プログラミング 14−13
(1997. 8. 21)

非ストリクトデータフロープログラム実行における スタックフレームの利用

日下部茂[†] 森本徹夫[†,☆] 雨宮真人[†]

データフローモデルに基づく言語は明示的な並列実行制御無しに容易に並列処理記述を可能とする．特に非ストリクトなデータフロー言語は，柔軟な記述と潜在的な高並列性を持つ．本稿では，特別の細粒度実行の支援機構を持っていないため，効率良い実装が容易でなかった汎用の計算機上への効率の良い実装法について議論する．まず，必要な非ストリクト性を保証しかつ関数呼出を効率化するためのコード生成について論じる．次に非ストリクトな方式とストリクトな方式をともにサポートする実行時システムについて述べる．予備評価の結果，汎用計算機においても，実用的な効率で非ストリクトなデータフロー言語プログラムを実行することが出来ることが確認できた．

Using Stack Frame for Non-Strict Dataflow Program

SHIGERU KUSAKABE,[†] TETSUO MORIMOTO [†,☆]
and MAKOTO AMAMIYA[†]

Dataflow-based non-strict functional programming languages have attractive features. In order to achieve high efficiency on stock machines, we want to use stack frame instead of heap frame for fine grain function instances, which may require dynamic scheduling As a static approach, we introduce a merging policy to a thread partitioning algorithm in order to find functions with a potentially strict call interface in non-strict dataflow language programs. To complement the static analysis, we provide a hybrid runtime mechanism which can dynamically change a suspended stack frame into a heap frame. The results of the preliminary performance evaluation indicate that we can reduce superfluous heap frame management and achieve practical performance even on stock machines by the static thread merging and the hybrid runtime system.

1. はじめに

我々は，並列実行のモデルとして，本質的に並列性を内在し，実行制御はデータ依存則にそって自動的になされるデータフローモデルに着目している．データフローモデルにもとづく言語は，値による自動的同期，明示的な並列実行指定が不要，など並列処理記述における魅力的な特徴を持っている．特に，ストリクトな言語ではなく非ストリクトな言語の方が，柔軟性の高い記述と高並列実行の両方を実現可能と考え，非ストリクトなデータフロー言語の研究を行なっている．

非ストリクトなデータフロー言語のプログラムをeagerに評価する方式(lenient方式)により，細粒度の高並列性を抽出でき，データフロー計算機では高い効率でプログラムを実行することが可能である[5)6)10)]．しかしながら，特別の細粒度実行支援機構を持たない計算機でlenient方式にもとづく実行方式を単純にとりいれても，並列実行制御や同期処理をソフトウェア的に実現するオーバヘッドにより，実行時の効率は大きく低下してしまう．

関数型言語をベースにしたデータフロー言語の場合，実行時に細粒度の関数インスタンスが非常に多く生成されるため，関数実行方式の効率向上は，プログラム全体の効率に大きな効果が期待できる．本稿では非ストリクトなデータフロー言語を既存の汎用計算機に実装する際に，プログラム中のストリクト性，特に関数レベルのストリクト性を活用し効率の向上をはかる手法について論じる．非ストリクト言語の実際のプログラムにおいては，特に構造体の非ストリクト性が有用で，プログラム記述においては構造体の非ストリクト性を利用したものが多いとされている[7)]．これは，実際のプログラムでは，関数に関してはあまり非ストリクト性が活用されておらず，多くの部分ではストリクトな方式でも実行可能と予想できる．

汎用の計算機で非ストリクト関数を実行する場合，関数インスタンスレベルでの動的スケジューリングを行なうために，ヒープメモリ上で実行インスタンスを管理するが，スタックを用いた関数の実行とくらべ効率はかなり低下する．言語が非ストリクト性を持つとしても，何らかの解析によってストリクトな関数を抽出でき，なおかつ，その関数が実行中サスペンドしなければ，ストリクトな関数をスタックを用いて実行することにより，プ

† 九州大学 知能システム学専攻
Dpt. Intelligent Systems, Kyushu University
☆ 現在，日本Victor
Presently with VICTOR CO. JAPAN, LTD.

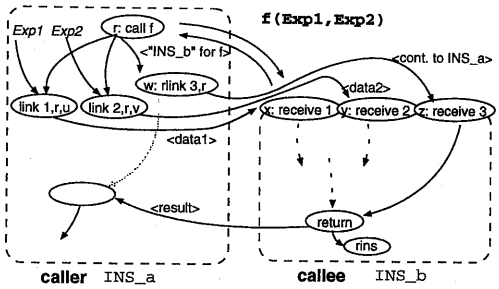


図1　細粒度の関数適用 f(Exp1,Exp2)

ログラムの実行効率を向上させることを目指す.

本稿では，まず2節で，細粒度の実行方式について述べ，次に3節でストリクトな実行方式の導入について議論する．4節では，必要な非ストリクト性を保証しかつ関数呼出を効率化するための実行機構とコード生成について述べる．評価の結果，前記の手法を用いることで，汎用計算機においても，実用的な速度で非ストリクトなデータフロー言語プログラムを実行することが出来ることが確認できた.

2. 細粒度実行

非ストリクトな実行のための柔軟な実行スケジューリングを可能にし，なおかつ細粒度の並列性を活用するために，細粒度の仮想計算機を設定する場合が多い[1)4)]．ここでは，特に関数呼出に焦点をおいて，その細粒度実行方式について述べる.

2.1 細粒度関数適用

各関数は，複数のスレッド (排他的に実行される命令列) から構成される．また，関数呼び出しの際には，引数は個別に別スレッドに渡されるため，複数の引数を持つ場合，完全に引数が揃わずとも部分的に計算を進めることも可能である．図1に，**Exp1** と **Exp2** を引数とした関数 **f** の呼出のコードの概略を示す．図中 caller 側の **call** は caller 側の実行コンテクストを生成するための，**link** は引数データを渡すための，**rlink** は caller 側の継続点を callee 側に渡すための命令ノードである．callee 側の **receive** は結果を受けとるための，**return** は指定された継続点へデータを渡す命令ノードである.

Exp1 と **Exp2** のうちどちらかの引数が求まれば，もう片方が求まっていなくても，部分的に計算を進めることでき，非ストリクト性の実現が可能で，理想的な実行環境であれば細粒度の並列性を活用し高効率が期待できる.

2.2 細粒度実行モデル

我々の実行モデルでは，各関数は一つ以上のスレッドから構成され，実行時に関数内ではスレッドレベルで動的スケジューリングがなされる．ある関数インスタンス内で実行可能なスレッドがなくなった場合，他の関数インスタンスのスレッドに実行を切替えることができるため非ストリクトなセマンティクスを持つ言語を eager に実行することが出来る (lenient 方式)．大きな遅延 (および予測できない大きさの遅延) を伴う命令を実行する場合は，命令の発行と返される結果に対する処理を分離し非同期に実行可能とする split-phase 操作とする．命令発火後他の実行可能なスレッドに実行を切替え，遅延をオーバーラップしスループットの向上を図ることも可能である．このような実行モデルに基づくコードは，細粒度並列処理をサポートする計算機では，ほぼダイレクトに効率良く実行することが可能と考えられる.

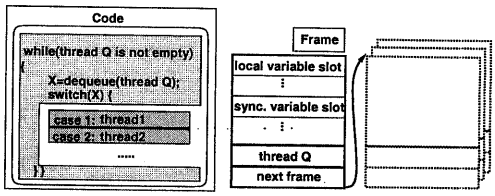


図2　インスタンス内のスレッドスケジューリング

各関数インスタンス毎に局所変数やスレッドの同期変数などの実行時コンテクストを保持するためのデータメモリ上の領域 (以下フレームと呼ぶ) を割り付ける．同一インスタンスに属するスレッドはフレームによりその実行コンテクストを共有する．関数が複数のスレッドで構成される場合，各スレッドごとの同期変数をコンパイル時に決められた値に設定し，スレッドの起動が試みられる度に継続スレッドの同期変数を1減らし，同期変数が0になったスレッドはスレッドキューに入れる．インスタンスはスレッドキューに並べられているスレッドを逐次に実行し，インスタンス終了命令が実行されれば終了する．インスタンス終了命令を実行する前に実行可能なスレッドがなくなった場合，そのインスタンスは実行をサスペンドし制御をインスタンスレベルのスケジューラにもどす．(図2参照)

インスタンスレベルのスケジューラでは，実行可能なインスタンスのフレームをフレームキューに並べて管理している．まず，スケジューラはフレームキューからフレームをとりだし，対応する関数コードにフレームを渡して制御を移す (インスタンスの起動又は再起動)．インスタンスが実行を終了して戻ってきた場合は，そのフレームを開放する．サスペンドして戻ってきた場合は，そのフレームをアイドルプールへ入れる.

アイドルプールに入れられた (サスペンドしているインスタンスの) フレームは，他のインスタンスからデータを受け取ることにより，スレッドが発火される場合がある．この場合は発火されたスレッドがスレッドキューに並べられ，サスペンドされていたインスタンスのフレームはフレームキューに並べられる．(図3参照)

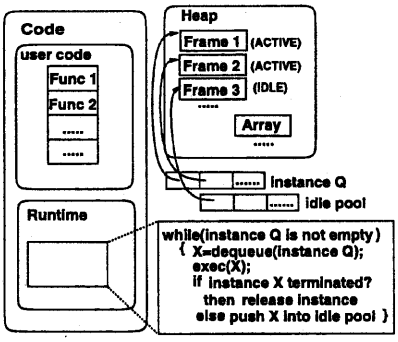


図3　スレッドレベルのスケジューリング

3. ストリクト実行の導入

3.1 汎用計算機上での細粒度実行

先に述べたような，仮想マシンコードを用いるとデータフロー計算機を対象にした実装では高い効率を期待できる．しかしながら，汎用計算機が実装の対象である場合，図2，図3に示すような実行システムをソフトウェアで実現する必要があり，細粒度処理の制御が大きなオーバヘッドとなる．

例題としてlowからhighまでの和を計算するプログラムについて考える．以下のプログラムから細粒度のコードを生成すると図4のようになる．図中の実線の矢印はスレッドの継続アークを表し，波線の矢印は間接的な継続アークを表す．実線で囲まれた部分はスレッドを表す．

```
function summ(low,high:integer) return(integer)
= if low = high then low
  else {let mid:integer = (low+high)/2
        in summ(low,mid) + summ(mid+1,high)};
```

汎用ワークステーション上に，図2，図3に示した実行システムを実現し，図4に示されたコードをもとにコードを生成し，実行時間を計測した．最初からC言語で直接記述したコードの実行時間も計測した☆

その結果，直接書いたCプログラムで約0.007秒，データフローコードで約4秒と桁違いの実行速度であった．図4に示されたコードでは，関数summの実行インスタンスはヒープメモリ上に確保され，引数lowとhighは別々のRECEIVEで渡されている．そのような処理はこのプログラムに対しては動作保証という観点からはオーバースペックで，汎用計算機上では効率良く実行できない，

3.2 スレッド結合

ここでは，非ストリクトセマンティクスを保証した上で，細粒度な状態のスレッド同士の結合を行ない，粗粒度な状態のスレッドを生成することについて検討する．スレッド結合アルゴリズムとして，separation constraint

☆ low=1, high=1000でオプションをつけずにgccでコンパイルし50MHzのSS10で実行

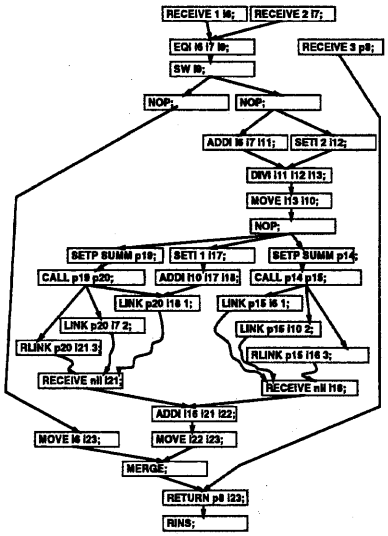


図4　細粒度コードのグラフ

partitioning[9]が提案されている．そのアルゴリズムはスレッド間の分離条件“他の計算体や構造体への要求を行なうスレッドと返答を受け取るスレッドは別々のスレッドに分割されていなければならない”というルールを素直に表現した手法であり，大域的な関数間解析手法と組み合わせることによって非常に高いスレッド結合能力を示す．また，スレッド結合過程において，対象計算機の特性を考慮して特定のスレッド同士を結合する/しないをコントロールできるという特徴を持つ．

ここでは，separation constraint partitioningをベースに用いるが，関数呼出の効率化を主目的に，以下のような優先順序を付けてスレッド同士の結合を行なうことにする．

(1)　インスタンス生成命令とインスタンス間データ転送命令を含むスレッド
(2)　その他のスレッド

スレッド結合の様子を先のsummプログラムを用いて示す．図5にスレッド結合の結果を示す．

図5中に示されるように，summでは，すべてのRECEIVEを一つのスレッドにまとめることが出来，関数をストリクトに実行できるようになっている．さらに，ファン・アウトを持つスレッドは条件分岐SWで終るものだけであり，関数内の実行時スレッド数は一つだけのため，汎用計算機上ではスタックを用いてこの関数を実行することが出来る．

先ほどの図5のsummのプログラムCコードから，関数の実行にスタックを利用するコードを生成し実行させたところ，約0.015秒で細粒度のコードと比べ桁違いの実行効率を達成でき，これはC言語で直接書いたCプログラムの約2倍程度の実行時間の長さである．

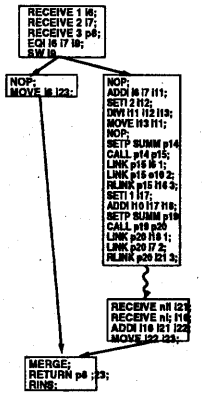


図5　スレッド結合後のグラフ

4. Hybrid 実行方式

非ストリクト言語のプログラム中で，数多く存在すると予想される，ストリクトに呼出可能な関数は先に述べた手法で効率良く実現出来る見通しが立った．しかしながら，非ストリクト言語の実際のプログラムにおいては，関数や条件分岐などいくつかのレベルでの非ストリクト性のなかでも，特に構造体の非ストリクト性が有用で，プログラム記述においては構造体の非ストリクト性を利用したものが多いとされている[7]．関数がストリクトに呼びだし可能だとしても，I-structures[3]のような非ストリクトな構造体へのアクセスを含む場合，その関数は実行がサスペンドする可能性がある．ここでは，ストリクトに呼び出せるが，サスペンドする可能性のある関数の効率良い実行方式を導入する．

柔軟なスケジューリングを保証するために，インスタンスフレームをヒープ上に確保するコストはかなり高い．そのため，静的解析でストリクトに関数を呼び出せる場合，スタックを利用したフレームの確保を積極的に行なう．しかしながら，サスペンドを考慮して以下の二通りのスタックを用いた関数実行をサポートする．

(1) 実行開始後，動的スケジューリングが行なわれないことが静的に保証されるインスタンスのフレームはスタック上に確保する (スタックフレーム)．

(2) 動的スケジューリングが必ず必要との確証はないが，不要であることを保証もできない場合には，suspensive 実行という方式でインスタンスのフレームを確保する．

静的にサスペンドしうるかどうかを決定できないインスタンス生成において，投機的にスタックフレームでインスタンスを実行するシステムを用いる．サスペンドすることなく正常終了することができればそのまま処理を続行し，サスペンドしてしまえば，その時点でヒープフレームを生成しヒープフレームベースのインスタンス管理システムに移行する．

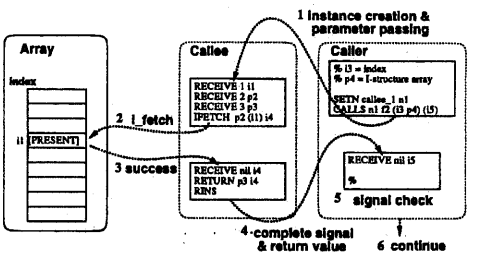


図6　suspensive 呼び出し (サスペンドなしに実行できる場合)

例として，個々の要素の値が生成されているかどうかわからない配列の要素へのアクセスを行なう関数の呼び出しについて，正常終了する場合と，サスペンドしてしまう場合について以下に説明する．正常終了する場合の実行の流れを図6に示す．破線は実行の順序を示す．

(1) 非ストリクトな配列操作を含む callee を caller がスタックフレーム上に呼び出す

(2) 非ストリクトな配列に callee から読み出し要求 (IFETCH) が出る

(3) 配列の要素の読み出しに成功し，callee は計算を続行

(4) callee は無事に計算を終了し，正常終了のシグナルと返り値を caller に返す

(5) caller ではシグナルをチェックし callee が無事に計算を終了したことを確認

(6) caller は計算を続行

サスペンドしてしまう場合の実行の流れを図7に示す．

(1) 非ストリクトな配列操作を含む callee を caller がスタックフレーム上に呼び出す

(2) 非ストリクトな配列に callee から読み出し要求 (IFETCH) が出る

(3) 配列の要素はまだ確定しておらず，読み出し要求は遅延される

(4) callee の実行はサスペンドするので，コンテクストを heap 上に作成

(5) callee はサスペンドしたことを知らせるシグナルと自分の heap 上のコンテクストへのポインタを知らせる

(6) caller ではシグナルをチェックし callee がサスペンドしたことを知り，自分の計算もサスペンドすることを検出

(7) 自分もコンテクストを heap 上に作成し継続情報をたどれるようセットする．

5. 評　　価

5.1 簡単な例題

配列 a と b を参照し，その積行列を生成する以下の関数について考える：

```
function matmul(a,b:AR;n:integer) return (AR)
= mkarray (i,j) in ([1..n],[1..n])
```

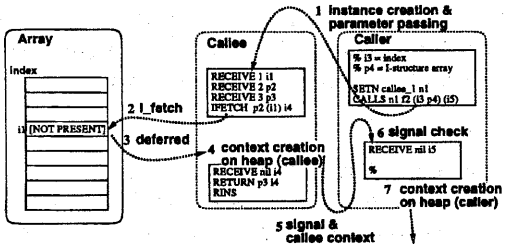


図7　suspensive 呼び出し (サスペンドする場合)

```
body foreach k in [1..n]
        sum a[i,k]*b[k,j];
```

ここで AR は array of real の略である．このプログラムでは mkarray で各要素を計算する body とその中の foreach において値を計算する際に関数インスタンスが起動される．このプログラムをサイズ 20 で実行した時の結果を表 1 に示す．

- heap 版では，すべてのインスタンス生成がヒープフレームとして生成され，配列のアクセスは IFETCH のままで split-phase 処理で行なわれる．
- suspensive 版では，配列のアクセスは IFETCH のまま．そのため，配列アクセスを含まない関数の呼び出しはスタックフレームによる呼び出しだが，配列アクセスを含む関数の呼び出しは suspensive による呼び出しとする．
- stack 版では，配列はストリクトと考えられそのアクセスはすべて (split-phase 処理ではない) 通常の FETCH に決定され，すべての関数が 1 スレッドで構成される．配列のアクセスを含む関数もスタックフレームによる呼び出しとなる．

関数 matmul の場合，引数である配列 a，b に着目して実行方式を選択する．配列 a，b に関して：

- 非ストリクトで，参照要求を出した時点での値の生成状況がわからない場合は heap 版，
- 参照要求を出した時点での値の生成状況を事前に確定できないが，かなり生成済みと期待できる場合は suspensive 版，
- 参照要求を出した時点で値が生成されていることがわかっている場合は stack 版

表 1 のデータは，配列 a，b の値は生成済みの条件で実行させた結果である．今回の測定条件では，実際は動的スケジューリングなしで実行が可能で，stack 版で実行すれば良いが，比較のため heap 版，suspensive 版でも実行してデータをとった．

- heap 版は非常に効率が悪い．これは非常に多数のスレッド間同期チェックとスレッド間コンテクストスイッチが行なわれたからである．配列操作のストリクト化が行なわれないため，配列へのアクセスがサスペンドされる状態も発生していた．
- stack 版と比べ suspensive 版は実行時間が 10%ほど遅い．これは IFETCH と suspensive 呼び出し中のサスペンドチェックの部分のオーバヘッドが影響しているためと考えられる．

	heap	suspensive	stack
実行時間 (sec)	9.4	4.1×10^{-2}	3.7×10^{-2}

表 1　matmul(size20) 実行結果

この結果より，suspensive 版に関して以下のことがいえる．

- 配列操作に動的なスケジューリングが実際は不要だった場合は，heap 版に比べ相当速く，stack 版と比べてもそれほど遅くない実行時間を達成できる．
- 配列操作に動的なスケジューリングが必要だとしても，suspensive 版を使えば対処可能であり，動的なスケジューリングが必要な操作と必要でない操作が混在していても対応可能である．
- 配列操作に動的なスケジューリングが必要かどうか静的にわからない場合に有効．

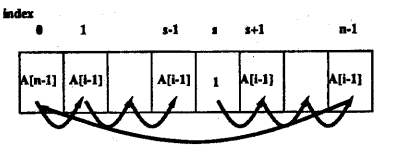
5.2　オーバヘッドの検証

実行時に結局動的スケジューリングが必要な場合に，

- 最初から heap 版としてヒープフレーム確保を選択しておく実行と，
- suspensive 版を用いてインスタンスを途中まで実行し，途中からヒープフレームへ変換する場合

のオーバヘッドについて調べる．

次のように，サスペンドするインスタンス数を変更可能な人為的な例題を用いて実験を行なった．




```
A:AI = mkarray i in ([0..n-1])
        body if i = s then 1
             elsif i = 0 then A[n-1]
             else A[i-1]
```

本例題プログラムでは，s がパラメータとなっており，要素生成順を静的には決定できない．配列要素値生成インスタンスは suspensive 呼び出しで行なわれる．配列は IFETCH でアクセスされ，suspensive が正常終了/ブロックされるかは IFETCH が正常終了/サスペンドされるかに依存する．コンパイルして実行した suspensive 版の結果を表 2 に示す．また，suspensive 呼び出しの効率を確かめるために，heap 版の実行結果を表 3 に示す．

二つの表を比較すると，

- 生成したインスタンスのサスペンド数
- 実行時間

	パラメータs			
	0	1000	4000	9999
heap instance	1	1001	4001	10001
suspend した heap ins.	0	1	1	1
suspensive instance	10000	10000	10000	10000
正常終了した suspensive	10000	9000	6000	1
suspend した suspensive	0	1000	4000	9999
正常終了した IFETCH	10000	8999	5999	0
suspend した IFETCH	0	1001	4001	10000
実行時間 (sec)	1.0×10^{-1}	3.6×10^{-1}	4.0	2.4×10^{1}

表2 配列生成 (suspensive call)(size10000)

	パラメータs			
	0	1000	4000	9999
heap instance	10001	10001	10001	10001
suspend した heap ins.	1	1001	4001	10000
正常終了した IFETCH	9999	8999	5999	0
suspend した IFETCH	1	1001	4001	10000
実行時間 (sec)	2.4×10^{1}	2.4×10^{1}	2.4×10^{1}	2.4×10^{1}

表3 配列生成 (ヒープフレーム call)(size10000)

	C	Sisal	Non-Strict
実行時間 (sec)	1.6×10^{-2}	3.7×10^{-2}	3.7×10^{-2}

表4 matmul(size20) 実行結果

は同等であった．このことは，インスタンスの起動前にヒープフレームを確保することと，インスタンスを途中まで実行してからヒープフレームを確保することはほぼ同等のコストであることを示している．

6. 議論および関連研究

6.1 他言語処理系との比較

行列積の例題を，C と Sisal で記述してコンパイルし実行させた結果を表4に示す．Sisal は，関数型言語であり，ストリクトセマンティクスを持つ言語である．Sisal は非データフロー計算機上でも Fortran と同程度の効率を得ることが可能であることが示されている[3]．

非ストリクトセマンティクスをもつデータフロー言語でも，適切なコンパイルを行なうことにより，ストリクトセマンティクスをもつ Sisal とほぼ同等の実行速度が得られた．ともに処理系のバックエンド言語として C を用いており，最初から C で記述したものと比較すると二倍程度の実行時間の長さであった．

6.2 関連研究

separation constraint partitioning[9]は，ここで用いている静的なスレッド生成法に非常に近い。関数レベルの非ストリクト性に関する分離制約の扱いは同等であるが，我々の手法の方が非ストリクトな構造体処理に関する分離制約の扱いが洗練されたものであるためより最適化されたコードを生成できる。

並行に実行される細粒度の計算体のフレームを実行時に効率良く管理する問題は，非ストリクトな関数型言語だけでなく並列オブジェクト指向言語でも問題になる[4][8][11]。本稿で述べた実行時システムは，TAM の 2 レベルスケジューリングと Concert システムの投機的スタック実行を組み合わせたような手法で，非ストリクト言語プログラムを効率良く実行することを目指したものである。

7. おわりに

本稿では，柔軟な記述力と潜在的な高並列性を持つものの，特別の細粒度実行の支援機構を持っていない計算機には効率良い実装が容易でなかった非ストリクトなデータフロー言語を，汎用計算機上へ効率の良く実装する手法を導入することを提案した．ここでは，必要な非ストリクト性を保証しかつ関数呼出を効率化するためのコード生成について論じた．次に非ストリクトな方式とストリクトな方式をともにサポートする実行時システムについて述べた．予備評価の結果，前記の手法を用いることで，汎用計算機においても，実用的な効率で非ストリクトなデータフロー言語プログラムを実行可能といえる．

参考文献

1) M.Amamiya and R.Taniguchi : "Datarol: A Massively Parallel Architecture for Functional Language", Proc. IEEE 2nd SPDP, pp. 726-735, 1990.
2) Arvind, R. S. Nikhil, and K. K. Pingali "I-strutures:Data Structures for Parallel Computing" Technical Report TR-87-810,Cornel University, Ithaca, New York 14853-7501, Feb, 1987.
3) D. C. Cann "Retire Fortran? A Debate Rekindled", CACM, Vol.35, No.8, p.p.81-89, 1992.
4) D. E. Culler et al. "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine" In Proc. of 4th ASPLOS, 1991
5) T. Kawano, S. Kusakabe, R. Taniguchi, and M. Amamiya. "Fine-grain multi-thread processor architecture for massively parallel processing"In *Proc. of HPCA'95* pp.308–317, Jan. 1995.
6) B. Lee and A. R. Hurson, "Dataflow Architectures and Multithreading," IEEE Computer, Aug. 1994, pp.27-39.
7) K. E. Schauser and S. C. Goldstein. "How Much Non-strictness do Lenient Programs Require?" In Proc. FPCA, 1995
8) J. Plevyak, V. Karamchet, X. Zhang, A. A. Chien "A Hybrid Execution Model for Fine-Grained Languages on Distributed Memory Multicomputers," In Proc. of Supercomputing'95. 1995.
9) K. E. Schauser, D. E. Culler, and S. C. Goldstein "Separation Constraint Partitioning - A New Algorithm for Partitioning Non-strict Programs into Sequential Threads" In Proc. of POPL, Jan. 1995.
10) A. Shaw, Arvind, R. P. Johnson, "Performance Tuning Scientific Codes for Dataflow Execution" Proc. of PACT96, pp.198–207, 1996.
11) K. Taura, S. Matsuoka, and A. Yonezawa, "*StackThreads*: An Abstract Machine for Scheduling Fine-Grain Threads on Stock CPUs" Proc. of JSPP94, pp.25–32, 1994.