整数線形計画法を用いたDNAコンピュータ制御コードの生成

阿部正佳† 萩谷昌己†

本論文では ANP-96 という DNA 計算の実験を自動的に行うロボットに対する，整数線形計画法を用いたコード生成について述べる．ANP-96 はプレートを置く 8 個のテーブル，プレートにさまざまな処理を行う IMU と呼ばれる装置，プレートを移動するロボットアーム等から成り立ち，与えられたプログラムに従って複数の実験操作を並列に実行することができる．DNA 計算の実験操作は種々の制約のもとで実行され時間もかかるので，複数の実験操作を自動的かつ効率的に実行することは重要である．一方で，現在の ANP-96 のプログラミング環境は，実験に本質的ではない低レベルな記述を必要とするため多くの手間を必要とし，誤ったプログラムにはロボットを破壊する危険もある．特に，プログラマは有限のテーブルを無駄なく割り当てるとともに，温度制御等の時間のかかる命令を効率良くスケジューリングしなければならない．我々は以上のようなプログラミングを自動化するために，テーブルのアロケーションと命令のスケジューリングを含む問題を，ANP-96 に依存しない一般的な抽象度で記述する枠組みを開発するとともに，その枠組みに基づく制御コード生成系を実装した．特に，この枠組みでは，各操作におけるリソースの受け渡しに関する記述を簡潔に表現することができる．また，上述の問題に対する最適な解を与える手法として，コンパイラの分野で提案された整数線形計画法による手法を利用した．この手法は処理時間が長く，したがって小規模のプログラムにしか適用できないのが欠点であるが，ANP-96 においては，一般にプログラムは比較的小規模であり，その実行回数とコストを考慮すると，時間をかけて最適化する価値がある．

Code Generation for a DNA Computer by Integer Linear Programming

Seika Abe† and Masami Hagiya†

In this paper, we describe code generation using integer linear programming for a robot called ANP-96 which automatically executes DNA computing experiments. The robot consists of 8 tables for placing plates, a device called IMU to do various operations on a plate, a robot arm that moves plates, etc., and can execute many experimental operations in parallel according to a given program. Since operations for DNA computing are executed under various constraints and may take long time, executing many operations automatically and efficiently is essential. On the other hand, the current programming environment of ANP-96 is troublesome since it requires programmers to specify low level details which are not essential to experiments, and it is even dangerous since erroneous programs may destroy the robot. In particular, programmers have to appropriately allocate a finite number of tables, and also efficiently schedule operations that take long time, such as those for thermal control. To automate such programming activities, we first designed a framework for specifying such problems including table allocation and operation scheduling, at an abstract level independent from ANP-96, and then implemented a code generator based on the framework. In particular, under this framework one can succinctly describe resource transfers in each operation. In the code generator, we employed the integer linear programming method developed in the field of compilers, which gives the optimal solution for the problems mentioned above. Although the method can only be applied to small programs due to its long execution time, programs of ANP-96 are in general relatively small, and worth optimizing because of the number of times and the cost to execute them.

1. はじめに

DNA コンピュータの研究は，Adleman が DNA 分子を用いてハミルトン経路問題を解いたことに始まるが[5]，この試みの意義は，DNA 分子によりさまざまな情報を表現できること，さらに，ハイブダイゼーション反応をはじめとする種々の反応を用いて，表現された情報を DNA 分子として操作できることを示した点にある．Adleman 以後，特に実用性な観点からは，DNA コンピュータの技術を遺伝子の発現解析や SNP 解析（染色体 DNA の個体間における差異の解析）に

† 東京大学情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

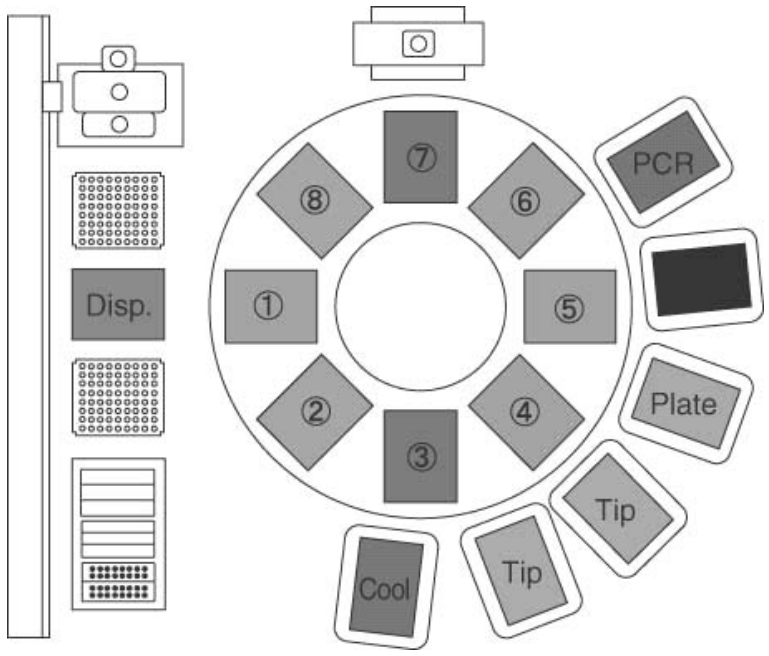


図 1 ANP-96 模式図
Fig. 1 ANP-96 simplified figure.

用いる試みが活発に行われている[2]．特に Suyama らは，DNA 分子を用いて 3SAT[16] 等の組合せ問題を解くように設計された実験操作が，発現解析や SNP 解析にそのまま適用できることを示した[14]．この場合，3SAT 等の組合せ問題は，遺伝子解析のためのベンチマークと考えることができる．ベンチマーク問題を正確かつ効率良く解くことができれば，遺伝子解析等の実用的な問題にも同様の正確さと効率を期待できる．

しかしながら，DNA コンピュータによる情報処理は，組合せ問題にせよ遺伝子解析にせよ，膨大な実験ステップを必要としているため自動化が不可欠である．そこで Suyama らは，本論文で扱うような自動ロボットを DNA コンピュータとして用いて遺伝子解析を行っている[4]．特に本論文で対象とする ANP-96 は，磁気ビーズをともなう実験操作（magtration）を完全に自動的に実行することができる．ところが，このような自動ロボットを制御するためのプログラムを書くことは自明ではない．構成要素の並列度を十分に引き出して，全体の実行時間を最短にすること自体が，困難な組合せ問題となってしまっている．

そこで本論文では，コンパイラの分野で開発された最適化技術を発展させ，自動ロボットのスケジュールを行う手法を提案している．本論文の最適化手法を用いることにより，たとえば従来よりも 10 倍実行が早くなる，というものではない．しかし，遺伝子解析のプログラムを実行する時間が，たとえば 1 回について 10 分から 9 分に短縮することができれば，100 人分の遺伝子解析を行う時間は，100 分=1 時間 40 分短縮できることになる．したがって，このような最適化を行うのに 10 分の計算時間がかかったとしても十分に見合うことになる．

ANP-96（Algorithmic NA Processor）[1]（**図 1**）は DNA 計算の実験を自動的に行うロボットであり，与えられたプログラムに従って複数の実験を同時に実行することが可能である．ANP-96 の詳細を説明することが本論文の目的ではないが，今後の議論のため，適当に抽象化，簡略化した形でこのロボットを説明する．ロボットの行うことを大まかにいえば，「入力」である複数の与えられた溶液から始まり，以下のような操作を繰り返し適用することで，新たな溶液（これも複数）を作り出すことである．この最終的な「出力」である溶液を調べることにより，実験が終了する．

(1) 特定の溶液を混合して，新しい溶液を作る（Mixing，懸濁）．
(2) 溶液から磁気ビーズという仕掛けにより，ある物質（溶液と考えてよい）を取り出すこと（Magtration，磁性粒子分離）．
(3) 溶液を特定の温度にまで暖めること（Warming，温度制御）．
(4) 溶液を特定の温度で一定時間維持すること

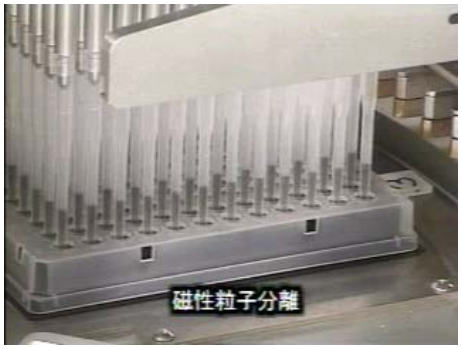


図 2 磁性粒子分離
Fig. 2 Magtration.

（Wait，維持）．

溶液は通常のプログラミング言語における「値」と考えることができる．この「値」を保持する「変数」に相当するものとして，プレートとチップがある．プレートは溶液を入れる器である（**図 2** の下の穴のあいた板）．またチップとはガラス製の先（下）細りの管（図 2 の複数の管）であり，溶液を保持することができる．実際にはチップは複数の管からなり，チップラックというチップを納める複数の穴を持つ容器に置かれており，チップに対する操作はこのチップの 1 セットに対して同時に行われる．プレートも同様に，複数の管に対応する複数の穴からなり，この穴の各々に溶液が入る．

プレートとチップの個数は事実上制限はないが，ロボットは上記の操作を有限（8 個）の実験場所（テーブル）で行うようになっている．また上記の (3) が可能なテーブルは 2 つしかなく，さらに (1) や (2) を行うための IMU という装置は 1 つしかない．以下がロボット本体の主要モジュールである．

(1) IMU（Integrated Magtration Unit）
(2) ストッカ
(3) ディスペンサ
(4) ターンテーブル
(5) ロボットアーム

ターンテーブルは上述の 8 個のテーブル（うち 2 つは温度制御可能）を備えた回転式の装置であり，それらテーブルの 1 つが IMU の直下にきている．IMU はチップを装着し，先細りの下部から溶液を排出，あるいは吸入することができる．たとえば懸濁という操作は IMU がチップを装着し，その下のテーブルに置かれたプレートにはチップと別の溶液が入っている状態で，IMU がチップの溶液の吸入，排出を繰り返すことで行われる．磁性粒子分離も同様であるが，溶液にビーズが入っており，IMU は磁気をかけてビーズをチップに付着させるという処理を行う．ストッカはプレートやチップラックを提供する装置，ディスペンサは溶液を提供する装置である．またロボットアームはテーブルに置かれたプレートやチップを別のテーブルに移動する装置である．

この有限個のテーブルは CPU の「レジスタ」に相当する．溶液である「値」は「変数」であるプレートやチップに保持されるが，それらが処理「演算」対象となるためには有限のテーブルのどれかに割り当てられる必要がある．あるテーブルはその仕事が終わったら，保持している「変数」を破棄し，別の変数を保持することができる．また，温度制御は時間のかかる命令であるが，IMU を必要としないので，その間に IMU は別の仕事を行うことができる．

これらを考慮したプログラムの最適化は，いわゆるレジスタアロケーションと命令スケジューリングにほかならない．しかしながら，現在の ANP-96 のプログラミングシステムは，このような最適化を行っておらず，プログラミングは上述の主要モジュールに対する直接的な命令からなる．すなわち実験に本質的ではない低レベルな記述を強制するため多くの手間を必要とし，誤ったプログラムにはロボットを破壊する危険もある．以下は，現在のシステムに対する自動化の要望事項である．

(1) 具体的なテーブル番号を意識したくない．
(2) 必要なくなったプレートやチップは自動的に破棄してほしい．
(3) 温度制御等に関して，時間的な制約を意識したくない．
(4) できる限りモジュールの並列性を活かしたい．
(5) 禁止された動作（禁止事項）の検出と排除．

これらの目標に向けて，我々は通常の言語処理系と同様な以下の構成を考えている．

(1) パーサ
ユーザは目標にあるような抽象度で記述できる高級言語でのプログラミングを可能にする．パーサがその意味的な正しさを検証し，禁止事項の検出を行う．そして正しいが最適化されていないアセンブラコードを出力する．ここでいうアセンブラコードとは，現在の ANP-96 の命令と同等のものだが，レジスタ（テーブル）アロケーションと最適な命令スケジューリングがまだ適用されていないものである．
(2) コード生成系
パーサが出力したアセンブラコードを入力とし，最適なレジスタ（テーブル）アロケーションと命

令スケジューリングを行い，最終的な ANP-96 の命令を生成する．

我々は，まずテーブルのアロケーションと命令のスケジューリングを含む問題を，ANP-96 に依存しない一般的な抽象度で記述するアセンブラ言語を定義するとともに，そのコード生成系を実装した．アロケーションとスケジューリングを同時に最適化する手法として整数線形計画法（ILP）に基づく手法を用いた．ILP に基づくこのような手法として，OASIC[9]，SILP[17] がある．これら既存の定式化と実装は通常の計算機を対象としたものであり，命令体系は RTL[13] に類似の固定されたセットを対象としている．一方我々は，アロケーションとスケジューリングが可能な抽象度のアセンブラ言語を対象としている．このため我々の手法の応用範囲は広いといえる．

このアセンブラ言語においては，命令は引数レジスタへの複数の同時代入の形をしている．既存の手法では命令が定義するレジスタはたかだか 1 つという条件が必要であったため，我々は SSA 変換を利用して，複数代入も扱えるようにした．また，このことによりすべてのリソースをレジスタとして統一的に表現することが可能となった．

2. 定 式 化

ここでは簡略化した ANP-96 の命令を通じて，ANP-96 のようなアロケーションとスケジューリングの最適化を含む機械制御のための汎用的なアセンブラ言語とコード生成系を提案する．

2.1 ANP-96 の素朴なアセンブラ言語表現

最初に，テーブルのみをレジスタと考え，基本的な命令について，素朴にアセンブラ言語を定めてみる．

- `newplate(t)`, `newrack(t)`
  テーブル `t` に新たなプレート，チップラックを置く．
- `attach(t)`, `detach(t)`
  IMU へテーブル `t` のチップを着，脱．
- `warm(t)`, `wait(t)`
  テーブル `t` の温度制御．
- `mix(t)`, `mag(t)`
  テーブル `t` の溶液で IMU のチップを懸濁，磁性粒子分離．
- `move(t1,t2)`
  テーブル `t1` 上の溶液をテーブル `t2` へ移動．

このアセンブラ言語は説明の都合上簡略化されている．たとえば，温度制御関係の命令では温度を指定する必要があるが，以下の議論で本質的ではないので簡略化している．30 度まで暖めるのであれば，30 という値を指定する必要があるが，これは命令のパラメータとはせず，命令コードで `warm30(t)` のように区別する．すなわち，引数は実レジスタを表す変数（以下仮想レジスタと呼ぶ）に限定する．また，命令の実行時間についての情報は明記されていないが，何らかの方法でコード生成系に提供されているものとする．

以下はこのアセンブラ言語によるプログラムの例である．ここで，テーブル `t0` にはある溶液の入った（分注済み）プレートが置かれているとする．

```
warm(t1)      t1 を暖める (時間かかる)
move(t0,t1)   t0 上の分注済みプレートを t1 へ
wait(t1)      t1 を特定温度で維持 (時間かかる)
move(t1,t2)   t1 上の成果物を t2 へ
mag(t3)       t3 を磁性粒子分離 (時間かかる)
mix(t2)       t2 を懸濁 (時間かかる)
```

次にコード生成系はこの各命令に実レジスタ（テーブル）を割り当て，かつ可能な限り並列実行するように命令をスケジューリングし，以下のようなアセンブラコードを出力する．横に並べられた命令は並列実行される．ここに，仮想レジスタ `t0`，`t2` には実レジスタ `T1` が，`t1` には `T3` が，`t3` には `T2` が割り当てられている．`warm`，`wait` の引数は温度制御可能な `T3`，`T7` から選ばれている．

```
warm(T3)              mag(T2)
move(T1,T3)
wait(T3)
move(T3,T1)
                      mix(T1)
```

このアセンブラ言語は直感的で分かりやすいが，その意味が明示的ではないという点が問題である．すなわち，コード生成系が実レジスタを割り当てる場合には，レジスタの生存区間解析が必要であるが，上記のような表現では各命令についての知識が必要である．たとえば温度制御に使われるレジスタ（テーブル）は 8 個あるテーブル中の 3，7 番のみが割り当て可能であること，`mix` と `mag` はともに IMU を必要とし，IMU は 1 つしかないので，これらの命令は並列には実行できないということ，等の知識である．このアセンブラ言語によれば，コード生成系の構造が ANP-96 に依存したものになってしまい，汎用性，拡張性に欠ける．

通常のコンパイラにおいては，命令の種類は一定で，その意味は厳密に定義されており，最適化フェーズからコード生成系に至るまで，その知識をもとに処理を行う．対象が通常のコンピュータであるから，あらかじめ定義された命令の組合せにより，広範囲の命令を

表現することができるので，命令についての知識を処理系が知っている，というアプローチは適切である．

しかし，我々の対象では四則演算のような基本的な操作をあらかじめ想定することは不可能である．このような状況でコード生成を行うための，アセンブラ言語への要求事項として以下が考えられる．

(1) あらかじめ意味の定まっている命令はない．
(2) 各命令の引数間のデータ依存関係が表現できる．
(3) 各命令の引数に割当て可能なレジスタ集合が指定できる．
(4) 各命令の実行時間が表現できる．
(5) 各命令の並列実行可能性の表現ができる．

2.2 汎用的アセンブラ言語

ここで，C で記述される以下のような命令 `foo` を考える．引数の型は割当て可能なレジスタ集合を表すと考える．たとえば `int = {i1, i2}`等．

命令 `foo(long x, int y, int z)`
意味 `x += y * z++;`

これを純粋な（副作用のない）関数 `foo_1`，`foo_2` の，`x`，`z` への同時代入の形で表すと，以下のようになる．

```
/* レジスタ集合の制約 */
x ∈ long, y ∈ int, z ∈ int
/* 同時代入 */
x = foo_1(x,y,z)
z = foo_2(z)
/* 関数の定義 */
long foo_1(long x,int y,int z)
  { return x+y*z; }
int  foo_2(int z)
  { return z+1; }
```

我々のコード生成系から見ると，引数のレジスタ集合，および引数のデータ依存関係のみあればよい．ここで，`foo_1`，`foo_2` は副作用のない関数であるから，その詳細がなくても引数のデータ依存関係は決定できる．よって，それら関数の定義を落とした情報を `foo` の「型」と考え，以下のように記述する．

```
foo : (x:long,y:int,z:int).x=(x,y,z),z=(z)
```

この記述のみから，`foo` のデータ依存関係は決定する．一般的に書けば，オペコード op の型宣言は以下の形となる．

$$op : (v_1{:}T_1, v_2{:}T_2, \ldots, v_n{:}T_n).\ v_1 = (v_i, \ldots),\ v_2 = (v_j, \ldots),\ \vdots\ v_n = (v_k, \ldots)$$

ここに $v_i{:}T_i$ は仮想レジスタ v_i に実レジスタの集合 T_i のある要素が割り当てられるということを意味する．また $v_i = (v_j, \cdots)$ は v_i が $(v_j, \cdots)$ に依存することを意味する．先の議論によれば，この $v_i = (v_j, \cdots)$ は $v_i = op\_i(v_j, \cdots)$ から関数 op_i を省略したものである．ここで op_i の定義が与えられていれば，v_i が単に $(v_j, \cdots)$ に依存する，というだけでなく，命令の実行により，引数レジスタの値がどのように変換するのかが決定できる．すなわち，仮想レジスタ v_i，v_j，v_k の値がそれぞれ v_i0，v_j0，v_k0 であり，op の型において $v_i = (v_j, v_k)$ があるとき，命令

$$op(v_1, \ldots, v_i, \ldots, v_n)$$

の実行により，仮想レジスタ v_i の値は $op\_i(v_j0, v_k0)$ となる．ここで，関数 op_i の定義が与えられていなくても，それを単なるコンストラクタと見なして命令を実行し，特定の点での特定の仮想レジスタの値をコンストラクタの合成の形で表すことは可能である．

このことを利用して我々のアセンブラプログラムの意味を定義する．すなわち，アセンブラプログラムの意味とは，プログラム開始時点での各仮想レジスタ v の値が $v0$ であるとし，上記の意味でプログラムを実行し，最後の命令の実行後に，その引数にある仮想レジスタの値のリストがそのプログラムの意味である．たとえば上の `foo` の例で，以下のプログラム

```
foo(x,y,z)
foo(x,y,z)
```

を実行した後の `x`，`y`，`z` の値のリストは以下のようになる．

```
foo_1(foo_1(x0,y0,z0),y0,foo_2(z0)),
y0,
foo_2(foo_2(z0))
```

ここまでで，アセンブラ言語への要求事項の (1) から (3) が満たされた．また要求事項の (4) については各命令の実行時間を指定すればよい．各命令の並列実行可能性の表現 (5) については，有限個の実行ユニットをレジスタとして表すことで可能であることを以下に示す．

2.3 レジスタによる装置と制約の表現

たとえば演算装置 ALU が 2 つあるような場合は，各演算命令の最初に ALU を使うことを以下のように明示すればよい．

```
/* レジスタ集合 */
ALU = {A1, A2}
/* 型宣言 */
op : (a:ALU, ...).a=()
```

これにより，たかだか 2 つの演算命令が並列に実行されることになる．ここで `op` の型の `a=()` は，`a` に他

の引数に依存しない特定の値が代入されることを意味する．実際に ALU に何かが代入されるわけではないが，この型付けにより「この命令は ALU を一瞬だけ使い，ALU には何も残さない」ということを表現することになる．

たとえば以下の例において，op1，op2 の間に誤ったデータ依存関係は生じない．

```
/* レジスタ集合 */
ALU = {A1, A2}
/* 型宣言 */
op1 : (a:ALU, x:X,y:Y).a=(),x=(y)
op2 : (a:ALU, x:X,y:Y).a=(),x=(y)
/* プログラム */
op1(a,x,y)
op2(a,z,w)
```

すなわち a=() により，各命令の a の生存区間はその命令の実行中のみになる．したがって，op1，op2 中の 2 つの a は異なる仮想レジスタと見なせ，同一のレジスタを割り当てることも可能であるし，またそれぞれに A1，A2 を割り当てて並列に実行させることも可能である．すなわちたかだか 2 つの並列実行がなされる．ここでもちろん「可能である」というのは，コード生成系がアロケーションとスケジューリングの双方を考慮しつつ最適化をする際，そのような選択肢が可能である，という意味である．

ここで特に ALU がシングルトン{A1}であった場合に生じる制約を考えると，op1，op2 は以前と同様にデータ依存関係を生じないが，同時には実行できない，という制約を表現することになる．ある命令とある命令の組合せに限り同時実行できない，という変則的な制約を，このような方法で記述することができる．

またもし上記の型において a=() が a=(a) であった場合，op2 の実行後の a の値は op2_1(op1_1(a0)) なので，op1，op2 の順序は入れ替えられないことになる．これを利用して，2 つの命令間の実行順序の制約も型付けにより表現できる．実行順序の制約は通常データの依存関係により自然に表現されているが，自然に表現できない場合でも，仮想的なレジスタ集合を導入することで表現可能となる．

次に入出力装置の型について考える．以下の out(o,x) は何らかの出力装置 o に，データ x を出力する命令を表現している．

```
/* レジスタ集合 */
OUT = {O1}
/* 型宣言 */
out : (o:OUT,x:DATA).o=(o,x)
/* プログラム：a,b,c を順に出力 */
out(o,a)
out(o,b)
out(o,c)
```

出力を単なる代入と考えると out の型は

```
out : (o:OUT,x:DATA).o=(x)
```

でもよさそうであるが，そうすると最後の命令での o の値は out_1(c0) となる．すなわちそれ以前の out 命令の実行順序の制約がなくなり，コード生成系は誤ったコードを生成してしまう．出力操作により，出力装置の次の状態は出力操作以前の状態と，出力されるデータに依存して決まると考えれば，上の型付けは自然である．

一方，入力操作により，入力装置の次の状態は，入力装置の以前の状態「のみ」で決定するものであるから，入力命令については以下のような型付けになる．

```
/* レジスタ集合 */
IN = {I1}
/* 型宣言 */
in : (i:IN,x:DATA).x=(i),i=(i)
```

2.4 ANP-96 のアセンブラ言語表現

以上の議論をもとに，IMU を明示し，温度制御可能なテーブルの集合もレジスタ集合として明示して，先の素朴な ANP-96 アセンブラ言語の例を書き換えると，以下のようになる．型の後ろの数字は実行時間（自然数）の指定である．

```
/* レジスタ集合 */
IMU = {IMU1}
TBL = {T1, T2, T3, T4, T5, T6, T7, T8}
T37 = {T3, T7}
/* 型と実行時間の宣言 */
newplate : (t:TBL).t=()                1
newrack  : (t:TBL).t=()                1
attach : (t:TBL,i:IMU).t=(t),i=(t)     1
detach : (t:TBL,i:IMU).t=(t,i),i=()    1
warm : (t:T37).t=(t)                  10
wait : (t:T37).t=(t)                  10
mix :(t:TBL,i:IMU).t=(t,i),i=(t,i)    10
mag :(t:TBL,i:IMU).t=(t,i),i=(t,i)    10
move : (t1:TBL,t2:TBL).t2=(t1),t1=()  1
/* プログラム */
warm(t1)      t1 を暖める (時間かかる)
move(t0,t1)   t0 上の分注済みプレートを t1 へ
wait(t1)      t1 を特定温度で維持 (時間かかる)
move(t1,t2)   t1 上の成果物を t2 へ
mag(t3,i)     t3 を磁性粒子分離 (時間かかる)
```

`mix(t2,i)` `t2` を懸濁 (時間かかる)

レジスタ集合 TBL と T37 は共通の実レジスタが入っている．このようにレジスタ集合同士は disjoint である必要はない．`newplate(newrack)` では新たなプレート（チップラック）がテーブル `t` に置かれる．命令実行後の `t` の値はそれぞれ `newplate_1()`，`newrack_1()` となる．`attach` 命令ではテーブル `t` 上のチップラックに収納されているチップが IMU`i` に装着される．この命令の型については，`t` はチップが抜き取られるという変化が起こるので `t=(t)` であり，IMU `i` は以前の IMU の状態とは無関係に `t` 上のチップが装着されるので `i=(t)` となる．装着されているチップをテーブル `t` に戻すのが `detach` である．その型については，まずチップが `t` に戻されることで `t` は変化するが，その結果は IMU `i` の状態，正確にはそれに取り付けられているチップの状態にも依存するので `t=(t,i)` であり，また IMU はチップがはずされて空の状態になるので `i=()` となっている．`warm`，`wait` のテーブル `t` は T37 でなければならず，`t` の以前の状態に依存した変化が起こるので `t=(t)` となる．`mix`，`mag` はテーブルと IMU の相互的な入出力といった型になる．

2.5 コード生成系

このようなアセンブラ言語を受け取り，コードを生成するコード生成系は以下のような処理系であり，その実装については次章で述べる．

- 入力
  - 各命令の型と実行時間
  - 各命令の引数が仮想レジスタであり，並列化されていないアセンブラ命令列
- 出力
  開始時刻が付加され，仮想レジスタが実レジスタに置き換えられたアセンブラ命令の集合．
- 目的
  プログラムの実行時間の最小化．
- 前提
  - 生存区間の重なる仮想レジスタには同一の実レジスタは割り当てられない．
  - 先行命令に依存する命令は，全先行命令が終了してから実行される．
  - 任意個数の命令が同時に実行できると仮定．すなわち，必要な並列性制約の明示的な表現が必要．

3. 実　　装

ここではコード生成系の実装について述べる．以後，単にレジスタといえば実レジスタを表すものとする．

レジスタアロケーションと命令スケジューリングの最適化は互いに排他的であることが知られている．たとえば以下の式を考える．

```
d = a * b + c
```

これをロード待ちが 1 クロックかかるレジスタマシンのアセンブラに変換すると，以下のようになる．

```
1 load  v1,a     ; v1=a
2 load  v2,b     ; v2=b
3 nop
4 mul   v3,v1,v2 ; v3=v1*v2
5 load  v4,c     ; v4=c
6 nop
7 add   v5,v3,v4 ; v5=v3+v4
8 store v5,d     ; d=v5
```

ここでまず仮想レジスタ vi にレジスタ Ri を割り当てた後，命令スケジューリングを行ったものが以下である．

```
1 load  R1,a
2 load  R2,b
3 nop
4 mul   R2,R1,R2
5 load  R1,c
6 nop
7 add   R2,R2,R1
8 store R2,d
  8 ステップ，2 レジスタ
```

一方命令スケジューリングの後にレジスタアロケーションを行ったものが以下である．

```
1 load  R1,a
2 load  R2,b
3 load  R3,c
4 mul   R4,R1,R2
5 add   R4,R4,R3
6 store R4,d
  6 ステップ，4 レジスタ
```

このように 2 つの最適化は適用順序により異なる結果となる．一般に `nop` を埋めるための命令移動によりレジスタの生存区間は長くなり，結果として多くのレジスタを消費し，レジスタ溢れの可能性が増大する．一方前者の適用順序ではレジスタアロケータがなるべく少ないレジスタで済むように努力する結果，多くのデータ依存関係を作ってしまい，スケジューラが命令を移動できる余地が少なくなる．

これら両者を同時に考慮した最適化問題は NP-HARD であることが知られており，効率的なアルゴリズムは存在しない．しかし，整数線形計画法（ILP）

に基づくもの[9),17)] や制約論理型プログラミングによる方法[8)] 等が提案されており，成功した実装例[12)] もある．以下に述べる実装は SILP[11),17)] をもとに，我々の定式化に沿うよう拡張したものである．

3.1 ILP（Integer Linear Programming）

ILP とは以下のような最適化問題である．ここに V は整数変数の集合で，以下 ILP 変数と呼ぶ．$\mathtt{v} \in V$ は目的変数，C は線形制約式の集合である．線形制約式とは整数係数の V の一次結合を $=$ または $\leq$ で比較した式である．

$$\begin{aligned} ILP &= (V, \mathtt{v}, C) \\ solver &: ILP \mapsto sol \\ sol &: V \to \boldsymbol{Z} \end{aligned}$$

$solver$ は与えられた ILP からその最適解を求めるプログラムである．最適解とは C のすべての制約を満たすような割当ての中で，目的変数 $\mathtt{v}$ を最小とするようなものである．

以下，コード生成系を ILP で表現する方法を述べる．まず入力プログラムは，必要な変換や前処理により，以下の形で与えられるものとする．

$$\begin{aligned} Prog &= \text{アセンブラ命令の配列} \\ Inst &= Prog \text{のインデックス} \\ Vreg &= \text{仮想レジスタの集合} \\ Rset &= \text{レジスタ集合の集合} \\ rset &: Vreg \to Rset \\ dura &: Inst \to \boldsymbol{N} \\ dep &= \{(i,j) \in Inst \times Inst \mid j \text{ が } i \text{ に依存 }\} \\ def &: Vreg \to Inst \\ refs &: Vreg \to \wp(Inst) \end{aligned}$$

以後命令 $i \in Inst$ とは $Prog$ 中の i 番目にある命令を意味する．$Vreg$ はプログラム中に現れる仮想レジスタの集合であり，前セクションの ANP-96 の例では $Vreg = \{\mathtt{t0}, \mathtt{t1}, \mathtt{t2}, \mathtt{t3}, \mathtt{i}\}$ である．$Rset$ はレジスタ集合の集合であり，先の例では $Rset = \{\{\mathtt{IMU1}\}, \{\mathtt{T1}, \mathtt{T2}, \ldots, \mathtt{T8}\}, \{\mathtt{T3}, \mathtt{T7}\}\}$ である．$rset$ は仮想レジスタに割当て可能なレジスタ集合を対応させる関数であり，命令の型宣言から決定する．$dura$ は命令の実行時間である．dep は命令 i が書き込んだ仮想レジスタを命令 j が参照している場合に生ずる（真の）依存関係であり，型宣言とプログラムの解析から求められる．

また，入力プログラムは SSA[7)] 形式で与えられると仮定する．この理由は 2 つある．1 つは逆依存や出力依存のような擬似的依存関係がなくなり，最適化の余地が増えることである．この仮定から，dep は真の依存関係のみを考慮する．より大きな理由は，我々のレジスタアロケーションは仮想レジスタ単位であり，後に説明する ILP への変換においては，SSA であるという仮定が必要となるからである．実際の実装では，与えられたプログラムへの前処理として SSA に変換している．この仮定より，仮想レジスタ v を定義する命令 $def(v)$ は一意に定まる．$refs(v)$ は v を参照する複数の命令である．

以上の入力に対して，各命令の開始時点を表す ILP 変数の集合 $V_{\mathtt{b}}$ を導入すると，ILP は実行ステップ数 $\mathtt{step}$ を最小とする以下のような問題となる．

$$\begin{aligned} ILP &= (V, \mathtt{step}, C) \\ V &= V_{\mathtt{b}} \cup V_{\mathtt{l}} \cup V_{\mathtt{r}} \\ V_{\mathtt{b}} &= \{\mathtt{b\_}i \mid i \in Inst\} \\ C &= C_{step} \cup C_{sch} \cup C_{reg} \\ C_{step} &= \{0 \leq \mathtt{b\_}i + dura(i) \leq \mathtt{step} \mid \\ &\qquad i \in Inst\} \end{aligned}$$

ここで C_{sch} は命令スケジューリング，C_{reg} はレジスタアロケーションの制約である．これらの制約の下で，ILP ソルバにより $\mathtt{step}$ が最小となるような解が求められる．$V_{\mathtt{b}}$ 以外の ILP 変数は後述する．以下，ILP 変数はこのようにタイプライタフォントで表す．また $\mathtt{v\_}x$ のような表現は，イタリック体の変数部分 x を適当な方法により変換して作った ILP 変数を表す．

3.2 制約の表現

命令スケジューリングの制約 C_{sch} は ILP で自然に表現できる．

$$C_{sch} = \{\mathtt{b\_}j - \mathtt{b\_}i \geq dura(i) \mid (i,j) \in dep\}$$

次に，レジスタアロケーションの制約について説明する．我々のアセンブラ言語では，各仮想レジスタ v には，ただ 1 つのレジスタ集合が割り当てられ，レジスタ集合どうしは disjoint である必要はなかった．しかし，後述するようにレジスタアロケーションの制約を ILP で表現するには，レジスタ集合どうしは disjoint であり，代わりに各仮想レジスタ v には複数の可能なレジスタ集合が対応するとした方が扱いやすい．よって，レジスタ集合の集合 $Rset$ と，仮想レジスタから実レジスタへの対応関数 $rset$ から作られる，以下の条件を満たす集合 $Rsets$ および関数 $rsets$ を導入する．

$$\begin{aligned}\bigcup Rsets &= \bigcup Rset\\ x, y \in Rsets &\rightarrow x \cap y = \emptyset\\ rsets &: Vreg \rightarrow \wp(Rsets)\\ \bigcup rsets(v) &= rset(v)\end{aligned}$$

この条件を満たす $Rsets$，$rsets$ は一意ではない．たとえば $Rsets = \{\{r\} \mid r \in Rset\}$ とシングルトンにまで分割してもよいが，ILP の求解の効率化のためには最小限の分割であることが望ましい．このような $Rsets$ は以下のアルゴリズムにより求められる．

$Rsets := \{\bigcup Rset\}$
foreach $s \in Rset$
  $Rsets := \bigcup\{\{x \cap s, x \setminus s\} \mid x \in Rsets\}$

すなわち，最も大きな分割から始めて，必要となる分割のみを繰り返すのである．$Rsets$ が求まれば，関数 $rsets$ は以下のように定義される．

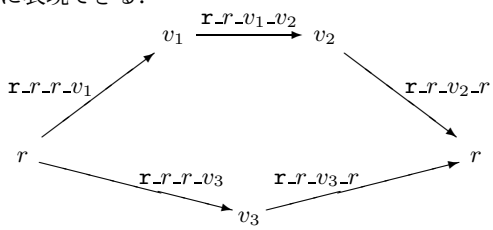
$$rsets(v) = \{s \in Rsets \mid \exists x \in rset(v)\ x \in s\}$$

以後，レジスタ集合といえば $Rsets$ の要素を指す．また，仮想レジスタ v のレジスタ集合といえば $rsets(v)$ のある要素を指す．

レジスタアロケーションの制約 C_{reg} が表現すべきものは，

(1) 各レジスタ集合 $r \in Rsets$ の要素数 $\#r$ を超えた割当てはできない．
(2) 各仮想レジスタ v は，そのレジスタ集合の要素が 1 つだけ割り当てられる．
(3) 生存区間が重なっている 2 つの仮想レジスタには異なるレジスタを割り当てる．

である．グラフカラーリング[6] 等の手法では，生存区間解析の結果に基づいてアロケーションを行うが，ここではスケジューリング最適化を同時に行うので，そのような解析をあらかじめ行うことはできない．そこでまず，項目 (3) を除いた制約を表現することを考える．すると項目 (1)，(2) のみの制約は ILP の典型的な応用例であるグラフの経路問題として，以下のように表現できる．



ここに v_i は r をレジスタ集合とする仮想レジスタであり，エッジはレジスタの受け渡しを表す．たとえば $r \rightarrow v_1$ というエッジはレジスタ集合 r から 1 つレジスタを取り出し，それを v_1 に割り当てる，ということを意味し，$v_1 \rightarrow v_2$ は v_1 が使用したそのレジスタを，次に v_2 に割り当てる，ということを意味している．また，$v_2 \rightarrow r$ は，そのレジスタを r へ返却するという意味である．以下のようなブール値（0 か 1）をとる ILP 変数を用意し，

$$\{\mathtt{r\_}r\mathtt{\_}x\mathtt{\_}y \mid x, y \in \{r, v_1, v_2, v_3\} \wedge x \neq y\}$$

その値が 1 のときに x から y へのエッジが存在するとしてグラフを表現すると，上述の項目 (1) は r から出ていくエッジを表す ILP 変数の総和が $\#r$ 以下という線形制約で記述でき，項目 (2) は各仮想レジスタ v_i に入ってくるエッジの総和と出ていくエッジの総和がともに 1 という線形制約で記述できる．

以下，項目 (3) の表現を含めた詳細を説明する．まず新たに以下の ILP 変数を導入する．

$$\begin{aligned}V_{\mathtt{l}} &= \{\mathtt{l\_}v \mid v \in Vreg\}\\ V_{\mathtt{r}} &= V_{\mathtt{rvv}} \cup V_{\mathtt{rrv}} \cup V_{\mathtt{rvr}}\\ V_{\mathtt{rvv}} &= \{\mathtt{r\_}r\mathtt{\_}v_i\mathtt{\_}v_j \mid\\ &\qquad r \in Rsets \wedge v_i, v_j \in Vreg\\ &\qquad \wedge r \in rsets(v_i)\\ &\qquad \wedge r \in rsets(v_j)\}\\ V_{\mathtt{rrv}} &= \{\mathtt{r\_}r\mathtt{\_}r\mathtt{\_}v \mid\\ &\qquad r \in Rsets \wedge v \in Vreg\\ &\qquad \wedge r \in rsets(v)\}\\ V_{\mathtt{rvr}} &= \{\mathtt{r\_}r\mathtt{\_}v\mathtt{\_}r \mid\\ &\qquad r \in Rsets \wedge v \in Vreg\\ &\qquad \wedge r \in rsets(v)\}\end{aligned}$$

ここに $V_{\mathtt{l}}$ は各仮想レジスタの生存区間長を表す ILP 変数の集合である．$V_{\mathtt{r}}$ は前述のグラフのエッジを表現するものであり，便宜上，仮想レジスタ間のエッジ（$V_{\mathtt{rvv}}$），レジスタ集合から出ていくエッジ（$V_{\mathtt{rrv}}$），およびレジスタ集合に入るエッジ（$V_{\mathtt{rvr}}$）の和集合として定義している．レジスタアロケーション制約 C_{reg} は便宜上 C_{reg_1} から C_{reg_6} に分けて記述する．

$$\begin{aligned}C_{reg} &= C_{reg_1} \cup C_{reg_2} \cup \cdots \cup C_{reg_6}\\ C_{reg_1} &= \{0 \leq v \leq 1 \mid v \in V_{\mathtt{r}}\}\\ C_{reg_2} &= \{\mathtt{b\_}j - \mathtt{b\_}i \leq \mathtt{l\_}v \mid\\ &\qquad v \in Vreg\\ &\qquad \wedge i = def(v) \wedge j \in refs(v)\}\end{aligned}$$

ここに C_{reg_1} は $V_{\mathtt{r}}$ の値をブール値に制限するものである．C_{reg_2} は生存区間 $V_{\mathtt{l}}$ に関する制約である．ここで以下の補助関数を定義する．

$$
\begin{aligned}
outr &: Rsets \to \wp(V_{\mathtt{rrv}}) \\
outr(r) &= \{\mathtt{r\_}r'\mathtt{\_}r'\mathtt{\_}v \in V_{\mathtt{rrv}} \mid r' = r\} \\
outv &: Vreg \to \wp(V_{\mathtt{rvv}} \cup V_{\mathtt{rvr}}) \\
outv(v) &= \{\mathtt{r\_}r\mathtt{\_}v'\mathtt{\_}x \in V_{\mathtt{rvv}} \cup V_{\mathtt{rvr}} \mid \\
&\qquad v' = v\} \\
outvr &: Vreg \times Rsets \to \wp(V_{\mathtt{rvv}} \cup V_{\mathtt{rvr}}) \\
outvr(v,r) &= \{\mathtt{r\_}r'\mathtt{\_}v'\mathtt{\_}x \in V_{\mathtt{rvv}} \cup V_{\mathtt{rvr}} \mid \\
&\qquad r' = r \wedge v' = v\} \\
invr &: Vreg \times Rsets \to \wp(V_{\mathtt{rvv}} \cup V_{\mathtt{rrv}}) \\
invr(v,r) &= \{\mathtt{r\_}r'\mathtt{\_}x\mathtt{\_}v' \in V_{\mathtt{rvv}} \cup V_{\mathtt{rrv}} \mid \\
&\qquad r' = r \wedge v' = v\}
\end{aligned}
$$

ここに $outr(r)$ はレジスタ集合 r から出ていくエッジ，$outv(v)$ は仮想レジスタから出ていくすべてのエッジ，$outvr(v,r)$ は仮想レジスタ v から出ていくエッジでレジスタ集合 r のもの，$invr(v,r)$ は仮想レジスタ v に入ってくるエッジでレジスタ集合 r のものを，それぞれ表している ILP 変数の集合である．以下が C_{reg} の残りである．

$$
\begin{aligned}
C_{reg_3} &= \{\textstyle\sum outr(r) \le \#\, r \mid \\
&\qquad r \in Rsets \wedge 1 \le \#\, outr(r)\} \\
C_{reg_4} &= \{\textstyle\sum invr(v,r) = \sum outvr(v,r) \mid \\
&\qquad v \in Vreg \wedge r \in rsets(v)\} \\
C_{reg_5} &= \{\textstyle\sum outv(v) = 1 \mid v \in Vreg\} \\
C_{reg_6} &= \{\mathtt{when}(\mathtt{r\_}r\mathtt{\_}v_1\mathtt{\_}v_2 = 1)\mathtt{b\_}j - \mathtt{b\_}i \ge \mathtt{l\_}i \mid \\
&\qquad \mathtt{r\_}r\mathtt{\_}v_1\mathtt{\_}v_2 \in V_{\mathtt{rvv}} \\
&\qquad \wedge i = def(v_1) \wedge j = def(v_2) \\
&\qquad \wedge (i,j) \notin dep\}
\end{aligned}
$$

ここに C_{reg_3} はレジスタ集合 r の要素数を超えない，という制約である．また C_{reg_4} と C_{reg_5} は，各仮想レジスタにただ 1 つのレジスタが割り当てられるという制約である．C_{reg_4} において，$r \in rsets(v)$ は v の本来のレジスタ集合 ($= \bigcup rsets(v)$) の一部なので，その制約式により等しい両辺の値は 0 または 1 であり，C_{reg_5} によりその $r \in rsets(v)$ のどれか 1 つが割り当てられることになる．最後の C_{reg_6} は，もし $\mathtt{r\_}r\mathtt{\_}v_1\mathtt{\_}v_2 = 1$ すなわち，v_1 から v_2 へレジスタが渡されるのであれば，v_2 の定義は v_1 の定義後，v_1 の生存区間長以上，後でなければならない，という制約である．ここでもし $(i,j) \in dep$ であれば，もとのその制約はデータ依存として記述されているはずだから不要である．ここでもし $\mathtt{r\_}r\mathtt{\_}v_1\mathtt{\_}v_2 = 0$ ならばそのような制約は必要なく，v_1 と v_2 は生存区間が重なっていてもかまわない．すなわちどうしても ILP 変数 $\mathtt{r\_}r\mathtt{\_}v_1\mathtt{\_}v_2$ の値により制約を「動的に」制御する必要がある．

3.3 ILP による条件式の表現

e をブール値の ILP 変数とし，線形制約式 $e = 1$ が満たされた場合のみ，制約式 $c \ge 0$ を有効とする `when` $(e = 1)$ $c \ge 0$ といった表現は本来 ILP では記述できないが，SILP では以下のようなトリックにより，この「コーディング」を行っている．

$$c \ge M(e-1)$$

ここに M は非常に大きな正の定数であり，この式自体は線形制約式である．ここでもし $e = 1$ であれば，$c \ge 0$ であるから，たしかに制約が有効となる．またもし $e = 0$ ならば $c \ge -M$ となる．よって c のとりうる範囲を見積もることができ，その最小値よりも $-M$ が小さいなら，事実上この制約は存在しない．C_{reg_6} の場合でいえば，それは最適化されていないプログラムのサイズと各命令の実行時間から見積もることが可能である．

この手法の拡張として，e がブール値でない場合を考える．まず値が負かどうかを ILP で動的に調べるコード，C で書けば以下の `negp`

```
#define negp(x,y)  y == (x<0 ? 1:0)
```

は次のように記述できる．

$$
\begin{aligned}
&0 \le y \le 1 \\
&x \ge -yM \\
&x < (1-y)M
\end{aligned}
$$

同様に以下の `zerop`

```
#define zerop(x,y)  y == (x==0)
```

は次のように記述できる．

$$
\begin{aligned}
&0 \le y \le 1 \\
&negp(x, t_1) \\
&negp(-x, t_2) \\
&y = 1 - t_1 - t_2
\end{aligned}
$$

このようなトリックは，基本的な枠組みに収まらないような，特殊な制約条件を記述するのに役立つ．

3.4 ILP の求解とコード生成

以上の方法で生成された ILP をソルバに与えて求解を行う．我々は商用の ILP ソルバである NUOPT[3] を使用した．ILP 変数で，$V_{\mathtt{r}}$ 以外は実数でもよく，実行時間や実行開始時点はの方が自然である．NUOPT では整数変数と実数変数を混在させることができる．

表 1 は Ultra Sparc 1 における求解時間である．各プログラムは付録（**図 3**）と同一の命令定義において，表左の式に差し換えて実行した結果である（2 番目の式が付録のもの）．表の「命令」は式を素朴にコンパ

```
register:
        G = {r0,r1,r2,r3,r4}
        D = {d0,d1}

instruction:
        load : (dev:D,x:G).dev=(),x=()                  dura=2
        add  : (dev:D,x:G,y:G,z:G).dev=(),x=(y,z)       dura=1
        sub  : (dev:D,x:G,y:G,z:G).dev=(),x=(y,z)       dura=1
        mul  : (dev:D,x:G,y:G,z:G).dev=(),x=(y,z)       dura=1
        div  : (dev:D,x:G,y:G,z:G).dev=(),x=(y,z)       dura=2
        ret  : (dev:D,x:G).dev=(),()=(x)                dura=1

program:
        ;; r = a*b + c/(d-e)
        load    dev a
        load    dev b
        mul     dev v1 a b
        load    dev c
        load    dev d
        load    dev e
        sub     dev v2 d e
        div     dev v3 c v2
        add     dev v4 v1 v3
        ret     dev v4

/* 以下出力結果 */

/* SSA 変換された 入力プログラム */
load    dev0 a
load    dev1 b
mul     dev2 v1 a b
load    dev3 c
load    dev4 d
load    dev5 e
sub     dev6 v2 d e
div     dev7 v3 c v2
add     dev8 v4 v1 v3
ret     dev9 v4

/* レジスタの割り当て */
dev7 : d1               e  : r4
dev1 : d1               v2 : r3
dev0 : d1               d  : r3
dev5 : d1               v3 : r2
dev9 : d0               c  : r2
dev8 : d0               v4 : r1
dev2 : d0               v1 : r1
dev6 : d0               b  : r1
dev3 : d0               a  : r0
dev4 : d0

/* 出力プログラム */
0: (load d0 r3)           (load d1 r4)
1: (load d1 r0)           (load d0 r2)
2: (load d1 r1)           (sub d0 r3 r3 r4)
3: (div d1 r2 r2 r3)
4: (mul d0 r1 r0 r1)
5: (add d0 r1 r1 r2)
6: (ret d0 r1)
```

図 3 コード生成例
Fig. 3 Example of code generation.

表 1 実行時間
Table 1 Execution time.

	命令	変数	制約	時間（秒）
a*b+c/d	8	15	175	1.7
a*b+c/(d-e)	10	19	259	6.1
a*(b+c)+d/(e-f)	12	23	359	241.5

イルした結果の命令数,「変数」は SSA 変換後の仮想レジスタ数,「制約」は生成された ILP 制約式の数，時間は求解時間である．命令や変数のわずかな増加により，実行時間は著しく増大している．本手法の適用範囲を広げるためにも，処理速度の向上が必要である．

4. おわりに

DNA 実験ロボット ANP-96 のコード生成系に向けて，従来のコンパイラ技術の枠組みを応用するための抽象的なアセンブラ言語を定義し，そのコード生成系を実装した．コード生成系の実装においては，レジスタアロケーションと命令スケジューリングを同時に最適化するために，ILP に基づく手法を用いた．

複数の最適化問題を ILP へ変換して同時に解くことにより，最適化コード生成を行うという既存研究として，アロケーションとスピル処理を同時に解く手法[10]，アロケーションとスケジューリングを同時に解く手法 OASIC[9]，SILP[11),17)]，命令選択，アロケーション，スケジューリングを同時に解く手法[15] 等がある．扱える問題の範囲が広ければ，定式化も複雑になり，処理時間も増大する．

我々のコード生成系においては，命令選択やスピル処理は無関係であり，定式化の簡潔さから SILP をベースとして実装を行ったが，その定式化および実装と比較して以下の点が異なっている．

(1) 抽象化されたアセンブラ言語を対象とする．
(2) アセンブラ命令は複数の同時代入からなる．
(3) すべてのリソースは仮想レジスタとして明示的に表現される．
(4) レジスタ集合間に共通のレジスタがあってもよい．
(5) 仮想レジスタ単位でのレジスタ割当て．
(6) SSA 変換の利用．

既存の定式化と実装は通常の計算機を対象としたものであり，命令体系は RTL[13] に類似の固定されたセットを対象としている．一方我々の定式化は，アロケーションとスケジューリングが可能な抽象度の言語を対象としているため，応用範囲は広いといえる．また，既存の定式化では演算装置やメモリ等は，通常の計算機を対象としているので，命令中に明示はされない．よってそれらのリソースの割当ては，レジスタ割当て同様の処理であるにもかかわらず，個別に定式化されている．またその定式化において，命令が定義（代入）する引数レジスタはたかだか 1 つという前提があり，レジスタアロケーションはレジスタを定義している命令に対してなされている．我々の場合，ANP-96 の例からも明らかなように，命令が複数の同時代入を表現できるというのは本質的である．また，このことからすべてのリソースをレジスタとして命令中に明示的に表現し，レジスタ割当ては命令単位ではなく仮想レジスタ単位で行う．これを ILP で表現するには，仮想レジスタを定義している命令はただ 1 つという仮定が必要になるが，それは SSA 変換により可能である．また，SSA 変換をすることにより，従来手法では個別に対応していた逆依存や出力依存といった擬似的なデータ依存を考慮しなくてもよい．

一方で，我々の実装を既存のものと比較した場合，対象言語が抽象的であるぶん，記述できる内容が少ない．特に命令の実行時間は定数が記述できるのみである．本来，命令の実行時間はその命令の前後や命令の仮想レジスタに割り当てられたレジスタに依存する．実装の最後に述べた ILP による条件式の表現等を利用して対処できるものもあるが，枠組みとしてどのように，どこまで定式化すべきかは将来の課題である．

参考文献

1) プレシジョン・システム・サイエンス株式会社：ANP-96 マニュアル．http://www.pss.co.jp
2) 萩谷昌己，横森　貴：DNA コンピュータ，培風館 (2001).
3) 株式会社数理システム：NUOPT マニュアル．
4) 陶山　明：分子コンピュータの実験，数理科学，No.445（7月号），pp.27–31, サイエンス社 (2000).
5) Adleman, L.: Molecular Computation of Solutions to Combinatorial Problems, *Science,* Vol.266, pp.1021–1024 (1994).
6) Briggs, P., et al.: Improvements to graph coloring register allocatin, *TOPLAS*, Vol.16, No.3, pp.428–455 (1994).
7) Cytron, R., et al.: Efficiently Computing Static Single Assignment From and the Control Dependence Graph, *TOPLAS*, Vol.13, No.4, pp.451–490 (1991).
8) Ertl, M.A. and Krall, A.: Optimal Instruction Scheduling Using Constraint Logic Programming, *PLILP 91*, LNCS 528 (1991).
9) Gebotys, C.H. and Elmasry, M.I.: Global Optimization Approach for Architectual Synthesis, *IEEE Trans. Computer-Aided Design of*

Integrated Circuits and Systems, Vol.CAD-12, No.9, pp.1266–1278 (1993).
10) Goodwin, D.W. and Wilken, K.D.: Optimal and Near-optimal Global Register Allocation Using 0-1 Integer Programming, *Software Practice and Experience*, Vol.26, No.8, pp.929–965 (1997).
11) Kästner, D.: Retargetable Code Optimization by Integer Linear Programming, Ph.D. Thesis, Saarland University (2000).
12) Kästner, D.: PROPAN: A Retargetable System for Postpass Optimizations and Analyses, *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems* (2000).
13) Stallman, R.M.: *Using and Porting GNU CC*, Free Software Foundation (1999).
14) Suyama, A., Nishida, N., Kurata, K. and Omagari, K.: Gene Expression Analysis by DNA Computing, *Currents in Computational Molecular Biology*, pp.12–13 (2000).
15) Wilson, T., et al.: An Integrated Approach to Retargetable Code Generation, *7th Int. Symp. on High-Level Synthesis* (*HLSS*) (1994).
16) Yoshida, H. and Suyama, A.: Solutions to 3-SAT by Breadth First Search, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol.54, pp.9–22 (2000).
17) Zhang, L.: SILP. Scheduling and Allocating with Integer Linear Programming, Ph.D. Thesis, Technische Fakultät der Universität des Saarlandes (1996).

付　　録

実 行 例

以下はコード生成系の実行例である（図 3）．プログラムは分かりやすさのため，簡単な算術式に対する RISC 風のアセンブラ命令を例としているが，処理系は宣言で与えられたもの以外，各命令についての知識はない．命令実行時間は `load` と `div` が 2 単位，他は 1 単位である．

レジスタ集合 `G` は 4 つの汎用レジスタ，`D` は 2 つの処理装置を表す．各命令の引数は，その命令が実行される処理装置を表す仮想レジスタ `dev` および，演算を行う汎用レジスタからなる．

(平成 15 年 12 月 20 日受付)
(平成 16 年 2 月 24 日採録)

阿部　正佳

昭和 59 年東京理科大学理工学部数学科卒業．ソフトウェア会社勤務を経て，平成 14 年東京大学大学院理学系研究科情報科学専攻修士課程修了．言語処理系に興味を持つ．

萩谷　昌己（正会員）

昭和 57 年東京大学大学院理学系研究科情報科学専攻修士課程修了．京都大学数理解析研究所を経て，現在，東京大学大学院情報理工学系研究科教授（コンピュータ科学専攻）．計算システムをモデル化し，特に演繹的な方法を用いて，その性質を計算機上で検証することに興味を持っている．最近では，電子計算機からなる計算システム以外にも，生物系や分子系も研究の対象としている．特に，分子コンピューティングの研究を行っている．