JAIST Repository
https://dspace.jaist.ac.jp/

Title	冗長化アルゴリズムからの耐故障データパス自動合成
Author(s)	坪石, 優
Citation	
Issue Date	2009-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/8116
Rights	
Description	Supervisor:金子峰雄, 情報科学研究科, 修士

Japan Advanced Institute of Science and Technology

修 士 論 文

冗長化アルゴリズムからの耐故障データパス自動合成

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

坪石 優

2009 年 3 月

修 士 論 文

冗長化アルゴリズムからの耐故障データパス自動合成

指導教員　金子峰雄 教授

審査委員主査　金子峰雄 教授
審査委員　宮地充子 教授
審査委員　浅野哲夫 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

0710051 坪石 優

提出年月: 2009 年 2 月

Copyright © 2009 by Yutaka Tsuboishi

概 要

集積回路の動作中に発生する故障に対して様々なアプローチの耐故障化技術が提案されている。本稿ではアルゴリズム三重化をスタート点として、高位合成の枠組みを用いて耐故障性が維持されるよう多数決回路導入、演算スケジュール、資源割当てを行なって耐故障データパス回路を合成する基礎概念の下での設計最適化について論じる。具体的には、多数決演算挿入位置が特定された下での耐故障性を維持した資源制約スケジュール・資源割当て問題に対して、整数線形計画法による厳密解法と、資源割当てを優先して行う発見的手法を提案する。

目 次

第 1 章　はじめに **1**

第 2 章　高位合成 **2**
2.1　高位合成の概要 2
2.2　演算のスケジューリング 3
2.3　資源割り当て 3

第 3 章　耐故障化手法 **5**
3.1　ハードウェアと故障のモデル 5
3.2　ボート演算による誤り訂正の方式 7
3.3　耐故障データパス 8

第 4 章　整数線形計画法によるデータパス合成 **12**
4.1　ILP 定式化 12
4.2　実験結果 16

第 5 章　発見的手法によるデータパス合成 **17**
5.1　概要 17
5.2　コーンの選択 17
5.3　コーンへの演算資源割当て 21
5.4　スケジューリング 25
5.5　実験結果 28
5.6　その他の手法による事例 30

第 6 章　結論 **41**
6.1　まとめ 41
6.2　今後の課題 41

第1章　はじめに

今日の集積回路製造技術の進歩は複雑で大規模な計算処理を高速に実行できる集積回路を構成することを可能にした。これにより、集積回路は我々の身近な製品にも応用され、その中核を担うようになった。しかしながら、動作中に発生する故障はリアルタイム性を求める用途にとって重大な障害に発展する危険性をはらむ。そのため、耐故障技術は集積回路を設計する上で重要なものとなる。

これまで、VLSI(Very Large Scale Integrated circuit) の信頼性の向上のために様々な耐故障研究が行われている。[3] 同一のモジュールを三重化して多数決をとるモジュール三重化や、モジュールを多数用意し故障が発生した際には故障箇所を切り離して再構成するモジュール配列の再構成はその成果の一つであるものの、実装面積の肥大化や再構成時にシステムを停止するためにリアルタイム性が損なわれるという欠点があった。

そこで、高いリアルタイム性と低いハードウェアオーバーヘッドを両立する代表的な手法として、計算アルゴリズム自身に誤り検出・訂正を行う冗長計算を導入する ABFT（Algorithm Based Fault Tolerance）が提案された [4] ものの、その適用は線形的な計算アルゴリズムに限定されそれ以外には原理的に適用できない。そのため、文献 [5] では二重化した計算アルゴリズムの動作を維持しつつ適宜多数決演算を導入し、アルゴリズムの構造を変換することで利用可能な資源を削減しながら誤り検出可能なデータパスを合成している。一方、本研究で扱う耐故障手法では、アルゴリズム三重化による誤り検出・訂正可能な手法であり計算アルゴリズムの構造を変換することができないため、スケジュールや資源割当て、多数決演算の挿入位置の決定に課題が残されていた [1]。

本研究では、整数線形計画法と発見的手法により、与えられる計算アルゴリズムとあらかじめ決定されたボート演算の挿入位置から、上記の耐故障性を持ったスケジュールの生成と資源割り当てを行う手法を提案した。整数線形計画法による解法では小規模のアルゴリズムであるならば実用時間内でスケジューリングを完了できることを示し、発見的手法による解法では厳密解に迫る解を導き出せることを示した。これにより、アルゴリズム三重化による誤り訂正・可能なスケジュールを合成する手法を確立するに至った。

第2章　高位合成

この章では、VLSI の設計階層のひとつである高位合成について説明する。

2.1　高位合成の概要

VLSI の設計では、レイアウト合成やトランジスタレベル、論理レベル、レジスタ転送レベル、そしてシステムレベルといった様々な階層がある。ここでの高位合成はレジスタ転送レベルを指し、計算アルゴリズムを入力とし演算器やレジスタ、マルチプレクサなどで構成されたデータパスを合成する設計技術である。本研究では、計算アルゴリズムをデータフローグラフと呼ばれる演算を頂点とした有向グラフで扱う。データフローグラフは演算集合を O、演算間の依存関係を E としたとき、G=(O,E) で表される。図 2.1 に計算アルゴリズムの例と図 2.2 に対応するデータフローグラフ G の例、そして図 2.3 に出力となるスケジュールの例を示す。図 2.1 のアルゴリズムの例では、演算 1、2、3 で計算された値が変数 e、d、f へそれぞれ収められる。図 2.2 のデータフローグラフでは、頂点 $+_1$ が O_1 を、頂点 $*_2$ は O_2 を、そして頂点 $+_3$ は O_3 をそれぞれ表現している。図 2.3 のスケジュールは図 2.2 のデータフローグラフを加算器と乗算器をそれぞれ１つづつ利用可能としてスケジューリングした例である。

入力 a, b, c
出力 d, e, f
example{
　$e = a + b;$　$//O_1$
　$d = a * c;$　$//O_2$
　$f = e + d;$　$//O_3$
}

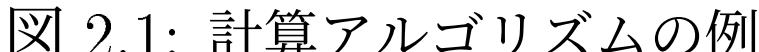
図 2.1: 計算アルゴリズムの例

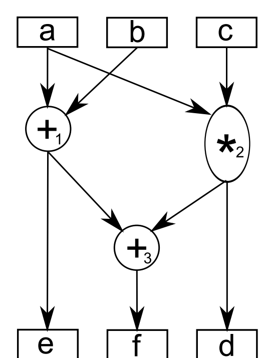

図 2.2: 対応するデータフローグラフ G の例

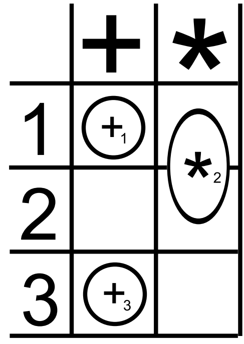

図 2.3: 出力されるスケジュールの例

2.2 演算のスケジューリング

演算のスケジューリングは、各演算間のデータの依存関係を保ちつつ各演算の実行時間を決定する処理である。スケジュールは $\sigma{:}V \to Z_+$ で表される。スケジューリングは、大きく分けて資源制約スケジューリングと時間制約スケジューリングの二種類がある。資源制約スケジューリングでは、各演算へつけられた優先順位に基づき各演算の実行時刻を決定するリストスケジューリングが主に用いられる。また、時間制約スケジューリングでは、スケジュールによる資源利用を確率的に評価しながら資源量が小さくなるようにスケジュールを決定する力学的スケジューリングが主に用いられる。なお、実行時間の上限と下限を見極めるための手段として資源数を無限大とした制約下で各演算の実行時間をできうる限り早く定める ASAP(As Soon As Possible)(図 2.4) や同じく資源数を無限大とした制約下で各演算の実行時間をできうる限り遅く定める ALAP(As Late As Possible)(図 2.5) が用いられる。

2.3 資源割り当て

資源割り当ては演算を実際に行う演算器の決定とデータを格納するレジスタの割り当てを行う処理である。演算器割り当ては F を演算器の集合としたとき $\rho{:}O \to F$ で表され、レジスタ割り当ては R をレジスタの集合としたとき $\xi{:}O \to R$ で表される。演算の種類には、加算や乗算、シフトそしてボート (多数決) などの種類があり、各演算に対して正しい演算器を割り当てる必要がある。データでは、割り当てようとするデータのビット長よりも大きいビット長を格納可能なレジスタへ割り当てる必要がある。変数では、演算を実行する時刻からデータを保持しなければならない時刻をライフタイムと呼ばれ、ライフタイムが重なる場合は同一のレジスタを割り当てることはできないが、ライフタイムの異なる場合は同一のレジスタを利用することができる。図 2.6 はレジスタ割り当て ξ の一例である。

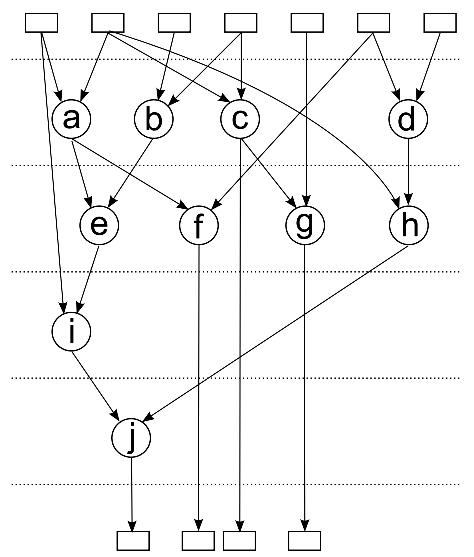

図 2.4: ASAP によるスケジューリングの例

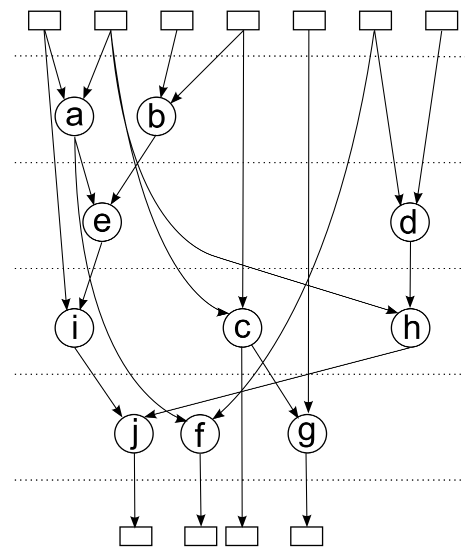

図 2.5: ALAP によるスケジューリングの例

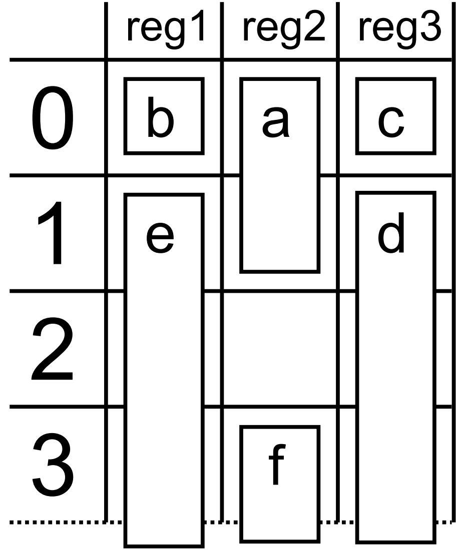

図 2.6: レジスタ割り当ての例

第3章　耐故障化手法

この章では、本研究で扱う耐故障手法の概要を説明する。

3.1　ハードウェアと故障のモデル

データパスの構成要素を演算器、レジスタ、マルチプレクサ、ボータ (多数決回路)、結線とする。それぞれの構成要素を以下のように定義する。

- 演算器は 2 入力 1 出力とする。
- レジスタは 1 入力 1 出力とする。
- マルチプレクサは多入力 1 出力とする。
- ボータは 3 入力 2 出力とする。
- 結線は 1 入力多出力とする。

これらの各構成要素に対して故障モデルを以下のように定義する。

- 演算器とその出力を、他の構成要素へ伝達する結線または演算器への入力信号を選択するマルチプレクサと、その出力を演算器へ伝達する結線が正しい値を出力しない。(図 3.1)
- レジスタとその出力を、他の構成要素へ伝達する結線またはレジスタへの入力信号を選択するマルチプレクサと、その出力をレジスタへ伝達する結線が正しい値を出力しない。(図 3.2)
- ボータがボートした結果出力される信号、または誤った値の入力元レジスタを指定する信号のいずれかもしくは両方と、その出力を他の構成要素へ伝達する結線が正しい値を出力しない。(図 3.3)

定義 1　上のいずれかに該当する故障が k 個だけであるときかつそのときに限り「k 故障」と呼ぶ。

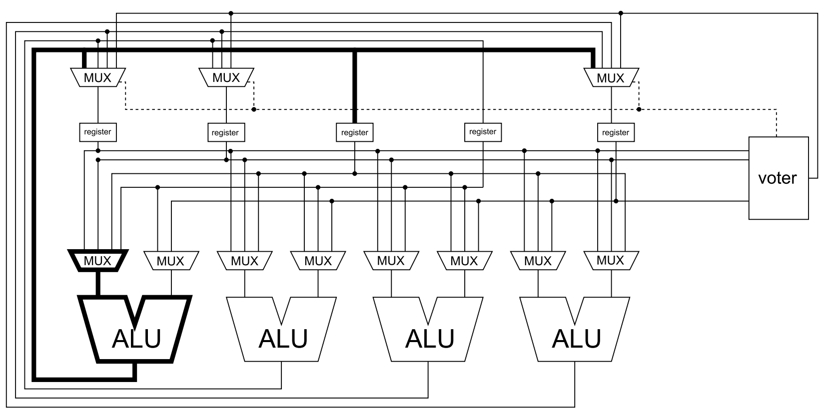

図 3.1: 故障モデル 1

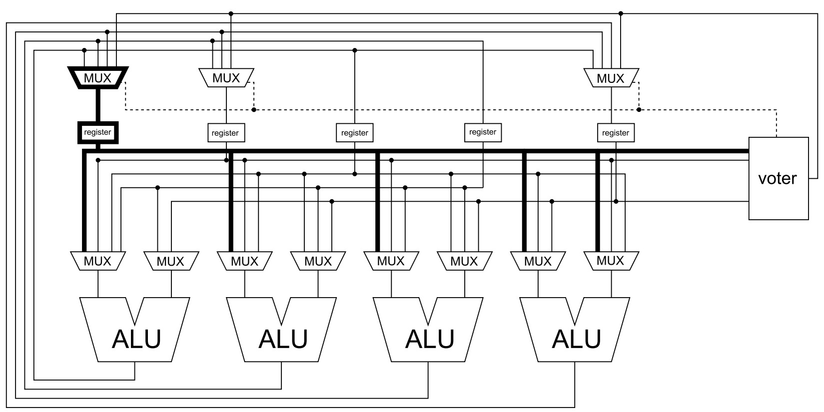

図 3.2: 故障モデル 2

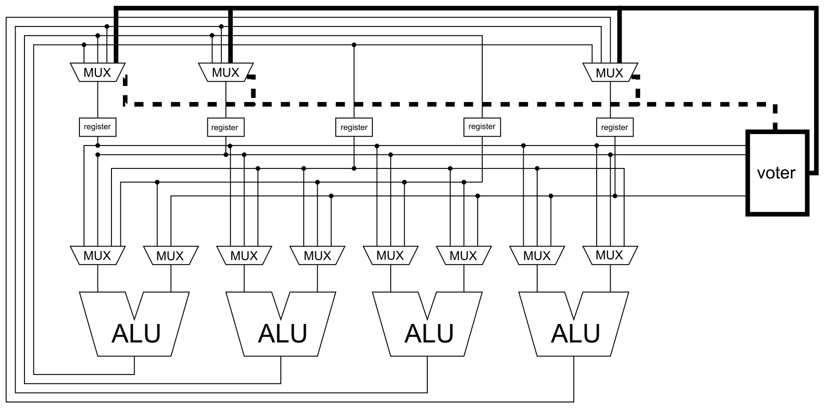

図 3.3: 故障モデル 3

3.2 ボート演算による誤り訂正の方式

ここでは、ボート演算による誤り訂正の方式について代表的なものと本研究での手法について説明する。図 3.4 は、モジュール三重化の例である。モジュール三重化による耐故障では、モジュール A から出力された値をボータによって多数決することで故障したモジュール A により出力された誤った値を、ボータより後で実行されるモジュール B へ送出しないことで耐故障化を実現する。しかしながら、ボータ自身が故障してしまった場合、ボータより後で実行されるモジュール B へ誤った値が送出される可能性がある。そこで、図 3.5 のようにボータ自体も三重化することでボータが故障してもボータより後で実行されるモジュール B へは誤った値が高々1 つ送出されることで耐故障化を実現できる。しかしながら、配線が複雑になるため製造コストが高くなる欠点がある。図 3.6 は本研究で扱う耐故障手法である。モジュール A から出力された値はそのままモジュール B へ渡されると同時に多数決を行う。多数決結果により誤りが発見された場合は、該当するモジュール B への入力は多数決結果を用いる。具体的には、ALU の計算結果はレジスタへ渡され必要なくなるまでその値を保持するが、レジスタへ収められた値をボータでボートし、誤った値を保持しているレジスタへはボート結果の値を上書きする。この方式では、ボータ自身が故障した際でもモジュール A のいずれかが故障した際でもモジュール B へは誤った値が高々1 つ送出されるため、耐故障化を実現できる。

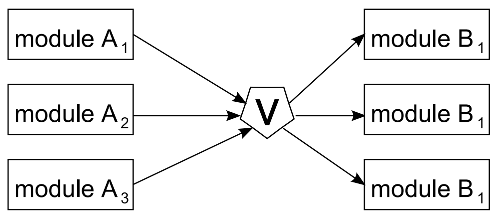

図 3.4: Triple Modular Redundancy

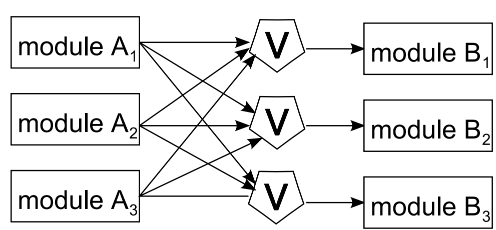

図 3.5: ボータも三重化された Triple Modular Redundancy

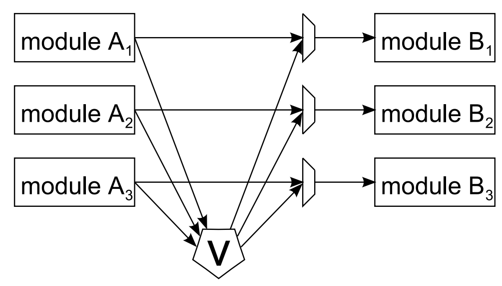

図 3.6: 本研究での耐故障手法

3.3 耐故障データパス

前述のハードウェアモデルおよび故障モデルを実現するために、与えられるデータフローグラフ G を三重化し、ボートする演算が出力される変数へボート演算を挿入することで耐故障化する。三重化されたデータフローグラフの演算をそれぞれ O_1、O_2、O_3 とし、その演算間の依存関係を E_1、E_2、E_3 とすると、三重化されたデータフローグラフは $\tilde{G}=(O_1 \cup O_2 \cup O_3, E_1 \cup E_2 \cup E_3)$ と表せる。

この耐故障システムには、以下の問題がある。

1. ボート演算を挿入する位置の決定
ボート演算は耐故障性ができうる限り高くなるように、かつ少ない実行時間でスケジュールできるような箇所へ挿入しなければならない。図 5.24 と図 5.26 はボータの挿入位置によってコントロールステップが変化する例である。また、図 5.25 は図 5.24 よりボータ挿入位置を増やすことによりコントロールステップが増加する例である。

2. アルゴリズムを時間とハードウェア上へのマッピング
スケジュールや演算器割り当て、レジスタ割り当て、ボート演算のボータへの割り当ては耐故障性を維持しなければならない。

定義 2 与えられる計算アルゴリズム G から生成される三重化したデータフローグラフ $\tilde{G}$ において、ひとつのボートした変数およびプライマリ出力からさかのぼって、プライマリ入力または他のボートにボートされた変数までの部分グラフをコーンと呼ぶ。(図 3.7)

ここで、三重化されたデータフローグラフから資源割り当てとスケジューリングをする際に必要となる 2 つ条件を導入する。

条件 1 与えられる計算アルゴリズムから生成される三重化したデータフローグラフのどのコーンの演算の集合または変数の集合に関して、互いに同じ演算器およびレジスタを含まない。

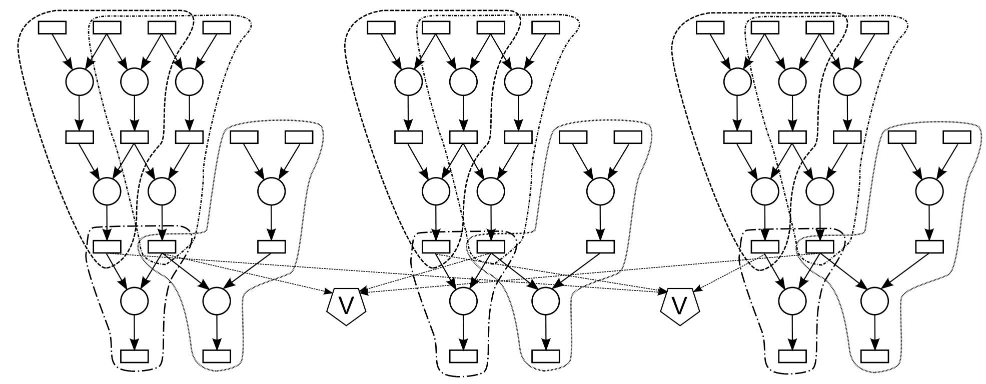

図 3.7: コーンの例

条件 2 与えられる計算アルゴリズムから生成される三重化されたデータフローグラフのコーンへの入力および出力に対してボートを行うボータはすべて異なるボータである。

このとき、以下の定理が成り立つことが知られている。

定理 1 条件１と条件２を満たすときかつそのときに限り、任意の単一故障に対して三重化されたデータフローグラフのそれぞれ三重化されたプライマリ出力の中に、誤りはそれぞれ高々一つである

図 3.9 にボータに着目した冗長化アルゴリズムによって耐故障化されたデータフローグラフを、図 3.9 に条件 1 と条件 2 を満たしす耐故障化されたスケジュールの例を示す。

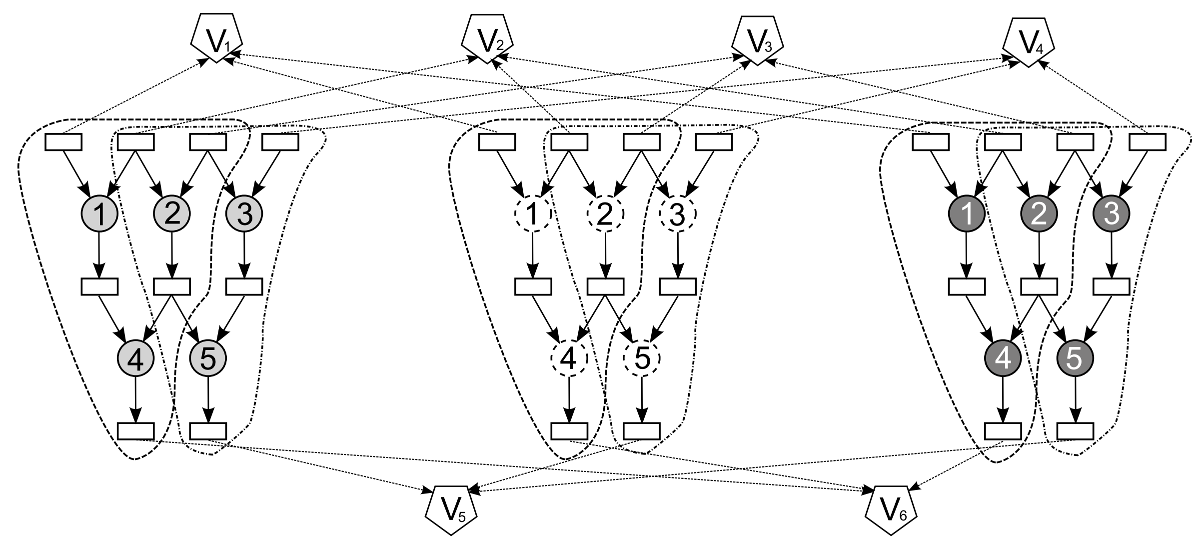

図 3.8: ボータに着目したデータフローグラフ

	ALU1	ALU2	ALU3	ALU4	ALU5	Voter1	Voter2	Voter3	Voter4
1						V_1	V_2	V_3	V_4
2	2	1	2	1	2				
3	3	3	3	4	1				
4	5	4	5	5	4				
5						V_6			V_5

図 3.9: 条件 1 と条件 2 を満たす耐故障化されたスケジュール例

耐故障データパスの合成問題を以下のように考えることができる。

入力

- 与えられる計算アルゴリズムから生成されたデータフローグラフ G_{dfg}
- 利用可能な演算器数 C_{num}

出力

- 耐故障化された演算スケジュール σ
- G_{dfg} へ挿入するボート演算の位置 $V_{location}$

制約

- σ は条件 1、条件 2 を満足する。
- σ は演算の先行関係を満足する。

なお、耐故障化されることで性能が悪化することを避けるためにボート演算は耐故障化されたデータフローグラフのスケジュール σ のスケジュール長 (実行に必要なステップ数)S_v が、ボートされていないデータフローグラフのスケジュール長 S と同等であり、かつ耐故障性が極大になるように挿入するべきだが、スケジューリングの手法が確立されていないためスケジュール長 S_v を得ることができない。ゆえに、ボート演算の挿入位置を発見することは困難である。そこで本研究では、ボート演算の挿入位置を考えない耐故障化された演算スケジュール σ を導出する。したがってボータ挿入位置 $V_{location}$ はあらかじめ与えられることとして考え、耐故障化された演算スケジュール σ を導出するための手

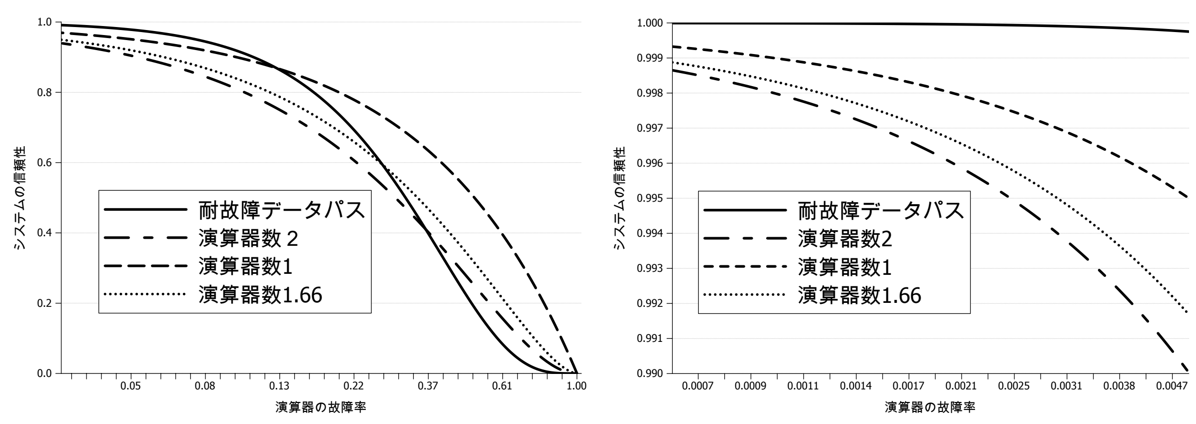

図 3.10: 演算器の故障率とシステムの信頼性の関係

法を提案する。ボート演算の挿入位置を決定する手法は、スケジューリング手法の確立を待ってから考える必要がある。

本研究で用いる耐故障技術の耐故障性について考える。議論を簡単にするため、すべての演算は演算器で実行可能とし、加算と乗算に分離しないこととする。また、演算器単体の故障確率を p とし、演算器数を α とする。このとき、耐故障化されていないデータパスが正しく動作する確率は $(1-p)^{\alpha}$ となり、耐故障化されているデータパスの正しく動作する確率は $(1-p)^{\alpha} + \alpha \cdot p \cdot (1-p)^{\alpha-1}$ となる [3]。ここでは、耐故障化されているデータパスの演算器数を 5 とする。また、耐故障化されていないデータパスの演算器数は、耐故障化されているデータパスより多重化されていないだけ少なく見積もる。耐故障化されていないデータパスの演算器数は $\frac{\text{演算器数}}{\text{多重化数}}$ で表し、演算器数は 5 で三重化されているため、耐故障化されていないデータパスの演算器数は $\frac{5}{3}$ となる。図 3.3 は、システム全体の信頼性を縦軸とし演算器単体の故障確率を横軸としたグラフである。耐故障データパスで演算器を 5 個用いた場合と、耐故障化されていないデータパスで演算器を 2 個、1.66 個、1 個用いた場合のシステム全体の故障率の関係を示している。耐故障データパスの信頼性が、耐故障化されていないデータパスで演算器数を 1.66 個用いる場合より下回るのは演算器単体の故障率が 0.26 を上回るときとなり、耐故障技術として有効なものである。

第4章　整数線形計画法によるデータパス合成

本研究では、与えられるデータフローグラフの演算の依存関係、利用可能な演算器の制約条件、ボータの割り当ての制約条件といった制限を、整数線形計画法による定式化を行い整数線形計画問題の求解ツールを用いてこれを解くことで厳密解となるスケジュールを得る。

4.1　ILP 定式化

三重化されたデータフローグラフ $\tilde{G}$ のうちの一つのデータフローグラフ G_l の演算 i のことを $O_i^{(l)}$ とする。変数 $x_{ij}^{(l)}$ は 1 のとき $O_i^{(l)}$ を実行ステップ j で実行することを示し、$v_{ik}^{(l)}$ は、1 のとき $O_i^{(l)}$ を演算器 k で実行することを示す。実行ステップの範囲は 0〜cs、演算器数は K、ボータ数は M とする。

1. 各演算 $O_i^{(l)}$ は 1 回のみ実行される。この制約は変数 $x_{ij}^{(l)}$ のすべてのコントロールステップ i の総和が 1 であることで表すことができる。すなわち、(4.1) 式で与えることができる。これをすべての $\{i, l\}$ の組み合わせでそれぞれ定式化する。

$$\sum_{j=0}^{cs} x_{ij}^{(l)} = 1 \tag{4.1}$$

2. 各演算 $O_i^{(l)}$ は、1 つの演算器のみで実行されなければならない。この制約は、変数 $v_{ik}^{(l)}$ のすべての演算器 k の総和が 1 であることで表すことができる。すなわち、(4.2) 式で与えることができる。これをすべての $\{i, l\}$ の組み合わせでそれぞれ定式化する。

$$\sum_{k=0}^{K} v_{ik}^{(l)} = 1 \tag{4.2}$$

3. 演算 $O_i^{(l)}$ と演算 $O_{i'}^{(l')}$ は同時に同じステップ j と同じ演算器 k を使うことはできない。この制約は (4.3) 式で与えられる。これをすべての $\{(i,i'),(l,l'),\mathrm{j}\}$ の組み合わせでそれぞれ定式化する。

$$x_{ij}^{(l)} + v_{ik}^{(l)} + x_{i'j}^{(l')} + v_{i'k}^{(l')} \leq 3 \tag{4.3}$$

図 4.1 は (4.3) 式を満たさない例で、演算 $O_1^{(1)}$ と $O_2^{(1)}$ が同一ステップで同一演算器を用いてスケジュールされているため、(4.3) 式の右辺が 4 となる。図 4.2 では、演算 $O_1^{(1)}$ と $O_2^{(1)}$ が同一ステップだが違う演算器を用いてスケジュールされているため、(4.3) 式の右辺が 3 となり (4.3) 式を満たす例である。

	ALU1	ALU2	ALU3
1	$O_1^{(1)}$ $O_2^{(1)}$		$O_1^{(2)}$
2			
3			

図 4.1: (4.3) 式を満たさない例

	ALU1	ALU2	ALU3
1	$O_1^{(1)}$	$O_2^{(1)}$	$O_1^{(2)}$
2			
3			

図 4.2: (4.3) 式を満たす例

4. 二つの演算 $O_i^{(l)}$ と $O_p^{(l)}$ の間に $O_i^{(l)}$ が $O_p^{(l)}$ に先行するという依存関係があるとき、$O_p^{(l)}$ が $O_i^{(l)}$ に先行してスケジュールされてはならない。この制約は、変数 x_{ij}^{l} を各コントロールステップ j で乗したもの総和することで演算 $O_p^{(l)}$ の実行ステップを表した $\sum_{j=0}^{cs} j \cdot x_{pj}^{(l)}$ と、変数 $x_{pj}^{(l)}$ を各コントロールステップ j で乗したものを総和することで演算 $O_i^{(l)}$ の実行ステップを表した $\sum_{j=0}^{cs} j \cdot x_{ij}^{(l)}$ とを比較し、$\sum_{j=0}^{cs} j \cdot x_{ij}^{(l)}$ が $\sum_{j=0}^{cs} j \cdot x_{pj}^{(l)}$ より小さくなる性質を利用することで表すことができる。従って、(4.4) 式で与えることができる。これをすべての $\{l, (O_i^l, O_p^l)\}$ の組み合わせでそれぞれ定式化する。

$$\sum_{j=0}^{cs} j \cdot x_{ij}^{(l)} < \sum_{j=0}^{cs} j \cdot x_{pj}^{(l)} \tag{4.4}$$

5. 二つの演算 O_i^l と $O_{i'}^l$ が三重化されたデータフローグラフの対応するコーンに含まれるとき、これらの演算は条件 1 により同一の演算器を利用してはならない。この制約は、$v_{ik}^{(l)}$ と $v_{ik}^{(l')}$ の和が 1 以下である性質を利用することで表すことができる。すなわち、(4.5) 式で与えられる。これをすべての $\{k, (O_i^l, O_{i'}^l), (l, l')\}$ の組み合わせでそれぞれ定式化する。

$$v_{ik}^{(l)} + v_{i'k}^{(l')} \leq 1 \tag{4.5}$$

図 4.3 は条件 1 を満たさない例であるため (4.5) 式を満たさない。なお、図 4.2 は (4.5) 式も満たす。

	ALU1	ALU2	ALU3
1	$O_1^{(1)}$	$O_2^{(1)}$	
2	$O_1^{(2)}$		
3			

図 4.3: (4.5) 式を満たさない例

変数 y_{ij} は 1 のときボート演算 O_i を実行ステップ j でボート演算することを示し、w_{im} はボート演算 O_i をボータ m を用いて実行することを示す。

6. ボート演算 O_i は 1 回のみ実行されなければならない。この制約は変数 y_{ij} の各コントロールステップでの総和が 1 であることで表すことができる。すなわち、(4.6) 式で与えることができる。

$$\sum_{j=0}^{cs} y_{ij} = 1 \tag{4.6}$$

7. ボート演算 O_i は 1 つのボータのみでボート演算されなければならない。この制約は w_{im} のすべてのボータの総和が 1 であることで表せる。すなわち、(4.7) 式で与えることができる。

$$\sum_{m}^{M} w_{im} = 1 \tag{4.7}$$

8. 同一時刻において、同一のボータでは高々一つのボート演算だけが実行されなければならない。この制約は (4.8) 式で与えられる。

$$y_{ij} + w_{im} + y_{i'j} + w_{i'm} \leq 3 \tag{4.8}$$

図 4.4 は (4.8) 式を満たさない例で、ボート演算 O_1 と O_2 が同一ステップで同一のボータを用いてスケジュールされているため、(4.8) 式の右辺が 4 となる。

	ALU1	ALU2	ALU3	voter
⋮	⋮	⋮	⋮	⋮
1	$O_1^{(1)}$	$O_1^{(2)}$	$O_2^{(2)}$	
2				O_1 O_2
3				

図 4.4: (4.8) 式を満たさない例

9. ボート演算 O_i と演算 $O_p^{(l)}$ の間に O_i が $O_p^{(l)}$ に先行するという関係があるとき、$O_p^{(l)}$ が O_i に先行してスケジュールされてはならない。この制約は、変数 $x_{pj}^{(l)}$ を各コントロールステップ j で乗したものを総和することで演算 $O_p^{(l)}$ の実行ステップを表したものと、変数 y_{ij} を各コントロールステップ j で乗したもの総和することで演算 O_i の実行ステップを表した $\sum_{j=0}^{cs} j \cdot y_{ij}$ とを比較し、$\sum_{j=0}^{cs} j \cdot x_{pj}^{(l)}$ が $\sum_{j=0}^{cs} j \cdot y_{ij}$ より小さくなる性質を利用することで表すことができる。したがって、この制約は (4.9) 式で与えられる。これをすべての $\{k, (O_i^l, O_p^l)\}$ の組み合わせでそれぞれ定式化する。

$$\sum_{j=0}^{cs} j \cdot x_{pj}^{(l)} < \sum_{j=0}^{cs} j \cdot y_{ij} \tag{4.9}$$

10. また、$O_{p'}^l$ が O_i に先行するという関係があるときの制約は、変数 y_{ij} を各コントロールステップ j で乗したもの総和することで演算 O_i の実行ステップを表した $\sum_{j=0}^{cs} j \cdot y_{ij}$ と、変数 $x_{p'j}^{(l)}$ を各コントロールステップ j で乗したものを総和することで演算 $O_{p'}^{(l)}$ の実行ステップを表したものとを比較し、$\sum_{j=0}^{cs} j \cdot x_{p'j}^{(l)}$ が $\sum_{j=0}^{cs} j \cdot y_{ij}$ より小さくなる性質を利用することで表すことができる。したがって、この制約は (4.10) 式で与えられる。これをすべての $\{k, (O_i^l, O_{p'}^l)\}$ の組み合わせでそれぞれ定式化する。

$$\sum_{j=0}^{cs} j \cdot y_{ij} < \sum_{j=0}^{cs} j \cdot x_{p'j}^{(l)} \tag{4.10}$$

11. なお、実行に必要なスケジュール長 S を最小化するには (4.11) 式を与え、S を最小化の目的変数とする。

$$\sum_{j}^{cs} j \cdot x_{ij}^{(l)} \leq S \tag{4.11}$$

4.2 実験結果

実験はすべてAMD Opteron250(2.4GHz) メモリ8GBの計算機上で行い、整数線形計画問題の求解ツールはILOG社のCPLEX 11とDonald Chai氏のgalena [2]の双方を用いた。なお、galenaでは制約式に含まれる変数は0または1のみの扱いとなるため、スケジュール長Sを含む制約式ではSを$128 \cdot S_7 + 64 \cdot S_6 + 32 \cdot S_5 + 16 \cdot S_4 + 8 \cdot S_3 + 4 \cdot S_2 + 2 \cdot S_1 + S_0$と置き換え、双方の整数線形計画問題の求解ツールの入力とした。

表4.1に実験結果を示す。演算器は加算乗算をともに処理でき、加算乗算ともに1ステップで実行可能とした。op.は演算数、voteはボート演算を挿入した位置、formulaは式数、variableは変数数をそれぞれ表す。G_{dfg}として、図5.16に示すdifferential equation benchmark(表4.1ではDEと表記) と図5.17に示すfour-order Jaumann wave digital filter benchmark (表4.1ではJWFと表記) を用い、演算器数C_{num}はいずれも5つ用いることとした。　演算数が10のdifferential equation benchmarkでは、galenaはいずれのパターンも1秒以内に解を導出したが、CPLEXではいずれのパターンも多くの時間がかかった。演算数が17のfour-order Jaumann wave digital filter benchmarkはdifferential equation benchmarkと比べ5倍以上の式を必要とするものの、galenaでは8.91秒という比較的短時間で解を導出できるパターンもあった。galenaはメモリの関係上解を導出できずに終了してしまうパターンがあるがいずれも暫定解では優秀な解を導出している。CPLEXでは計算開始1時間後の暫定解では優秀な解を導出できていない。

表4.1: 整数線形計画法によるスケジューリング結果

benchmark	op.	vote	formula	variable	CPLEX		galena	
					solution	time(sec)	solution	time(sec)
DE	10	i	26627	504	7	250†	7	0.38
		i,h	26570	520	8††	>3600	7	0.64
		e	26627	504	7	1627†	7	0.22
		e,h	26450	520	7	726†	7	0.45
		h	26748	504	8††	>3600	7	983.79
JWF	17	m	126311	1204	19††	>3600	13	8.91
		j	124001	1204	18††	>3600	12‡	>3600
		j,h,i	121527	1250	19††	>3600	12‡	>3600

† 暫定解が下界値へ達するまでの時間

‡ メモリ不足により停止したため暫定解を掲載

†† 計算開始後1時間以上経過したため計算を中止し暫定解を掲載

[2]http://www.eecs.berkeley.edu/ donald/code/galena/ フリーウェア

第5章　発見的手法によるデータパス合成

前章では、整数線形計画法により解を求めようとしたものの、解を導出するまでの時間が莫大なものとなり大規模なアルゴリズムでは解を得ることが困難となる。この章では、大規模なアルゴリズムを与えた場合でも高速に解を得ることのできる発見的手法について議論する。

5.1　概要

図5.1に発見的手法の概要となるフローチャートを示す。まず与えられるデータフローグラフ G_{dfg} とボート演算の挿入位置 $V_{location}$ より個々のコーンを生成し、それぞれへあらかじめ資源を割り与えた後、スケジューリングの際にコーンへ与えられた資源を用いてコーンごとにスケジュールする。これは、スケジュール中に逐次演算資源を割り当てる従来手法では耐故障性を持ったスケジュールを合成することが困難なためである。

図5.2にアルゴリズムを示す。まず、**InsertVote**関数により与えられる計算アルゴリズムから生成されたデータフローグラフ G_{dfg} へボート演算を挿入することで耐故障化されたデータフローグラフを生成し、G へ格納する。次に、**FigurationCone**関数により耐故障化されたデータフローグラフ G からコーンを生成する。変数 T は、詳しくは5.4章で説明することになるが**ResourceAllocation**関数で効率よく資源割当てをするために必要になる値で、変数 T は0から2の間で**while**ループを回るたびに0→1→2→0とローテーションする。**while**ループでは、**ChoiceCone**関数によりある基準に従ってスケジュールが最小化されるようなコーンを選択し、**ResourceAllocation**関数により選択されたコーンへ資源を割り当てる。最後に**Scheduling**関数によりコーン内の演算をハードウェアと時間上にマッピングする。これをコーンすべてがスケジューリングされるまで**while**ループにより繰り返す。

5.2　コーンの選択

スケジューリングをする際に、スケジュールするコーンの順序によってスケジュール結果に影響を与える。図5.3は、図5.17に示すfour-order Jaumann wave digital filter benchmarkをコーンの選択を考慮せずスケジュールしたものである。図5.24と比較すればわかるが、この例では1ステップ増加しておりスケジュール結果に影響を与えることがわ

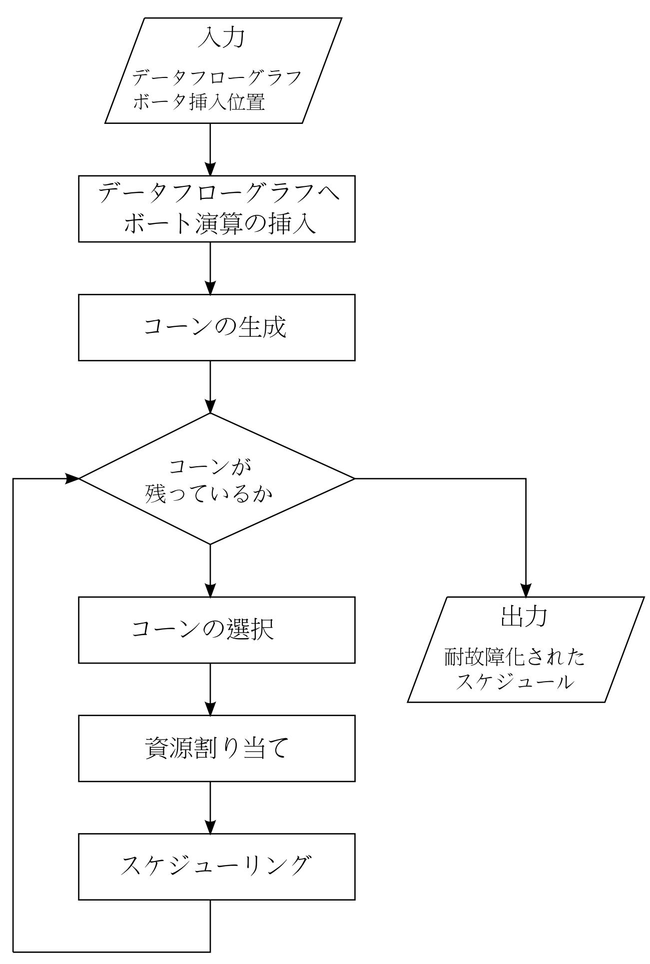

図 5.1: 提案手法フローチャート

入力　与えられる計算アルゴリズムから生成されたデータフローグラフ G_{dfg}
G_{dfg} へ挿入するボート演算の位置 $V_{location}$
利用可能な演算器数 C_{num}
出力　耐故障化された演算スケジュール σ

HeuristicAlgorism(G_{dfg}, $V_{location}$, C_{num}){
　G ← **InsertVote**(G_{dfg}, $V_{location}$) ;
　C ← **FigurationCone**(G) ;
　T_{array} ← {1, 2 ,0} ;
　$T = 0$;
　while($G \neq \emptyset$){
　　C_{picked} ← **ChoiceCone**(C) ;
　　o, o_{share} ← **ResourceAllocation**(C_{picked}, σ, T, C_{num}, C) ;
　　σ ← **Scheduling**(C_{picked}, σ, C, o, o_{share}) ;
　　G ← $G \backslash \{C_{picked}\}$;
　　T ← T_{array}[T] ;
　}
}

図 5.2: 発見的手法の概要アルゴリズム

かる。なお、スケジューリングルーチンは次節で紹介する手法を用いており、スケジューリングに性能差はない。

この節ではスケジュール長 S を最小化するために、与えられるデータフローグラフより生成される複数のコーンから、スケジューリングを開始すべきコーンを選択する方法について議論する。本研究で扱うスケジューリングは ALAP の結果をプライオリティに繁栄させたリストスケジューリングであるが、コーンの選択でもそれを踏襲した手法で対応する。そのため、各演算に Max - ALAP 値のラベルを付け、コーンを選択する基準としてラベルを活用する。(図 5.4)

ここで、次の定義を導入する。

定義 3　コーンに含まれる演算につけられているラベルの数値のうち、最大となるものを「コーンの最大ラベル」と呼ぶ。(図 5.5)

定義 4　コーンに含まれる演算に付けられているラベルの数値のうち、最小となるものを「コーンの最小ラベル」と呼ぶ。(図 5.6)

定義 5　任意のコーン C_x への入力となるすべての変数へ、プライマリ入力の値またはボートされた値が格納されているときコーン C_x を「選択可能なコーン」と呼ぶ。

	ALU1	ALU2	ALU3	ALU4	ALU5
Step1	a_0	a_1	a_2	b_2	b_0
Step2	d_0	d_1	d_2	e_2	e_0
Step3	g_0	g_1	g_2	h_2	h_0
Step4	j_0	j_1	j_2	c_2	c_0
Step5	b_1	k_2		f_2	$_0$
Step6	e_1	m_0	m_2	i_2	k_0
Step7	h_1	n_2	o_2	l_2	i_0
Step8	c_1	o_0	k_1		n_0
Step9	f_1	p_0	l_0		
Step10	i_1	q_0	p_2		
Step11	m_1	l_1	n_1		
Step12	o_1		q_2		
Step13	p_1				
Step14	q_1				

図 5.3: コーン選択を考慮しなかったスケジュール例 (ボート：j)

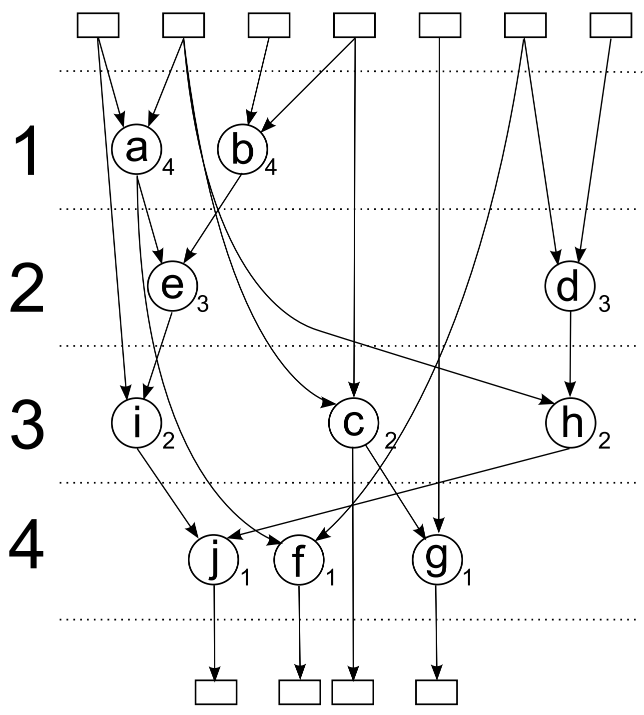

図 5.4: ラベル (Max - ALAP 値) の例

図 5.5 と図 5.6 は、最初に選択可能なコーンを太めの実線または点線で演算を囲っている。黒く示されている部分それぞれが最初に選択可能なコーンの最大ラベルの付いている演算と、コーンの最小ラベルの付いている演算である。

コーンを選択する具体的なアルゴリズムは図 5.7 の通りである。

図 5.7 のアルゴリズムは、与えられる計算アルゴリズムとボート演算の挿入位置により生成されるコーン集合 C を入力として与える。まず、コーン集合 C から **Executed** 関数によりすでにスケジュールされているコーンを導き、それを C から除した集合を C_c へ格納する。次のステップでは、コーン集合 C_c に含まれるコーンから **nonPreped** 関数によりコーンへの入力となるすべての変数の実行ステップが決定されていないコーンを導き、それを C_c から除した集合を再び C_c へ格納する。ここまでのステップで、スケジューリングへ進むことのできるコーンが選び出される。

これ以降が、実行に必要なステップ数を最小化するためのコーンを選択するステップとなる。max−ALAP のラベルを付けれたデータフローグラフのリストスケジューリングでは、スケジュールされた演算を除くデータフローグラフでのクリティカルパス上に存在する演算からスケジュールすることでスケジュール長 S を最小化する。そこで、コーンでもスケジュール済みのコーンを除くデータフローグラフでのクリティカルパス上に存在するコーンからスケジュールすることでスケジュール長 S を小さく抑える。そのために、C_c に格納されているコーンから、**MaxLabel**$_{\text{コーンの最大ラベル}}$ 関数によりコーンの最大ラベルの最大ラベルを導き、L_{max} へいったん格納し、**nonMatch**$_{\text{コーンの最大ラベル}}$ 関数によりコーンの最大ラベルが L_{max} とは一致しないコーンを導き、それを C_c から除する。

次に、**MaxLabel**$_{\text{コーンの最小ラベル}}$ 関数によりコーンの最小ラベルの最大ラベルを導いたものを L_{min} へいったん格納し、**nonMatch**$_{\text{コーンの最小ラベル}}$ 関数によりコーンの最小ラベルが L_{min} とは一致しないコーンを C_c から除する。これは、今回選ばれるコーンがスケジュールされることにより **nonPreped** 関数で導かれていたコーンから L_{max} に該当するラベルを持つコーンを除することで、次のコーンを選択する際に選択の幅を広げることで実行ステップを抑えるためである。

5.3 コーンへの演算資源割当て

選択されたコーンをスケジューリングする際に、耐故障性を維持するために耐故障条件を満たすよう演算資源を割り当てなければならない。この節では、スケジューリングする際に必要となる演算資源を選択されたコーンへ割り当てる手法について説明する。

ここで、次の定義を導入する。

定義 6 各演算器において、選択されたコーンで最初に実行される演算の実行が可能となるステップを始点として任意の演算器が最後に使われるステップまでの、演算器が利用されていない時間的空間の合計をそれぞれ「演算器のアイドルステップ数」と呼ぶ。

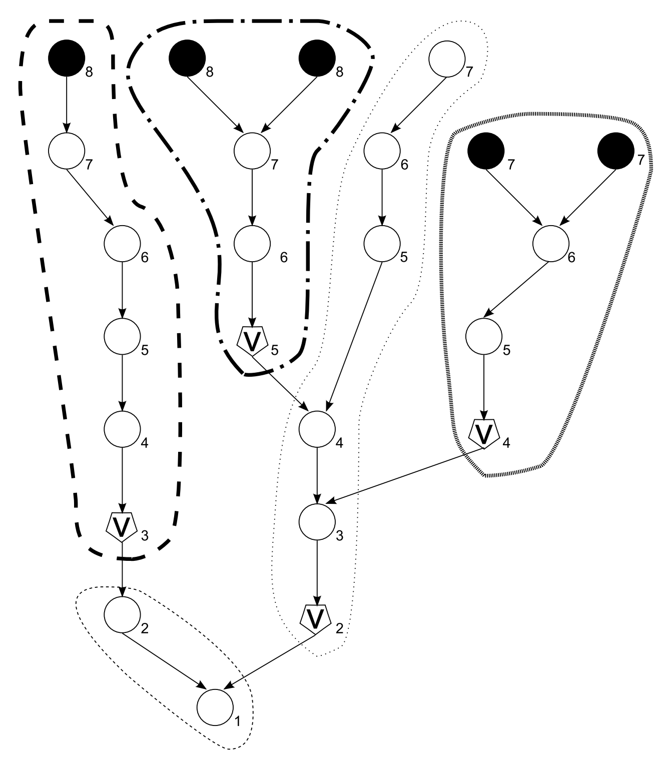

図 5.5: コーンの最大ラベル

三重化されたデータフローグラフの任意のコーンへ資源を割り当てる際、演算資源の利用状況によりそれぞれのコーンの終了ステップにばらつきが生じることがある。これを防ぐため、割り当てる演算器の利用度合いをもとに割り当てる演算器を動的に決定することで、各コーンへ与えられる演算器のアイドルステップ数の平滑化を目指す。

ここで、表 5.1 から表 5.3 の 3 つのテーブルを示す。

このテーブルは、三重化されたデータフローグラフの任意のコーンそれぞれに対して資源を割り当てる際にもちいるもので、数字は演算器のアイドルステップ数の量の順位を表しており三重化されたデータフローグラフの任意のコーンは、それぞれ異なる行の数字の配列に従って演算器を割り当てられる。

テーブルはスケジューリングを実施するたびにテーブル 1 からテーブル 2 へ、テーブル 2 からテーブル 3 へ、テーブル 3 からテーブル 1 へローテーションされて使用される。コーンが他のコーンと演算を共有している場合は、共有している演算を同一の演算器で割り当てるためにアイドルステップ数の多い演算器を各三重化されたデータフローグラフの任意のコーンそれぞれへ、共有している演算のために少なくとも 1 つの演算器を共有用として割り当てる。この場合、共有していない演算のために割り当てられる演算器は、テーブルから共有している演算に割り当てた演算器を除外したものをローテーションして用いることで対処する。

なお、このテーブルの数値は 1 列目から任意の列までの各行の合計値がある程度一致す

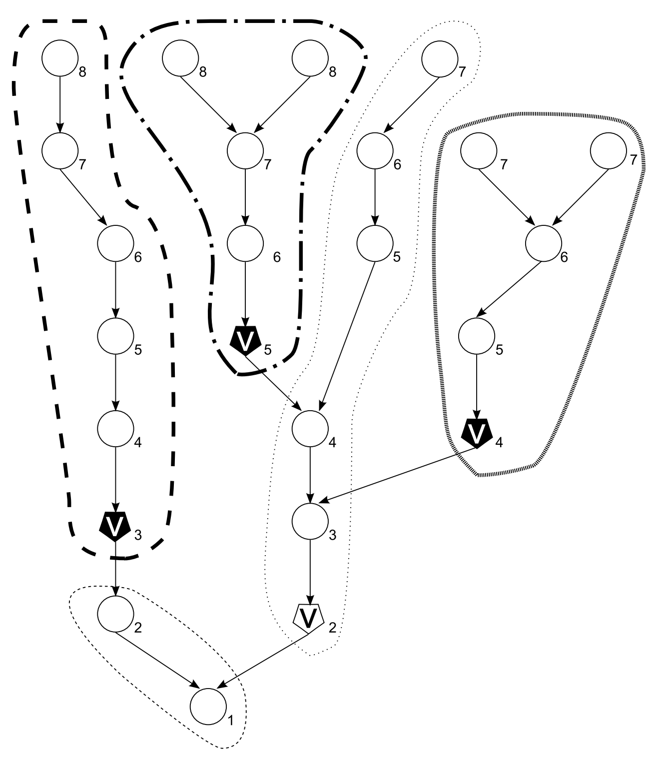

図 5.6: コーンの最小ラベル

入力　生成されたコーンの集合 C
出力　選択されたコーン C_{picked}

ChoiceCone$(C)\{$

$C_c \leftarrow C \backslash \{$ **Executed**(C) $\}$;

$C_c \leftarrow C_c \backslash \{$ **nonPreped**(C_c) $\}$;

$L_{max} \leftarrow$ **MaxLabel**$_{\text{コーンの最大ラベル}}(C_c)$;

$C_c \leftarrow C_c \backslash \{$ **nonMatch**$_{\text{コーンの最大ラベル}}(C_c, L_{max})$ $\}$;

$L_{min} \leftarrow$ **MaxLabel**$_{\text{コーンの最小ラベル}}(C_c)$;

$C_c \leftarrow C_c \backslash \{$ **nonMatch**$_{\text{コーンの最小ラベル}}(C_c, L_{max})$ $\}$;

return Head(C_c);

$\}$

図 5.7: コーンの選択ルーチン

表 5.1: 演算器割り当てテーブル 1

G_1	1，	6，	7，	12，	13，	18，	19
G_2	2，	5，	8，	11，	14，	17，	20
G_3	3，	4，	9，	10，	15，	16，	21

表 5.2: 演算器割り当てテーブル 2

G_1	2，	5，	9，	10，	14，	17，	21
G_2	3，	4，	7，	12，	15，	16，	19
G_3	1，	6，	8，	11，	13，	18，	20

表 5.3: 演算器割り当てテーブル 3

G_1	3，	4，	8，	12，	15，	16，	20
G_2	1，	6，	9，	10，	13，	18，	21
G_3	2，	5，	7，	11，	14，	17，	19

るように配置されており、それを 3 つのテーブルで行と列を互いにローテーションしながら配置されている。このように配置することで、三重化された各コーンにある程度均等に演算資源が配当されるよう配慮している。

図 5.8 にこのステップのアルゴリズムを示す。まず、それぞれの演算器のアイドルステップ数を導くために必要となる S_{min} を導くために **MinStep** 関数により C_{picked} に含まれる任意の演算が実行可能になるステップの最小値を S_{min} へ格納する。次に、選択されたコーン C_{picked} と同時に資源を割り当てる必要があるため、C_{picked} と演算を共有する他のコーンを **SearchSharedCone** 関数により $C_{relations}$ へ格納する。そして、演算器を割り当てるための基準となるアイドルステップ数を **CountIdleTime** 関数により数え o_i へ格納し、**DescendingSort** 関数により降順にソートすることで順位付けする。さらに、各コーンへ実際に演算を割り当てるために **Allocation** 関数により選択されたコーンと演算を共有する他のコーンへの演算器を割り当てる。

図 5.9 から図 5.12 は、**Allocation** 関数の処理についてわかりやすく記述したものである。**Allocation** 関数では、選択されたコーンに他のコーンと演算の共有している場合と共有していない場合では処理が異なる。演算を共有していない場合は、単純に $Table_T$ で与えられる演算器の順位のうち、C_{num} 位までの演算器を割り当てる。(図 5.9)

演算を共有している場合は複雑になる。まず、$Table_T$ で与えられる演算器の順位のうち C_{num} 未満の三の倍数位 r までを共有している演算用に割り当て、各コーンの演算に対しては T をローテーションしながら r より大きい順位の演算を割り当てる。(図 5.10 から図 5.12) 図 5.13 は、図 5.12 と $Table_T$ との関連性を示す。共有部分にテーブル 1 の最初の 1 列目の順位の演算器を割り当て、その他のコーンの演算へはテーブル 1 の 2 列目の順位

に基づく演算器割り当てと、ローテーション後のテーブル 2 の 2 列目の順位に基づく演算器割り当てがなされていることを表している。

入力 選択されたコーン C_{picked} : $C_{picked_1} \cup C_{picked_2} \cup C_{picked_3}$
スケジュール σ
使用するテーブル番号 T
生成されたコーンの集合 C
利用可能な演算器数 C_{num}
出力 演算資源割り当て o : $o_1 \cup o_2 \cup o_3$
C_{picked} と共有する演算を持つコーンの演算資源割り当て o_{share}
ResourceAllocation(C_{picked}, σ, T, C_{num}, C){
$S_{min} \leftarrow$ **MinStep**(C_{picked}) ;
$C_{relations} \leftarrow$**SearchSharedCone**(C_{picked}, C) ;
$o_i \leftarrow$**CountIdleTime**(S_{min}, σ) ;
DescendingSort(o_i) ;
$o, o_{share} \leftarrow$ **Allocation$_{\mathbf{picked}}$**($table_T$, o_i, C_{picked}, C_{num},$C_{relations}$) ;
return o, o_{share} ;
}

図 5.8: コーンへの演算資源割当てルーチン

5.4 スケジューリング

ここでは、選択されたコーンのスケジューリング手法について説明する。Hu のアルゴリズムとして知られる、演算のラベルである max−ALAP の値が大きい演算からスケジュールしていくリストスケジューリングで行う。図 5.14 のアルゴリズムは、スケジューリングの具体的なアルゴリズムである。入力として選択されたコーン C_{picked}、これまでされてきたスケジュール σ、演算資源割当て o と o_{share} を与える。

まず、選択されたコーン C_{picked} の未スケジュールの演算を **nonScheduled** 関数により集合 $O_{nonScheduled}$ へ格納する。次に、**while** ループ内で **MaxLabel** 関数により未スケジュール演算 $O_{nonScheduled}$ からラベルが最大となる演算のうちの一つを o_{picked} へ格納する。さらに、演算 o_{picked} の実行可能となる時刻を **PossibleStep** 関数により導き $S_{possible}$ へ格納する。その後、**ChoiceComputingUnit** 関数により演算 o_{picked} へ割り当て可能な時刻と演算器をそれぞれ c_{pick} と S へ格納する。そして、**OperateAllocation** 関数によりスケジュール σ へ割り当てられる。最後にスケジュールされ終わった演算 o_{picked} を未スケジュールの演算集合 $O_{nonScheduled}$ から除き、$O_{nonScheduled}$ が空集合になるまで **while** ループを繰り返す。

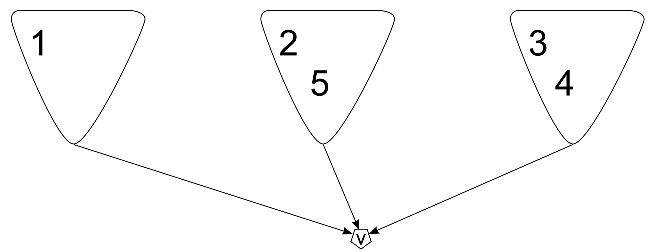

図 5.9: コーンへの資源割り当て (演算の共有が無い)

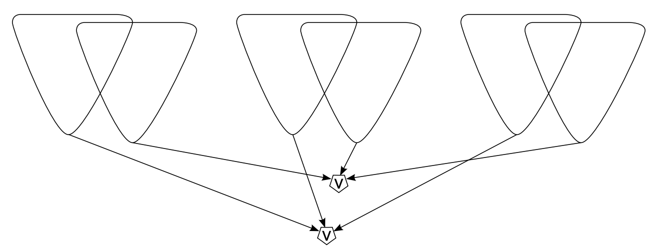

図 5.10: コーンへの資源割り当て (演算の共有がある) Step1 初期状態

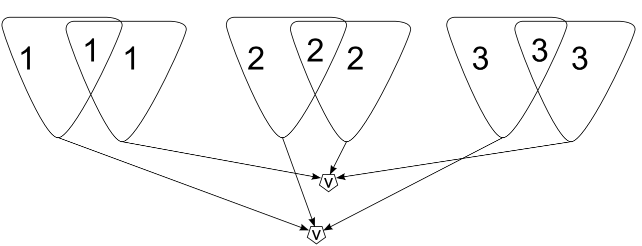

図 5.11: コーンへの資源割り当て (演算の共有がある) Step2 共有部分への割り当て

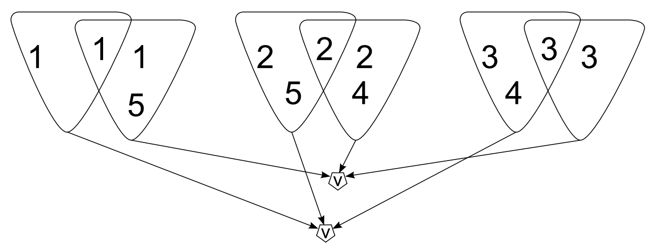

図 5.12: コーンへの資源割り当て (演算の共有がある) Step3 その他の部分への割り当て

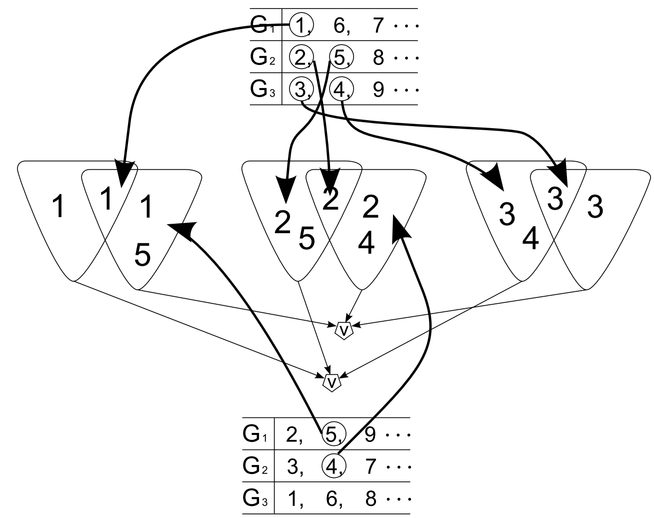

図 5.13: コーンへの資源割り当て (演算の共有がある) テーブルとの対応

まず、選択されたコーン C_{picked} と共有する演算を持つコーンに $table_T$(ここではテーブル 1) の最初の列の値から共有用の演算器を割り当てる。次に $table_T$ の 2 列目の値から選択されたコーン C_{picked} の共有されていない演算用として演算器を割り当てる。さらに、$table_{T_{array}[T]}$(同テーブル 2) の 2 列目の値から、選択されたコーン C_{picked} と共有演算を持つコーンの、共有されていない演算用として演算器を割り当てる。

入力 選択されたコーン C_{picked} : $C_{picked_1} \cup C_{picked_2} \cup C_{picked_3}$
　スケジュール σ
　演算資源割り当て o : $o_1 \cup o_2 \cup o_3$
　共有演算資源割り当て o_{share} : $o_{share_1} \cup o_{share_2} \cup o_{share_3}$
出力 スケジュール σ

Scheduling(C_{picked}, σ, o){
　$O_{nonScheduled} \leftarrow$ **nonScheduled**(C_{picked}) ;
　do {
　　$o_{picked} \leftarrow$**MaxLabel**($O_{nonScheduled}$) ;
　　$S_{possible} \leftarrow$**PossibleStep**(o_{picked}) ;
　　$c_{pick}, S \leftarrow$ **ChoiceComputingUnit**(o, o_{share}, $S_{possible}$, σ) ;
　　$\sigma \leftarrow$ **OperateAllocation**(c_{pick}, S, o_{picked}, σ) ;
　　$O_{nonScheduled} \leftarrow O_{nonScheduled} \backslash o_{picked}$;
　}**while**($O_{nonScheduled} \neq \emptyset$)
　return σ ;
}

図 5.14: スケジューリングルーチン

なお、**ChoiceComputingUnit** 関数は、図 5.15 に示すアルゴリズムで実装されている。**ChoiceComputingUnit** 関数では、演算資源割当て o と共有演算資源割当て o_{share}、演算 o_{picked} が実行可能になる時刻 $S_{possible}$、そしてスケジュール σ を入力とし、演算 o_{picked} をマッピング可能な演算器と時刻を出力する。

まず、利用可能な演算器を o_{temp} へ格納し、共有演算器を利用中であるかを判定するためのフラグ用変数 u へ 0 を代入する。次に、**while** ループで o_{temp} へ割り当てられた演算器のうちの一つを c_{pick} へ格納し、**OperateAllocation** 関数で、演算器 c_{pick} がステップ $S_{possible}$ へ演算 o_{picked} を割り当て可能かを確認後、割り当て可能であれば演算 o_{picked} を割り当てる。すでに演算器 c_{pick} がステップ $S_{possible}$ へ別の演算が割り当てられていた場合は、コーンへ割り当てられた別の資源を用いて割り当てるが、それでも割り当てできない場合は次のステップ $S_{possible} + 1$ で先の手順を繰り返す。

5.5 実験結果

実験は、発見的手法によるスケジューリングを Intel Core2Duo E7200 (3.6GHz) メモリ 4GB の計算機上で、解の正確さを比較するための整数線形計画法によるスケジューリングを AMD Opteron250(2.4GHz) メモリ 8GB の計算機上で行い整数線形計画問題の求解ツールとして Donald Chai 氏の galena を用いた。また、ボート演算以外の演算はすべて ALU によって行うこととし、ALU での実行にかかる時間はいかなる演算も 1 ステップを要することとした。なお、すべての実験で用いる演算器数は 5 とし、発見的手法では共有用演算器数を 3、他のコーンと共有していない演算用の演算器を 2 として割り当てた。

図 5.4 に実験結果を示す。また、そのときのスケジュールの例を図 5.20、図 5.21、図 5.22、図 5.23、図 5.24、図 5.25、図 5.26、図 5.27、図 5.28、図 5.29 に示す。benchmark は入力となるデータフローグラフである。DE は Differential Equation Benchmark(図 5.16) を、JWF は four-order Jaumann wave digital filter benchmark(図 5.17) を、IDCT は IDCT column-mise(図 5.18) を、16FFT は 16point Fast Fourier Transform benchmark(図 5.19) をそれぞれ示す。データフローグラフの頂点数はそれぞれ 10、17、67、81 で 16FFT が最も大きい。vote はボート演算の挿入位置を示し、heuristic は発見的手法によるスケジューリング結果で、ILP solver は整数線形計画法による解を参考値として記載した。なお、図 5.4 の ILP solver の項目で time が N/A となっている部分は、整数線形計画問題の求解ツールへ入力する制約式を記述したファイルが数百 MB に達し、求解ツールへ入力することができないため実験を行っていない。そのため、このような整数線形計画法では事実上計算することのできない大規模なデータフローグラフでの発見的手法の精度を評価するために ideal lower という値を与える。ideal lower は、データフローグラフの演算数から三重化されたデータフローグラフのうち一つのデータフローグラフあたりの演算器数で割ったもので、どのように最適化されてもこの数値未満のステップ数でのスケジュールは不可能となる下限の数値となる。

提案手法は、Differential Equation Benchmar ではボート位置によって悪化するパター

入力　演算資源割り当て o : $o_1 \cup o_2 \cup o_3$
共有演算資源割り当て o_{share} : $o_{share_1} \cup o_{share_2} \cup o_{share_3}$
演算が可能になる時刻 $S_{possible}$
スケジュール σ
出力　割り当て可能な演算器 c_{pick}
割り当て可能な時刻 $S_{possible}$

```
ChoiceComputingUnit(o, o_share, S_possible, σ){
♯ :
    o_temp ← o ;
    u ← 0;
    do{
        if(o_temp = ∅){
            if(u = 0){
                o_temp ← o_share ;
                u ← 1 ;
            } else {
                S_pssible ← S_pssible + 1;
                goto ♯;
            }
        }
        c_pick ← Head(o_temp) ;
        o_temp ← o_temp\c_pick ;
    }while(CheckSpace(c_pick, S_possible) = ∅)
    return c_pick,S_possible ;
}
```

図 5.15: 演算器選択ルーチン

ンがあるもののほとんどのパターンで厳密解と同程度の答えを導出することができた。また、four-order Jaumann wave digital filter benchmark に関しても厳密解より悪化するパターンがあるものの厳密解と同程度の答えを導出することができた。IDCT column-mise では、ideal lower が 41 に対しスケジュール長は 44 と 43 となりスケジュール長が長い。しかしながら 16point FFT benchmark では ideal lower が 49 に対しスケジュール長は 50 となり良い結果を導けている。計算時間では、提案手法は今回用いたベンチマーク回路ではいずれも短時間で終了することができた。

表 5.4: 発見的手法のスケジューリングによる実験結果

benchmark	op.	ideal† lower	vote	heuristic		ILP solver	
				solution	time(sec)	solution	time(sec)
DE	10	7	i	7	<0.001	7	0.38
			i,h	8	<0.001	7	0.64
			e	7	<0.001	7	0.22
			e,h	7	<0.001	7	0.45
JWF	17	11	m	13	<0.001	13	8.91
			j	12	<0.001	12‡	>3600
			j,h,i	14	<0.001	12‡	>3600
IDCT	67	41	36,37,38,39 40,41,42,43	44	0.001	N/A‡	>3600
			21,22,23,30,31 33,34,35,36,37 39,40,42,43	43	0.002	N/A	N/A
16FFT	81	49	23,24,25,26,27 31,32,33,34,35 36,37,38,39,40,41	50	0.002	N/A	N/A

† ideal lower $= \frac{\text{データフローグラフの演算数 } op.}{\text{演算器数 } 5/\text{多重化数 } 3}$

‡ メモリ不足により停止したため暫定解を掲載

5.6 その他の手法による事例

図 5.5 に、提案手法においてコーンの選択に異なる手法を用いた例を示す。

この章のアルゴリズムを計算機上に実装する際、与えられるデータフローグラフの頂点の接続情報を保持するデータ構造に FILO(First In Last Out) となる構造を構成している。その仕様上の性質と入力とするデータフローグラフの特殊性によると思われる原因に

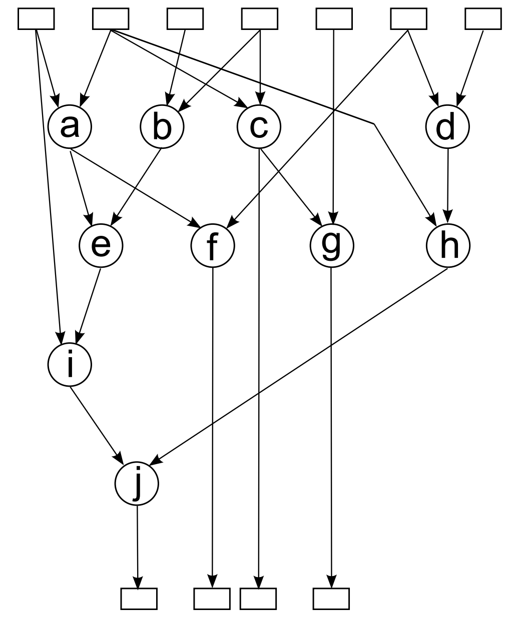

図 5.16: Differential Equation Benchmark

より、大規模なデータフローグラフにおいて意図しない改善が得られたため記録を残す。これは、コーンの選択手法の性能を検討する際に発見したもので、コーンの選択を実行せず直接スケジューリングルーチンへ渡すと発生する。なお、この節の説明はすべて状況説明となる。

入力となるデータフローグラフの特殊性について説明する。本来、データフローグラフはC言語などで記述される動作記述から変換されることで生成される。しかしながら、実験で入力としたデータフローグラフはASAPによりスケジューリングがなされていた状態の図形で提供されている。それを手作業で接続関係を記述したファイルを作成していた関係上、頂点の接続関係情報は左から右、上から下へ向かって記述される。また、ASAPでスケジューリングされたデータフローグラフの図形は、その見栄えを良くするために左側に位置する頂点のラベルが大きくなるように配置されている。

計算機上に実装したプログラムは、FILOの性質に従って接続関係を記述したファイルとは逆方向に、データフローグラフの図形を下から上へ、右から左へと読んでいく。すなわち、コーンの規模の小さいもの、ラベルの小さい演算から選択される傾向が高くなる。注意すべきことは必ずしもラベルの小さい演算から読まれるとは限らないことだ。図5.18のIDCT column-miseがその例外のもっともたるものだが、結果となるスケジュールは改善されている。従って、発生原因の詳細は不明と言わざるを得ない。

なお、入力となる頂点の接続関係を記述したファイルの接続関係情報の記述の順序をランダムに入れ替えると、この手法では結果が悪化し(図5.3)、提案手法では悪化しないことがわかっている。

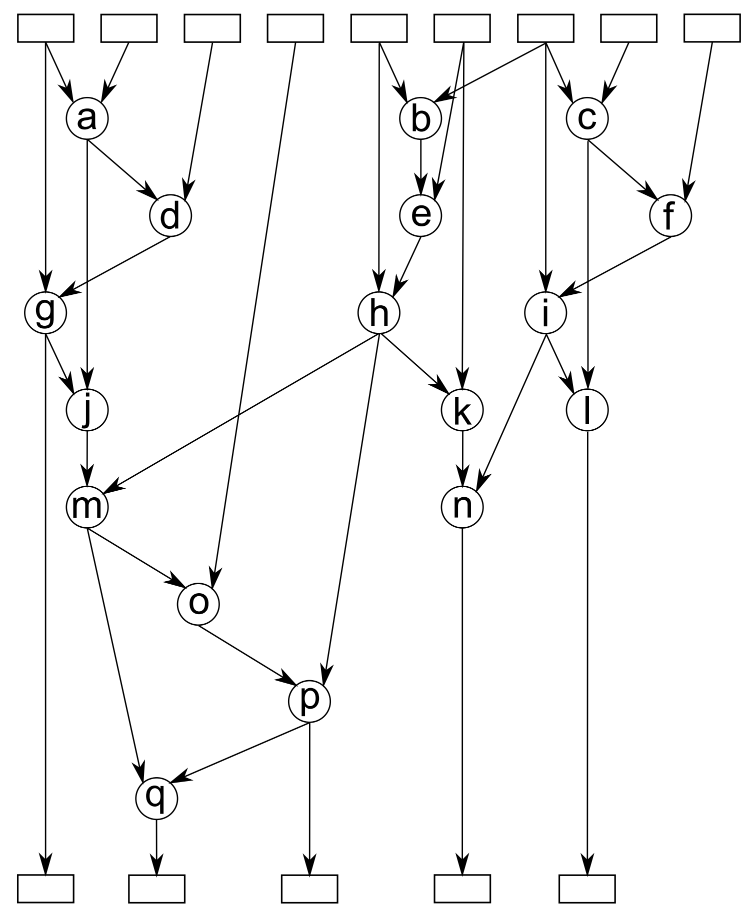

図 5.17: four-order Jaumann wave digital filter benchmark

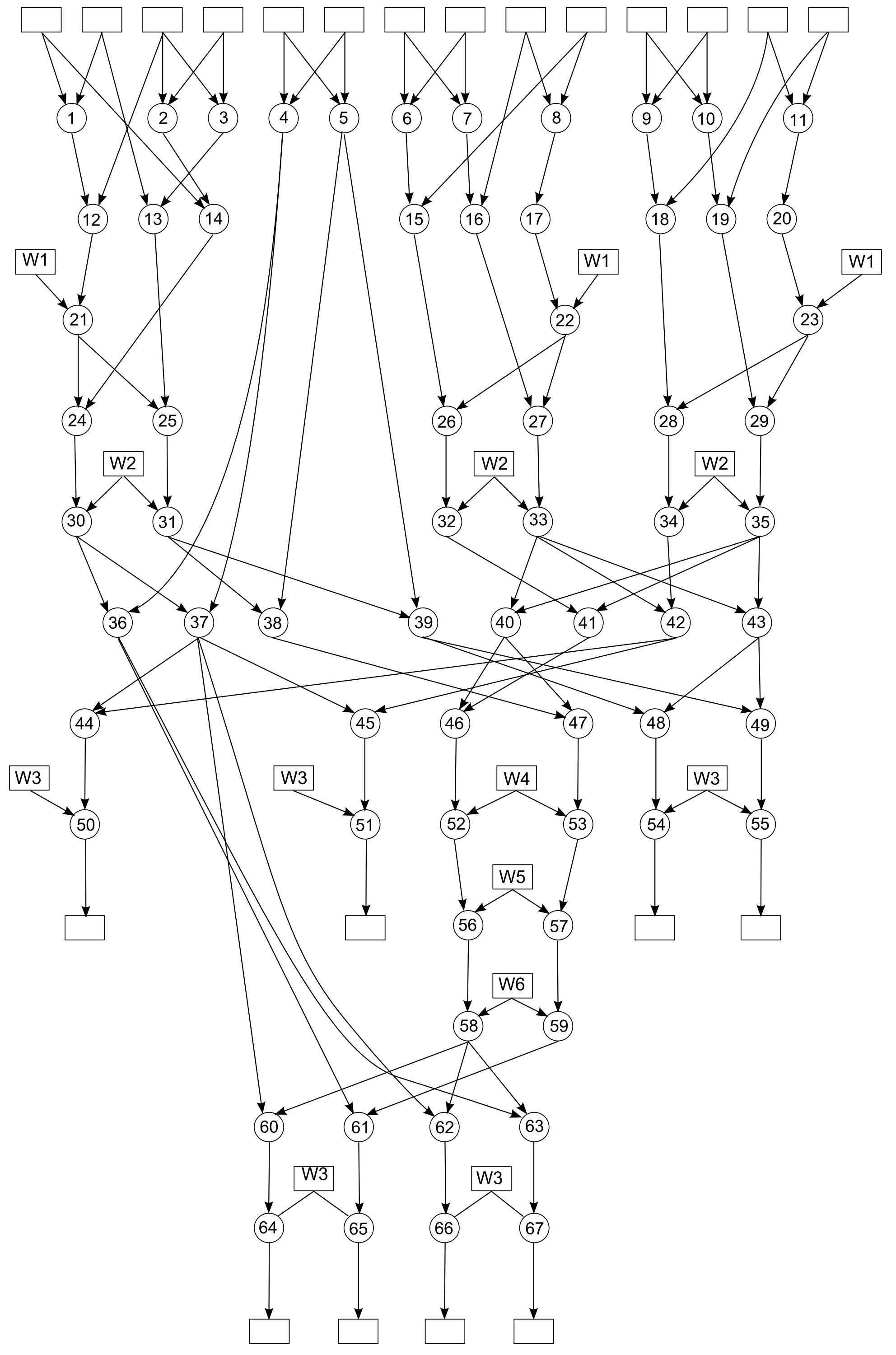

図 5.18: IDCT column-mise

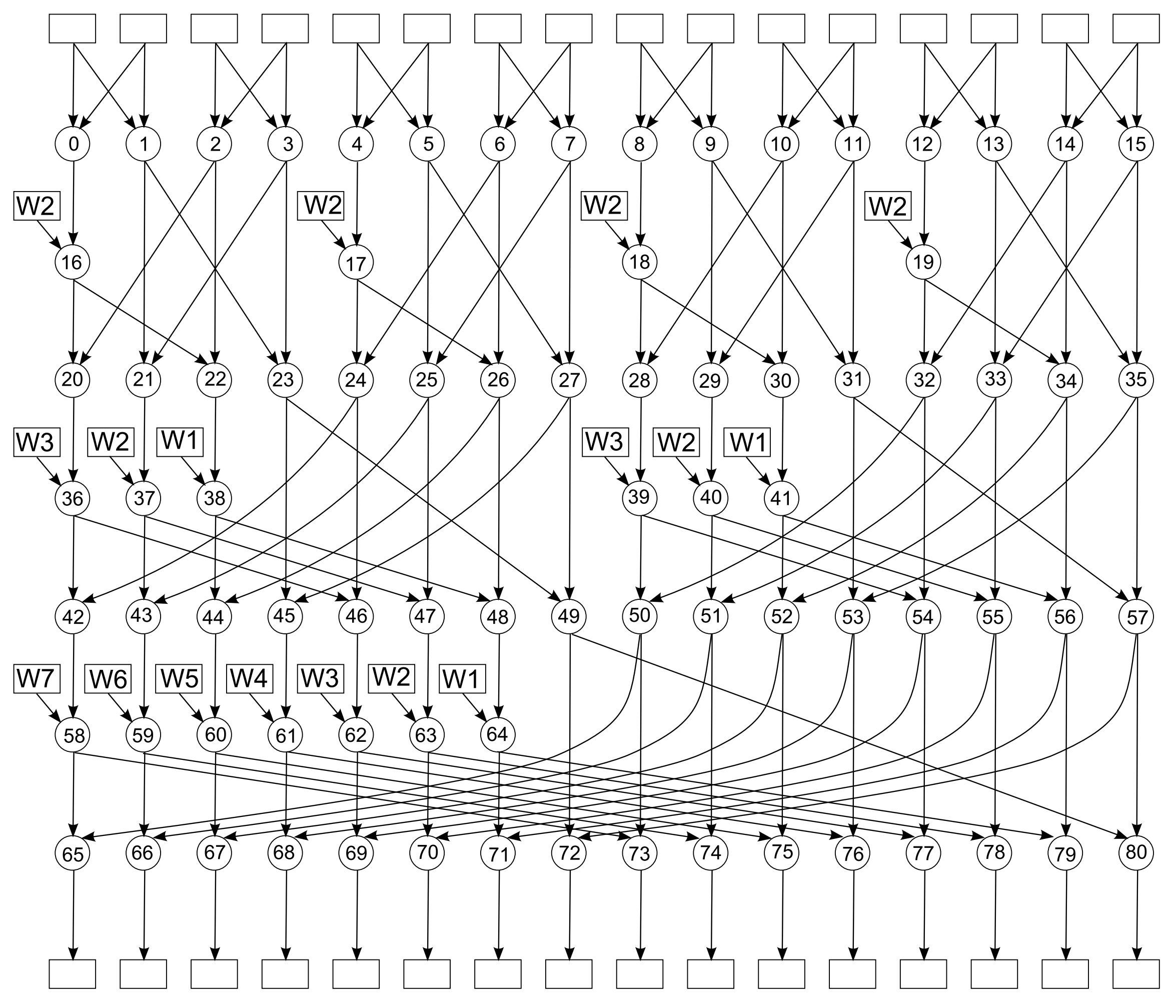

図 5.19: 16point FFT benchmark

	ALU1	ALU2	ALU3	ALU4	ALU5
Step1	a_0	a_1	b_2	b_1	b_0
Step2	f_0	d_1	a_2	e_1	e_0
Step3	d_2	h_1	e_2	i_1	i_0
Step4	h_2	c_0	i_2	f_2	f_1
Step5		$_g0$	d_0	c_1	c_2
Step6	j_2	j_1	h_0	g_1	$_g2$
Step7			jj_0		

図 5.20: DE 発見的手法によるスケジューリング (ポート：i)

	ALU1	ALU2	ALU3	ALU4	ALU5
Step1	a_0	a_1	b_2	b_1	b_0
Step2	f_0	d_1	a_2	e_1	e_0
Step3	d_2	h_1	e_2	i_1	i_0
Step4	h_2	c_0	i_2	f_2	f_1
Step5	c_1	g_0	d_0	c_2	
Step6	g_1		h_0	g_2	
Step7					
Step8	j_2	j_0	j_1		

図 5.21: DE 発見的手法によるスケジューリング (ボート：i, h)

	ALU1	ALU2	ALU3	ALU4	ALU5
Step1	a_0	a_1	b_2	b_1	b_0
Step2	f_0	d_1	a_2	e_1	e_0
Step3	d_2	h_1	e_2	f_2	f_1
Step4	h_2	c_2	e_0	c_1	c_0
Step5	i_2	i_1	h_0	i_0	g_0
Step6	j_2	j_1	j_0	g_1	
Step7		g_2			

図 5.22: DE 発見的手法によるスケジューリング (ボート：e)

	ALU1	ALU2	ALU3	ALU4	ALU5
Step1	a_0	a_1	b_2	b_1	b_0
Step2	f_0	d_1	a_2	e_1	e_0
Step3	d_2	h_1	e_2	f_2	f_1
Step4	h_2	c_0	d_0	c_1	c_2
Step5	i_0	i_1	h_0	i_2	g_2
Step6		g_0	g_1		
Step7	j_0	j_1		j_2	

図 5.23: DE 発見的手法によるスケジューリング (ボート：e, h)

	ALU1	ALU2	ALU3	ALU4	ALU5
Step1	a_0	a_1	a_2	b_2	b_0
Step2	d_0	d_1	d_2	e_2	e_0
Step3	g_0	g_1	g_2	h_2	h_0
Step4	j_0	j_1	j_2	c_2	c_0
Step5	b_1	k_0	k_2	f_2	f_0
Step6	e_1	m_2		i_2	m_0
Step7	h_1	o_2	n_2	l_2	o_0
Step8	c_1	p_2	m_1		p_0
Step9	f_1	q_2	o_1		q_0
Step10	k_1		p_1		i_0
Step11	i_1	n_0	q_1		
Step12	n_1	l_1	l_0		

図 5.24: JWF 発見的手法によるスケジューリング (ボート：j)

	ALU1	ALU2	ALU3	ALU4	ALU5
Step1	a_0	a_1	a_2	b_2	b_0
Step2	d_0	d_1	d_2	e_2	e_0
Step3	g_0	g_1	g_2	h_2	h_0
Step4	j_0	j_1	j_2	c_1	c_2
Step5	n_1	c_0	f_2		
Step6	e_1	f_0	i_2		
Step7	h_1	i_0			
Step8	f_1				
Step9	i_1	m_1	k_1	m_0	m_2
Step10	k_2	o_1		o_0	o_2
Step11	l_2	p_1	l_0	l_1	p_2
Step12	n_2	q_1	n_1	p_0	q_2
Step13		k_0		q_0	
Step14		n_0			

図 5.25: JWF 発見的手法によるスケジューリング (ボート：j,h,i)

	ALU1	ALU2	ALU3	ALU4	ALU5
Step1	a_0	b_1	b_2	a_2	a_1
Step2	b_0	e_1	e_2	d_2	d_1
Step3	d_0	h_1	h_2	g_2	g_1
Step4	e_0	c_1	c_2	j_2	j_1
Step5	g_0	f_1	f_2	m_2	m_1
Step6	h_0	k_1	i_2		k_2
Step7	j_0	i_1		k_0	n_2
Step8	m_0	n_1		l_2	l_1
Step9	c_0				
Step10	f_0		o_2	o_1	o_0
Step11	i_0		p_2	p_1	p_0
Step12	l_0		q_2	q_1	q_0
Step13				n_0	

図 5.26: JWF 発見的手法によるスケジューリング (ボート：m)

表 5.5: コーンの選択手法を変更した実験結果

benchmark	op.	ideal† lower	vote	heuristic		different	
				solution	time(sec)	solution	time(sec)
DE	10	7	i	7	<0.001	7	<0.001
			i,h	8	<0.001	9	0.001
			e	7	<0.001	8	<0.001
			e,h	7	<0.001	8	<0.001
JWF	17	11	m	13	<0.001	13	0.001
			j	12	<0.001	12	<0.001
			j,h,i	14	<0.001	13	<0.001
IDCT	67	41	36,37,38,39 40,41,42,43	44	0.001	41	0.002
			21,22,23,30,31 33,34,35,36,37 39,40,42,43	43	0.002	42	0.001
16FFT	81	49	23,24,25,26,27 31,32,33,34,35 36,37,38,39,40,41	50	0.002	49	0.002

† ideal lower $= \frac{\text{データフローグラフの演算数 } op.}{\text{演算器数 } 5 / \text{多重化数 } 3}$

	ALU1	ALU2	ALU3	ALU4	ALU5
Step1	11_0	11_1	11_2	6_2	6_1
Step2	8_0	8_1	8_2	15_2	15_1
Step3	20_0	20_1	20_2	1_1	1_2
Step4	10_0	10_1	10_2	3_1	3_2
Step5	6_0	17_1	17_2	12_1	12_2
Step6	17_0	23_1	23_2	21_1	21_2
Step7	23_0	19_1	19_2	13_1	13_2
Step8	19_0	22_1	22_2	25_1	25_2
Step9	22_0	29_1	29_2	26_2	26_1
Step10	15_0	35_1	35_2	32_2	32_1
Step11	29_0	7_1	7_2	41_2	41_1
Step12	26_0	16_1	16_2	31_1	31_2
Step13	35_0	27_1	27_2	5_1	5_2
Step14	32_0	33_1	33_2	38_1	9_0
Step15	41_0	40_1	1_0	43_2	40_2
Step16	7_0	39_2	28_0	9_1	43_1
Step17	16_0		12_0	18_1	18_0
Step18	27_0		21_0	28_1	28_0
Step19	33_0		13_0	34_1	34_0
Step20	38_2		25_0	40_0	42_0
Step21	43_0		31_0	42_1	2_2
Step22	39_1	46_1	5_0	2_1	14_2
Step23	46_2	38_0	9_2	14_1	24_2
Step24	52_2	52_1	18_2	24_1	30_2
Step25	56_2	56_1	28_2	30_1	4_2
Step26	37_2	58_1	34_2	4_1	36_2
Step27	58_2	36_1	42_2	37_1	46_0
Step28	47_0	47_2	39_0	47_1	52_0
Step29	53_0	53_2	2_0	53_1	56_0
Step30	57_0	57_2	14_0	57_1	58_0
Step31	59_0	59_2	24_0	49_2	59_1
Step32	55_2	49_0	30_0	48_1	49_1
Step33	48_0	55_0	4_0	54_1	55_1
Step34	36_0	37_0	54_0		
Step35		48_2			
Step36	63_2	62_1	63_0	63_1	61_1
Step37	67_2	66_1	67_0	67_1	65_1
Step38	61_0	61_2	62_2	62_0	60_0
Step39	65_0	65_2	66_2	66_0	64_0
Step40	45_0	54_2	60_1	60_2	45_2
Step41	51_0	45_1	64_1	64_2	51_2
Step42		51_1		44_1	44_2
Step43		44_0		50_1	50_2
Step44		50_0			

図 5.27: IDCT 発見的手法によるスケジューリング (ポート：36, 37, 38, 39, 40, 41, 42, 43)

	ALU1	ALU2	ALU3	ALU4	ALU5
Step1	11_0	11_1	11_2	8_2	8_0
Step2	20_0	20_1	20_2	17_2	17_0
Step3	23_0	23_1	23_2	22_2	22_0
Step4	8_1	1_1	1_2	1_0	10_0
Step5	17_1	12_1	12_2	12_0	19_0
Step6	22_1	21_1	21_2	21_0	29_0
Step7	10_1	10_2	7_2	7_0	35_0
Step8	19_1	19_2	16_2	16_0	7_1
Step9	29_1	29_2	27_2	27_0	16_1
Step10	35_1	35_2	33_2	33_0	27_1
Step11	3_1	3_2	3_0	9_0	33_1
Step12	13_1	13_2	13_0	18_0	9_1
Step13	25_1	25_2	25_0	28_0	18_1
Step14	31_1	31_2	31_0	34_0	28_1
Step15	9_2	2_2	2_0	2_1	34_1
Step16	18_2	14_2	14_0	14_1	40_1
Step17	28_2	24_2	24_0	24_1	5_0
Step18	34_2	30_2	30_0	30_1	4_2
Step19	40_2	40_0	$a55_1$	5_2	6_1
Step20	39_1	39_0	4_0	39_2	15_1
Step21	4_1	37_0	6_2	37_2	6_1
Step22	37_1	6_0	15_2	36_0	32_1
Step23	38_2	15_0	26_2	43_2	$_1$
Step24	47_2	26_0	32_2	42_1	46_1
Step25	53_2	32_0	41_2		52_1
Step26	57_2	41_0	46_2		56_1
Step27	59_2	46_0	52_2		58_1
Step28	60_1	52_0	56_2	62_1	36_2
Step29	64_1	56_0	58_2	66_1	38_0
Step30	43_1	58_0	62_2	60_2	47_0
Step31	62_0	60_0	66_2	64_2	53_0
Step32	66_0	64_0	43_0		57_0
Step33	42_0	36_1	42_2		59_0
Step34	49_2	38_1	49_1	49_0	48_1
Step35	63_2	47_1	55_1	63_0	63_1
Step36	67_2	53_1	48_0	67_0	67_1
Step37	61_2	57_1	54_0	55_0	61_0
Step38	65_2	59_1	54_2	48_2	$_0$
Step39	55_2	61_1	51_2	54_2	$_1$
Step40	45_0	65_1	44_1	45_1	44_2
Step41	51_0		50_1	51_1	50_2
Step42				44_0	
Step43				50_0	

図 5.28: IDCT 発見的手法によるスケジューリング (ポート:21, 22, 23, 30, 31, 33, 34, 35, 36, 37, 39, 40, 42, 43)

	ALU1	ALU2	ALU3	ALU4	ALU5
Step1	0_0	0_1	0_2	1_2	1_0
Step2	2_0	2_1	2_2	3_2	3_0
Step3	16_0	16_1	16_2	4_1	21_0
Step4	20_0	1_1	22_2	22_1	22_0
Step5	36_0	3_1	38_2	38_1	38_0
Step6	21_2	4_0	21_1	20_2	20_1
Step7	37_2	6_0	37_1	36_2	36_1
Step8	4_2	17_0	23_2	6_1	37_0
Step9	6_2	24_0	8_2	17_1	26_0
Step10	17_2	23_1	26_1	8_0	24_1
Step11	26_2	8_1	24_2	10_0	5_0
Step12	23_0	10_1	24_2	18_0	7_0
Step13	30_0	18_1	18_2	28_0	27_0
Step14	41_0	5_1	30_2	39_0	30_1
Step15	28_1	7_1	41_2	5_2	41_1
Step16	39_1	25_1	25_0	7_2	28_2
Step17	27_1	11_2	27_2	11_0	39_2
Step18	25_2	9_2	11_1	9_0	12_0
Step19	29_2	12_1	9_1	12_2	29_0
Step20	40_2	19_1	29_1	19_2	40_0
Step21	31_0	14_1	40_1	14_2	19_0
Step22	34_2	32_1	34_1	13_1	14_0
Step23	32_0	31_2	32_2	15_1	34_0
Step24	13_2	13_0	48_1	33_1	31_1
Step25	15_2	15_0	64_1	48_2	35_1
Step26	48_0	35_0	35_2	64_2	33_2
Step27	64_0	47_1	33_0	56_2	47_0
Step28	56_0	63_1	56_1	47_2	79_2
Step29	46_0	79_0	79_1	63_2	63_0
Step30	62_0	55_1	46_2	55_2	55_0
Step31	78_2	46_1	78_1	45_2	78_0
Step32	54_0	62_1	62_2	61_2	45_1
Step33	45_0	54_1	54_2	53_2	77_0
Step34	61_0	44_1	77_2	77_1	61_1
Step35	53_0	60_1	76_2	44_2	53_1
Step36	44_0	76_0	43_0	60_2	76_1
Step37	60_0	52_1	59_0	52_2	42_2
Step38	52_0	43_1	75_2	43_2	75_1
Step39	75_0	59_1	51_0	59_2	58_2
Step40	74_0	51_1	42_0	51_2	50_2
Step41	42_1	71_1	58_0	74_2	74_1
Step42	58_1	70_1	50_0	73_2	71_0
Step43	50_1	73_0	70_2	71_2	69_1
Step44	73_1	78_1	68_0	69_2	67_2
Step45	70_0	67_1	67_0	68_2	66_2
Step46	69_0	65_1	66_0	65_0	65_2
Step47	66_1	49_0	49_1	49_2	
Step48		57_0	57_1	57_2	
Step49	$_{8}00$	72_0		80_2	80_1
Step50	72_1				72_2

図 5.29: 16FFT 発見的手法によるスケジューリング
(ボート：23, 24, 25, 26, 27, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41)

第6章　結論

6.1　まとめ

本研究では、与えられる計算アルゴリズムとボータ挿入位置から誤り検出訂正可能なデータパスの生成手法を考案した。整数線形計画法によるスケジューリングでは、頂点数17程度の小規模のデータフローグラフを入力として与えた場合ボート演算の挿入位置の条件によっては実用時間内で解を導出できることを明らかにした。発見的手法によるスケジューリングでは、コーンに対して他のコーンとの共有関係を考慮して資源を割り当て、コーンごとにスケジュールし各演算へ資源を割り当てることで、きわめて短時間で解を導出することができ、かつ整数線形計画法による厳密解と遜色ない解を得ることのできる場合があることを明らかにした。また、コーンの選択によっては性能が改善される例があることを明らかにした。

6.2　今後の課題

提案手法では、すでにボート演算の挿入位置を決定していたが、スケジュール長を可能な限り抑えつつボート演算を増やすことで耐故障性をあげるためにこれを自動化する必要がある。また、コーンの選択手法によってスケジュール長が改善されることがわかっているため、これの最適な手法を考案する必要がある。レジスタ割り当ての実装も今後の課題である。

謝辞

本研究を行うにあたり、日頃から暖かくご指導いただいた金子峰雄教授、ならびに岩垣剛助教に心より感謝いたします。

また、有益なご助言、ご検討をいただいた金子研究室の皆様方に心より感謝いたします。

参考文献

[1] 尾塩和亮, ”冗長化アルゴリズムにおける資源共有を考慮した耐故障データパス合成”, 北陸先端科学技術大学院大学, 修士論文, 2002.

[2] Petra Michel, Ulrich Lauther, Peter Duzy, ”The Synthesis Approach To Digital System Design”, 1992.

[3] 当麻善弘, 南谷崇, 藤原秀雄, ”フォールトトレラントシステム構成と設計”, 1991.

[4] Kuang-Hua Huang, and Jacob A.Abraham, ”Algorithm-Based Fault Tolerance for Matrix Operations”, IEEE Transactions Computers, Vol.c-33, No.6, June 1984.

[5] G.Lakshminarayana, A.Raghunathan, N.K.Jha, ”Behavioral Synthesis of Fault Secure Controller/Datapaths Based on Aliasing Probability Analysis”, IEEETransactions Computers, Vol.49, Num.9, pp.865-885, September 2000.