INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

A COMPILER OPTIMIZATION FRAMEWORK FOR CONCORDIA PARALLEL C

WEN LIANG

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

SEPTEMBER 1998

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Wen Liang**

Entitled: **A Compiler Optimization Framework for Concordia Parallel C**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair

_____

_____

_____

_____

_____ Supervisor

Approved _____
Chair of Department or Graduate Program Director

_____ 19 _____ _____
Dr. Nabil Esmail, Dean
Faculty of Engineering and Computer Science

Abstract

A Compiler Optimization Framework for Concordia Parallel C

Wen Liang

In this thesis, we present the design and implementation of a compiler optimization framework for the Concordia Parallel C (CPC). CPC is the working language for the Concordia Parallel Systems Simulator (CPSS), whose purpose is to study interactions between parallel runtime systems and languages and their impacts on performance.

The major challenge of designing a compiler optimization framework is to choose an appropriate Intermediate Representation (IR) that is able to support the solving of dataflow equations efficiently and applying corresponding transformations easily. The demanding issue in implementing such a framework is to make it general enough to meet the requirements of various dataflow analyses and transformations.

We choose Control Flow Graph (CFG) as our intermediate representation because it is a general and mature compiler IR. We developed a two-pass algorithm that can build CFG from the CPC Abstract Syntax Tree (CPC-AST). Based on CFG representation, we implemented an optimization framework that can solve dataflow equations in either forward and backward manner. As a demonstration example, we showed how to solve the Common Subexpression Elimination (CSE) problem using the optimization framework.

To make our optimization framework general yet efficient, we pay special attention on data structures used in our implementation. Sets are the common data structure used in describing dataflow equations and their solution algorithms. We use linked lists to represent sets and implement various set operations such as intersection, union and member on linked lists.

Our optimization framework is general yet efficient to solve various compiler optimization problems including CSE, live variable analysis, and reaching definition. The experimental results show that the implemented CSE optimization does improve program performance.

Acknowledgements

I would like to thank Prof. Lixin Tao for his thesis supervision for one year.

I am grateful to the professors in the Computer Science department especially, Dr. Lixin Tao, who taught me so many courses and offered me guidance, advice and encouragement, Dr.Butler, who gave an excellent course on Software Design Methodologies and Object-Oriented Design (COMP647), Dr. Opatrny, who gave wonderful courses on Discrete Structures and Formal Languages(COMP536) and Compiler Design(COMP 642). Thanks also go to the administrative assistants of the CS department, especially Ms. Halina Monkiewicz and Ms. Stephanie Robert. Their friendliness and administrative support have made graduate students' life much easier. Thanks must also go to the UNIX system administrators, who have provided superior system support and services.

Special thanks go to other committee members of my master thesis defence, Professor Yuke Wang and Professor Manas Saksena who spent a lot of time to read my thesis and gave me constructive critiques. Thanks also go to the chairman of my master thesis defence, Professor Klasa Stan who made the whole thesis defence go smoothly.

My good friends always deserve my appreciation. They have shared with me not only my accomplishments but also my frustration.

My family has been very supportive, especially my husband who give me his unconditional love and support. He helped to proofread this thesis and always gave constructive critiques, which make this thesis more readable and have fewer mistakes. I also thank my parents who always encourage me to overcome various difficulties in my life and study.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Rapid advances in VLSI technology have provided new challenges to compiler and architecture designers in the development of both uniprocessor and multiprocessor systems. In order to effectively exploit the ample resources provided by these new architectures, aggressive compilation techniques and innovative architecture designs are essential. New approaches in compiler technology are required to suit the different architecture design philosophies emerging today, from RISC machines to multiprocessors architectures. It is essential that compilation techniques and architecture models are developed together, so that the effects of one on the other can be studied.

In this thesis, we present the design and implementation of a compiler optimization framework for the Concordia Parallel C (CPC) [Tao96]. CPC is the working language for the Concordia Parallel Systems Simulator (CPSS) [Tao96], whose propose is to study interactions between parallel runtime systems and languages and their impacts on performance.

Uniprocessor performance has been increased dramatically (more than a factor of ten) since mid 80s. One of the main reasons of such success is the close interactions between compiler writers and computer architects. New architecture features such as *pipelining* [Kog81], *multiple issuing* [Fis83, CNO+87], and *branch predication* [MH86], have been exposed to compiler writers so that new compiler optimization techniques have to be developed to exploit those innovative architectural features.

The design of a good optimizing or parallelizing compiler is crucial in the development of high performance single and multi-processor architecture systems. An

optimizing compiler performs a series of code-improving transformations, such as constant propagation [WZ85], common subexpression elimination [ASU86], and instruction scheduling [GM86], before producing efficient machine code. In order to perform any sort of optimizing transformation, it is essential to collect accurate information about the variables used in the program. Data-flow analysis is a process of collecting information about definitions and uses of variables in a program. Typical examples of traditionally performed data-flow analyses are *reaching definitions*, *live-variable* analysis, and *last-use* information. A different kind of analysis that is receiving an increasing attention is alias and array dependency analysis[Bar78, Ban79, CK89, LR92]. Optimizing compilers make use of data-flow and alias information to produce efficient code. Furthermore, parallelizing compilers need this information to extract parallel threads from a sequential program.

Intra-procedural data-flow analysis, i.e., analyzing one procedure at a time [ASU86], has been widely studied and implemented in existing compilers. Gathering information about many interacting procedures, known as inter-procedural analysis, is essential to accurately analyze large programs. However, in this thesis we focus on intra-procedural analysis since it is a base for inter-procedural analysis.

The major challenge of designing a compiler optimization framework is to choose an appropriate Intermediate Representation (IR) that is able to support for solving data-flow equations efficiently and applying corresponding transformations easily. The demanding issue in implementing such a framework is to make it general enough to meet the requirements of various data-flow analysis's and transformation's.

1.1 Thesis Contributions

This thesis concentrates on the development of a compiler optimization framework for CPC.

The first important contribution is the selection of an intermediate representation suitable for various high-level analyses and optimizations. A major portion of the analyses and optimization transformations takes place on the intermediate code. Thus the appropriate choice of an intermediate representation is vital in the design of an optimizing compiler. We choose Control Flow Graph (CFG) as our intermediate representation because it is a general and mature compiler IR.

The second contribution is the development of a two-pass algorithm that builds a CFG from CPC Abstract Syntax Tree (CPC-AST). Traditionally, an intermediate representation consists of three-address statements and control flow graphs[ASU86]. The program represented by these three address statements is partitioned into basic blocks, where each basic block consists of a sequence of consecutive statements with no branches in between. The optimizations are performed on control flow graphs, in which the edges represent flow of control and the nodes represent basic-blocks. CPC-AST is not a three-address format and the *goto* statement is allowed in CPC-AST. In the two-passes algorithm, we use the structural analysis technique to build a CFG for each CPC-AST construct such as *if*, *for-loop*, and *while-loop*. During this first pass, the potential targets of *goto* statement have been collected and remembered in a goto target table. Then, in the second pass, the *goto* statement can search the table to find out its destination basic block.

The third contribution is that we have implemented an optimization framework based on the CFG representation, which can solve data-flow equations in either *forward* and *backward* manner. As a demonstration example, we will show how to solve the Common Subexpression Elimination (CSE) problem using the optimization framework. However this optimization framework is general yet efficient to solve other compiler optimization problems. *Live variable analysis*, and *reaching definition* are just named as a few.

To make our optimization framework general yet efficient, we pay special attention on data structures used in our implementation. Sets are the common data structure used in describing data-flow equations and the related algorithms. We use linked lists to represent sets and implement various set operations such as *intersection*, *union* and *member* on linked lists.

1.2 Thesis Organization

The rest of the thesis is organized as follows.

In chapter 2, we survey related work on building compiler infrastructures, concentrated on the compiler optimization framework. In chapter 3, we overview the CPPE environment, concentrated on the structure of CPC-AST. In chapter 4, we discuss criteria that compiler optimizations should obey and propose the problem that this

thesis is interested in. In Chapter 5 we present our CFG building algorithm, which is based on a structural analysis technique to construct a CFG for each CPC syntax construct. In Chapter 6, we describe an iterative dataflow analysis framework which contains a general dataflow equation solver that is able to solve a variety of dataflow analysis problems. In Chapter 7, we illustrate the effectiveness of the framework by applying it to solve an optimization problem, common subexpressions elimination (CSE). In Chapter 8, we present the experimental results by applying the CSE optimization on a suite of benchmarks and analyze the impact of such an optimization on program performance. Finally, we conclude with summary and future work in chapter 9.

Chapter 2

Literature Survey

Research in automating the analysis and optimization phases of a compiler is still in its early stages. Few tools exist to help build optimizer, which are usually large and complex, since they must perform many program transformations to get the best code.

Sharlit[TH92] is a system which is designed to simplify building of optimizers in compilers. Sharlit merges the data-flow collection phase and the optimization phases; it takes in a specification and performs *one* type of data-flow analysis and a code-transformation that relies on that analysis. This works on the traditional flow-graphs and basic-blocks. Sharlit uses the following abstractions to develop global analyses and optimizations in a modular fashion.

- The nodes of the flow graph.
- Values that flow through the flow graph.
- Flow functions that represent the effect of flow graph nodes and paths on the flow values.
- Action routines to perform the optimization.
- Rules to combine the flow functions to other flow functions for path simplification.

Sharlit performs intra-procedural analyses and optimizations on flow-graphs. The data-flow analyses consists of four major components:

1. the control flow analysis that summarizes the structure of the flow graph.

2. the path simplifier, generated from the path-simplification rules in the input description and uses control flow information to eliminate some flow nodes.

3. the iterator makes use of the flow functions and iterates to find a solution for the data-flow equations.

4. the propagator that uses the action routines to perform the optimization.

Whitfield and Soffa[WS91] describe the automatic generation of global optimizers. They have introduced a General Optimization Specification Language (GOSpel) and an optimizer **gen**erator (GENesis) that is used to create global optimizers from compact, declarative specifications made in GOSpeL. The specifications mainly consist of a set of preconditions and the actions to optimize the code. The preconditions, in turn, consist of the code pattern to match and the global dependence information (i.e., the control and data dependencies that are required for the specific optimization). The actions take the form of primitive operations that make up the optimization transformation. GENesis analyzes GOSpeL specifications and produces the optimizer. It first produces code (i) for the data structures defined, (ii) for matching the required code pattern, (iii) for checking if the particular data dependences hold, and (iv) for performing the required optimizing transformations. Thus, unlike Sharlit, they do not generate the data-flow analyzer along with the optimizer, but they do assume the data-flow information such as anti, output and flow dependence relations are already computed and available. Further, they also work on flow-graphs, and do not perform any inter-procedural optimization.

In the MUG2 Compiler Generating System[GGMW82], Wilhelm describes separate analyses and optimization phases which work on a structured abstract syntax tree intermediate representation. Global data-flow analysis is specified using modified attribute grammars and the abstract syntax tree, decorated with the data-flow information, is called an *attributed program tree.* In a single analysis pass, which may be made up of several semantic analysis passes, global data-flow information is collected as attributes associated with nodes in a program tree. Attributes are classified as either *derived* or *inherited*, and they are evaluated according to the rules specified for every different kind of node. The optimization passes are implemented as tree

transformations, and could also update the data-flow information present in the tree nodes.

Vortex is an optimizing compiler infrastructure for object-oriented and other high-level languages [DDG+96]. It targets both pure object-oriented languages like Smalltalk and hybrid object-oriented languages like C++ and Java. Vortex currently incorporates high-level optimizations such as static class analysis, class hierarchy analysis, automatic inlining. It also includes a collection of standard intra-procedural analyses such as common subexpression elimination.

A central piece of supporting infrastructure in the Vortex compiler is its iterative dataflow analysis (IDFA) framework. All of the analyses and transformations rely on this framework to manage the details of iterative dataflow: control flow graph traversal, merging dataflow information at control flow merges, fixed-point convergence testing for loops, and graph transformations. (in spirit, Vortex's IDFA is similar to the Sharlit system.)

McCAT is a compiler environment developed for researching compiling issues for C programs [HDE+92], especially for high-level dataflow analyses such as points-to (alias) and heap analysis. The intermediate representation used in McCAT is called SIMPLE, in which each statement is simplified into a three address format while high-level structures such as array reference and pointer deference, and high-level control structures such as loops and conditionals, are still retained. Many high-level dataflow analyses have been implemented on McCAT, including constant propagation, reaching definition, alias analysis, array dependence analysis, read/write set analysis, and function inlining.

The SUIF compiler infrastructure is a part of the national compiler infrastructure project (USA). The infrastructure is based on the SUIF (Stanford University Intermediate Format) parallelizing compiler developed at Stanford University [AALL93].

The primary objective in the SUIF compiler design is to develop an extensible system that supports a wide range of current research topics including parallelization, object-oriented programming languages, scalar optimizations and machine-specific optimizations. The SUIF group strives to develop an architecture that is modular, easy to extend and maintain, and supportive of software reuse. By adopting object oriented programming techniques, each dataflow analysis can be added into the SUIF environment independently. Many dataflow analyses have been implemented on SUIF, among them array dependence analysis is their most famous research work.

Even though McCAT and SUIF have implemented many dataflow analyses, there is no general iterative analysis solver on both systems.

In our framework, we took a less ambitious approach in which we separate analysis and transformation phases. It is the user's responsibility to provide C routines to do certain analysis and transformations since the techniques are still immature to automatically generate such routines.

Recently, Olaf Chitil have developed a version of CSE for a lazy functional programming languages and found several advantages. First, the referential transparency of these languages makes the identification of common subexpression very simple. Second, more common subexpression can be recognized because they can be of arbitrary type whereas standard common subexpression elimination only shares primitive values. However, because lazy functional languages decouple program structure from data space allocation and control flow, analyzing its effects and deciding under which conditions the elimination of a common subexpression is beneficial proves to be quite difficult. They developed and implemented the transformation for the language Haskell by extending the Glasgow Haskell compiler and measured its effectiveness on real-world programs.

We implemented the CSE optimization based our general iterative analysis solver. Our CSE optimization is applied to an imperative language based on C.

Chapter 3

Introduction to Concordia Parallel C

In this thesis, we present the design and implementation of a compiler optimization framework for the Concordia Parallel C.

Concordia Parallel Programming Environment (CPPE) consist of two parts: Concordia Parallel Systems Simulator (CPSS) and Concordia Parallel C (CPC).

CPPE is a useful tool to identify performance bottlenecks of parallel code, study the interaction of various system components of a parallel system, and investigate the interaction between compiler optimizations and system performance. CPPE is also a self-contained programming environment in which students can learn and practice parallel programming. CPPE contains about 500K well documented C source code and runs on most platforms including PCs and workstations. On Unix and Windows-NT it can simulate more than 4000 parallel processors. As part of the prototype CPPE also designed a special wormhole routing simulator for on-line network simulation.

CPSS is a compact performance debugger prototype, designed for researching on performance of parallel systems. CPSS performs simulation of various parallel systems and their communications subsystems.

The CPC compiler includes:

- a compiler frontend, transforming code in the Concordia Parallel C (CPC) to an abstract syntax tree;
- a code generator, generating intermediate code for the Concordia Parallel Systems Simulator (CPSS).

Our work is to add an optimization framework between the frontend and the code generator so that code generated can be run or simulated fast. In the following, we will describe the CPC abstract syntax tree (AST), which is the output of the front-end and the input of our optimization framework.

3.1 Concordia Parallel C

The CPC (Concordia Parallel C) language is based on the popular programming language C [KR88] and enhanced with new features to support parallel programming. The CPC language supports both shared-memory and message-passing programming paradigms. Parallel features of CPC support the creation of parallel processes, the definition of virtual parallel architectures, process communications through channel variables and mapping of parallel processes to virtual processors. CPC preserves existing sequential features of the C language.

User will write portable parallel programs on virtual architectures of his choice in an extended version of C (CPC), which supports MPI communications standard and high-level channel abstractions of message passing. The CPC compiler will translate the source code into an abstract syntax tree, and generate either simulation code for CPSS or object code for particular parallel systems. For our research, we add one component into the CPC compiler, which can also dump *sequential* C code from a CPC-AST tree.

3.2 CPC AST

The structure of CPC-AST is a structured intermediate representation chosen for CPC.

The main strength of CPC-AST is its simple yet expressive structure. Complete C source code for its implementation as well as its executable are within 200K bytes, and can run on any system supporting C compiler, lex and yacc (or their compatibles). CPC-AST has a syntax matching well to the ANSI definition. All information in the source code are represented by four types of nodes and their interconnection.

CPC-AST is designed for portability. It is a shared front-end for CPC, from which code can be generated for different parallel machines. The program written in

CPC is architecture-independent. To run the existing code in CPC on a new parallel machine, one only needs to add a new backend for that machine to generate code for that machine from CPC-AST.

CPC-AST is designed for research. Because of its concise yet expressive nature, CPC-AST is strongly typed. While much of important information is explicit, new implicit information can be easily derived from CPC-AST with minimum effort.

CPC-AST is designed for easy usage. All information, including type chains, symbol tables, structure tables, and internal code representation can be visualised on display and traversed at will.

3.2.1 AST Nodes

In AST all information in CPC source programs is represented in four types of nodes and in the interconnection among these nodes. These four types of nodes are *symbol* nodes, *link* nodes, *value* nodes, and *statement* nodes.

Figure 1 draws four nodes used in the CPC to represent the four different types of nodes.

- Symbol nodes represent identifies, which may be a variable, a function, a user-defined type, an enumeration element, or a bit field.
- Link nodes are used to represent type structure.
- Value nodes represent expressions made up of constants or occurrences of symbols in the statement of the code.
- Statement nodes represent the statements in C.

3.2.2 AST Structure

In the CPC-AST, a compound statement has three components: `symbol_list`, `struc_list`, and `stmt`.

- `symbol_list` points to a linked list of all symbols for variable and typedefed new types. Theses symbols can be declared either at the beginning of a compound statement or outside any function with the scope of whole program (source file).

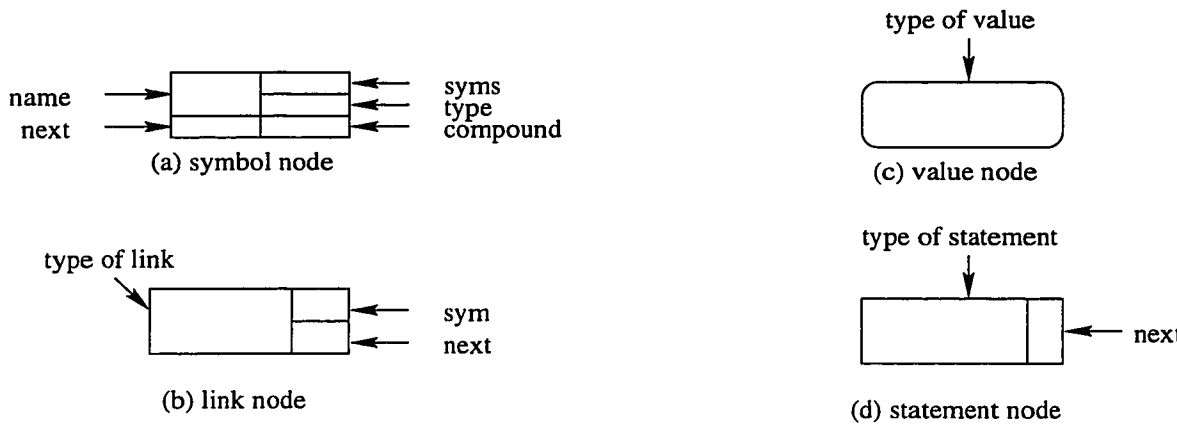


Figure 1: Graphical representation of nodes

All variables & structures, global or local, are inserted into the hash table when they are under parsing. And then they are unhooked (deleted) from the hash table. At end, there is not any information for AST in hash table.

- `struc_list` points to a linked list of all symbols for structures, unions, and enumerations declared at the beginning of this compound statement or outside any function for whole program (global structures).
- `stmt` points to a linked list consists of a sequence of statements of a compound statement.

Let's take a close look at how a function is presented in CPC-AST. Figure 2(a) lists simple C code and Figure 2(b) lists its AST tree.

Function `f` is represented as a *symbol* node, in which there are three fields pointing to:

- a *function* link node, followed by the return type of the function;
- a *parmeter* list;
- a *compound* node, pointing to a compound of statements.

The compound statement consists of three lists: statements, symbols, and structures. By following the *stmt* link, we can find a sequence of statements which form the body of the compound statement. The sequence is composed by *expr* node in Figure 2.

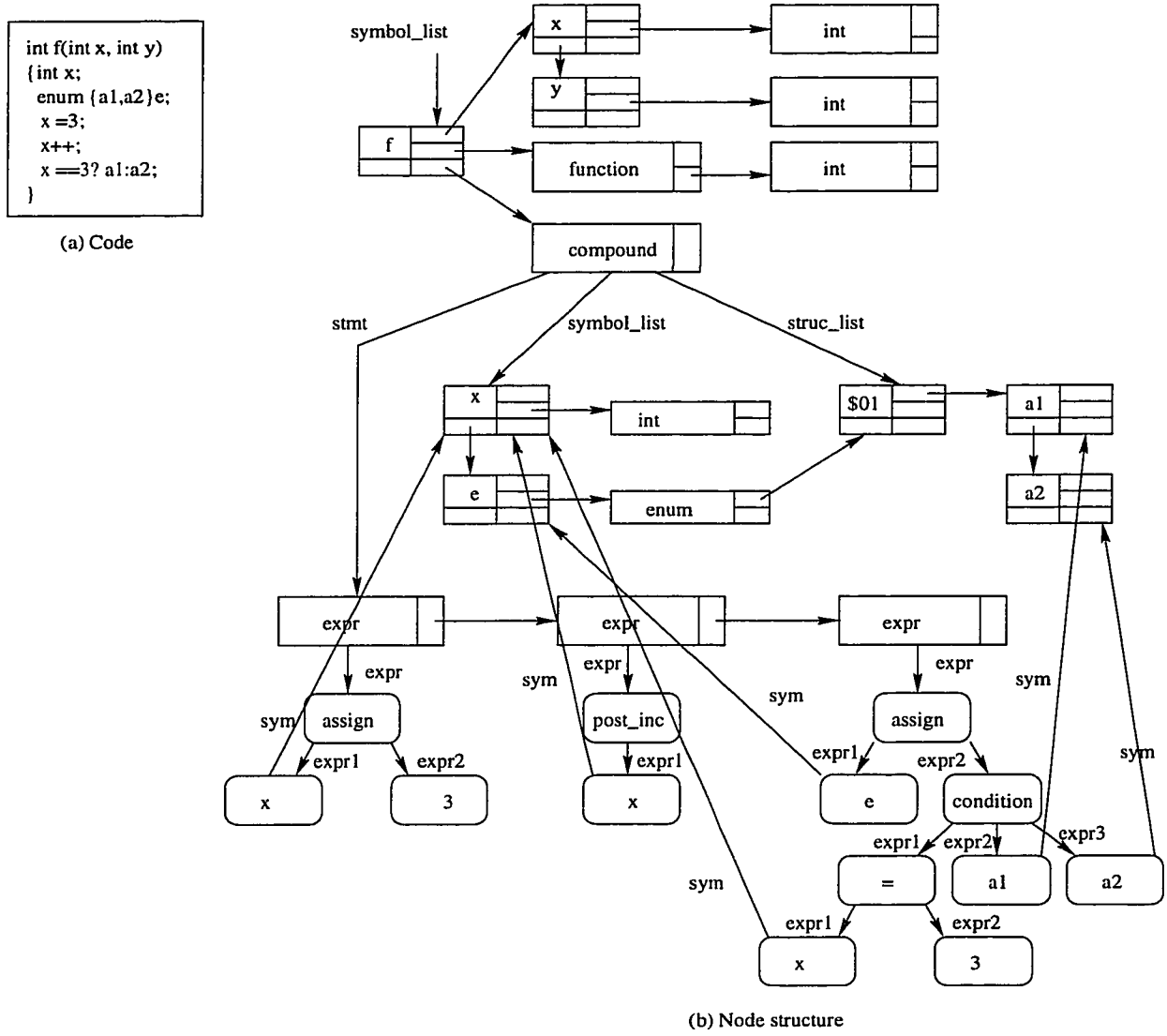


Figure 2: A compound statement

To find an expression statement, we should follow a list of *expr* nodes headed by `stmt`. In an *expr* node, field `expr` points to *value* node representing an expression.

For a *value* node, it can be any C expression type such as assignment, post-increment. The operands of an expression are presented by `expr1` and `expr2` fields, pointing to the related operand symbols.

In our implementation, we need to follow from the top-level symbol list to enter a function and then the top-level compound node of the function to find a statement list. Then we need to follow from the statement list to find each statement to build CFG for the function. To compute common subexpressions, we need to go down another level, tracing the *expr* node from a statement node to compute available expressions for each statement.

Chapter 4

Optimizations, Properties and Problem Statement

To create an efficient target language program, a programmer needs more than an optimizing compiler. In this section, we will describe criteria for optimizing transformations and specify the compiler optimization problems that are most interesting to us.

4.1 Criteria for Compiler Optimizations and Transformations

Optimization may be valuable in improving the performance of the object code produced by a compiler. Before going into some details, let's discuss the meaning of the word "optimize" in terms of compiler optimizations and transformations. We must point out that "optimization" is a misnomer–only very rarely does applying optimizations to a program result in object code whose performance is *optimal*, by any measure. Rather, optimizations generally *improve* performance, sometimes substantially, although it is entirely possible that they may decrease it or make no difference for some (or even all) possible inputs to a given program. In fact, like so many of the interesting problems in computer science, it is formally undecidable whether, in most cases, a particular optimization improves (or, at least, does not worse) performance. Some simple optimization, such as algebraic simplifications, can slow a program down only in the rarest cases (e.g., by changing placement of code in a cache memory so

as to increase cache misses), but they may not result in any improvement in the program's performance either, possibly because the simplified section of the code could never have been executed anyway.

We expect that the best program transformations are that yield the most benefit for the least effort. The transformations provided by an optimizing compiler should have several properties.

First, a transformation must preserve the meaning of programs. That is, an "optimization" must not change the output produced by a program for a given input, or cause an error, such as a division by zero, that was not present in the original version of the source program. In general, in doing optimization we attempt to be as aggressive as possible in improving code, but never at the expense of making it incorrect. To describe the latter objective of guaranteeing that an optimization does not turn a correct program into incorrect one, we use the terms *safe* or *conservative*. Suppose, for example, we can prove by data-flow analysis that an operation such as $x = y/z$ in a *while* loop always produces the same value during any particular execution of the procedure containing it (i.e., it is loop-invariant). Then it would generally be desirable to move it out of the loop, but if we cannot guarantee that the operation never produce a divide-by-zero exception, than we must not move it, unless we can also prove that the loop is always executed at least once. Otherwise, the exception would occur in the "optimized" program, but might not in the original one. Alternatively, we can protect the evaluation of y/z outside the loop by a conditional that evaluates the loop entry condition. The influence of this criterion is so important that at all times we take the "safe" or "conservative" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

Second, a transformation must, on the average, speed up program by a measurable amount. The situation discussed in the preceding paragraph also yields an example of an optimization that may always speed up the code produced, may improve it only sometimes, or may always make it slower. Suppose we can show that z is never zero. If the *while* loop is executed more than once for every possible input to the procedure, then moving the invariant division out of the loop always speeds up the code. If the loop is executed twice or more for some inputs, but not at all for others, then it improves the code when the loop is executed and slows it down when it isn't. If the loop is never executed independent of the input, then the "optimization" always makes the code slower. Of course this discussion assumes that other optimizations,

such as instruction scheduling, don't further rearrange the code.

Not only is it undecidable what effect an optimization may have on the performance of a program, it is also undecidable whether an optimizations is applicable to a particular procedure. Although properly performed control- and data-flow analyses determine cases where optimizations do apply and are safe, that cannot determine all possible such situations.

In general, there are two fundamental criteria that decide which optimizations should applied to a procedure (assuming that we know they are applicable and safe), namely, *speed* and *space*. Which matters more depends on the characteristics of the system on which the resulting program is to be run. If the system has a small main memory and/or a small cache, minimizing code space may be very important. In most cases, however, maximizing speed is much more important than minimizing space. For many optimizations, increasing speed also decreases space. On the other hand, for others, such as unrolling copies of a loop body, increasing speed increase space. Other optimizations, such as tail merging, always decrease space at the cost of increasing execution time. For each individual optimization, it is important to consider its impact on speed and space.

Third, a transformation must be worth the effort. It does not make sense for a compiler writer to spend the intellectual effort to implement a code-improving transformation and to have the compiler spend the additional time compiling source programs if this effort is not repaid when the target programs are executed. Some transformations can only be applied after detailed, often time consuming, analysis of the source program, so there is little point in applying them to programs that will be run only a few times. For example, a fast, non optimizing, compiler is likely to be more helpful during debugging or for "student jobs" that will be run successfully a few times and thrown away. Only when the program in question takes up a significant fraction of the machine's cycles does improved code quality justify the time spent running an optimizing compiler on the program.

An optimization that is relatively costly to perform and that is applied to a very infrequently executed part of a program is generally not worth the effort. Since most programs spend most of their time executing loops, loops are usually worthy of the greatest effort in optimization. Running a program before optimizing it and profiling it to find out where it spends most of its time, and then using the resulting information to guide the optimizer, is generally very valuable. But even this needs to be done with

some caution: the profiling needs to be done with a broad enough set of input data to run the program in a way that realistically represents how it is used in practice. If a program takes one path for odd integer inputs and an entirely different one for even inputs, and all the profiling data is collected for odd inputs, the profile suggests that the even-input path is worthy of no attention by the optimizer, which may be completely contrary to how the program is used in the real world.

Fourth, the transformation order applied to a program matters. It is generally true that some optimizations are important than others. Thus, optimizations that apply to loops, global register allocation, and instruction scheduling are almost essential to achieving high performance. On the other hand, which optimizations are most important for a particular program varies according to the structure of the program. For example, for programs written in object-oriented language, which encourage the use of many small procedures, procedure integration(which replaces calls to procedures by copies of their bodies) and leaf-routine optimization (which produces more efficient code for procedures that call on others) may be essential. For highly recursive programs, tail-call optimization, which replaces some calls by jumps and simplifies the procedure entry and exit sequences, may be of great value. For self-recursive routines, a special case of tail-call optimization called tail-recursion elimination can turn recursive calls into loops, both elimination the overhead of the calls and making loop optimization applicable where they previously were not. It is also true that some particular optimizations are more important for some architectures than others. For example, global register allocation is very important for machines such as RISCs that provide large numbers of registers, but less so for those that provide only a few registers.

4.2 Framework for optimizations and Transformations

Dramatic improvements in the running time off a program – such as cutting the running time from a few hours to a few seconds – are usually obtained by improving the program at all levels, from the source level to the target level, as suggested by Figure 3. At each level, the available options fall between the two extremes of finding a better algorithm and of implementing a given algorithm so that fewer operations

are performed.

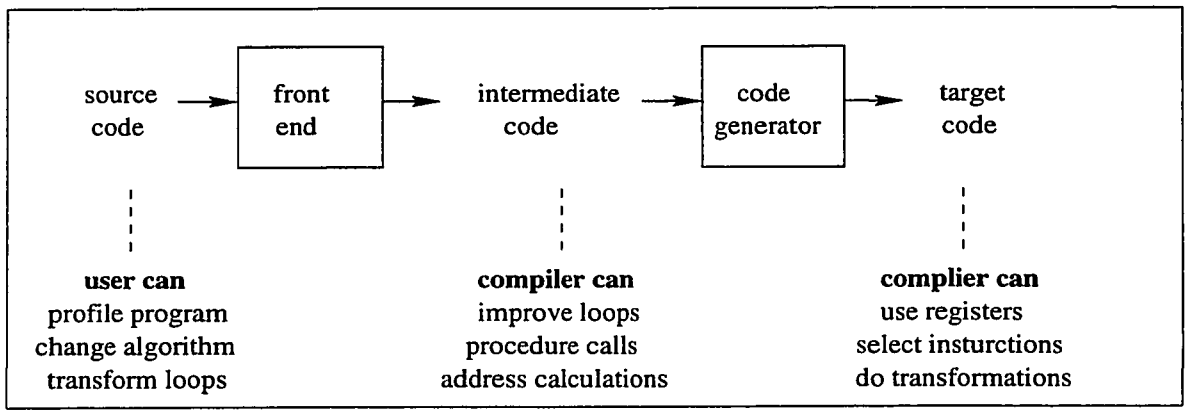


Figure 3: Places for potential improvements by the user and the complier.

We are interested in the techniques needed to analyze and transform a program at an intermediate level since at this level of abstraction transformations are independent on both high-level programming languages and low-level machine details. Figure 4 gives the overview of such a code optimization and transformation framework, The code-improvement phase consists of control-flow and data-flow analysis followed by the application of optimizing transformations. The code generator produces the target program from the transformed intermediate code.

Our work is to add such a framework on top of the CPC compiler so that the compiler optimization framework will perform necessary analyses and optimizing transformations. We use the existing CPC code generator to emit CPSS code. We also develop a "code generator" to produce C code to verify the code correctness.

The organization in Figure 4 has the following advantages:

1. The operations needed to implement high-level constructs are made explicit in the intermediate code, so it is possible to optimize them. For example, array address calculation may be exposed explicitly so that common subexpression may be detected in computing the address.

2. There exist many good optimization algorithms based on graph theory, which can be used in building such a framework. For example, all loops can be translated into cycles in terms of graph theory. Thus, optimizing cycles is equivalent

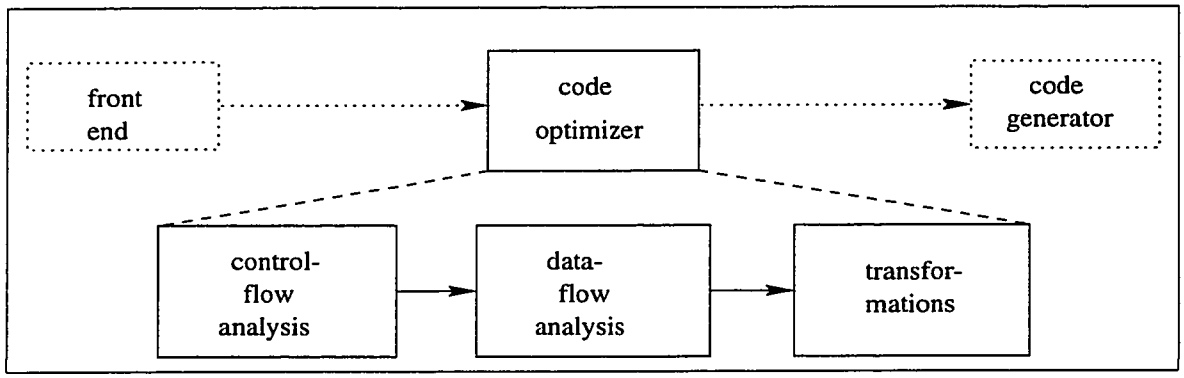


Figure 4: Organization of the code optimizer.

```
void quicksort(m,n)
{
  int i,j;
  int v, x;

  if (n<=m) return;
  /* fragment begins here */
  i = m-1; j = n; v = a[n];
  while(1) {
    do i = i-1; while(a[i]<v);
    do j = j-1; while(a[j]<=v);
    if (i >= j) break;
    x = a[i]; a[i] = a[j]; a[j] = x;
  }
  x = a[i]; a[i] = a[n]; a[n] = x;
  /* fragment ends here */
  quicksort(m,j); quicksort(i+1,n);
}
```

(a). C code for quicksort

```
(1) i = m-1              (16) t7 = 4*i
(2) j = n                (17) t8 = 4*j
(3) t1 = 4*n             (18) t9 = a[t8]
(4) v = a[t1]            (19) a[t7] = t9
(5) i =i+1               (20) t10 = 4*j
(6) t2 = 4*i             (21) a[t10] = x
(7) t3 = a[t2]           (22) goto(5)
(8) if t3<v goto (5)     (23) t11 = 4*i
(9) j = j-1              (24) x = a[t11]
(10) t4 = 4*i            (25) t12 = 4*i
(11) t5 = a[t4]          (26) t13 = 4*n;
(12) if t5>v goto(9)     (27) t14 = a[t13]
(13) if i>=j goto(23)    (28) a[t12] = t14
(14) t6 = 4*i;           (29) t15 = 4*n;
(15) x = a[t6]           (30) a[t15] = x
```

(b). Three-address code

Figure 5: C program and its three-address representation.

to optimize any loop.

3. The intermediate code can be (relatively) independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for a different machine.

In the code optimizer, programs are represented by control flow graphs, in which edges indicate the flow of control and nodes represent basic blocks (see Chapter 5). Unless otherwise specified, a program means a single procedure.

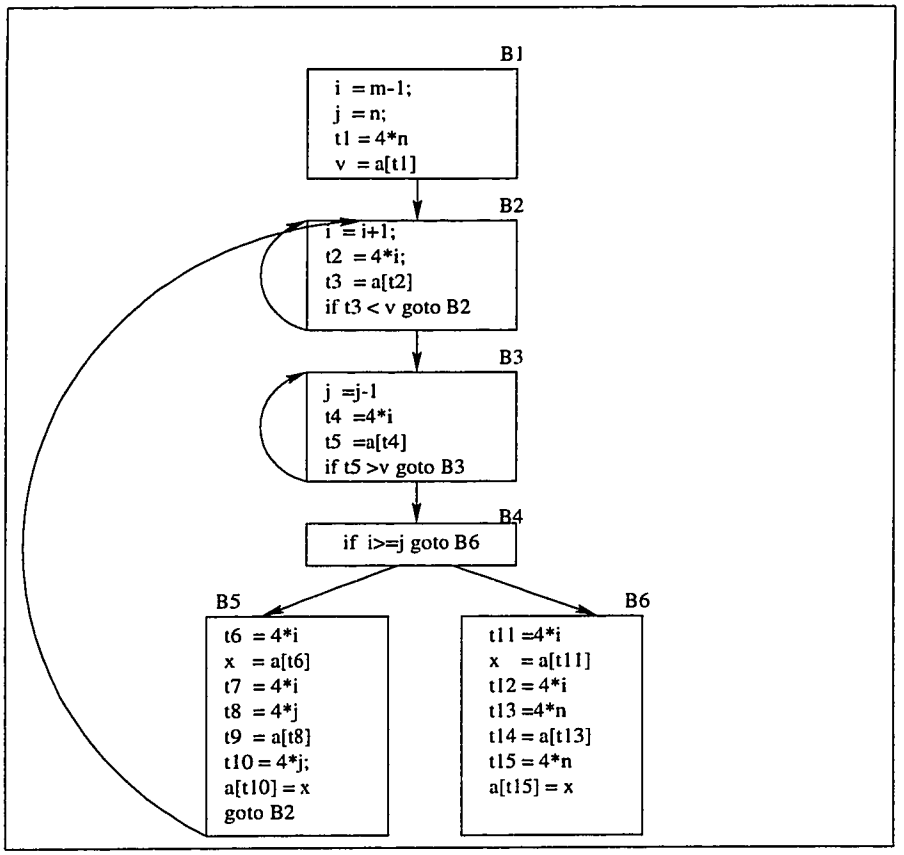


Figure 6: Control flow graph of the C program

Figure 5(a) is a C program and Figure 5(b) lists its three-address format. Figure 6 contains the control flow graph for the program in Figure 5. B1 is the initial node. All conditional and unconditional jumps to statements in Figure 5 has been replaced in Figure 6 by the jumps to the block of which the statements are leaders.

In Figure 6, there are three loops. B2 and B1 are loops by themselves. Blocks B2, B3, B4 and B5 together from a loop, with entry B2.

The control flow graph showed in Figure 6 is selected as our compiler framework intermediate representation, since it is used pervasively in most of compiler optimization frameworks. The problem interested in this thesis is to build a compiler optimization framework based on such control flow program representation.

The problem we are interested in is to build a compiler optimization framework, which can be used to study compiler transformations and their impacts on program performance, especially the influences on CPSS. Since this framework is targeted for CPSS in which Concordia Parallel C (CPC) is its working language, CPC is naturally chosen as the input programming language of our framework. CPC is a dialect of C, with extensions to express the concepts of parallelism, communication, and synchronization used in parallel processing. In this thesis study, the optimizations are targeted to the *sequential* part of CPC, which is mainly C part of CPC.

Analyzing a C program and collecting *accurate* data-flow information is still a dream of human being. In reality, only *conservative* estimated information can be collected from the program. This problem becomes even complicated when C *pointer* and *array* variables are involved. The *alias* analysis is a process to compute whether two pointer variables point to the same memory location. The *array dependence* analysis is a process to compute whether two array references are actually visiting the same memory location. The techniques of performing the alias analysis and array dependence analysis are under active study within the academic world. Implementing such analyses need tremendous man-power which is beyond the scope of this project. Besides our work is the first step towards building a general optimization framework, our analyses and transformations will be concentrated on C *scalar* variables.

There are two scopes of data-flow analyses used in compiler optimizations, *intra-procedural* and *inter-procedural* analyses. The latter analyzes the interactions among procedures and may expose more opportunities to optimize code at the cost of more expensive analysis techniques. Intra-procedural analysis is also called *global* analysis, which deals with the scope of a single function body. In this thesis, we focus on the intra-procedural analysis since it is the base to build a framework for inter-procedural analysis. Furthermore, many transformations are independent of which analysis is used since what they rely on is the results of the dataflow analyses. Thus even though we implemented our framework based on the intra-procedural analysis, we

have made our design open enough to contain a future inter-procedural analysis.

In conclusion, our design goal is to build a general optimization framework that supports an intra-procedural analysis for scalar variables and later we can apply the corresponding transformations based on the data-flow analyses gathered.

As a demonstration example, we implemented the *common subexpression elimination* (CSE) transformation within our framework to show the generality and effectiveness of the framework.

4.3 An Example of Function-Preserving Transformations – Common Sub-expression

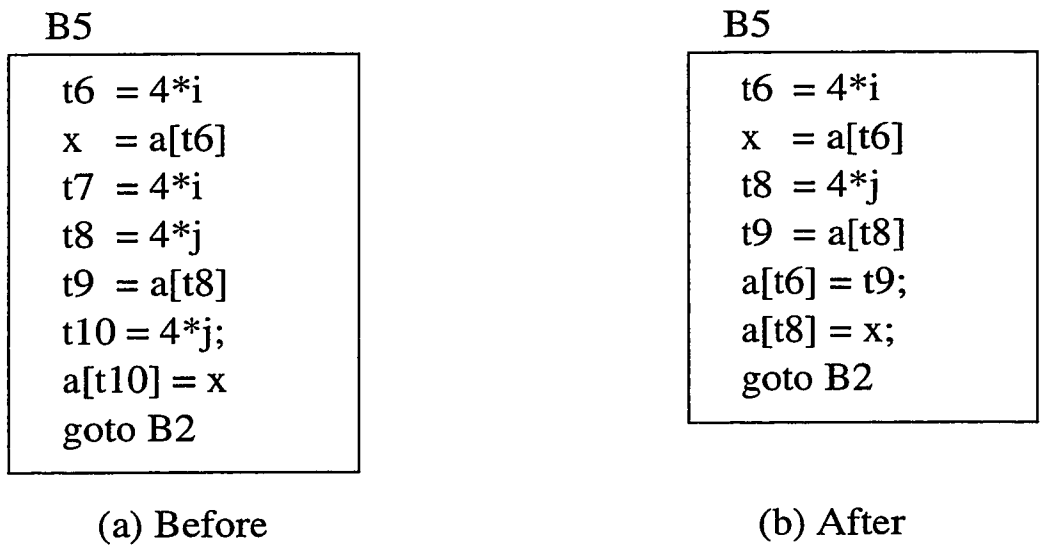


Figure 7: Local common subexpression elimination.

There are a number of ways in which a complier can improve a program without changing the function it computes. Common subexpression elimination is a common example of such function-preserving transformations.

Frequently, a program will include several calculations of the same value. Some of these duplicate calculations cannot be avoided by the programmer because they lie blow the level of detail accessible within the source language. For example, block B5 shown in Figure 7(a) recalculates *4*i* and *4*j*.

An occurrence of an expression E is called a *common subexpression* if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value. For example, the assignments to *t7* and *t10* have the common subexpression *4*i* and *4*j*, respectively, on the right side in Figure 7(a). They have been eliminated in Figure 7(b), by using t6 instead of t7 and t8 instead of t10.

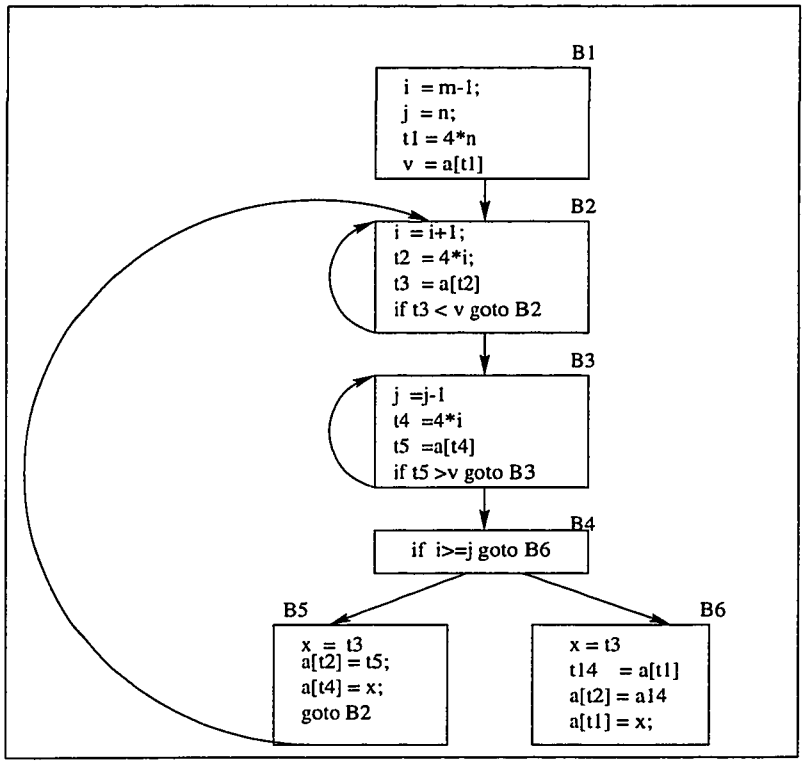


Figure 8: B5 and B6 after common subexpression elimination.

`Example` Figure 8 shows the result of eliminating both global and local common subexpressions from blocks B5 and B6 in the control flow graph of Figure 6. We first discuss the transformation of B5 and then mention some subtleties involving arrays.

After local common subexpressions are eliminated, B5 still evaluated *4*i* and *4*j*, as shown in Figure 7(b). Both are common subexpressions; in particular, the three statements

```
t8    = 4*j;
t9    = a[t8];
a[t8] = x;
```

in B5 can be replaced by

```
t9    = a[t4];
a[t4] = x;
```

using *t4* computed in block B3 . In Figure 8, observe that as control passes from the evaluation of *4*j* in B3, there is no change in *j*, so *t4* can be used if *4*j* is needed.

Another common subexpression comes to light in B5 after t4 replaces t3. The new expression *a[t4]* corresponds to the value of *a[j]* at the source level. Not only does *j* retain its value as control leaves B3 and then enters B5, but *a[j]*, a value computed into a temporary t5, does too because there are no assignments to elements of the array *a* in the interim. The statements

```
t9    = a[t4];
a[t6] = t9;
```

in B5 can therefore be replaced by

```
a[t6] = t5;
```

Analogously, the value assigned to *x* in block B5 of Figures 7(b) is seen to be the same as the value assigned to *t3* in block B2. Block B5 in Figure 8 is the result of eliminating common subexpressions corresponding to the values of the source level expression *a[i]* and *a[j]* from B5 in Figures 7(b). A similar series of transformations has been done to B6 of Figure 8.

The expression *a[t1]* in blocks B1 and B6 of Figure 8 is not considered a common subexpression, although *t1* can be used in both places. After control leaves B1 and before it reaches B6, it can go through B5, where there are assignments to *a*. Hence, *a[t1]* may not have the same value on reaching B6 as it did on leaving B1, and it is not safe to treat *a[t1]* as a common subexpression.

We will show in Chapter 7 that how to use our optimization framework to perform such transformation.

Chapter 5

Control-Flow Analysis: Building CFG Based on CPC-AST

Control Flow Graph (CFG) is a widely used compiler intermediate representation to facilitate data-flow analyses and transformations. Nodes in the flow graph represent computation, and the edges represent the flow of control. In this chapter, we will introduce the process of building a CFG based on a program's CPC-AST representation.

First, we will generally describe what a CFG is. Then, we will introduce some techniques to build a CFG. Next, we will give a formal definition of a CFG and present a structural analysis based algorithm to construct CFGs for CPC programs.

5.1 Introduction to Basic Blocks and Control Flow Graph (CFG)

Optimization requires that we have compiler components that can construct a global "understanding" of how programs use the available resources. The compiler must characterize the control flow of programs and the manipulations they perform on their data, so that any unused generality that would ordinarily result from unoptimized compilation can be stripped away; thus, less efficient but more general mechanisms are replaced but more efficient, specialized ones.

When a program is read by a compiler, it is initially seen as simply a sequence of characters. The lexical analyzer turns the sequence of characters into tokens, and

the parser discovers a further level of syntactic structure. The result produced by the compiler front end may be a syntax tree or some lower-level form of intermediate code. However, the result, whatever its form, still provides relatively few hints about what the program does or how it does it. It remains for control-flow analysis to discover the hierarchical flow of control within each procedure and for data-flow analysis to determine global (i.e, procedure-wide) information about the manipulation of data.

Before we consider the formal techniques used in control-flow and data-flow analysis, we present a simple example. We begin with the C routine in Figures 9(a), which computes, for a given $m \geq 0$, the m^{th} Fibonacci number. Given an input value m, it checks whether it is less than or equal to one and returns the argument value if so; otherwise, it iterates until it has computed the m^{th} member of the sequence and returns it. In Figures 9(b), we give a translation of C routine into an intermediate representation (IR).

```
unsigned int fib(m) {
unsigned int f0, f1, f2,i;

f0 = 0,
f1 = 1,

if (m <= 1)
  return m;
else {
  for (i=2; i<=m; I++) {
    f2= f0 + f1;
    f0 = f1;
    f1 = f2;
  }
 return f2;
}
}
```

(a). A C routine that computes Fibonacci numbers

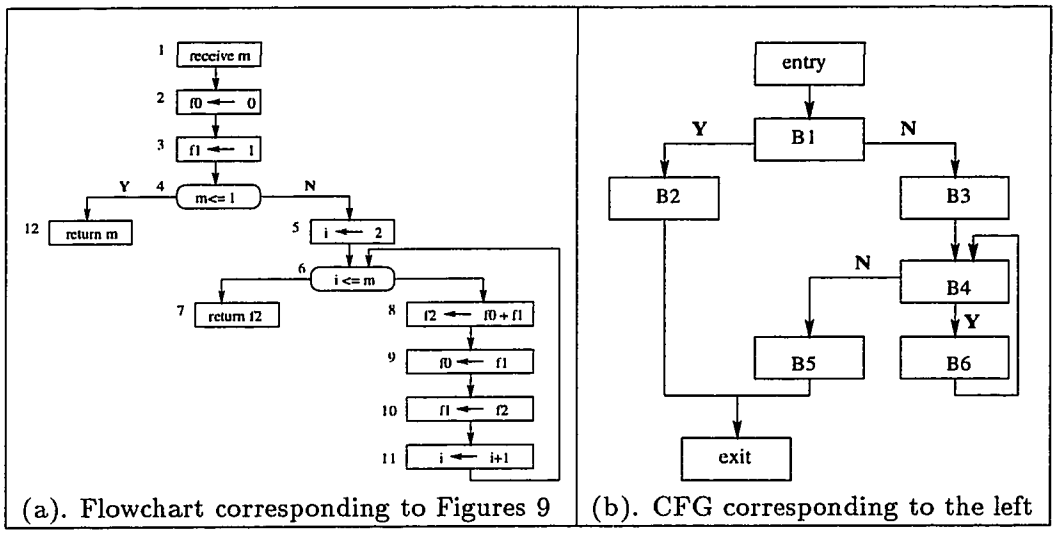
```
1:      receive m
2:      f0 = 0
3:      f1 = 1
4:      if (m <= 1) goto L3
5:      i = 2
6: L1:  if (i<=m) goto L2
7:      return f2
8: L2:  f2 = f0+f1
9:      f0 = f1
10:     f1 = f2
11:     i  = i+1
12:     goto L1
13:L3:  return m
```

(b). Intermediate code for the C routine on the left.

Figure 9: An example of fibonacci

Our first task in analyzing this program is to discover its control structure. One

might protest at this point that the control structure is obvious in the source code–the routine's body consists of an *if-then-else* with a loop in the *else* part; but this structures is no longer obvious in the intermediate code. Further, the loop might have been constructed of *if*s and *goto*s, so that the control structure might not have been obvious in the source code. Thus, the formal methods of control-flow analysis are definitely not useless. To make their application to the program clearer to the eye, we first transform it into an alternate visual representation, namely, a flowchart, as shown in Figure 10(a).



(a). Flowchart corresponding to Figures 9 (b). CFG corresponding to the left

Figure 10: Flowchart and flowgraph corresponding to the fibonacci program.

Next, we identify basic blocks, where a basic block is informally, a straight-line sequence of code that can be entered only at the beginning and exited only at the end. Clearly, nodes 1 through 4 form a basic block, which we call *B1*, and nodes 8 through 11 form another, which we call *B6*. Some node forms a basic block itself; we make node 12 into B2, node 5 into *B3*, node 6 into B4, and node 7 into B5. Next we collapse the nodes that form a basic block into a node representing the whole sequence of *IR* instructions, resulting in the so-called control flowgraph of the routine shown in 10(b).

For technical reasons that will become clear when we discuss backward data-flow analysis problems, we add an *entry* block with the first real basic block as its only successor, an *exit* block at the end, and branches following each actual exit from the

routine (blocks *B2* and *B5*) to the *exit* block.

Next, we identify the loops in the routine by using what are called *dominators.* In essence, a node A in the flowgraph dominates a node B if every path from the *entry* node to B includes A. It is easily shown that the dominance relation on the nodes of a flowgraph is antisymmetric, reflexive, and transitive, with the result that it can be displayed by a tree with the *entry* node as the root, For our flowgraph in Figure 10(b), the dominance tree is shown in Figure 11, in which a parent *immediately* dominates its children.

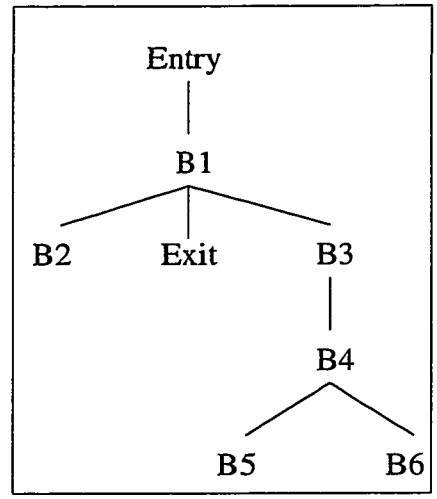


Figure 11: Dominance tree for the flowgraph of fibonacci program

Now we can identify a loop using the dominate relation. Please note that edges in control flowgraph are *directed*, representing control flow transfers from one program point (statement) to another. A back edge in the flowgraph is an edge whose tail dominates its head. For example, the edge from *B6* to *B4* shown in Figure 10(b) is a back edge. A loop consists of all nodes dominates by its entry node (the tail of the back edge) from which the entry node can be reached (and the corresponding edges) and having exactly one back edge within it. Thus, *B4* and *B6* form a loop with *B4* as its entry node, as expected, and no other set of nodes in the flowgraph does.

5.2 Approaches to Control-Flow Analysis

There are two main approaches to control-flow analysis of a single routine, both of which start by determining the basic blocks that make up the routine and then constructing its flowgraph. The first approach uses dominators to discover loops and simply notes the loops it finds for use in optimization. This approach is sufficient for use by optimizers that do data-flow analysis by iteration. (this method is also called *iterative* method.)

The second approach, called interval analysis, includes a series of methods that analyze the overall structure of the intervals. An interval is a region of a program, which can be abstracted maintaining some program properties. The nesting structure of the interval forms a tree called a control tree, which is useful in the structuring and speeding up data-flow analysis. The most sophisticated variety of interval analysis, called structural analysis, classifies essentially all the control-flow structures in a routine. It is sufficiently important that we devote a separate section to it.

Most current optimizing compiler use dominators and iterative data-flow analysis. And while this approach is the least time-intensive to implement and is sufficient to provide the information needed to perform most of the optimizations discussed below, it is inferior to the other approaches in three ways, as follows:

- The interval-based approaches are faster at performing the actual data-flow analyses, especially for structural analysis and programs that use only the simple types of structures.
- The interval-base approaches (particularly structural analysis) make it easier to update already computed data-flow information in response to changes to a program (changes made either by an optimizer or by the compiler user), so that such information need not be recomputed from scratch.
- Structural analysis makes it particularly easy to perform the control-flow transformations as we'll discuss in the following.

Thus, we feel that it is essential to present all three approaches and to leave it to the compiler implementer to chose the combination of implementation effort and optimization speed and capabilities desired.

Since the structural analysis is the mostly used method among interval analysis techniques in practice, we will concentrate on introduction to this analysis technique.

5.3 Structural Analysis

Structural analysis is a more refined form of interval analysis. Its goal is to make the syntax-directed method of data-flow analysis (developed by Rosen for use on syntax trees [Ros77, Ros79]) applicable to lower-level intermediate code. Rosen's method, called *high-level data-flow analysis*, has the advantage that it gives, for each type of structured control-flow construct in a source language, a set of formulas that perform conventional (bit-vector) data-flow analysis across and through them much more efficiently than iteration does. Thus this method extends one of the goals of optimization, namely, to move work from execution time to compilation time, by moving work from compilation time to language-definition time– in particular, the data-flow equations for structured control-flow constructs are determined by the syntax and semantics of the language.

Structural analysis extends this approach to arbitrary flowgraph by discovering their control-flow structure and providing a way to handle improper regions. Thus, for example, it can take a loop made up of ifs, gotos, and assignments and discover that it has the form of a *while* or *repeat* loop, even through its syntax gives no hint of that.

It differs from basic interval analysis in that it identifies many more types of control structures than just loops, forming each into a region, and as a result, provides a basis for doing very efficient data-flow analysis. One critical concept in structural analysis is that every region it identifies has exactly one entry point, so that, for example, an irreducible or improper region (such as the rightest region shown in Figure 13) will always include the lowest common dominator of the set of entries to the strongly connected component that is the multiple-entry cycle within the improper region.

Figure 12 and Figure 13 give examples of typical acyclic and cyclic control structures, respectively, that structural analysis can recognize.

In the following section, we will use the structural analysis technique to build a CFG for a CPC program.

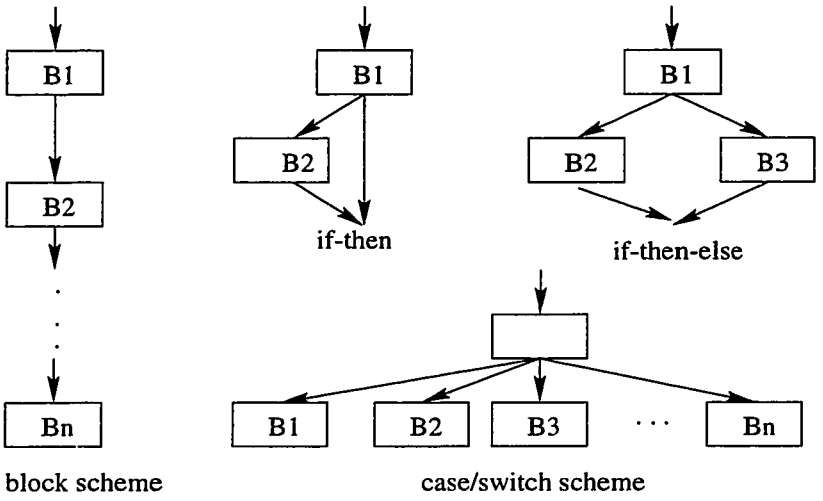


Figure 12: Some types of acyclic regions used in structural analysis

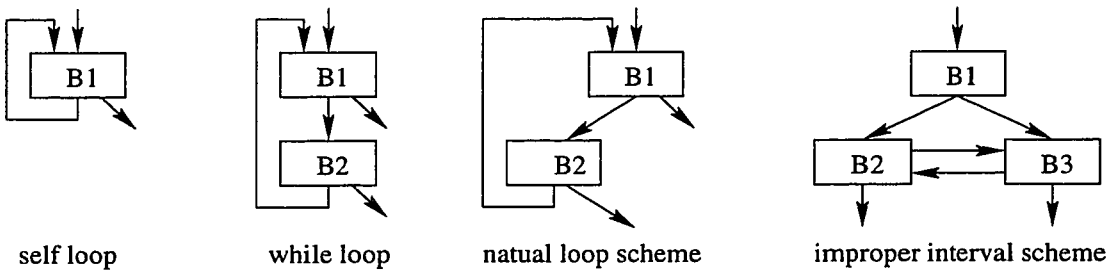


Figure 13: Some types of cyclic regions used in structural analysis

5.4 Basic Blocks

Since all the approaches require identification of basic blocks and construction of the flowgraph of the routine, we discuss these topics next.

Formally, a *basic block* is a maximal sequence of instructions that can be entered only at the first of them and exited only from the last of them.

There are a sequence of consecutive statements in a basic block in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [ASU86].

Thus, the statement in a basic block may be (1) the entry point of the routine, (2) a target of a branch , or (3) an instruction immediately following a branch or a return. Such instructions are called *leaders*. To determine the basic blocks that compose a routine, we first identify all the leaders, and then, for each leaders, include in its basic block all the instructions from the leader to the next leader or the end of the routine, in sequence.

The easiest control flow is a linear chain, in which control flow passes each statement consecutively. *Basic Block* (BB) is a formal definition for such a sequence of statements.

```
t1 = a * a;
t2 = a * b;
t3 = 2 * t2;
t4 = t1+ t3;
t5 = b * b;
t6 = t4+ t5;
```

Figure 14: Example of Basic Block

The sequence of statements shown in Figure 14 forms a basic block, in which control enters from the beginning ($t1 = a * a$) and leaves at the end ($t6 = t4 + t5$).

The algorithm that computes basic blocks based on three-address statement representation is well known, and we quote as follows [ASU86]:

Algorithm1.1: Partition into basic blocks.

Input. A sequence of three-address statements.

Output. A list of basic blocks with each three-address statement in exactly one block.

Method

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are following.
   (a) The first statement is a leader.
   (b) Any statement that is the target of a conditional or unconditional goto is a leader.
   (c) Any statement that immediately follows a goto or conditional statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but excluding the next leader or the end of the program.

The following example gives the fragment of source code in Figure 15(a) which computes the dot product of two vectors a and b of length 20. A list of three-address statements performing computation is shown in Figure 15(b). Two basic blocks are shown in Figure 15(c). The first basic block consists of two statements and the second basic block consists of ten statements, forming the loop body.

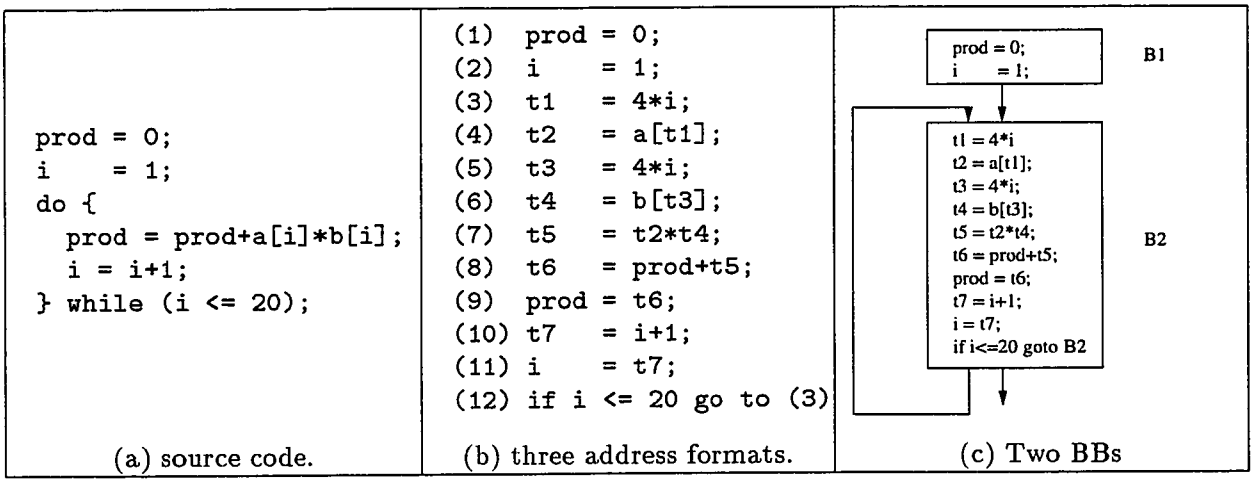


Figure 15: Example of three-address statements and basic blocks

However, partitioning basic blocks from a high-level AST-represented program is not as straightforward as the above algorithm, because each *compound* statement such as IF, WHILE, and FOR statements, consists of several basic blocks and can not be treated as one basic block. Before we go into details on partitioning basic blocks for compound statements, let's first define what is control flow graph.

5.5 Control Flow Graphs

Now, having identified the basic blocks, we characterize the control flow in a procedure by a rooted, directed graph with a set of nodes, one for each basic block plus two distinguished ones called *entry* and *exit*, and a set of (control-flow) edges running from one basic block to another in the same way that control-flow edges of the original flowchart connected the final instructions in the basic blocks to the leaders of basic blocks; in addition, we introduce an edge from *entry* to the initial basic block(s) of the routine and an edges from each final basic block (i.e., a basic block with no successors) to *exit*. The *entry* and *exit* blocks are not essential and are added for the technical reasons – they make many of our algorithm simpler to describe. (See, for example, in the data-flow analysis performed for global common-subexpression elimination, we need to initialize the data-flow information for the *entry* block differently from all other blocks if we do not ensure that the *entry* block has non edges entering it.) The resulting directed graph is the *control flow graph* of the routine.

We assume that we are giving a flowgraph $G =< N, E >$ with node set N and edge set $E \subseteq N * N$, where $entry \in N$ and $exit \in N$. We generally write edges in the form $a \leftarrow b$, rather than $< a, b >$.

Further, we define the set of *successor* and *predecessor* basic blocks of a basic block in the obvious way, a *branch node* as one that has more than one successor, and a *join node* as one that has more than one predecessor. We denote the set of successors of a basic block $b \in N$ by $Succ(b)$ and the set of predecessors by $Pred(b)$. Formally,

$$Succ(b) = \{n \in N || \exists e \in E \text{ such that } e = b \leftarrow n\}$$

$$Pred(b) = \{n \in N || \exists e \in E \text{ such that } e = n \leftarrow b\}$$

Therefore, a *Control flow graph* (CFG) tries to capture the potential control flow transfer between two program points. Within a basic block, the control transfer is

trivial due to the definition of basic blocks. Thus, a CFG focuses on the transfer of control among basic blocks.

Once flow-of-control information is added onto the set of basic blocks, the resulting graph is called a *control flow graph* (CFG). The nodes of the CFG are basic blocks. Two nodes are distinguished as *entry* and *exit*. There is a directed edge from block B1 to block B2 if

1. there is a conditional or unconditional jump from the last statement of B1 to the first statement of B2, or
2. B2 immediately follows B1 in the order of the program, and B1 does not end in an unconditional jump.

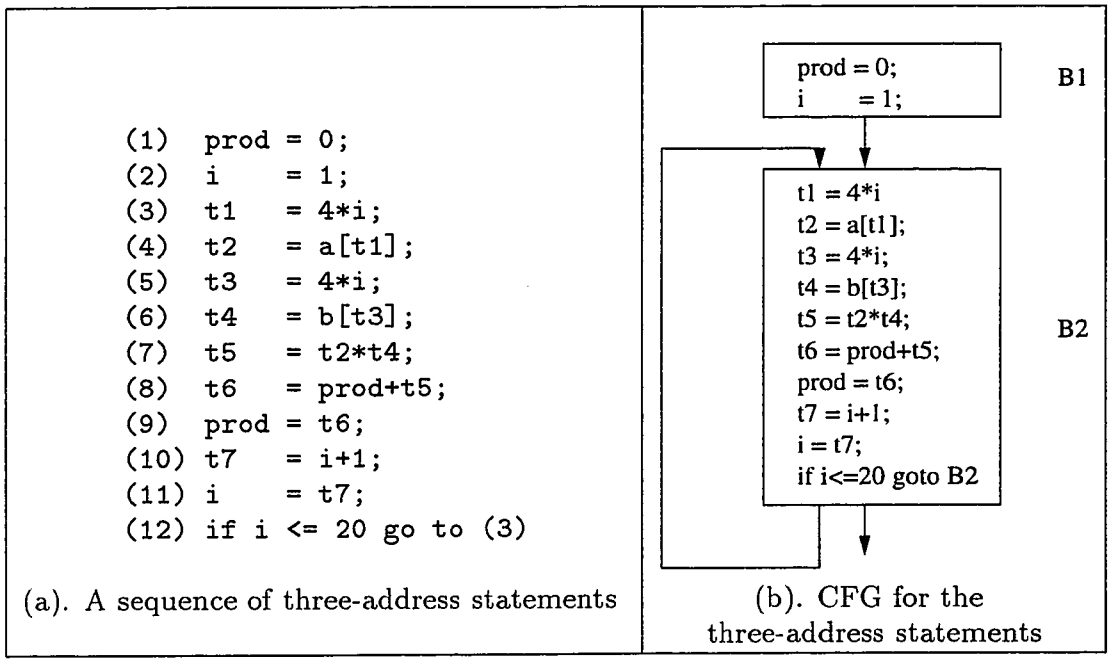


Figure 16: An example of CFG base on three-address

For example, the edge from B1 to B2 shown in Figure 16(b) is due to that control can enter the loop body after initialization. The edge from B2 to B2 itself is due to that there is a conditional jump to the loop header from the end of the loop.

As we mentioned before, partitioning basic blocks from a high-level AST-represented program is not as straightforward as the well known algorithm based on three-address representation. The difficulty lies in the fact that each *compound* statement such as IF, WHILE, and FOR statement, consists of several basic blocks and cannot be treated as one basic block. Thus, we need to design partitioning algorithms for these high-level constructs to derive the corresponding basic blocks. We also need to obey the original control transfer specified by those high-level constructs and build appropriate CFGs.

Structural analysis is more appropriate than iterative method in constructing CFG for a CPC program, since high level structures are still retained in the CPC AST representation. In the following, we will show such an algorithm which builds a CFG for a CPC program.

5.6 Building CFG based on CPC-AST

5.6.1 Representation of Basic Blocks and CFG in CPC-AST

Basic blocks can be represented by a variety of data structures. For example. after partitioning the three-address statements by *Algorithm 1.1*, each basic block can be represented by a record consisting of a count of the number of quadruples (such as (operator,operand-1,operand-2,destination)) in the block, followed by a pointer to the leader (first quadruple) of the block, and by the list of predecessors and successor of the basic block. An alternative is to make a linked list of the quadruples in each block.

A CFG is a graph. Usually there are two means to represent a graph, *adjacency matrix* and *adjacency list* [CLR90]. In our implementation, we choose a method similar to the adjacency list representation since it is an efficient representation in terms of space for sparse graphs. For each basic block, we use two lists, one is the predecessor list and the other is the successor list of the basic block to represent flowgraph structure. It is very convenient to build CFGs with double directed edges so that forward and backward analyses can be supported in the same way.

CFG per Function of CPC-AST

In CPC-AST, a function definition is represented as a *symbol* node. Since we are interested in the *global* analyses, a CFG for each function needs to be built. To represent such a CFG, we first add one extra field in the symbol node definition, to store a CFG for the function. Thus, the `sym` data structure is argumented by the following field:

cfg_graph: points to a *cfg* graph of the current function.

The data structure of CFG graph is shown in Figure 17.

```
#define MaxBBNum (1024 * 2)
typedef struct cfg_graph {
  int           num;
  bb_cfg        bbs_array[MaxBBNum];
} cfg_graph;
```

Figure 17: Data structure of a CFG graph

Each CFG has two fields:

- num: the total number of BBs in the CFG graph.
- bbs_array[MaxBBNum]: an array of BBs in the CFG graph.

A CFG graph consists of two parts, $CFG = (V, E)$, where V is a set of basic blocks and E is a set of edges, representing control transfer between BBs. In Figure 17, we use an array of `bb_cfg` to represent basic blocks. Edges of BB are encoded in data structure `bb_cfg` (see below).

Data Structures For Basic Blocks

The data structure for BB is shown in Figure 18.

Each CFG BB contains the following fields:

- *type:* There are four types of nodes in CFG:
  - A *normal* basic block node. See B1, B3, B4 in Figure 19 and B1, B4, B7 in Figure 20.

```
enum node_type { NORMAL, MERGE, END, FUNCTION_END } ;

typedef struct bb_cfg {
  enum node_type        type;
  struct edges          *pred;
  struct edges          *succ;
  struct stmtlist       *front;
  struct stmtlist       *tail;
  void                  *in;
  void                  *out;
} bb_cfg;
```

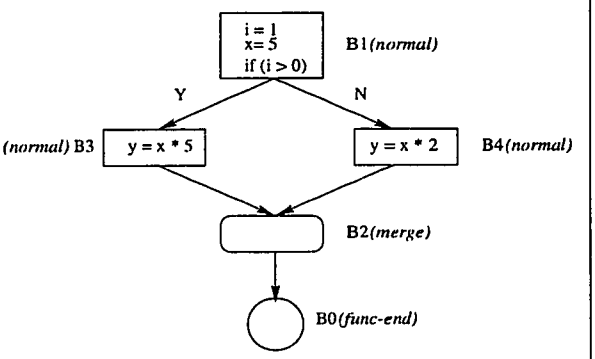
Figure 18: Data structure of BB

```
int foo()
{
 int x, y;
 int i;

 i = 1;
 x = 5;

 if (i>0)
   y = x*5;
 else
   y = x*2;
}
```

(a). An if-then statement



(b). CFG nodes of if-then statement

Figure 19: An example of CFG nodes of if-then statement.

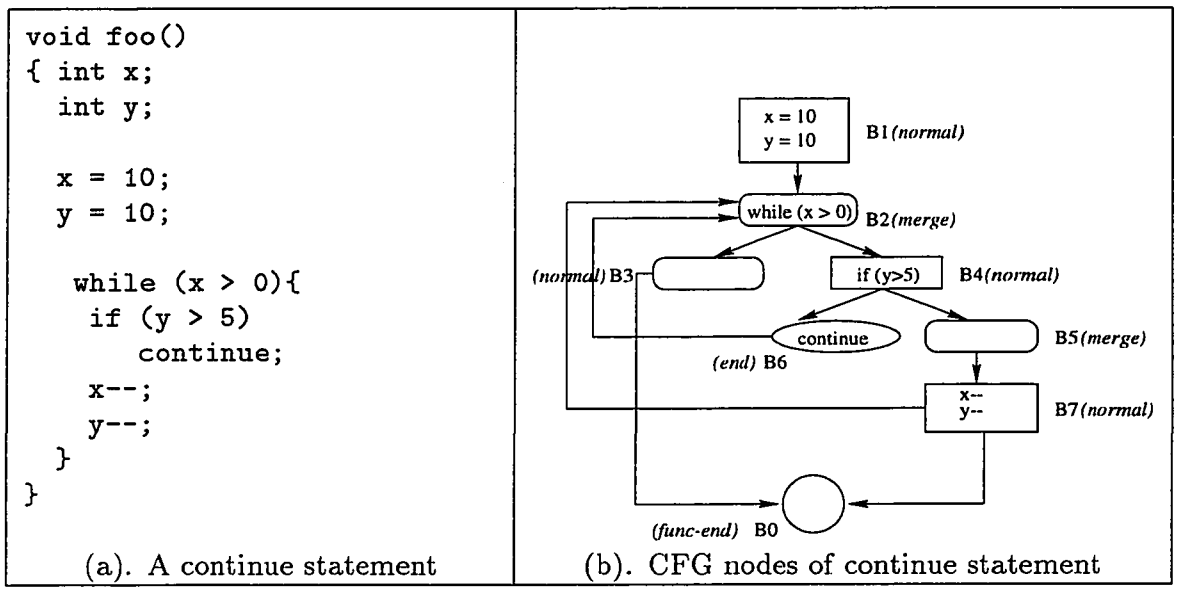


Figure 20: An example of CFG nodes of continue statement.

  - A *func-end* node, which represents the *exit* point of a function body. See B0 in Figure 19 and B0 Figure 20.
  - A *merge* node, which merges control flow of its predecessors. For example, a conditional statement usually has one *merge* node, combining control flow from either *then* or *else* branch. See B2 in Figure 19 and B2, B3, B5 in Figure 20.
  - An *end* node, which is only used in `continue`, `break` and `goto` statements to represent the end of a basic block. See B6 in Figure 20.

- *pred:* pointer pointing to the predecessors of a current node.
- *succ:* pointer pointing to the successors of a current node.
- *front:* pointer pointing to the first statement of a current BB.
- *tail:* pointer pointing to the last statement of a current BB.
- *in:* a temporary storage used in computing input of BB's.
- *out:* a temporary storage used in computing out of BB's.

The reason that we use double-linked lists for both CFG edges and the statement sequence within the block is to support forward and backward analyses in an equal-foot manner.

We purposely set the type *in* and *out* fields as `void`, since in such doing, we can store any type of data-flow information through explicit type casting. Thus, if users want to use data-flow information stored in these two fields, they have to recast anything retrieved from the above two fields into appropriate types. This scheme is not very clean but general to support a framework to hold any type of dataflow information.

Since the number of predecessors and successors of a BB may be more than one, fields *pred* and *succ* are actual pointers to linked lists. The data structure for such edge linked list is shown in Figure 21.

```
typedef struct edges {
  bb_cfg        * node_ptr ;
  struct edges  * next;
} cfg_edge;
```

Figure 21: Data structure of CFG edge base on CPC-AST

Each CFG edge contains the following fields:

- *node_ptr:* pointer pointing to a node in the CFG graph.
- *next:* pointer pointing to next edge in the CFG graph.

Each edge is stored twice in CFG. For example, if there is an edge: $BB_1 \rightarrow BB_2$, then BB_2 will appear at the linked list pointed by the *succ* field of BB_1 and BB_1 will appear at the linked list pointed by the *pred* field of BB_2. As we mentioned before, the reason that an edge appears twice is that we want to support efficient *forward* and *backward* analyses within our framework. Generally speaking, in forward analysis we need to access the *out* fields of all *predecessors'* of a BB; and in the backward analysis, we need to access the *in* fields of all *successors'* of a BB. By supporting accessing an edge in double directions, both type of analyses can be supported efficiently.

Similarly fields *front* and *tail* are pointers to linked lists since statements in a BB may be more than one. Furthermore, the linked list pointed by the *front* is a *forward* linked list (original text order), and the one pointed by the *tail* is a *backward* linked list. It is necessary to have such a *double-linked* list since we want our framework to be general to support both *forward* and *backward* analyses.

In order to deal with *goto* statements, we build a mapping table, remembering which BB a statement belongs to. In so doing, we can easily find the target BB for each goto. We use an array of *stmt2bb* structure elements to store such information. Figure 22 lists the *stmt2bb* definition.

```
typedef struct stmt2bb {
  struct stmt   *stmt_ptr;
  bb_cfg        *bb_ptr;
} stmt2bb ;
```

Figure 22: Mapping from statements to BBs

Each *stmt2bb* contains the following fields:

- *stmt_ptr:* pointer to a statement.
- *bb_ptr:* pointer to a current BB contain the statement.

Thus, we can easily find the BB pointer using a statement pointer as hashing index.

In our two-passes CFG building algorithm (see next section), the mapping table is built during the first pass of analyzing a function and visited during the second pass when goto statements are processed.

5.6.2 Main Control Algorithm of Building CFG Based on CPC-AST

In Figure 23, we list the two-passes algorithm to build a CFG based on the CPC-AST representation.

```
building_cfg_function(symbol *sym)
{
   initial_cfg(sym);                      /*initialize the current cfg_graph   */
   end_bb     = get_bb_cfg(FUNCTION_END);/*create a func_end node for exit    */
                                         /*point a function body*/
   current_bb = get_bb_cfg(NORMAL);/*create a normal node for the current BB  */
   build_cfg_stmt(sym->compound);        /*building cfg for each BB           */
   if (!IsEnd(current_bb))
    insert_two_edges(current_bb, end_bb);/*Connect the current BB to the      */
   store_cfg(sym);      /*store the total number of BB in the cfg_graph       */
   process_goto(sym);   /*build cfg for goto statement by go through the each */
                        /*statement of the current function                   */
}


build_cfg_stmt(stmt *s)
{ for(;s;s=s->next){         /*when there is a statement left in function body*/
    /* Preallocate a basic block */
    if (IsEnd(current_bb)) {           /*if current-BB is end of a basic block*/
      current_bb=get_bb_cfg(NORMAL);    /*create a normal node for current-BB */
    } else if (IsMerge(current_bb) ){/*if current-BB is merges pointer of its */
                                     /* predecessors                          */
       prev=current_bb ;              /*previous-BB point to current-BB   */
       current_bb=get_bb_cfg(NORMAL);/*create a normal node for the current BB*/
       insert_two_edges(prev,current_bb); /*connect previous-BB to current- BB*/
    }
    if(s->label&&(!IsBBEmpty(current_bb))){/*deal with goto statement         */
    /*if there is Label: statement, the statement list should be separated  */
    /*into two parts with label as boundary*/
      prev = current_bb ;             /*previous-BB point to current-BB   */
      current_bb = get_bb_cfg(NORMAL);/*create a normal node for current-BB   */
      insert_two_edges(prev, current_bb);/*connect previous-BB to current-BB  */
    }
    if(s->code!=C_COMPOUND)/*if code type of the statement is not C_COMPOUND  */
                                       and the BB in stmt2bb table         */
    switch (s->code){  /*there is the detail algorithm for the particular type*/
      case C_NOP: {...}/*of a statement                                      */
      ...
    }
  }
}
```

Figure 23: Algorithm for building CFG

A function definition is represented as a CPC-AST symbol, which is the input to function `building_cfg`. At the beginning of `building_cfg`, we do some initializations and allocate two BBs, one represents the *exit* point of the function and the other is for the current empty BB. Then, function `build_cfg_stmt` is called to recursively build the CFG according to the algorithms described later in this chapter. At the end of `building_cfg`, we call another function `process_go` to connect `goto` statements with their target BBs.

Within function `build_cfg_stmt`, based on structural analysis technique we specify constructing rules for each C syntax construct. Then we follow such rules for each statement and build its own CFG graph accordingly. Before we start to build a CFG for a statement, we first check the following three cases:

1. If current BB is an END node, we allocate an empty BB for the current statement, which will be used to take all statements of the BB to which the current statement belongs.

2. If current BB is a MERGE node, we need to build an edge, linking the merge node with the newly created current empty BB.

3. If current statement is a *label* statement, it is a potential target for a *goto* statement, we end the current BB and create a new BB. At the same time, we connect the last BB with the newly created empty BB.

After checking these special cases, we insert this statement into *stmt2bb* table, preparing for the second pass to deal with *gotos*. Finally depending on the type of the statement being visited, we call an algorithm to deal each C statement structure accordingly.

5.6.3 The Algorithms of Building CFG for Each C Component Based on CPC-AST

Our CFG is based on CPC-AST. There are four types of nodes in CFG, *normal, merge, end,* and *func-end*, as we mentioned last section.

In Figure 24, we give the top-level algorithm to build CFG by recursive traversal of a CPC-AST tree. Variable *current-BB* denotes the basic block under consideration and *end-BB* represents the function exit point. Procedure *partition-bb*, see the

```
build_cfg_statement(S)
{
  for each statement S do{
    if  S is a compound statement
      build_cfg_statement(S);
    else
      partition-bb(S);
  }
}
```

Figure 24: Build a CFG based on structural analysis for a program represented in CPC-AST

following parts of this Chapter, processes each individual statement.

For the time being, we assume there is no *goto* statement to make the algorithm easier to be understood. We will come back to deal with *goto* statement in the later section.

Expression Statement

For an *Expression* statement, we insert it to the current basic block, current-BB.

Return statement

For a *Return* statement, add an new edge between the current-BB and the end-BB. Allocate a new BB and assign it to current-BB.

If_then Statements

For each IF-THEN statement, we need to create a *merge* node and add three edges. See Figure 25.

1. Insert the condition expression of *if* statement into previous basic block. Let's use `prev_BB` to denote it.

2. Prepare an empty BB for the first BB of *then* branch. Let's use `first_BB` to denote it. Connect `first_BB` to `prev_BB`. This edge represents the control transfer when condition is true.

3. Set current-BB to `first_BB` and recursively build CFG for *then* branch.

4. Create a *merge* node for the end of IF-THEN statement. Let's use `end_if_BB` to denote it. Connect current-BB (the last BB of *then* branch) to `end_if_BB`. Connect the `prev_BB` to `end_if_BB`, representing when condition is false.

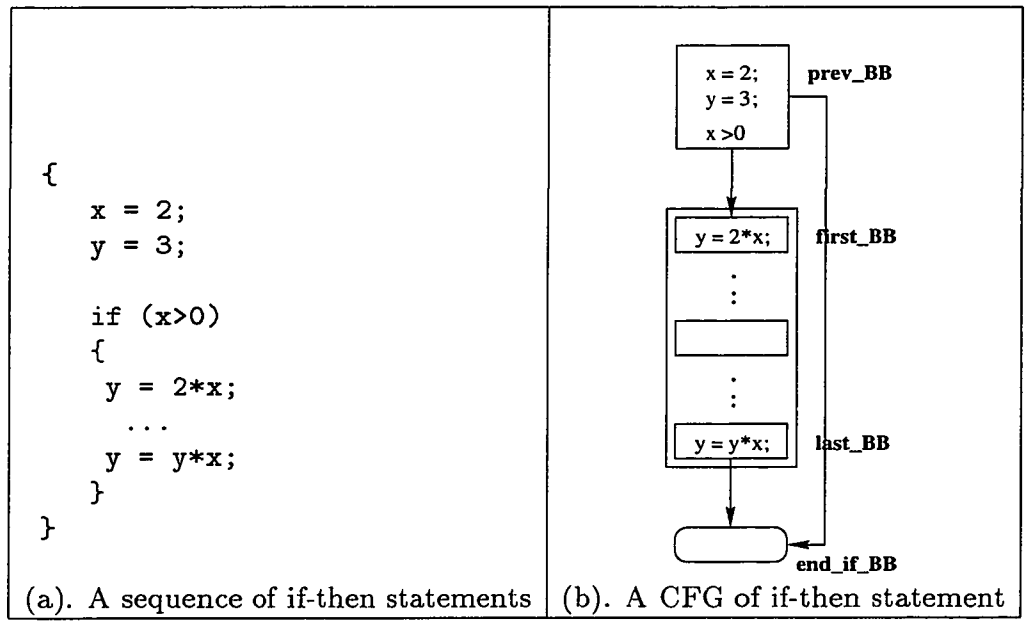


Figure 25: An example of if-then statement.

If_then_else statement

For each IF_THEN_ELSE statement, we need to create a *merger* node and add four edges. See Figure 26.

1. Insert the condition expression of *If_Then_Else* statement into previous basic block. Let's use `prev_BB` to denote it.

2. Prepare an empty BB for the first BB of *then* branch. Let's use `first_then_BB` to denote it. Connect `first_then_BB` to `prev_BB`. This edge represents the control transfer when condition is true.

3. set current_BB to `first_then_BB`. Recursively build CFG for *then* branch.

4. Prepare an empty BB for the first BB of *else* branch. Let's use `first_else_BB` to denote it. Connect `first_else_BB` to `prev_BB`. This edge represents the control transfer when condition is false.

5. Set current_BB to `first_else_BB`. Recursively build CFG for *else* branch.

6. Create a *merge* node for the end of IF_THEN_ELSE statement. Let's use `end_if_then_else_BB` to denote it. Connect the current-BB(the last BB of *then* branch) to `end_if_then_else_BB`, representing when condition is true. Connect the current_BB(the last BB of *else* branch) to `end_if_then_else_BB`, representing when condition is false.

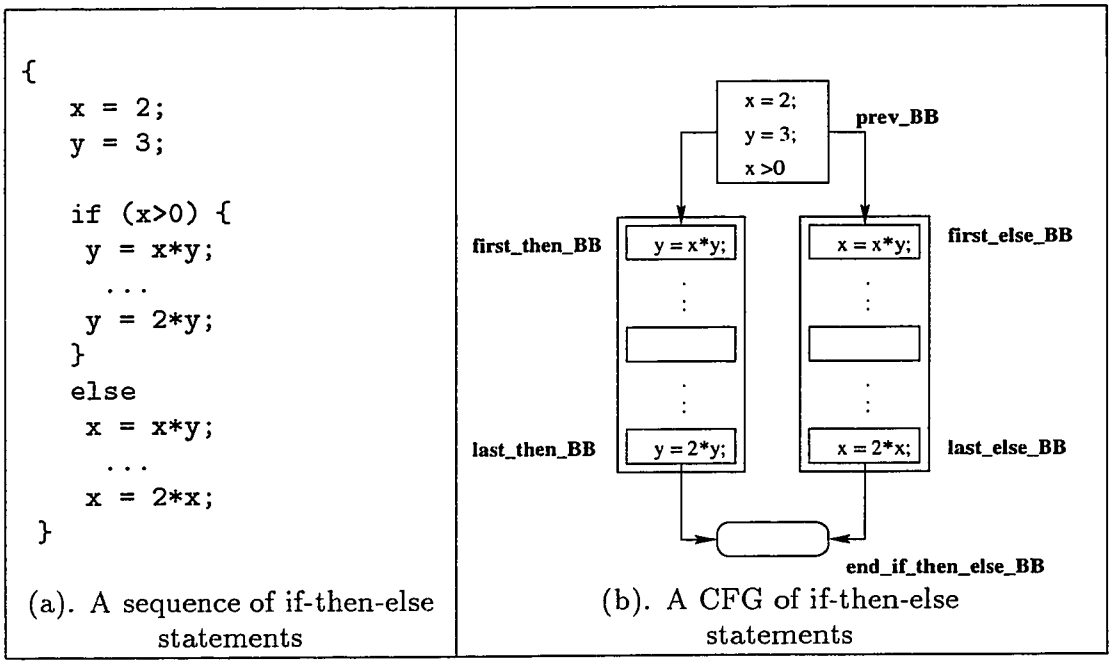


Figure 26: An example of if-then-else statement.

While loop

For each WHILE statement, we need to create two *merge* nodes and add four edges. See Figure 27.

1. Create a *merge* node for the test expression of the *While* statement. Let's use `test_BB` to denote it. Connect `prev_BB` to the `test_BB`.

2. Create a *merge* node for the end of WHILE statement. Let's use `end_while_BB` to denote it. connect the `test_BB` to the `end_while_BB`, representing the condition is false.

3. Push the `test_BB` on the continue-stack for continue statement. Push the `end_merge_BB` on the break-stack for break statement.

4. Prepare an empty BB for the first BB *while* loop body.Let's use `first_BB` to denote it. Connect the `test_BB` to `first_BB`. This edge represents the control transfer when condition is true.

5. Set current-BB to `first_BB`. Recursively build new cfg until the end of the *while* loop and current-BB is changed accordingly. Connect the current-BB(the last BB of the while loop body) to the `test_BB`.

6. Pop the `test_BB` from continue-stack for continue statement. Pop the `end_while_BB` from break-stack for break statement.

Do-While loop

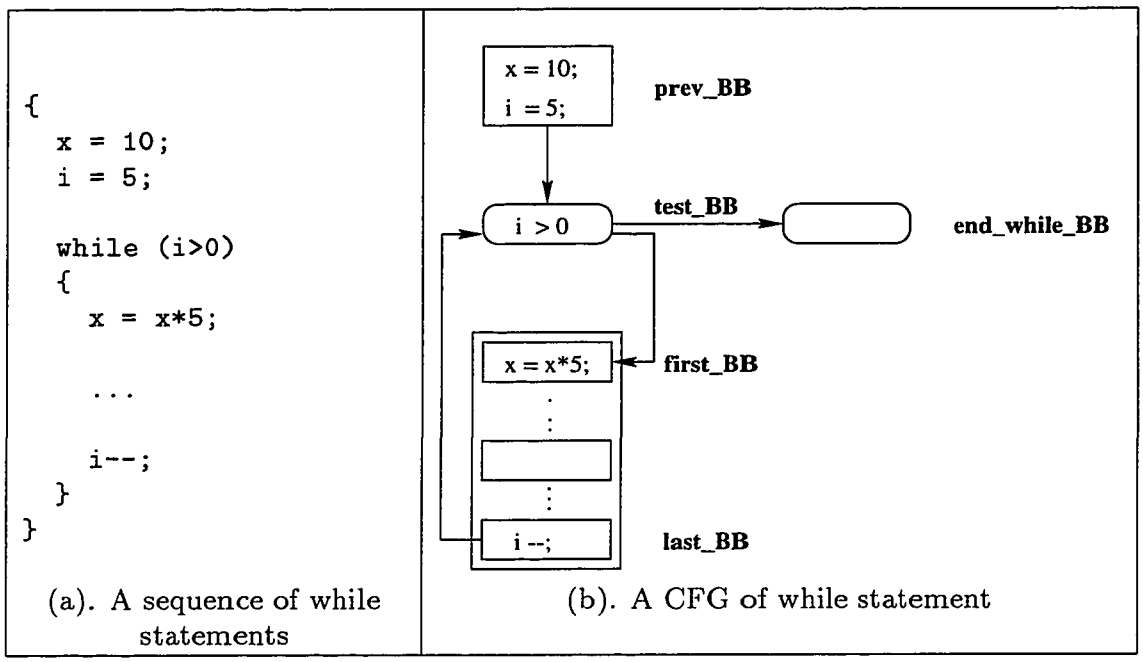
For each DO-WHILE statement, we need to create three merger node and add five edges. See Figure 28.

1. Create a new *merger* node at the beginning of *do-while* loop, Let's use `do-while_BB` to denote it. Connect `prev_BB` to the `do-while_BB`.

2. Create a *merge* node for the test expression of the *while* statement. Let's use `test_BB` to denote it. Connect the `test_BB` to the `do_while_BB` when the condition is true.

3. Create a *merge* node for the end of *do-while* statement. Let's use `end_do-while_BB` to denote it. Connect the `test_BB` to the `end_do-while_BB`, representing the condition is false.

4. Push the `test_BB` on the continue-stack for continue statement. Push the `end_do-while_BB` on the break-stack for break statement.



(a). A sequence of while statements

(b). A CFG of while statement

Figure 27: An example of while statement.

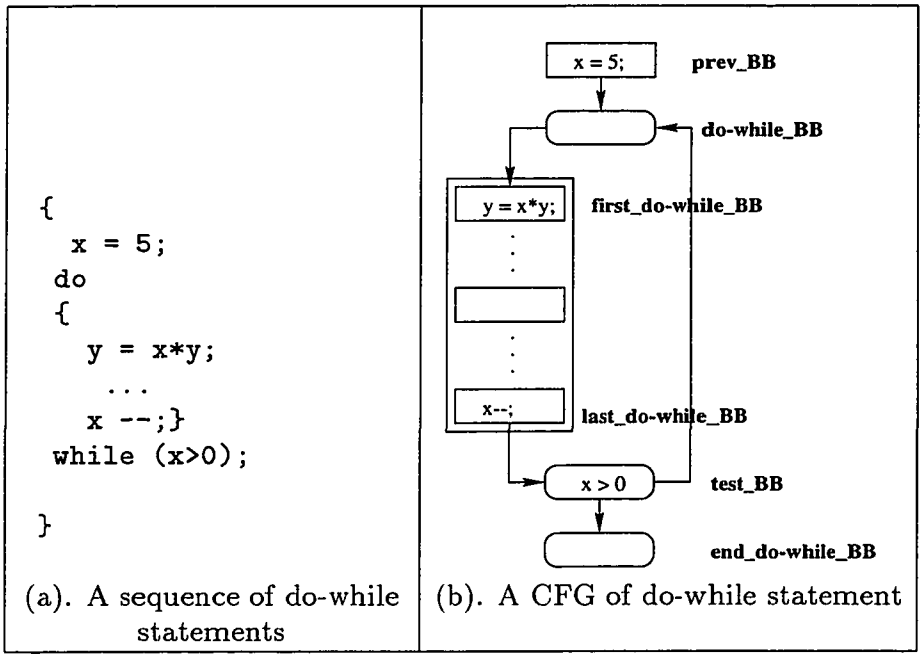
5. Prepare an empty BB for the first BB of *do-while* loop body. Let's use `first_do-while_BB` to denote it. Connect the `do-while_BB` to the `first_do-while_BB`. This edge represents the control transfer when condition is true.

6. Set current-BB to the `first_do-while_BB`. Recursively build CFG for the do-while loop body until it ends, and current-BB is changed accordingly. Connect the current-BB(the last BB of *do-while* loop body) to the `test_BB`.

7. Pop the `test_BB` from continue-stack for continue statement. Pop the `end_do-while_BB` from break-stack for break statement.

For loop

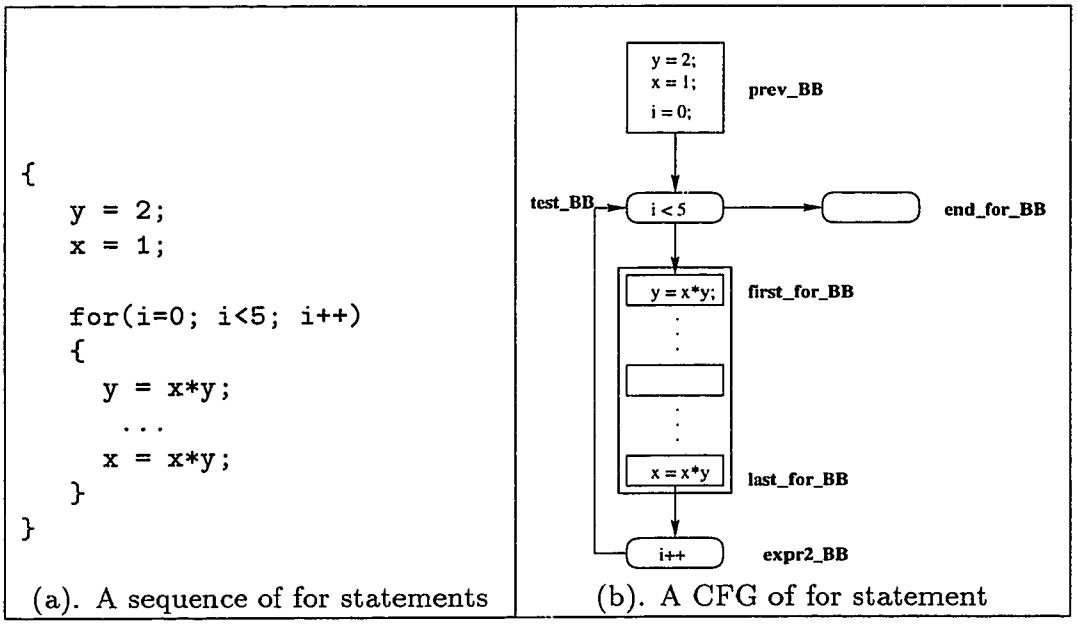
For each FOR statement, we need to create three *merge* nodes and add five edges. See Figure 29.



(a). A sequence of do-while statements

(b). A CFG of do-while statement

Figure 28: An example of do-while statement.



(a). A sequence of for statements

(b). A CFG of for statement

Figure 29: An example of for statement.

1. Insert the expr1 of the *for* statement into previous basic block. Let's use `prev_BB` to denote it.

2. Create a *merge* node for the test expression of the *for* statement. Let's use `test_BB` to denote it. Connect the `prev_BB` to `test_BB`.

3. Prepare an empty BB for the first BB of *for* loop body. Let's use `first_for_BB` to denote it. Connect the `test_BB` to the `first_for_BB`, representing the condition is true.

4. Create a *merger* node for the end of *for* statement. Let's use `end_for_BB` to denote it. Connect the `test_BB` to the `end_for_BB`, representing the condition is false.

5. Push the `test_BB` on the continue-stack for continue statement. Push the `end_for_BB` on the break-stack for break statement.

6. Set current-BB to the `first_for_BB`. Recursively build the cfg until the end of *for* loop body and the current-BB is changed accordingly.

7. Create a merge node for the expr2 of the *for* statement Let's use *expr2_BB* to denote it. Connect the current-BB(the last BB of the loop body, let's use `last_for_BB` to denote it) to the *expr2_BB*. Connect the *expr2_merger_BB* to the *test* point.

8. Pop the `test_BB` from continue-stack for continue statement. Pop the `end_merge_BB` from break-stack for break statement.

Break statement

1. Get a merge node BB from break-stack. Let's use `end_BB` to denote it .

2. Connect current-BB to the `end_BB`. Then, Set the type of current-BB to *end*. Let's use `break_BB` to denote it.

See Figure 30.

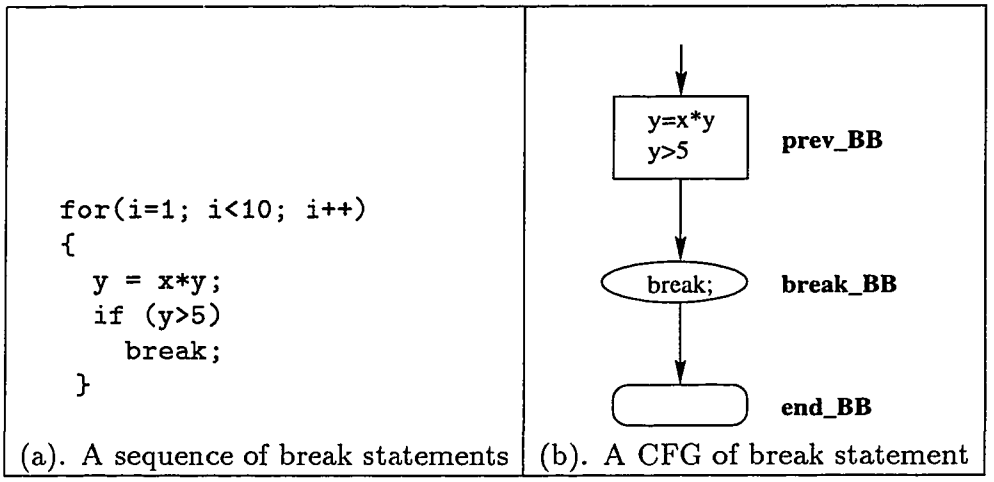


Figure 30: An example of break statement.

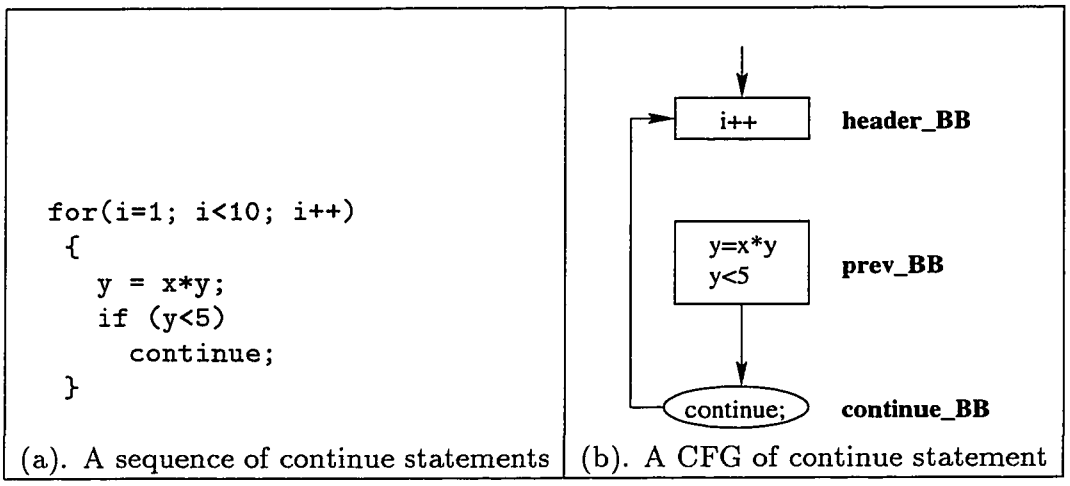
Continue statement

1. get a merger node from continue-stack. Let's use `header_BB` .
2. connect current BB to the `header_BB`. Then, Set the type of current-BB to *end*. Let's use `continue_BB` to denote it. See Figure 31.

Switch statement

For each SWITCH statement, we need to create one *merger* node and add one edge. See Figure 32.

1. Insert the switch() of the *switch* statement into previous basic block. Let's use `prev_BB` to denote it.
2. Create a *merge* node for the end of SWITCH statement. Let's use `end_switch_BB` to denote it. Connect the `prev_BB` to the `end_switch_BB`, representing the condition is false.
3. Push the `prev_BB` on the continue-stack for continue statement. Push the `end_switch_BB` on the break-stack for break statement.



(a). A sequence of continue statements (b). A CFG of continue statement

Figure 31: An example of continue statement.

4. Prepare an empty BB for the first BB of *switch* body. Let's use `first_switch_BB` to denote it.

5. set current-BB to the `first_switch_BB`. Recursively build CFG for the *switch* body until it ends and the current-BB is changed accordingly.

6. Pop the `prev_BB` from continue-stack for continue statement. Pop the `end_switch_BB` from break-stack for break statement.

Case statement

1. Get a node from the continue-stack. Let's use `header_BB` to denote it. Connect the `header_BB` to the current BB, let's use `case_BB` to denote it. See Figure 33.

Default statement

1. Get a node from the continue-stack. Let's use `header_BB` to denote it. Connect the `header_BB` to the current BB.

2. Get a node from the break-stack. Let's use `end_BB` to denote it. Connect current BB to the `end_BB`.See Figure 34.

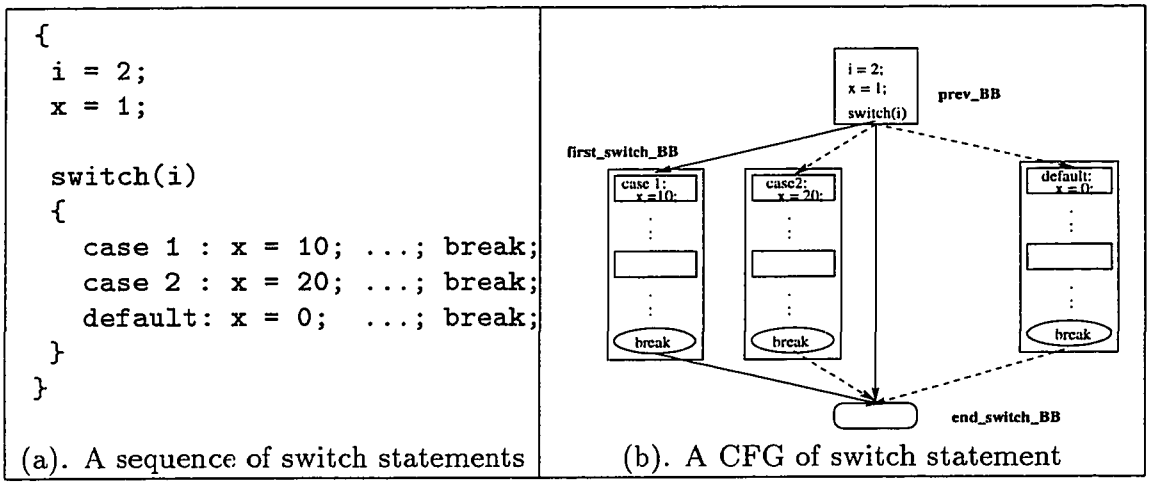


Figure 32: An example of switch statement.

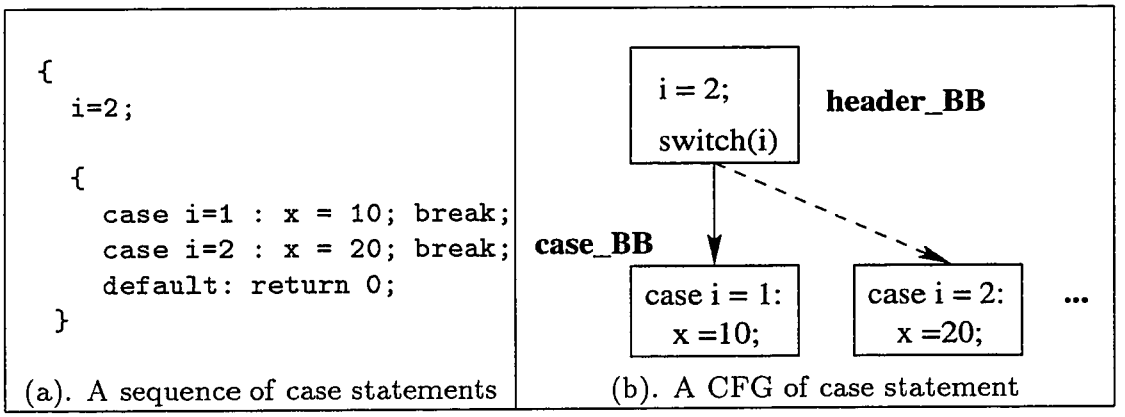


Figure 33: An example of case statement.

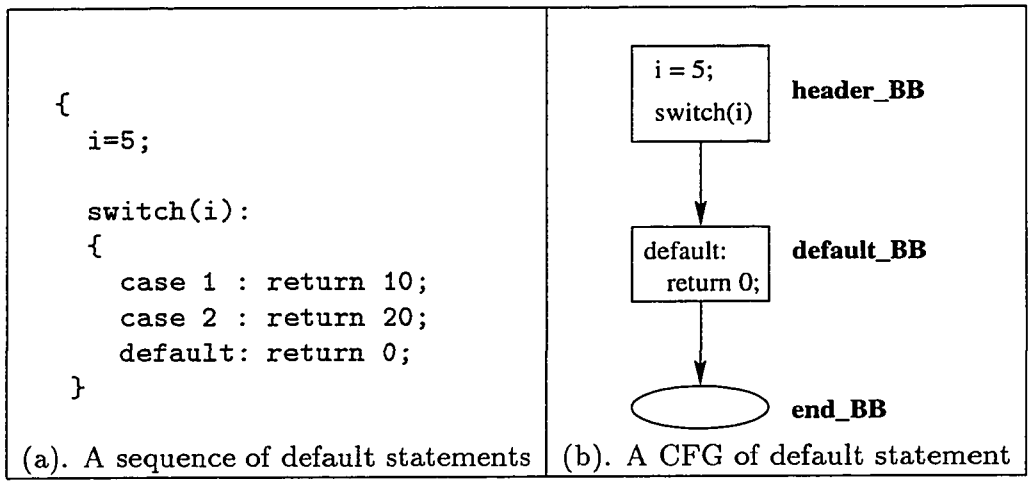


Figure 34: An example of default statement.

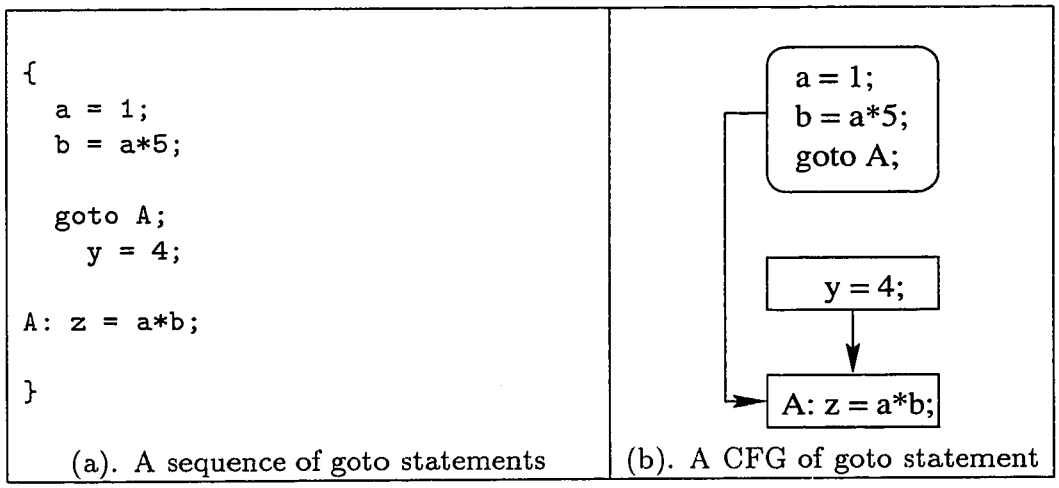


Figure 35: An example of goto statement.

Goto Statement

For a *goto* statement, we need go through the sequence of statements twice. See Figure 35.

1. At first step, when there is a *Label:*, we separate the sequence statements into two parts. The statements before the *Label:* belong to prevBB and the statements including the *Label:* belong to currentBB. When there is a *goto*, just to initialize the type of currentBB as END.

2. At second step, if there is a *goto* statement in the function body, we need go through the cfg again and try to find its target. Then we create an edge from the *goto* to the target *label*. Since there is a *stmt2bb* table being built during the first pass, searching the table to find the target of the goto becomes easy.

Chapter 6

Iterative Dataflow Analysis Framework

In this chapter, we first review the *safety* principle that a compiler optimization must obey. Then we introduce the lattice theory which forms the foundation of dataflow analyses. Next, we present a general solver which can be used to solve a system of dataflow equations. We illustrate *forward* and *backward* dataflow analysis techniques through *available expression* and *live variable* problems respectively. Finally, we present an implementation scheme of a general dataflow equation solver on top of the CPC compiler.

6.1 Safe vs. Aggressive in Compiler Optimizations

The purpose of data-flow analysis is to provide global information about how a procedure (or a large segment of a program) manipulates its data. For example, constant-propagation analysis seeks to determine whether all assignments to a particular variable at some particular point necessarily give it the same constant value. If so, a use of the variable at that point can be replaced by the constant.

The spectrum of possible data-flow analyses ranges from program understanding by abstract execution of a procedure, which might determine, for example, that it computes the factorial function, to much simpler and easier analyses such as the reaching-definition problem .

In all cases, we must be certain that a data-flow analysis gives us information that does not misrepresent what the procedure being analyzed does, in the sense that it must not tell us that a transformation of the code is safe to perform that, in fact, is not safe. We must guarantee this by carefully designing the data-flow equations and by being sure that the solution to them that we compute is, if not an exact representation of the procedure's manipulation of its data, at least a conservative approximation of it. For example, for the reaching-definition problem, where we determine what definitions of variables may reach a particular use, the analysis must not tell us that no definitions reach a particular use if there are some that may. The analysis is conservative if it may give us a larger set of reaching definitions than it might if it could produce the minimal result.

However, to obtain the maximum possible benefit from optimization, we seek to pose data-flow problems that are both conservative and, at the same time, as aggressive as we can make them. Thus, we shall always attempt to walk the fine line between being as aggressive as possible in the information we compute and being conservative, so as to get the greatest possible benefit from the analyses and code-improvement transformations we perform without ever transforming correct code to incorrect code.

6.2 Basic Concepts: Lattices, Flow Functions, and Fixed Points

We now proceed to define the conceptual framework underlying data-flow analysis. In each case, a data-flow analysis is performed by operating on elements of an algebraic structure called a *lattice*. Elements of the lattice represent abstract properties of variables, expressions, or other programming constructs for all possible executions of a procedure–independent of the values of the input data and, usually, independent of the control-flow paths through the procedure. In particular, most data-flow analyses take no account of whether a conditional is true or false and, thus, of whether the `then` or `else` branch of an `if` is taken, or of how many times a loop is executed. We associate with each of the possible control-flow and computational constructs in a procedure a so-called flow function that abstracts the effect of the construct to its effect on the corresponding lattice elements.

In general, a *lattice* L consists of a set of values and two operations called *meet*, denoted $\sqcap$, and *join*, denoted $\sqcup$, that satisfy several properties, as follows:

1. For all $x, y \in L$, there exist unique z and $w \in L$ such that $x \sqcap y = z$ and $x \sqcup y = w$ (closure).
2. For all $x, y \in L$, $x \sqcap y = y \sqcap x$ and $x \sqcup y = y \sqcup x$ (commutativity)
3. For all $x, y, z \in L$ $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ and $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$ (associativity).
4. There are two unique elements of L called *bottom*, denoted $\bot$, and *top*, denoted $\top$, such that for all $x \in L$, $x \sqcap \bot = \bot$ and $x \sqcup \top = \top$ (existence of unique top and bottom elements).

Most of the lattices we use have bit vectors as their elements and meet and join are bitwise and and or, respectively. The bottom element of such a lattice is the bit vector of all zeros and top is the vector of all ones. We use $\mathtt{BV}^n$ to denote the lattice of bit vectors of length n.

A function mapping a lattice to itself, written as $f : L \rightarrow L$, is *monotone* if for all x and y $x \sqsubseteq y \Longrightarrow f(x) \sqsubseteq f(y)$. For example, the function $f : BV^3 \rightarrow BV^3$ as defined by

$$f(< x_1 x_2 x_3 >) =< x_1 1 x_3 >$$

is monotone, while the function $g : BV^3 \rightarrow BV^3$ as defined by

$$g(< (000) >) =< 100 >$$

and

$$g(< x_1 x_2 x_3 >) =< 000 >$$

otherwise is not.

The *height* of a lattice is the length of the longest strictly ascending chains in it, i.e., the maximal n such that there exist $x_1, x_2, ..., x_n$ such that

$$\bot = x_1 \sqsubset x_2 \sqsubset .. \sqsubset x_n = \top$$

As for other lattice-related concepts, height may be dually defined by descending chains. Almost all the lattices we use have finite height, and this, combined with

monotonicity, guarantees termination of our data-flow analysis algorithm. For lattices of infinite height, it is essential to show that the analysis algorithm halt.

In considering the computational complexity of a data-flow algorithm, another notion is important, namely, effective height relative to one or more functions. The *effective height* of a lattice L relative to a function $f : L \rightarrow L$ is the length of the longest strictly ascending chain obtained by iterating applying $f()$, i.e., the maximal n such that there exist $x_1, x_2 = f(x_1), x_3 = f(x_2), ..., x_n = f(x_{n-1})$ such that

$$x_1 \sqsubset x_2 \sqsubset x_3 \sqsubset .. \sqsubset x_n \sqsubseteq \top$$

The effective height of a lattice relative to a set of functions is the maximum of its effective heights for each function.

A *flow function* models, for a particular data-flow analysis problem, the effect of a programming language construct as a mapping from the lattice used in the analysis to itself. We require that all flow functions be monotone. This is reasonable in that the purpose of a flow function is to model the information about a data-flow problem provided by a programming construct and, hence, it should not decrease the information already obtained. Monotonicity is also essential to demonstrating that each analysis we consider halts and to providing computational complexity bounds for it.

The programming construct modeled by a particular flow function may vary, according to our requirement, from a single expression to an entire procedure.

A *fixed point* of a function $f : L \rightarrow L$ is an element $z \in L$ such that $f(z) = z$. For a set of data-flow equations, a fixed point is a solution of the set of equations, since applying the right-hand sides of the equations to the fixed point produces the same value. In many cases, a function defined on a lattice may have more than one fixed point.

For formal definitions of lattice, readers are suggested to refer to [HU79, Lid98]. The dataflow analysis theory based on lattice theory can be found in [Kil73, FKU75, KU77].

6.3 Iterative Data-flow Analysis

In order to do code optimization and a good job of code generation, we need to collect information about the program as a whole and to distribute this information to each

block in the control flow graph. For instance, we can use knowledge of global common subexpression to eliminate redundant computations. And this is a good example of *data-flow information* that an optimizing compiler collects by a process known as *data-flow analysis*. We first present an iterative implementation of forward analysis. Methods for backward and bidirectional problems are easy generalizations.

Iterative analysis is the easiest method to implement and, as a result, the one most frequently used. It is also of primary importance because the transformation of the common subexpression elimination needs to be able to do iterative analysis.

We assume that we are given a flowgraph $G =< N, E >$ with `entry` and `exit` blocks in N and desire to compute $in(B), out(B) \in L$ for $B \in N$ where $in(B)$ represents the data-flow information on entry to B and $Out(B)$ represents the data-flow information on exit from B, give by the dataflow equations

$$\begin{aligned} in(B) &= \begin{cases} init & \text{for } B = entry \\ \sqcap_{P \in Pred(B)} \; out(P) & \text{otherwise} \end{cases} \\ out(B) &= F_B(in(B)) \end{aligned}$$

where *init* represents the appropriate initial value for the data-flow information on entry to the procedure, $F_B()$ represents the transformation of the dataflow information corresponding to executing block B, and $\sqcap$ models the effect of combing the data-flow information on the edges entering a block. Of course, this can also be expressed with just $in()$ function as

$$in(B) = \begin{cases} init & \text{for } B = entry \\ \sqcap_{P \in Pred(B)} \; F_P(in(P)) & \text{otherwise} \end{cases}$$

If $\sqcup$ models the effect of combining flow information, it can also be used in place of $\sqcap$ in the algorithm presented in Figure 36. The value of $Init$ is usually $\bot$ or $\top$.

The algorithm given in the Figure 36, uses just $in()$ functions; The strategy is to iterate applications of defining equations given above, maintaining a worklist of blocks whose $in()$ values have changed on the last iteration, until the worklist is empty.

The inputs to this algorithm are:

- a set of CFG nodes N (basic blocks);
- an *entry* node in the CFG;
- initial value *Init* on lattice L;

```
worklist_Iterative(in set of Node : N,
                   in Node        : entry,
                   in Node  --> L : FP,
                   out Node --> L : dfin,
                   in L           : Init)

{
 Node : B, P;
 set of Node : Worklist;
 L    : effect, totaleffect;


    dfin(entry) = Init;

*   worklist = N-{entry};
    for each B in N do
      dfin(B) = U;

    do
    {
*     B = take one from Worklist;
      Worklist -= {B};
      totaleffect = U;
      for each P in Pred(B) do
      {
        effect = F(P, dfin(P));
        totaleffect = combining the data-flow information
                      of totaleffect and effect;
        if dfin(B) <> totaleffect {
          dfin(B) = totaleffect;
*       Worklist = Worklist and {B};
        }
      }
    } while Worklist is empty set;
}
```

Figure 36: Worklist algorithm for iterative data-flow analysis (statements that manage the worklist are S1, S2, S3)

- transformation function FP, $F_b(x)$ is represented by $F(B, x)$;
- *output* function which produces L type of output for each node.

Initially the worklist contains all blocks in the flowgraph except *entry*, since its information will never change. Then we initialize all output nodes (dfin) to **U** (In Figure refworklist, we use **U** to denote $\perp$). Then, we enter a loop which selects a basic block B from the worklist and computes its output based on its predecessor's output. If B's output has been changed, B is put back into the worklist so that its output will have another chance to be recomputed. Since the effect of combining information from edges entering a node is being modeled by $\sqcap$, the appropriate initialization for `totaleffect` is U, representing $\perp$.

The computational efficiency of this algorithm depends on several things: the lattice L, the flow functions $F_B()$, and how we manage the worklist. While the lattice and flow functions are determined by the dataflow problem we are solving, the management of the worklist is independent of it. Note that managing the worklist corresponding to how we implement the statements marked with asterisks in the Figure 36. The easiest implementation would use a stack or queue for the worklist, without regard to how the blocks are related to each other by the flowgraph structure. On the other hand, if we process all predecessors of a block before processing it, then we can expect to have the maximal effect on the information for that block each time we encounter it. This can be achieved by beginning with an ordering we encounter in reverse postorder and continuing with a queue. Since in postorder a node is not visited until all its depth-first spanning-tree successors have been visited, in reverse postorder it is visited before any of its successors have been. If A is the maximal number of back edges on any acyclic path in a flowgraph G, then $A + 2$ pass through the `do-while` loop are sufficient if we use reverse postorder [Muc97]. Note that it is possible to construct flow graphs with A on the order of $|N|$, but that this is very rare in practice. In almost all cases $A \leq 3$, and frequently $A = 1$.

In practice, a system of dataflow equations that relate information at various points has the following typical form:

$$out[S] = (gen[S] \cup in[S]) - kill[S]$$

and can be read as, "the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows

through the statement." Such equations are also called *data-flow equations.*

The details of how data-flow equations are set up and solved depend on three factors.

1. The notions of generating and killing depend on the desired information, i.e., on the dataflow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with the flow of control and defining *out[S]* in terms of *in[S]*, we need to proceed *backwards* and define *in[S]* in terms of *out[S]*. These analysis problems are called backward analysis problems.

2. Since data flows along control paths, data-flow analysis is affected by the control constructs in a program. In fact, when we write *out[S]* we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

3. There are subtleties that go along with such statements as procedure calls, assignments through pointer variable, and even assignments to array variables. However, dealing these issues are beyond the thesis topic.

6.4 Available Expressions

An expression such as $x+y$ is *available* at a point p if every path (not necessarily cycle-free) from the initial node to p evaluates $x+y$, and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y. For available expression we say that a block *kills* expression $x+y$ if it assigns (or may assign) x or y and does not subsequently recomputes $x+y$. A block *generates* expression $x+y$ if it definitely evaluates $x+y$ and does not subsequently redefine x or y.

The primary use of available expression information is for detecting common subexpressions. For example, in Figure 37, the expression $4*i$ in block B3 will be a common subexpression if $4*i$ is available at the entry point of block B3. It will be available if i is not assigned a new value in block B2, or if, as in Figure 37(b), $4*i$ is recomputed after i is assigned B2.

We can easily compute the set of generated expression for each point in a block, working from beginning to end of the block. At the point prior to the block, assume

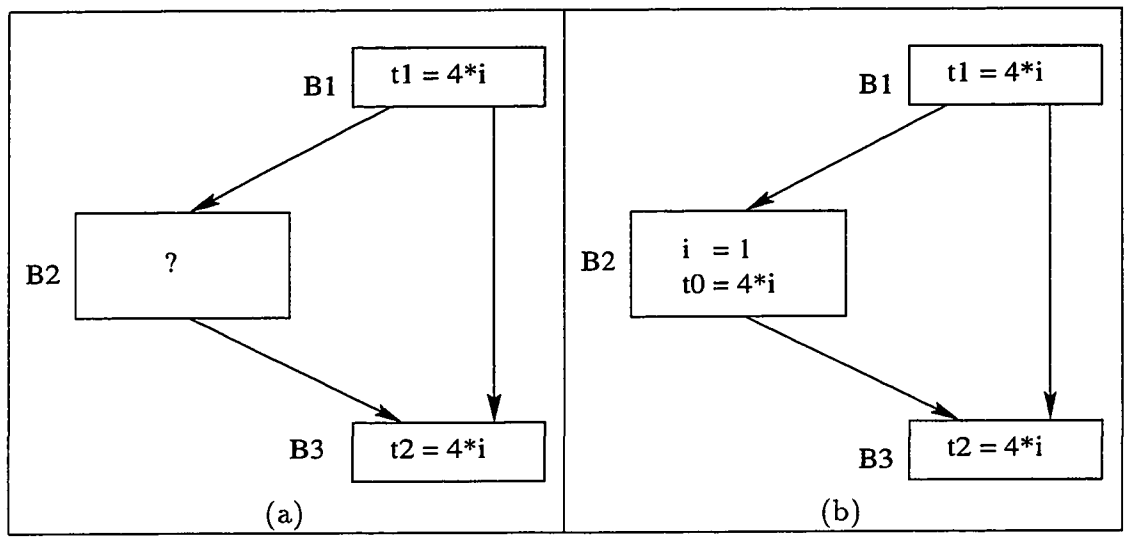


Figure 37: A potential common subexpression across blocks

no expressions are available. If at point p set A of expressions is available, and q is the point after p, with statement x:=y+z between them, then we form the set of expressions available at q by the following two steps.

1. Add to A the expression y+z;
2. Delete from A any expression involving x.

Note the steps must be done in the correct order, as x could be the same as y or z. After we reach the end of the block, A is the set of generated expressions for the block. The set of killed expressions is all expressions, say, y+z such that either y or z is defined in the block, and y+z is not generated by the block.

Consider the four statements listed in Figure 38. After the first, b+c is available. After the second, a-d becomes available, but b+c is no longer available, because b has been redefined. The third does not make b+c available again, because the value of c is immediately changed. After the last statement, a-d is no longer available, because d has changed. Thus no statements are generated, and all statements involving a, b, c, or d are killed.

Suppose U is the "universal" set of all expressions appearing on the right of one or more statements of the program. For each block B, let *in[B]* be the set of expressions

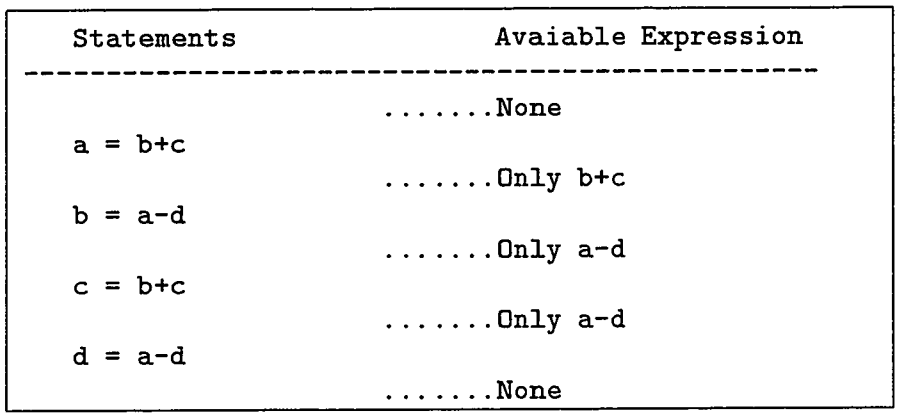


Figure 38: Computation of available expressions.

in U that are available at the point just before the beginning of B. Let *out[B]* be the same for the point following the end of B. We define *e_gen[B]* to be the expressions generated at B and their operands have not been modified before their usages; define *e_kill[B]* to be the set of expressions in U killed in B, i.e, any expression whose operands have been modified by B.

The following equations relate the unknowns *in* and *out* to each other and the known quantities *e_gen* and *e_kill*.

$$
\begin{aligned}
out[B] &= (in[B] \cup e\_gen[B]) - e\_kill[B] \\
in[B] &= \cap_{(\text{P is a predecessor of B})} (out[P]) \quad \text{for B is not } entry \\
in[entry] &= \emptyset \quad (\text{where } entry \text{ is the initial block})
\end{aligned}
$$

Here we should first notice that *in* for the initial node (*entry*) is handled as a special case. This is justified on the grounds that nothing is available if the program has just begun at the initial node, even through some expression might be available along all paths to the initial node from elsewhere in the program. If we did not force *in[entry]* to be empty, we might erroneously deduce that certain expressions were available before the program started. The second, the confluence operator is intersection. This operator is the proper one because an expression is available at the beginning of a block only if it is available at the end of all its predecessors.

It may not be obvious that by starting with the assumption "everything, i.e the set U, is available everywhere" and eliminating only those expressions for which we can discover a path along which it is not available, we do reach a set of truly available

expression. In the case of available expressions, it is conservative to produce a subset of the exact set of available expressions, and this is what we do. The argument for subsets being conservative is that our intended use of the information is to replace the computation of an available expressions by a previously computed value (see the algorithm of computing CSE in the next Chapter), and not knowing an expression is available only inhibits us from changing the code.

In Figure 39, there is a cycle surrounding B2, which represents a loop in the CFG. We shall concentrate on a single block B2, to illustrate the effect of the initial approximation of *in[B2]* on *out[B2]*. Let G and K abbreviate *gen[B2]* and *kill[B2]*, respectively. The data-flow equations for block *B2* are:

$$\begin{aligned} in[B2] &= out[B1] \cap out[B2] \\ out[B2] &= (G \cup in[B2]) - K \end{aligned}$$

These equations have been rewritten as recurrences in Figure 39, with I^j and O^j being the jth approximations of *in[B2]* and *out[B2]*, respectively. The figure also shows that starting with $I^0 = \emptyset$ we get $O^1 = O^2 = G$, while starting with $I^0 = U$ we get a large set for O^2. Note that *out[B2]* equals O^2 in each case, because the iterations each coverage at the points shown.

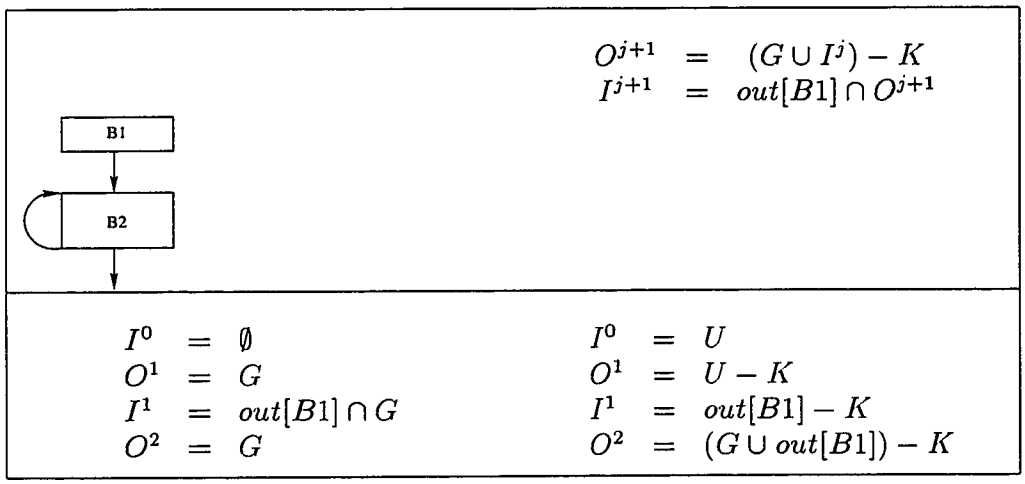


Figure 39: Initializing the *in* sets to $\emptyset$ is too restrictive

Intuitively, the solution obtained starting with $I^0 = U$ using

$$out[B2] = (G \cup out[B1]) - K$$

is more desirable, because it correctly reflects the fact that expressions in *out[B1]* that are not killed by *B2* are available at the end of *B2*, just as the expressions generated by *B2* are.

Now we give the details of the algorithm.

Algorithm. Available Expression

Input. A flow graph G with *e_kill[B]* and *e_gen[B]* computed for each block *B*. The initial block is *entry*.

Output. The set *in[B]* for each block B.

Method. We use an iterative approach, starting with the "estimate" $in[entry] = \emptyset$ and converging to the desired values of in and out finally. As we must iterate until the *in*'s and hence the *out*'s converge, we use a boolean variable *change* to record on each pass through the blocks whether any *in* has changed. The algorithm is sketched in Figure 40.

```
(1)  in[entry] = Empty-set;
(2)  out[entry]= Empty-set; /* in and out never change for the entry node */
(3)  for (B != entry) do
(4)    out[B] = U - e_kill[B];  /* initial estimate is too large */
(5)  change = true;
(6)  while change do
(7)  { change = false;
(8)    For each basic block B do
(9)    { in[B]  = intersection of (out[P]) (P, for all predecessors of B);
(10)     oldout = out[B];
(11)     out[B] = (e_gen[B] union in[B]) - e_kill[B];
(12)     if (out[B] != oldout)
(13)       change = true;
       }
     }
```

Figure 40: Available expressions computation

We can see that the algorithm will eventually halt because *out[B]* never decrease in size for any *B*; once a definition is added, it stays there forever. Since the set of all definitions is finite, there must eventually be a pass of the while-loop in which *oldout*

= *out[B]* for each B at line (12). Then *change* will remain `false` and the algorithm terminates. We are safe terminating then because if the *out's* have not changed, the *in's* will not change on the next pass. And, if the *in's* do not change, the *out's* cannot, so on all subsequent passes there can be no changes.

6.5 Live-Variable Analysis

A number of code improving transformations depend on information computed in the direction opposite to the flow of control in a program; we shall consider one now. In *live-variable* analysis we wish to know for variable `x` and point `p` whether the value of `x` at `p` could be used along some path in the control flow graph starting at `p`. If so, we say `x` is *live* at `p`; otherwise `x` is *dead* at `p`.

An important use for live-variable comes when we generate object code. After a value is computed in a register, and presumably used within a block, it is not necessary to store that value if it is dead at the end of the block. Also, if all registers are full and we need another register, we should favour using a register with a dead value, since that value does not have to be stored.

In contrast to the *available expression* analysis, we define *in[B]* to be the set of variables live at the point immediately *after* block B and define *out[B]* to be the same at the point immediately *before* block B. Therefore, we actually use the control flow graph in a *backward* way. Let *def[B]* be the set of variables definitely assigned values in B prior to any use of those variable in B, and let *use[B]* be the set of variables whose values are used in B prior to any definition of the variables. Then the equations relating *def* and *use* to the unknowns *in* and *out* are:

$$\begin{aligned} out[B] &= use[B] \cap (in[B] - def[B]) \\ in[B] &= \cup_{(\text{S is a successor of B})} out[S] \\ in[exit] &= \emptyset \end{aligned}$$

The first group of equations says that a variable is live coming into a block if either it is used before redefinition in the block or it is live coming out of the block and is not redefined in the block. The second group of equations says that a variable is living coming out of a block if and only if it is live coming into one of its successors.

Compared to the dataflow equations used in computing available expressions, these are the differences:

- This is a backward analysis problem. The searching (fixed points) direction is from *exit* to *entry* and the output of successors of a node rather than that of predecessors are used to compute the input set of the node.
- The "meet" operator is set union.
- The initial value for *exit* node is empty since no variable will be "live" after a function finishes its execution.

The similar algorithm used to compute available expressions can be adapted to compute live variables through changing the algorithm with consideration of the above differences accordingly.

6.6 Sets, Operators, and General Solver

In Section 6.3, we have introduced an abstract algorithm which can be used to solve any dataflow equation.

To implement such an abstract algorithm, we need to specify the followings:

- How to represent *in* and *out* on Lattice L?
- How to manager the work list?
- How to construct the "meet" operator on L?
- How to know the search direction?

In this section, we will describe an iterative dataflow solver built based on set and its operations.

6.6.1 Abstract Sets and Meet Operators

Sets are the basic concept used in describing data-flow equations, such as *in* and *out* sets. They are naturally selected as the mapping objects of program properties. On the other hand, the power of sets and set operators forms a lattice L. Therefore, an abstract set element can model dataflow analysis results while sets itself can be used to represent the *in* and *out* of a dataflow equation. In addition, set operations can

be used to implement the "meet" operator. All these make sets and their operators form the core of an iterative data-flow equation solver.

There are many ways to represent sets, *forests*, *bit-vectors*, and *linked list* are named as a few. In our current implementation, we use the linked list representation, because it has low implementation costs and is flexible to represent any set of objects.

Based on the linked list representation, we can implement various set operations such as *intersection*, *union*, *member*, and *insert*. These operations are used to support the *meet* operators used in both the forward and backward data-flow analyses, such as computing available expression and live variable analysis.

6.6.2 General Solver

Our general dataflow analysis framework is based on a general dataflow equation solver, which takes a specification of a system of dataflow equations and performs the corresponding dataflow analysis. The dataflow equation specification consists of:

- Direction of analysis: *forward* or *backward*.
- Set merge operator: *intersection* or *union*.
- Gen set: the results generated by a statement.
- Kill set: the results killed by a statement.
- The precedence between *gen* and *kill* sets: which one is applied first.

Depending on such a data-flow analysis specification, the general solver will combine the appropriate set operators and operate on related fields of each basic block to perform dataflow analysis.

For example, computing available expression is a *forward* analysis and uses set *intersection* to merge results from predecessors. The *gen* and *kill* sets are the ones used in Section 6.4.

Please recall that there are two linked lists in our basic block representation (see Chapter 5). These lists are designed to support the general dataflow equation solver.

Still taking the available expression analysis as example, the *in* set of a basic block is computed as an intersection of all *out* sets of its predecessors. The solver then needs to visit the *out* fields of all predecessors of the basic block. This can be done easily

because the predecessors of a basic block is stored as a linked list and the header is pointed by *pred* field of the basic block.

For the available expression analysis, within a basic block the solver walks along the *next* field of a statement linked list, whose header is pointed by the *front* field of the basic block. For each statement, it will call the corresponding *gen* and *kill* routines to its compute available expressions. The details of the linked lists are introduced in Section 6.7.

For managing the worklist, we also delay its description until Section 6.7 since some user code is necessarily embedded to make the worklist management clear.

Please note that some interfaces between the general solver and data-flow specifications are C routines, such as *gen* and *kill* sets. Users still need to write certain C code to use the solver. However, the framework provided can make user's life easier in developing a particular data-flow analysis program.

6.7 CSE Example

In this section, we use the common subexpression analysis as an example to show how to use the general framework.

6.7.1 Data Structures for Solving CSE

As we mentioned before, the statements within a basic block are linked by a double-linked list. Figure 41 shows its structure definition.

Every statement-list element contains the following fields:

- *type:* there are two type for a statement list element,
  - *EXPR:* the type of the statement list is C_EXPR in CPC-AST.
  - *STMT:* any statement which is not C_EXPR in CPC-AST.
- *stmt_ptr:* points to the statement node of CPC-AST.
- *expr:* when the statement list is EXPR, it points to the expression node of CPC-AST.
- *prev:* pointer pointing to the previous statement list element.

```
enum stmt_ele_type { STMT, EXPR } ;

#define DataflowInfoSize 1
#define CommonSubExpr    0
#define ReachingDef      1              /* Not used */
#define LiveVars         2              /* Not used */

typedef struct stmtlist {
  enum stmt_ele_type    type ;
  struct stmt           *stmt_ptr;
  struct value          *expr;
  struct stmtlist       *prev;
  struct stmtlist       *next;
  void                  *dataflow[DataflowInfoSize];
} stmtlist ;
```

Figure 41: Data structure of the statement list in a BB

- *next:* pointer pointing to the next statement list element.
- *dataflow[DataflowInfoSize]:* store dataflow analysis information. This is an array representation, storing different dataflow analysis results.

Due to the double linked list representation of the statement sequence within a basic block, both forward and backward analyses can be supported in an equal foot manner. We use a *void pointer* type to denote any potential result of dataflow analysis. It is the user's responsibility to do the appropriate casting to deposit to and retrieve from any dataflow information through the fields. The above representation for *statement* is general enough to support any dataflow analysis.

In order to support the CSE transformation (see Chapter 7), a special data structure is designed to hold a list of most recent definition points for a common subexpression. This structure is particular to the CSE transformation and it should be provided by user programmers.

In Figure 42, every definition list element contains the following fields:

- *s_def:* pointer points to the statement in the original CPC-AST node, in which the common expression is most recently being computed.

```
typedef struct def_list {
   struct stmt     *s_def;      /* Defition point */
   struct value    *val;
   struct def_list *next;
} def_list;
```

Figure 42: Data structure of definition list for CSE

- *val:* pointer points to the expression of the statement node which contains the recent definition expression.
- *next:* pointer points to the next definition list element.

To compute common subexpressions, we also designed a special expression list. In Figure 43, every expression list element contains the following fields:

- *val:* pointer points to the expression field of a statement.
- *list:* pointer points to the statement of definition point.
- *next:* pointer points to the next element.

```
typedef struct expr_set {
   struct value    *val;
   struct def_list *list;       /* Defition point */
   struct expr_set *next;
} expr_set ;
```

Figure 43: Data structure of expression sets

The *expr_set* structure is used to represent *in* and *out* sets. The data structure of the set header descriptor is as follows.

In Figure 44, every set header contains the following fields:

- *type:* there are two types of set header:

```
enum set_type {Empty, Full} ;

typedef struct set_header {
  enum set_type type;
  expr_set     *set;
  int           count;
} set_header;
```

Figure 44: Data structure of set header.

- *EMPTY:* the set is empty when there is no CSE information for the current statement.
- *FULL:* the set is an "universal" set which contains all common subexpressions in a program.

These two values are very useful in initializing the *in* or *out* sets.

- *set:* store the common subexpressions.
- *count:* the total number of definition point set within this set header.

6.7.2 Algorithm for Solving CSE

In Figure 45, we list the main control algorithm of our iterative solver for solving the CSE problem based on the CPC AST representation. Except for statement $S5$ which is special to the CSE analysis and will be introduced in Chapter 7, other parts can be used in any dataflow analysis.

After having the size of a CFG, the main control algorithm starts by initializing each basic block. For the CSE problem, we can initialize set *in* empty and set *out* universal. Then, the algorithm enters a *while loop*, which will only stop when the *ins* or *outs* are not changed any more.

Please note that for each iteration, we need to do extra initialization such as the one done at statement $S3$. It sets each node unvisited, preparing for the search to be started from the scrash. The real search of the CFG is performed in function *search_bb*.

```
solving_cfg(symbol *sym)
{
    bb_num  = graph-> num;        /* get the total BB number of a CFG */
    bb_base = graph-> bbs_array; /* get the BB of the CFG */
    /* initialize the input of each BB is EMPTY */
S1: initial_cfg_sets(bb_base, bb_num, 1, 1);
    /* bb_0 is reserved for FUNCTION END */
    /*get the first BB of the cfg_graph*/
    first_bb = bb_base + 1;
S2: while (changes) {      /* while the input of BB has been changed */
      changes = 0;         /* assume no changes for input of BB at first */
S3:   initial_search();    /* initialize each BB as not visited,
                              and the queue as empty */
      enqueue(first_bb);   /* put the first BB in the queue */
S4:   search_bb(bb_base, 1);   /* visit each BB in the CFG */
    }
S5: compute_cse(bb_base, bb_num);/*computer CSE */
}
```

Figure 45: Algorithm for solving CSE

In Figure 46, we list the search algorithm to solve the CSE dataflow analysis.

There is a useful ordering of the nodes of a flow graph, know as *breath-first* ordering, which is generalization of the breath-first traversal of a tree. A breath-first ordering can be used to detect loops in any flow graph; it also helps speed up iterative data-flow algorithm such as available expression. The breath-first ordering is created by starting at the initial node and searching the entire graph, trying to visit a node before its "successors". Of course, this definition is somehow inappropriate if there is a cycle in the flowgraph. However, it is correct if we eliminate the back edges (in our mind) and make the flowgraph a Direct Acyclic Graph(DAG).

In Figure 46, we list *breath-first* search algorithm to search all basic blocks of a CFG based on the CPC-AST-representation. The parameter *dir* is used to indicate the search direction. The algorithm performs as follows:

- At line 1 when there is a BB in the queue, we take it out. The worklist is organized as a queue in our framework.
- From line 4 to 10, depending on the search direction, the algorithm selects the appropriate fields. For example, if the problem is a forward problem, predecessor and successor fields will be set the *pred* and *succ* variables accordingly.

```
search_bb(bb_cfg *bb_base, int dir)
{
1: while (bb_ptr=dequeue()) { /* Take out a BB from the queue */
2:   i  = bb_ptr - bb_base;
3:   bb_visited[i] = 1;       /* Mark the BB as visited */
4:   if (dir) {               /* For a forward data-flow analysis */
5:     pred = BBPred(bb_ptr);
6:     succ = BBSucc(bb_ptr);
     }
7:   else {                   /* For a backward data-flow analysis */
8:      pred = BBSucc(bb_ptr);
9:      succ = BBPred(bb_ptr);
10:  }
11:  if (pred)      /* merge inputs by the direction */
12:     in = merge_inputs(pred, dir);
13:  else
14:     in = (void *) creat_set(Empty);

15:  if (dir) {    /* forward data-flow analysis */
16:    BBIn(bb_ptr) = in;
17:    first = BBStmt(bb_ptr);
18:    prev_out = &BBOut(bb_ptr);
19:  }
20:  else {       /* backward data-flow analysis */
21:    BBOut(bb_ptr) = in;
22:    first = BBLastStmt(bb_ptr);
23:    prev_out=&BBIn(bb_ptr);
24:  }
25:  in  = copy_exist_set(in);
26:  out = available_expr(in, first);  /*compute available expression  */
27:  if (!two_sets_same(*prev_out, out)) { /* same as before ? */
28:     changes = 1;
29:     *prev_out=out;
       /* Breadth First Order Searching */
30:     while (succ) {
          /* find next successor */
31:       temp_ptr = BBNode(succ);
32:       i        = temp_ptr - bb_base;
33:       if (bb_visited[i] == 0) /* if the BB has not been visited*/
34:         enqueue(temp_ptr);    /* put the BB in the queue*/
35:       succ = NextEdge(succ);  /* go to next successor */
36:    }
37:  }
38:}
}
```

Figure 46: Algorithm of searching BB for solving CSE

- From line 11 to 14, the *in* set is computed based on the *dir* and the above selection.
- From line 15 to 24, the computed *in set* is put back and an appropriate *out set* is selected.
- Line 26 actually computes available expressions.
- Starting line 27, if we detect a difference between two *out* sets, the new one is put back and loop iteration flag *change* is set so that the main control algorithm will search the CFG again.
- Line 30 – 36 implements a "breadth-first' search by adding the successors of the node into the queue.

Please note that except the available expression computation, other parts can be applied to any dataflow analysis problem. Therefore, within the framework, a user needs only to provide his own analysis C routines to compute the corresponding dataflow analysis. The worklist management (queue plus breadth-first search in this case) and the selection of right fields are automatically controlled by the general solver. This will greatly facilitate users to develop their own dataflow analysis programs.

Chapter 7

CSE Transformation

Now that we have the mechanism to determine control flow and data flow information, we next consider optimizations that may be valuable in improving the performance of the object code produced by a compiler.

The optimization covered in this chapter deals with common-subexpression elimination, which finds computations that are always performed at least twice on a given execution path and eliminates the second and later occurrences of them. This optimization requires data-flow analysis to locate redundant computation and almost always improves the performance of programs it is applied to.

7.1 Common-Subexpression Elimination

An occurrence of an expression in a program is a *common subexpression* if there is another occurrence of the expression whose evaluation always precedes this one in execution order and if the operands of the expression remain unchanged between the two evaluations. The expression $a + 2$ in blocks *B3* in Figure 47(a) is an example of a common subexpression, since the occurrence of the same expression in *B1* always precedes it in execution and the value of a is not changed between them. *Common-subexpression elimination* is a transformation that removes the re-computations of common subexpressions and replaces them with uses of saved value. Figure 47(b) shows the result of transforming the code in Figure 47(a). Note that, as this example shows, we cannot simply substitute b for the evaluation of $a + 2$ in block *B3*, since *B2* changes the value of b if it is executed.

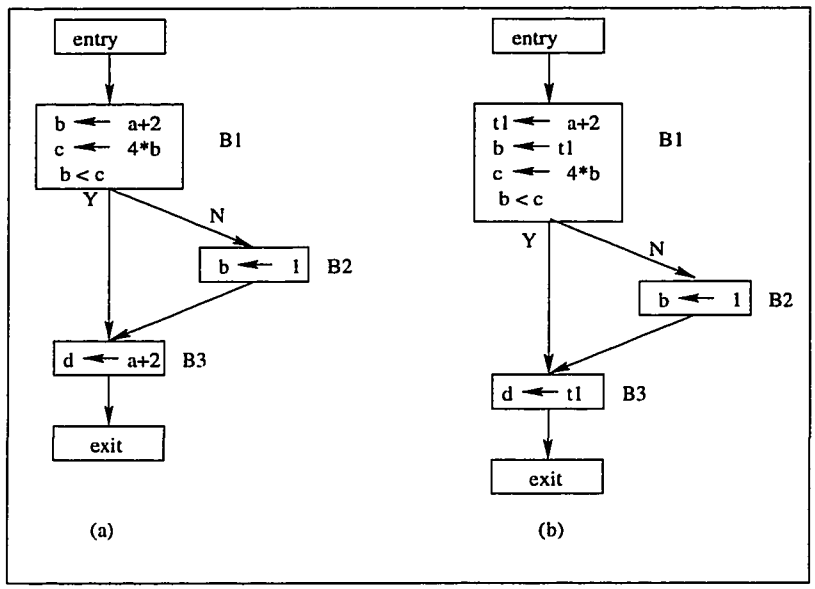


Figure 47: (a) Example of a common subexpression, namely, a+2, and (b) The result of doing common-subexpression elimination on it

Note that common-subexpression elimination may not always be worthwhile. In this example, it may be less expensive to recompute $a + 2$ (especially if a and d are both allocated to registers and adding a small constant value in a register can be done in a single cycle), rather than to allocate another register to hold the value of $t1$ from *B1* through *B3*, or, even worse, to store it to memory and latter reload it. Actually, there are more complex reasons why common-subexpression elimination may not be worthwhile that have to do with keeping a super-scalar pipeline full or getting the best possible performance from a vector or parallel machine. In chapter 8, we will show that some of the tested benchmarks do not achieve the expected performance improvement, probably due to those reasons.

Optimizers frequently divide common-subexpression elimination into two phases, one local, done within each basic block, and the other global, done across an entire control flowgraph. The division is not essential, because global common-subexpression elimination catches all the common subexpression that the local form does and more, but the local form can often be done very cheaply while the intermediate code for a basic block is being constructed and may result in less intermediate code being produced.

7.2 Global Common-subexpression Elimination

As indicated above, global common-subexpression elimination takes as its scope a control flowgraph representing a procedure. It solves the data-flow problem know as *available expressions*, which we discussed briefly in Chapter 6 and which we now examine more fully. An expression *exp* is said to be *available* at the entry to a basic block if along every control-flow path from the entry block to this block there is an evaluation of *exp* that is not subsequently killed by having one or more of its operands assigned a new value.

In determining what expressions are available, we use EVAL(i) to denote the set of expressions evaluated in block i that are still available at its exit and KILL(i) to denote the set of expressions that are killed by block i. To compute EVAL(i), we scan block i from begin to end, accumulating the expressions evaluated in it and deleting those whose operands are later assigned new values in the block. An assignment such as $a \leftarrow a + b$, in which the variable on the left-hand side occurs also as an operand on the right-hand side, *does not* create an available expression because the assignment happens *after* the expression evaluation.

The expression $a + b$ is also evaluated in the block, but it is subsequently killed by the assignment $a \leftarrow j + a$, so it is not in the EVAL() set for the block. $KILL(i)$ is the set of all expressions evaluated in other blocks such that one or more of their *left-hand* operands are assigned to in block i, or that are evaluated in block i and subsequently have one or more of its *left-hand* operands been assigned to in block i. To give an example of a $KILL()$ set, we need to have an entire control flow graph available, so we will consider the flowgraph in Figure 48. The $EVAL(i)$ and $KILL(i)$ sets for the basic block are as follows:

$EVAL(entry) = \emptyset$		$KILL(entry) = \emptyset$	
$EVAL(B1) = \{a+b, a*c, d*d\}$		$KILL(B1) = \emptyset$	
$EVAL(B2) = \{a+b\}$		$KILL(B2) = \{c*2, a*c\}$	
$EVAL(B3) = \{a*c\}$		$KILL(B3) = \emptyset$	
$EVAL(B4) = \{d*d\}$		$KILL(B4) = \emptyset$	
$EVAL(B5) = \emptyset$		$KILL(B5) = \{i+1\}$	
$EVAL(exit) = \emptyset$		$KILL(exit) = \emptyset$	

Now, the equation system for the data-flow analysis can be constructed as follows. This is a forward-flow problem. We use $AEin(i)$ and $AEout(i)$ to represent the sets

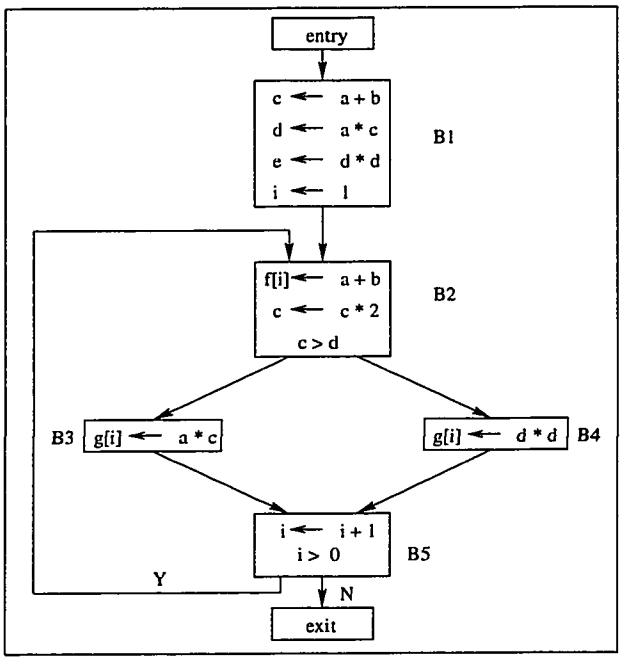


Figure 48: Example flowgraph for global common subexpression elimination

of expressions that are available on entry to and exit from block i, respectively. An expression is available on entry to block i if it is available at the exits of all predecessor blocks, so the path-combining operator is set *intersection*. An expression is available at the exit from a block i if it is either evaluated in the block and not subsequently killed in it, or if it is available on entry to the block and not killed in it. Thus the system of data-flow equations is

$$\begin{aligned} AEin(i) &= \cap_{j \in Pred(i)} AEout(j) \\ AEout(i) &= (EVAL(i) \cup AEin(i)) - KILL(i) \end{aligned}$$

In solving the data-flow equations, we initialize $AEout(i) = U_{exp}$ for all blocks i, where U_{exp} can be taken to be the universal set of all expressions, or as it is easy to show,

$$U_{exp} = \cup_i EVAL(i)$$

is sufficient.

For our example in Figure 48, we use

$$U_{exp} = \{a + b, a * c, d * d, c * 2, i + 1\}$$

The first step of a worklist iteration procedures

$$AEin(entry) = \emptyset$$

and the second step produces

$$AEin(B1) = \emptyset$$

Next, we compute

$$\begin{aligned} AEin(B2) &= \{a + b, a * c, d * d\} \\ AEin(B3) &= \{a + b, d * d\} \\ AEin(B4) &= \{a + b, d * d\} \\ AEin(B5) &= \{a + b, d * d\} \\ AEin(exit) &= \{a + b, d * d\} \end{aligned}$$

There are changes for the input of each block. At end, we get

$$\begin{aligned} AEin(B2) &= \{a + b, d * d\} \\ AEin(B3) &= \{a + b, d * d\} \\ AEin(B4) &= \{a + b, d * d\} \\ AEin(B5) &= \{a + b, d * d\} \\ AEin(exit) &= \{a + b, d * d\} \end{aligned}$$

And additional iterations produce no further changes.

Next, we describe how to perform global common subexpression elimination using the $AEin(i)$ data-flow function. We proceed as follows:

For each block i and each expression $exp \in AEin(i)$ that is evaluated in block i,

1. Locate the first evaluation of exp in block i.
2. Search backward from the first occurrence to determine whether any of the operands of *expr* have been previously assigned to in the block. If so, this occurrence of *expr* is not a global common subexpression; proceed to another expression or another block as appropriate.

3. Having found the first occurrence of *expr* in block i and determined that it is a global common subexpression, search backward in the control flowgraph to find the occurrences of *expr*, such as in the context $v \leftarrow exp$, that caused it to be in $AEin(i)$. These are the final occurrences of *expr* in their respective blocks; each of them must flow unimpaired to the entry of block i; and every flow path from the *entry* to block i must include at least one of them.

4. Select a new temporary variable ti. Replace the first instruction ins that uses exp in block i by ti and replace each instruction that uses exp identified in step(3) by

$$\begin{aligned} ti &\leftarrow exp \\ v &\leftarrow ti \end{aligned}$$

In summary, algorithms for performing the code-improving transformations introduced before rely on data-flow information. We also have seen how this information can be collected. Here we consider common subexpression elimination and transformations for removing redundant computation. There are other transformations such as loop invariant removal and eliminating induction variables which can influence performance. If more than one transformation is implemented in a compiler, it is possible to perform some of the transformations together. However, we shall present the ideas underlying the transformations using CSE only.

The emphasis in this section is on global transformations that use information about a function as a whole. Global data-flow analysis does not usually look at within basic blocks. Global transformations are therefore not a substitute for local transformations; both must be performed. For example, when we perform global common subexpression elimination we shall only be concerned with whether an expression is generated by a block and not with whether it is recomputed several times within a block.

7.3 Perform the CSE Transformation Based on CPC-AST

In this section, we will describe how to perform the CSE transformation based on *available expressions* computed by our general dataflow solver and thus perform the

corresponding transformations for programs represented in CPC-AST.

7.3.1 Available Results of CSE Dataflow Analysis

As described in chapter 6, we can use our general dataflow equation solver to solve the system equations for *available expressions*. The results of such dataflow analysis is stored not only in each basic block but also on each statement (assignment in this case). The information stored on each assignment can be used to perform the CSE transformation.

To perform step 3 of the algorithm described in previous section, which traces back the CFG to find the occurrences of an expression, for each assignment we use a linked list, *def_list*, to denote all most recent definition points of the expression, which appears in the right hand of the current assignment. In so doing, we only need to look into the list to find out all occurrences of the expression.

Therefore, the following transformation algorithm used for the CPC-AST representation is based on the dataflow analysis results of *def_list*.

7.3.2 Data Structures for CSE Transformation

After *def_list* is computed, we can do transformation to eliminate common subexpression in a CPC-AST. To distinguish the first occurrence of an expression, we need add two new fields in the statement node [Tao96] as following:

- visited: a field specifies if the statement has been visited or not.
- new_sym: a field points to a new symbol such as *ti* generated.

Assume a statement S computes an expression $v = exp$ and S appears in two *def_lists* of two later assignments S_1 and S_2. Suppose S_1 is first processed and after performing the CSE transformation, the original code will be changed to:

$$\begin{aligned} S: &\quad ti &= &\quad exp; \\ S': &\quad v &= &\quad ti; \\ &\quad \vdots & & \\ S_1: &\quad \ldots &= &\quad ti; \end{aligned}$$

By setting the *visited* field of the statement node of S, we can avoid S being transformed again when statement S_2 is processed.

The *new_sym* field is used to hold the newly created variable holding the common expression. After S_1 has been processed, the *new_sym* field of S will point to ti. Then when S_2 is processed, we can simply refer to this newly created variable and change S_2 into:

$$S_2 : \quad \ldots = ti;$$

7.3.3 Algorithm for the CSE Transformation

In Figure 49, we list the algorithm performing the CSE transformation based on the CPC-AST representation.

During the transformation, we walk through a CPC AST and change tree nodes directly.

A function definition is represented as a symbol, which is the input to function *transform_cse*. At the beginning of *transform_cse*, we need to know the total number of basic blocks in the CFG graph and the first basic block of the CFG graph. Then we start to visit each basic block to do the CSE transformations until the last basic block.

Then, we visit each statement and do CSE transformation only for *assignment* statements. Before we start to transfer an assignment to a new form, we need to perform the following checks step by step:

- following the CPC AST *stmt_list* to make sure the statement is an expression and the expression is an assignment.
- following the *def_list* to make sure there is a CSE definition for the statement.
- following the statement list to make sure the statement has not been visited yet.
- assume the current statement is `t=x+y` and it has not been visited. If its CSE *def_list* is not empty, we creat new symbol (new_sym) to represent the common subexpression such as `(x+y)` and insert it into the current function declaration. Then we creat a new statement like `new_sym = x+y`, and insert it before the original definition statement. After this, we change the current statement to: `t = new_sym`. If the statement found in the *def_list* has been visited, we simply

```
transform_cse(SYMBOL *sym, bb_cfg *bb_base, int bb_num)
{
  temp_num = 0;/*use for creating a new symbol of first operand of CSE*/
  for (i=1; i<bb_num; i++)  {/*while there is BB in the cfg_graph  */
    bb_ptr    = bb_base + i;
    first_stmt = BBStmt(bb_ptr);/*get the first statement from the current BB */
    while (first_stmt) {     /* while there is a statement(s=x+y) in the BB */
      if (IsExpr(stmts)) {   /* the statement is  an expression? */
        v = StmtExpr(stmts);   /* get the expression*/
        left_expr = v->expr1;  /* get the first operand of the expression*/
        def_s = (def_list*)InfoCSE(stmts);/*get definition point of
                                        CSE of the expression*/
        if  (def_s) {        /*if there is CSE for the expression */
          if (!StmtDef(def_s)->visited){/*if the statement never being visited
                                  in the definition list*/
            temp_num = temp_num+1;
            left_sym = left_expr->sym;
            new_sym  = creat_temp(left_sym, temp_num);/*creat a new symbol*/
            adding_temp(sym, new_sym); /*add the 'new_sym' in the declaration*/

            /*creat a new statement 'new_sym=x+y'*/
            new_stmt = creat_stmt(new_sym,v);

            /*add the new statement 'new_sym=x+y' before the current statement*/
            adding_stmt(StmtDef(def_s), new_stmt);

            /*transform the statement 's=x+y' to the 's=new_sym' */
            do_transform(StmtStmt(stmts), new_sym);
          }

          /*remember the new symbol of CSE*/
          new_sym = StmtDef(def_s)->new_sym;

          /*get the next definition point from the definition list*/
          def_s = NextDef(def_s)

          /*while there is still a definition point in the definition list*/
          while (def_s){
            if (StmtDef(def_s)!= NULL)
              /* transform the statement 't=x+y' to 't=new_sym' */
              do_transform(StmtStmt(stmts),new_sym);
            def_s = NextDef(def_s);
          }
        }
      else   /* there is no CSE */
        printf("%s has no cse\n",  c_value_str(v));
      }
      stmts = NextStmt(stmts);  /*get the next statement from the current BB*/
    }
  }
}
```


Figure 49: Algorithm for the CSE transformation

do the transformation `t = new_sym`, where new_sym can be retrieved from the new_sym filed of the statement node.

Chapter 8

Experimental Results and Analysis

We have implemented an iterative dataflow-analysis solver in the CPC compiler. Using this framework, we have implemented an optimization transformation, *common sub-expression elimination* (CSE). We report our experiments on four *sequential* benchmarks, running on Sun Sparc workstations and the CPSS simulator. Our experimental results show that for some benchmarks the optimizing transformation achieves a significant improvement over the unoptimized programs on both Sun workstations and the simulator.

In the following, we first introduce the benchmarks used, then describe two experimental environments and methods. Finally we analyze the optimization effects on performance.

8.1 Benchmarks

We have implemented four benchmarks: *quick-sort*, *double-loop*, *single-loop*, and *my-suite* to test effectness of the CSE optimization.

- **Quicksort**

  Quicksort uses recursive sorting algorithm [CLR90, C.A62]. This algorithm involves splitting an array of data through a 'middle' point, called pivot. The algorithm first partitions the array into two parts so that items to the pivot's left are smaller than the pivot, and the items to the right are bigger. The

algorithm is then called recursively so that it will partition the two subordinate arrays on either side of the pivot until the entire array is sorted.

In the partition phase, two loop index variables are used to scan the loop, searching for the location of a pivot. One index is used for scanning from low to high direction and the other is used from high to low. The pivot is detected when both indexes are "met" in between. This loop based scanning involves many array references and it provides opportunities for the CSE optimization.

To simulate the address calculation of array reference, we translate the integer based array index reference into a low-level byte based address-calculating format. In so doing, many CSE optimization opportunities are exposed to our compiler.

For example, the address calculation of `a[i]` is transformed as:

```
t1 = sizeof(int) * i;
t2 = base + t1;
```

By exposing address calculation such as $t1 = sizeof(int) * i$, common subexpressions can be found more easily by our compiler.

- **double-loop**

  There is a double *for-loop*, in which CSEs appear in the innerest loop. We expect to see that the CSE optimization can have a significant improvement on performance.

- **single-loop** There is a single *for-loop*, and its loop body consists of a conditional statement. CSEs appears in both arms of the *if* statement. Due to the dynamical nature of branch, we expect to see a marginal performance improvement for this benchmark.

- **my-suite**

  This benchmarks consists of various C statements such as `case`, `do-while`, `goto` statements. It comes from the CPSS benchmark suite, and it was used mainly for testing the robustness of the CPC compiler. Since many loops exist in this benchmark, we also expect to see a significant improvement on performance.

8.2 Experiment Setting

Our experiments have been done using the CPSS simulator and a Sun Ultra-sparc workstation running Solaris 2.5.1. On a Sun workstation, the GNU C compiler, *gcc* (version 2.7.2), is used [Sta92].

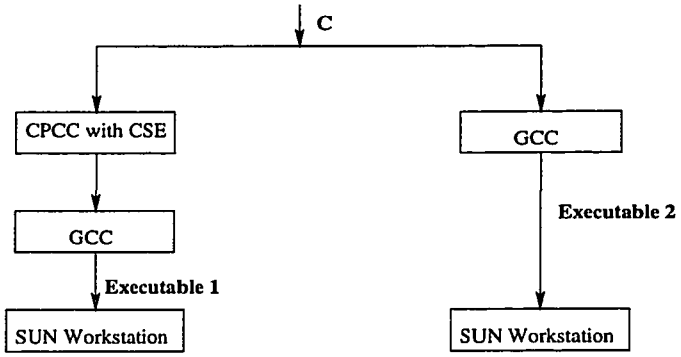


Figure 50: Experiment settings on a Sun workstation

Figure 50 shows the flow graph on compiling optimized or unoptimized code, running on a Sun workstation.

To run a program on a Sun workstation, we first let the program passing through our optimizing compiler, generating an optimized code with the CSE optimization applied; then we call *gcc* compiler to generate executable code. To see the effect of our CSE optimization, we turn off gcc optimizations by setting the optimization switch of the gcc compiler to $-O0$.

```
#include <time.h>
clock_t start, finish, duration;
...
start  = clock();
process();
finish = clock();
printf("The process took %f seconds to execute\n",
           ((double) (finish-start)) /CLOCKS_PER_SEC );
```

Figure 51: C routine measuring execution time

On Sun workstations, we use standard UNIX routines to measure time. The timing is measured by the method showed in Figure 51 and the unit is *second* [HJ95].

Figure 52 shows the flow graph on compiling optimized or unoptimized code, running the CPSS simulator.

To run an optimized program on CPSS, we first call our compiler to emit optimized code. The CSE optimization is applied during this pass. Then we call the CPC compiler to generate CPSS code. Finally we run the code on the CPSS simulator.

To run an unoptimized program on CPSS, we use the CPC compiler to emit CPSS code and run the code on the CPSS simulator hereafter.

On CPSS, the timing is reported by the simulator and we use the reported sequential execution time (cycles) as program total execution time.

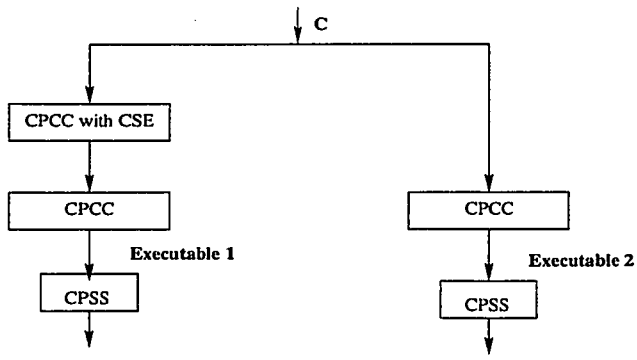


Figure 52: Experiment settings on the CPSS simulator

8.3 Static Profiling

Count of	quicksort	double-loop	single-loop	my-suite
Iterative Loops	2.0	2.0	2.5	2.0
CSE numbers	5.0	14.0	6.0	29.5

Table 1: Number of iterative solver iterations and the CSE expressions

Table 1 lists two numbers of counts, measured at compiler time. The first is the average number of iterations in which the iterative dataflow solver is in the state of

solving dataflow equations. The second is the average number of CSE expressions found. Both calculations are based per function base and the averages for a program is computed as a mean of all functions contained. The first metric indicates the overhead of the iterative solver. The smaller the number is, the less overhead of the solver. The second metric indicates the potential performance improvement. The bigger of the counts is, the more significant improvement we expect to see at runtime.

On average, the number of iterations for the iterative solver is about 2. This indicates that the overhead of the solver is very low. You may notice that the count of iterative loops of single-loop is 2.5. This is because we calculate the number of iterations for the iterative solver based on per function. Thus the numbers showed are the average value of iterations of all functions for a program. For example, the *single-loop* benchmark has two functions, one iterates twice and the other iterates three times, therefore the average iteration count is $(2 + 3)/2 = 2.5$.

The number of CSE transformations for each benchmarks ranges from two to a few tens. However, please note that this count is not an absolute indicator for performance improvement, since some common subexpression will never be executed such as those lie on the *false* conditional branch at runtime. For instances, the number of CSE transformations for the *single-loop* benchmark is 6 on average. However, this number is the sum of those CSEs appeared in both arms of a conditional statement, which is a constitute statement of the loop.

8.4 Runtime Measurement

In each run, we measure the total program execution time. For each version of a program, several runs have been launched and the average execution time is then computed as representative execution time for that version of the program.

We compare the execution times of two versions of programs.

8.4.1 Runtime Performance on Sun Workstations

Table 2 lists the percentage of improvement in optimized program execution time over un-optimized program, running on input size of 1000, 2000, 5000, 10000 and 20000. We use the relative improvement rate, r, to measure the execution-time improvement. Let T^{unopt} and T^{opt} be the total execution times for un-optimized and optimized

programs respectively. Then, r is defined as:

$$r = (T^{unopt} - T^{opt})/T^{unopt}$$

The bigger r is, the CSE optimization has more impact on execution time. Thus, r reflects the execution-time improvement.

Benchmarks	op/no-op	Input Size					Ave r (%)
		1000	2000	5000	10000	20000	
Quick Sort	op	0.51	2.07	12.82	51.39	206.42	
	no-op	0.53	2.08	12.94	51.72	207.06	
	r (%)	3.77	0.48	0.92	0.60	0.31	1.22
Double Loop	op	0.46	1.88	11.82	47.43	192.00	
	no-op	0.70	2.78	17.38	69.61	277.79	
	r (%)	34.8	32.3	31.9	31.8	30.8	32.30
Single Loop	op	0.00	0.00	0.00	0.01	0.01	
	no-op	0.00	0.00	0.01	0.01	0.02	
	r (%)	0.00	0.00	1.00	0.00	3.00	0.80
My-Suite	op	9.36	18.62	46.71	93.22	186.58	
	no-op	20.00	40.00	99.81	200.02	399.62	
	r (%)	53.00	53.40	53.20	53.30	53.20	53.26

Table 2: Execution times of two versions of programs, optimized vs. unoptimized on Sun workstations

From Table 2, we can see:

- For most benchmarks, the experimental results met with our expectation. On average, the optimized versions of *double-loop* and *my-suite* reduce the execution time by 32.30% and 53.26% respectively. The reason for such significant improvements is that many CSEs appear within the loop body and the impact of eliminating such common subexpressions can be seen in each loop iteration. Therefore, the accumulated effects can be seen clearly for the whole benchmarks.

- The influence of the CSE optimization on the quick-sort benchmark is not as great as we expected, even though there is a little positive impact. We will analyze this phenomenon in the next section.

- The effect of the CSE optimization can be seen only if the paths which have common subexpressions have been executed at runtime. Conditional statements

may divert real control flow to the paths which do not have CSEs. Therefore, the impact of the CSE optimization on conditional statements may not be as great as those based on loops only. That may explain why there is little impact of the CSE optimization on the single-loop benchmark. This also confirms our observation that the number of CSE transformations is only an indicator of potential performance improvement. It is by no means an absolute performance metric.

8.4.2 Performance Improvement on CPSS

Table 3 lists execution times of two versions of programs and related performance improvement running programs on the CPSS simulator. Since the benchmarks run on a simulator, the input sizes measured is smaller than those listed in table 2.

Benchmarks		Input Size					Average r
		50	100	200	400	500	
Quick-Sort	op	98545	347570	1295620	4991720	7739770	
	no-op	94485	331935	1234335	4749135	7361535	
	r (%)	-4.29	-4.71	-4.96	-5.11	-5.14	-4.84
Double Loop	op	80820	316620	1253220	4986420	7783020	
	no-op	116020	457020	1814020	7228020	11285020	
	r (%)	30.34	30.72	30.92	31.01	31.03	30.80
Single Loop	op	2491	4841	9341	18341	22841	
	no-op	3373	6623	12523	24323	30223	
	r (%)	26.15	26.91	25.41	24.59	24.43	25.50
My-suite	op	112388	429740	1679440	6638840	10348540	
	no-op	149988	574938	2249838	8899640	13874540	
	r (%)	25.07	25.25	25.40	25.40	25.41	25.30

Table 3: Execution time of two versions of program, optimized vs. unoptimized on the CPSS simulator.

Except the Quick-Sort benchmark, other benchmarks show the same trends as Table 2 shows. Furthermore due to lack of other compiler optimization influences such as *register allocation*, *instruction scheduling*, and *cache management*, the effects of the CSE optimization on CPSS is more evident than those showed in table 2. For example, on average the execution time of unoptimized Double-Loop is reduced by

about 31%. In other words, this shows that even though the CSE optimization can have great impact on performance, in a real compiler due to the influence of other optimization factors, the impact of the CSE optimization might not be as great as it is expected to be.

Because the Quick-Sort benchmark does not show as much positive improvement as we could expect, we will concentrate on analyzing why the CSE optimization has little or even negative improvement on Quick-Sort.

First, let's see what kinds of CSEs appear in the Quick-Sort benchmark.

```
1:
2: i   = i+1;
3: cse = sizeof(int)*i;
4: t2  = cse;
   ...
5: if( cond )
6:    goto 2 ;
   ...
7: t6 = cse;
```

Figure 53: CSE pattern in the Quick-Sort benchmark

Figure 53 lists the CSE pattern appeared in the benchmark. A common sub-expression is detected within a loop (line 2 - 6). However, within a loop, there is no second-time use of the value of the common sub-expression. The first time-saving access of the expression is outside the loop at line 7. Therefore such outside-loop CSE will not produce great performance improvement.

Besides, other compiler optimizations have to be considered in analyzing performance, such as the register allocation [Muc97]. Once a CSE transformation is applied, the lifetime of the variable holding the common subexpression is extended. For example, the lifetime of variable `cse` has extended from line 2 – 6 (loop) to line 2 – 7 (outside loop included). This may force the compiler generating spilled code if there is no enough register to hold the expression value. Then when the value is used later, CPU has to fetch it from memory. For modern RISC or superscalar architectures, memory latency is the bottleneck of system performance. Therefore, some CSE transformations may not be profitable if the register pressure (lacking of

registers) is big.

Then. why the extra assignment (line 4) does not cause the performance degradation in a Sun workstation as it does on the CPSS simulator?

On a modern RISC architecture, instructions are executed in pipeline and multi-issued, and data-bypass is widely used to pass data directly to the instructions appeared in the same pipeline [HP96]. Therefore, the extra assignment does not incur extra cost on Ultra Sparcs.

However on the CPSS simulator, the sequential timing is measured differently. Assignment `a = b` takes 3 cycles while `a = b * c` takes 5 cycles. Thus, the extra assignment introduced by the CSE optimization greatly slowdowns the program execution. This explains the negative impact we noticed when running Quick-Sort on CPSS.

Chapter 9

Conclusions and Future Work

This thesis concentrates on the development of a compiler optimization framework for CPC.

Based on the structural analysis technique, we have developed a two-pass algorithm that builds a CFG from a CPC Abstract Syntax Tree. Based on the CFG representation, a general iterative solver for solving dataflow equations has been built. This solver forms the core of compiler optimization framework, and it can solve global data-flow equations in either *forward* and *backward* manner. As a demonstration example, we have showed how to solve the common subexpression elimination problem using the optimization framework.

We have applied the CSE optimization on a suite of four benchmarks. Static and runtime information about program performance have been collected, by running program on both workstations and the CPSS simulator. The experimental results show that:

- The iterative solver is very efficient and the iteration number for solving dataflow equations is very low.
- The CSE optimization has a significant impact for some benchmarks.
- Other compiler optimization factors such as register allocation might influence the effectiveness of the CSE optimization.

We also identify some limitations of the current optimization framework. For example:

- Programmers need to expose CSEs explicitly such as in the array index case. For finding this type of CSE, another lower-level representation seems necessary.
- Array and pointer variables are pervasive in C programs. More advanced analysis techniques need to be designed and implemented to support optimizations on array and pointer variables [Ban79, CK89, BCCH94, HHN94].
- The current implementation of the framework is by no means perfect. More efficient implementations need to be done, such as using bit-vectors to implement set operations.

We will continue research along these directions and our goal is to build an efficient yet general compiler optimization framework, supporting the Concordia Parallel Programming Environment (CPPE).

Bibliography

[AALL93] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Amy W. Lim. An overview of a compiler for scalable parallel machines. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 253–272, Portland, Oregon, August 12–14, 1993. Intel Corp. and the Portland Group, Inc., Springer-Verlag. Published in 1994.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Ban79] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 29–41, San Antonio, Texas, January 29–31, 1979. ACM SIGACT and SIGPLAN.

[Bar78] Jeffrey M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

[BCCH94] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, number 892 in Lecture

Notes in Computer Science, pages 234–250, Ithaca, New York, August 8–10, 1994. Springer-Verlag. Published in 1995.

[C.A62] C.A.R.Hoare. Quicksort. *Computer Journal*, 5(1), 1962.

[CK89] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59, Austin, Texas, January 11–13, 1989. ACM SIGACT and SIGPLAN.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press; McGraw-Hill Book Company, Cambridge, Massachusetts; New York, New York, 1990.

[CNO+87] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David P. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, Palo Alto, California, October 5–8, 1987. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 15(5), October 1987; *SIGPLAN Notices,* 22(10), October 1987.

[DDG+96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *OOPSLA '96*, 1996. http://www.cs.washington.edu/research/projects/cecil/www/www/Papers/.

[Fis83] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, Stockholm, Sweden, June 13–17, 1983. *Computer Architecture News,* 11(3), June 1983.

[FKU75] Amelia Fong, John Kam, and Jeffrey Ullman. Application of lattice algebra to loop optimization. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 1–9, Palo Alto, California, January 20–22, 1975. ACM SIGACT and SIGPLAN.

[GGMW82] Harald Ganzinger, Robert Giegerich, Ulrich Möncke, and Reinhard Wilhelm. A truly generative semantics-directed compiler generator. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 172–184, Boston, Massachusetts, June 23–25, 1982. ACM SIGPLAN. *SIGPLAN Notices,* 17(6), June 1982.

[GM86] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, Palo Alto, California, June 25–27, 1986. ACM SIGPLAN. *SIGPLAN Notices,* 21(7), July 1986.

[HDE+92] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 406–420, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag. Published in 1993.

[HHN94] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 208–216, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

[HJ95] Samuel P. Harbison and Guy L. Steele Jr. *C a reference Manual.* Prentice-Hall, Inc., 1995.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 2nd edition, 1996.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Publishing Company, 1979.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of ACM Symposium on Principles of Programming*

Languages, pages 194–206, Boston, Massachusetts, October 1–3, 1973. ACM SIGACT and SIGPLAN.

[Kog81] Peter M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill Book Company, New York, New York, 1981.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 2nd edition, 1988.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[Lid98] Rudolf Lidl. *Applied abstract algebra.* Springer-Verlag, 1998.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, California, June 17–19, 1992. *SIGPLAN Notices,* 27(7), July 1992.

[MH86] Scott McFarling and John Hennessy. Reducing the cost of branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403, Tokyo, Japan, June 2–5, 1986. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 14(2), June 1986.

[Muc97] Steven S. Muchnick. *Advanced Compiler design and Implementation.* Morgan Kaufmann Publishers, Inc., 1997.

[Ros77] Barry K. Rosen. High-level data flow analysis. *Communications of the ACM*, 20(10):712–724, October 1977.

[Ros79] Barry K. Rosen. Data flow analysis for procedural languages. *Journal of the ACM*, 26(2):322–344, April 1979.

[Sta92] Richard M. Stallman. *Using and Porting GNU CC.* Cambridge, Massachusetts, June 1992. Available via anonymous ftp from `prep.ai.mit.edu`.

[Tao96] Lixin Tao. Parallel programming environment prototype cppe, 1996. http://www.cs.concordia.ca/ faculty/lixin/spark/.

[TH92] Steven W. K. Tjiang and John L. Hennessy. Sharlit—a tool for building optimizers. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 82–93, San Francisco, California, June 17–19, 1992. *SIGPLAN Notices,* 27(7), July 1992.

[WS91] Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 120–129, Toronto, Ontario, June 26–28, 1991. *SIGPLAN Notices,* 26(6), June 1991.

[WZ85] Mark N. Wegman and Frank Kenneth Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, New Orleans, Louisiana, January 13–16, 1985. ACM SIGACT and SIGPLAN.

Appendix A

A.1 Benchmark of Single Loop

```
void test_for(int count)
{ int a1, b1, c1,d1,e1;
  int a2, b2, c2,d2,e2;
  int a3, b3, c3,d3,e3;
  int x, y, z , u, v;
  int i,j; int m,n,o,p,q;
  int t1, t2, t3, t4, t5, t6, t7, t8, t9,t10;

  m = 10; n =15; o =20; p = 1; q = 2;
  u = n*o; v = o*p; x = m*n;
  y = n*p; z = y*z;
  t1 = m*o;   t2 = m*o*q ; t3 = m*o*q;
  t4 = n*p;   t5 = 200;  t6 = m*o*q;
  t7 = n*p*o; t8 = 400; t9 = o*p;
  t10 = 50;

  a1 = t1*t2; b1 = t3*t4; c1 = t5*t6;
  d1 = t7*t8; e1 = t9*t10;

  for (i=0; i<count; i++) {
    if (i < 100) {
      a3 = t1*t2; b3 = t3*t4; c3 = t5*t6;
      o = x*y; p = y*z;
    }
```

```
    else {
      d3 = t7*t8; e3 = t9*t10;
      q = u*v; m = q*o;
    }
    a2 = t1*t2; b2 = t3*t4; c2 = t5*t6;
    d2 = t7*t8; e2 = t9*t10;
  }
}

int main()
{
  test_for(100);
}
```

A.2 Benchmark of Double Loops

```
int main()
{ int a, b, c,d,e, f;
  int i, x, y, z , u, v, w;
  int j,m,n;
  int t1, t2;
  int count;

  for (count =0; count <10 ;count++) {
    x = 10; y = 20;
    u = x*y*10; v = x*y;
    t1 = u-2; t2 = v+2;

    for (x=0; x<count; x++) {
      a = u*v;   b = u*v;   c = u*v; v = u*v;
      d = t1*t2; e = t1*t2; z = t1*t2;
    }
  }
}
```

A.3 Benchmark of My-suite

```
int main()
{ int a, b, c,d,e, f;
  int i, x, y, z , u, v, w;
  int j,m,n;
  int t1, t2, t3;
  int count;

 for (count =0; count <10 ;count++)      /* test for statement */
 {
  x = 1; y = 1; u = 2; v = 2;
  t1 = 200; t2 = t1*u; t3 = t1*t2;

  for (x=0; x<count; x++){
    c = u*v;
    d = t1*t2*t3;
    e = t1*t2;
    z = t2*t3;
    if (y > 5)
       continue;                        /* test continue statement */
  }

  y = count; x = 1; f = u*v;
  d = t1*t2; e = t1*t2; z = t1*t2*t3;
  c = t2*t3;
  do{                                   /* test do-while statement */
    c = u*v; f = t1*t2;
    y--;} while ( y >0 );

  x += 1;
  if (x) {                              /* test if-then-else statement */
    c = u*v; d = t1*t2; f = u*v;
    z = t1*t2; e = t1*t2; x = y*z;
  }
  else {
```

```
    a = t1*t2; c = u*v; x = y*z;
    c = u*v; d = t1*t2; f = u*v;
    z = t1*t2; e = t1*t2;
  }
  w = y*z; f = t1*t2;

  x = 3;
  switch ( x ){                          /* test switch statement */
   case 1 :  x = 2;
             c = u*v; f = u*v; d = t1*t2;
             z = t1*t2; e = t1*t2;
             break;
   case 2 :  x = 3;
             f = t1*t2; d = t1*t2; c = u*v;
             z = t1*t2; e = t2*t3;
             break;
   case 3 :  x = 4;
             c = u*v; f = u*v; d = t1*t2*t3;
             z = t1*t2; e = t1*t2*t3;
             break;
  }

  i = count;
  a = x+1; b = y+1; c = u*v;
  d = t1*t2; z = t1*t2; e = t1*t2;
  while (i>0) {                           /* test while statement */
    c = x+1; d = y+1;
    z = t1*t2; e = u*v; f = t1*t2;
    i = i-1;
  }
  w = y+1; z = x+1;

  a = 1;
  goto B;                                /* test go-to statement */
  b = 4;
B:b = 8;
```

```
  z = t1*t2; e = u*v; f = t1*t2;
 }
}
```

A.4 Benchmark of Quick Sort

```
static void byte2int(char *v_int, char *s_arry)
{ int i;
   for(i=0; i< sizeof(int); i++)
     v_int[i] = s_arry[i];
}

int partition(char *A, int m, int n)
{
  int x, my_temp, i, j;
  int t1,t2,t3,t4,t5,t6,t7,t8,t9,t10;

  i = m-1;
  j = n+1;
  t1 = sizeof(int)*m;
  byte2int((char *) &x, &A[t1]);          /* x = A[t1] */
C: i = i+1;
  t2 = sizeof(int)*i;
  byte2int((char *) &t3, &A[t2]);         /* t3 = A[t2] */
  if( t3<x ) goto C ;
D: j = j-1;
  t4 = sizeof(int)*j;
  byte2int((char *) &t5, &A[t4]);         /* t5 = A[t4]; */
  if(t5>x ) goto D ;
  if (i >= j) goto E ;
  t6 = sizeof(int)*i;
  byte2int((char *) &my_temp, &A[t6]);    /* x = A[t6]; */
  t7 = sizeof(int)*i;
  t8 = sizeof(int)*j;
  byte2int((char *) &t9, &A[t8]);         /* t9 = A[t8]; */
  byte2int(&A[t7], (char *)&t9);          /* A[t7] = t9 */
```

```
  t10 = sizeof(int)*j;
  byte2int( &A[t10], (char *)&my_temp);  /* A[t10] = x; */
  goto C;
E: return (j);
}

void quicksort(int  *A, int low, int high)
{
  if (low < high)
    { int q;
      char *B;

      B = (char *)A;
      q = partition(B, low, high);
      quicksort(A, low, q);
      quicksort(A, q + 1, high);
    }
}

int main()
{
  int foo[100],  n;
  int i;

  n = 100;
  for (i = 0; i < n; i++) {
    foo[i] = n-i ;
  }
  quicksort(foo, 0, n-1);
  return 0;
}
```