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

Title	マルチスレッド型プロセッサによる関数型言語の実行方式に関する研究
Author(s)	Jin, Yu
Citation	
Issue Date	2004-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1810
Rights	
Description	Supervisor:日比野 靖, 情報科学研究科, 修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

マルチスレッド型プロセッサによる
関数型言語の実行方式に関する研究

金 ウク (210027)

北陸先端科学技術大学院大学 情報科学研究科

2003 年 2 月 13 日

キーワード: 並列、SML、スレッド、トークン、下降型解析、記号表.

概 要

本論文では、マルチスレッド型プロセッサアーキテクチャと関数型言語の特徴組み合わせることにより、関数型言語の高効率実行を目的とする。関数型言語、論理型言語、オブジェクト指向型のいずれのパラダイムも逐次的なアーキテクチャで効率良くシミュレートされるが、それらのパラダイムに内在する並列実行可能性は無視されている。また、関数型プログラムの実行に適したマルチスレッド型プロセッサアーキテクチャが提案されているが、マルチスレッド型プロセッサに適用したコンパイラは提案していなかった。本研究では、並列性検出と複数のスレッドの生成およびスレッド間の同期を行うコンパイラを提案する。まずは、Standard ML に着目する。

1 はじめに

本論文では、マルチスレッド型プロセッサアーキテクチャと関数型言語の特徴組み合わせることにより、関数型言語の高効率実行を目的とする。かつてプログラミング言語のアーキテクチャによるサポートはアプリケーション固有のサポートの間接的な方法として伝統的に一般的であった。しかし高級言語計算機は顧られなくなった。現在は RISC を中心としたマイクロプロセッサ時代を迎ている。高級言語計算機はソフトウェアの課題をハードウェアで解決するという試みであった。しかし計算機のユーザ評価は高級言語計算機の達成度ではなく総合的な費用効率の良さである。ソフトウェアの課題としては主に次の 2 つがある。１つは、ソフトウェア開発の方法を工学的に捉えようとするアプローチである。もう１つは、プログラミングのパラダイムを根本から見直し、これまでのノイマン型アーキテクチャに基づく命令型のパラダイムから他のパラダイムへ転換を図るものである。この中でも関数型、論理型プログラミングは並列処理あるいは並列計算機アーキテクチャの観点から関心が持たれた。しかしながらこれも総合的な費用効率の良さとは一致しない。費用効率という観点からは、より簡単な構成、より少ない資源で実現されるマシンの方が、性能で若干劣っていたとしても優位に立つ。従って、高級言語計算機は成立し得ない。

Copyright © 2004 by yu jin

一般にプログラム言語の処理は言語の表層的な構文の処理、演算実行順序制御、演算実行に分れる。このうち、第一の構文の処理はコンパイラに任せるのが費用効率の点から優れている。一方、ハードウェアによる高速化に最も効果があるのが演算実行部である。ハードウェアによる高速化は並列動作による効果である。高級言語処理に関して残された部分は、演算実行の制御である。これに対しては現在は命令パイプラインを有する逐次型アーキテクチャマシンを対象した命令生成と命令スケジューリングにのみ関心を払われた。関数型、論理型、オブジェク指向型は逐次型アーキテクチャで容易にシミュレートできるが、これらのパラダイムに内在する並列実行可能性は無視されている。また、各々のパラダイムの並列言語よるの並列実行に多くの研究があるが、言語の並列実行つまり並列性を検出するコンパイラについてはほとんど関心が払われていない。

2 関数型言語とその処理方式

本論文では、Standard ML に着目し、そのパラダイムに内在する並列実行可能性を探求する。関数型言語の効率良い実行を妨げる原因としては主に 2 つある。まず、プログラム中で頻繁に起きる関数の呼び出しによる、スタック操作の命令の実行である。この操作は通常のパイプラインではハザードを生じる。マルチスレッド型プロセッサの導入によって、ハザードを回避する。次に、プログラムに内在する並列実行可能性は無視されている。厳密な意味での関数型言語は副作用を持たない。プログラムはある関数 f で表され、プログラムの実行は入力 x に対する f の **適用** $f(x)$ を評価することと定義される。関数 f の定義中には別の関数や f 自身が用いられる。例えば、$f(x) = g(h(x), k(x))$ のような場合、この評価は、一番外側の g の評価から行うこともできれば、中の $h(x)$ や $k(x)$ の評価から行うこともでき、$h(x)$ と $k(x)$ を同時に評価することもできる。厳密な関数言語では、これらの異なる評価順序のいずれを用いても評価結果は同じであることが保証されている。この性質は並列処理の可能性を示している。従って、並列性検出とスレッド間の同期を行うコンパイラの検討が必要であると考えられる。

3 提案する並列性の検出と同期処理

本研究では、並列検出とスレッド間の同期処理を行うコンパイラの実装に必要な、データ依存関係と同期処理を以下のように提案する。

3.1 データ依存関係の検査方法

SML の式は、いつもある環境の元で式の評価を行う。環境とは、値によって束縛された変数の集合である。したがって、式は環境によって評価される。SML では、静的スコープ規則を持つ言語であり、変数の値は、その変数が定義された時点の値に固定され、以後

は変化することはない。この規則は関数定義にも適用される。関数本体の束縛変数以外の自由変数、それらはの変数の意味は、関数が定義された時点でのその変数に束縛されている値であり、関数が実行される時点での同一名の変数の定義とは関係ない。

従って、本研究では環境に注目しながら、データ依存関係の検査をします。コンパイル時、フロントエンドでの各段階で記号管理表に環境を作り、後の処理段階では記号管理表を調べることにより、データ依存関係をチェックする。記号管理表に新たに環境という属性を追加する必要がある。

3.2 式の間での同期処理

式の実行時間が種々であり、式の間に、並列処理をした後で同期を行う必要がある。式の値が、部分式の評価が完全に終わる前に値を返すと正しくない結果を招く。そんなの誤りを避けるために、本研究では同期処理を以下のように行う。

式の全体を評価する時、式の要素 (**部分式**) の評価はその式の値と完了状況を知るマークを一緒に返すことにする。要素が全完了したら、1 に、そうじゃないと 0 を返すようにする。例えば、次のような式を評価する時、

val Exp = exp_1 + exp_2 + exp_3;

exp_1;exp_2;exp_3 が並列実行可能なら、同期処理はマーク A,B,C をチェックすることによって判断する。図 1 は、提案する方法による上の式の同期処理の説明図である。

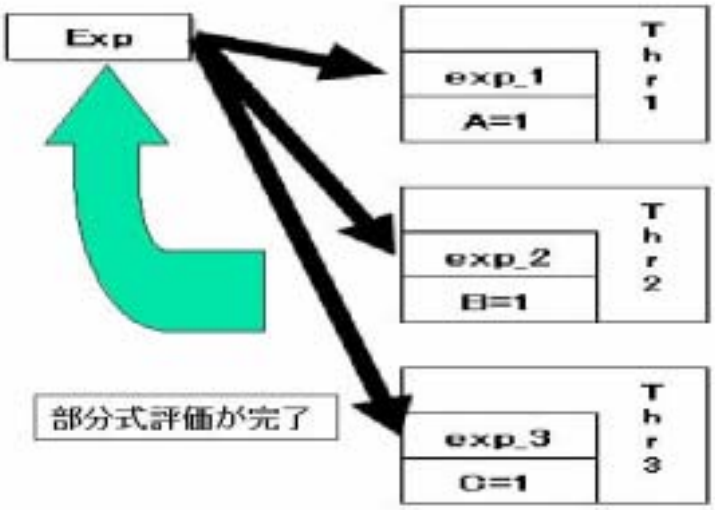

図 1: 関数合流のための同期処理

start exp_1; exp_2; exp_3;
val A = 0 ; B = 0 ; C = 0;
val Exp[A,B,C] = case [A,B,C] **of** [1,1,1]
=> Evaluation
| _ => Suspension

4　コンパイラの実装

4.1　字句解析

字句解析はコンパイラの最初のフェーズである。そのおもな仕事は文字を順に読みながら、構文解析の要求によってトークンの列を出力する。字句解析は、” 次のトークンの獲得” という要求を構文解析から受け取ると、入力文字を読み込んでいって、次のトークンを確定する。字句解析の実装に当たって、以下のように約束をする。データ型は整数型で、キーワードを識別子として使うのを禁止する。トークンを表 1 のように規定する。

トークン	::=	予約語	|	識別子	|	数字列	|	特殊記号				
予約語	::=	andalso	|	orelse	|	if	|	then	|	else	|	fun
	|	val	|	let	|	in	|	end	|	ture	|	false
識別子	::=	英字	|	識別子		英数字						
英数字	::=	英字	|	数字								

表 1: トークンの規定

字句解析は主に、空白の読み飛ばし; トークンの種類の判定; トークンの読み込み処理三つの部分で構成される。字句解析は常に、１つ先の文字を先読みをし、それが現在の字句に追加されるべき文字か次の文字の先頭をなすものかなどの動作もする。

4.2　構文解析

構文解析とは、字句解析から受渡されるトークンの組み合わせを文法の意味にマッチさせ、結果として木構造に変化する処理を示す。ここで扱う文法は表 2 のようにする。一般に構文解析は木構造のルートから解析する下降型解析と、リーフから解析する上昇型解析の 2 種類がある。本研究では、下降型解析を用いて構文解析を行う。下降型解析は下向き構文解析ともいう。記号列を先頭から順に一度だけ見て行くことと下向きということから、解析は最左導出を順に作っていくことである。

この文法は「1 文字を先読みすれば、曖昧なく構文解析できる。」であった。そのために構文解析を行うためには「先読みトークン」が必要になる。構文解析処理の中では（先読みトークン、構文木）の組を受渡す。

この構文解析の処理の中で、トークンに関する情報を記号表に登録する。つまり、変数名と関数名にたいして登録と検索を行う。構文誤りの処理は本研究の中心ではないので、簡単化にした。構文誤りを見つけるとただちに解析を止め、誤りの回復とかは行わないようにした。

プログラム	::=	phrase_list					
phrase_list	::=	フレーズ					
	|	フレーズ	phrase_list				
フレーズ	::=	宣言文					
	|	式					
式	::=	式 4					
	|	LET	宣言文	IN	式	THEN	
	|	IF	式	THEN	式	ELSE	式
式 4	::=	式 3					
	|	式 4	,	式 3			
式 3	::=	式 2					
	|	式 3	=	式 2			
	|	式 3	<>& 式 2				
式 2	::=	式 1					
	|	式 2	+	式 1			
	|	式 2	-	式 1			
式 1	::=	式 0					
	|	式 1	*	式 0			
	|	式 1	/	式 0			
式 0	::=	基底式					
	|	式 0	基底式				
基底式	::=	整数					
	|	識別子					
	|	論理値					
	|	(	式	)			
	|	()					
宣言文	::=	VAL	式				
	|	FUN	式				

表 2: 文法の規定

4.3 記号表の管理と命令コードの生成

原始プログラム中に現れる識別子は、変数、関数を表している。個々の識別子は、型や有効範囲など、様々の情報を持っている。コンパイラはこれらの情報をプログラムから抽出して、記号表に保持する。変数名、関数名などは、原始プログラムにおいて何らかの対象を表している。これらはプログラムの宣言部において、型などの情報とともに宣言される。またこれらはプログラムの実行部において参照される。

記号表は識別子の種別に関わらず全体として一つにまとまっていることが普通である。しかし、本研究では変数名と関数名を別の表として実装する。記号表には変数名のみを入れ、関数名は別の表にする。変数名の記号表の宣言は次のように配列として実装する。

```
(*****変数名*****)
type binding = {count:int, level:int,
value:{tag:int,boundvalue:int} list}
type bucket = (string*binding) list
type table = bucket Array.array
val t : table = Array.array(SIZE,nil)
```

この中で、count は同じの名前の識別子を管理するのに使い、同じの名前の宣言が出る度に count 値を一増加する。参照の時は、count 値が一番高い (一番最近で宣言した名前を) 識別子の level に対応する値を参照する。level は局所宣言文を出会う度に、値を増加し、level 値番目の value リストを変更または参照する。局所宣言を出る度に、level の値をインクリメントする。tag は識別子が定義されているか否かの情報を記録している。

関数表も、単純な配列として実装している。配列の要素は主に、name,parameter, funcbody,idtable、四つのフィールドを持つレコードで実装する。funcbody は定義された関数を意味し、idtable は関数の定義の所で現れる識別子の記号表である。関数定義の中に global な変数がある時、トップレベルの表を参照する。ローカルならその関数ことの idtable に登録する。

命令コードの生成は構文解析から得た構文木と記号表を用いて、ハンドコンパイラしながら生成することにする。

5 緒言

この論文を通じて、分かるように、マルチスレッド型プロセッサと関数型言語を結合することにより、関数型言語の高効率が可能である。でも、現在単に理論的の確定に止まり、今後の課題としては実装によりその理論をもっと確認する必要がある。