

step 1 ASAP スケジュールと ALAP スケジュールの計算
step 2 各ノード n_i の開始時間候補と終了時間候補の計算
　　計算 1 を用いた計算
　　計算 2 を用いた計算
step 3 各ノード n_i の開始時間候補と最後の演算における $asap_e(n_i)$ よりステップを算出

図 4 コントロールステップの計算法

以上のような状況から, ステップの時間間隔は細かくすればよいということでもなく, むしろ, 非同期式回路の性質をうまく利用して決定することが重要である. そうすることで, 非同期式回路の性質を保ちながら, 効率よくスケジューリングを行うことが可能となる.

4. 演算の終了時間を基にしたステップの計算

演算の終了時間を基にしたステップの計算手順は以下の通りである.

まず始めに, As Soon As Possible(ASAP) と As Late As Possible(ALAP) スケジューリングを行い, クリティカルパス遅延とクリティカルパス遅延を変えない各ノードの最も遅い開始時間と終了時間を計算する (**step 1**). ASAP スケジューリングとは, できるだけ早く演算が実行されるようスケジュールしたものであり, ALAP スケジューリングとは, できるだけ遅く演算が実行されるようスケジュールしたものである. 図 5 は, 図 2 の DFG に対する, ASAP スケジュールと ALAP スケジュールである. なお, ASAP スケジューリングによって算出されたノード n_i の開始時間を $asap_s(n_i)$, 終了時間を $asap_e(n_i)$ とする. また, ALAP スケジューリングによって算出されたノード n_i の開始時間を $alap_s(n_i)$, 終了時間を $alap_e(n_i)$ とする. 図 6(a) は, 各ノードの $asap_s(n_i)$, $asap_e(n_i)$, $alap_s(n_i)$, $alap_e(n_i)$ を表す.

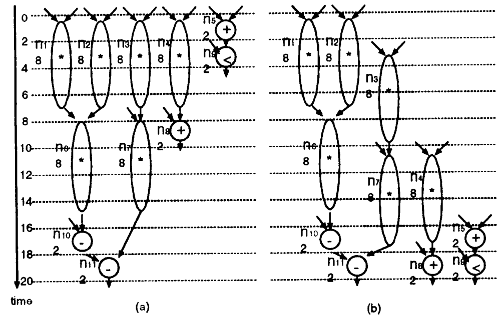

図 5 (a)ASAP スケジューリング, (b)ALAP スケジューリング

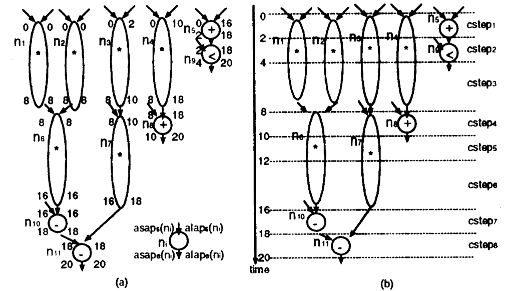

図 6 (a)ASAP/ALAP スケジュールから算出された時間, (b) 計算されたステップ

次に, スケジューリングが可能なノード $n_i(asap_s(n_i) < alap_s(n_i)$ を満たすノード) の開始時間候補と終了時間候補を計算する (**step 2**). また, ノード n_i の開始の引き金となるノード (tirgger ノード)$n_j(j \neq i)$ も計算する. なお, trigger ノードを計算する理由は, ノード n_i があるステップにスケジュールされるとき, その引き金となるノードが n_j のみならば, n_j も一緒にスケジュールされる必要がある. そのための計算である.

なお, 開始時間候補と終了時間候補の計算は, 以下の 2 つからなる.

（1） **n_i とその successor 以外の全てのノード n_j からの計算 (計算 1):** n_j の $asap_e(n_j)$ が, n_i の $asap_s(n_i)$ と $alap_s(n_i)$ の範囲内にあるなら, $asap_e(n_j)$ を n_i の開始時間候補 S_{n_i} に, $asap_e(n_j)$ に n_i の最大遅延時間を加えたものを n_i の終了時間候補 E_{n_i} に加える. また, n_j を n_i が $asap_e(n_j)$ から計算を開始するための trigger ノードとする.

（2） **n_i の predecessor ノード n_j からの計算 (計算 2):** n_j の終了時間候補 E_{n_j} の全てを S_{n_i} に加える. また, n_j を n_i がそれらのときから計算を開始するための trigger ノードとする.

例として, 図 6(a) におけるノード n_8 の開始時間候補, 終了時間候補を計算する. 上記の計算 1 より, n_8 の開始時間候補 S_{n_8} は $S_{n_8} = \{8, 16, 18\}$ であり, 終了時間候補 E_{n_8} は $E_{n_8} = \{10, 18, 20\}$ である. また, 開始時間 8 における trigger ノードは n_1, n_2, n_3, n_4, 開始時間 16 における trigger ノード

Repeat until 全ての演算がスケジュールされる

```
step 1:  前節で説明された計算法を用いたステップの計算
step 2:  Time frame の計算
step 3:  Distribution graph の作成
step 4:  スケジューリングが可能な開始時間候補の全てで,
         self force を計算
step 5:  predecessor や successor の force を self force
         に加算
step 6:  trigger ノードの force を self force に加算
step 7:  self force の値が最も小さくなるステップ
         に演算をスケジュール
```

End repeat

```
step 8:  スケジュールエッジの付加
```

図 7 非同期式回路向けに改良された FDS アルゴリズム

は, n_6, n_7, 18 における trigger ノードは n_{10} である. 次に計算 2 より, n_8 の開始時間候補 S_{n_8} は $S_{n_8} = \{8, 10, 12, 16, 18\}$ となる. これは, 計算 1 によって n_8 の predecessor である n_4 における終了時間候補が $E_{n_4} = \{8, 10, 12, 16\}$ となるからである. また, 終了時間候補 E_{n_8} は $E_{n_8} = \{10, 12, 14, 18, 20\}$ となる. さらに, n_4 は, 開始時間 10 と 12 における trigger ノードとなる.

最終的に, 全てのノードの開始時間候補と最後の演算における $asap_e(n_i)$ よりステップが算出される (**step 3**). 図 6(b) は, 図 6(a) の例において計算されたステップを表す.

なお, ここで提案されたステップの計算法は, FDS アルゴリズム以外のステップを基にスケジューリングを行う他の手法でも利用することが可能である.

5. 非同期式回路のスケジューリングを対象とした FDS アルゴリズム

この節では, 前節で述べたステップの計算法を利用した FDS アルゴリズムについて説明する.

FDS アルゴリズム [4] は, 時間制約に基づいたスケジューリングアルゴリズムの一つで, Paulin らによって提案された. 時間制約に基づいたスケジューリングアルゴリズムとは, 与えられた時間制約 (クリティカルパス遅延) を変えない範囲で, 最適なリソース数が得られるよう演算をスケジューリングするアルゴリズムである. 最適なリソース数を求めるにあたって FDS アルゴリズムは, 与えられた時間制約の範囲内で, リソースの使用がバランス良くなるよう演算をスケジュールする. FDS アルゴリズムは他の手法とくらべ計算量が比較的小さいということより, 従来より広く用いられている.

図 7 は, 非同期式回路のスケジューリングのために, 前節で述べたステップの計算法を利用した FDS アルゴリズムを表す. なお, 元々の FDS アルゴリズムにはない部分には下線が引かれている.

アルゴリズムでは, まず始めに, 前節で述べたステップの計算法を用いてステップを計算する (**step 1**). 計算されたステップの例は, 図 6(b) のとおりである.

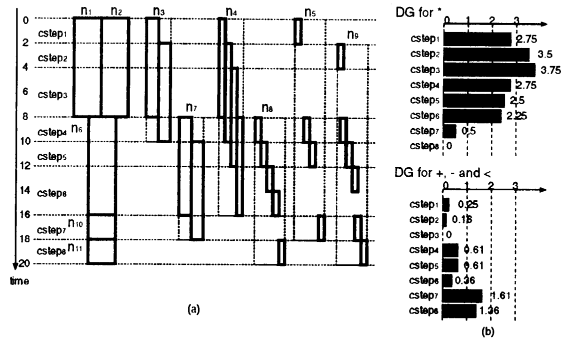

図 8 (a)Time frame, (b)DG

次に, ステップと一緒に計算された開始時間候補 S_{n_i} のそれぞれに演算がスケジュールされたときのステップの範囲を示した time frame を計算する (**step 2**). 例えば, 図 6(b) のノード n_4 場合, 開始時間候補 S_{n_4} は $S_{n_4} = \{0, 2, 4, 8\}$ である. これらの時間に対応したステップ $cstep_1$, $cstep_2$, $cstep_3$, $cstep_4$ にノードをスケジュールしたときの time frame は, $cstep_1$ から $cstep_6$ までとなる. 図 8(a) は, 全てのノードの time frame を表す. なお, 複数の開始時間候補を持つノードは, それぞれの開始時間候補にスケジュールされたときの time frame を表している.

Time frame の計算後, 演算がステップ $cstep_j$ にスケジュールされる可能性を示した probability($Prob(n_i, cstep_j)$) を基に, 各ステップでのリソースの使用率をグラフ化した Distribution Graph(DG) を計算する (**step 3**).

まず, $Prob(n_i, cstep_j)$ は以下のように計算される.

$$Prob(n_i, cstep_j) = \frac{1}{n_i\text{の開始時間候補の数}} * n_i\text{が } cstep_j \text{にスケジュールされる回数}$$

例えば, 図 6(b) におけるノード n_4 がステップ $cstep_3$ にスケジュールされる可能性は, $Prob(n_4, cstep_3) = 1/4 * 3 = 0.75$ である. n_4 の開始時間候補の数は 4 であり, n_4 がステップ $cstep_3$ にスケジュールされるのは, 開始時間が 0, 2, 4 のそれぞれから開始されたときの 3 回である.

DG は, 各ステップでのリソースの使用率を表したものであり, 以下のように計算される.

$$DG(type, cstep_i) = \sum_{cstep_i\text{で } type \text{ が一致するノード } n_i} Prob(n_i cstep_i)$$

ここで, type は, 利用される演算器の種類を表す. 図 6(b) の例では, 乗算は乗算器で, それ以外の演算は ALU で実行されるものと仮定している. 乗算器の type を 0 とすると, $DG(0, cstep_1)$ は, $DG(0, cstep_1) = Prob(n_1, cstep_1) + Prob(n_2, cstep_1) + Prob(n_3, cstep_1) + Prob(n_4, cstep_1) = 1+1+0.5+0.25 = 2.75$ となる. 図 8(b) は, 計算された DG を表す.

DG の計算が終了したら, 全ての開始時間候補に対応したステップで, 実際に演算がスケジュールされたときの全体のリ

ソースの使用がどれだけバランス良くなったかを示す self force を計算する (**step 4**). ノード n_i をあるステップ $cstep_j$ にスケジュールしたときの self force ($SelfForce(n_i, cstep_j)$) は，以下のように計算される．

$$SelfForce(n_i, cstep_j) = \sum_{cstep_k \in n_i の\ timeframe} DG(type, cstep_k)*x(n_i, cstep_k)*w(type)$$

ここで，n_i が $cstep_j$ にスケジュールされることによって，$cstep_k$ でもリソースが必要となるならば，

$$x(n_i, cstep_k) = 1 - Prob(n_i, cstep_k)$$

一方，不必要となるならば，

$$x(n_i, cstep_k) = -Prob(n_i, cstep_k)$$

となる．$w(type)$ は演算器にかかる weight を表す．$SelfForce(n_i, cstep_j)$ が負の値をもつときは，リソースの使用がバランス良くなっていることを表し，正の値をもつときは，バランスが悪くなっていることを表す．

例として，図 6(b) のノード n_3 をステップ $cstep_1$ にスケジュールしたときの $SelfForce(n_3, cstep_1)$ を計算する．なお，n_3 に対応する演算器は乗算器で (type は 0)，$w(0)$ は乗算にかかる最大遅延時間とする．$SelfForce(n_3, cstep_1)$ は，
$SelfForce(n_3, cstep_1) = (DG(0, cstep_1) * x(n_3, cstep_1) * w(0)) + (DG(0, cstep_2) * x(n_3, cstep_2) * w(0)) + (DG(0, cstep_2) * x(n_3, cstep_2) * w(0)) + (DG(0, cstep_4) * x(n_4, cstep_1) * w(0)) = (2.75*0.5*8) + (3.5*0*8) + (3.75*0*8) + (2.75*(-0.5)*8) = 0$
となる．

あるノード n_i があるステップ $cstep_j$ にスケジュールされると，その predecessor や successor がスケジュールされるステップが一意に定まることがある．そういったときは，predecessor や successor の self force を $SelfForce(n_i, cstep_j)$ に加算する (**step 5**).

同様に，あるノード n_i があるステップ $cstep_j$ にスケジュールされるとき，その演算の開始の引き金となる trigger ノードが唯一つの場合，その trigger ノードがスケジュールされるステップも一意に定まる必要がある．そういったときは，trigger ノードの self force を $SelfForce(n_i, cstep_j)$ に加算する (**step 6**). こうすることで，演算の開始が，対象となる非同期式回路内で必ず保証されるようになる．Trigger ノードの考慮は，動作保証のみならず，スケジューリングの質を高めることにも一役買っている．なぜなら，trigger ノードも考慮することで，あるノードのスケジュールによる影響をより詳しく調べることが可能となるからである．

最終的に，全ての演算が全ての開始時間候補で self force を計算した後，最も self force が小さくなるようなスケジュールを選択する (**step 7**).

以上のような作業が，全ての演算がスケジュールされるまで繰り返される．

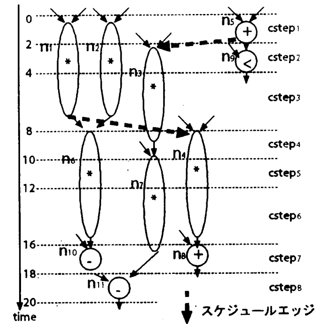

図 9 スケジューリング結果

スケジューリングの終了後，演算の開始の引き金となる trigger ノードと依存関係が無い場合，スケジュールエッジと呼ばれる新たな依存エッジを DFG に追加する．スケジュールエッジの追加は，制御回路の合成の際に利用される．trigger ノードが複数ある場合は，そのなかの一つを選択し，スケジュールエッジを追加する．図 9 は，図 6(b) の DFG に，改良された FDS アルゴリズムを適用した際に得られたスケジューリング結果を表す．

6. 実　　験

非同期式データパス回路のスケジューリングのために改良された FDS アルゴリズムを用いて実験を行う．

実験では，幾つかの DFG に提案手法を適用したときの，合成結果の質 (すなわち，演算器数) と計算量 (ここでは，self force の計算がどのくらい行われたか) を通常の FDS アルゴリズムを用いたときのものと比較する．なお，従来手法を用いたあとに必要とされるタイミング調整についてはここでは特に触れない．

提案手法はまだ完全に実装されたわけではないが，スケジューリングのメインとなるステップの計算と self force の計算の部分は実装が完了している．従って，この節で表されるデータは，実装されたものを用いて得られた結果である．実装は，C 言語で行われ，gcc 2.95.3 でコンパイルを行った．実験に用いた CPU は，SUN の UltraSPARC-2e(502MHz) で，メモリは 512MB である．

6.1 実　験　1

ここでは，非同期式回路向けに改良された FDS アルゴリズムを用いて，differential equation solver(DIFFEQ)，AR-Lattice(AR)，Elliptic Wave Filter(EWF) のスケジューリングを行う．用いることのできる演算器は，乗算器 (8ns) と ALU(2ns) の 2 つと仮定する．

表 1 に，実験結果を示す．表における，node，latency，cstep，iteration，force，resouce はそれぞれ，DFG のノード数，初期 ASAP アルゴリズムで得られた時間制約，ステップ数，アルゴリズムのイテレーション数，計算された self force の数，スケジューリングによって必要とされるリソースの数を表す．なお，従来の FDS アルゴリズムにおける cstep の time interval は，2 である (乗算器と ALU における最大遅延時間の最大公約数).

表1　実験1の結果

			従来の FDS アルゴリズム				改良された FDS アルゴリズム			
name	node	latency	cstep	iteration	force	resource	cstep	iteration	force	resource
DIFFEQ	11	20	10	3	535	ALU*2,MUL*3	8	4	204	ALU*2,MUL*3
AR	28	34	17	6	2,740	ALU*2,MUL*6	11	5	1,038	ALU*3,MUL*6
EWF	34	46	23	8	1,892	ALU*3,MUL*4	18(17)	5	704	ALU*3,MUL*4

表2　実験2の結果

		従来の FDS アルゴリズム				改良された FDS アルゴリズム			
delay	latency	cstep	iteration	force	resource	cstep	iteration	force	resource
4ns	12	6	3	185	ALU*2,MUL*3	6	3	167	ALU*2,MUL*3
7ns	18	18	3	1818	ALU*2,MUL*3	8	4	204	ALU*2,MUL*3
8ns	20	10	3	535	ALU*2,MUL*3	8	4	204	ALU*2,MUL*3
10ns	24	12	3	779	ALU*2,MUL*3	8	4	204	ALU*2,MUL*3

改良された FDS アルゴリズムを利用すると，DIFFEQ と EWF の場合，スケジューリングにかかる計算量がおさえられ，なおかつ解の質 (演算器の数) も従来の FDS アルゴリズムと同等のものが得ることができるということがわかる．計算量を抑える原因は，1) スケジューリングの計算にふさわしい cstep を用いたということと，2)trigger ノードの存在によって，アルゴリズムのイテレーション数が減った (EWF の場合) ということがあげられる．あるノードをスケジュールしたときに，そのノードの実行開始の引き金となる trigger ノードのスケジュールもまた一意に決まることがあるので，イテレーションの数が減少する．なお，cstep の計算は，スケジューリングの最中に再計算されるので，EWF のように最初は 17 であったものが，増加するということもある．これは，直前にスケジューリングされたノードの影響で，開始時間候補が増えることに起因する．

AR の場合，計算量を抑えることはできたが，解の質は悪くなっている (ALU が 1 つ余分に必要)．この原因は，self force の計算の際に用いられた weight に起因する．乗算器の weight を 8(ALU は 2) としてしまっているため，スケジューリングにおける ALU の数への影響が無視できるくらい小さく見積もられていることが原因である．この問題を解決するためには，weight の値の取り方や self force における weight の貢献を考え直す必要があるだろう．

6.2 実　験　2

ここでは，DIFFEQ を例にとり，乗算器の遅延時間を $4ns$, $7ns$, $10ns$ に変えた場合の計算量と解の質を比較する．

実験の結果を表 2 に示す．この実験からわかることは，乗算器の遅延時間を変えた場合，従来の手法では，ステップ数の変化 (ステップの数を ALU と乗算器の遅延の最大公約数によってきめている) に応じて，self force の計算量も変化することがわかる．一方，乗算器の遅延を $7ns$, $8ns$, $10ns$ と設定して改良された手法を適用した場合，ノードがスケジュールされてもステップ数に変化がなく，なおかつ開始時間候補にも変化がないので，計算量にも変化がないことがわかる．こういったことから，適切なステップを計算することによって，無駄な計算を無くすことが可能となる．

7. ま　と　め

本稿では，FDS アルゴリズムを利用した非同期式データパス回路のスケジューリングについて報告した．FDS アルゴリズムのような既存の手法では，非同期式回路の性質を考慮することなくスケジューリングを行うので，無駄な計算やタイミング調整が必要となってしまう．そのため，本稿では，演算の終了時間に基づいてコントロールステップを計算する方法を提案し，それを利用した FDS アルゴリズムについて報告を行った．今後の課題として，提案した手法の完全自動化とコントロールステップに基づいた他のスケジューリング手法での検討を行うつもりである．

文　　献

[1] G.De Micheli. Synthesis and Optimization of Digital Circuits. McGraw-hill, 1994.

[2] S.Govindarajan. Scheduling Algorithms For High-Level Synthesis. Technical paper, March 1995.

[3] B. M. Bachman, H. Zheng, and C. J. Myers. Architectural Synthesis of Timed Asynchronous Systems. In *Proc. of IEEE International Conference on Conputer Design*, Oct. 1999.

[4] P. G. Paulin and J. P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Transaction on Computer-Aided Design*, vol.8, no.6, pp.661–679, June 1989.