計算機アーキテクチャ　63－17
（1986・11・21）

要求駆動型計算の最適化アルゴリズムの評価

Evaluation of Optimization Algorithm for Parallel Demand-Driven Computation

小野　諭†　　鶴岡　行雄††　　高橋　直久†
Satoshi ONO　　Yukio TSURUOKA　　Naohisa TAKAHASHI
（†NTT電気通信研究所　　††電気通信大学）
（NTT Electrical Communications Laboratories, University of Electro-Communications）

あらまし　関数型言語の並列処理に適した要求駆動型計算の最適化アルゴリズムを、データフローモデルのシミュレータを用いて評価した。最適化は、必須引数の先評価とデマンド波及の効率化および不要演算の除去の三点に関して行った。最適化したプログラムは、データ駆動型計算に対しては結果に影響を与えない計算を行なわない点で、また、純粋な要求駆動型計算に対してはデマンド波及に起因する遅延をへらし計算の並列性を引き出す点で優れていることが確認された。たとえば、式 tarai(3,2,0) の計算の場合、演算ユニット無限個の並列処理の仮定のもとでは、最適化の結果、データ駆動・純粋要求駆動の場合に比べ、計算時間がそれぞれ２４％，４８％短縮されることがわかった。また、演算ユニットを１個とする逐次処理の場合には、計算時間がそれぞれ５６％，５０％短縮されることがわかった。

１．　まえがき

関数型言語は副作用を排除した宣言型言語の一種であり、記述の簡潔性、並列処理可能性、プログラム検証・変換の容易性など、種々の優れた特徴を有している。そして、関数型言語を高速・並列に実行するため、データフローマシンやリダクションマシンなど、言語指向型アーキテクチャの研究も活発である[1]。

データフローマシンは、データの到着をもって命令の実行をスケジュールする並列処理マシンであり、制限された関数型言語[2]と特に高い適合性がある。ただし、データフローマシンによるデータ駆動型計算は式の並列内側評価[3]に対応するため、関数のノンストリクト性[4]を正しく実現できないという欠点をもっている。他方、リダクションマシンで通常用いられる要求駆動型計算は、逐次関数のクラス[4]のノンストリクト性を正しく実現できるものの、関数適用における引数の並列評価可能性が損なわれる他、デマンド伝播のオーバヘッドが大きい、評価が中途まで進んだ関数適用が大量に生成される、などの欠点が存在する。

このような問題点を解決するため、ストリクトネス解析[5]に基づく要求駆動型計算の最適化の研究[6][7]が行なわれている。筆者らは先に、要求駆動型計算を並列処理で実行する場合に適した最適化アルゴリズムを提案した[8][9]。この方法では、必須引数の先評価とデマンド波及の効率化および不要演算の除去の三点に関して最適化を行っている。

本稿では、データフローモデルのシミュレータを使用し、上記最適化の効果を評価した結果について報告する。シミュレーションでは、動的色割り当てを行なう循環パイプライン型データフローマシン[10]を簡略化したモデルを使用する。そして、高度に再帰的な関数 tarai および不要引数を含む関数 easy の２種類について、それぞれ、データ駆動・純粋要求駆動・最適化要求駆動の３方法で評価し、並列処理における最適化の効果を明らかにする。

本稿の構成は、次のようになっている。２．では、議論の準備として、使用した最適化アルゴリズムについて概説する。３．では、シミュレーションで採用したデータフローモデルについて説明する。４．では、シミュレーション結果およびそれに対する考察について述べる。

２．　最適化アルゴリズムの概要

2.1　関数の部分クラスと計算法の安全性

以下、本稿で扱う関数は、すべて構造化演算子（たとえば LISPの cons 関数など）のないフラットなドメイン上で定義され、かつ履歴依存性を持たない単調関数[3]であると仮定する。

関数は、引数の必須性により、ストリクト関数とノンストリクト関数とに二分される。一般に、ある引数が未定義であると結果も必ず未定義になる時、その引

---

†† 現在、ＮＴＴ基礎研究所・情報通信基礎研究部・第二研究室にて学外実習中

数は必須であるという。ストリクト関数とは、すべての引数が必須な関数のことである。加算、最大値などはストリクト関数である。これに対し、条件(if_then_else)関数や逐次論理和(引数を左から計算し、真のものがあれば結果は真、すべてが偽であれば結果は偽)、並列論理和(引数のうちひとつでも真のものがあれば結果は真、すべてが偽であれば結果は偽)などはノンストリクト関数である。

ノンストリクト関数の一部を含む部分クラスとして逐次関数が存在する。これは、どの引数が必須であるかが、その引数の計算前に判定可能な関数である。ストリクト関数の他、条件関数、逐次論理和などは逐次関数であるが、並列論理和は逐次関数ではない。

ところで、関数型言語では、関数名をその定義本体により展開し、得られた式を簡約化することで計算が進行する。計算過程において必須な展開／簡約化のみを行う計算法は、安全であるという。安全でない計算法を用いると、本来停止すべき計算が停止しなくなる危険性がある。

データ駆動型評価(引数すべての評価が終了した時点で、外側の関数を評価する)は、ストリクト関数に対し安全である。これに対し、要求駆動型評価(関数を外側から評価し、必要に応じて引数を評価する)は、逐次関数に対しても安全である[4]。

2.2 要求駆動型計算の最適化手法

本稿で用いる最適化アルゴリズム[9]は、必須引数の先評価、デマンド波及の効率化、および、不要演算の除去の三点に関して最適化を行っている。以下、それらの手法についてデータフローグラフ表現に依存しない形で説明する。

(1) 必須引数の先評価

関数適用 f(e) において、引数となっている部分式 e が関数 f の必須引数である場合、式 e の計算は常に安全である。この性質を利用し、任意の式に対し、その式の計算に必須な部分式を内側からデータ駆動で計算することを、必須引数の先評価と呼ぶ。

たとえば、

```
f(x,y) = if y>0 then x+y else x-y fi ;
```

とした時、関数 f(x,y) の必須引数は {x, y} である。ここで、式 f(f(1+2,4), -5) の計算に必須引数の先評価を適用してみる。この場合、部分式 f(1+2,4) は一番外側の関数 f の必須引数であり、部分式 1+2 は内側の関数 f の必須引数となっている。したがって、まず部分式 1+2 が計算されて3となり、その結果を用いて部分式 f(3,4) の計算が行なわれ、値7が得られる。最後に f(7,-5) が計算され、最終結果 12 を得る。

必須引数の先評価を行なうと、引数の計算を要求するシグナルであるデマンドを、実引数となっている部分式に波及させる必要がなくなる。たとえば、ストリクト関数のみからなる式を計算する場合、データ駆動型計算と全く同一の計算手順となる。したがって、デマンドの波及による遅延なしに計算を開始でき、式計算における並列性を完全に引き出すことができる。

(2) デマンド波及の効率化

ノンストリクト関数の引数のうち、必須ではないが計算結果に無関係でもない引数(保留引数)については、外側の関数からのデマンド発生を待って実引数の計算を開始しなければならない。この場合、純粋な要求駆動型計算では、実引数を与える部分式のうち最外側の関数にデマンドを波及させる。これに対し、最適化要求駆動型計算では、デマンドが発生した時点で新たに必須になる式を解析し、それらがデータ駆動型で計算されるようにデマンドを伝搬させ、デマンド波及を効率化する。

たとえば、

```
f(x,y,z) = if x>0 then y else y+z fi ;
g(x)     = if x=0 then 0 else x+g(x-1) fi ;
```

とした時、関数 f(x,y,z) において、x, y は必須引数であり、z は保留引数である。また、関数 g(x) はストリクト関数である。ここで、式 f(-1, 2, g(3)) の計算を考えると、

```
    f( -1, 2, g(3) )
→   if -1>0  then 2 else 2+g(3) fi
→   if false then 2 else 2+g(3) fi
→   2+g(3)
```

となる。条件関数の else 部が選択された時点で、f の第三引数 g(3) が新たに必須になり、この部分式にデマンドが伝搬される。以下データ駆動で計算が進行し、2+g(3) → 2+6 → 8 となり、最終結果 8 が求まる。この計算例では、デマンド伝搬は1回のみであり、純粋要求駆動型計算に比較し、デマンド波及が大幅に削減されていることがわかる。

(3) 不要演算の除去

計算結果にけして影響を与えない引数や部分式をそれぞれ不要引数、不要部分式とよぶ。不要引数／部分式の存在は、プログラムの誤りや冗長性を表している。最適化によりこれらの不要演算を取り除くようにする。

たとえば、関数

```
f(x,y) = if x=0 then 1 else f(x-1,f(x,y)) fi ;
```

について考える。関数 f(x,y) において、x は必須引数であり、y は不要引数である。また、関数適用 f(x-1, f(x,y)) の第二引数である f(x,y) は不要部分式である。この関数をデータ駆動で計算すると、不要部分式 f(x,y) の計算のところで無限ループに陥ってしまい、

結果が求まらない。他方、要求駆動で計算すると、任意の非負整数xに対し1となる。

最適化により不要演算を除去すると、上記の関数は、

f'(x) = if x=0 then 1 else f'(x-1) fi ;

という一引数関数f'と等価な関数に変換される。関数f'(x)はストリクト関数であり、条件関数の部分以外は、データ駆動で計算することができるようになる。

2.3 最適化要求駆動型計算の実現法

関数適用の実行過程において、部分式が計算されるが、その結果のことを部分値と呼ぶ。本節では、要求駆動型計算を最適化する上で重要な、部分値とデマンドに関するふたつの概念を導入する。

・部分値総合必須値 (Partial Resulting-Value Total Requisite Value Set: TRVS)

ある部分値aを計算する場合、前もって必ず計算しておかなければならない引数や部分値の集合を、aの部分値総合必須値と呼び、$\mathcal{T}$(a)と表現する。ここで、$\mathcal{T}$(a)には、a自身も含めるものとする。

たとえば、前節(2)で取り上げた関数fを、部分値ごとに名前をつけて、以下のように書き直す。

```
f(x,y,z) = { a=(x>0) ;   b=y+z ;
             c=if a then y else b fi ;
             return c }
```

このとき、$\mathcal{T}$(c)={a,c,x,y}である。なぜなら、部分値cを計算するためには部分値aおよび引数xが必須であり、また、**then** 部と **else** 部のいずれにおいても引数yが必須だからである。これに対し、部分値bおよび引数zは **else** 部のみで必須であり、$\mathcal{T}$(c)には含まれない。

関数の最終結果の TRVS に含まれる引数は、必須引数であり、逆もまた正しい。たとえば上記関数fの必須引数はxおよびyである。

・デマンド生成必須値 (Demand-Generated Requisite Value Set: DRVS)

関数定義内にノンストリクト関数が存在する場合、その保留引数を与える部分値や引数の評価を、デマンドが発生するまで保留しなければならない場合がある。これは、その部分値／引数がデマンドにより新たに必須となるからである。ある関数適用において、保留引数が必須になったことを伝達するデマンドアーク da に対し、そこにデマンドが発生することにより新たに必須となる部分値／引数の集合を、 da のデマンド生成必須値と呼び、$\mathcal{R}$(da)と表現する。

たとえば、上記関数fにおいて、部分値cを与える条件関数の **then**部と **else** 部のデマンドをそれぞれ dc_1, dc_2 とすると、$\mathcal{R}(dc_1)=\{\ \}$, $\mathcal{R}(dc_2)=\{b,z\}$ となる。すなわち、**then** 部が選択された場合に新たに必須になる値は存在しないが、 **else** 部が選択された場合には、新たに部分値bおよび引数zが必須になる。

上記概念を用いて、前節に述べた最適化を以下のようにして実現する。

i) 必須引数は先評価して値渡しする。すなわち、部分値rを与える関数適用において、部分値rが必須となった場合、その必須引数を先評価してから関数を展開する。このとき、展開した関数内でのデマンド波及を効率化するため、必須引数を先評価するだけでなく、$\mathcal{T}$(r)に属する部分値全体をデータ駆動で計算する。

たとえば、上記関数fの場合、関数適用 f(x,y,z) の結果が必須になった時点で、必須引数x,yを先評価する。また、関数の展開後、最終結果である部分値cの TRVS に含まれる部分値aとcをデータ駆動で計算する。

ii) 保留引数は必要になった時点で評価し値渡しする。このため、保留引数aが必須と判明した時点でデマンド da をデマンドアークを経由して関数呼び出し側に伝達し、保留引数aの評価を開始する。この場合も、デマンド波及の効率化のため、保留引数aへのデマンド発生を契機として、デマンド生成必須値 $\mathcal{R}$(da) に属する部分値をデータ駆動型評価する。

たとえば、上記関数 f(x,y,z)において、**else** 部のデマンドdc_2 が発生した時点で保留引数zおよび部分値bをデータ駆動で計算する。

iii) 関数定義において、その最終結果である部分値の TRVS にも含まれず、また関数定義内に存在するいずれのデマンドの DRVS にも含まれない引数／部分値は、不要引数／部分値である。これらに対応する演算は除去する。

たとえば、前節(3)で取り上げた関数fを、部分値ごとに名前をつけて、以下のように書き直す。

```
f(x,y) = { a=(x=0) ;     b=x-1 ;
           c=f(x,y) ;    e=f(b,c)   ;
           i=if a then 1 else e fi ;
           return i }
```

このとき、最終結果である部分値iに対し、$\mathcal{T}$(i)={a,i,x}である。また、部分値iを与える条件関数の **then** 部と**else** 部のデマンドをそれぞれ di_1, di_2 とすると、$\mathcal{R}(di_1)=\{\ \}$, $\mathcal{R}(di_2)=\{b,e\}$ となる。したがって、これらのいずれにも含まれないcとyは、それぞれ不要部分値，不要引数である。

部分値の TRVS やデマンドの DRVS を求めるには、関数間にわたる広域データフロー解析を必要とする。

最適化アルゴリズムの詳細については、文献(9)を、また、その基礎となる広域データフロー解析アルゴリズムについては文献(11,12)を参照されたい。

3．　データフローモデルと評価システム

3.1　データフローモデルと関数適用

本稿では、関数型言語の実行モデルとして、動的色割り当てを行なうデータフローモデルを基本とし、いくつかの拡張を行なっている。以下、それらの拡張点と重要な概念について説明する。

（1）データフローモデルの拡張

デマンドによる計算の制御を可能にするため、部分値を示す値トークンを運ぶ値アークに加えて、デマンドトークンを運ぶデマンドアークを設ける。本稿では、値アークを実線矢印で、デマンドアークを破線矢印で示す。なお、ノードに向かうアークを、そのノードの入力アーク、逆方向のアークを出力アークと呼ぶ。

ノードには、演算ノードと補助ノードがある。演算ノードは、さらに基本ノード、定義ノード、マクロノード、の３種類に分類される。基本ノードは、四則演算のようにマシンで直接実行されるシステム組み込みの関数を表す。定義ノードは、関数定義された関数を表すノードであり、関数適用時に展開される。マクロノードは、コンパイル時に複数のノードに展開されるノードであり、条件関数ノードがこれに相当する。演算ノードは複数の値入力アークとひとつの値出力アークを持ち、また、複数のデマンド出力アークやひとつのデマンド入力アークを持つことがある。

補助ノードには、図１に示すように、値を分配する分配ノード、複数の入力をマージするマージノードのほか、ＤＭＶＤ(Demand Merge Value Distribution)ノード[9]がある。このノードは、複数のノードからひとつのノードにデマンドが出される場合において、二番目以降のデマンドを吸収すると共に、要求を出したノードに計算結果を分配するために使用される。

ＤＭＶＤノードの機能を説明する。このノードは、それぞれのデマンド入力アークから、０または１個のデマンドトークンを受け取る。最初のデマンドトークンが到着した時点で、デマンド出力アークにデマンドトークンが送出される。二番目以降のデマンドトークンは、ノード内で吸収される。値入力アークに値トークンが到着すると、そのコピーがデマンドを出した各アーク対の値出力アークに送出される。デマンド入力アークにトークンが到着した時点で既に値トークンが到着していた場合は、そのコピーが直ちに対応する値出力アークに送出される。

（2）関数適用法

関数型言語の実行の基本となる関数適用の方法は、データ駆動、純粋要求駆動、最適化要求駆動の各計算法により異なっている。図２に各計算法それぞれの関数適用法をまとめた。ここで、関数 ｆ(ｒ,ｓ,ｉ) において、ｒは必須引数、ｓは保留引数、ｉは不要引数とし、結果の部分値をｖとする。

データ駆動型計算では、引数は常に関数適用前に計算される。引数の到着後、関数が展開される。

純粋要求駆動型計算では、引数を与える値入力アークそれぞれに対し、デマンド出力アークが対応付けられる。関数は、デマンド入力アークへのデマンド発生を契機に展開される。実引数の計算要求はデマンド出力アークにより伝えられ、引数の計算結果が対応する値入力アークにより渡され、結果が求まる。

最適化要求駆動型計算では、引数の種別により取扱いが異なる。必須引数ｒは先評価されるので、値入力アークのみが付けられる。一方、保留引数ｓに対しては、純粋要求駆動の場合と同様に、デマンド出力アーク ds が対応付けられる。不要引数ｉに対しては、アークは生成されない。最適化要求駆動の場合、関数展開法としてデマンド展開と自己展開の２種類がある。デマンド展開は、純粋要求駆動の場合と同様にデマンド入力アークにより関数展開を制御する方法であり、関数展開を引数の到着とは独立に制御しなければならない場合に使用する。自己展開は、必須引数の到着をもっ

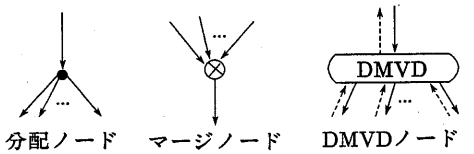


図1 補助ノード

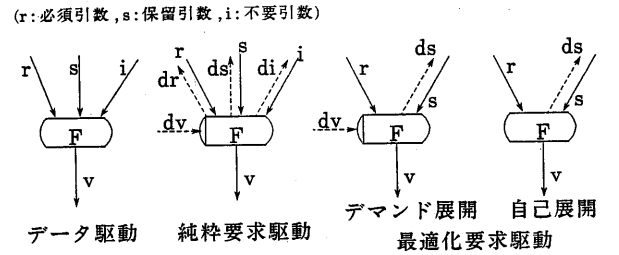


図2 関数適用法

て関数展開を開始する方法で、最適化アルゴリズムによりデータ駆動で計算可能と判定された部分に使用される。

3.2 最適化要求駆動型計算の実行法

前節で述べた最適化要求駆動型計算をデータフローマシンで実行するには、次のようにする。まず、プログラムより、データフローグラフを生成する。このグラフに対し、広域データフロー解析に基づくグラフ変換を施す。そして、変換後のグラフに対するデータ駆動による解釈が、もとのプログラムの最適化要求駆動型計算と等価になるようにする。本節では、グラフの変換、およびその変換後のプログラムの動作を簡単な例を用いて説明する。

次の関数 p(x,y,z) を考える。

```
p(x,y,z) =
  {s1=f1(g1(x,y)); t1=f2(f3(g2(x,y)));
   u1=f4(g3(y,z));
   a=if s1 then t1 else u1 fi ;
   return a}
```

ここで、f_1〜f_4 および g_1〜g_3 は、それぞれ一引数および二引数のストリクト関数とする。関数p(x,y,z)において、引数 x,y は必須引数であり、引数zは保留引数である。

関数pの変換前のデータフローグラフを図3に、また、変換後のグラフを図4に示す。図4において、関数f_1〜f_4, g_1〜g_3 はストリクト関数だから、すべての引数に値渡しが使用される。条件関数については、**bool** 部は値渡しされ、**then** 部, **else** 部には、それぞれ、デマンドアーク dt_1 および du_1 が張られる。また、この例において、$\mathcal{A}(a)=\{a, s_1, s_2, x, y\}$ であり、$\mathcal{R}(dt_1)=\{t_1, t_2, t_3\}$、$\mathcal{R}(du_1)=\{u_1, u_2, z\}$ である。この関数には、不要引数／部分値は存在しない。

図3のプログラムの動作を順に説明する。関数pが展開され、必須引数x,yが渡されると、部分値s_2 続いてs_1 がデータ駆動で計算され、それにより条件関数が展開される。もし、**then** 部である dt_1 にデマンドが発生すると、関数g_2 がデマンド展開され、部分値t_3 が生成される。このt_3 により部分値t_2 引き続いてt_1 がデータ駆動で生成され、t_1 が関数に渡されて最終結果aを得る。もし、条件関数の **else** 部である du_1にデマンドが発生したとすると、そのデマンドは関数pの引数zのデマンドとして関数pを展開した側に伝達される。引数zの計算が終了してそのトークンが関数p に到着すると、部分値u_2 , u_1 が同様にして生成され、u_1 が条件関数に渡されて最終結果aを得る。

3.3 評価システム

（1）評価システムの構成

最適化の効果を明らかにするため、評価システムを試作した。その構成の概略を図5に示す。システムは、コンパイラ部、最適化部、シミュレータ部およびシステム制御部からなっている。コンパイラ部は LISP のS式表現で記述された関数型言語プログラムを入力とし、データフローグラフを出力する。最適化部はプログラムを広域解析し、データフローグラフの変換を行なう。シミュレータ部はデータフローグラフを実行し、各種のデータを結果として出力する。システム制御部はシステム全体の動作を制御している。

最適化部は、関数間解析部，関数内解析部，グラフ変換部からなっている。関数間解析部はプログラム中のすべての関数を同時に解析し、各関数の引数依存属

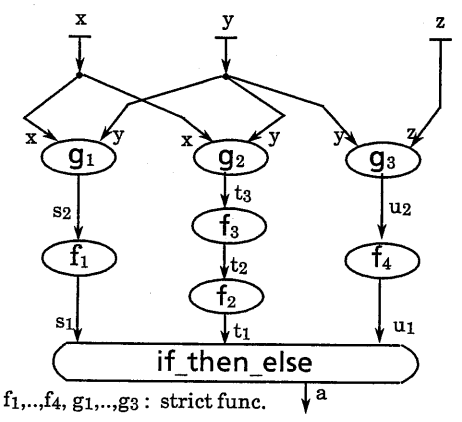


図3 関数 p(x,y,z) (変換前)

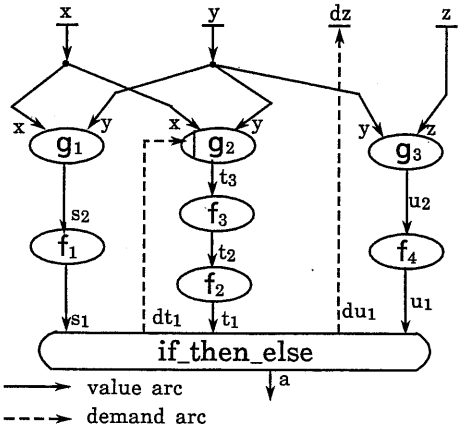


図4 関数 p(x,y,z) (変換後)

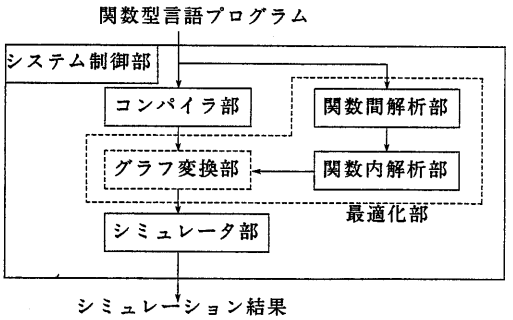
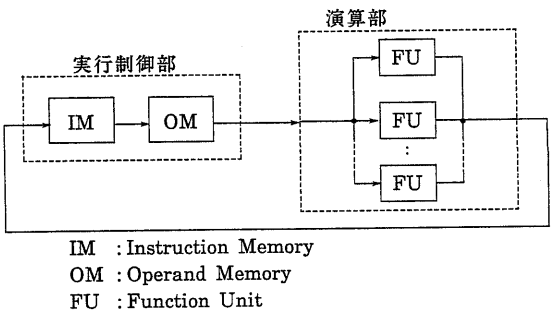


図5 評価システムの概略



IM : Instruction Memory
OM : Operand Memory
FU : Function Unit

図6 シミュレーションモデル

性集合(ＰＤＰＳ)[12]を求める。関数内解析は、それぞれの関数に閉じた解析を行い、参照する関数の PDPS をもとにして、節 2.3 で説明した TRVS, DRVS など、最適化に必要な情報を計算する。グラフ変換部は、最適化要求駆動型計算を実現できるように、データフローグラフを変換する。

以上の評価システムを、記号処理言語 TAO[13]を用いて記述した。規模は、1.7k ステップである。ただし、現在の評価システムはグラフ変換部が未完成であるため、次章に述べる例のグラフ変換は、最適化部の解析結果をもとにしてハンドコーディングにより行った。

(2) シミュレーションモデル

シミュレーションの基本となるデータフローマシンのアーキテクチャとしては、循環パイプライン型データフローマシン[10]を想定した。その構成を図6に示す。このマシンは、命令メモリ(ＩＭ), オペランドメモリ(ＯＭ), 演算ユニット(ＦＵ) から構成されている。ＩＭはデータフローグラフを格納する。また、ＯＭは値トークンやデマンドトークンを格納するとともに、実行可能な命令を選択する。ＦＵは、ＯＭで選択された命令を実行する。これらの要素をパイプライン状に接続し、データパケットを循環させることによりプログラムを実行する。

シミュレーションにおいては、簡単のため、以下のことを仮定した。

＊基本ノードは任意個の入力をもつ１出力のストリクト関数とする。
＊ＦＵ数は無限個とする。
＊分配ノードの実行時間は０クロックとし、それ以外の補助ノードおよびすべての基本ノードの実行時間は１クロックとする。
＊定義ノードに関する以下の処理は、すべて１クロックとする。

・関数を展開すること
・展開された関数へ引数を渡すこと
・展開された関数へデマンドや値を受け取るアドレスを渡すこと
・展開された関数から呼び出し側の指定されたアドレスへデマンドや値を渡すこと。

＊ＩＭ, ＯＭの実行時間およびパイプラインの遅延時間は無視する。

また、関数適用では、以下のようにした。

＊データ駆動型計算では、第一引数の値の到着をもって関数を展開する。ただし、条件関数についてのみは、デマンドにより実引数を計算する要求駆動を用いる。
＊最適化要求駆動型計算の自己展開では、最も左側の必須引数の到着をもって関数を展開する。
＊純粋要求駆動型計算および最適化要求駆動型計算のデマンド展開では、デマンド入力アークへのデマンド到着をもって、関数を展開する。
＊展開された関数において、結果の値が求まった場合は、計算中のものがあっても、直ちに呼び出し側に値を返す。
＊計算終了は、すべての起動された関数適用が終了した時点とする。

４． シミュレーション結果とその考察

4.1 サンプルプログラム

最適化の評価のための例題として、次のふたつの再帰関数 tarai と easy を用いた。

```
tarai(x,y,z) =
     if x>y then tarai(tarai(x-1,y,z)
                       tarai(y-1,z,x)
                       tarai(z-1,x,y))
            else y fi ;
```

```
easy(x,y) =
    if x=0 then 0 else easy(x-1,easy(y,x))  fi ;
```

関数 tarai(x,y,z) において、xとyは必須引数であり、zは保留引数である。この関数は高度に再帰的であり、かつ保留引数を含むので、データ駆動における不要な関数適用の影響や、最適化要求駆動における必須引数の並列計算の効果を明らかにするのに適している。

関数 easy は、データ駆動による計算が安全でなく、また並列性を持たない場合の例として用いた。この関数をデータ駆動で計算すると次に示す関数 $easy_1$ と等価になり、また、要求駆動（不動点計算規則）で計算すると関数 $easy_2$ と等価になる。

```
easy1(x,y) =
    if ( (x=0) ∨ (x>0 ∧ y=0) ) then 0
                               else ω fi ;
easy2(x,y) =  if x≥0  then 0 else ω fi ;
```

ここで、記号ωは、計算結果が未定義（無限ループ）になることを表す。したがって、式 easy(2,2) は、要求駆動で計算すると0になるが、データ駆動で計算すると未定義となる。直観的にはあまり明らかではないが、関数 easy(x,y) において、xは必須引数、yは不要引数である。したがって、不要引数yを除去すると、この関数はストリクト関数となり、条件関数部分を除き、データ駆動で評価しても安全になる。この関数は、不要演算の除去の効果、および、必須引数の先評価によるデマンド波及遅延の除去の効果を明らかにするのに適している。

4.2 グラフ変換結果とその考察

前節で述べた関数 tarai および easy のデータフローグラフをそれぞれ、図7および図8に示す。これらの図において、(a)はデータ駆動、(b)は最適化要求駆動で計算するグラフである。純粋要求駆動で計算するデータフローグラフは、データ駆動のグラフにおいて、値アークと反対方向にデマンドアークを張ったものとほぼ等しいので省略した。

関数 tarai(x,y,z) の場合、データ駆動と最適化要求駆動の違いは、図7に示されるように、保留引数である第三引数zに現れている。データ駆動では無条件にこの引数が計算されているのに対し、最適化要求駆動では、引数計算がデマンドにより制御されている。

関数 easy(x,y) の場合、両者の違いは不要引数である第二引数yに現れている。図8から明らかなように、最適化要求駆動では、第二引数に関連した不要演算が、グラフからすべて除去されている。

表1に純粋要求駆動を含む各々のデータフローグラフの静的なコード量を示す。不要演算を含まない関数 tarai の場合、ノード数およびデマンドアーク数は、データ駆動の場合が最も少ない。これは、要求駆動におけるデマンド制御のオーバヘッドのためである。たとえば、データ駆動ではゲートノード1個が使われているだけであるが、純粋要求駆動ではＤＭＶＤノード3個、また最適化要求駆動ではゲートノード4個が使われている。要求駆動相互の比較では、最適化要求駆動の方が純粋要求駆動に比べ、ＤＭＶＤノードが単純なゲートノードに置換されている点、デマンドアークが 20 から8へ大幅に削減されている点で優れている。

関数 easy の場合、不要演算を除去したため、ノード数，アーク数とも最適化要求駆動が最も少なくなっている。

4.3 シミュレーション結果

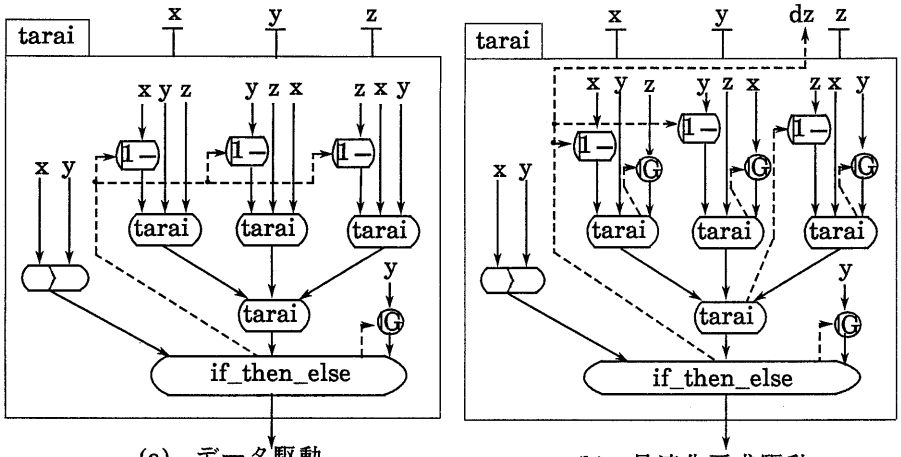
前節に示したデータフローグラフを用い、評価システムにより、式 tarai(3,2,0) と easy(2,2) の計算をシミュレートした。その結果を表2および図9，10にまとめた。

表2に示した評価項目を下記に説明する。

並列計算時間，総命令実行数とは、それぞれ、演算ユニットを無限個，一個とした場合の計算終了までのクロック数である。また、命令を演算処理，デマンド制御，関数適用制御の3種類に分類して、それぞれの命令実行数をカウントした。演算処理命令とは、基本ノードや条件関数マクロノードなど、式の簡約化に対応する命令である。デマンド制御命令とは、ゲートノードやＤＭＶＤノードのように、関数展開を制御するデマンドを扱う命令である。また、関数適用制御命令とは、関数の展開や引数，デマンド，結果値などを受け渡すための命令である。上記3種類の命令の実行数の和は、総命令実行数に等しい。関数適用回数とは、定義ノードが展開された回数である。

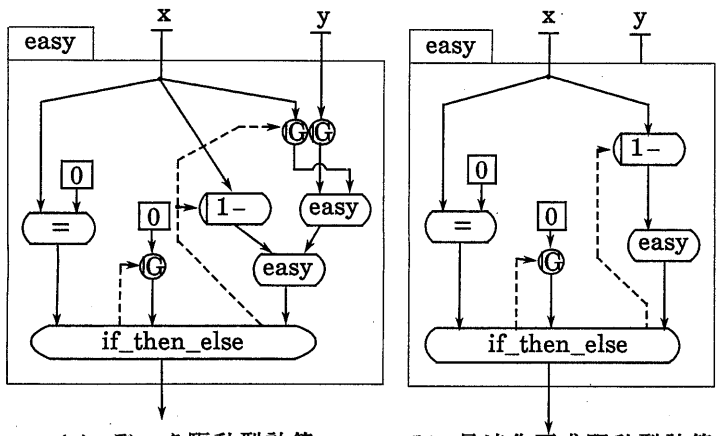
図9に、式 easy(2,2) の計算過程における瞬間並列度（並列に実行可能な命令数）の時間変化を示した。並列度が1クロックだけ凸形に高くなっている部分は、定義ノードが展開され、引数が渡されている部分である。節 4.1 で述べたように、式 easy(2,2) をデータ駆動で計算すると、無限ループに陥っている。要求駆動では、いずれの方法でも逐次的に計算が進み、3回の関数適用の後に停止している。

図10は、式 tarai(3,2,0) の計算過程における関数並列度（同時に展開されている定義ノード数）の時間変化を示したものである。この値は、データフローマシンにおいて主要な資源のひとつである同時に使用中の色の数に相当している。



(a) データ駆動　　(b) 最適化要求駆動

図7 関数 tarai(x,y,z)のデータフローグラフ



(a) データ駆動型計算　　(b) 最適化要求駆動型計算

図8 関数 easy(x,y)のデータフローグラフ

(a) 関数 tarai の場合

	データ駆動	純粋要求駆動	最適化要求駆動
ノード数	10	12	13
値アーク数	22	21	25
デマンドアーク数	4	20	8

(b) 関数 easy の場合

	データ駆動	純粋要求駆動	最適化要求駆動
ノード数	8	7	5
値アーク数	14	13	9
デマンドアーク数	3	9	2

表1 データフローグラフの静的コード量

4.4 シミュレーション結果への考察

（1）データ駆動型計算 対 要求駆動型計算

表2からわかるように、最適化および純粋要求駆動型計算は、関数適用回数と演算処理命令実行数が等しい。これは、両者とも結果に必須な計算のみを実行するからである。これに対し、データ駆動では結果に不要な計算も行なうため、不要引数を持たない tarai の場合ですら、関数適用回数，演算処理命令実行数とも2倍以上に増加している。この結果、最適化要求駆動はデータ駆動に比べ、並列実行時間が 46 から 35 に24% 減少し、総命令実行数も 56% 減少している。

データ駆動では、図10に示されるように関数並列度も一時的に高い値となり、システムに負荷となっていることがわかる。

（2）純粋要求駆動型計算 対
最適化要求駆動型計算

(i) デマンド波及遅延の除去効果

本質的に並列性を持たない関数 easy の例により、デマンド波及にかかる遅延を除去した効果を調べる。最適化要求駆動では値呼びが使われるため、表2に示されるように、デマンド制御命令実行数が大幅に減少している。この結果、図9でも明らかなように、短かい間隔で関数展開が起きている。これに対し、純粋要求駆動では、デマンド制御にかかる命令が多数必要なため、相対的に長いクロックを要している。この間隔は、並列処理によっても改善できない。最適化の結果、並列計算時間は 40% 以上減少している。

(ii) 引数並列計算の効果

並列性を持つ関数 tarai の例により、最適化による並列性の改善効果を考察する。図10に示されるように、最適化したものは、純粋要求駆動より早く関数並列度が立ち上がっている。これは、必須引数の先評価によりデータ駆動と同様に高い並列度で効率よく実行されているからである。これに対し、純粋要求駆動型計算ではデマンド伝播により全体的に計算が遅延している。この結果、最適化要求駆動は純粋要求駆動に比べ、並列実行時間、総命令実行数が、それぞれ 48%，56% も減少している。

データ駆動で実行すると、引数の計算が終了した段階で直ちに色資源を解放できる。し

(a) 式 tarai(3,2,0) の実行結果

	データ駆動	純粋要求駆動	最適化要求駆動
並列計算時間	46	68	35
総命令実行数	179	158	78
演算処理命令実行数	64	26	26
デマンド制御命令実行数	13	51	6
関数適用制御命令実行数	102	81	46
関数適用回数	17	7	7

(b) 式 easy(2,2) の実行結果

	データ駆動	純粋要求駆動	最適化要求駆動
並列計算時間	−注)	39	22
総命令実行数	−	48	26
演算処理命令実行数	−	12	12
デマンド制御命令実行数	−	12	1
関数適用制御命令実行数	−	24	13
関数適用回数	−	3	3

注) "−"は計算が停止しないことを示す。

表2 シミュレーション結果

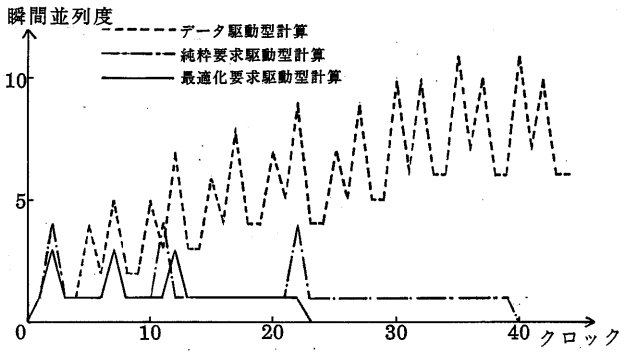


図9 式 easy(2,2)の実行における瞬間並列度

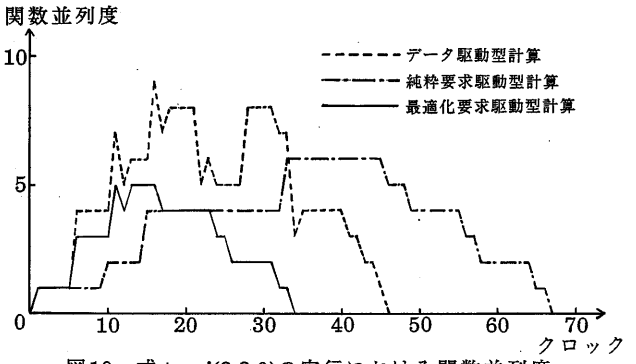


図10 式 tarai(3,2,0)の実行における関数並列度

たがって、デマンド波及の最適化は、計算時間を短縮するのみならず、色資源の節約に寄与していることも図10 からわかる。

５．　むすび

関数型言語の並列処理に適した要求駆動型計算の最適化アルゴリズムを、データフローモデルのシミュレータを用いて評価した。最適化は、必須引数の先評価とデマンド波及の効率化および不要演算の除去の三点に関して行った。

最適化したプログラムは、データ駆動型計算に対しては結果に影響を与えない計算を行なわない点で、また、純粋な要求駆動型計算に対してはデマンド波及に起因する遅延をへらし計算の並列性を引き出す点で優れていることが確認された。

たとえば、式 tarai(3,2,0) の計算の場合、演算ユニット無限個の並列処理の仮定のもとでは、最適化の結果、データ駆動・純粋要求駆動の場合に比べ、計算時間がそれぞれ２４％，４８％短縮されることがわかった。また、演算ユニットを１個とする逐次処理の場合には、計算時間がそれぞれ５６％，５０％短縮されることがわかった。

今後は、リスト処理を含むより広範なクラスの関数への最適化アルゴリズムの拡張、幽霊トークンの除去、ガーベッジ・コレクションの効率化などを検討していきたい。

謝辞　日頃よりご助言いただく電気通信大学の石井正博教授，伊藤秀一助教授並びにＮＴＴ基礎研究所・情報通信基礎研究部の雨宮真人第一研究室長，日比野靖第二研究室長に感謝します。また、言語処理系やシミュレータなどにつきご討論いただいた武末勝主幹研究員，長谷川隆三主幹研究員をはじめ第二研究室の諸氏に感謝します。

文　献

(1) P.C.Treleaven, D.R.Brownbridge, and R.P.Hopkins, "Data-Driven and Demand-Driven Computer Architecture", ACM Computing Surveys, 14, 1, pp.93-142 (1982)

(2) 雨宮，長谷川，小野："データフロー計算機用高級言語 Valid"，NTT 電気通信研究所研究実用化報告,33, 6, pp.1167-1181 (1984)

(3) Z.Manna: "Mathematical Theory of Computation", McGraw-Hill (1974)

(4) J.Vuillemin: "Correct and Optimal Implementations of Recursion in a Simple Programming Language", J. Computer and System Sciences, 9, pp.332-354 (1974)

(5) C.Clack and S.L.P.Jones, "Strictness analysis - a practical approach", in J.P.Jouannaud (ed.) Lecture Notes in Computer Science, 201, Functional Programming Languages and Computer Architecture, Springer-Verlag, pp.35-49 (1985)

(6) 関，井上，谷口，嵩："関数型言語ＡＳＬ／Ｆのコンパイル時における最適化"，信学論(D)，J67-D，10，pp.1115-1122 (昭59-10)

(7) J.Tanaka: "Optimized Concurrent Execution of an Applicative Language", Ph.D thesis, Univ. of Utah (1984).

(8) 小野，高橋："データフローマシンにおける関数型言語の最適化された要求駆動型評価",データフローワークショップ１９８６，pp.39-48，信学会データフローアーキテクチャと並列処理時限研究専門委員会 (1986)

(9) 小野，高橋："関数型言語における要求駆動型評価の並列処理向き最適化"，信学論(D)，J70-D，2 (1987) (掲載予定)

(10) M.Amamiya, M.Takesue, R.Hasegawa and H.Mikami, "Implementation and Evaluation of A List-Processing-Oriented Data Flow Machine", Proc. 13th Ann. Int. Symp. Computer Architecture, pp. 10-19 IEEE (1986)

(11) S.Ono, N.Takahashi and M.Amamiya, "Non-Strict Partial Computation with a DataflowMachine", 京都大学数理解析研究所講究録，547 pp.196-229 (1985)

(12) 小野，高橋："再帰関数系における依存属性集合の計算法"，信学論(D)，J69-D，5，pp.714-723 (1986)

(13) I.Takeuchi, H.Okuno and N.Ohsato, "A List Processing Language TAO with Multiple Programming Paradigms", New Generation Computing, Springer-Verlag 4, 4 (1986) (掲載予定)