INTERMEDIATE REPRESENTATIONS FOR PARALLEL LANGUAGES ON CGRAs

DIANYONG ZHANG

Intermediate Representations For Parallel Languages on CGRAs

by

© Dianyong Zhang

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of Engineering

Faculty of Engineering And Applied Science

Memorial University of Newfoundland

December 2007

St. John's Newfoundland and Labrador

Abstract

Coarse Grained Reconfigurable Arrays (CGRAs) are reconfigurable computing architectures based on word-width processing elements, in contrast to Field Programmable Gate Arrays (FPGAs), which use bit-width processing elements. CGRAs have faster time-to-market than Application Specific Integrated Circuits (ASICs) and, for many applications, faster execution than FPGA. In contrast to traditional sequential language, we aim to develop a parallel object-oriented programming language(HARPO/L), that can be compiled to CGRAs configuration files to execute. This requires that compiler exploits the parallelism from language itself and target architecture. This thesis mainly shows a set of intermediate representations (IR) in the compiler for HARPO/L, which contains all information from the source program and is easier to be analyzed and optimized.

Based on the HARPO/L language features, we use object dependence graphs (ODGs) to represent the relations among objects. The augmented concurrent control flow graphs show threads among objects. After the interobject analysis and optimizations, an executable dataflow graph, the final IR form, can be produced. The IR can then be used as input to the compiler back end.

This research offers a significant advance on the HARPO/L Compilation approach. It will benefit to finish advanced HARPO/L programming Language executing in high diversity CGRA architectures.

Acknowledgments

First of all, I would like to thank my supervisor, Professor Theodore S. Norvell, for his support and guidance through my graduate career in Memorial University of Newfoundland. This thesis would not have been as solid without his insight and guidance. I particularly appreciate his introducing me to the field of parallel processing and getting me started on compilation techniques for explicitly parallel programs. I have learned from him how to view the world from many different perspectives and how to make an idea concrete.

I am grateful also to my wife, Jinghua Nie. Her love and care encourage me to finish this thesis.

Finally, I dedicate this thesis to my parents, and I appreciate their endless love and support.

Contents

Abstract ii

Acknowledgments iii

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 CGRA Introduction 1

1.2 Compiler Introduction 4

1.2.1 Front End 5

1.2.2 IR Generation and Optimization 5

1.2.3 Back End 7

1.3 Introduction to HARPO/L 7

1.3.1 HARPO/L Specification 8

1.3.2 Example 11

1.4 The Outline Of The Thesis 13

1.5 Main Contributions in This Thesis 14

2 Related Work **16**

2.1 The Introduction of Intermediate Representation 16

2.2 Several Common IRs 17

2.2.1 Control Flow Graph (CFG) 17

2.2.2 Program Dependence Graph(PDG) 18

2.2.3 Parallel Program Graph(PPG) 22

2.2.4 Single Static Assignment(SSA) 24

2.2.4.1 Traditional Single Static Assignment(SSA) 24

2.2.4.2 Concurrent Single Static Assignment(CSSA) 26

2.2.5 Other IRs 27

2.3 Data Flow Graph and Call graph 29

3 Object Dependence Graph **31**

3.1 Some Issues on Object Dependence Graph(ODG) 31

3.2 ODG Specification 34

3.3 Algorithm for Constructing ODG 38

4 Augmented Concurrent Control Flow Graph **42**

4.1 Introduction 42

4.2 Memory Consistency Models 43

4.2.1 The Classes of Parallel Programs 43

4.2.2 Memory Consistency Models 44

4.2.3 Memory Consistency in HARPO/L 46

4.3 Augmented Concurrent Control Flow Graph (ACCFG) 47

4.4 Concurrent Single Static Assignment (CSSA) 50

4.5 Function Placement 54
4.5.1 Placing ϕ Function 55
4.5.2 Placing ψ Function 56
4.5.3 Placing π Function 57
4.5.4 Placing ξ Function 58
4.6 Summary 62

5 InterObject Analysis **63**
5.1 Introduction 63
5.2 Interobject Optimization 65
5.3 Deadlock Analysis and Solution 68

6 Optimization In Parallel Programs **71**
6.1 Introduction of Optimization 71
6.2 Common Subexpression Elimination(CSE) 73
6.2.1 Common Subexpression Elimination 73
6.2.2 A CSE Example in Sequential Programs 73
6.2.3 A CSE Example in Parallel Programs 74
6.2.4 A CSE Example in Parallel Programs with Synchronizations . 75
6.2.5 CSE Analysis in Parallel Programs 76
6.3 Dead Code Elimination 82
6.3.1 Dead Code Elimination 82
6.3.2 Dead Code Elimination in Sequential Programs 82
6.3.3 Dead Code Elimination in Parallel Programs 83

6.3.4 Dead Code Elimination in Parallel Programs with Synchronization . . . 84
6.3.5 Dead Code Elimination Analysis in Parallel Programs 85
6.4 Code Motion 86
6.4.1 Code Motion 86
6.4.2 Code Motion in Sequential Programs 86
6.4.3 Code Motion in Parallel Programs 87
6.4.4 Code Motion in Parallel Programs with Synchronization . . . 90
6.4.5 Code Motion Analysis in Parallel Programs 92
6.5 Loop Fusion and Loop Fission 95
6.5.1 Loop Fusion and Fission 95
6.5.2 Loop Fusion and Loop Fission in Sequential Programs 96
6.5.3 Loop Fusion and Loop Fission in Parallel Programs 98
6.5.4 Loop Fusion in Parallel Programs with Synchronization 100
6.5.5 Loop Fusion and Loop Fission analysis in Parallel Programs . 101
6.6 Atomic Fusion/Fission in Parallel Optimization 104
6.7 Summary 107

7 Conclusions and Future Work **108**

7.1 Conclusion 108
7.1.1 Thesis Summary 108
7.1.2 Thesis Contributions 110
7.2 Open Issues for Future Work 111

Bibliography **112**

List of Tables

1.1 Some CGRA Architectures 2

List of Figures

1.1 A structure of Function Unit [34] 3

1.2 A CGRA Mesh Architecture example 3

1.3 A HARPO/L Source Code 12

1.4 General Block Diagram 14

2.1 An Example of Control Flow Graph 19

2.2 Data Dependence and Control Dependence example 19

2.3 An Example of PDG 22

2.4 An Example of PPG 24

2.5 A Straightforward SSA example 25

2.6 A SSA example with joint node 25

2.7 An example on Dataflow Graph 30

3.1 a simple ODG example 37

3.2 The construction of ODG 41

4.1 The Parallel Program Classification of Vivek 44

4.2 An Example of ACCFG 50

4.3 An Example on the ϕ function 51

4.4 An Example on the ψ function 52
4.5 An Example on the π function 53
4.6 An Example on the ξ function 54
4.7 Calling Procedure in Parallel Program 60

5.1 Some Special Cases for Interprocedural Optimization 67
5.2 An Example on Calling Deadlock 69

6.1 Common Subexpression Elimination in sequential programs 73
6.2 CSE in parallel programs 74
6.3 CSE in Parallel Program with Synchronization 77
6.4 CSE in parallel programs 78
6.5 Atomic Fusion/Fission Analysis on CSE in parallel programs 80
6.6 The Introduction of Local Variable for The Fission of Atomic Operation in Parallel Programs CSE Optimization 81
6.7 Dead Code Elimination 83
6.8 Dead Code Elimination in parallel programs 83
6.9 Example 1 on Dead Code Elimination in parallel programs with synchronization 84
6.10 Example 2 on Dead Code Elimination in parallel programs with synchronization 85
6.11 Code Motion in Sequential Programs 87
6.12 Code Motion in Parallel Programs 88
6.13 Code Motion in Parallel Programs 89
6.14 Code Motion Out of Parallel Control Flow in Parallel Programs . . . 89

6.15 An Unsuccessful Code Motion in Parallel Programs [31] 91
6.16 A Successful Code Motion Example in Parallel Programs [31] 91
6.17 Code Motion With Fusion/Fission Analysis in Parallel Programs . . . 93
6.18 Code Motion of Parallel Control Flow With Fusion/Fission Analysis in Parallel Programs 94
6.19 Fusion/Fission Analysis of Code Motion With Synchronization in Parallel Programs 95
6.20 Loop Fusion and Loop Fission Example in Sequential Programs . . . 97
6.21 Loop Fusion in Sequential Programs 98
6.22 Loop Fusion and Loop Fission in parallel Programs 99
6.23 Loop Fusion and Loop Fission in Parallel Programs 100
6.24 Loop Fusion and Loop Fission with a Synchronization Statement in Parallel Programs [31] 101
6.25 Atomic Fusion/Fission Analysis for Loop Fusion and Loop Fission in Parallel Programs 103
6.26 An Example to Use The Theorem For Safe-Fission Indentification . . 107

Chapter 1

Introduction

1.1 CGRA Introduction

Coarse Grained Reconfigurable Arrays are reconfigurable computing architectures, which are based on word-width processing elements, in contrast to Field Programmable Gate Arrays (FPGA), which use bit-width processing elements. The coarse granularity greatly reduces delay, area, power consumption and configuration time, compared with FPGA, at the cost of loss of flexibility. CGRAs allow faster time-to-market than Application Specific Integrated Circuits (ASIC), and for many applications, faster execution than microprocessors. CGRA Architectures bridge the gap between ASIC's and microprocessors. Nowadays the rapidly evolving market of mobile and personal digital devices creates a higher demand for reconfigurable computing technology. Their functionality and adaptability lead to an increasing consideration of coarse grained reconfigurable architectures.

To present, many such architectures have been proposed [8].

In Table 1.1, we can find numerous differences in fabric architecture, granularity,

Table 1.1: Some CGRA Architectures

Structure	Architecture	Granularity	Fabrics
DP-FPGA	2-D Array	1 and 4 bit multigranular	routing channels
KressArray	2-D Mesh	selectable path	Multiple NN Bus segments
Colt	2-D Array	1 and 16 bit	sophisticated
Matrix	2-D Mesh	8 bit multigranular	8NN length4 global lines
RAW	2-D Mesh	8 bit multigranular	8 NN switched connections
Garp	2-D Mesh	2 bit	Global semi global lines
REMARC	2-D Mesh	16 bit	NN full length buses
Morphosys	2-D Mesh	16 bit	NN, length global lines
CHESS	Hexagon Mesh	4 bit multigranular	8 NN and Bus
DReAM	2-D Array	8 and 16 bit	NN, Segmented buses
CS2000	2-D Array	16 and 32 bit	inhomogenous Array
MECA	2-D Array	multigranular	not disclosed
CALISTO	2-D Array	16 bit multigranular	not disclosed
FIPSOC	2-D Array	4 bit multigranular	not disclosed
RAPID	1-D Array	16 bit	segmented buses
PipeRench	1-D Array	128 bit	not disclosed
PADDI	Crossbar	16 bit	Central Crossbar
PADDI-2	Crossbar	16 bit	multiple Crossbar
Pleiades	Mesh/Crossbar	multigranular	Multiple Segmented Crossbar

topology and mapping methods. Furthermore, most of the architectures are only laboratory research products, and only few are commercial products such as the eXtreme Processing Platform (XPP). Commonly, CGRAs consist of an array of function units (FU) or processor elements (PE). Each FU includes a certain number of Arithmetic Logical Units (ALUs) and registers.

Connection methods between FUs differ. Based on these differences, they can categorized as linear array, mesh and crossbar.

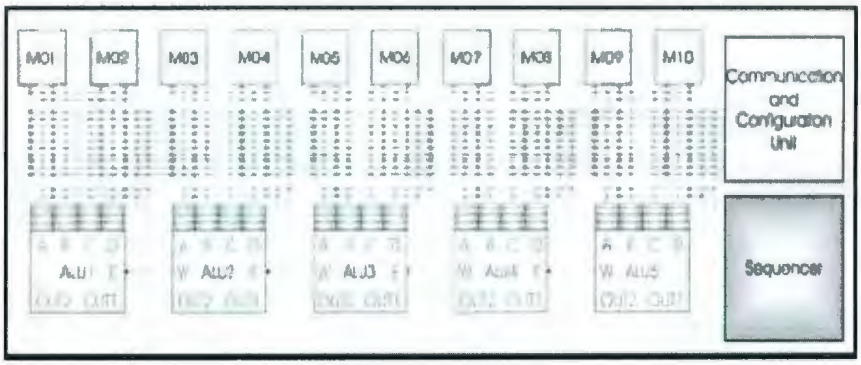


Figure 1.1: A structure of Function Unit [34]

Linear array architectures arrange FUs in one or several linear arrays, typically with nearest neighbor(NN) connection, with the aim of mapping pipelines onto it.

Mesh-based architectures arrange their FUs in a rectangular way with horizontal and vertical connections. Figure 1.2 shows the Chess architecture, which is one kind of mesh-based architecture.

Crossbar-based architectures connect their FUs with a full crossbar switch. This

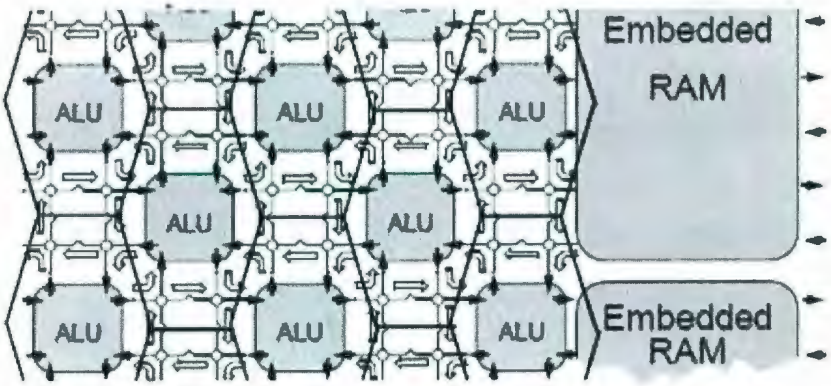


Figure 1.2: A CGRA Mesh Architecture example

is a most powerful communication network and provides more communication re-

sources.

These diversified structural features lead to big challenges for compiler designs because flexible available resources and communication imply more complicated representations for the resources and strategies of mapping, routing and scheduling.

1.2 Compiler Introduction

In this thesis, we mainly discuss a new intermediate representation (IRs) for compilers. So, it is necessary to introduce some compiler theory at first. A compiler is a program that translates text written in a computer language (the source language) into another computer language (the target language) [33]. Compilers can be one of these following categories:

- A source-to-source compiler transforms a high level language to another high level language.
- Stage compiler produces lower level language, such as assembly language or configuration files for a reconfigurable hardware platform.

Most modern compilers have three stages: front end, optimization and back end. Compilers use transformations of program representation in the front end to create intermediate representations for optimization. After optimization, the program representations are sent to the back end for scheduling, routing and mapping.

1.2.1 Front End

The front end analyzes source code in text to build an intermediate representation. Usually, it uses a symbol table. This is a data structure, mapping each symbol in the source code to associated information such as type and scope. The front end often has several phases. First, the lexical analyzer breaks the source code text into small atomic language units (tokens). Conditional compilation and macro substitution may happen in this stage. Next comes the parsing phase. This phase parses the token sequence to identify the syntactic structure of the program. Last, semantic analysis checks the program to ensure it obeys the rules of the language.

1.2.2 IR Generation and Optimization

Usually, IRs use graphs which represent the structure of programs. Compilers use IRs to do flow analysis and optimizations.

Typical analyses include variable define-use (DU) and use-define (UD) chains, and elsewhere dependence analysis, alias analysis, pointer analysis etc. The call graph and control flow graph are usually also built during the analysis phase. These two graphs are also IRs. In addition, a compiler including more than one Intermediate Representation is possible. In this thesis, we develope several IRs for representing and optimizing CGRA Language programs. IRs can be divided into high-level IR, middle-level IR and low-level IR. Higher level IR has less dependence on target architecture. Low-level IR is often constrained by target architectures. For CGRA language, target architecture diversities make us choose higher IRs. Furthermore, IRs in different levels can be transfered among each other. The standard for a good IR is to represent

complete source code information and facilitate optimization.

Facilitating optimization is the main goal of IR design. In some cases, source code transforming into an IR is only utilized for an efficient optimization. The purpose of optimizations is improving performance. In computer programming, optimization means to modify code and its compilation settings on a given computer architecture to produce more efficient software. The most common optimization goal is to reduce the time taken to execute a program. The less consideration goal is to reduce the amount of memory occupied, or the power consumed by a program. It has been shown that some code optimization problems are NP-complete.

Usual optimization techniques include the following themes: *avoiding redundancy, exploiting parallelism, decreasing jumping, increasing code locality, and avoiding memory accesses*. Sometimes these themes can conflict with one another.

We can categorize optimization techniques into loop optimizations, data-flow optimizations, SSA-based optimizations, and interprocedural optimizations. The operating objects of loop optimization are loop statements in source language. The usual optimizations are loop fission, loop fusion, loop inversion, loop unrolling etc. Data flow optimizations come from data flow analysis. They obtain optimization effects by analyzing how data propagate along control edges in the control flow graph. Common subexpression elimination, constant folding, and constant propagation are examples of data flow optimization. Static Single Assignment (SSA) is a common IR in which every variable is assigned in only one place. SSA-based optimizations include global value numbering and sparse conditional constant propagation. Interprocedural optimization works on the entire program. Typical interprocedural optimizations are procedure inlining, interprocedural dead code elimination, interprocedural constant

propagation, and procedure reordering. Usually, interprocedural analysis is needed before actual optimizations, these include interprocedural alias analysis, array access analysis, and construction of call graph [33, 20].

Traditional programing languages use sequential control flow. When parallel control flow is introduced, the original optimization algorithms may produce improper results. The HARPO/L is a programming language which includes parallel control flow and multithreads. To compiler HARPO/L we must extend and adapt the sequential optimization methods. We will give more intermediate representation and optimization details in later chapters.

1.2.3 Back End

The back end is the last stage in a compiler. The back end uses the IR to produce configuration files, which control data execution for the target architectures. Although some optimizations are finished in optimization stage, back end makes further optimization. But these optimizations are often associated with specified target architectures. Compilers must attempt to utilize all available hardware resources and structure features to optimize the code. Back-end tasks include routing, mapping and scheduling.

1.3 Introduction to HARPO/L

The HARPO/L was developed by Dr. Theodore S. Norvell at Memorial University of Newfoundland in 2006. The aim is to develop a language which can be applied to most CGRA platforms, as well as CPUs and FPGAs. Programmers can use it to

implement various user functions in software. Its compiler will finish hardware design and implementation. This will decrease the time to market and increase product flexibility.

In order to adapt to CGRA features, HARPO/L was designed to be an object-oriented language with parallel control flow. All following contents refers to [21].

1.3.1 HARPO/L Specification

$N \rightarrow E$	Nonterminal N can be an E
$\underline{(}E\underline{)}$	Grouping
E^*	Zero or more
E^{*F}	Zero or more separated by Fs
E^+	One or more
E^{+F}	One or more separated by Fs
$E^?$	Zero or one
$\underline{[}E\underline{]}$	Zero or one
$E \mid F$	Choice

Programs in the HARPO/L consist of sets of classes, interfaces and objects.

Interfaces, primitive types, arrays, and classes are all types. Primitive types include boolean, integer and real. Integer and real types have 8 bits, 16 bits, 32 bits and 64 bits corresponding to CGRA architecture granularity.

Classes define a user type. Classes may be generic or nongeneric. A generic class can have one or more generic parameters.

$$\mathit{ClassDecl} \rightarrow (\textbf{class}\ \mathit{Name}\ \mathit{GParams}^?\ (\textbf{implements}\ \mathit{Type}^*)^?\ (\mathit{ClassMember})^*$$

[**class**])

In the above grammer rule, the *name* is the name of the class. The *types* are the interfaces, which the class implements. The *ClassMembers* can be fields, methods and threads. Specifically, fields are objects that are within objects. Field declarations define the part-whole hierarchy. We will use the information to identify the relation of objects in HARPO/L programs. Field declarations have the following form:

$$Field \rightarrow Access\ \textbf{obj}\ Name{:}Type\ [:= InitExp]$$

In this expression, *Access* can be private or public. If an object, which is defined in this field declaration, is private, it is only accessible in objects, whose type is this class. If an object is public, it can be used by other objects of other classes. Public property can cause shared-variable-conflict-use problem, although private fields can still be shared by multiple threads in same object. This will be discussed in a future chapter.

A method declaration declares a method, but not its implementation. The implementation body is located within a thread. It has the following form:

$$Method \rightarrow Access\ \textbf{proc}\ Name((Direction\ [Name :]\ Type)^*)]$$

In this rule, access has same meaning as for fields. The *Direction* can be in or out. The "in" means input parameter of this procedure and "out" means it is an output parameter.

A thread declaration defines a thread. In an object, whose type is not a primitive type, there may be zero, one, or more than one thread; otherwise, in an object, whose type is primitive type, there are no any threads. For those objects in which there are

more than one threads programmers ensure the threads coordination.

As there is no dynamic allocation, all objects are created at compile time. We can correctly conclude programs consist of a number of simple objects and complex objects; and we can analyze their relations at compile time.

Each thread contains a block. A block is a sequence of statements. In HARPO/L, there are assignment statements, local variable declarations, method call statements, sequential control flow, parallel control flow, and method implementation.

A method call statement causes a calling of a procedure. Its form is:

$$\textit{Statement} \rightarrow \textit{ObjectId.Name}\ (\textit{Args}) \mid \text{Name}(\textit{Args})$$

The *ObjectId* is the name of object called. The *Name* is the name of a procedure. The *Args* is the argument list, which is sent to the called procedure.

On the other hand, called procedures have the following form:

Statement → (**accept** *MethodImp* (| *MethodImp*)* [**accept**])

MethodImp → *Name* ((*Direction Name* : *Type*)*) [*Guard*] $Block_0$ [**then** $Block_1$]

Guard → **when** *Expression*

When a thread reaches an accept statement, it must wait until there is a call to one of the implemented methods and the corresponding guard is true. Once there is one method call, the accept can execute; the input parameters are passed in, and the $Block_0$ is executed. The output parameters are copied back to the calling thread. The method call statement and method implementation statement are calling and called relation. They guarantee only one calling thread execute the called procedure at any time.

In HARPO/L, sequential control flow has *if* statement (conditional), *while* and

for(loop) as usual. Their definitions are same as the traditional sequential programs. In HARPO/L, parallel control flow includes two forms.

Statement → (**co** Block (|| Block)* [**co**]) (1)

Statement → (**co** Name : Bounds Block [**co**]) (2)

We observe that expression (1) has the symbol **co** at its beginning and end, one or more blocks are seperated by the symbol ||. This represents that several threads can execute at the same time. This is a cobegin-coend structure. In expression (2), all iterations can be executed in parallel. This is a parallel do loop. We call it pardo loop. These parallel control flow and multithreads in objects make HARPO/L programs explicitly parallel.

1.3.2 Example

We can see a HARPO/L example in the following.

```
(class FIFO [in capacity : int, type T extends primitive]
     public proc deposit(in value : T)
     public proc fetch(out value : T)
     private obj a : T(capacity)
     private obj front := 0
     private obj size := 0
     (thread
       (wh true
         (accept
```

```
            deposit( in value : T ) when size < capacity
                a( (front + size) % capacity ) := value
                size := size + 1
        |   fetch( out value : T ) when size > 0
            value := a(front)
            front := (front + 1) % capacity
            size := size - 1
        accept)
    wh)
  thread)
class)
```

Figure 1.3: A HARPO/L Source Code

In this example, we define a class FIFO. The *capacity* is its parameter. It represent the length of FIFO. Type T is the subclass of type primitive. In it, there are two methods "deposit" and "fetch". When other objects use objects of FIFO type, values will be sent through the deposit or fetch procedures. When size is less than capacity, a value can be deposited at the end. When size is larger than zero, a value can be fetched from the beginning. This class simulates a procedure of FIFO.

1.4 The Outline Of The Thesis

In order to satisfy the HARPO/L features, we develop a set of Intermediate Representations for HARPO/L compiler. Please refer to Figure 1.4. The source code of HAROP/L is parsed by the HAROP/L parser. Abtract syntax tree can be produced. In the object dependence graph, we can obtain the relation of object. For Objects, we know their control flow relations by augmented concurrent control flow graph. After interprocedural analysis, some optimizations can be made. Concurrent data flow graph can be produced for the backend of compiler. The final configuration files can be created for CGRA hardware platform.

The rest of this thesis is organized as following:

Chapter 2 introduces other researchers' related work on CGRA architecture and compiler techniques and approaches.

Chapters 3 to 7 discuss the HARPO/L complier intermediate representations.

Chapter 3 defines the Object Dependence Graph specification and algorithms on it.

Chapter 4 explains this Augmented Concurrent Control Flow Graph specification and implementation algorithm.

Chapter 5 discusses interobject analysis.

Chapter 6 gives several optimizations in parallel programs.

Chapter 7 describes data flow form,the final intermediate representation form.

Finally, Chapter 8 contains conclusion and a discussion of future work.

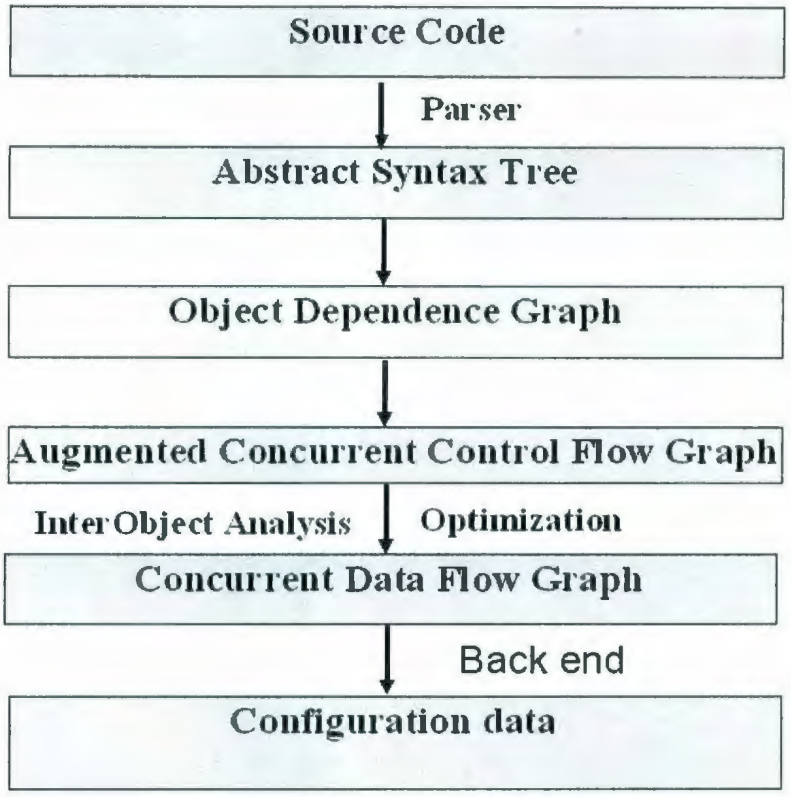


Figure 1.4: General Block Diagram

1.5 Main Contributions in This Thesis

This thesis concentrates on control and data flow analysis, program representation, and parallel and sequential optimizations for a compiler for a new programming language, **HARPO/L** (Hardware Parallel Objects Language), targetted to the Coarse Grain Reconfigurable Array(CGRA) architectures.

The main contributions are given as follows:

- We develop a set of Intermediate Representations and corresponding algorithms

for the compiler, which is used for HARPO/L [21].

- We define Object Dependence Graphs (ODG), which collect all object information in HARPO/L programs, and include relations among these objects. Related definitions and algorithms will be given.

- We extend Concurrent Control Flow Graphs (CCFG) [9] to Augmented CCFGs (ACCFGs). ACCFGs can represent control flow relations within objects in HARPO/L programs.

- We explain several optimizations including Common Subexpression Elimination, Dead Code Elimination, Hoistable Access and Loop Fusion in parallel programs, and how they differ from the corresponding optimizations in sequential programs.

- In order to transform ACCFG to executable dataflow form, we provide possible solutions to convert ϕ, π ,ψ and ξ functions, which appear in the ACCFG, to dataflow form. These functions are used to solve various different multithread cases in parallel programs. This executable dataflow form is used as input to the compiler back end.

Chapter 2

Related Work

2.1 The Introduction of Intermediate Representation

The compiler design for a new programming language is a complicated engineering problem. In this procedure, Intermediate Representation (IR) plays an important role. A good IR can improve program efficiency and decrease execution time of target configuration files. The standard for judging a good IR is its ability to represent all information from the source code and to be easily optimized. The optimized IR will be sent to the back end for mapping, routing and scheduling.

IR design is largely an art, not a science. We must consider whether to use an existing representation. If an existing intermediate language is not used, there are many decisions to be made in the design of the new one. If an existing one is to be used, there are considerations to be acknowledged of its appropriateness for both the language to be compiled and target architecture, possible resulting costs, and the

saving in reuse of an existing design and code. There is also an issue of whether the intermediate form is appropriate for certain optimization to be performed. Some optimizations may be too hard to do at all on a given intermediate representation, and some may take much longer to do than they would on another representation.

When we design a new IR, the issues considered should include: what is its level, organizational structure and expressiveness; whether it is appropriate for many optimizations or a certain particular optimization; whether it is apt for the generation of configuration file for the target architectures. In addition, it is possible to use more than one IR in one compiler. This requires translation of one IR to another in the compilation process. Each IR may be appropriate for one particular task only.

Researchers have made many efforts on Intermediate Representation. Some of the well-known are CFG, PDG, PPG and SSA. We will give brief introductions to each of them. This is helpful in order to get some ideas and inspirations from them.

2.2 Several Common IRs

2.2.1 Control Flow Graph (CFG)

Control Flow Graphs (CFG) have been the usual representation for control flow relationships of a program, and is widely used for many compiler optimizations and static analysis tools. In [2] and [12], the CFG definition was given as follows:

Definition 2.1 A control flow graph $CFG = (N, Ecf, TYPE)$ is a directed multigraph.

- N is a set of nodes. The CFG node represents an arbitrary sequential computation,

such as a basic block, a statement or an operation.

- $Ecf \subset N \times N \times \{ T, F, U \}$, a set of labeled control flow edges. For control edge, there are conditional (F and T)and unconditional edges(U).

- $TYPE : N \rightarrow \{ START, STOP, PREDICATE, COMPUTE \}$, a node type mapping.

The notation represents all paths that might be traversed through a program during its execution. In CFG, $START$ and $STOP$ are two distinguished nodes: the $START$ node, along which control enters into CFG, and the $STOP$ node, along which all control flow leaves. $START$ has no incoming edges and one outgoing unconditional edge. $STOP$ has no outgoing edges and incoming edges.

The $PREDICATE$ nodes represent a conditional branch, which has two outgoing edges "T" ($true$) and "F" ($false$) labels.

The $COMPUTE$ and $START$ nodes have exactly one outgoing edge with label "U" ($unconditional$).□

Traditional CFG is a sequential representation of a program without parallel structure. We can see an example of a CFG in Figure 2.1.

2.2.2 Program Dependence Graph(PDG)

J.Ferrabte and K.J Ottenstein [12] proposed Program Dependence Graphs (PDG) in 1987. PDG has two dependence relations: Control Dependence and Data Dependence.

Data dependence was proposed by Ottenstein in 1978 [23]. A data dependence exists between two statements, whenever a variable appearing in one statement may

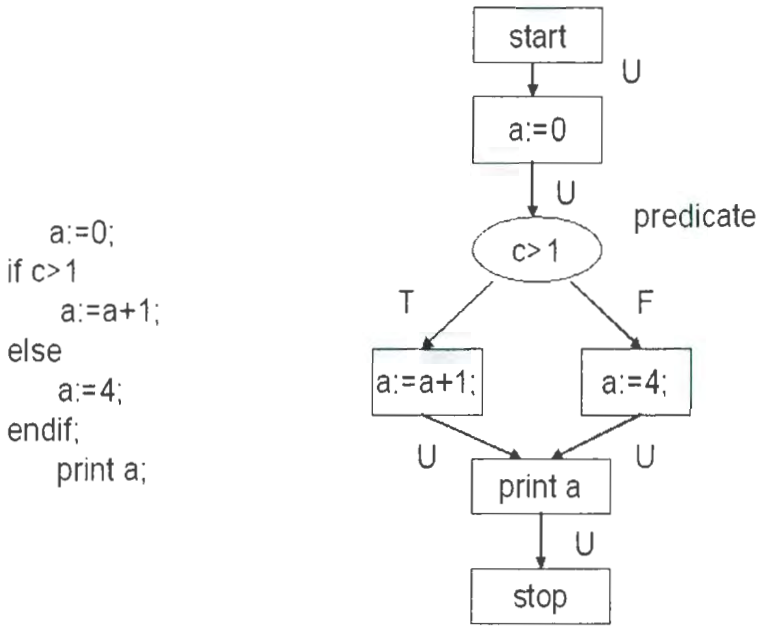


Figure 2.1: An Example of Control Flow Graph

have an incorrect value if these two statements are reversed. A control dependence exists between a statement and the predicate whose value immediately controls the execution of the statement.

Data dependences are only used to represent the relevant data flow relationships

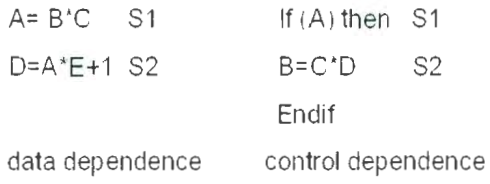


Figure 2.2: Data Dependence and Control Dependence example

of a program. Control Dependences represent only the essential control flow relationships of a program. Control dependence graphs are derived from the usual control

flow graphs and dominator trees. These two dependence relations are shown in the Figure 2.2. In the example of data dependece, if we reverse S1 and S2, we will get a different value of variable D. On the other hand, only when the condition of S1 is satisfied, S2 can be executed. This is a control dependence.

In a PDG, the program is represented as a graph, in which the nodes usually represent statements and predicate expressions. The edges connecting nodes represent both the data values on which the node's operations depend, and the control conditions on which the execution of the operations depend. The set of all dependences for a program may be viewed as inducing a partial ordering on the statements and predicates in the program that must be followed to preserve the semantics of the original program.

Definition 2.2 A Program Dependence Graph PDG=(N, E_{cd}, E_{dd}, TYPE) is a directed multigraph in which every node is reachable from the root. [12]

- N is a set of nodes. Nodes usually representing statements and predicates.

- $E_{cd} \subseteq N \times N \times \{ T, F, U \}$, a set of labeled control dependence edges.

An edge $(a, b) \in E_{cd}$ identifies a control dependence. Node a must have type $REGION$ or $PREDICATE$, if a has type $PREDICATE$, then (a, b) must be labeled "T" or "F" and b can only be executed if the predicate evaluates to true, in the case of T, or false, in the case of F. If a has type $REGION$, (a, b) must be labeled U, and b can only be executed after a.

- $E_{dd} \subseteq N \times N \times \{ LI, LC \}$, is a set of data dependence edges.

An edge $(a, b) \in E_{cd}$ identifies a data dependence from node a to node b, which must be synchronized. LI and LC can be identified as *loop-independent* and *loop-carried*.

- $TYPE : N \rightarrow \{ START, PREDICATE, COMPUTE, REGION \}$, a node type mapping.

$START$ and $PREDICATE$ node types are similar to the counterpart in a CFG. The $COMPUTE$ node has no outgoing control edges in a PDG. The $REGION$ node serves as a summary node for a set of control dependence successors in a PDG. □

Data dependence can explain that two statements access or modify the same resource. There are three data dependence relations here.

- Flow dependence: A statement S2 is flow dependent on S1 if and only if S1 modifies a resource that S2 reads, and S1 precedes S2 in execution.

S1 x := 10

S2 y := x + c

- Anti-dependence: A statement S2 is anti-dependent on S1 if and only if S2 modifies a resource that S1 reads, and S1 precedes S2 in execution.

S1 x := y + c

S2 y := 10

- Output dependence: A statement S2 is output dependent on S1 if and only if S1 and S2 modify the same resource, and S1 precedes S2 in execution.

S1 x := 10

S2 x := 20

We can see an example of a PDG in Figure 2.3. The statements (1), (2), and (4) are connected the node region. But statement (3) can happen only when the

statement (2) is true. In the statement (4), the value of variable a can come from (1),(2) and (3). So there are data dependence relations there.

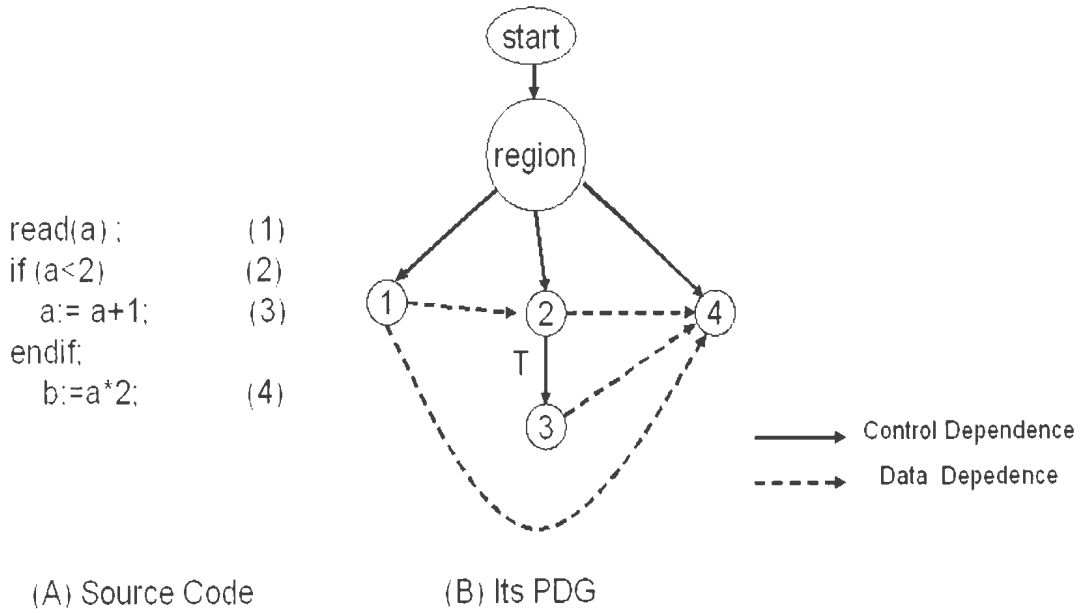


Figure 2.3: An Example of PDG

2.2.3 Parallel Program Graph(PPG)

Vivek Sarkar and Barbara Simons proposed Parallel Program Graphs (PPG) in 1993 [32]. PPG is actually a compound of CFG and PDG. The CFG is a sequential representation lacking of dependences, and PDG is a parallel representation of sequential programs. When putting them into one PPG graph, it is a parallel representation of sequential and parallel program as of parallel control flow edges and synchronization edges. PPGs contain control edges that represent parallel flow of control, and synchronization edges impose ordering constraints on executing PPG nodes. MGOTO nodes are used to create new threads of parallel control. The PPG

is a possible intermediate representation for explicit parallel programs.

Definition 2.3 A Parallel Program Graph $PPG = (N, Econt, Esync, TYPE)$ is a rooted directed multigraph in which every node is reachable from the root using only control edges.

-N is a set of nodes.

-$Econt \subset N \times N \times \{ T, F, U \}$, is a set of control edges

-$Esync \subseteq N \times N \times SynchronizationCondition$, is a set of synchronization edges. This edge is decided by a synchronization condition.

-$TYPE : N \rightarrow \{ START, PREDICATE, COMPUTE, MGOTO \}$, a node type mapping.

The $START$ node and $PREDICATE$ node are similar to their counterpart in a CFG or a PDG. The $COMPUTE$ node may have outgoing control edge or no one. The node $MGOTO$ is used as a construct for creating parallel threads of computation. A new thread is created for each successor of a $MGOTO$ node. □

PPGs have been shown to be useful for solving various problems, including optimization, vectorization, code generation for VLIW machines, and merging versions of programs.

We can see an example in Figure 2.4. Since the node $MGOTO$ create parallel thread to compute, after variable a is read, (3), (4), and (5) can be executed in parallel. But they use the same value of variable a. So the synchronization edges are used.

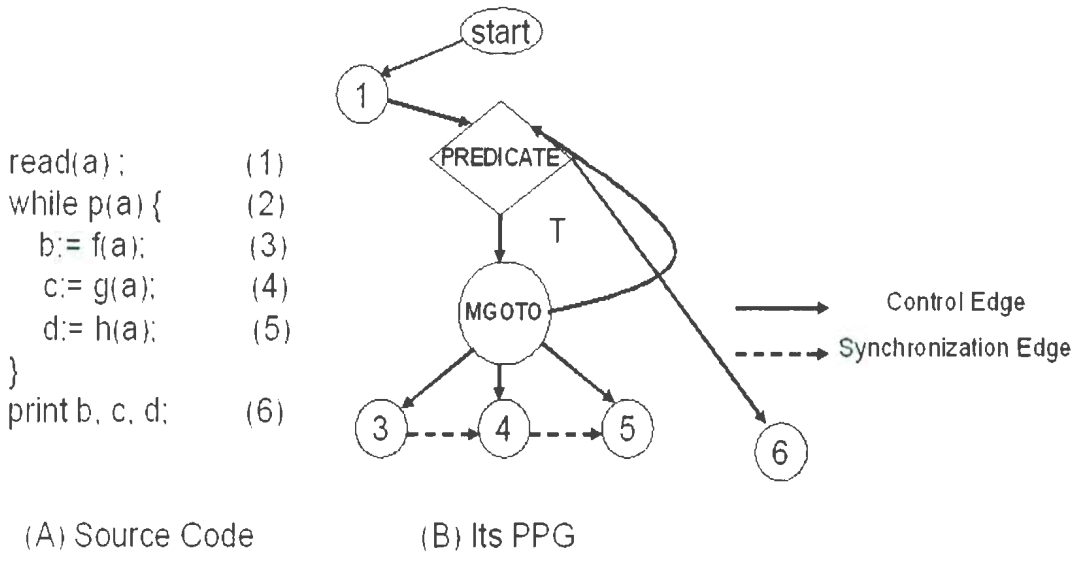


Figure 2.4: An Example of PPG

2.2.4 Single Static Assignment(SSA)

2.2.4.1 Traditional Single Static Assignment(SSA)

R.Cytron, J. Ferrante and B. Rosen, working at IBM, developed Single Static Assignment(SSA) in the 1980s [33].

As its name suggests, SSA only reflects static properties. In SSA, every variable is assigned only once. Existing variables in the original IR are split into many versions. These versions typically are created by the original name with a subscript. Every new assignment is assigned to one of versions of a variable. A *use* of a variable with a particular definition means the *definition* and *use* have exactly the same name in the SSA form. This simplifies and makes more effective several kinds of optimizing transformations, including constant propagation, value numbering, invariant code motion, strength reduction, and partial redundancy elimination.

Figure 2.5 shows a simple instance of an SSA. In this example, the left is the original form and the right is the SSA form. The variable a_1 uses the definition of a_0 in the first statement. The variable a_0 has the same value in both statements. This is an example of straight code for SSA form.

However, there is a problem when we meet a joint point, where two or more

```
a:=0;        a0=0
a:=a+1;      a1=a0+1
```

Figure 2.5: A Straightforward SSA example

control flow paths merge. Multiple definitions of a variable may reach the joint point. This will result in a violation of single assignment property. We can see an example in Figure 2.6.

In this example at control flow merge join node, compiler must decide which

```
    a:=0;                    a0:=0;
if c>1                   if c0>1
    a=a+1;                   a1=a0+1;
else                     else
    a=4;                     a2=4;
endif;                   endif;
    print a;             a3=Φ(a1,a2)
                         print a3;

(a) Source Code          (b) SSA Form
```

Figure 2.6: A SSA example with joint node

variable a_1 or a_2 can be chosen. In order to achieve this purpose, SSA introduces ϕ function. a_3 is assigned either a_1 or a_2, depending on control flow. A ϕ function has one argument for each incoming control path; the k^{th} argument to a ϕ function is the

incoming value along the k^{th} path. The value of ϕ function is one of the arguments and the selection depends on the control flow path followed by the program. ϕ functions are always inserted at the beginning of a basic block, and are considered to be executed simultaneously before execution of any other code in the block. The advantages of SSA are the following:

- The *definition* of a variable dominates its *use*. Some optimization algorithm will be more efficient by taking advantage of this property. It also simplifies the analysis, transformation, and optimization of IR.
- SSA chains are simpler to store and update than use-def chains.
- Since the unrelated uses of the same variable in the original program become different variables in the SSA form, this eliminates false dependencies.

2.2.4.2 Concurrent Single Static Assignment(CSSA)

SSA is a good Intermediate Representation, and it had been widely applied in many compilers. However, it has a drawback. It cannot solve multithread accessing shared variable which is an ordinary case in parallel programs [9, 10]. With silicon technology's development, the number of processors in hardware platform is increased. Parallel programming languages are becoming popular. Processors may access a shared variable concurrently without any fixed ordering of accesses. This leads to data races and nondeterministic behavior. Classical SSA cannot account for updates to shared variables in threads.

A new SSA form used for parallel programs, Concurrent Single Static Assignment(CSSA), has been proposed by Lee, Padua and Midkiff in [9, 10].

In CSSA, control flow joins still use ϕ function representation which is the same as the SSA ϕ function. As multithreads are introduced, CSSA uses ψ and π functions for multithread confluence cases. ψ function mainly solves threads in parallel control flow confluence joint such as coend and enddo. π function is for the use of shared variable with conflict edges in different threads. In this thesis, we make an extension for CSSA, so more detail will be given in a later chapter.

The CSSA form has the following properties [9, 10]:

- All uses of a variable are reached by exactly one static assignment to the variable.
- For a variable, the definition dominates the *use* if they are not arguments of ϕ, or ψ or π functions.

2.2.5 Other IRs

Besides those above, some IRs were also proposed based on different considerations. We briefly introduce them as follows.

- Dependence Flow Graph

  K.Pingali, M. Beck and R. Johnson in [11] proposed Dependence Flow Graph. Dependence Flow Graphs are a synthesis of ideas from data dependence graphs and the dataflow model of computation. Similar to data dependence graph, the dependence flow graph can be viewed as a data structure in which edges represent dependencies between operations. For each edge in the data dependence graph, there is a corresponding path in the dependence flow graph. The

differences are that dependence flow graph is executable. It is a generalization of the data driven execution semantics of dataflow graph.

- System Dependence Graph(SDG)

  S.Horwitz, T.Reps and D.Binkley proposed System Dependence Graph [30]. N.Walkinshaw and M. Wood [22] extended it for Java. System Dependence Graph extends previous dependence representations to incorporate collections of procedures (with procedure calls), rather than just monolithic programs. It is a multigraph. It maps control and data dependencies between the statements. Statements are categorized according to whether they contribute to the structure of a program. Each category is represented differently on the graph. When these different graphs are combined, they provide a graph program representation. System dependence graph is difficult to visualize in a graph, because it is composed of a large number of different types of nodes and edges.

- Multithread Dependence Graph

  Zhao has extended system dependence graphs for the multithread case [35]. Besides the traditional dependence relation, the author added synchronization and communication dependence relations. Synchronization dependence relation is categorized into two relations. One is wait-notify relation, and the other is stop-join relation. The previous is suitable to wait() and notify() or notifyall() methods. The latter is a thread calling join() method of another thread which may proceed only after this target thread terminates. In these two cases, their dependence edges are put into multithread dependence graph.

- Value Dependence Graph(VDG)

D.Weise and R.F. Crew proposed Value Dependence Graphs (VDG) [5] in 1994. A VDG is a graph representation that accurately captures dependencies in a program, without being tied to the original shape of the program. Nodes are the individual operators in the program, and edges represent how the operands of one operator are dependent on the outputs of other operators. By breaking down the statements into single operations, the original statement structure of the program is lost. By giving up information about the shape of the program, the VDG can represent the program's behavior very accurately and concisely.

- Program Dependence Web(PDW)

  Program Dependence Web is an extension of PDG and SSA. It was proposed by R.A. Ballance,A.B. Maccabe and K.J.Ottenstein in [26] in 1990. In PDW, the source program first is converted into PDG in SSA form. But the difference is that the gate function replace ϕ function in PDW. So a PDW includes dependence information between statements and the gate function for control flow join case. The value of the Gate function is decided with control conditions analysis. Later, gate functions are added to control the flow values into control regions in which those values are used. Finally, the dataflow form and target architecture constraints are separately produced for mapping and scheduling stages.

2.3 Data Flow Graph and Call graph

In the previous section, we mentioned the data flow graph. It is a graphical representation of the flow of data through an information system. It can reflect the

execution order of operations and their data dependencies. With a dataflow graph, users are able to visualize how the system will operate, what the system will accomplish and how the system will be implemented. If we take dataflow graph as final IR stage, the workload in mapping and scheduling can be decreased. We can see an example in Figure 2.7.

Call graph [33] is a directed graph that reflects calling relations among subrou-

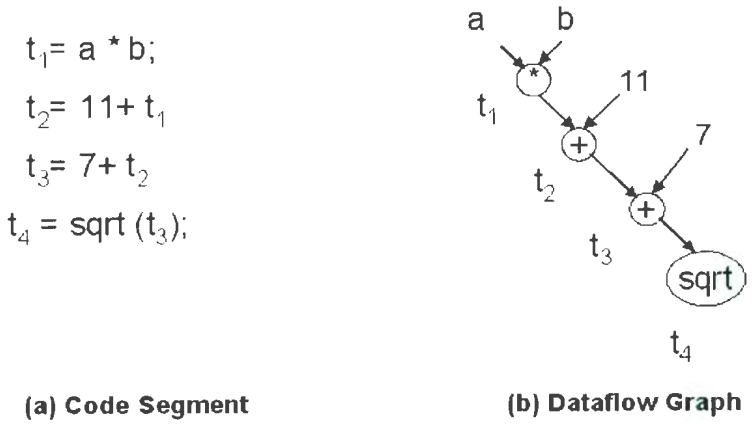


Figure 2.7: An example on Dataflow Graph

tines in a program. It shows the control flow of a program and it can be determined partially using a static analysis. However, sometimes some subroutines are decided or created at run time. In these cases, call graphs from static analysis in compile time cannot reflect call relations. Call graphs are distinguished between context-insensitive and context-sensitive kinds. For context-insensitive graphs, each procedure is represented as a node, and the arrows are all the possible calls between the nodes. For context-sensitive graphs, the parameters of the function call are considered. The call graph of a program that does not use recursion is a directed acyclic graph.

Chapter 3

Object Dependence Graph

In this chapter, we will introduce a new Intermediate Representation, Object Dependence Graph (ODG), for the HARPO/L Language Compiler. First, we will explain its feasibility and necessity. Then we will give the definition of ODG, which can represent relations between objects in HARPO/L language programs. Furthermore, we will present an algorithm for constructing the ODG.

3.1 Some Issues on Object Dependence Graph(ODG)

In this thesis, the main issue is Intermediate Representation design of compilers for HARPO/L language. First, some preprocess of source codes is made in front end including *Lexing, Parsing and Abstract Syntax Tree analysis*. After that, source codes should be correct and have no spelling, syntax and semantic errors. But they are probably low in efficiency and need to be processed by compiler to remove redundant and dead codes. This needs choosing or designing of some appropriate intermediate representation forms to optimize it.

In order to achieve this purpose, we proposed object dependence graph (ODG). In chapter one, we introduced the HARPO/L Language specification. As we know, the HARPO/L Language is an object-oriented programming language similar to Java and C++. All program codes in HARPO/L language consist of sets of interacting objects, each of which can receive messages, process data, and send messages. We can call some of them Complex Object (*CO*), and call the other Simple Object (*SO*).

Definition 3.1 Complex Object (CO)

A Complex Object is an instance of class with threads in HARPO/L Language. Only in complex object there exist threads. Each complex objects has one or more threads. Complex object can include other complex objects and simple objects. □

Definition 3.2 Simple Object (SO)

A Simple Object is an instance of primitive type or an instance of class including no threads in HARPO/L Language. Primitive type has Integer, Real and Boolean types. Simple Object can represent numbers. There is no thread in Simple Object. Simple Object cannot include other complex object. □

The definition of Simple Object in HARPO/L Language is similar to the definition of variable in conventional sense. Although there are many SOs in ODG, main subject in parallel programs is CO. That is because threads and parallel control flow are main elements in HARPO/L programs. In addition, we should indicate HARPO/L Language is a kind of 'static' language. This means all objects in HARPO/L language are produced in compile time.

In computer science, compile time refers to either the operations performed by a compiler or the programming language requirements that must be met by source code for it to be successfully compiled. The operations performed at compile time usually

include syntax analysis, various kinds of semantic analysis and code generation [33].

The opposite is runtime. In computer science, runtime describes the operation of a computer program during its execution [33].

In most programming languages such as C, PASCAL or FORTRAN, the compile time and runtime are obviously distinguished and cannot be interchanged. Generally, source codes are compiled (either machine code or an IR) first, and then the compiled code is run. No program execution is possible till the whole program is compiled. Once program execution begins, no more code can be added. In this case, the user can communicate with the running program system, only when the program contains a procedure, which reads text typed or mouse action etc, and performs their actions.

In HARPO/L Language, we also use this model. All objects used in the program are produced in compile time. And this is important for the construction of ODG, because this makes it possible for static object analysis in compile time and the construction of an object relation graph. Otherwise, if objects are created at runtime, we will not be able to use object dependence graph as an intermediate representation for HARPO/L language.

In addition, practical programs can include a huge number of objects and classes. If we add more analysis procedures and representations, they can cause the IR hard to be expressed in a single form. Data and control dependence analysis is still difficult to visualize into graphs such as in system dependence graphs (SDG) for Java program (introduced in Chapter 2).

For the similar reason, object analysis isn't used. However, in the HARPO/L language we can use the object relation based on the following reasons. First, HARPO/L

language has fewer types than usual programming languages. The HARPO/L data type includes Int(8, 16, 32 or 64 bits),Real(8, 16, 32 or 64 bits), Boolean, class types , interface , arrays and Generic types. Second, there are fewer relations among classes. The HARPO/L includes inheritance relation between classes. But there are no their polymorphism relations. These make possible to finish the object relation analysis. Because the number of objects are finite, we implement program analysis and optimizations in compile time, though it can last a few minutes.

3.2 ODG Specification

Definition 3.3 Object Dependence Graph (ODG)

A Object Dependence Graph $ODG = (N, E, Type)$ is a directed graph, in which each node is connected to others with relation edges.

1. N is a set of all objects in the program. Objects include COs and SOs.
2. $E \subseteq N \times N$, is a set of relation between objects. Object can connect to another object with one relation. These relations are represented in edges.
3. $Type : E \rightarrow \{ Knows, Partof \}$, is edge type. □

In the set, all objects have their own names. The name consists of two parts: its location and object name in class declaration. For example, if there are two objects whose names are 'object a', but their host objects are different: one is object b, the other is c. We classify these two objects with $b.a$ and $c.a$. They are two different

objects in programs.

Arrays is a data structure consisting of a group of elements having a single name that are accessed by indexing. Each element in an array has the same data type. Usually elements in this array occupy contiguous area of storage. The difference of elements between single-dimensional or multi-dimensional array is using different indexes to access elements in arrays. The different elements in arrays are different objects. We can identify them with array name and their index numbers such as a[0]. This is an object, whose name is a[0].

We also take care of global objects in programs. We can identify these global objects since their locations are outside class range. We can also identify them by their names. These global objects can be used by other objects (other objects have *know* relation for them); They are not part-of other objects (other objects have no *part-of* relation for them).

There are two types of edges. One is *part-of*, and the other is *knows*. *Part-of* relation means an object can be a part of another object. In informal terms, small object is a part of big object. A CO can include SOs and COs, but SOs cannot include other COs and SOs. The *know* relation means one object may use another object.

These edges are directed. The rules on direction are: If edge is *part-of* type, the edge direction is from the object included by another object; If the edge is *know* type, the edge direction is from object that is using the object.

Actually, for *know* relations we can identify it by a straightforward way. If an object 'A' has *know* relation with another object 'B', that means object 'A' calls a procedure in object 'B'. Furthermore, if object 'A' has *know* relation with object 'B',

they have no *part-of* relation at that same time.

Besides the above, we can see these following cases in Figure 3.1.:

1. If CO1 has *part of* relation with CO2, CO2 is a decendent-of CO1. If CO3 is a part of CO2, CO3 is also indirectly a decendent-of CO1. But we do not need to connect a part-of edge with them. We can say *part-of* relation has transitivity. But it is not necessary to connect CO1 and CO3 with *part-of* edge.

2. If CO2 is a part of CO1, it is impossible that CO1 is a part of CO2.

3. If CO2 is a part of CO1, it is impossible that CO1 has *know* relation with CO2.

4. If CO1 knows CO2 and CO2 knows CO1, this may produce a deadlock. A special analysis is necessary. We will discuss this in Chapter 5.

5. If both CO1 and CO2 know an object CO3, in which there is at least one procedure. This can be a call procedure. CO1 and CO2 race for access and use right of CO3. For calling and called problems, we make a further discuss in Chapter 4.

6. If CO2 is a part of CO1, CO3 known by CO2, we cannot conclude the CO3 known by CO1.

We conclude some ODG properties as follows:

- ODG can reflect object relation in programs. When we construct CFG, we need these relations to ensure the connection among different objects.

  ODG reflects the relations of all objects in programs, but this is only the first step for HARPO/L IR analysis. For the next step, we will analyze control flow

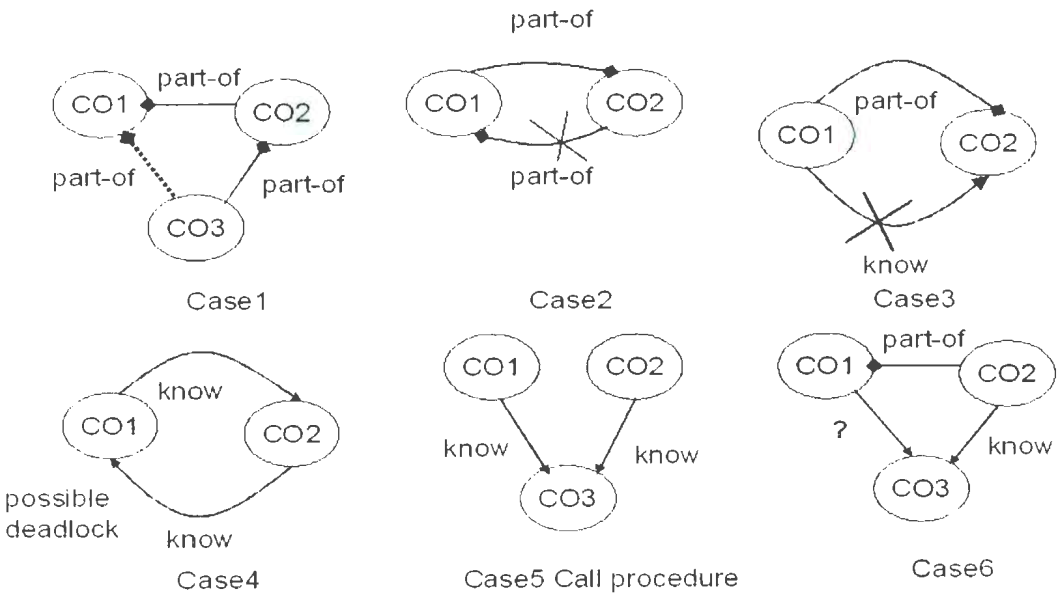


Figure 3.1: a simple ODG example

relation in objects. Obviously, it is realistic that we separately analyze each object and combine them together. But the connection relation of different objects makes it hard to connect them together. The ODGs provide the help to implement this procedure. We can find all object items of the HARPO/L programs, and connect them with *part-of* and *know* relations. The know relation comes from calling procedure and shared variables used using in a CFG. We can analyze these skeleton COs and connect them with conflict edges and calling relation edges. Each CO like this can have its own parallel control flow, which can include multiple threads. We can represent and analyze these COs in Augmented Concurrent Control Flow Graph, which will be introduced in the next chapter.

- ODG reveals calling relation and global shared variables.

  The traditional interprocedural method uses call graph. In HARPO/L, call-

ing procedures use accepted statements to finish coordinating several calling requests in different threads. We can find calling relation from ODG. Global share variable use will also be found from ODG.

- ODG benefit to distinguishe thread relations in objects.

  Essentially, there are no obvious differences between threads in the same object in different objects. Both of them can be represented with an edge. However, we can find the following differences among them. For threads in the same object, programmers often use explicit synchronization methods to enforce execution order such as post-wait, semaphore or unlock-lock. For threads in different objects, multiple threads interfere with each other in the form of implicit synchronization such as calling and called.

- ODG does not take the program statement's order information, and it only reflects object relations.

  Because the construction of the ODG does not come from all instructions in source code, it does not take instruction order information. We can think that it is a preprocess procedure. Based on ODG analysis, we can learn information which is needed in next steps.

3.3 Algorithm for Constructing ODG

In ODG, the main elements are all objects in programs. However, source code includes many classes besides objects. We know an object is an instance of a class. So we can transfer each class into its corresponding object instances. In addition,

we know ODG does not contain any program execution order information. These execution orders are included in threads in objects. We use ACCFG to represent it in chapter 4.

In order to construct ODG, we should collect all object names, types and their use relations. We can obtain the information from the field declaration of each class and class parameters. This is a straightforward method to obtain program ODG from these two sources. Field declaration has the following form:

$$\text{Field} \rightarrow (\text{public} \mid \text{private})\ \textbf{obj}\ \text{Name} : \text{Type}\ [:= \text{InitExp}]$$

This expression includes object accessibility (public or private), name, type information and field name.

Class parameter is another aspect considered. The classes often include some parameters. These parameters imply some 'knows' relations. This is a happening call case. An example can be seen in Figure 4.7.

In order to construct ODG, we use two steps: Collection, and Connection.

1. **Collection**

   First we collect all objects in HARPO/L programs. For each class, all instances will be collected. These objects from certain classes, contain objects listed in the field declarations. All these objects are a part of the host object. There are part-of relations here. If some of these objects come from other classes, these objects can also include objects listed in their class field declaration section.

2. **Connection**

   After collecting all objects in programs, we connect them with each other. There

are two relations here. For host object and objects listed in their field declaration, the relation is ′ part-of ′. The direction is from field to host. For object and object from its class parameters, the relation is ′ knows ′. The direction is from host object to argument.

We can see an example of ODG construction in Figure 3.2. In this example, we list class declarations in Figure 3.2(A). There are three classes: A, B and C. All object procedures start in main class. In Figure 3.2(B), their ODG is shown. We can know from this ODG, that object a, derived from class A, includes objects a1, a2, and a3. In addition, object a and object b both have knows relation with object c. This is a shared use for object c. For this case, we can know that object a and object b race for the use of object c.

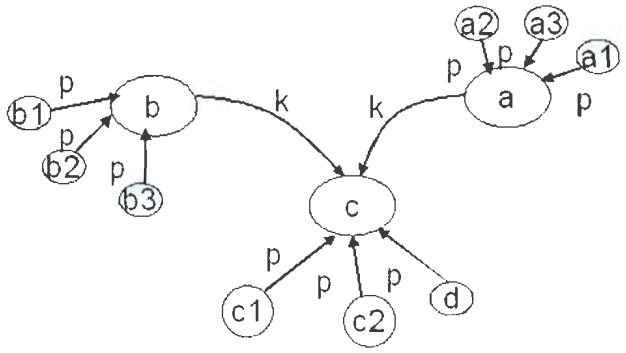
When several different objects include objects, whose names are same, these objects with same names do not represent the same objects. We need to use *parentobjectname.childobjectname* for representing it. In this way, the ODG is constructed. It shows the hierarchy of objects in programs. In this example, we can pick up object a, b and c as subjects analyzed in ACCFG analysis. In other words, these three objects are program skeletons.

The rules of identifying skeleton objects are: Firstly, if a skeleton object can not be a part of other object. That means this object has no ′ part-of ′ relation with others. Secondly, these objects can be used by other objects. The relation of skeleton object and other object is only ′ knows ′.

main class

class A [in c:C]	class B [in c:C]	class C	public obj c:C
obj a1: int =0;	obj b1: int =0;	obj c1: int =0;	public obj a[c]: A ;
obj a2: int =0;	obj b2: int =0;	obj c2: int =0;	
obj a3: int =0;	obj b3: int =0;	obj d: int =0;	public obj b[c]: B;

(A) Object Collection



(B) Connect object to ODG

Figure 3.2: The construction of ODG

Chapter 4

Augmented Concurrent Control Flow Graph

4.1 Introduction

In this chapter, we firstly discuss the principles and relations of memory consistency models and possible strategies in HARPO/L. Secondly, after briefly introducing CCFG method in [10], which can represent source code with parallel and sequential behaviors, we present its extension, Augmented Concurrent Control Flow (ACCFG), for HARPO/L. Thirdly, ACCFG is translated into Concurrent Single Static Assignment (CSSA) form. Lastly, we will introduce several confluence functions including ψ, ϕ, π and ξ and give their placement algorithms.

4.2 Memory Consistency Models

4.2.1 The Classes of Parallel Programs

Parallel programs are different from the sequential programs, since they include parallel control flow, such as Cobegin Coend, and thread synchronization. The classification of parallel programs can benefit our comprehension for parallel programs and consistency models.

In [27], Vivek summarized parallel programs into three different categories: deterministic parallel programs, nondeterministic data-race-free parallel programs and nondeterministic parallel programs with data races.

In deterministic parallel programs, the same input through multiple paths always produces the same output. In these parallel programs, there are no data races between different threads. In nondeterministic data-race-free parallel programs, the same input through multiple different paths can produce different outputs. These parallel programs include deterministic parallel programs and some synchronizations such as acquire-release. The shared variable accesses are controlled by control flow or data synchronization. In these programs, programmers can guard programs to free from data races. The nondeterministic parallel programs with data races are most complicated. Its output is nondeterministic and data races are allowed in it. So the programmers cannot determine the program behaviors in advance. Concurrent data accesses are not protected by synchronization.

In Figure 4.1, we can see the examples of these three classes. In the class 1, there are no data race. In the class 2, the synchronization enforce control execution order. In the class 3, there exist data race, the variable a in the statement $b := a + 1$ can

use the definition of $a := 1$ or $a := 2$. This cannot be decided in advanced.

```
obj a: int =0;
obj b: int =0;
obj c: int =0;
(co
   a:=1;
||
   b:=2;
   c:=3;
co)
```

(A) Class 1

```
obj a: int=0;
obj b: int=0;
(co
   a:=1;
   post(s);
||
    wait(s);
    a:=2;
    b:=a+1;
co)
```

(B) Class 2

```
obj a: int=0;
obj b: int=0;
(co
    a:=1;
||
    a:=2;
    b:=a+1;
co)
```

(C) Class 3

Figure 4.1: The Parallel Program Classification of Vivek

4.2.2 Memory Consistency Models

In distributed shared memory system, there are many possible data consistency models. The system supports a given model if operations on memory follow specific rules. A data consistency model specifies a contract between programmers and systems. If programmers follow certain model rules, the system keeps memory consistent and the results of memory operations will be predictable. Memory consistency models include strict consistency, sequential consistency, causal consistency, release consistency, eventual consistency, delta consistency, atomicity consistency, and weak consistency [33]. The following consistency models are listed from the strong to the weak.

- **Strict Consistency**

  Strict consistency is most stringent for memory coherence. This consistency obeys this principle [33]: Any read to a memory location X will get the value, which is the most recent write to X. Based on this rule, all write operations

and values will be instantly visible to all processes. After the writes are done, all subsequent reads will see the value immediately. Furthermore, a read can get the current value immediately, no matter how fast the next write is done. However, this leads to inefficiency since there are more data movement and synchronization requirements than those programs really need.

- **Sequential Consistency**

  Sequential consistency is a slightly weaker model than strict consistency. It was defined by Lamport "the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program." [16]

- **Weak Consistency**

  In weak consistency model, synchronization variable accesses are sequentially consistent. The weak cinsistency model obeys this rule: Only after all previous write operations are finished, a synchronization variable accesses can be allowed. And after all previous synchronization variable access are finished, the following read and write operations can be allowed. In this model, all access operations to sychronization variables are seen by all processes in the same order. It must make sure after all writes are completed, new read and write operations can be proceed.

- **Release Consistency**

  Release consistency includes two operations (acquire and release). Before a data write to a memory object, a node must acquire the object by acquire operation, and later release it. Within acquire and release operation, the operation consists

of the critical section. The system is said to provide release consistency, if all write operations by a certain node are seen by the other nodes after the former releases the object and before the latter acquires it[33].

- **Entry Consistency**

  Entry consistency also uses critical sections. In this model, acquire and release accesses are also at the start and end of each critical section. Furthermore, Entry Consistency requires each shared variable to be associated with some synchronization variable such as a lock or barrier. If elements of an array require to be accessed independently in parallel, all these elements must use different locks [33].

4.2.3 Memory Consistency in HARPO/L

In the HARPO/L, programmers can use synchronization methods to control data access for shared variables. In some cases, the HARPO/L program includes some data races, which cannot be decided in advance. This is class 3 parallel programs.

When we use strict memory consistency, programs are executed in the order of the statements. We can be illegal to insert a write between two consecutive read accesses. In this memory consistency, programs will lose the flexibility of optimization. In weaker memory consistency model, we have to guarantee the seperation of the synchronization variables and write/read operations. In the HARPO/L, we use the sequential consistency model, because of its simplicity and the availability of proof methods for it.

4.3 Augmented Concurrent Control Flow Graph (ACCFG)

Jaejin Lee [9] introduced Concurrent Control Flow Graph (CCFG). This is a rather good intermediate representation for explicitly parallel programs. It has some similarity to parallel program graphs [32], control flow graphs [2] and parallel dependence graphs [12]. It contains conflict edges for shared variable, synchronization edge for explicit synchronization statement and control flow edges. CCFG can reflect basic parallel language features in HARPO/L, but some extra elements (with accept statement in HARPO/L) urge us to extend it to adapt to HARPO/L. We call this new IR Augmented Concurrent Control Flow Graph (ACCFG).

Object-oriented languages such as Java allow multiple threads, and compilers analylze and coordinate the muliple threads' relations. In HARPO/L, we obtain the relation of threads by object dependence graph analysis. These COs can be analyzed with ACCFG analysis. These COs can be connected with shared variable and call procedure.

The definition of ACCFG:

Definition 4.1 Augmented Concurrent Control Flow Graph

An Augmented Concurrent Control Flow Graph (ACCFG) is a directed graph $G = (N, E, N_{type}, E_{type})$.

1. N is the set of nodes in G. Each node is a program basic block.

2. E is the set of edges in G. There are four types of edge. E_{ct} , E_{sy}, E_{cf} and E_{cl}. E_{ct} is the set of control flow edges.

E_{sy} is the set of synchronization edges, which show the order enforced by synchronization operations. Here synchronizations include parallel loop end nodes, and ordinary synchronization statements such as post wait, unlock-lock and semaphore.

E_{cf} is the set of conflict edges. Conflict means two memory references in different threads refer to the same memory location and at least one is a write. So, conflict edges mean a shared variable used between different threads in the same object or different objects.

E_{cl} is the set of calling edges. Accept statements accept calling requests from other threads. These calling threads use calling edges to reach accept node.

3. N_{type} is a function which tells the types of nodes. The node types include *Start*, *Exit*, *Cobegin*, *Coend*, *DoallBegin*, *DoallEnd*, *Compute*, *Calling*, *AcceptStart*, *Return*, and *Header*.

4. E_{type} is a function which tells the types of edges in the graph.□

The *Start* and *Exit* nodes are special nodes which have respectively no predecessors and no successors in a ACCFG. Several threads are created at a *Cobegin* node. These threads are merged at a *Coend* node. *DoallBegin* and *DoallEnd* are similar to *CObegin* and *COend*, and this kind of loop can run in parallel when no data dependences exist between any two iterations with different index values.

A *Branch* node is the same as one in the sequential program. If it is a loop header, it is called header node. Both branch nodes and header nodes contain conditions for branching. Compute nodes contain a sequence of assignment statements.

The direction of a synchronization edge is from the node which contains a trig event variable signal to the node which has a wait for the same event variable. Conflict edges are bidirectional edges in ACCFG, which join two basic blocks, in different threads, that refer the same shared variable.

The accept statement is a special operation in HARPO/L. It is a called node, and several calling threads may race for access for it. Following the node is a critical section, in which there are some computation procedures. From these threads sending calling requests, calling edges are used to connect calling nodes and acceptStart nodes. The calling nodes locate in the place the called procedure is used. There is a *return node* corresponding to each accept statement. The result computed will be sent back to that calling thread.

In the Figure 4.2, we can see object c call the procedure $proc1$. The value is sent to accept node, and go through the computation $y := sin(1 + x)$. The result is sent back to calling node.

In explicit parallel programs, there may be many parallel loop control flow structures, such as co loops in HARPO/L or pdo. Multiple threads run in parallel in these parallel loop structures. At their loop end nodes(coend and pdoend), these threads meet and will be synchronized. But these synchronization behaviors are not explicity, and limited within this loop end node. There are no synchronization edges for it. An example of ACCFG is shown in 4.2.

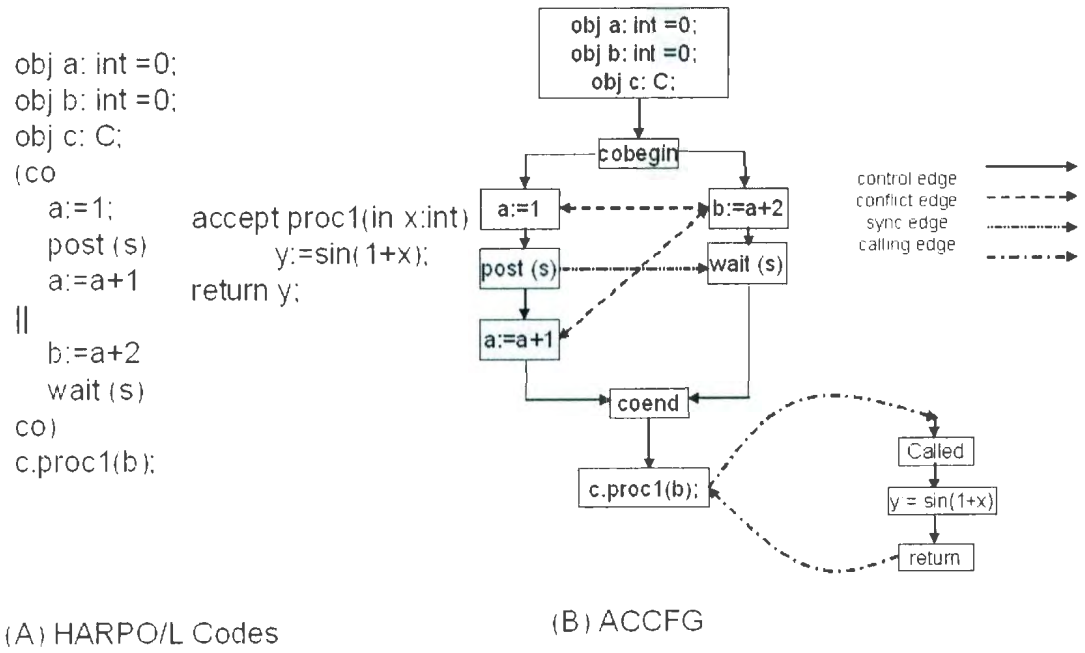


Figure 4.2: An Example of ACCFG

4.4 Concurrent Single Static Assignment (CSSA)

After we finish analysis on Complex Objects with ACCFG, we transform them into CSSA. We can also use functions ϕ , ψ and π, which are similar to those in the CSSA Form in [9]. But we add a new function ξ for the confluence function in calling nodes. These node types are defined as followed:

Definition 4.2 (ϕ function)

A ϕ function of the form $\phi(v_1,v_2...,v_n)$ is placed at a node (except for a coend node or endpdo node) where control flow edges join. n is the number of incoming control flow edges at the node. The value of $\phi(v_1,v_2...,v_n)$ is one of the v_i's values and the selection depends on control flow path followed by the program. [10]

In chapter 2, we introduced the ϕ function in SSA form. In the parallel programs, there are no difference from the sequential programs on the definition of ϕ function. Here we use the same example in Figure 4.3. In this example, the value of $\phi(a_1,a_2)$ comes from a_1 or a_2. The selection depends on whether the control condition c_0 is greater than 1.

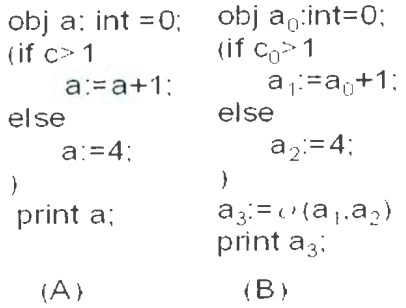


Figure 4.3: An Example on the φ function

Definition 4.3(ψ function)

A ψ function for a shared variable v has the form $\psi(v_1,v_2...,v_n)$, where n is the number of threads merging at a coend node or an endpdo node where the ψ function is placed. The value of $\psi(v_1,v_2...,v_n)$ is one of the v's values and the selection depends on the interleaving of statements in the threads merging at the node. [10]

In the Figure 4.4 (a),ψ function is located at the end of the co loop. The arguments of the ψ function come from the definitions in different threads of CO parallel loop. The selection of values does not depend on the control flow, but the interleaving of statements in different threads.

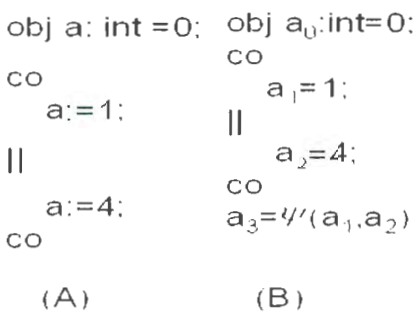


Figure 4.4: An Example on the ψ function

Definition 4.4(π function)

A π function of the form $\pi(v_1,v_2,...,v_n)$ for a shared variable v is placed where there is a use of the shared variable with t possible definitions. n is the number of reaching definitions to the use of v through the incoming control flow edges and incoming conflict δ^t edges. The value of $\pi(v_1,v_2,...,v_n)$ is one of the vs. The selection depends on the interleaving of statements in the threads computing the vs. [10]

In the Figure 4.5, an example of π function is shown. The statement $b := a$ is a use of variable. But variable is a shared variable between two different threads. The definitions $a_1 := 1$ and $a_2 := 4$ reach the use through the conflict edge. The value of π function is 1 or 4. The selection depends on the interleaving.

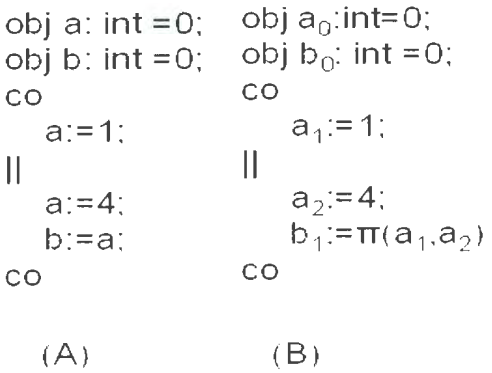


Figure 4.5: An Example on the π function

Definition 4.5(ξ function)

A ξ function of form $\xi(v_1,v_2...,v_n)$ is placed in called node (accept node), where calling edges merge. The n is the number of call merging at this node where the ξ function is placed through calling edge. v_n is the parameter value passed from the nth calling thread. The value of $\xi(v_1,v_2...,v_n)$ is one of the vs. The value selected is sent to the critical section following ξ function. The selection depends on the order of calls.

A ξ function example is showed in the Figure 4.6. Accept statement acts as an accept node, which can accept calling edges. The calling edges come from different objects, which use the procedure of the same object. In this example, both objects b and c use the proc of object a. But they use different parameters. The different parameters are sent to the proc of object. We use ξ function $x_3 := \xi(x_1, x_2)$ to solve this case. Only one value can be chosen to compute in next statement $x_4 := x_3 + 1$. The selection of value depends on the calling threads and accept node.

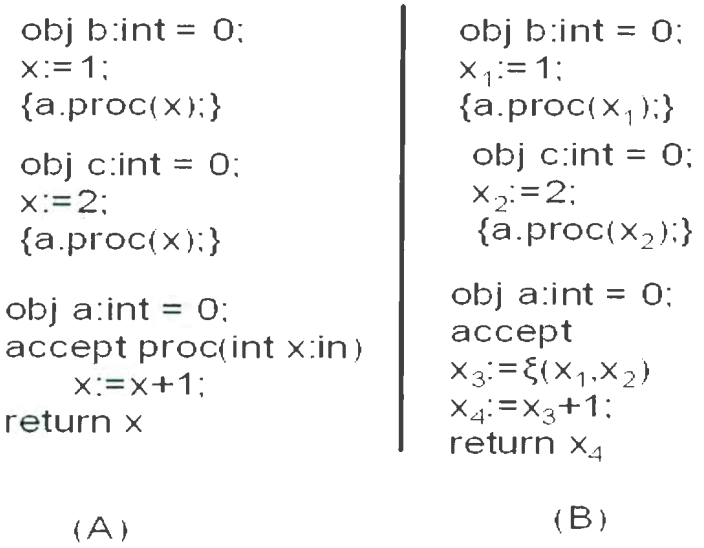


Figure 4.6: An Example on the ξ function

The φ, ψ and π functions in this thesis are the same as in [9]. The ξ function handles accept nodes in HARPO/L. In addition, the accept statement has some synchronization behaviors. Several calling threads merge in this node, and only one can obtain the access right. After that, the called node will refuse other calling threads. Computation happens in the critical section following called node. After finishing the critical section, the result of the computation will be sent back to that calling thread. The called node will be available again and can accept other requests.

4.5 Function Placement

In order to transform ACCFG into CSSA, these functions must be placed on their correct places. We will introduce these placement methods as follows.

4.5.1 Placing ϕ Function

Because ϕ function is also used in traditional SSA for sequential program, much research has been made on it. But they obey these rules: there is more than one path to join nodes; each path is not an empty path; all paths are not equal to each other; The output node is only selected from one of the paths at the same time.

Cytron and Ferrante in [24] present an efficient algorithm for generating SSA form from an arbitrary control flow graph and its dominator tree. This process first figures out at which join points to insert ϕ functions, then it inserts ϕ function. In these functions, the number of arguments are equal to the number of control flow predecessors of the join point, which includes some definitions of the variable reached, and renamed definitions and uses of variables.

This method uses the dominance frontiers and dominator trees. The dominator tree is computed using the Lengauer-Tarjan algorithm and path-compression. In [24], Cytron gives the two-pass algorithm on the dominance frontier. Using the dominance frontiers, we can decide the locations of each ϕ function for each variable in the original program and add the ϕ function for each variable at the dominance frontier of every node. The variables are renamed by placing a subscript to an original variable. All variables renamed will be checked by traversing the dominator tree in order to ensure each of these variables is defined exactly once. This is the most efficient algorithm currently known for a general flow graph, but it requires several passes over the instructions of the graph. About Cytron's placement ϕ function algorithm, more details can be found in [24].

In [18], Brandis and Mssenbck give a simpler algorithm. It is a method to gen-

erate SSA form in a single pass directly from the source text of a program. But this method must be limited to a structured program. The structured program only includes assignments and structured statements such as IF, CASE, WHILE or FOR and no GOTO statements. In a structured program, if compiler meets the join nodes, the ϕ function will be inserted in the places, and even this procedure can be finished in parsing.

The HARPO/L in our project include usual IF-ENDIF, FOR, WHILE, Parallel Loop, but no GOTO statements. So, we can think of it as a structured programming language. We can use this algorithm in [18]. Otherwise, we should pay more attention to the difference because of parallel loop control in HARPO/L. The Cobegin-Coend and Pardo loops bring control join nodes. If we treat them also as condition statements, like IF statement, this will generate an incorrect ϕ function at this node. Multithread join points are parallelly executed, and different from the sequential case. These nodes should use ψ function for them. So, it is a more accurate definition, ϕ functions for a variable should be placed at points in the parallel program where two or more definitions of the variable reach through control flow paths in the program that do not execute in parallel.

4.5.2 Placing ψ Function

When more than one thread updates the same variable, more than one SSA name of the variable may reach this use after parallel blocks. These names must be merged to preserve the SSA properties. However, this merge does not come from control flow branching, ψ function is used for it. A parallel merge means that a variable will be

updated in more than one parallel thread, and the updated value must be used in codes following the parallel block, regardless of which section finishing the update.

The algorithm to place ϕ functions given in [24, 7] is also used to place ψ functions. However, this algorithm can introduce many spurious ϕ functions. These are easily identified, since they will have only a single reaching modified SSA name at the parallel merge. After identifying them, we can remove them.

4.5.3 Placing π Function

ϕ and ψ functions cannot represent all confluences of values because the definition of a shared variable may reach a "use" from different threads. In [9], Jaejin Lee used π function to represent it.

Since HARPO/L is a parallel programming language, some objects in different threads can act as the shared variables. This causes the data race. The π function is used to solve this problem.

When some synchronization statements are introduced in HARPO/L programs, we must know some paths cannot be reached because of the order enforced by some synchronization statements. These paths must be found and removed.

We can use the same method as [9] on π function placement. In addition, data races in different threads can come from called procedure, where several threads call. We use ξ function to solve it.

4.5.4 Placing ξ Function

ϕ , υ and π functions can basically solve common parallel program issues. However, for HARPO/L, a new challenge must be faced. In sequential programs, calling procedures are regarded as parts of sequential programs. We can copy the called procedure codes to the calling nodes. The usual method is to use call graphs to represent calling relations. In HAPRO/L programs, called nodes will face the control race from different threads. If we can know the calling order of several calling threads in advance, then the called procedure can run with values passed from calling nodes one by one in order, and return the corresponding result. However, the order cannot be provided in advance. So the ξ function is introduced to solve the data race of multiple calling threads.

In HARPO/L, calling procedures are finished by several calling procedures in multithreads racing for the called procedure. This is a thread confluence and control racing issue. Each thread passes its parameter values to the called procedure. The computation is finished in the critical section of the called procedure and the result is sent back to the calling thread. Each time the called procedure only accepts one request. Other requests will be refused and will wait for the next chance when the procedure finishes.

This is a complicated process. However in HARPO/L, all objects are produced in compile time. We can know all threads before the program runs. When several threads call a called procedure, accept node will coordinate and schedule these threads execution orders and keep only one thread running each time. After the computation is done, the result value is sent back to the caller.

When only one calling thread in program sends a request for a called procedure, this provides a chance for optimizations. Since accept statements do not need to coordinate all calling requests from different threads at this time, accept statement can be removed. But the computation part in the called procedure is still useful to calling procedure. We can copy computation procedures to the calling procedure in calling nodes. Of course, the return node corresponding to the accept statement is also removed.

We can see an example in Figure 4.7. In subgraph(a), object "a" and object "b" all use the same procedure, but the procedure belongs to different objects. Object "a" uses the procedure of object "a1", and object "b" uses the one of object "b1". This cannot cause the race situation. In subgraph(b), object "a" and "b" both use the procedure of object "c". The threads in objects "a" and "b" will race for the right to access the procedure in object "c". The computation result will return to object "a" or object "b".

```
obj a:                  obj b:
obj c1: c               obj c2:c
obj a1:int =1           obj b1:int =2
a1:= a1+c1.proc1(1):    b1:= b1+c2.proc1(2):
print a1;               print b1;
```

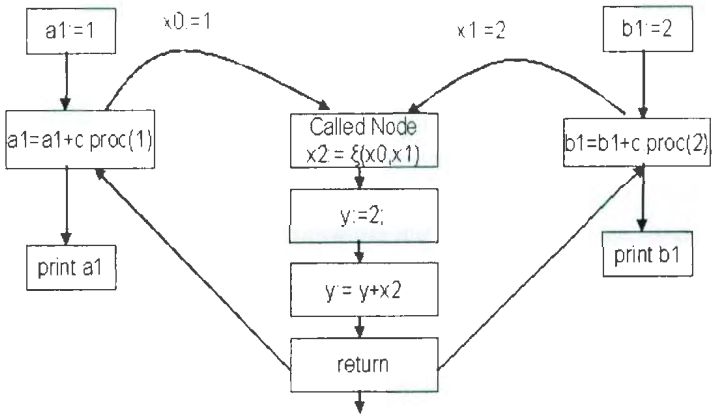
```
obj a: (in c ):         obj b: (in c):
obj a1:int =1           obj b1:int =2
a1:= a1+c.proc1(1);     b1:= b1+c.proc1(2|
print a1;               print b1;
```

```
obj c: int =0;
accept proc1(in x:int)
    y := 2;
    y:=y+x;
return y;
```

```
obj c: int =0;
accept proc1(in x:int)
    y := 2;
    y:=y+x;
return y;
```

(A) not a ξ case

(B) a ξ case



(C) ξ CCSA

Figure 4.7: Calling Procedure in Parallel Program

In this sense, the behavior in this node is similar to π function, because the definition of the values of passing parameters in different calling procedures will race for a use in the called procedure. But we must know the following differences in this node:

- Path from definition of variable in calling threads to use of variable in called thread is a calling edge, not a conflict edge as in π function.

- The number of the inputing arguments of this node function is equal to the number of calling edges, but not the number of conflict edges. For each calling edge, only one parameter can be passed.

- After the called node accepts a calling request, other calling requests will not be discarded, but will wait for the next available chance to be processed. In π function, only one definition can be chosen, the other will be discarded.

- Calling node only accept one request at a time, and the value will be sent to the critical section following the called node. Computation will be finished in this section. After that, the result will be sent back to the calling thread, and the called node will be available to accept other calling threads. Whereas π function has no computation section and return step.

- The choice of calling threads are decided not only by the latest access thread order, but also by the called guard node state. For π function, the output value is decided by the multithread interleaving, or we can say, the latest access thread can obtain the use right.

For ξ function placing, we must find calling nodes, called nodes, and their return nodes and make some possible optimizations as above. Since called nodes represent accept statements in HARPO/L, we can trivially pick them up. The return nodes are located in the tail of method implementation. For calling methods, we can know their places from ODG and control flow analysis in ACCFG. There are calling relations among objects in ODG. In ACCFG, calling nodes are located in the place where they are in the form of "the name of called object . its method name". Calling edges

connect caller and callee. If calling edge is the only one, that means accept node does not need to coordinate multiple threads. Then the computation can be moved to calling thread. This decreases communication overhead. the optimization mentioned above is processed. We give pseudo codes for placing ξ function as follows:

Placing a called node in an accept statement.

Placing a return node at the end of method implementation(marked for the end of computation)

The arguments of the ξ function comes from calling procedures through calling edges.

4.6 Summary

In this chapter, we firstly discuss various memory consistency model and related strategies in HARPO/L. Secondly, we made Concurrent Control Flow Graph (CCFG) extension as Augumented Concurrent Control Flow Graph (ACCFG) for HARPO/L. Thirdly, we introduced Concurrent Single Static Assignment(CSSA),which is the following transformation of ACCFG. Lastly, we analyzed and discussed the confluence function in CSSA and related placement algorithm.

Chapter 5

InterObject Analysis

In this chapter, the main issues are interobject problems. First, we will introduce some basic topics on interobject analysis. Then we will discuss some interobject optimizations. Lastly, we will analyze interobject procedure call deadlock problem and give its possible solutions.

5.1 Introduction

In HARPO/L language, one object can contain more than one thread, so there is also interprocedural problem in different threads in the same object. In the other case, different objects can have their own threads, and between these threads there can exist interthread relations. However, in compiler's view there are no obvious differences between these two cases. When we separate several threads in the same object into different objects which have only one thread, they will become similar. So here, we can think of interobject relations similar to interthread relations. Their analysis methods and algorithms can be replaced from each other.

Compilers use interobject analysis (interprocedural analysis) to optimize intermediate representations in global view. This differs from intraprocedural analysis because many of the benefits of interprocedural analysis derive from improving the effectiveness and applicability of optimizations within individual procedures, rather than transforming the relationships among procedures.

The interprocedural analysis can include several aspects IR4:

- Interprocedural control flow analysis, which comes from the construction of a program's control flow graph.
- Interprocedural data flow analysis, which includes both flow-sensitive and flow-insensitive side-effect analysis and constant propagation.
- Interprocedural alias analysis.
- Interprocedural optimization.
- Interprocedural register allocation.

Interprocedural alias analysis determines whether two pointers in a program may refer to the same object or whether array references refer to the same object such as $a(i)$ and $a(j)$. In the HARPO/L language, the object name refers to a unique memory address. We need not consider this.

Interprocedural optimization mainly relies on the analysis of how functions and variables are used throughout a program, and seeks to reduce or eliminate duplicate identical calculations and inefficient use of memory, and to simplify iterative sequences such as loops. Interprocedural optimization can also remove dead code. Interprocedural optimization is an important compiler behavior in compile time. Usual

interprocedural optimizations can be achieved by automatic recognition of standard libraries, localization of statically bound variables and procedures, partitioning and layout of procedures from their calling relationships, and global alias analysis.

In compiler optimization, register allocation is the process of multiplexing a large number of target program variables onto a small number of CPU registers. The goal is to keep as many operands as possible in registers to maximise the execution speed of software programs. Register allocation can happen over a basic block (local register allocation) or a whole function/procedure (global register allocation) [33]. In current compilers, the usual register allocation method is register spilling.

Interprocedural register allocation handles interprocedural registers and minimize execution time, given the register requirements of individual procedures in a program. The usual methods are to use interprocedural register allocator. These allocators relied on heuristic. Steven and Charles presented a save-free interprocedural register allocator and an interprocedural register allocator that spills registers as necessary across calls in [15].

5.2 Interobject Optimization

In HARPO/L Language, we use object dependence graph (ODG) and Augmented Concurrent Control Flow Graph (ACCFG) as the intermediate representations of its compiler. Calling relations can be obtained from the ODG and ACCFG of programs instead of call graph used in traditional methods. The ODG reflects calling relations between objects; The ACCFG provides the information of calling sites, because a calling procedure uses the explicit statements in HARPO/L Language. We can say

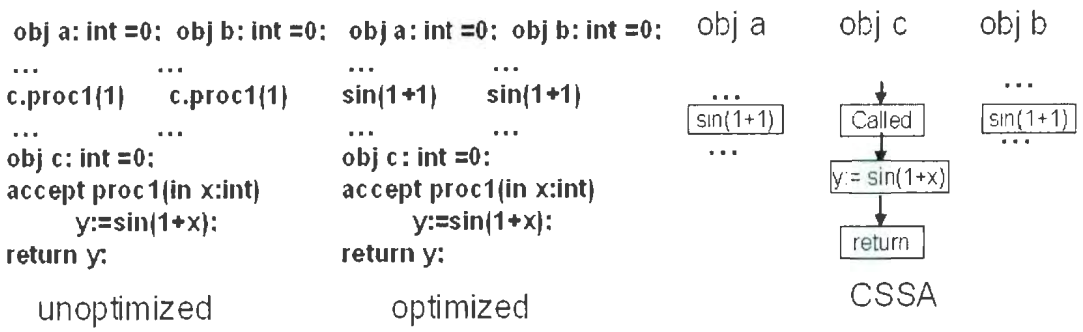
ACCFG represents program in a global view. All threads and objects can be included in it. Its control flow analysis can be regarded as interprocedural control flow analysis. This ACCFG will be transformed into dataflow graph as our final form of IRs.

In HARPO/L programs, procedure call is achieved with accept statements. Several calling procedures in different threads race the access right of this accept statement. But we should consider these special cases:

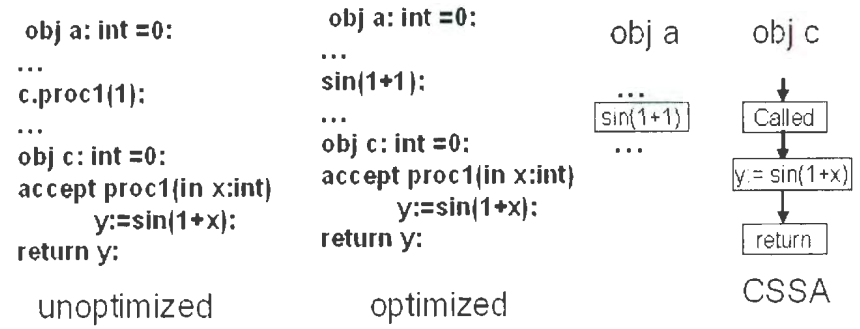
1. if these calling procedures pass the same constant values to the called node, we can duplicate a copy of the intermediate form of called procedure in each place and remove the calling edge.

2. if there is only one calling procedure for the called procedure, we can duplicate a copy of the intermediate form for the called procedure in calling procedure and remove the calling edge from IR.

These two special cases are showed in Figure 5.1. In the Figure 5.1 (A), two procedures pass the same parameter to accept nodes. Under the normal cases, two calling threads will race the 'using' right. But we duplicate the computation procedure in accept node to the calling place and remove the calling edge. In the Figure 5.1 (B), Only one calling procedure stays in the programs. We can copy the computation procedure to calling node and remove the calling edge.



(A) Special Case 1



(B) Special Case 2

Figure 5.1: Some Special Cases for Interprocedural Optimization

The elimination of unnecessary bounds checking within procedures is another interprocedural optimization. Many source codes are written to create and manipulate arrays of arbitrary size, but the size of arrays used in programs is only determined in the main program. According to the analysis of this information, we can resize and tailor array size to result in nontrivial speedup. However, the bound checking in a hardware implementation such as in CGRA is too expensive and in HARPO/L array, size is fixed after expansion of generics. So we didn't apply this technique in HARPO/L compiler.

Beside the above, the interprocedureal optimizations in HARPO/L can include interprocedural dead code elimination, loop invariant code motion, common subexpression elimination. Since these interprocedural optimizations have their intraprocedural counterparts, we will introduce both aspects of them in Chapter 6.

5.3 Deadlock Analysis and Solution

A deadlock is a situation referring to a specific condition, when two or more processes are waiting for resources in a circular chain. Deadlock is a common problem in multiprocessing.

For deadlock problems, we can divide them into two classes: deadlock resulting from programs containing synchronization. The other is from the calling procedure. Therefore, we can also consider deadlock problems arising from inter procedures. In the following section, we will discuss both these aspects.

In HARPO/L programs, there are synchronization, control flow, data race. Among its intermediate representations, we use ACCFG. In an ACCFG, nodes are the program statements, and the edges include conflict edge, synchronization edge, control flow edge and calling edge. These edges are directed. The synchronization edge represents execution orders imposed by synchronization. The direction is from the statement trigging an event variable to the statement of waiting this variable. For example, for a post-wait synchronization the direction of synchronization edge walk from "`post"` to "`wait"` on the event variable.

The introduction of synchronization edges makes deadlock possible. Deadlock occurs if there is a cyclic wait, so that every node in the cycle is waiting for one of the

other nodes in the cycle to proceed. Here deadlock implies that there is infinite loop.

On the other hand, deadlock derived from calling procedures. When each of two or more calling procedures in different threads act as caller and callee each other, this can cause a calling deadlock. We can find this deadlock relation from the ACCFG of programs. We can see an example in Figure 5.2.

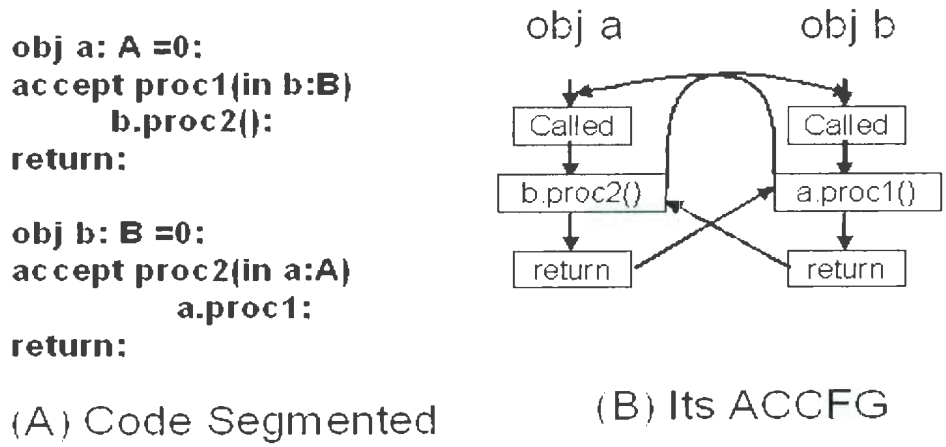


Figure 5.2: An Example on Calling Deadlock

In order to avoid a deadlock, the most conservative solution is to prohibit synchronization introduction. But the synchronization is the basic element in parallel programs. Vivek and Babara give some analysis in [28]. More methods are demonstrated in [6, 4, 25, 3]. Of course, detecting the possibility of a deadlock before it occurs is a real challenge. We only use these methods to detect possible deadlock in certain specific conditions. In general cases, it is impossible to develop a universal algorithm for all possible deadlocks in advance.

In order to avoid a deadlock, some possible methods can be applied. First, we

can remove all mutual exclusion conditions; this means that no process may have exclusive access to a resource. This is an extreme method, however it may still not completely prevent a deadlock. Secondly, post-wait conditions may be removed by requiring processes to request all the resources they will need before starting. Or we ask processes to release all of their resources before requesting all the resources they will need. These two methods are inefficient. Thirdly, since a process must be able to have a resource for a certain amount of time, we have to use a hierarchical system and set certain priorities to for retaining a resource. A priority algorithm allows preemption including lock-free and wait-free algorithms and optimistic concurrency control.

Chapter 6

Optimization In Parallel Programs

In this chapter, we will discuss optimization techniques in parallel programs. First some basic introduction will be given. Then we will discuss and analyze the optimizations in parallel programs, including Common Subexpression Elimination, Dead Code Elimination, Hoistable Code, and Loop Fusion. And we will introduce fussion/fission methodology for keeping the optimization safe. Lastly, we will give a theorem and give its proof.

6.1 Introduction of Optimization

Optimization is an important part of a compiler. In some sense, intermediate representations are designed to obtain better optimization performance.

Optimization is the process of modifying a system to make some aspects of work more efficiently or use fewer resources [33]. Optimizations can include many aspects: tradeoff and bottleneck are two essential issues. Usually, optimizations reduce readability to improve performance. This makes programs harder to maintain and debug.

In addition, optimizations often make use of special cases of source codes, such as loop, constant and redundant variable etc. Corresponding to these cases there also exist loop unroll, loop fusion, common subexpression elimination, and dead code elimination etc [33].

In this chapter, we will concentrate on optimization techniques in parallel programs. Although we desire that the same optimizations applied in sequential programs can also be applied to programs written in parallel languages, those methods are not always feasible. Parallel control flow interleaving causes the optimized programs to produce nondeterminate results, which causes us either to use no parallel optimization or restrict sharing of data used in parallel programs. Both of them are not what we want. So a general analysis can give us a better comprehension of when an optimization may be safely applied in parallel programs.

J.Lee in [17] and L.Lamport in [10] separately showed: if all executions of a transformed program are sequentially consistent, the transformation will be safe. This means that the optimized programs produce only results that could have been produced by the original program. The original and optimized programs need not be identical.

We will explain optimization techniques in parallel programs by using several examples. For parallel structure, we mainly use CObegin-Coend parallel control flow, since other parallel control flow structures are similar to it.

6.2 Common Subexpression Elimination(CSE)

6.2.1 Common Subexpression Elimination

Common Subexpression Elimination is a common optimization technique. It is a transformation that removes repeated computations of common subexpressions and replaces them with uses of the saved values. Computation repeated may be complicated function or a simple expression operation. They can be saved only by the cost of using a variable.

6.2.2 A CSE Example in Sequential Programs

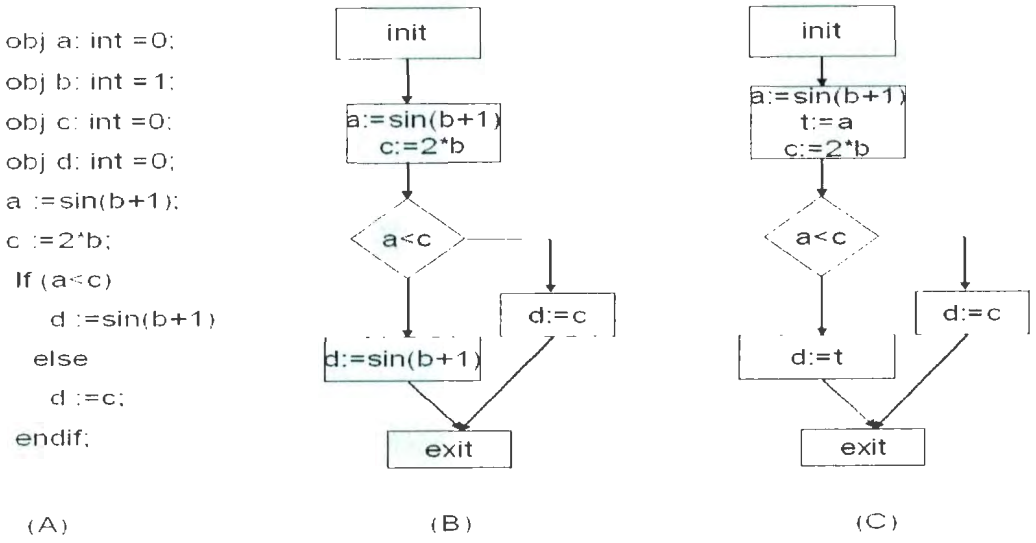


Figure 6.1: Common Subexpression Elimination in sequential programs

A sequential program example of CSE can be seen in Figure 6.1 (A). (B) is its control flow graph. This is a sequential source code segment because it contains the sequential control flow. The repeated part is $sin(b+1)$. The *Sin* function is an

expensive computation function. Variable a and d have the same value, $sin(b+1)$. In this way, twice computations of Sin function must be executed. In order to decrease the computation workload, we can use variable t to store the value of $sin(b+1)$, and variable d only load the value of variable t, in which stores the value of $sin(b+1)$. This procedure is shown in (C). This saves the repeated computation on $sin(b+1)$. So this means it increases the efficiency of programs. The second same $sin(b+1)$ factor is removed from its original program.

6.2.3 A CSE Example in Parallel Programs

CSE in parallel programs is not so easy as in sequential programs. An example of a parallel program CSE can be seen in Figure 6.2.

In Figure 6.2 (A) are similar codes to the sequential one, but it includes a

```
obj a: int =0;
obj b: int =1;
obj c: int =0;
obj d: int =0;
 (co
T1:
S1:   a:=sin(b+1);
S2:   c:=b*2;
S3:   d:=sin(b+1);
       ||
T2:
S4:    b:=2;
   co)

  (A)
```

```
obj a: int =0;
obj b: int =1;
obj c: int =0;
obj d: int =0;
 (co
T1:
S1:   t:=sin(b+1);
S1':  a:= t;
S2:   c:=b*2;
S3:   d:= t;
       ||
T2:
S4:    b:=2;
      co)

  (B)
```

Figure 6.2: CSE in parallel programs

parallel control flow *co* structure. In Figure 6.2 (B), we directly use the sequential CSE method. If thread T2 can be ignored, this transformation is correct, since the value of b is unchanged from statement S1 to statement S3. But if the execution sequence is $S1$, $S1'$, $S2$, $S4$, $S3$, the result of the program cannot be equal to any result from the original program. Therefore, when the sequential CSE method is directly used in parallel programs, this can lead to incorrect results. In order to avoid the errors, we must guarantee the execution order of statement S4 before S1'.

6.2.4 A CSE Example in Parallel Programs with Synchronizations

In parallel programs, there are more than one thread in them. These threads can communicate and coordinate with various synchronization methods. In *HARPO/L*, all synchronizations are via *accept* and *atomic*, but we also can discuss other synchronization methods for parallel programs such as post/wait, semaphores, barriers, or monitors. These synchronization methods can be implemented with *atomic*. For example, nonfair semaphores can be implemented as

```
P(s) :  obj x := false
       (wh (not x) do
           (atomic (if s > 0 then s:=s-1  x:=true) )
        )
V(s) : (atomic s:=s+1)
```

An atomic operation in computer science refers to a set of operations that can be combined so that they appear to the rest of the system to be a single operation

with only two possible outcomes: success or failure.To accomplish this, two conditions must be met:[33].

1. Until the entire set of operations completes, no other process can know about the changes being made (indivisibility);

2. If any of the operations fail then the entire set of operations fails, and the state of the system is restored to the state it was in before any of the operations began.

In Figure 6.3, we illustrate two examples with post-wait synchronization method. The codes in Figure 6.3 (A) and (C) are similar to the one in Figure 6.2. The difference between Figure 6.3 (A) and (C) are only the place of Post-Wait. When we use CSE in Figure 6.3 (A). The transformation is not correct. The value of d in (B) is changed by this CSE. In (A), because of post-wait synchronization, a predefined order can be obtained. In $d := sin(b + 1)$, the value of b is equate to 2. The CSE transformation set b to 1 in $d := t$. We cannot make CSE optimization in (A). However, though the example in Figure 6.3(C) has certain predefined order, the CSE transformation is a safe one. Since the post-wait synchronization is introduced, statements in thread1 can execute as in sequential programs. We can make a CSE optimization in (C).

6.2.5 CSE Analysis in Parallel Programs

CSE is a common optimization in sequential programs. We can think a CSE is safe in sequential programs, since all instructions execute in sequential order. But in parallel programs, parallel control flow and multthread synchronization complicate the CSE analysis. How to guard a CSE optimization in parallel programs is correct?

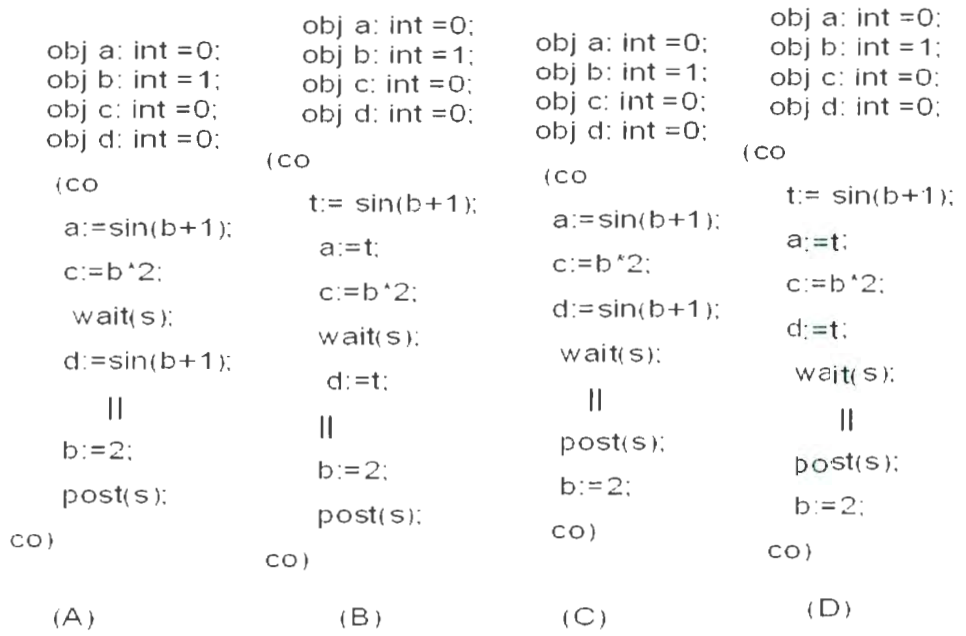


Figure 6.3: CSE in Parallel Program with Synchronization

CSE optimization uses a variable to store the result of complicated computation, and replaces the same complicated computation with this variable. In Figure 6.2, a CSE optimization will be safe if it has no thread2. This is because there are only uses of variable b in the left thread. The $b := 2$ in thread 2 is a definition of variable b. The complicated computation replaced with a variable t includes the use of variable b. The use of variable b in the original complicated computation can come from the different definitions of variable b. If we only simply use a variable t to replace all complicated computation, this removes certain constrains on the use-def chains of the variable b. This CSE optimization is unsafe.

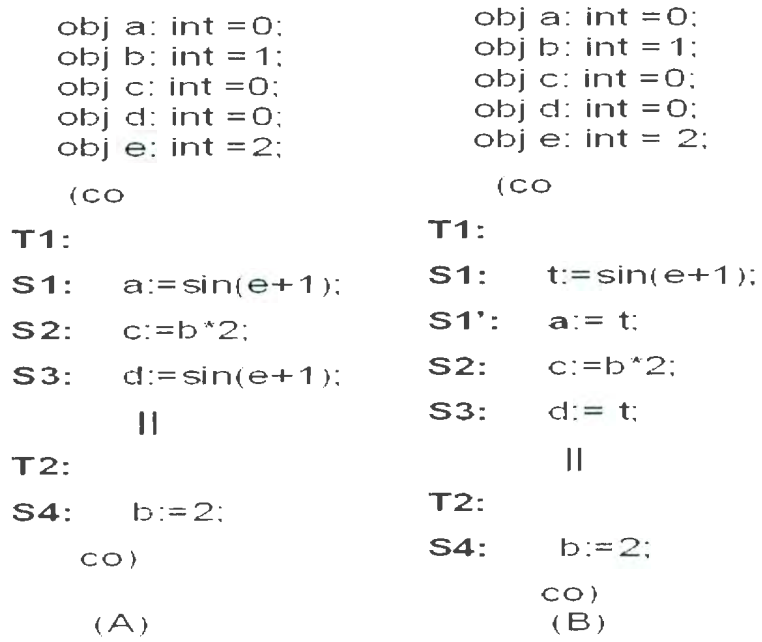


Figure 6.4: CSE in parallel programs

In Figure 6.4, in the complicated computation $sin(e+1)$ of statement S1 and S3, both uses of variable e come from the same definition of variable e. The definition of variable b in thread 2 cannot affect the $sin(e+1)$. This CSE optimization is safe.

We will make a further analysis on the valid CSE in parallel program in Figure 6.5. In this Figure, the codes are similar to Figure 6.2. In order to clarify the problem, we introduce two terms: *fusion* and *fission*. These two concepts come from the atomic operations in computer science.

First, in the Figure 6.5 (A) the transformation from (1) to (2) changes the statements $a := sin(b+1)$ and $d := sin(b+1)$ into atomic operation $< a := sin(b+1)d := sin(b+1) >$. This is an example of fusion. Actually, in step 1 variable b can use the different definition, but in step 2 the variable b uses the same definiton. The values of variable b in $a := sin(b+1)$ and $d := sin(b+1)$ will be always same because of

atomic operation. This decreases the possiblity of choosing value of variable b, so this reduces the nondeterminism of the results. Reducing the nondeterminism will be always reasonable. This transformation is called as *fusion*. It means several possible selections fuse into one.

Then, between (2) and (3), optimization can happen safely since the same definition of variable b reaches both uses. This CSE optimization in parallel is safe in the range of atomic operations.

Last between (3) and (4), the atomic operation is split. This will increase the possible selections of variable b from different definition. This procedure is called *fission*. This means the uses of variables, whom orginally come from the one definiton, can come from the different definitons. This increases the nondeterminism of the results. This creates hazards that can produce the incorrect results. But in this Figure (A), we can know there is only one use for the variable b. Provided no other definitions of variable a can reach $d := a$, so this fission is safe.

In the Figure 6.5 (B), we use the similar methods for these codes. The procedures are different from the codes in the (A) since $c := b * 2$ is inserted between $a := sin(b + 1)$ and $d := sin(b + 1)$. Since the fusion is always correct, we can transform (1) to (2) and get $< a := sin(b + 1); c := b * 2; d := sin(b + 1); >$. In the range of these atomic operations, because the value of the same variable are same, we can make CSE optimization in the atomic operations. But when we make a fission to remove the atomic operations, the use of variable b in $a := sin(b + 1)$ and $c := b * 2$ will change the possibility from the same definitions to the different definitions. This increases the nondeterminism of the results. So the fission can cause the incorrect result. The CSE optimization is not safe.

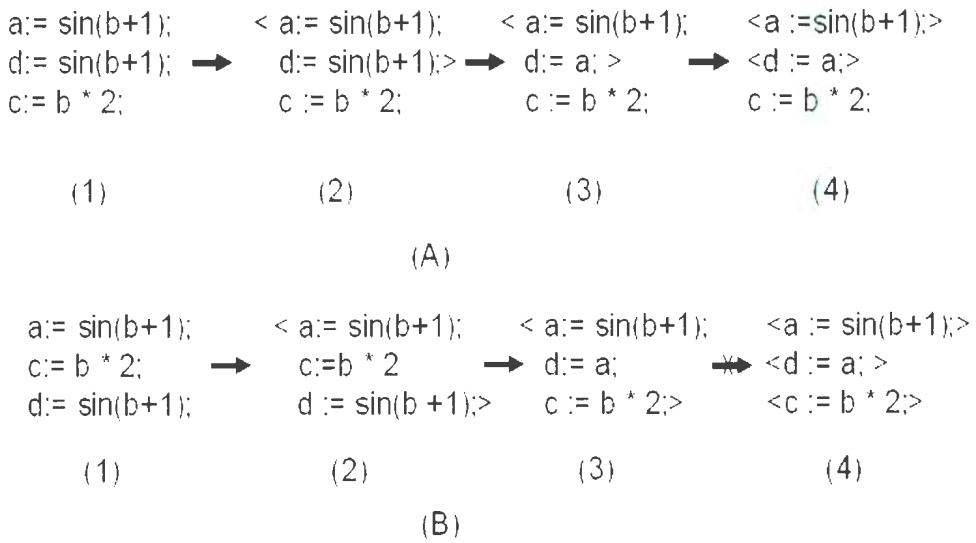


Figure 6.5: Atomic Fusion/Fission Analysis on CSE in parallel programs

This problem can be solved with the introduction of local variables. Here, local variables mean these variables can't be written by other threads. If we can keep local variables same in the thread, which includes atomic operations. The fission for the atomic operation will be safe. This procedure can be seen in the Figure 6.6.

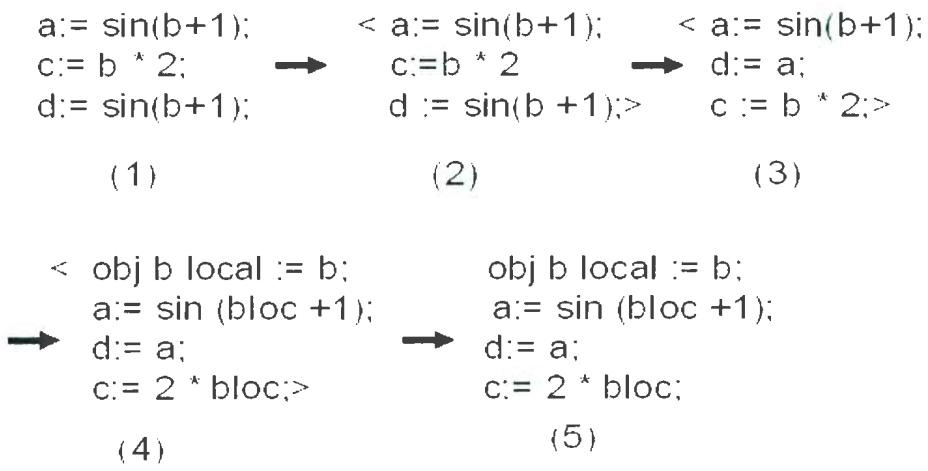


Figure 6.6: The Introduction of Local Variable for The Fission of Atomic Operation in Parallel Programs CSE Optimization

The synchronization can cause some enforced execution order. In Figure 6.3, the complicated computation $sin(b+1)$ locates in thread1, and a new definiton of variable b locates thread 2. This seems to cause CSE optimization incorrect. However, the synchronization operations post and wait cause $d := sin(b+1)$ before $b := 2$. The two use of variable b in both $sin(b+1)$ still come from the same definition. So this CSE optimization is still safe.

Until now, identifying whether the CSE optimizations are safe become the question, which is about whether a fission is safe? This question will be addressed in section 6.6.

6.3 Dead Code Elimination

6.3.1 Dead Code Elimination

Dead code is code which cannot be reached in any execution of the program, or produces no change in memory that affects later statements. It can be removed from the programs. Dead code includes code that can never be executed (unreachable code), and codes that only affect dead variables, which are variables that are irrelevant to the program [33].

6.3.2 Dead Code Elimination in Sequential Programs

Although programmers do not intentionally produce dead codes in programs, some optimizations such as copy propagation can introduce dead codes.

In sequential programs, the usual dead code elimination method is to check the use-def chain of the variables used in essential instructions in programs. If an instruction is in the use-def chain, it is not a dead code, otherwise if its only use is in a definition of itself, it is a dead code. These essential instructions specify those instructions which can affect programs, such as input/output, function call and control flow. If this instruction is a dead code, it can be removed and will not affect the results of the programs.

In Figure 6.7, code execution is linear because of sequential program. Variable "a" cannot be 0, and the condition $a! = 0$ cannot be satified. The statement $b := 2$ cannot be executed and is dead code. It can be safely removed.

```
obj a: int =0;
obj b: int =0;
b:=a
   if (a!=0)
      {b:=2 }
a:=1
```

Figure 6.7: Dead Code Elimination

6.3.3 Dead Code Elimination in Parallel Programs

Generally, the dead code elimination in parallel programs is similar to the one in sequential programs. However, because there are parallel control structures and multithreads synchronization in parallel programs, dead code elimination is more complicated than in sequential programs.

We can see an example in Figure 6.8. This example uses similar codes to the one in Figure 6.7. If these codes execute in sequential order or there no thread 2, the $b := 2$ will be dead code. We can remove them as in sequential programs. But since the parallel control flow is in this example, the definition of variable a in thread 2 can be used in $if(a! = 0)b := 2$. This segment codes are not dead code. It cannot be removed.

```
obj a: int =0;
obj b: int =0;
co
   b:=a
  if (a!=0)
    {b:=2 }
||
   a:=1
co
```

Figure 6.8: Dead Code Elimination in parallel programs

6.3.4 Dead Code Elimination in Parallel Programs with Synchronization

When we consider the synchnization on dead code elimination, the situation is more complicated. We can see an example in Figure 6.9. In this Figure (A), S1 can be removed since it is a *blank* operation, if we obey the definition of *dead code*. In Figure 6.8(B), the program optimized produces different results from the one of programs unoptimized. The reason for causing this problem is that this *blank* operation is the synchronization condition of multithread synchronizations. We cannot remove the synchronization condition.

We can see another example on DCE in parallel programs with synchronization in

```
obj a: int =0;                obj a: int =0;
obj b: int =0;                obj b: int =0;
co                            co
T1:                           T1:
S1: while (a!=0) {  }         S1:  b:=a;
S2: b:=a;                     ||
||                            T2:
T2:                           S2:  a=1;
S3:  a:=1                     co
co
 (A)                           (B)
```

Figure 6.9: Example 1 on Dead Code Elimination in parallel programs with synchronization

Figure 6.10. Beside similar codes to the one in Figure 6.9, synchronization operations post-wait are added. The synchronization post-wait cause the enforced execution order. The $a := 1$ can not be executed. That is the definition of variable a can not

be used by the $while(a! = 0)$. It can be removed safe.

```
obj a: int =0;
obj b: int =0;
co
T1:
S1: while (a!=0) {  }
S2: b:=a;
S3: wait(s);
||
T2:
S4:  post(s);
S5:  a:=1
co
```

Figure 6.10: Example 2 on Dead Code Elimination in parallel programs with synchronization

6.3.5 Dead Code Elimination Analysis in Parallel Programs

Dead code elimination for parallel programs is essentially the same as the sequential programs. But we must consider the affects on synchronization operation and the interactions of shared variables among threads.

The basic principle is still to analyze the use-def chains. The synchronization can cause some enforced execution orders. These execution orders must be considered in use-def chain analysis. This example was showed in Figure 6.10.

Dead code is often produced since the condition statement in control flow cannot be satisfied. In parallel programs, if the condition statements of control flow have shared variable in different threads, we must make further analysis on it. If in other threads, there are only the use of this shared variable, the dead code can be removed safely. That is because there are not new definition, which can change the value of

this shared variable in the condition statements. Or, the new definition of shared variable in other threads probablly cause the original condition not to be satisfied. This will be indetified again. If the condition is satisfied, these codes are not dead codes. Otherwise, they are dead code and can be removed.

Fussion and Fission methods are not fit for dead code elimination. From above, the dead code elimination can come from the analysis of use-def. If certain codes is not reachable by control flow, we can accurately identify whether a segment of program code is dead code.

6.4 Code Motion

6.4.1 Code Motion

Code Motion (also called Code Hoisting) is another useful optimization in sequential programs. Its operating subject is the loop invariant. In sequential programs, if we find invariants in loops and remove them to before loops, or after the loop, it does not affect the semantics and results of the programs. This transformation can store constant into registers and not having to calculate the address and access the memory/cache line at each iteration [33]. This decreases its computation workload in loops and makes programs more efficient. These operations are called a code motion.

6.4.2 Code Motion in Sequential Programs

In sequential programs, there are two conditions which make it possible: Firstly, all variables in these operations have the same value in the new location as in the

original location. Secondly, no read of the computed variable receives a different value than in the original program.

In most cases, the loop invariant is moved to before the loop. But in some cases they are moved to after loop. Here, we will only give examples which move the loop invariant to before the loop.

In Figure 6.11 (A), the computation $sin(4)$*ϕ+1 is invariant in loop. We can use a variable b to replace it. Once we finish the computation before the loop and store the value into variable b. This decreases the iteration computation workload with the value of variable b.

```
obj a: int =1;
while (a<10)
{
 a:=a+sin(4);
}
```

```
obj a: int =1;
obj b:= sin(4);
while (a<10)
{
 a:=a+b;
}
```

Figure 6.11: Code Motion in Sequential Programs

6.4.3 Code Motion in Parallel Programs

In this section, we give the examples of Code Motion, which includes no synchronization in parallel programs. In Figure 6.12 (A), in thread1 a while loop is included. The $a + sin(4)$ is an invariant. It can be moved outside. The $d = a + sin(4)$ use a CSE optimization. In thread 2, there is no shared variableswhich is used in thread1 and synchronization operation. So there is a sequential order in the thread1. The definition of $a := 2$ kills the definition of $a := 1$. This code motion is safe. The result

is showed in Figure 6.12 (B).

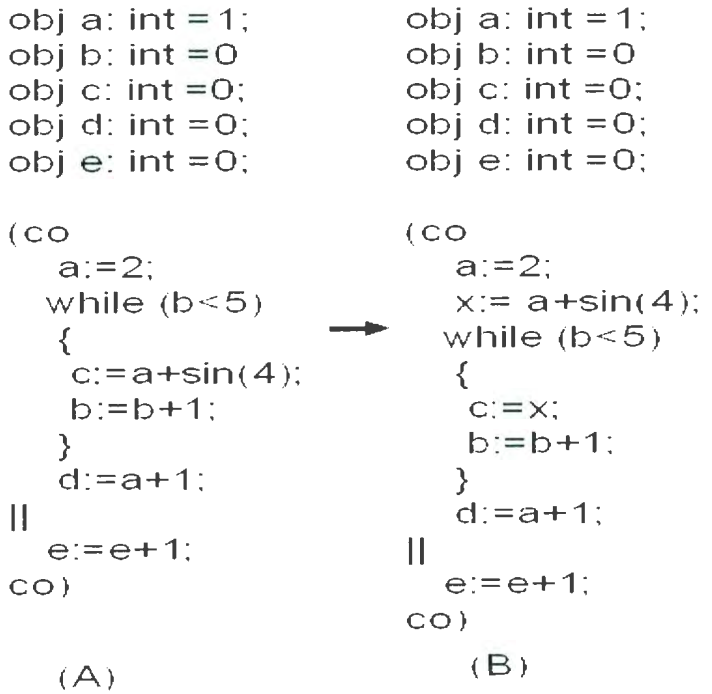


Figure 6.12: Code Motion in Parallel Programs

In Figure 6.13, similar codes are showed. The slight difference is that the definition of variable a is moved to thread 2. In this case, the $c := a + sin(4)$ is not loop invariant and cannot be moved out of while loop. That is because the definition of variable a in thread 2 can be used in the while loop in thread 1. We cannot guarantee $c := a + sin(4)$ is loop invariable. So the code motion is incorrect.

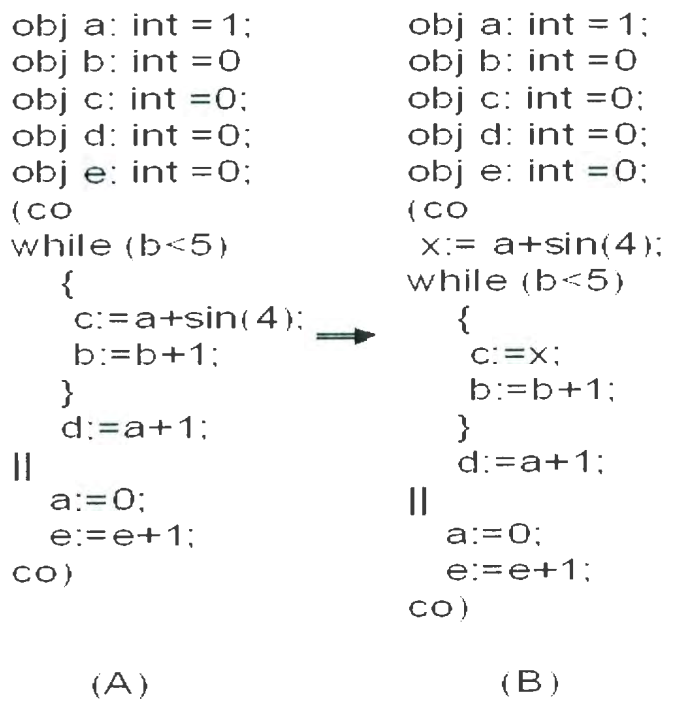


Figure 6.13: Code Motion in Parallel Programs

In addition, in the parallel program code, motion can be not only move out the while or for loop in a thread, but also move out from the parallel control flow. We can see an example in Figure 6.14.

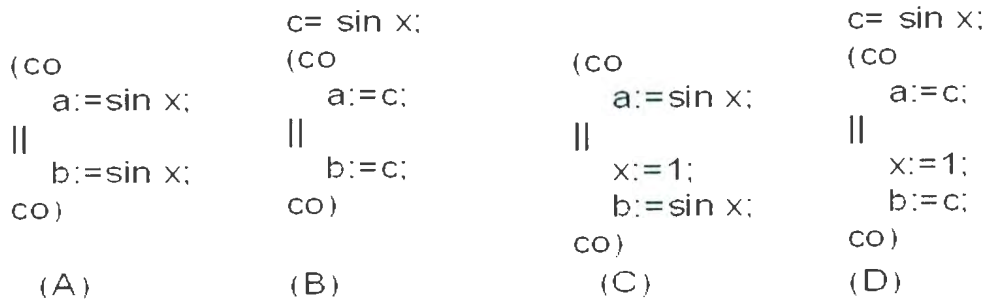


Figure 6.14: Code Motion Out of Parallel Control Flow in Parallel Programs

In Figure 6.14 (B) *sin.x* are moved out of two threads in parallel control struc-

ture *CO*. They are replaced with c because of $c = sin x$. However, in the (D) this kind of tranformation is not successful, because the variable x in codes moved has a different definition. In $b = sin x$, the value of variable x comes from $x := 1$. When we replace $sin x$ with c, the value of variable x in $b = c = sin x$ come from $x := 0$. So this is a incorrect code motion.

6.4.4 Code Motion in Parallel Programs with Synchronization

When we added the synchronization into programs, the code motion analysis become more complicated.

We can see an example in Figure 6.15. In Figure 6.15 (B), we remove S3 and S5 out of the loop, as in sequential programs. The result of the segments is different from Figure 6.15 (A). That means the transformation is incorrect.

However, the case in Figure 6.16 is different from the one in Figure 6.15. In the Figure 6.16, though the codes are similar, the results of both Figure 6.16 (B) and (A) are the same. This is a successful optimization.

```
obj a: int =0;
obj b: int =0;
obj t: bool =true;
co
S1: if(...)
S2:    wait(t);
S3:    a:=b;
    else
S4:    wait(t);
S5:    a:=b;
    endif;
||
S6:   b:=1;
S7:   post(t);
co
```

(A)

```
obj a: int =0;
obj b: int =0;
obj t: bool =true;
co
S1:   a:=b;
S2: if(...)
S3:    wait(t);
    else
S4:    wait(t);
    endif;
||
S5:   b:=1;
S6:   post(t);
co
```

(B)

Figure 6.15: An Unsuccessful Code Motion in Parallel Programs [31]

```
obj a: int =0;
obj b: int =0;
obj t: bool =true;
co
S1: if(...)
S2:    wait(t);
S3:    a:=b;
    else
S4:    wait(t);
S5:    a:=b;
    endif;
||
S6:   c:=b;
S7:   post(t);
co
```

(A)

```
obj a: int =0;
obj b: int =0;
obj t: bool =true;
co
S1:   a:=b;
S2: if(...)
S3:    wait(t);
    else
S4:    wait(t);
    endif;
||
S5:   c:=b;
S6:   post(t);
co
```

(B)

Figure 6.16: A Successful Code Motion Example in Parallel Programs [31]

What is the success key for a code motion optimization? In Figure 6.15, the statement S6 is a new definition of the variable "b" and the statements S3 and S5 are the use for variable "b". The original definition of variable "b" is in the initial part of the segments, and its value is 0. When we use a code motion optimization, we change the execution order of the program. In Figure 6.15, the compiler decides which definition the use of variable "b" comes from. But when we remove S3 and S5 out of the loop, the program cannot decide which is the correct definition of variable "b". So this leads the transformation to be incorrect. In Figure 6.16, the statement S6 is a use of variable "b". The transformation in the (B) still keeps these use-def pair relations, so the transformation is correct.

6.4.5 Code Motion Analysis in Parallel Programs

In the code motion optimization of parallel program, if we can keep all variables in a statement of a loop movable, this statement can be moved outside the loop without changing the program's meaning. In other words, we cannot violate the sequential consistency to complete a hoistable access optimization.

When we discuss the CSE optimization in parallel programs, we introduce Fussion/Fission method for the CSE analysis. This method is also fit to code motion analysis in parallel program.

First, we can see the examples in the Figure 6.12 and Figure 6.13. Both of them have similar codes. The analysis with Fusion/Fission methods can be seen in Figure 6.17.

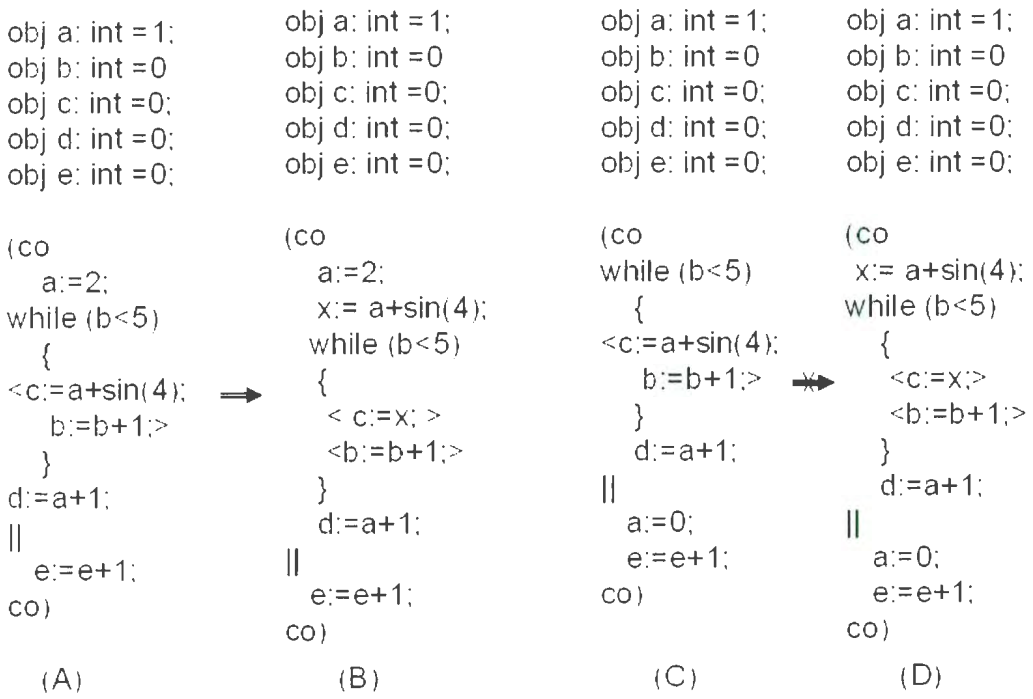


Figure 6.17: Code Motion With Fusion/Fission Analysis in Parallel Programs

In the Figure 6.17 (A), $c = a + sin(4)$ and $b := b + 1$ are put into an atomic operation. This is a fussion and is safe. In the (B), we split this atomic operation into two atomic operation. This is a fission. If we can know the fission safe, we can conclude the code motion is safe. Since this fission procedure don't introduce a new definiton of variable a, we can think this fission safe. However, although the fusion in the (C) is similar to the one in the (A), the fission introduces a new definition of variable a in other thread and increases the nondeterminism of result in the (D). This fission is unsafe.

We can see another example on code motion in parallel control flow in the Figure 6.18. In the Figure 6.18 (A) and (C), they are both fusion. The only difference is a definition of variable x in one of thread in the (C). In the (B), the fission can

introduce any new definition for variable x and is safe. But in the (D), the fission introduce a new definiton x:=0 for $c = sin.x$. This changes the result and increase the nondeterminism. This fission is unsafe.

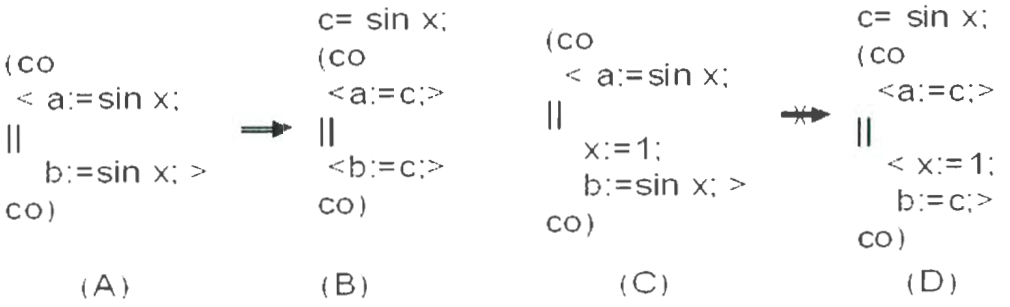


Figure 6.18: Code Motion of Parallel Control Flow With Fusion/Fission Analysis in Parallel Programs

We also can use the same method to analyze code motion with synchronization in parallel program. The fussion/fission analysis of examples in the Figure 6.15 and 6.16 are showed in the Figure 6.19. When there are synchronization operation in parallel program, we must make analysis on the enforced order from synchronization. In the Figure 6.19(B), the fission introduce a definition on variable $b := 0$. This change the value of variable $b := 1$ in the fussion. The fission is unsafe.

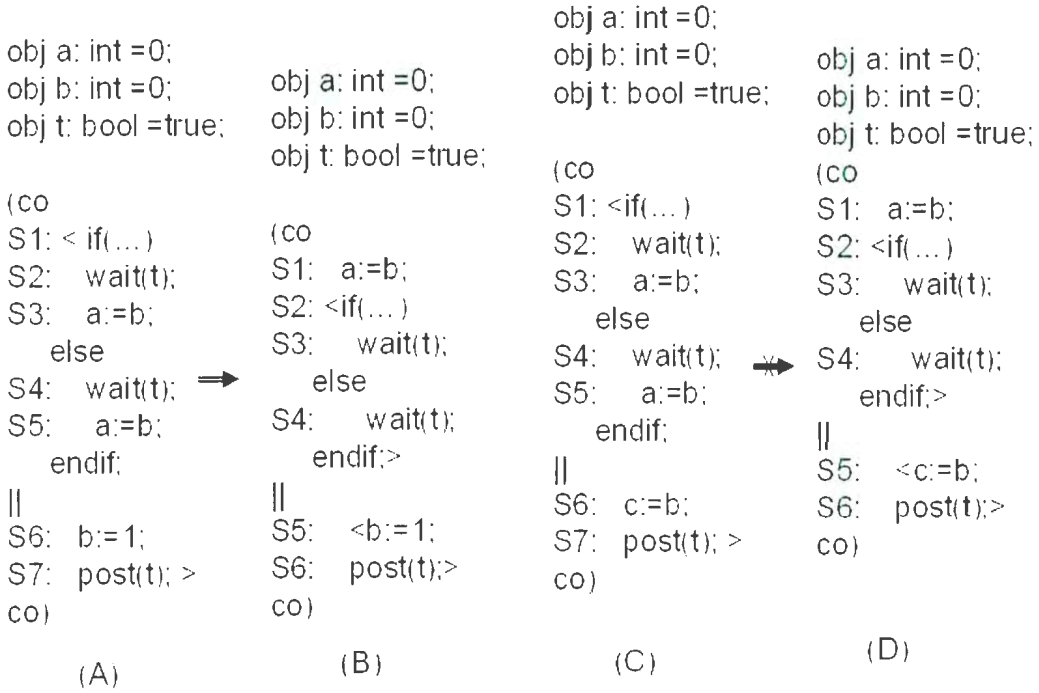


Figure 6.19: Fusion/Fission Analysis of Code Motion With Synchronization in Parallel Programs

From above, we can change the problem from whether the Code Motion optimizations are safe to whether a fission is safe? This question will be also addressed in section 6.6. We can read some other code motion algorithm in [1, 11, 13].

6.5 Loop Fusion and Loop Fission

6.5.1 Loop Fusion and Fission

Loop optimizations are most important parts in computer programming and compiling. They can improve cache performance and provide more effective use of parallel processers.

Loop fusion is a transformation that merges multiple loops into a single loop. However, in some cases one single loop cannot provide better performance than two loops, because of data locality increased within each loop. In these cases, a single loop should be transformed into two loops. This is called loop fission. Loop fission breaks a loop into multiple loops over the same index range but each taking only a part of the loop's body. This can achieve better utilization of locality of reference [33].

6.5.2 Loop Fusion and Loop Fission in Sequential Programs

For loop fusion, we fuse the same or close spaced loop together. This decreases loop overhead, increases computational density, benefits to the improvement in software pipelining, and increases the efficience of cache locality. For loop fission, we split the loop into multiple loops. This reduces register pressure, seperates loop with condition into condition-free and condition-containing loop to isolate dependencies, and benefits to loop interchange.

In the Figure 6.20, we give two simple examples for loop fission and loop fussion. We also can find these two procedures are reverse from this figure. In the Figure 6.20 (A) two loops have the same number of loop bound. So the statements S1 and S2 were merged into one loop. And in (B), the two statements in loop are splitted into two loops.

The optimization of Loop Fusion and loop fission are safe in sequential programs, when variable def-use chain dependences within the two loops occur in the same order before and after loop fusion and fission.

```
obj a: int =0;          obj a: int =0;
obj b: int =0;          obj b: int =0;
     for (i:=0,10,)     for (i:=0,10){
S1:    {a:=a+1;}           a:=a+1;
     for (j:=0,10)         b:=b+1;
S2:    {b:=b+1;}         }
```

(A) Loop Fusion

```
obj a: int =0;          obj a: int =0;
obj b: int =0;          obj b: int =0;
for (i:=0,10){               for (i:=0,10,)
   a:=a+1;              S1:    {a:=a+1;}
   b:=b+1;                   for (j:=0,10)
 }                      S2:    {b:=b+1;}
```

(B) Loop Fission

Figure 6.20: Loop Fusion and Loop Fission Example in Sequential Programs

We must keep both loop bodies have no reference to each other's data before loop fusion. Otherwise loop fusion can be unsuccessful. We can see an example in the Figure 6.21. We will avoid this case in the following section.

```
obj a: int =0;
obj b: int =0;
obj c: int=0;
for (i:=0,10,)
  {b:=a+b;}
for (j:=0,10)
  {c:=b+c;}
```

(A)

```
obj a: int =0;
obj b: int =0;
obj c: int=0;
for (i:=0,10,)
  {b:=a+b;
   c:=b+c;}
```

(B)

Figure 6.21: Loop Fusion in Sequential Programs

6.5.3 Loop Fusion and Loop Fission in Parallel Programs

In the parallel programs, the situations become more complicated because of parallel control structure. We still use the CO parallel structure. In the Figure 6.22 (A) the thread 2 has no shared variable and data, so the loop fusion is safe. And (B) the fission is safe because of the same reason.

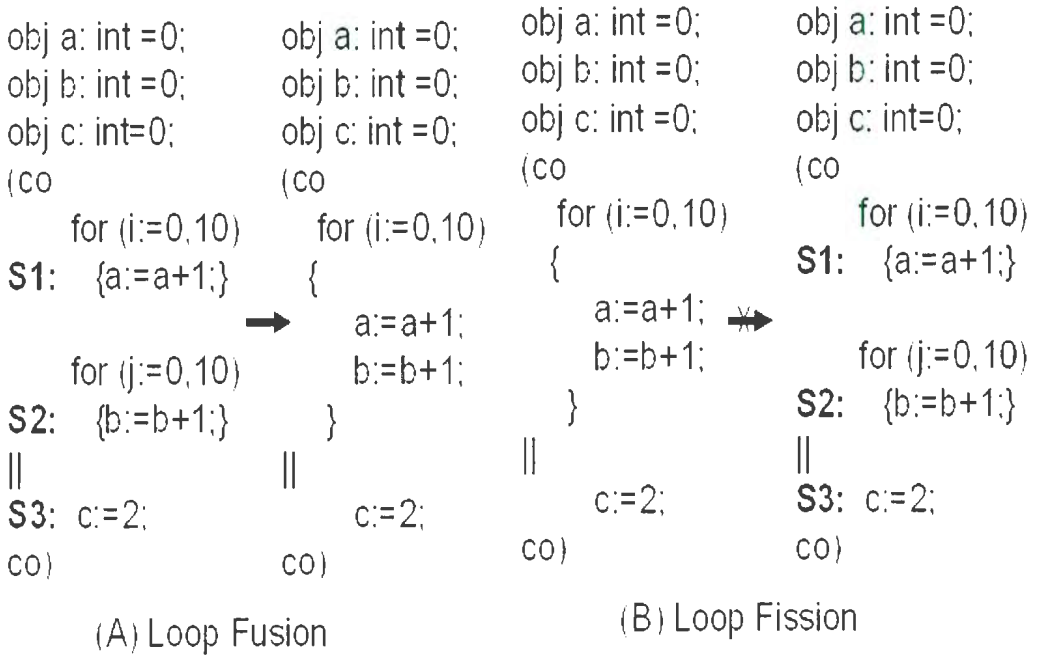


Figure 6.22: Loop Fusion and Loop Fission in parallel Programs

However, when there are shared variables in different threads, the situations become different. The Figure 6.23(A) shows an example on loop fussion. The thread 2 has the shared variable within the loops in the thread 1. The statement S1 must be executed before S2, but S2 can be executed before S1 in its program code optimized in sequential program rule. In the same reason, the loop fission example in the Figure 6.23 (B) S2 can be executed before S1 and after S1, but the program transformed reduce it to only S1 before S2. These optimizations violate the sequential consistency. They are all unsafe.

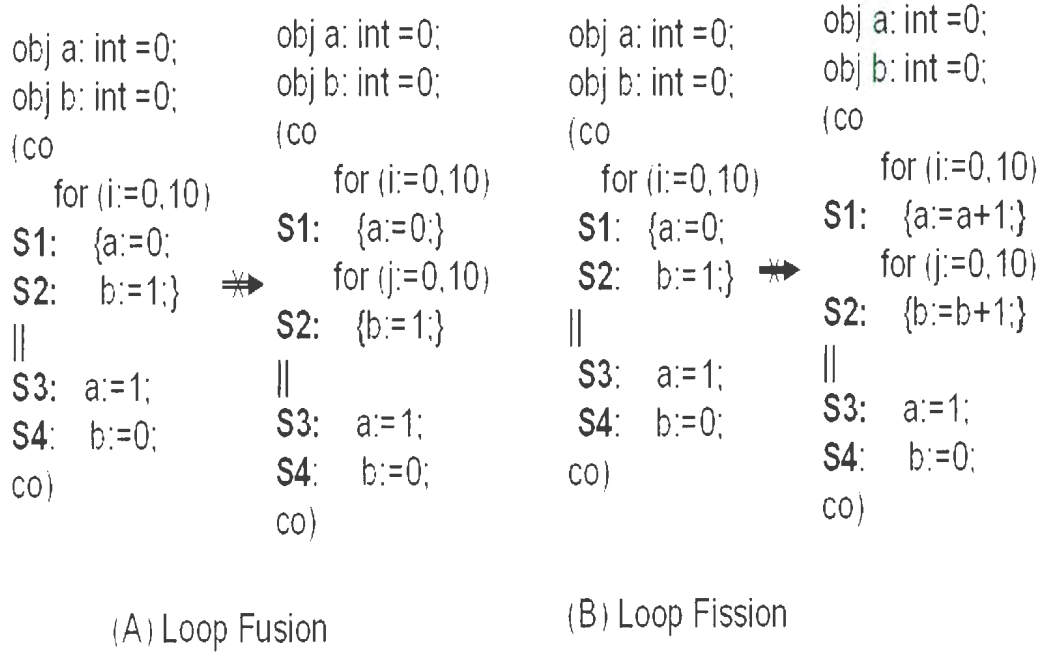


Figure 6.23: Loop Fusion and Loop Fission in Parallel Programs

6.5.4 Loop Fusion in Parallel Programs with Synchronization

We use an example to illustrate parallel programs with synchronization in Figure 6.24. In Figure 6.24 (A), the first loop includes two statements, and the second loop includes one statement. Their structures are similar to each other. We can merge them together. However, we find that the result of program optimized is different from the original one.

That is because the statements S1 and S5 are the synchronization conditions, which keep the execution order that S2 must be executed before S3. The merging of the two loops makes it possible, S3 executes before S2. For the similar reason, the loop fission example in Figure 6.24 (B) is also not safe since it violates the program sequential consistency.

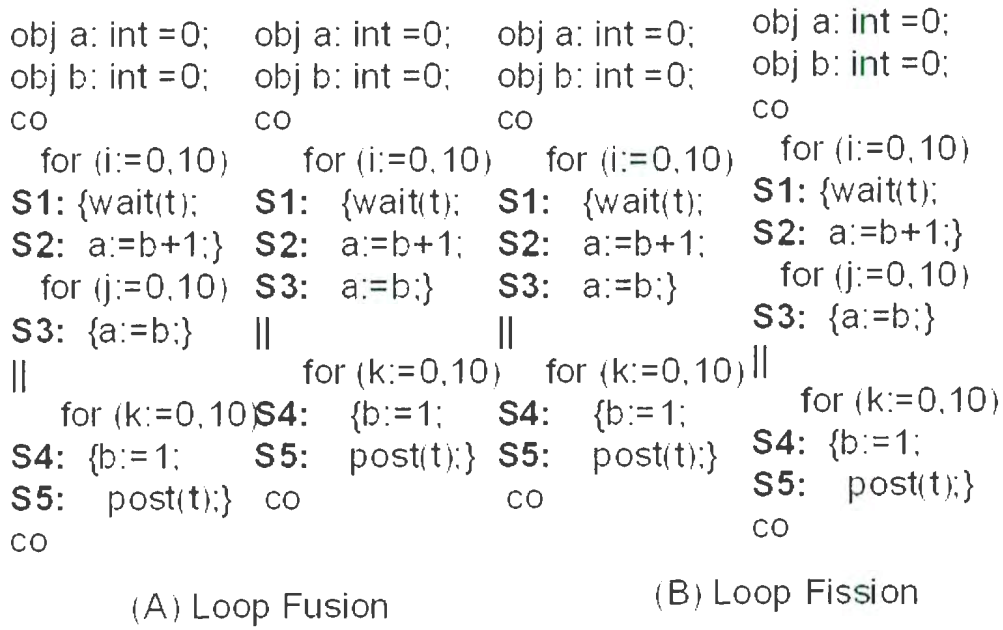


Figure 6.24: Loop Fusion and Loop Fission with a Synchronization Statement in Parallel Programs [31]

6.5.5 Loop Fusion and Loop Fission analysis in Parallel Programs

In parallel program, if some optimizations on loop fusion and loop fission violate the sequential consistency, these optimizations are unsuccessful. Usually these optimization transformations change the execution order of stores and fetches in the original program. In the sequential programs, all data dependences and control flows guarantee the transformation not to violate the sequential consistency. So these optimizations are safe in sequential programs. But in parallel programs, if there are no shared variables between different threads, the loop optimization is also safe, because this is similar to the sequential programs. Otherwise, we have to make more analysis

to ensure the sequential consistency in the parallel program.

In [29] D.Shasha and M.Snir proposed their method. In this method, for parallel program they used an instance level conflict graph. The nodes are statement instances connected by directed arcs representing the execution order of statement instances and undirected edges representing conflicts between statement instances. A mixed cycle is a cycle that contains both conflict edges and program arcs. A minimal mixed cycle is a mixed cycle C, who has no other mixed cycle C'. The authors summarized two conditions, which ensure sequential consistency:

-Conservative condition:

If the execution order represented by the program arcs contained in all mixed cycles is enforced then the execution of the program is sequentially consistent.

-Minimal condition:

Any set of program arcs whose enforcement ensures sequential consistency is a superset of the program arcs contained in minimal mixed cycles.

If an optimization shows that one or more program arcs involved in a minimal mixed cycle is not being enforced after the transformation is performed, the optimization can ensure sequential consistency and is not safe. In addition, in [31, 19] S.P.Midkif and D.A.Padua also introduce some useful methods.

We also use Fusion/fission methods for the analysis. We use an example in the Figure 6.25, which is similar to the one in the Figure 6.23. In the Figure 6.25 (A) a fusion was made in (1). Naturally in the range of this atomic operation, it is safe to combine two loop body. But in the (3) we split a atomic operation into two, this fission is ussafe, since S3 and S4 can introduce new definition of variable a and b. This fission isn't safe and this loop fusion isn't safe. In the (B), a loop fission is dis-

played. A fusion is in (1). In the range of atomic operation, a loop fission is safe. But in the (3), S3 and S4 in other thread introduce new definitions of variable a and b. This fission is unsafe and the loop fission is unsafe. Until now, we used fusion/fission method to identify the loop fusion and loop fission. The key problem is to how to keep the fusion/fission safe. This will be discussed in next Section.

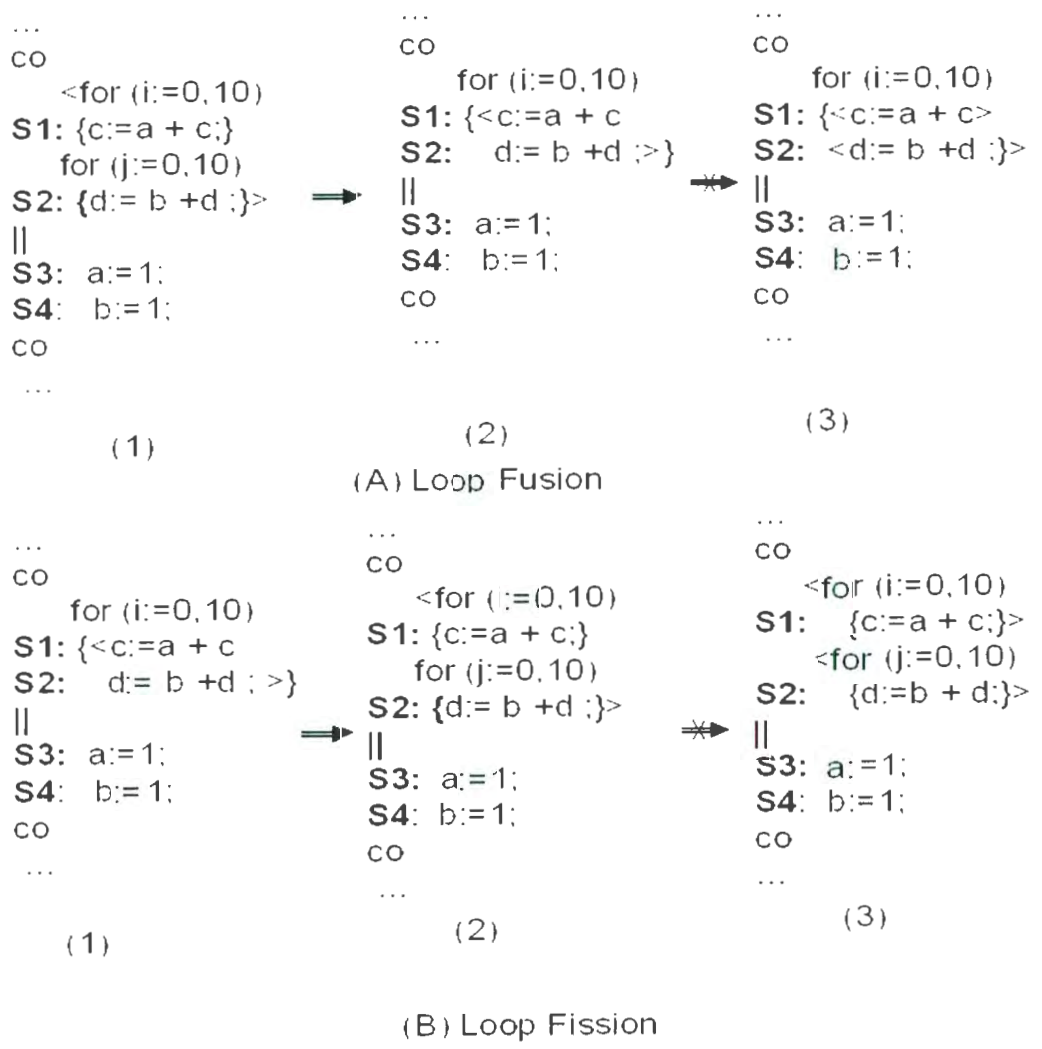


Figure 6.25: Atomic Fusion/Fission Analysis for Loop Fusion and Loop Fission in Parallel Programs

6.6 Atomic Fusion/Fission in Parallel Optimization

As shown in the preceeding sections, the atomic fusion/fission method can be used for identifying safe optimization in parallel programs. As far as we know this is original. In this section, we will discuss rules for safe atomic fusion/fission.

First, we need to know atomic fussion is always safe. We can regard each statement as an atomic operation in programs. When we make a fusion to these atomic operations, variable values in atomic operations mergeed must be and always be one of values of variables in original programs. So results of programs will be one of possible results in programs unchanged. This fusion only decrease the nondeterministic of programs. We can call this is safe. However, atomic fission is different. It can be safe or unsafe. An unsafe fission can introduce errors.

In order to identify safe fission, we need a rule to check it. We need to find conditions under which

$$C[< FG >] \parallel P \quad = \quad C[< F >< G >] \parallel P$$

In this formula, <> represents atomic operation. F and G mean statements, which are splitted. P represents all other threads. C represents a context. This formula represents a normal atomic fission in parallel programs. If we can prove the sides of this formula are equivalent, this atomic fission is safe.

We can define $P \subseteq Q$ as each trace of Q is equivalent to a trace of P and then $P \equiv Q$ as $P \subseteq Q$ and $P \supseteq Q$. We can show that

$$C[< FG >] \parallel P \quad \supseteq \quad C[< F >< G >] \parallel P$$

Each trace represents a sequence of operations, which can be inserted by other threads. The operations appear in the trace including read global memory, write global memory and local operations. Let T_F be some trace of $< F >$ and let T_G be a trace of $< G >$. T_P represents the operations inserted by other threads into between T_F and T_G. For *equivalence*, logically equivalent statements have the same logical content. Semantically equivalent have the same truth value in every model [33]. We define equivalent for traces as these traces can produce same results in this model.

We have this lemma:

Any trace T is equivalent to a trace UVW

where U involves only local operations reads of global variables from a set A, V involves only local operations and writes to variables from a set B, and W involves only local operations and reads from variables in a set $C \subseteq B$

Theorem:

Assume that for each variable x

- if G reads x then there is no parallel write of x that reaches the statement $\langle FG \rangle$ and
- if F writes x then that write reaches uses only in F and in G.

We have:

$$C[< FG >] \parallel P \quad \supseteq \quad C[< F >< G >] \parallel P$$

Proof:

Let $T_0\ T_F\ T_P\ T_G\ T_1$ be a trace of $C\left[\langle F \rangle \langle G \rangle\right] \| P$ in which T_F and T_G are traces

of $< F >$ and $< G >$ respectively, we need to show that there is an equivalent trace of $C\,[\langle FG \rangle]\,\|P$.

$$T_F \quad T_P \quad T_G$$

So for any T_P, where $T_P \equiv U\,\{A\} \quad V\,\{B\} \quad W\,\{C\}$ and $C \subseteq B$.

$$\equiv T_F \quad U\,\{A\} \quad V\,\{B\} \quad W\,\{C\} \quad T_G$$

When we have condition $RG \cap B = \emptyset$, RG means variables read by T_G. $RG \cap B = \emptyset$ means all variables read by T_G can not be written.

$$\equiv U\,\{A\} \quad T_F \quad V\,\{B\} \quad W\,\{C\} \quad T_G$$

When we have condition $WF \cap A = \emptyset$, WF means variables written by T_F. $WF \cap A = \emptyset$ means variables written by T_F can not be read.

$$\equiv U\,\{A\} \quad T_F \quad T_G \quad V\,\{B\} \quad W\,\{C\}$$

In the Figure 6.26 (A), the G is $a := b + 1$, and F is $b := 1$. For G, variable b can get value from S3 in the other thread. This doesn't satify the condition "there is no parallel write of x that reaches the statement". We can't get a safe-fission. However, in (B) the write in F only is limited in F and G. There are any variable written in other threads. We can get a safe-fission.

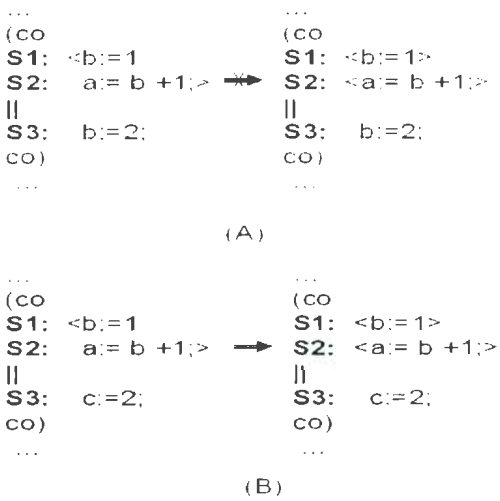


Figure 6.26: An Example to Use The Theorem For Safe-Fission Indentification

6.7 Summary

In this chapter, we first introduce program optimization and some differences between sequential and parallel programs. Then we used examples to introduce and explain differences of CSE, DCE, Code Motion and Loop Fusion and Loop Fission in parallel programs from in sequential programs. And we used a new method: atomic Fusion/Fission to check safe parallel optimizations. Lastly, we gave a rule to identify safe atomic Fusion/Fission and a proof for this rule.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

7.1.1 Thesis Summary

In this thesis, main issues are the IR design on the compiler of a new programming language *HARPO/L*, which can be executed in the *CGRA* hardware platform. We developed a sequence of procedures to transfer the program source code in the *HARPO/L* language into the appropriate data flow graph, which acts as the input of backend. The structure of the thesis is:

Firstly, we provided the fundamental introduction on *CGRA* architectures, Compiler theory and *HARPO/L* Language Specification.

The *CGRA* includes several different architectures, where connections, communication means, fabrics and components can be different. But we can think of them as consisting of many function units connected in various ways. These architectures can be categorized into linear-array, mesh and crossbar.

Then we introduced compiler theory. Modern compilers usually have three stages (front end, optimization and back end). In front end, the compiler performs parsing and lexical analysis and produces an abstract syntax tree (AST). After this front end, source codes have no more syntax and lexical errors. After that, the compiler can make optimizations. In the back end, the compiler produces the configuration files of target architectures from the IR. Back end activities include mapping, routing and scheduling.

In order to design the IR of the compiler of the *HARPO/L* language, we developed a new methodology for it. Firstly, we started this process from analyzing relations between objects in the *HARPO/L* language programs. We developed the object dependence graph for this purpose. Secondly, we made control flow analysis with augmented concurrent control flow graph for each skeleton complex objects, which can be obtained from an ODG. Thirdly, the ACCFGs were transformed into CSSA form. The CSSA form includes ϕ, ψ, π and ξ functions to represent various cases of data flow confluence. We separately introduced various function node placement methods. Fourthly, we considered the interprocedural relations among objects. Interprocedural analysis is a global view analysis and can be used for some optimizations. Traditional interprocedural analysis use the call graph and inline calling, but we used the information from ODG to make optimizations and prevent a call deadlock. Fifthly, we discussed the optimization behaviors in parallel programs. We individually discussed Common Subexpression Elimination(CSE), Dead Code Elimination (DCE), Code Motion, and Loop Fusion and Fission. The traditional sequential optimization algorithms cannot be directly used in parallel programs. Some modifications must be made. We developed Atomic Fusion/Fission method to identify

various safe optimizations in parallel programs.

7.1.2 Thesis Contributions

HARPO/L is a programming language, which executes on various hardware platforms, including CGRAs and microprocessors. The purpose of this thesis was to develop the IRs for the compiler of the *HARPO/L* language. The IRs can be used in various optimizations and as input to the compiler back end.

In this thesis, the main contributions were the following:

1. We proposed Object Dependence Graph (ODG). The ODG is used for representing the relations among objects in HARPO/L language programs. These objects can be divided into two categories. One is the simple object, and the other is the complex object. The relations include *part-of* and *knows*. We can find skeleton objects of the HARPO/L programs from it.

2. We extended Lee's CCFG [9] to Augmented Concurrent Control Flow Graph (ACCFG). The ACCFG is also a kind of control flow graph, but it adds more edges and more node types. The programs in ACCFG form are translated into the CSSA form. The CSSA is an extended form of Single Static Assignment (SSA), which is often used in the traditional sequential programs. In the SSA form, only the ϕ function is used for representing the control flow condition merging. But in the CSSA, Lee proposed two new functions ψ and π for representing the confluence of threads. We added a new function ξ for calling procedures in the *HARPO/L* language. The placement of this function was provided.

3. We made some interprocedural analyses and optimizations for the *HARPO/L* language. For some usual optimization techniques, such as dead code elimination, common subexpression elimination, and hoistable access detection, we gave their behaviors in parallel programs with instances. We developed the Atomic Fusion/Fission method to identify various safe optimization in the *HARPO/L* language. In addition, we discussed possible deadlock, which can come from parallel programs and calling procedure, and the solutions were provided.

7.2 Open Issues for Future Work

Although we provided a comprehensive study on the intermediate representation for the compiler of *HARPO/L*, there are still some aspects that can be improved in the future.

The Parallel Optimizations

In the Atomic Fusion/Fission method for optimization in the *HARPO/L*, we use atomic operations for current method. But we will extend it to further cases, such as semi-atomic operation.

The interprocedural optimizations

In this thesis, we discussed the two interprocedural optimizations on the elimination of unnecessary bounds checking and the simplifying of the same procedure calling. We hope to develop more interprocedural optimization techniques to improve the compiler's efficiency. **Test and compare**

Although we made extension and proposed new methods, we need to know the improvement from these IRs and methods. Some experimences need to be done.

Bibliography

[1] K. Y. Arvind Krishnamurthy. Optimizing parallel programs with explicit synchronization. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–204, 1995.

[2] A.V.Aho, R.Sethi, and J. Ullman. *Compilers:Principle, Techniques and Tools*. Addison Wesley, 1986.

[3] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng.*, 22(3):161–180, 1996.

[4] F. D. Deadlock detection without wait-for graphs. *Parallel Computing*, 17(1):1377–1383, 1991.

[5] M. E. Daniel Weise, Roger F. Crew and B. Steensgaard. Value dependence graphs: representation without taxation. *ACM*, pages Page:297–310, 1994.

[6] W. Haque. Concurrent deadlock detection in parallel programs. *Int. J. Comput. Appl.*, 28(1):19–25, 2006.

[7] M. W. Harini Srinivasan, James Hook. Static single assignment for explicitly parallel programs. *Annual Symposium on Principles of Programming Languages*, pages Page 260 272, 1993.

[8] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. *Int'l Conference on Design Automation and Testing in Europe*, pages 642 649, 2001.

[9] P. M. Jaejin Lee and D. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. *In Proceedings of The 10th International Workshop on Languages and Compilers for Parallel Computing*, pages 114 130, Aug 1997.

[10] P. M. Jaejin Lee and D. Padua. Basic compiler algorithms for parallel programs. *Principles Practice of Parallel Programming*, pages 1 12, April 1999.

[11] J. V. Jens Knoop, Bernhard Steffen. Optimal code motion for parallel programs. *Knoop, J., Steffen, B., and Vollmer, J. Optimal code motion for parallel programs. To appear in Proceedings of the 12 th Workshop on Alternative Konzepte fur Sprachen und Rechner, Physikzentrum Bad Honnef, Germany*, 2(4), May 1995.

[12] J.Ferrante and K.Ottenstein. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and System*, 9(3):319 349, July 1987.

[13] J. Knoop and B. Steffen. Code motion for explicitly parallel programs. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and*

practice of parallel programming, pages 13 24, New York, NY, USA, 1999. ACM Press.

[14] M. B. K.Pingali and R.Johnson. Dependence flow graphs: an algebraic approach to program dependencies. *Annual Symposium on Principles of Programming Languages*, pages 67 78, Jan 1991.

[15] S. M. Kurlander and C. N. Fischer. Minimum cost interprocedural register allocation. *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 230 241, 1996.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans*, pages Page 690 691, Sep 1979.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28:Page 690 691, 1979.

[18] H. M. Marc M. Brandis. Single-pass generation of static single assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):Page 1684 1698, 1994.

[19] S. P. Midkiff, D. A. Padua, and R. Cytron. Compiling programs with user parallelism. *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 402 422, 1990.

[20] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997.

[21] T. S. Norvell. The cgra language specification, 2006.

[22] M. N.Walkinshaw and M.Wood. The java system dependence graph. *Third IEEE International Workshop on Source Code Analysis and Manipulation*, page 55, Sept 2003.

[23] K. J. OTTENSTEIN. Data-flow graphs as an intermediate program form. *Ph.D. dissertation*, Aug 1978.

[24] J. R.Cytron. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):Page 451 490, 1991.

[25] M. C. Rinard. Analysis of multithreaded programs. *Proceedings of the 8th Static Analysis Symposium*, 2001.

[26] A. B. M. Robert A. Ballance and K. J. Ottenstein. The program dependence web: A representation supporting control,data, and deman-driven interpretation of imperative languages. *ACM*, pages Page:257 271, 1990.

[27] V. Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. *Proc. of the 10th International Workshop on Languages and Compilers for Parallel Computing, LNCS Springer-Verlag*, pages 94 113, 1997.

[28] V. Sarkar and B. Simons. Parallel program graphs and their classification. pages 633 655, 1994.

[29] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282 312, 1988.

[30] T. S.Horwitz and D.Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on programming Languages and System*, Vol.12:Page 26 60, January 1990.

[31] D. P. S.P. Midkiff. Issues in the compile-time optimization of parallel programs. *International Conference on Parallel Processing*, 1990.

[32] B. S. Vivek Sarkar. Parallel program graphs and their classification. *Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 633 655, Aug 1993.

[33] www.wikipedia.org. The free encyclopedia www.wikipedia.org.

[34] G. J. M. S. Yuanqing Guo. Mapping applications to a coarse grain reconfigurable system. *Lecture Notes in Computer Science*, 2823/2003:221 235, 2003.

[35] J. Zhao. Multithreaded dependence graphs for concurrent java programs. *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, 1999.