九州大学学術情報リポジトリ
Kyushu University Institutional Repository

データフロー関数型言語の並列化コンパイラにおける配列の静的コピー除去

日下部, 茂
九州大学システム情報科学研究院知能システム学部門

岡崎, 芳希
九州大学システム情報科学研究院知能システム学部門

谷口, 倫一郎
九州大学システム情報科学研究院知能システム学部門

雨宮, 真人
九州大学システム情報科学研究院知能システム学部門

https://hdl.handle.net/2324/5722

出版情報：並列処理シンポジウム, 1995年, pp.161-169, 1995-05
バージョン：
権利関係：

データフロー関数型言語の並列化コンパイラにおける配列の静的コピー除去

日下部茂[†] 岡崎芳希[‡] 谷口倫一郎[†] 雨宮真人[†]
[†]九州大学総合理工学研究科，[‡]現：九州電力(株)

概要

データフローモデルに基づき関数型のセマンティクスを持つ言語では，構造データを一部だけでも更新する場合，概念的にはコピー操作により新しい構造データを生成する必要がある．並列性の抽出を容易にし実行効率を向上させるコピー操作もあるが，そうでないコピー操作は削減しなければならない．本稿では，コンパイルの中間コードとして細粒度並列仮想マシンコードを設定し，その中間コードに対して行う配列の静的コピー除去法について述べる．

Abstract

Dataflow functional languages have attractive features in writing parallel programs. However semantics of such languages require a copy operation in updating even a small fragment of a data aggregate. Although such copy operations may ease exploitation of parallelism, we should eliminate copy operations that would not contribute performance improvement. In this paper, we present our copy elimination method on a fine grain virtual machine code in compiler.

1 はじめに

データの授受と計算の同期を同一視するデータフローモデルは自然な形で並列性を制御することが可能な計算モデルである．データフローモデルに基づき関数型のセマンティックスを持つ言語は，本質的に並列性を内在しているという魅力的な特徴を持っている．このような言語は破壊的代入を認めていないため，言語処理系では副作用を考慮することなくプログラム中の暗黙の並列性を容易に利用することが出来る．しかしながら破壊的代入がないため，例えばある構造データを一部だけでも更新する場合，概念的にはコピー操作により新しい構造データを生成する必要がある．このような操作で並列性の活用は容易となるが，現実には構造データのコピー操作のコストは高く，効率向上に寄与しないコピー操作は削減しなければならない．実行効率を落さず構造体の更新操作を実現するために，言語が，破壊的更新はされない構造データと更新は常に破壊的にされる構造データの2種類を用意し，危険だと認識した上でプログラマが責任を持って構造データの破壊的更新を行い効率を上げることを認める手法もある．しかし我々はプログラマが破壊的更新を意識することなく記述したプログラムから処理系が自動的に最適なコードを抽出することを目標としている．

構造体データのコピー操作による並列性の活用を考えた場合，オーバーヘッドと効率向上の実際のトレードオフは，実装対象の計算機毎に異なる．そのため，処理系が自動的に最適なコードを生成することは容易でない．我々はまず機種非依存の中間コードの段階において可能な限り静的に構造データのコピー除去を行い，その後，各対象アーキテクチャを考慮したコピー除去を行うアプローチをとる．本稿では，コンパイルの中間コードとして細粒度並列仮想マシンコード DVMC (Datarol Virtual Machine Code) を設定し，その DVMC 上で行う構造データの静的コピー除去の方法を提案し，そのような中間コードレベルでもかなりの構造体のコピー除去が可能なことを示す．単純なデータ依存解析だけでは，除去可能なコピー操作の数は限られるため，本方式では人工的な依存関係を作り除去可能なコピー操作

Static Array Copy Elimination in Lenient Dataflow Functional Language
Shigeru Kusakabe[†], Yoshiki Okazaki[‡], Rin-ichiro Taniguchi[†], Makoto Amamiya[†]
[†]Department of Information Systems, Graduate School of Engineering Sciences, Kyushu University / [‡]Kyushu Electric Power, Co.

の数を増加させる方法をとっている．人工的な依存関係を付加する際に，効率向上に寄与する可能性のある並列性を失わないよう依存関係付加を控え目になっても，問題中の並列性が少ない場合はほとんどの構造体のコピー操作を除去できることを例題を通して評価した．

本稿では，2節でコンパイラの概要，3節で中間コードの概要について述べ，4節でコピー除去法について述べる．また，幾つかの例題プログラムについて本方式を適用し評価した結果を6節で示す．

2　コンパイラの構成

我々はデータフローモデルに基づく言語が本質的に並列処理記述に適していると考えているが，従来データフロー言語は特別のハードウエア上でのみ実装されることが多かった．我々はデータフロー言語も汎用並列計算上で効率良く実装することは可能と考えており，以下のようなコンパイル方針をとっている．図1に示すように，ソースプログラムは，字句解析，意味解析，データ依存解析を経て細粒度並列仮想マシンコードDVMC(Datarol Virtual Machine Code)という，各機種に共通の中間コードに変換される．DVMCはグラフ表現可能なマルチスレッド・コントロールフローコードで，同一の実行コンテクストを共有する一つ以上の命令列である命令スレッドと，それらスレッドの実行順序関係を表す継続スレッド指定(グラフではアークで表現)から構成される．このDVMCに対し，共通部分式の除去，機種非依存レベルのスレッド合成など，機種非依存の最適化を施すことで，各機種共通の最適中間コードを得る．本稿で述べる仮想マシンレベルの構造データの静的コピー除去もこのフェーズで行なう．機種非依存の最適化を行った後，各機種毎に異なる並列処理操作コスト，通信コストに関するコストパラメータを読み込み，各機種向けにスレッドの粒度調整を行ったコードを生成する [6].

3　仮想マシンコード DVMC

3.1　仮想マシンとDVMC

DVMCを実行する仮想マシンDVM(Datarol Virtual Machine)は任意の形態のネットワークで結合された一つ以上のプロセッサエレメントから構成され，各プロセッサ毎にサイズ無制限のデータメモリ(レジスタファイル)と命令メモリおよび(仮想単一空間の)大域構造データメモリSM(Structure Memory)を持つ．SM上には，任意サイズの連続領域を確保することができ，配列，レコード等の構造データはこのSM上に実現される．インスタンス間の配列などの受け渡しは，データコピーではなくSM上のポインタを受け渡して行なう．

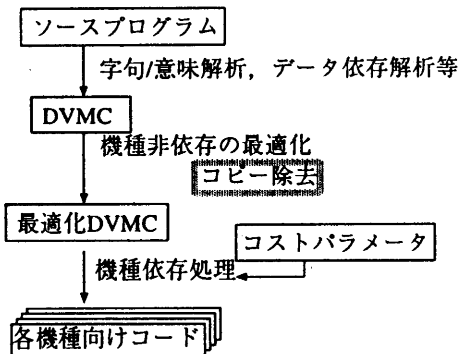

図1: コンパイラの構成

仮想マシンではデータフロー関数型プログラムの実行時に生成される関数適用インスタンス程度の粒度で並列実行を行い，各インスタンス毎に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Frame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Frameによりその実行コンテクストを共有する．各インスタンスは，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，スレッドレベルの並列処理が行なわれる．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ノンストリクト(lenient)なセマンティクスを持つ言語を実装することが出来る．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．

DVMCの各スレッドは原則として自分を指している全てのアークからコントロールトークンを受け取ると，発火する．例外としてeureca同期を実現するmerge命令[1]を先頭とするスレッドのみ，1つでもコントロールトークンを受け取ると発火する．な

[1]以降に出てくるDVMC命令の詳細については，本稿の末尾に付録として添付されている表を参照されたい(実際のDVMCは演算部や継続指定部などから構成されるが，表は演算部のみを説明している)．

お，merge ノードを先頭とするスレッドへは複数のスレッドからトークンが来ることはない．スレッドは発火するとスレッド内の命令列を実行し出力アークへフローを流す．

3.2 配列処理の DVMC の例

例として，配列 A の i 番目の要素と j 番目の要素を入れ換えた配列を返す式

```
update(update(A,[j],A[i]),[i],A[j])
… (1)
```

を DVMC に変換したものを図 2に示す．

長円で囲まれているのがノードであり，この DVMC では，1 ノードが 1 命令に対応している．ノード間をつなぐ矢印がコントロール・フローを表すアークである．アークはその出発ノードの命令の種類によって，遅延することなく後続命令へ制御を移すことができる命令から出るアークと，他のプロセッサとの通信や関数インスタンスの生成のため遅延が大きいと考えられ split-phase として扱われる命令から出るアークに分けられ，図ではそれぞれ実線と破線に区別されている．(破線が split-phase 命令からのアークを表している．)

図 2の DVMC において，命令 fetch A i ai は，配列 A の i 番目の要素を読み出し，ai へ格納する．命令 copy A A' は，配列 A 全体をコピーし，新たに生成される配列を A' とする．命令 replace A' j ai は，配列 A' の j 番目の要素を ai の値に破壊的に書き換える．copy と replace を用いることで一部を更新した新しい配列を生成している．これらの配列操作の命令は，潜在的に予測不可能な遅延を生じる可能性があるため (例えば実機上で，配列要素を複数のプロセッサに分散配置させている場合，自プロセッサだけで処理できず通信を生じる場合がある)，命令の実行は split-phase と考え，命令ノードからのアークは破線となる．ノード中の A，i，aj 等は実際にはポインタやレジスタを表す記号であるが，ここでは便宜上変数のように表現している．一番下の replace ノードを実行した後の配列 A"が，(1) 式の結果となる．

4 コピー除去と node-reordering

本節では，(1) 式を例に DVMC に対して行う機種非依存レベルでの構造データのコピー操作の除去法について述べる．

まず，コピーを除去できるための条件を説明する．次に単純な依存関係の解析だけではコピーが十分に除去できないことを示す．つづいて，その除去可能な条件を満たすコピー操作を増やすために，潜在的な遅延時間を増加させない範囲で，DVMC のノードの並べ替え (node-reordering) を行う方法について説明する．

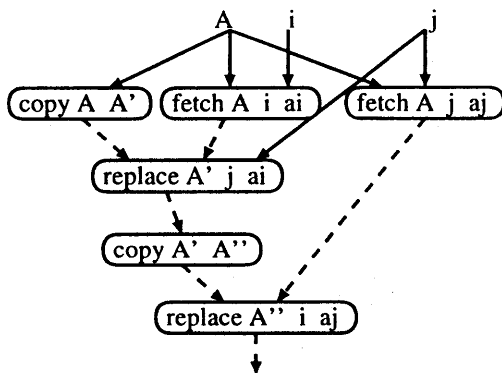

図 2: (1) 式の DVMC

4.1 コピー操作の除去

ここで述べるコピー操作の除去とは，具体的には，DVMC 中の各 copy ノードが，次の条件を満たしている場合，その copy ノードを move ノードに変換 (それにともない split-phase 操作でなくなりグラフではノードから出るアークも破線から実線に変換) することである．命令 move A A' は，配列 A を指すポインタを A' へ格納する．

【コピー操作が除去可能な条件】
ある copy ノードに着目した場合，その copy ノードと同じ構造データにアクセスを行なうノードが次のいずれでもないとき，着目している copy ノードがその配列への最後のアクセスと保証できるため，そのコピー操作は除去可能である．

(1) その copy ノードより後に実行される

(2) その copy ノードと並列に実行される

図 2の DVMC の場合，左下の copy ノードは上の条件を満たしており，他に配列 A' に対しアクセスを行なうノードがないので，配列 A' をコピーすることなく，破壊的に上書きすることが出来る．つまり，この copy ノードを単にレジスタの内容を移す操作を行う move ノードに変換する．また move ノードの実行は split-phase ではないので，move ノードか

ら出るアークを実線に変える (図3). 一方, 左上の copy ノードは, 上の条件を満たしておらず, そのノードと並列に, 配列 A に対する要素参照 (右上の fetch ノード) が実行され得るので, 副作用の生じる可能性があり, このコピー操作は除去できない.

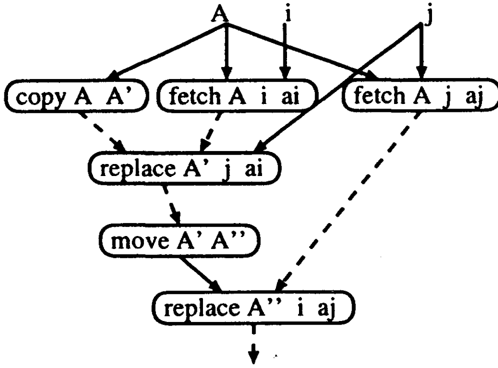

図 3: (1) 式についてデータ依存解析のみでコピー除去した DVMC

4.2 node-reordering

単純にデータ依存関係しか反映していない DVMC には前述の条件を満たす copy ノードはまだ少ない. 除去可能なコピー操作を増やすために, DVMC のノードを並べ換える操作 (node-reordering) を行う.

先の例で図 2の DVMC の左上のコピー操作は, このままでは除去することができない. もし, copy ノードの実行を, 右の 2 つの fetch ノードの実行後に延期できれば, そのコピー操作を除去することができる.

【人工依存アーク ACA】
ある copy ノードと同じ構造データにアクセスし, 並列に実行され得る各ノードからその copy ノードへ ACA(Artificial Control Arc) を挿入する. これによりその copy ノードの実行は他のアクセスノードの実行以降に延期される.

ただし, ACA の挿入によって人工的な依存関係が生じ, 並列性の損失や, クリティカルパス長の延長などの弊害が起こり得る. そのため, ACA の挿入の際は増加を見込めるコピー除去数と, 人工的な依存関係の発生により失うものとのトレードオフを考慮して行なう必要がある.

ここで, DVMC の深さを次のように定義し, node-reordering の際に用いることで, ACA 挿入による潜在的遅延時間の増加を防ぐ.

【DVMC の深さ】
あるノードにおける DVMC の深さを, グラフの最上ノードからそのノードまでたどる際に通過する split-phase 操作の最大数 (破線のアークの最大本数) と定義する.

例えば図 2の部分 DVMC の場合, 一番下の replace ノードの深さは 3 である.

実際の各種並列計算機の上では構造データの配置法やアクセス所要時間も異なり, また活用可能な並列性の粒度や並列度も異なる. しかしながら仮想計算機レベルでは DVMC 全体の深さが大きくならなければ遅延の影響を被って実行時間が増大することはないとみなし, 次の条件を満たしている場合のみ ACA を挿入する.

【ACA 挿入の条件】
ある copy ノードと並列に実行される, 同じ構造データに対するアクセスノードの深さの最大値が, その copy ノードに対応する replace ノードの深さ未満である.

図 2の DVMC の場合, 右上の 2 つの fetch ノードから左上の copy ノードへ ACA が挿入される. (1) 式について, node-reordering を行い, コピー操作の除去を施した DVMC(冗長なアークは取り除いてある) を図 4に示す. 図 4の一番下の replace ノードの深さは 2 である. 図 2の DVMC に比べ, 全てのコピー操作を除去できており, DVMC 全体の深さも 1 減っている.

5 コピー除去法

本節では node-reordering を行ないながら除去可能なコピー操作を解析する方法を示す. なお, 以降, 先祖や子孫などの用語は文献 [5] に従う. 以下に述べる解析は, あるコピー操作に着目し, そのコピー操作が含まれる関数内で行なう解析である. 対象プログラムの DVMC 中の copy ノードの集合を C とし, $c \in C$ であるすべての c について, c が属する関数定義の DVMC を対象にして以下の解析を行なう.

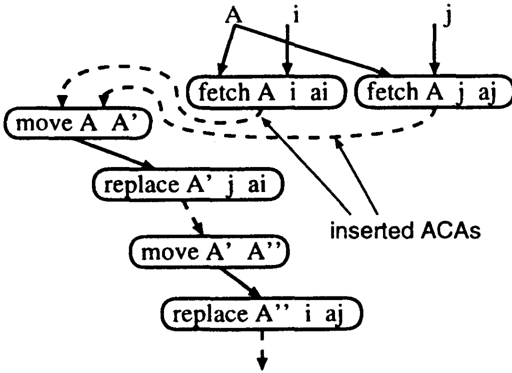

図4: (1) 式についてACAを用いコピー除去を行ったDVMC

解析手順

(i) c が存在する関数のDVMC中で，c の第1オペランド (コピー対象の配列を指すポインタ) と同じ配列を指す可能性のあるポインタ集合を条件分岐も考慮したうえで求め，その集合を P_c とする．

(ii) c が実行されるとした場合にとり得る実行パス上にあり $p \in P_c$ である p へアクセスを行う命令の集合 O_c を求める．ただし，$c \notin O_c$ とする．

(iii) DVMC中の c の先祖のノード集合を A_c，子孫のノード集合を D_c とし，O_c を以下のように分ける：

- c の先祖 O_{c_a}: $A_c \cap O_c$,
- c の子孫 O_{c_d}: $D_c \cap O_c$,
- およびそれ以外の $O_{c_{para}}$: $O_c - (O_{c_a} \cup O_{c_d})$.

ここで $O_{c_{para}}$ は，c とは並列に実行され得るノード集合である．

(iv) 次の条件のいずれかが満たされる場合，配列をコピーしなければ副作用が生じるおそれがあるので，c は move ノードに変換できない (解析終了).

1. $O_{c_d} \neq \phi$.
2. $\exists$return $\in O_c$.
3. $\exists$link $\in O_{c_{para}}$.

いずれも満たされない場合は次ステップへ．

(v) $o \in O_{c_{para}}$ である o についてDVMCの深さを求め，その最大値を d_{max} とする．また，c に対応する replace ノードのDVMCの深さを $d_{replace}$ とする．

(vi) $d_{max} \geq d_{replace}$ ならば，ACAを挿入してコピーを 削減することによる効率向上以上に逐次化による実行時間の増大のおそれがあるので，ACAは挿入せず，c は move ノードに変換せず解析を終了．そうでなければ $o \in O_{c_{para}}$ である o それぞれから，c へACAを挿入し次へ．

(vii) $p \in P_c$ であり，かつ関数適用時に引数として渡されてくる p それぞれについて，関数間解析を行う．その結果，c を move ノードに変換可能と判明した場合，c を move ノードに変換し解析を終了する．変換できないことが判明した場合，それまでに挿入したACAを全て取り去り解析を終了する．

上記解析において，解析精度を上げるため，条件式や関数間の解析に関しては以下の点を考慮している．

条件式

- 配列にアクセスするノードのうち，条件節内に存在するもの (プログラムの then 節，else 節中で実行されるもの) は，実行されるかどうか静的には分からないため，ACAの挿入においてはそれらのノードから直接ACAを挿入することはできない．このようなノードからACAを挿入する場合は，条件式の最終ノード (merge ノード．条件式では必ず実行される) から挿入する．この場合も，DVMC全体の深さが増加しないよう考慮する．

関数間解析　解析手順の (vii) で呼び出される関数間解析は，関数の呼び出し関係の，呼び出され側から呼出側をボトムアップにたどるという方針をとっている．実際の解析は，着目する構造体が copy 命令のオペランドではなく，関数間で参照が受け渡されている構造体であることを除き，上記の関数内解析とほぼ同様である．関数間解析で考慮している主な点を以下にあげる．

1. 関数適用においては，呼び出されている側の関数を解析することで，呼出側の引数を渡す link 命令の子孫や，継続を渡す rlink の先祖の関係などをできるだけ詳細に解析する．
2. ポインタ集合 P_c の決定の際には，関数適用の引数と返り値を解析することで精度の高い

決定を行なう.

3. 解析手順 (ii) で関数適用を含む場合の返り値に対する O_c の判定時には，呼出され側の関数の解析も行ない精度の高い判定を行なう.
4. 関数間解析では，再帰関数での解析の無限ループを避けるため，一度解析したものにはマークを付け，解析時はマークをチェックする.

6 評価

6.1 コピー除去数

前節で示した方法を V 言語のコンパイラ [4] に組み込み，いくつかの例題プログラムに適用した結果を表 1に示す. 適用した例題プログラムは，クイックソート，マージソート，バブルソート，挿入ソート，ガウス-ジョルダン法，逆行列，騎士巡歴問題，高速フーリエ変換，およびリバモアループ (2, 4, 5, 10, 11, 14, 17, 19, 20 番) である. 表中，初期数は解析対象の DVMC 中の copy ノードの個数を表している. これはソースプログラム中の，配列要素の更新を行う update 式の個数と一致している. 除去数はコピー除去によって move ノードへ変換された copy ノードの個数を表す. 右端は，コピー除去の際 DVMC に挿入された ACA の本数を表す.

プログラム	初期数	除去数 () 内は ACA 数		残
		ACA 無	ACA 挿入	
quicksort	2	1		1
mergesort	3			3
bubblesort	2	1	1 (2)	
inssort	2	1	1 (1)	
gauss-jordan	2		2 (5)	
inverse	4	2	2 (4)	
knight	2	2		
fft	10	4	6 (14)	
loop02	1		1 (3)	
loop04	1	1		
loop05	1		1 (1)	
loop10	10	1	9 (9)	
loop11	1		1 (1)	
loop14	2		2 (2)	
loop17	6	6		
loop19	2		2 (2)	
loop20	2	1	1 (4)	
合計		20	29 (48)	
	53	49 (48)		4

表 1: 除去された copy ノード数 (空欄は 0 を表す)

表 1により，かなりのコピー操作を除去できていることが示されており，本稿で述べたような仮想マシンを設定した中間言語レベルの解析でもかなりのコピー操作を取り除くことが可能であるといえる. 表から node-reordering による ACA 挿入の効果が大きいことも分かる. 今回取り上げたもののうち，逐次的に配列を更新していく例題については，コピー操作をすべて除去できている. 潜在的に利用可能な並列性を保存するため，人工的な依存関係をむやみに増加させないような ACA 挿入法を用いても，本質的に逐次的なコピー操作は除去可能といえる.

表 1から，クイックソートは本質的に並列なアルゴリズムを用いているため，除去されないコピー操作があるが，このコピー操作は後の機種依存の最適化によっても除去することはできない. これにより，本質的に並列性がなく除去可能なコピー操作は，本稿で述べたような仮想マシンを設定した中間言語レベルの解析により取り除くことが可能であるといえる.

6.2 実装実験

コピー除去を行なった DVMC を AP1000 用のコードに変換し AP1000 上で 1 台のセルを用いて実行時間 (秒) を測定した結果を表 2に示す. この表より仮想マシンレベルの解析によるコピー除去でも，かなりの効率向上が期待できることがわかる. また node-reordering によるコピー操作の除去数の増大が，実行時間の減少にそのまま反映されており，node-reordering の有効性も確認できた. 同一アルゴリズムの C プログラムと比較すると一番差が大きい場合で数倍遅いが，V 言語は lenient セマンティクスを持ちコンパイラは細粒度並列処理を実現するコードを生成するため，そのオーバヘッドが原因と考えられる.

プログラム	コピー非除去	コピー除去版		C プログラム
		ACA 無	ACA 挿入	
bubblesort (1024 要素)	348	175	2.11	0.641
gauss-jordan (64x65 行列)	367	367	1.08	0.232
fft (1024)	16.1	8.50	0.613	0.184
loop20 (n=256, ,$loop$=1000)	96.2	55.9	6.35	5.39

表 2: AP1000(1 セル) 上での実行時間 (秒)

複数台のセルを用いた実行も行いコピー除去の効果を確かめたが，DVMC を AP1000 用のコード

に変換する際に単純な変換しか施していないため，通信のオーバヘッドなどが原因で絶対時間としてはかなり遅いものとなった．複数台のセルを用いる場合の効率の良いコード生成のためには，分散メモリ並列計算機上での配列の最適分散法やAP1000用のコード最適化など本稿で述べた解析の範囲を越えた処理が必要である．今後，以下のような検討を行ない，分散メモリ計算機上での効率の良いコード生成を行なう予定である．

- 仮想マシンの再設定：例えば分散メモリ並列計算機を対象とする場合，構造体メモリが分散している仮想マシンを設定し，そのコードに対しコピー除去解析を行なう．この場合は仮想マシンレベルで構造体メモリアクセスのコストが異なるため，ACA挿入判定に用いる深さの定義も再検討する必要があると考えられる．
- 機種依存部での解析：機種により最適な配列分散法は異なるため，このコピー除去法を機種依存部に取り込み，各機種に適するよう分割された部分配列に対しコピー除去解析を行なう．本稿で述べたコピー除去手法は分割後の部分配列にも問題なく適用できると考えている．

6.3 関連研究

関連研究に[1]や[2]がある．文献[1]の研究も関数型言語での構造体の効率の良い実装法に関するものであるが逐次処理を対象としており，本研究のように並列処理を前提としたものではない．文献[2]の研究は適用型データフロー言語SISAL[7]を対象としたコピー除去の研究で本研究と高い関連を持っている．そこでもnode-reorderingにより除去可能なコピー操作を削減する手法をとっているが，そこで提案されているコピー除去法では人為的なアーク(本稿のACAに対応)が冗長に挿入される，並列性が制限されるなどの問題が生じている[3]．

これらの研究に比べ，本稿で述べた手法は以下のような特徴を持つ．

- 文献[5]で提案された手法をもとに，冗長に挿入されるACAがな残らないような最適化を行なう．
- 本質的に並列性がない部分のコピー操作を除去し，利用可能な並列性は制限していない．
- 解析は全て静的に行なう．SISALでは実行時のリファレンス・カウンタの値により動的にコピーの制御を行なうノードを持つ中間言語を用いているが，リファレンス・カウンタを用いること自体に(特に分散メモリ並列計算機では)大きなオーバヘッドを伴うため，そのような中間言語を採用していない．

7 おわりに

本稿では，関数型プログラム実行のための，機種非依存の中間コードDVMCにおける構造データの静的コピー除去の方法について述べた．また，本方式を用いることにより，仮想マシンレベルでもかなりのコピー操作が除去できることを例題を用いて示した．

参考文献

[1] Adrienne Bloss: "Update Analysis and the Efficient Implementation of Functional Aggregates", Proc. 4th Functional Programming Languages and Computer Architecture Conference, pp.26-38, 1989.

[2] David C. Cann: "Compilation Techniques for High Performance Applicative Computation", Ph.D. thesis, Cololado State University, 1989. CSU Technical Report CS-89-108.

[3] Steven M. Fitzgerald: "Increasing Parallelism for an Optimization that Reduces Copying in IF2 Graphs", Proc. SISAL'93, pp.74-84, Oct. 1993.

[4] 日下部茂, 他:"超並列V言語とその実行方式", 並列処理シンポジウムJSPP'94論文集, pp.41-48, May 1994.

[5] 立花徹，他："データフロー解析による関数型言語Validのコンパイル法"，情報処理学会論文誌，Vol.30，No.12，p.p.1628-1638，1989.

[6] 山下欣宏, 他:"既存並列計算機を対象とした細粒度並列仮想マシンコードDVMCからのコードスケジューリング", 情報処理学会全国大会予稿集 Vol.4, pp.151-152, Sep. 1994.

[7] A. P. Wim Böhm, R. R. Oldehoeft, D. C. Cann, and J. T. Feo: "SISAL Reference Manual, Language version 2.0", Colorado State University - Lawrence Livermore National Laboratory, 1992.

付録：DVMCの抜粋 (演算部)

format	description
receive *slot reg*	callee 側のインスタンス間データ受取命令．*slot* 番スロットのデータを *reg*へセットする．*slot* 番スロットのデータは，caller 側の **link**, **rlink** 命令で送る．
return *rp reg*	callee からのインスタンス間データ転送，スレッド起動命令．*rp* が指す場所へ *reg*の値を送る．このとき，*rp* を渡した caller 側の **rlink** 命令の継続命令を起動する．
rins	インスタンス解放命令．カレントインスタンスを解放する．**rins** 命令以降，カレントインスタンスは使用できなくなるので，継続命令は存在しない．同一インスタンスを使用する全命令の中で，最後に実行しなければならない．
call rp_f rp_i	関数インスタンス生成命令．rp_fで生成する関数を指定する．rp_iへは，生成した関数インスタンスへのポインタがセットされる．
link *rp reg slot*	caller からのインスタンス間データ転送，スレッド起動命令．*rp* で参照するインスタンスの *slot* 番スロットへ *reg*の内容を送る．**link** 命令実行後，データを受け取ったインスタンスでは，**receive** 命令より始まるスレッドが起動される．
rlink *rp reg slot*	caller 側のインスタンス間データ受取り，スレッド起動命令．*rp* で参照するインスタンスの *slot* 番スロットへ *reg*へのポインタを送る．**rlink** 命令実行後，ポインタを受け取ったインスタンスでは，**receive** 命令より始まるスレッドが起動される．
sw *ri*	条件分岐命令．*ri* が真の場合 then 部の継続へ，偽の場合 else 部の継続へコントロールを渡し，制御を切替える．
merge	eureca 同期命令．複数の命令より起動されるが，いずれか一つの命令から起動されれば，直ちに次命令を起動する．sw 命令による複数に分岐したスレッドの合流点等に用いる．
move reg_i reg_j	reg_iの内容を reg_jへセットする．reg_iの内容は reg_jのタイプに解釈される．
copy rp_1 rp_2	rp_1の持つフラグの情報により配列全体のコピーを行い，新配列を rp_2に格納する．
fetch *rp* (ri_1 ... ri_n) *reg*	*rp* が指す n 次元配列のインデクス (ri_1 ... ri_n) の要素の内容を *reg*に格納する．
replace *rp* (ri_1 ... ri_n) *reg*	*rp* が指す n 次元配列のインデクス (ri_1 ... ri_n) の要素の内容を reg に書き換える (破壊的).

注：表中，reg_i は任意タイプのレジスタ，$r\{i|r|p\}_j$ は命令に応じ iReg/rReg/pReg を表す (添え字 (j) は区別の必要がない場合は省略)．ここで iReg(Integer type Register) は整数値および bool 値を格納 (bool 値としては 0 を false，0 以外を true と解釈)，rReg(Real type Register) は実数値を格納，pReg(Pointer type Register) はポインタを格納する．