A Portable Machine-Independent Global Optimizer —

Design and Measurements

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

by

Frederick C. Chow

December 1983

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

(Medicine & Computer Science)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

Abstract

This dissertation addresses the topic of portable and machine-independent program optimization on a standard, well-defined intermediate code. The feasibility, advantages and problems of this approach of implementing an optimizer are discussed. We also look into issues on the design of the intermediate code, and the features in the intermediate code needed to support machine-independent optimization.

A number of new techniques in program optimization are developed. A concise and more generalized method for performing copy propagation, and a new method to perform redundant store elimination are introduced. The partial redundancy algorithm is formulated and generalized to strength reduction, thus enabling common subexpression elimination, code motion and strength reduction to be performed at the same time. The concept of partial redundancy in stores is derived from partial redundancy in expressions and applied in performing forward code motion. Using these techniques, it is possible to integrate previously separate transformations into common processes and have them performed together. As a result, it is possible to do all common global optimizations in a small number of passes. This approach can also substantially reduce the implementation complexities and running time of optimizers in general, with no sacrifice in the optimizations performed.

A register allocation algorithm based on the coloring algorithm and suitable for use in the machine-independent context is introduced. The algorithm performs well independent of the number of registers available. A parameterization of register allocation cost and saving enables us to cater to the characteristics of different machines.

An implementation of the above optimization techniques in the machine-independent optimizer UOPT is presented. We look into the interactions between the different types of optimizations, and how the phase structures can be organized to take these interactions into account. The optimization performance, efficiency and the relative importance among the different types of optimization transformations are studied according to timing measurements, optimization statistics and by variation in optimization parameters.

Finally, the effectiveness of portable machine-independent optimization on a number of target machines that support the intermediate code is discussed, based on optimization performance data in the different machines and comparisons of machine characteristics. Intuitive ways to predict the effectiveness of some types of optimizations with respect to specific architectural features are furnished. The overall evaluation confirms the advantages of using portable, machine-independent optimization in a retargetable compiler system.

This thesis was submitted to the Department of Electrical Engineering and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This dissertation represents part of the programming language and compiler development work at Stanford University for the S-1 computer, under Contract No. 2213801 from the Lawrence Livermore National Laboratory. The development of the S-1 computer is funded by the Office of Naval Research of the U. S. Navy and the Department of Energy.

To my parents

Preface

The subject of program optimization has been dealt with in many text-books on compiler construction as one aspect of the compilation process. It has seldom been treated in an isolated manner, separate from the influences of other parts of the compiler and as a coherent, self-contained piece of software. The development of the UOPT optimizer has provided the opportunity to address optimization in such a setting. This thesis focuses on the subject of machine-independent optimization in depth. Following a brief look into the design issues of the intermediate code, a complete range of optimization techniques are covered in detail, from algorithms to practical aspects of implementation. The integration of the various kinds of optimizations into a practical production optimizer is also addressed. The optimization topics covered are concluded with a performance evaluation of the actual optimization results.

I first started on this work about three and a half years ago, when Gio Wiederhold and John Hennessy first suggested to me the possibility of building a local optimizer on U-Code. Later on, John Hennessy continued to guide me along in developing and implementing the global optimizer UOPT. Before this, I have never thought that the complete task of implementing a global optimizer can be handled by a single person. UOPT has set a precedent by showing that this is indeed possible.

I am very indebted to John Hennessy, for his excellent and continuous guidence; and to Gio Wiederhold and Forest Baskett, for the advice they have given me on numerous occasions. A number of people have affected the outcome of this work, and I have benefited from interacting with them. I wish to thank Peter Nye, who co-ordinated and standardized the software and documentation of the U-Code environment at Stanford, and also implemented the DEC 10 code generator; David Schnepper, who wrote the procedure integrator Pmerge; Per Bothner, who brought up the 68000 code generator; Gregory Boyd and Steve Tjiang, who did the VAX code generator; Chris Rowen, who implemented the MIPS code generator; Mahadevan Ganapathi and Vivek Sarkar, who built the FOM code generator; and Wes Witte, who implemented the S-1 code generator. I appreciate the companionships of Kyu-Young Whang and Edwin Pednault, who shared my office during these years. This research has been supported by the S-1 project, and my thanks also extend to all members of the S-1 project staff at Stanford and the Lawrence Livermore Laboratory.

Above all, I am grateful to my father and mother, for the care and support they have given me all through the years. I dedicate this thesis to them.

Contents

1. Introduction

Lowering software cost has been one of the main concerns among computer professionals ever since the use of computers. In the software world, compilers have been among the most important and prevalent pieces of software. In the last decade, newly emerging machine architectures, coupled with the need to support the growing number of programming languages, have made it increasing important to systematize and automate the construction of compilers for the purpose of shortening new compiler development time and reducing the cost of construction and maintenance. The conventional approach to compiler construction has been to build a separate compiler for each programming language and machine combination. This results in language and machine dependencies being spread throughout the compilers. Algorithms and code structures that are common to the compilers are duplicated in each implementation. For a given programming language, the individually-developed compilers often create incompatibilities across different machines. For a given machine architecture, the different programming languages supported may not be able to reflect uniform hardware characteristics due to the completely separate compiler implementations.

Much of the work on portable compilers has involved the use of intermediate code. Using a standard intermediate representation for a programming language enhances the portability of the language. This also makes possible the division of compilers into front-ends for lexical and syntactical analysis and back-ends for code generation. An intermediate language can be made to act as the common interface between the language-dependent front-ends and machine-dependent back-ends of a compiling system. By using a single intermediate form, $(p \times m)$ compilers can be replaced by p front-ends and m back-ends. This also helps ensure the machine-independence of the source languages and language-independence of the code generators [Stee61].

With the use of intermediate code, compiler automation can be applied to the front-ends and back-ends separately. Research into automating the process of parsing program text into intermediate representations has resulted in the successful construction of parser generators that are now in common use. Using these translator writing systems, it is sufficient to specify the grammar of the source language. The syntax analyzers construct tables from syntax descriptions and use the tables to drive program analysis. Recently, attention has been turned towards automating the code-generating back-ends, and retargetable code generation has become an increasingly important area [Grah80] [Gana82]. Using modular approaches to code synthesis, these code portable generators are parameterized with respect to machine descriptions. By giving them different sets of machine parameters, the code generators can be adapted and retargeted to produce code for different new machines.

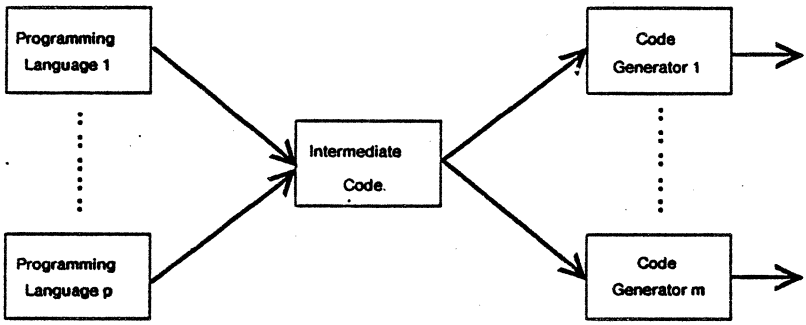


Fig. 1.1 Use of intermediate code in a compiling system

When a common intermediate form is used, there exists the opportunity to construct program optimizers that use the intermediate form as their input and output languages. Optimizations can be divided into machine-independent optimizations and machine-dependent optimizations. Since machine-dependent optimizations take full account of the instructions and hardware features of the underlying machines, they are usually performed by code generators, and the transformations are mostly confined to local regions of the program code. Machine-independent optimizations, if performed independent of the code generators, can be made available for all target machines. By doing machine-independent optimizations on the intermediate code, the optimizations can be made independent of the source languages as well, at least to the extent that the intermediate code is language-independent. Intermediate forms of program code have often been used for optimization purposes in classical monolithic compilers [Aho77]. In spite of this, conventional optimizers depend a great deal on other parts of the compilers and are not capable of independent existence, even if the optimizations performed are independent of the target machines.

In this thesis, a self-contained global optimizer on a machine-independent intermediate language is presented. Compiler front-ends translate source programming languages to this intermediate language, called U-Code. The optimizer inputs the intermediate program code, performs machine-independent optimizations and outputs an optimized version of the program in the same intermediate language. The code-generating back ends will translate the intermediate code to target machine code. The result is a portable compiler module performing machine-independent optimizations. By existing independently of any front-end and back-end, its applicability across multiple machines and source languages is guaranteed. Apart from widening its usages, this approach also eliminates the needs of the front-ends and back-ends to attempt optimizations that have been performed by the optimizer, enabling them to specialize

and concentrate on their forms of processing. This contributes to modularity and clarity of interface among the various components of the compiler system. Fundamentally, this approach also makes code optimization an easily affordable and available facility in program translation environments that use the same intermediate code.

1.1. Related Work

The importance of code optimization has been recognized since the days of the first Fortran compilers. The loss of object code efficiency has been inherent to high-level language programming. Most programming language compilers do some forms of code optimization, although the extents to which they perform optimization differ widely. They usually incorporate their own sets of well-defined, limited transformations to improve running times for most executions. The term *optimizing compilers* refers to compilers that perform more substantial code optimization in the compilation process. In recent years, as the use of compiler-compilers gradually becomes entrenched and retargetable code generation begins to gain wide acceptance, the need to apply the same idea of retargetability in the construction of optimizers is recognized. In this section, we survey optimization-related works which display the built-in capability of being transportable and machine-independent.

The Production-Quality Compiler-Compiler (PQCC) Project [Leve79] at Carnagie-Mellon University has as its goal the building of a truly automatic compiler-writing system. PQCC extends compiler-compiler techniques in parser generation to include the production of optimizers and code generators. The system operates from descriptions of both the source languages and the target computers. Tables are generated from the language and machine descriptions and used to guide the operation of the skeleton compiler. Both machine-independent and machine-dependent optimizations are performed. In the case of machine-dependent optimizations, attempts were made to parameterize optimization techniques so that they can be moved from one target machine to another by changing only the set of tables describing the machines. The compiler is divided into a number of phases which operate serially. This allows the decomposition of the PQCC into manageable portions. The different phases do not rely on each other for their operations, and can run in stand-alone modes. A uniform intermediate representation is used as input and output for the machine-independent phases. Machine-independent optimizations performed are code motion and elimination of redundant computations and various local optimizations. Register allocation, code selection, peephole optimizations and other optimizations requiring detailed knowledge of the instructions are performed on a linear form of code that retains only minimal target machine independence in the final phases.

The PL.8 compiler project of IBM [Ausl82] accepts multiple source languages and produces

high quality object code for several different machines. It divides the compilation process into translation, optimization, register allocation and final assembly. Optimization is further partitioned into as many independent operations as possible to make them reliable and easy to implement. Each optimization is repetitively performed because one may provide new opportunities for another. A low level intermediate language is used whose semantics matches the computational semantics of the limited set of target machines, and whose level is low enough to expose all instructions that will be executed on the target machines. Global optimization and register allocation are performed on this code, and further optimization on the machine-code level for individual machines is unnecessary. The intermediate language is partly machine-dependent, and is at a lower level than some of the target CPU's. The compilation and optimization methods are biased towards machines with regular and simple register-register architectures.

The Experimental Compiling System (ECS), also undertaken at IBM [Alle80], uses a new compiler construction methodology [Harr76] in which compilers for a variety of source languages and target machines can be developed. Language semantics is specified by writing defining procedures which take the place of code generators and code macros. Programs together with the defining procedures are expressed by a single program schema, called IL, which can represent programs at different levels of semantics in the compilation and optimization processes. As a result, an optimizer can be constructed which deals with several levels of expansion of a program. High-level code is expanded to lower-level code via procedure integration, and analysis and optimization are then used to tailor code to its particular context. The system permits varying degrees of optimization by repeated application of procedure integration and an extensive collection of machine-independent optimizations. A primitive language version of the IL is produced which reflects the operations of the target machine. A final machine tailoring phase generates the target machine code.

The Universal Compiling System (UCS) [Gyll79] at Sperry Corp. is a unified compiling system for a set of languages and architectures. Through the use of an intermediate text and symbol table, source language dependent processes are separated from architecture dependent processes. A common global optimizer is used between the front-ends and back-ends.

The MUG2 compiler generating system at the Technical University of Munich [Wilh81] is an effort to produce optimizing compilers from language and machine descriptions. Description tools and generators for multi-pass semantic analysis, code optimization and code generation are offered. The description tools can completely describe optimization passes like global data analysis, constant propagation and folding and invariant code motion from while loops.

The Amsterdam Compiler Kit (ACK) [Tane83] is a compiler-building system that consists of a number of parts that can be combined to form compilers with various properties. The tool kit

consists of eight components: the preprocessor, separate front-ends, the peephole optimizer, the global optimizer, the back-end, the target machine optimizer, the universal assembler/linker and the utility package. The front-ends output an intermediate code, which is the machine language for a simple stack machine called EM. The peephole optimizer and the global optimizer perform machine-independent optimizations on this intermediate code. The peephole optimizer [Tane82] is driven by a pattern/replacement table that specifies how specific patterns of instruction sequences within a window can be replaced by more efficient ones. This optimization process involves only pattern matching and substitutions. The global optimizer examines the program as a whole and performs more extensive transformations. The back-end target machine optimizer and universal assembler/linker are driven by machine-dependent driving tables, which tell how the EM code is mapped onto the target machine's assembly language. The target machine optimizer performs optimizations involving idiosyncracies of the target machine that cannot be included in the EM-to-EM optimizers.

The target-independent optimizers described above have been developed as built-in components of large, comprehensive compiler-generating systems, and they can only operate in their specific program translation environments. There are other target-independent optimizers which exist in more distinct fashions from the front-ends and back-ends and whose modes of operation are more independent. The types of optimizations they perform are more limited in scope.

In [Frai79], a source- and target-independent code optimizer is described which uses an intermediate language in the form of N-tuples. The optimizer performs only local expression optimization and common subexpression elimination. The principal role of the optimizer is in gathering information about operand usages in a target-independent manner which enables the target-dependent code generator to fold constants, avoid redundant loads and stores, and perform more efficient register allocation.

A retargetable peephole optimizer, PO, is presented in [Davi80] which performs peephole optimization on object code. Given an assembly language program and a symbolic machine description, PO simulates pairs of adjacent instructions and, wherever possible, replaces them with an equivalent single instruction. It can be easily retargeted by changing machine descriptions. It can serve to supplement machine-dependent optimizations performed by the code generators which can be locally optimal but may be suboptimal when juxtaposed. This results in a further division of labor in the code generation phase which can simplify the code generator.

OPTIMA [Wilk83] is another portable optimizer on an intermediate code — the Pascal PCODE. It outputs QCODE, which is a portable code for a machine that retains the stack configuration but is generalized to exhibit memory areas and a parameterized number of general-purpose and floating-point registers. OPTIMA performs only local optimizations. The first

stage transforms PCODE by the peephole optimization method which is also table-driven. The output is saved in a doubly-linked list of tuples which represents the PCODE in triple forms and includes other information needed for later optimization and code generation. The second stage operates on the tuples generated to perform optimizations in array element offset computation and eliminate locally redundant operations. The third stage performs register allocation and generates the output QCODE. The second and third stages use machine descriptions in their processing. The output QCODE is translated into assembly code of target machines by macro expansions.

The portable C compiler [John78] also contains a limited number of machine-independent optimizations and some register-related optimizations that have to be adapted when porting to new machines.

The UCSD Machine-independent Pascal Code Optimization project [Site79] set out to build an optimizer that performs optimization on standard Pascal P-Code. In the process, they defined the Universal P-Code (U-Code) which is designed specifically to include enough information for optimization purposes. Though ideas were presented for implementing the optimizer, the implementation was never completed, but portions of the results do demonstrate the feasibility and practicality of optimization on U-Code. The intermediate language used by the global optimizer presented in this thesis is based on the U-Code as originally defined by the UCSD group.

1.2. Background of This Work

This thesis research was undertaken as part of the Stanford U-Code Compiling System. This system was originated as the software project to develop programming language support for the Stanford-1 (S-1) multiprocessor architecture being developed at the Lawrence Livermore Laboratory [Hail79] [Livi83]. The project involves the support of standard Pascal and the writing of a Fortran compiler that implements the Fortran66 Standard [Chow80]. In the process, the Pascal P-Code was adopted as the intermediate code common to both Pascal and Fortran, and a common code generator was written that translates P-Code to S-1 machine code.

Later, the UCSD Machine-independent Pascal Code Optimization project was undertaken. The S-1 was then intended as one of the beneficiaries of the optimizer that was to be built. As the UCSD group went on to define the U-Code language to be used as the medium of their optimizer, the Stanford S-1 project began to adopt U-Code as the intermediate code. The UCSD optimization project was not able to reach completion [Site79b]. As a result, the author of this thesis undertook the independent project to build an intermediate code global optimizer

at Stanford. The content of this thesis, together with the production optimizer UOPT, represent the bulk of this work.

In the meantime, the U-Code Compiling System at Stanford began to enlarge in scope. The Pascal front-end was extended to Pascal* which expands the features supported and enlarges its capability [Henn82b]. The Fortran Compiler was extended to support Fortran77. A front-end that translates a subset of C to U-Code was also implemented. A procedure integrator for U-Code was implemented separately; when invoked as a pre-pass for UOPT, the procedure integrator can allow the intra-procedural optimizations of UOPT to extend beyond the procedure boundaries of the original programs. The Stanford Retargetable Code-Generation Project was started. The goal of this project is to build a code generator using a code generation skeleton and scheme such that the code generator can be ported to a different machine by just rewriting a small portion of the code. To take advantage of retargetable code generation, the S-1 code generator was rewritten using the retargeting methodology. Code generators for the DEC 10 and VAX, the host computers where most of the compilers were constructed, have also been written for testing and demonstration purposes, and will eventually be adopted as the resident compilers. As part of the Stanford University Network (SUN) project, a MC68000 code generator was also written for the SUN Work Station. A code generator is being developed for the MIPS Microprocessor Project at Stanford [Henn82c] [Henn83]. A code generator for the Fortran Optimized Machine (FOM), an experimental architectural project at IBM [Bran82], is also being undertaken at Stanford [Gana80]. An accompanying product of this latter project is a code generator for the IBM 370. The MIPS, FOM and 370 code generators are not related to the retargetable code generator project, although they use U-Code as the input intermediate code.

The U-Code compiling system at Stanford [Nye83] is a portable and retargetable compiler project which has goals similar to those of the various projects surveyed in Section 1.1. What distinguishes this project from others is that the U-Code intermediate language together with its related software facilities are the only connecting links among the various components of the system. We do not attempt a large system that is so integrated that the various components could not work independently when taken out of the system, and so extensive that the whole system is hard to install, maintain and modify. Instead, the different components of the system are separately implemented, the only requirement being that they conform to the U-Code standard. The separation also means that modules of the system can be optionally run on any given compilation. Since U-Code is a well-defined and popular intermediate language, it is only necessary for a new installation to use the same U-Code in order to be able to make use of the different software provided in the compiling system. Thus, the restrictions imposed by

the different components of the compiling system are minimal. New front-ends, back-ends or middle-ends can be freely and independently implemented whenever the needs arise. The whole system is simple and modular. We think that this approach can result in a greater degree of acceptance of our software by outside sources, and may also lead to eventual popularization and standardization of a single intermediate code in program compilation.

1.3. Objectives and Contributions

This dissertation deals with the design of a machine-independent optimizer. While the optimization output of the optimizer is machine-independent, the optimizer is also portable in that it is operational under a wide range of dissimilar compilation and operating system environments. The portability attribute dictates that the optimizer must be able to operate in a stand-alone mode, independent of the front-ends and back-ends. Moreover, this self-contained characteristic makes it unnecessary to recode the analysis and optimization parts of the optimizer several times for the purpose of exhaustive optimizations. The optimization pass can be re-run as many times as desired.

A key to the portability of the optimizer is the fact that it performs optimization on an intermediate language and outputs the optimized code in the same intermediate language. The presence of the optimizer as a middle pass in the compilation sequence should not have substantial impact on the front-ends for them to specifically accommodate its presence. The code generating back-ends should have to do little, if any, to initially take advantage of all the optimizations done by the optimizer. Apart from contributing to clean interfaces, this also serves to ensure that the performances of the front-ends and back-ends will not suffer if the user selects not to use the optimizer in his compilation. In practice, few code generators are perfect in being able to handle all kinds of input intermediate code sequences well, and nearly all code generators have some built-in expectations of the kinds of code sequences they see most often. After the optimizer has been accepted as the middle pass, the code generators can be gradually made to utilize specific optimized code constructs to their full advantages.

Since the optimization medium is an intermediate code, emphasis is not placed on machine-dependent optimizations, which are better done in the code generation phases. On the other hand, a main goal in this thesis is to include as many useful machine-independent optimizations as possible in the portable optimizer. These include all common local and global optimization transformations. Register allocation, which is slightly machine-dependent, is included since this can take advantage of the global flow analysis performed in the optimizer. All these optimizations are integrated together so that they can take advantage of each others' results.

1.3. Objectives and Contributions

Optimization techniques have developed and appeared in the literature for more than a decade. The most common optimizations consist of different transformations that bear little relationship to each other. In conventional program optimizers, these transformations are implemented and performed separately, often by case analysis of the program text. This conventional approach, though easily comprehensible, creates great program complexities in the implementation effort due to the different nature of the various optimizations and the large number of special cases to be taken care of under each category. The whole optimization process is often broken down into a number of separate passes and filters in order to make the optimization effort manageable, but this usually seriously degrades the optimization speed.

The global optimization approach presented in this thesis represents a departure from conventional global optimizer designs, and is another contribution of this thesis. Central to our global optimization framework is the use of the partial redundancy elimination algorithm as the underlying theme. The goal is to shift as much processing as possible to the data flow analysis phases. Apart from simplifying the individual program transformation processes, our approach also makes possible the identification of previously separate global optimizations as being special cases of some common processes. As a result, the optimizer is able to do all common global optimizations in a small number of passes. This approach leads to a reduction in program complexities and implementation efforts compared with conventional techniques. The result is a closely-knit, concisely implemented global optimizer† that is also fast compared with conventional optimizer doing the same optimizations. These optimization techniques are applicable to global optimizers in general. By implementing these new techniques, the machine-independent optimizer provides a working model that can be followed by other optimizers.

Register allocation is another area where a new approach is tried in this thesis. We have designed a register allocation scheme for use in the machine-independent context. We introduced a parameterization of the cost and saving in register allocation that can cater to the characteristics of different machines. No constraint is imposed on the front-ends. The register allocation algorithm is a combination of a local method based on usage counts and the global method that uses priority-based coloring. The relative importance of the two can be varied. The algorithm is efficient and yields reasonable solutions with most target machine register configurations.

As a component in a retargetable compiling system, the optimizer provides the opportunity to study the effects of the same optimizations on different machines. The optimizations in UOPT are performed without specific target machines in mind. It is expected that the percentage improvements in execution speeds of the same optimized programs will differ among machines.

---

† UOPT is written in 13000 lines of Pascal code.

We offer interpretations for some of the differences in performance based on evaluations of machine characteristics, and we also provide some intuitive ways to predict the effectiveness of some types of optimizations with respect to specific architectural features.

Apart from these, the machine-independent optimizer also plays a role in supporting architectural experimentation. Using the data on optimization performances on different machines, it is possible to determine the machine characteristics that can best benefit from the optimizations performed. Efforts can then be made to design machine architectures which will exhibit superior performance in a compilation environment that provides intermediate code optimization, much as architectures have been developed with particular programming languages or code generation techniques in mind. Such investigations can have an impact on the evolution of future machine architectures.

1.4. Optimizations Performed

The global optimizer presented in this thesis, UOPT, performs most standard local and global optimizations. It operates on a procedure by procedure basis, and performs all bit-vector data flow analyses short of inter-procedural analysis. A separate procedure merger can be used as a pre-pass to perform procedure integration.

Apart from dead code elimination, there is not any optimization that changes the control flow structure of the program. The fact that the control flow graph does not change during optimization simplifies the internal structure of the optimizer. Apart from the computation of loop-nesting depths for register allocation, none of the optimizations performed requires detailed control flow analysis. The following is a list of the optimizations included in UOPT:

1. Stack height reduction in expression evaluation.
2. Constant propagation.
3. Constant expression evaluation.
4. Address collapsing in array expressions.
5. Dead code elimination.
6. Copy propagation.
7. Common subexpression elimination.
8. Loop-invariant expression optimization.
9. Partial redundancy suppression by backward code motion.
10. Loop induction expression optimization (strength reduction).
11. Linear function test replacement and induction variable elimination.
12. Redundant store elimination.
13. Dead variable elimination.

14. Partial redundancy suppression by forward code motion.
15. Optimization of positions to save computations in temporaries.
16. Global register allocation and assignments.

Program optimization aims at improving the execution speed and reducing the code space and storage requirements. In some transformations, conflict exists between these two objectives in that one can be fulfilled only at the expense of the other. The main objective in UOPT is to optimize running time. In some cases, code sections are duplicated and re-introduced with the effect of increasing code speed while sacrificing code space. These occur especially in partial redundancy suppression and some loop induction expression optimizations. Register allocation actually introduces extra register transfer instructions that would not otherwise be present in the program. Some of these new code may not be reflected in the final object code after the code generation phase. We have not included any transformation that optimizes only space. The most important code space optimizations can be efficiently done in the code-generating back-ends on the machine instruction level, since the optimizations are mostly local in nature.

1.5. Organization of This Thesis

The remainder of this thesis is divided into six chapters.

Chapter 2 examines issues in the design of intermediate languages from the point of view of supporting and expressing machine-independent optimizations. Important features of the intermediate language U-Code, the medium of optimization in this thesis, are also presented.

Chapter 3 covers the optimization methods. Some new optimization algorithms are formulated. The theories and motivations behind them are presented, together with explanations as to how they represent improvements over traditional optimization techniques.

Chapter 4 discusses the feasibility and limitations of performing register allocation and assignments at the intermediate code level. The coloring algorithm is modified and adapted for use in the intermediate-code environment of UOPT. The register allocation algorithm is presented, and issues related to performances, efficiency and implementation complexities are discussed.

Chapter 5 addresses the more practical aspects in the overall design, organization and implementation of the UOPT as a production optimizer. The interactions between the different types of optimizations are examined, and a specific order for performing the various optimizations is developed. Some data on the execution time requirements of the optimization phases are given. The optimization data structures in UOPT are presented. The actual methods used for

the collection of data flow information are examined. The effects of using procedure integration prior to entering UOPT are also discussed.

Chapter 6 evaluates the performance of UOPT, with respect to the optimizations performed and their effects on different target machines. Data on the contributions to overall performance of the different types of optimizations are presented. This indicates the relative importance of the various optimizations. We also study how optimization performance can be affected by some program and machine parameters. The effects of the common optimization results on a number of target machines with different machine characteristics are studied and compared. The machines considered are the DEC 10, 68000, VAX, MIPS, FOM and S-1. Means for predicting the effectiveness of some types of optimizations on different machines based on architectural features are developed. The overall evaluation serves to indicate the benefits of portable, machine-independent optimization in a retargetable compiler system.

Chapter 7 gives some concluding remarks, and suggests areas for further work.

2. The Intermediate Code

The use of intermediate languages in program translations has received increased attention in recent years [Chow83a]. Intermediate languages have traditionally been used to bridge the semantic gap between high-level source languages and low-level target code. Later, intermediate languages were defined as aids in the bootstrapping of self-compiling compilers into host machines [Amma75]. An interpreter, written in a language already available in the host machine, is used in the initial bootstrap phase. Once the interpretive language processor is available, the front-end together with the code-generation parts are rewritten in the language of the compiler. The interpreter can also serve to enhance the portability of the front-end compiler by standardizing the definition of the intermediate language [Bush79]. Present-day parser-generators output the results of syntactic analysis in the form of some symbolic representations. Retargetable code-generators use intermediate code as the starting points for generating object code.

The intermediate language used in a compiler system affects its portability, compilation and code generation efficiencies, and the source languages that can be supported. Its role as the interface between the machine-independent front-end and the machine-dependent code-generating back-end has a tremendous impact on the overall design of the different components of the system. When we include program optimization in the picture, the choice of the intermediate code becomes all the more important. The intermediate code affects the optimizations performed, the means of expressing the optimization results and the optimization efficiency. The portability, source- and machine-independence of the optimizer also depend on these same aspects of the intermediate code.

2.1. Goals of Intermediate Languages

Since the intermediate code affects so many different aspects in a compiling system, the following set of possible goals can be considered in designing and choosing an intermediate language:

1. The intermediate language should be able to support as many source languages as possible.

2. Interpretation of the code should yield the correct computational result without knowledge of the programming language origin of the code. All language operations should be clearly and explicitly expressed.

3. It should contain only a small number of op-codes and constructs for uniform representation of differing language semantics and source level constructs.

4. It should be in symbolic form, with no machine-dependent representation of computation whenever possible. For example, real constants should be represented as character strings.

5. It should have a simple and uniform syntax, and program representation should be compact. The context should not contain special declaration sections. Complete program information should be reflected in the code itself. Symbolic names and declarative information, if needed, should be put in separate symbol table files.

6. It should include information useful in optimization and code generation if the information can be gathered from the source code.

7. There should be maximum exposure of computations for purposes of optimization.

8. It should introduce no ambiguity in the control flow and data flow information to be collected. Such ambiguity sometimes comes from the certain characteristics of the source languages, and should be resolved by the compiling front-ends.

9. There should be some presence of the concepts of memory hierachy, including registers, to reflect storage structures in real machines.

Obviously, no single intermediate language is superior to all others in terms of meeting the above goals. Moreover, some of the above goals are hard to satisfy fully in the real world. Some arbitrary design decisions may lead to different language definitions. In the following, we discuss the important criteria from the point of view of performing machine-independent optimization. We limit our consideration to algebraic languages (Pascal, Fortran, C, etc).

2.2. The Level of the Intermediate Code

Program optimization can be performed at different levels of program code in the program translation process. At the high level, there is program optimization by source to source transformation [Schn73] [Palm75] [Love76] [Arsa79]. At the lower end, optimization is performed on the target machine code. The optimization at the low level usually involves using many machine parameters, and is highly machine-dependent. Most code generators perform some degree of target code optimization.

While it is possible to perform machine-independent optimization at any level of program code, an intermediate code level midway between the source and the target code has been the predominant choice. The main reasons are:

1. **Source and target independence:** Optimization at the source code level is language-dependent. Optimization at the target code level is machine-dependent. Optimization at the intermediate code level can be both language- and machine-independent.

2. **Visibility of optimizable code:** Source languages usually contain language implementation details which are inaccessible at the source code level, and can only be optimized after the high-level operations have been expanded into lower-level code. For example, offset computation in array references cannot be optimized at the source level. Also, similar source level text may convey different underlying operations. For example, the same symbolic variable name can specify both direct or indirect memory references. In general, the lower the level, the more opportunities we can find for performing optimization. But if the level is too low, machine characteristics creep in. Also, low-level machine details obscure the collection of information needed to perform optimization.

3. **Number of code constructs:** Source languages contain numerous high-level constructs which can be broken down to a much smaller number of low-level constructs. At the intermediate level of code, the optimizer only needs to deal with the limited number of intermediate level constructs. For example, computed `GOTO` statements in Fortran are represented similar to `CASE` statements in Pascal. Within the same source language, different loop constructs can be uniformly represented using jumps at the intermediate code level. At the target code level, the number of constructs again increases due to the instruction repertoire of the machine.

Performing machine-independent optimization on the intermediate code level does have limitations. Procedure invocations, manipulations of the display, various accesses via static and dynamic links cannot usually be optimized since the runtime organization is invisible at the intermediate code level.

With intermediate code, it is sometimes necessary to express the presence of computed quantities (temporaries) that need not exist when realized in the code of the target machines. This is because machine instructions may contain constructs more complex and high-level than the intermediate code. For example, address computations can be implicit in many addressing modes. Boolean evaluation often automatically sets the condition code that can be used to advantage in conditional jumps. Some data type conversions may correspond to no-op in some underlying machines, but this cannot be assumed on the intermediate code level. These are limitations we have to live with under the context of target-independent optimization.

2.3. The Form of the Intermediate Code

In this section, the different forms of intermediate code are considered with respect to their impact on optimization. Intermediate representations generally fall into one of the following three classes:

1. Tuples: This class comprises quadruples, triples, indirect triples (Section 7.6 of [Aho77]) and n-tuples [Frai79]. Indirect triples are triples with one level of indirection, in the form of a list of pointers to the triples, to provide flexibility in moving statements around.

2. Trees: They are usually associated with program graphs that represent the program statements and convey the overall program structure. Directed acyclic graphs (DAG), i.e. a group of trees with shared sub-trees, is also included under this category, since they belong to an optimized form of trees.

3. Linear representations (expressions): This class comprises the reverse Polish (prefix), the standard Polish (postfix) and the infix notations. Infix has the disadvantage of requiring the use parentheses, and is mainly suited for human comprehension.

To provide adequate program representation, the above classes do not exist in the pure form, because of the fact that special operations need to be specified at different points in the code. For example, jumps, function calls and other control constructs have to be allowed in the middle of an expression.

Some intermediate languages are in the form of an assembly code for an abstract machine, which may be a stack machine or a general register machine. We do not specifically consider these intermediate forms, since they either correspond to one of the above classes or are too low-level to be regarded as general intermediate representations.

We now want to consider which of the above forms of code are logically equivalent. Two forms of code are logically equivalent if a representation in one form can be freely converted to a unique representation in another form. Let us consider these forms under two different levels of representation requirements — without DAGs and with DAGs.

If we do not include DAGs in our consideration, then, with the exceptions of quadruples, all the above forms of code can be shown to be logically equivalent. The reasonings are as follows:

- Given a tree structure, the corresponding postfix can be formed by a post-order traversal of the tree, writing out the symbol for each node during the traversal. Similarly, the prefix form can be formed by a pre-order traversal of the tree, and the infix form can be formed by an in-order traversal, though in the latter case, parentheses need to be written out at every internal node. Conversely, given either postfix, prefix or infix notation, the corresponding tree structure can be formed. Such a process is similar to the parsing done by a syntactic analyzer according to the grammar specified.

- In the triple or indirect triple representation, each triple entry consists of an operator and its two operands, which can be regarded as representing an internal node of a tree. If

an operand is a leaf, the variable or constant is directly named. If the operand is another subtree, then it points to the entry for the internal representing the root of the subtree.

- N-tuples is a generalization of triples by enabling the specification of an arbitrary number of operands to be combined by the same operator. An N-tuple can be converted to a set of triples, and thus can be converted to and from trees.

- Quadruples are not logically equivalent to the others because they involve the definition of many temporary names which do not exist in the other representations. The extra information contents residing in the re-uses of the temporaries make quadruples different from the other forms. But if we impose the restriction that each temporary can be defined only once, then we in effect convert the quadruples to the triple representation.

Program flow graphs can be represented correspondingly in any of the above forms. These are usually in the form of jump instructions or pointers, depending on the context. We regard binary trees and program flow graphs as the canonical representation, since it is the easiest to visualize, analyze and manipulate.

At the second level, we require that the code also represents DAGs. In this case, only trees and triples (direct or indirect) are equivalent. The reason the rest are not equivalent to trees is as follow:

- To represent DAGs in postfix and prefix, it is necessary to define temporaries to store the results of common subexpressions. Again, the extra information contents residing in the re-uses of the temporaries make them logically different from trees and triples.

- In quadruples, there will also be extra temporaries used to store the results of common subexpressions. These temporaries are intermixed with the other temporaries that are present even without DAGs.

Therefore, trees and triples are the cleanest forms of program representation, because they do not require the definitions and uses of temporaries. In light of this, we consider quadruples, postfix and prefix as program representations at a lower level than trees and triples, with quadruples being the lowest of all the forms, since they express extra details about the usage of temporary names.

Quadruples stand apart from the others as a distinct form of code with many characteristics of its own. It is important to evaluate its advantages and disadvantages with respect to optimization.

Advantages of Quadruples:

1. Quadruples are closest in format to many target machine instruction sets, since machine instructions by and large perform single operations and store the results.

2. Every expression is broken down and named, making it is easier to move computations around.

3. The presence of numerous temporaries makes it possible to perform optimization related to the temporaries (e.g. subsumption).

4. The temporaries allow the optimizer to perform more register optimization, since registers containing by-products of arithmetic operations are not hidden to the optimizer.

Disadvantages of Quadruples:

1. Quadruples limit the machine-independence of the optimizer, since not all machines have the 3-address instruction format.

2. Since whole expressions are broken down, it is difficult for the optimizer to manipulate whole expressions, or perform transformations that involve tree-restructuring like stack-height reduction. Deep common subexpressions are harder to recognize.

3. It is possible that some of the named temporaries need not be present in the object code, and the subsumption of these temporaries in turn creates overhead.

4. Temporaries not allocated in registers are not necessarily of help to the code generators, since the temporaries may duplicate registers that need to be used as operands due to restrictions imposed by instruction formats: For example, in some machine instruction formats, the operands must be register-residing, so that the temporary must first be transferred to a register.

5. Even for temporaries residing in registers, the benefits may also be restricted by non-orthogonality in the instruction set architecture. For example, some machines require the operands of multiplication to be in specific registers, so that additional register moves are often required.

6. Temporaries occurring as intermediaries in address computation expressions may also be superfluous since whole address expressions may be translated to individual operand addressings using special addressing modes, or there may exist specific op-codes that map to the expressions.

In summary, from the machine-independent optimization standpoint, since postfix, prefix and quadruples are at a lower level of semantics, it can be concluded that trees and triples are the preferred intermediate forms. If the input program code does not contain DAGs, as in most unoptimized programs, then postfix and prefix are just as good as trees and triples. If postfix or prefix is used, then the optimized output can use generated temporaries to represent DAGs. The tree representation is more a structure than a form of program code, and so cannot be considered as a choice for intermediate code, but rather as a preferred form of internal representation. Triples, postfix and prefix can readily be converted to internal tree representation by the optimizer.

2.4. Other Requirements

Next, we consider other features of intermediate code that can enhance its use as a medium for machine-independent optimization.

To support optimization related to address computation, the intermediate code must include the effect of storage binding. All symbol references in the source program must have been replaced by their memory addresses. Without the specification of offsets, address collapsing and similar address-dependent optimizations cannot be performed. Moreover, the use of addresses allows the optimizer to necessarily distinguish between local, non-local and static variables, and detect storage relationships like equivalences, which affect the data flow information being collected and analyzed.

Register allocation optimization identifies variables, temporaries and evaluation results that should reside in registers at different regions of the code. Such optimization can be specified by attaching a register attribute to variables, which may also identify the register number. This method of specification does not allow the assignment of different variables to the same register throughout the course of a procedure, unless some kind of range specification (e.g. range of current statement, basic block or procedure) is used. An alternative is to regard registers as specific memory elements in the intermediate code, specified by either addresses or register numbers, which are to be mapped to actual machine registers. These registers can be grouped to different classes if required by the target machine architecture. By treating the registers as distinct objects, register-to-memory or memory-to-register transfer operations can be explicitly specified in the optimized intermediate code. Under this scheme, the optimizer is allowed to determine and specify the optimal positions for placement of register transfer code in addition to performing register allocation and assignment. Efficient register management is important for the speed of the optimized code.

Common subexpressions can be expressed by using attributes to flag expressions which are redundant and do not need to be computed more than once. This method of specification, however, does not convey the fact that there is cost associated with the saving of a computed expression and the later re-use of it. Also, the responsibility of allocating the temporaries or registers to store the expressions has to be left to the code generators. The alternative is to have the temporary together with the code that saves the computed expression explicitly specified. In this case, apart from optimizing the allocation of temporaries, the optimizer can go a step further to determine the best positions to insert the save code.

As the result of these additional requirements, an intermediate language suitable for optimization has to be of a lower level than the traditional intermediate representation which is completely machine-independent. But the level of the code must not be so low as to affect its portability.

In addition to the above, the intermediate code should be a widely used form of code. This serves to increase the acceptability and applicability of the optimizer.

2.5. The Overall Compilation and Optimization Plan

Though the optimizer transforms intermediate code independent of the source, knowledge of source language features can help it make better decisions in some cases. For example, in Fortran, all references to the global static memory can be treated as local references, and all non-static memory elements are either parameters or compiler-generated temporaries. Both these facts are not true in Pascal. Thus, if the intermediate code supports more than one source programming language, the intermediate code should contain some identification of the source that produces the code, or should be able to indicate the key features that may not be visible in the code itself. If these are not known, the optimizer has to make the worst assumptions to safeguard against incorrect optimized output.

On the other hand, the machine-independent optimizer also requires the knowledge of some machine parameters. These machine parameters include the different types of memory (storage hierarchy), the word lengths, the sizes of the data types, the structure of the activation records, the number and classes of registers and estimates of transfer cost between registers and memory. For the optimizer to be portable and machine-independent, these machine parameters must not be built into the optimizer.

There are two ways to make machine parameters available to the optimizer. The first method is to have the intermediate code contain all necessary machine descriptions, using special option specification instructions if necessary. Such a scheme has the disadvantage of making the

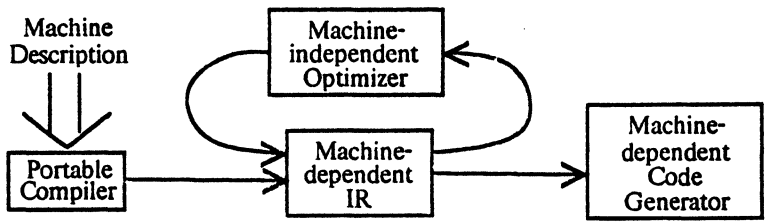


Fig. 2.5.1 Machine-dependent Intermediate Representation (IR)

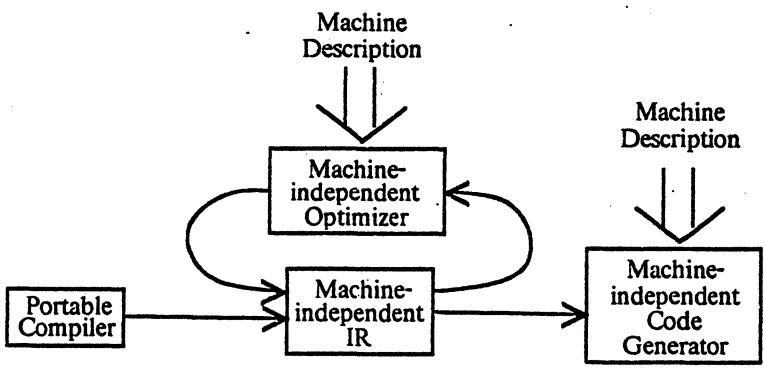


Fig. 2.5.2 Machine-independent Intermediate Representation (IR)

intermediate code and thus the compiling front-end machine-dependent. The common strategy is to supply the machine parameters to the portable front-end separately, either by conditional compilation or by look-up during execution of the front-end (Fig. 2.5.1). The intermediate code in this case is usually closer to the form of a code generation language based on an abstract machine model, and the code generators follow the interpretive code generation scheme [Gana82].

The second method is to feed the machine parameters to the optimizer directly. This can be done either by conditional compilation or by separate look-up while performing optimization (Fig. 2.5.2). This scheme allows the intermediate code and the front-ends to be totally machine-independent. The corresponding code generators usually follow the pattern-matched or table-driven code generation schemes.

2.6. The U-Code Intermediate Language

From the above discussions, it can be concluded that an intermediate language in postfix, prefix or triples form, at a level low enough so as to reflect the results of storage-binding and the availability of registers, is the ideal choice for performing machine-independent optimizations. The U-Code intermediate language is one that satisfies most of these criteria.

U-Code originated as an intermediate form for the Pascal language. The idea of an intermediate language for Pascal existed from the first portable Pascal compiler [Amma75], which emitted the Pascal pseudo-code P-Code [Nels79]. While P-Code is adequate as an intermediate code for translation purposes, it does not lend itself well to supporting optimization. U-Code, short for Universal Pascal code, incorporates P-Code as the base language along with the additional information to allow for optimization at the intermediate code level [Perk79] [Nye81]. By putting in minor extensions, U-Code has been made applicable for representing Fortran programs as well [Chow80]. Thus, U-Code is largely source-independent.

U-Code programs are in the form of a linear list of instructions, with each instruction identified by an operator. It is basically a form of reverse Polish notation and is defined in terms of an evaluation stack used to specify all computations. Control flow is specified using labels and jump instructions. In a U-Code program, all variables in the source program have been resolved into addresses in a hypothetical machine. Information about the symbolic names as used in the source program resides in separate symbol table files which are used only by a debugger. The run-time organization of the abstract stack machine is characterized according to the run-time model of Pascal, with a memory stack containing procedure activation records and the heap for dynamically allocated data records and static and dynamic links. Each activation record is divided into areas for representing different types of stored objects which can be parameters, local variables or temporaries. In addition, there are global (static) memory areas and registers. The registers are divided into classes to provide for special-purpose registers such as address registers or floating point registers. Thus, storage structures in the underlying machines are adequately represented.

The different memory areas including registers are referred to by unique memory types. Variables local to a procedure are referred to by the number of the procedure they are in and their offsets within their particular memory areas in the stack frame of the latest instantiation of the procedure. Global variables are referenced by their offsets in their particular static block. Although the U-machine is primarily a bit machine, it also has a word size, which is the size of an unpacked integer on the target machine, and an addressable unit, which is the smallest unit that can be directly addressed on a target machine.

Objects in memory and on the stack always have associated data types. As in real target machines, two objects with different data types or different sizes can occupy the same location in memory. Objects never overlap different memory areas. Each data type has an implied size when the data object is on the computation stack, except for the Set type. Data objects of type M are never loaded on the stack. Instead, their addresses are loaded, and all operations are performed indirectly. The size of the data object in memory may be less than its size on the stack, as is the case in packed records and arrays. Thus, many of the U-Code instructions have size specifications in addition to data types.

U-Code programs are not completely portable, since a given version of a U-Code program does contain machine-dependent parameters. These parameters are given to the portable front-ends according to the scheme of Fig. 2.5.1. The machine dependent parameters in U-Code programs are minimal, and they include the word and byte sizes, the default sizes of each data types and their alignment restrictions, and the structure of the activation records. Highly machine- and system-dependent mechanisms, like the use of the display, passing of parameters, procedure linkage conventions are not expressed or visible in U-Code. Currently, code generators exist that translate U-Code to object code for the DEC 10/20, VAX, MC68000, S-1, MIPS and FOM. They belong to the interpretive model of code generation. A U-Code interpreter written in Pascal also exists [Bush79].

U-Code is not completely language-independent in that it supports the Pascal model of static and run-time organizations, and the semantics of most operations follows that of Pascal. Most of these assumptions are visible in the U-Code context, and by suitable simple extensions, it is possible to make U-Code support most algebraic languages (e.g. Algol, C, PL/1).

The U-Code stack is usually implemented by registers in the underlying machine, and the U-Code operators describe the operations to be performed on the items on the stack. This stack orientation, however, causes inflexibility in the way that the items on the stack can be manipulated, since only the top items can be operated on. When an item is loaded on the stack in U-Code, many code generators do not actually load the item until the time that it is involved in computation. This is because an item may reside at a lower part of the stack for a long time while many other computations occur on the items near the top of the stack. A problem that arises involves storing when the stack is non-empty. Such a store can change the value of a location which has previously been loaded and still resides further down the stack. This complicates the implementation in those code generators that delay loads. However, the stack orientation of U-Code is inherent in the postfix form of code.

The storage relationships among the data objects are adequately represented, so that data-flow information can be collected with no ambiguity. Each data object is uniquely identified

by its memory type, block number and offset. Local, non-local, indirect memory references are distinguished. Storage relationships are clearly expressed by the size specifications of the data objects, so that equivalences and overlapping objects can be recognized. In array references with associated offset computation, the base address and length specifications in the LDA instruction precisely indicate the range of addresses where the resultant array element can be located. Possible side effects and aliases can be recognized. These enable the optimizer to pinpoint data objects that can be affected in memory references and assignments, which helps it prevent unsafe optimizations.

In addition, the instruction set of U-Code is versatile enough to expresse most needed operations. All program computations are exposed, and all implicit conversions are specified whether or not they translate into actual machine operations (e.g. the CVT instruction). Common subexpressions can be saved using the NSTR (non-destructive store) instruction. The RLOD and RSTR instructions permit the specifications of transfer operations between registers and memory. All these features make U-Code suitable as a medium for performing machine-independent optimizations.

The U-Code optimizer, UOPT, gets most machine parameters from the input U-Code itself. In addition, a few other machine parameters that are not available from the U-Code are set in the optimizer by conditional compilation. Included in them are the sizes of the various data types, which are needed in performing constant expression computations. Some parameters about the stack frame are also given for the purpose of deciding where to allocate temporaries generated by the optimizer. For the purpose of performing register allocation, information about the number and kinds of registers, the cost of register-memory transfer operations and comparisons of register and memory fetch times is needed. Thus, the optimization plan used by UOPT is a mixture of the two schemes shown in Section 2.5.

Only a small portion of the information present in U-Code programs output by the front-ends is intended for use by only the optimizer. The optimizer does not introduce its own U-Code constructs in expressing optimizations performed. The code generators do not have to distinguish between optimized and unoptimized U-Code in inputing and translating programs in order to take full advantage of the optimizations performed. Thus, both the front-ends and back-ends need not specifically accommodate the presence of the optimizer. The efficiencies of both the front-ends and back-ends are not affected.

Appendex A gives more details about the U-Code intermediate language.

3. Optimization Algorithms

In this chapter, the optimization algorithms in UOPT are presented. Section 3.1 describes the local optimization algorithms. The remaining sections address global optimizations. Local optimization is performed before global optimization because the latter has to rely on information gathered during the local phase. All the global optimizations are based on data flow analysis, and they are closely related to each other because some of them use similar global data flow attributes, and some of them are performed at the same time. The global optimization algorithms are characterized by ubiquitous bit vector operations, especially when solving data flow equations, which represent the bulk of the processing. Apart from constructing the program flow graph while inputing the program, no control flow analysis is needed in any of the global optimizations.

3.1. Local Optimizations

Local optimizations refer to the optimizations done within individual basic blocks [Aho72] [Bagw70]. A basic block is a straight-line block of code of maximal length with no branch except at the entry or exit. Maximal length is a desirable feature since it increases the opportunities for the various local optimizations. A basic block corresponds to a node in the control flow graph representation of a program.

The local optimizations in UOPT are done by straight transformation on the program-representing data structures. The building of these structures and the local optimizations also serve to prepare for the global optimization phases. After the local optimization phase, the more unified code form exhibiting more commonly-occurring code structures can serve to expose more global optimization opportunities. There is no peephole optimization pass on the intermediate code as in [Tane81] and [Wilk83], since our local transformations already include many of these peephole optimizations, and the rest can be done by case analysis of specific code constructs at appropriate times during the local transformations and the later code re-emission. In general, peephole optimization on intermediate code is useful and cost-effective only when no other major optimization transformation is present, so that there is no other mechanism or process available on which to overlay the checks for the occurrences of specific code constructs. In the case of UOPT, the precise internal representation of the program code and different kinds of code transformation make it unnecessary to do peephole optimization by pattern-matching specific code sequences in the intermediate code text.

3.1.1. Value Numbering

Value numbering is a technique for recognizing commonly occurring computations within a basic block [Cock70]. It is an efficient method for building directed acyclic graphs (DAG's) using a hash table and a triple representation.

The hash table is used for storing all expression trees. Each entry in the hash table is either an operand (leaf) or an operator (internal) node. The hash table index of each entry corresponds to its unique value number. For operator nodes, the table entry is in the form of a triple consisting of (op, l, r). The l and r fields are the value numbers of the left and right subtrees or leaf operands respectively†. The entries are determined by hashing using the open addressing with linear search scheme. Variables are hashed according to their addresses, and constants are hashed according to their values. Internal nodes are hashed according to the triple (op, l, r). Since the same entry can be hashed to by entries that are not identical, collision in hashing is resolved by entering the new entry in the next empty entry down the table. Thus, in finding the table entries for expressions and operands, hashing is accompanied by searching, and the uniqueness of value numbers is guaranteed by the resolution of collisions. For commutative operators, l and r are allowed to be interchanged in searching for a match. To retain information about the order of occurrences of the expressions in the basic block, a linked list representing the statements in their execution order in the basic block is used. These statement nodes point to the expression trees in the hash table that they reference (Fig. 3.1.1).

Local common subexpressions are recognized when two expressions yield the same value number. To prevent the recognition of common subexpressions that are identical but which no longer yield the same results because some of the operands have been assigned new values, it is necessary to assign new value numbers for later occurrences of the same expressions. This is effected by the *killing* of variable entries. A variable is killed whenever there is an assignment that can potentially alter its value (Section 5.3). This not only applies to direct assignments, but to indirect assignments as well. The effects of *aliases* and *equivalences* have to be included also. After a variable entry has been killed, it is prevented from being recognized in the searching that follows the hashing. Thus, a new entry with a new value number will automatically be created out of an empty entry. Since the variable is given a new value number, any expression that directly references it will have a different l or r operand value number; after hashing, the value number of this expression will have no relation to the value number of the identical expression that references the variable with the old value. The same applies to any larger expression in which the expression is nested. Thus, expressions do not need to be killed, since the different l or

† Only the l field is used for unary operators.

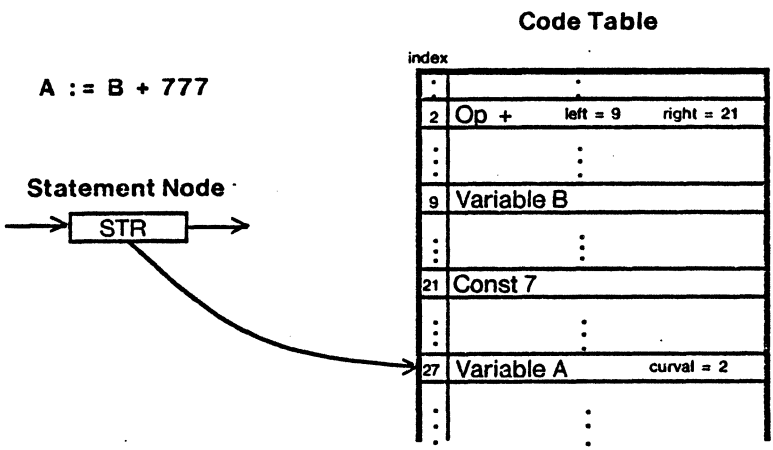


Fig. 3.1.1 Internal Representation of Basic Block Code

r operand value numbers automatically prevent them from being wrongly recognized. Constants also do not ever need to be killed, since they always represent the same values in computations.

In arrays, each array element is not assigned a value number. In fact, address computations and their subsequent indirect references are treated no different than other expression trees. Thus, an expression that leads to referencing a memory element is assigned a unique value number. These indirect references, which include indirect loads (ILOD's) and indirect comparisons (IEQU's, ILES's, etc.), also need to be considered for being killed, since they belong to the category of memory references. It is possible that the value at the address yielded by an address expression is changed between its multiple references via the address expression. This situation is taken care of by killing the entry of the indirect operator.

The optimizer removes redundant assignments in a basic block. Each direct assignment to a variable usually results in creation of a new value number for the variable, but if the variable has not been directly or indirectly (as in aliases and equivalences) referenced, then the previous assignment can be eliminated, and the same value number is used for the variable with the newly assigned value.

3.1.2. Local Copy Propagation

In the representation for variables in the hash table, a *value* field gives the value number of

the expression that was previously assigned to each variable in the basic block if there has been such an assignment (Fig. 3.1.1). The optimizer performs local copy propagation by looking up this field whenever a variable is referenced. If the *value* field indicates a previously assigned expression, variable or constant, the assigned expression is used instead of the variable itself. This implicitly creates an additional common subexpression reference. A special case is when the assigned expression contains operands whose values have been changed, as indicated by their having been killed. In this case, no copy propagation is performed.

Local copy propagation is useful for a number of reasons. First, a variable reference is replaced by a copy, which will be made fast since the later register allocation phase will allocate registers to store intermediate quantities which are referenced more than once. Second, by substituting variables with their values, it is possible to recognize more common subexpressions, since two or more variables with the same assigned values are identically mapped. Third, a larger common subexpression can be successively constructed across statement boundaries. Lastly, more redundant assignments can be exposed, since eliminating all references to a variable before the next assignment to it makes the first assignment redundant.

Example.

$$\begin{array}{ll} a = b \times c & t = b \times c + e \\ d = a + e & d = t \\ \quad \text{becomes} & \\ f = b \times c + e & f = t \\ a = d & a = t \end{array}$$

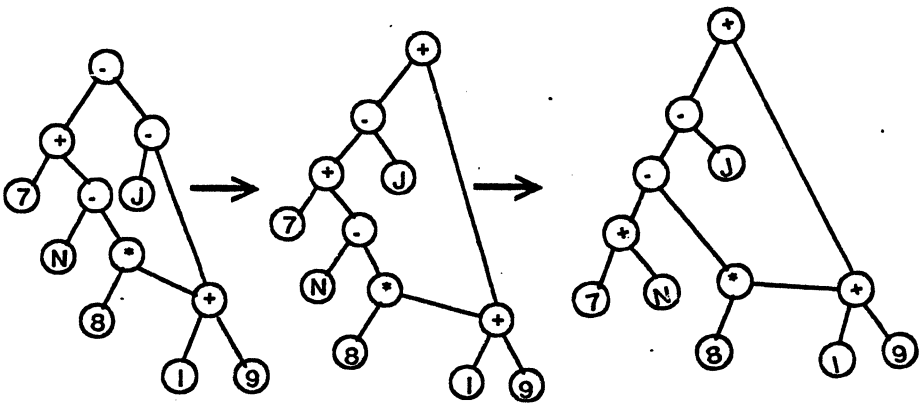
where t is a temporary

Local copy propagation automatically performs local constant folding for variables, when the copied expression is a constant value. This can potentially lead to more opportunities for constant arithmetic later.

3.1.3. Stack Height Reduction

Since the evaluation stack in U-Code is usually realized as registers in the target machine after translation, minimizing the height of the stack during expression evaluation can reduce the chance of spill-over of the stack items from the registers into slower memory. In the internal tree representation of the expression code, the goal of the transformation is to make the larger expressions appear on the left of the binary operators as much as possible.

There are two approaches to stack height reduction. The first method involves re-association between operators of the same precedence level. Tree restructuring is applied so that the tree is reduced to the left-associative form with each operator node weighted on the left-hand side. This



Stack height = 5 Stack height = 4

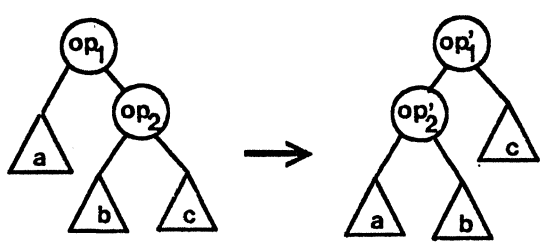
Fig. 3.1.2 Stack Height Reduction by Re-association

process leaves the order of appearances of the operands intact. To avoid destroying common subexpressions, the transfer of operands into and out of common subexpression subtrees is specifically avoided (Fig. 3.1.2). The algorithm for tree restructuring is recursive, and is applied to each internal node:

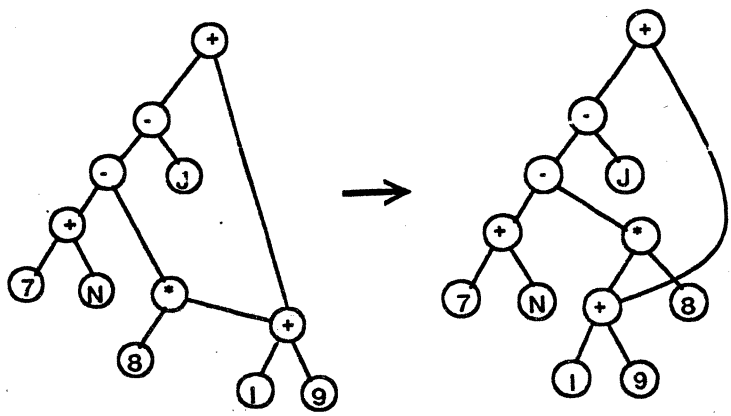
Algorithm *Restructure.*

1. Call *Restructure* for the right subtree of the current node.

2. If the operator of the right son is of the same rank, transfer the right son's left son to the left of the current node by creating a new internal node on the left side, and make the right son's right son the new right son:



3. Call *Restructure* for the left subtree of the current node. □



Stack height = 4 Stack height = 3

Fig. 3.1.3 Stack Height Reduction by Swapping Left and Right

After the re-association transformation, any expression containing only operators of the same precedence level can be evaluated with a stack height of two. Operators that can be transformed by re-association, grouped by their precedence levels, are: (a) +, −, IXA (indexing on address), (b) ×, floating point /, (c) AND, (d) OR, (e) INT (set intersection) and (f) UNI (set union).

The second method involves reversing the order of the operands of a binary operator so that the one with higher stack height is evaluated first (Fig. 3.1.3). For non-commutative operators, the two top items have to be swapped afterwards to preserve the correctness of the code. The extra swap does not usually cause extra object code to be generated. Apart from expression trees, this transformation is also applicable to statement operators which reference more than one expressions. Such statement operators include ISTR (indirect store), MOV (record copy), NEW (create record) and DSP (dispose of record).

After stack height reduction, all expressions containing $(2^n - 1)$ or fewer operands can be evaluated with a maximum stack height of n.

3.1.4. Constant Arithmetic

This involves replacing an operator with constant operands by the constant value obtained by performing the computation during optimization. Related to this are the reduction of an AND operator with a FALSE operand to FALSE, and the reduction of an OR operator with a TRUE

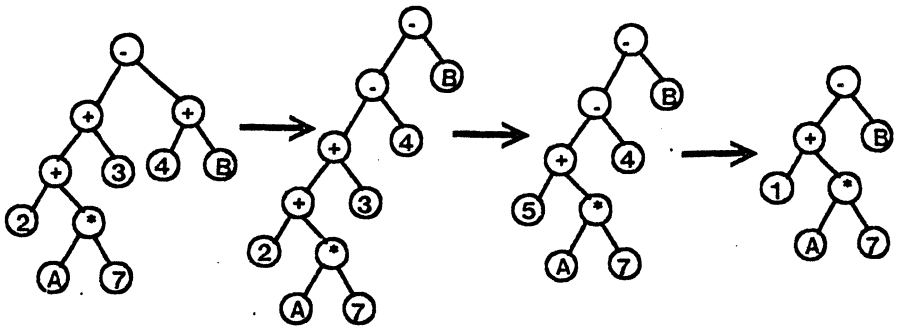


Fig. 3.1.4 Constant Collapsing

operand to TRUE. An AND operator with a TRUE operand is removed, and so is an OR operator with a FALSE operand. These operations for AND and OR also have corresponding operations for set intersection and union (INT and UNI). Bound checks of constant operands are performed, and any bound check error is reported. Decrements and increments of addresses can be folded into ILOD and ISTR instructions. When the operands of these same ILOD and ISTR instructions are constant addresses, direct loads and direct stores can be used instead. Conditional jumps with constant conditional expressions are either removed or replaced by unconditional jumps depending on the conditions evaluated.

An additional type of constant arithmetic is the combination of non-adjacent constants belonging to separate nodes of a tree. This is performed in conjunction with the tree-restructuring algorithm above (Fig. 3.1.4). After the tree is converted to the left-associative form, a constant can be moved downwards along the left-weighted branch to combine with another constant. This process is repeated until only a single constant is left hanging along a branch made up of operators of the same precedence level.

Another optimization related to constant arithmetic is the application of the distributive law. In the expression $a \times (b+c)$, when a and either b or c are constants, applying the distributive law to yield $(a \times b) + (a \times c)$ allows two constants to be combined. The resulting expression has the same number of operations, but under the condition that there are adjoining operators of the same precedence level as the + operator, this transformation can create opportunities for stack height reduction and constant collapsing (Fig. 3.1.5). If this condition is not met, the distributive law transformation is not applied.

The above transformations in constant arithmetic are performed by a single recursive procedure *ConstArith* which also makes use of the earlier *Restructure* algorithm. *ConstArith* is applied to each internal node regarded as the root of a subtree:

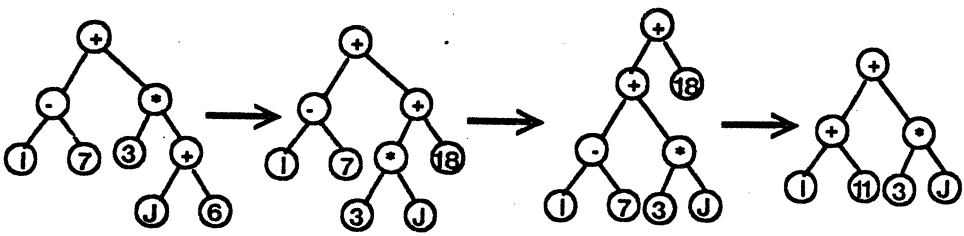


Fig. 3.1.5 Application of the Distributive Law

Algorithm *ConstArith.*

1. Call *ConstArith* for the right subtree of the current node.
2. Call *Restructure* for the current node.
3. Call *ConstArith* for the left subtree of the current node. (This completes the conversion of the subtree at the current node to the left-associative form, and also guarantees that there is at most one constant left hanging along the left-associative branch.)
4. If the right subtree is a constant or the operator of the current node is `INC` or `DEC`, then
   (a) if the left subtree is a constant, then apply the operator to combine the constants and convert the current node to a constant bearing the value of the result;
   (b) if the left subtree is not a constant, then if there is a constant further down the left-associative branch (or there is a `INC` or `DEC`), call *MergeConst* which combines the constant at the right son (or the `INC` or `DEC` parameter) to the lower constant and deletes the current node.
5. Apply the distributive law if this is beneficial. □

Stack height reduction by re-association and the merging of non-adjacent constants are not applied across common subexpression subtrees, since these transformations may render the common subexpressions invalid.

3.2. Overview of Global Optimization Strategy

Global optimiations rely heavily on the availability of global data flow information computed by data flow analysis. A global optimizing pass typically begins with a data flow analysis phase. Subsequently, the appropriate pattern matching and code manipulation operations are undertaken to perform the given optimization. The data flow analysis phase can be concisely and efficiently performed for the different types of optimizations. The second program manipulation

phase is not as straighforward, and usually requires a much more substantial amount of code to implement. Repeated passes over the program code are often needed to detect all possible optimizations. Since the program manipulations for the different types of optimizations are different in nature, the whole global optimization process is inevitably divided into a large number of passes, all of which have their own data flow analysis and program manipulation phases. The program manipulation phases are *ad hoc* and bear little relationship to each other.

The central strategy of our global optimization approach is to let data flow analysis assume a greater role in processing optimization transformations. The goal is to shift as much processing as possible to the data flow analysis phases. Apart from computing data flow information, the data flow analysis phases also take up the responsibility for determining the actual code transformation (insertions, deletions) to be performed. Although the data flow analyses become more involved, the program manipulation phases are much more simplified. Since data flow analysis can be implemented by a well-established set of code, the overall global optimization structure can be made much more manageable.

Because the program manipulation portion of the processing is reduced in size and complexities, our approach also makes possible the identification of the following three broad categories of global optimizations:

1. Uses of copy information — This includes copy propagation and constant propagation.

2. Backward code motion and backward redundancies — This includes global common subexpression elimination, loop-invariant expression removal and partial redundancy elimination.

3. Forward code motion and forward redundancies — This includes the elimination of fully or partially redundant stores, dead variable elimination, loop-invariant assignment removal and the optimization of temporary saves.

Optimizations belonging to the same category are similar in nature and not distinguished from each other. They are performed concurrently by the same process. Thus, it can be seen that the above three optimizations already include up to 80 per cent of all useful global optimization transformations. Moreover, since the data flow analyses can determine all the desired transformations at once, no incremental update of data flow information is required after each change to the code. Updates of data flow information is needed only between the small number of global optimization passes. Thus, it can be seen that the global optimization framework in UOPT offers significant advantages in reducing the complexities of both the optimizer implementation and the optimization phase structure. The optimization speed is also enhanced.

3.3. Boolean Attributes for Global Optimization

The global optimization of programs requires the knowledge of data flow information within procedures. This data flow information, in the form of booleans, can be divided into *local* and *global* attributes. A procedure text is represented by a directed control flow graph, with each node in the graph representing a basic block. A local attribute depends only on the basic block in which a variable, expression or assignment occurs. A global attribute is determined by the interaction of the local attributes in the set of basic blocks.

In this section, the attributes which are used in our global optimization algorithms are defined. We also consider how these attributes can be collected or computed from the program.

3.3.1. Local Data Flow Attributes

Our ideas of boolean attributes apply to variables, expressions and assignments (or definitions). These attributes are defined in terms of basic blocks. Some attributes use the entries or exits of basic blocks as points of reference, and some refer to entire basic blocks. The direction of flow considered may be forward or backward in relation to the flow of control of the program.

There are three local attributes for variables, defined as follows:

ANTLOC - (Locally Anticipated, Locally Live or Locally upward-exposed) A variable is locally anticipated in a basic block if there is a use of the variable (which excludes assignment to the variable) within the block, and the value of the variable can in no way be affected if the use of the variable is moved to the entry of the block. In other words, there is no assignment in the block preceding the use of the variable which can potentially alters the value of the variable†.

AVLOC - (Locally Available) A variable is locally available in a basic block if there is a use of the variable within the basic block, and the value of the variable will stay the same if the use of the variable is moved to the exit of the block.

ALTERED - (Killed locally) A variable is altered in a basic block if its value may be modified by executing the code of the basic block. The variable does not necessarily have to appear in the basic block for it to be altered.

The above three attributes are made to apply to expressions by replacing the word *variable* in the above definitions by *expression*. The attributes of expressions represent stronger qualifi-

† The optimizer will try its best to decide if a given assignment can alter the value of a variable. If the information provided to it is not sufficient for making such a decision, it will regard that the variable can possibly be altered by the assignment, for the sake of safety.

cations than the corresponding attributes of the components of the expressions. An expression is ALTERED in a block if any variable within the expression is. If an expression is ANTLOC in a block, then any component of the expression must also be ANTLOC. A constant appearing in a basic block is always ANTLOC, AVLOC and not ALTERED. ANTLOC is a backward attribute and AVLOC is a forward attribute.

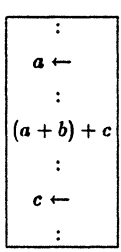
In applying the local attributes to assignments, the values assigned together with the variable being stored into are considered:

ANTLOC - An assignment is locally anticipated in a basic block if the assignment occurs within the block and the effect of the assignment on the result of executing the code of the block will be the same if the assignment is moved to the entry of the block. In other words, the assigned expression is ANTLOC, and the assigned location is unaltered and not used anywhere in the block before the assignment‡.

ALTERED - An assignment is altered in a basic block if the value of the assigned expression or the assigned location may be modified by executing the code of the block, and there is no use of the assigned variable in the block; if the assignment actually occurs in the block, then its own code is excluded from consideration in the determination of its ALTERED attribute. To state it in another way, an assignment is not ALTERED if there is no effect on the execution result by moving the assignment from one end of the block to the other end.

A variable, expression or assignment is not ALTERED if there is an occurrence in the block and that occurrence is both ANTLOC and AVLOC. An item can be both ANTLOC and AVLOC but ALTERED since there can be two occurrences and the altering is due to the code between the two occurrences.

Example.



	a	b	c	$a+b$	$(a+b)+c$
ANTLOC	F	T	T	F	F
AVLOC	T	T	F	T	F
ALTERED	T	F	T	T	T

‡ In accordance to U-Code syntax, the code for computing the assigned expression is always computed before the actual storing into the assigned location.

We have used code movement to characterize the above attributes. The reason is that these attributes will be used among other things in solving the feasibility of various kinds of code motion in the subsequent global optimizations. Also, the availability of complete information is critical. Side effects, aliases and equivalences often make it hard to obtain the exact use or definition information of a data item. In such cases, the most pessimistic assumption is made in obtaining the information in the attributes.

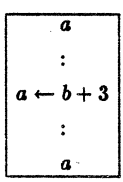
In the case of assignments, there are additional local attributes with slightly different meanings:

PAVLOC - (Partial Local Availability) An assignment is partially locally available in a basic block if the assignment occurs within the block and the assigned location still holds the value of the assigned expression which also has not changed before the exit of the block. In other words, the values of the assigned variable and assigned expression are not altered in the code of the basic block following the assignment.

ABSALTERED - An assignment is absolutely altered in a basic block if there is code in the basic block that can potentially alter the value of the assigned expression or the assigned location, excluding the effect of the assignment itself if it exists in that block.

The attributes PAVLOC and ABSALTERED differ from ANTLOC and ALTERED respectively in that the former do not take into account the usage of the assigned variable in the relevant region. The definitions of PAVLOC and ABSALTERED do not rely on code movements. PAVLOC is a weaker property than AVLOC. An assignment that is PAVLOC is not necessarily AVLOC, but an assignment that is AVLOC must be PAVLOC. An assignment that is ABSALTERED must also be ALTERED in a basic block, but an assignment that is ALTERED is not necessarily ABSALTERED. The PAVLOC and ABSALTERED are not used for solving code motion, whereas ANTLOC and ALTERED are. The former can be regarded as static data flow attributes and the latter can be regarded as dynamic data flow attributes.

Example.



	$a \leftarrow b + 3$
ANTLOC	F
ALTERED	T
ABSALTERED	F
PAVLOC	T

3.3.2. Global Data Flow Attributes

In constrast to local optimizations, global optimizations take into account the procedure's large scale structure in performing transformations. In defining the global attributes, we can just extend the meanings of anticipability and availability:

- A variable, expression or assignment is *anticipated* at a given point if all paths leading from it contains an instance of the computation, and the computation placed anywhere along the paths always deliver the same result.

- A variable, expression or assignment is *available* at a given point if all paths leading to the point contains an instance of the computation, and the computation placed anywhere along the paths always deliver the same result.

Partial anticipability and availability are weaker properties:

- A variable, expression or assignment is *partially anticipated* at a given point if at least one path leading from the point contains the computation, and the computation placed anywhere along the path always deliver the same result.

- A variable, expression or assignment is *partially available* at a given point if at least one path leading to the point contains the computation, and the computation placed anywhere along the path always deliver the same result.

The global attributes are usually applied to the entries and exits of basic blocks. ANTIN, AVIN, PANTIN and PAVIN denote these attributes at the entries of basic blocks, and ANTOUT, AVOUT, PANTOUT and PAVOUT denote these attributes at the exits. In practice, the attributes for different variables, expressions and assignments can be aggregately represented using bit vectors, with each bit position allocated to a variable, expression or assignment. The resultant bit vector operations substantially speed up the computations involving the attributes by a factor depending on the word size of the host computer.

The following system of boolean equations defines the global availability attributes based on the corresponding local attributes. The subscript i identifies the attribute as being for the ith basic block.

Availability System:

$$\mathtt{AVIN}_i = \begin{cases} \mathtt{FALSE} & \text{if } i \text{ is the entry block;} \\ \prod_{j \in \mathtt{Pred}(i)} \mathtt{AVOUT}_j & \text{otherwise.} \end{cases} \tag{3.3.1}$$

$$\mathtt{AVOUT}_i = \mathtt{AVLOC}_i + \neg\mathtt{ALTERED}_i \cdot \mathtt{AVIN}_i.$$

The first equation says that an item is available at the entry to a basic block if and only if it is available at the block exits of all its predecessors. The second equation says that a variable is available at the exit of a basic block if it is either locally available there or is available at the entry of the block and is not changed inside that block.

The other groups of global data flow attributes can similarly be computed by solving systems of boolean equations:

Anticipability System:

$$\mathtt{ANTOUT}_i = \begin{cases} \mathtt{FALSE} & \text{if } i \text{ is the exit block;} \\ \prod_{j \in \mathrm{Succ}(i)} \mathtt{ANTIN}_j & \text{otherwise.} \end{cases} \tag{3.3.2}$$

$$\mathtt{ANTIN}_i = \mathtt{ANTLOC}_i + \neg\mathtt{ALTERED}_i \cdot \mathtt{ANTOUT}_i.$$

Partial Availability System:

$$\mathtt{PAVIN}_i = \begin{cases} \mathtt{FALSE} & \text{if } i \text{ is the entry block;} \\ \sum_{j \in \mathrm{Pred}(i)} \mathtt{PAVOUT}_j & \text{otherwise.} \end{cases} \tag{3.3.3}$$

$$\mathtt{PAVOUT}_i = \mathtt{AVLOC}_i + \neg\mathtt{ALTERED}_i \cdot \mathtt{PAVIN}_i.$$

Partial Anticipability System:

$$\mathtt{PANTOUT}_i = \begin{cases} \mathtt{FALSE} & \text{if } i \text{ is the exit block;} \\ \sum_{j \in \mathrm{Succ}(i)} \mathtt{PANTIN}_j & \text{otherwise.} \end{cases} \tag{3.3.4}$$

$$\mathtt{PANTIN}_i = \mathtt{ANTLOC}_i + \neg\mathtt{ALTERED}_i \cdot \mathtt{PANTOUT}_i.$$

The above data flow equations can be solved using an iterative algorithm, as given in [Kild73] and [Hech73]. It involves applying the above equations to the nodes of the control flow graph until the information stabilizes. Depending on the initializations of the unknowns, different solutions can be obtained that satisfy the systems of equations. In the case of the conjunction operator $\prod$, the wanted solution is the one with the largest number of true bits. If the unknowns are initialized to TRUE, the unknowns will converge to the largest solution as iteration progresses. For the disjunction operator $\sum$, the wanted solution is the one with the smallest number of true bits. If the unknowns are initialized to FALSE, the unknowns will converge to the smallest solution during iterations.

There are other local and global attributes which are specific to the kinds of global optimization they support. These will be described in due course.

Appendix B presents some more details in programming data flow analysis using the iterative algorithm.

3.4. Copy Propagation

Copy propagation traditionally involves statements of the form $a \leftarrow b$. After determining all places where this definition of a is used, it may be possible to eliminate this statement by substituting b for a in all references of a. Standard algorithms for performing this copy propagation can be found in [Aho77].

The treatment of copy propagation in UOPT is slightly more generalized. Any assignment of the form $a \leftarrow \langle \text{expr} \rangle$ is considered, where $\langle \text{expr} \rangle$ is not limited to being a single variable. The copy propagation involves replacing variables by their known assigned expressions. In the case that the expression is a constant, the effect is global constant propagation.

By making use of the attributes defined in the previous section, our algorithm to perform copy propagation is simpler and more elegant than traditional ones. It turns out that the attributes PAVLOC and ABSALTERED together with the global attributes derived from them already contain most of the information needed to copy propagate. Let AVIN and AVOUT be the global attributes that indicate the availability of assignments. By substituting PAVLOC and ABSALTERED into Eq. (3.3.1), AVIN and AVOUT can be solved as follows:

Availability of Assignments:

$$\texttt{AVIN}_i = \begin{cases} \texttt{FALSE} & \text{if } i \text{ is the entry block;} \\ \prod_{j \in \texttt{Pred}(i)} \texttt{AVOUT}_j & \text{otherwise.} \end{cases} \tag{3.4.1}$$

$$\texttt{AVOUT}_i = \texttt{PAVLOC}_i + \neg\texttt{ABSALTERED}_i \cdot \texttt{AVIN}_i.$$

THEOREM 3.4.1. *A use of the variable a in basic block n can be replaced by the expression $\langle expr \rangle$ if all of the following conditions are met:*

(a) The assignment $a \leftarrow \langle expr \rangle$ is $\texttt{AVIN}_n$.

(b) The replaced variable a is $\texttt{ANTLOC}_n$.

(c) The expression $\langle expr \rangle$ is $\texttt{ANTLOC}_n$ if inserted at the point of the variable a in block n.

PROOF. Condition (a) implies that the assignment $a \leftarrow \langle \text{expr} \rangle$ is the only assignment to a reaching block n, and that both the values of a and $\langle \text{expr} \rangle$ have not been changed in the paths

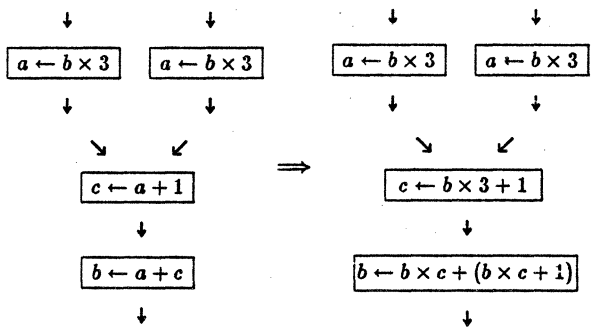


Fig. 3.4.1 Multiple Copy Propagation

that lead to block n. Condition (b) and (c) guarantee that the same is true in the region in block n preceding the point where a occurs. □

The algorithm to perform copy propagation can now be specified. The algorithm is applied to each variable reference in each basic block.

Algorithm *CopyPropagate.*

1. For each reference of a simple variable a, in basic block i, in which $\texttt{ANTLOC}_i$ is true, look for an assignment which is of the form $a \leftarrow \langle \text{expr} \rangle$ whose $\texttt{AVIN}_i$ is true. If this is found, then check that the expression $\langle \text{expr} \rangle$ if inserted at that point will cause its $\texttt{ANTLOC}_i$ to be true.

2. If the expression $\langle \text{expr} \rangle$ can be found in 1, then replace the occurrence of a by $\langle \text{expr} \rangle$. Apply the algorithm recursively to each variable reference in $\langle \text{expr} \rangle$. □

Since each new insertion of an expression creates new occurrences of variables in the basic block, the algorithm *CopyPropagate* is applied recursively in step 2 to ensure that copy propagation is done completely. At the termination of the algorithm, no more copy propagation can be performed in the program code (Fig. 3.4.1).

It is to be noted that if the attributes AVLOC and ALTERED were used in Eq. (3.4.1) instead of PAVLOC and ABSALTERED, the resultant condition to be satisfied in step 1 of the algorithm would be stronger than needed.

When a variable is replaced by its known assigned expression $\langle \text{expr} \rangle$, the resultant code could be worse if the expression is large. However, in all cases, the expression $\langle \text{expr} \rangle$ is a global common subexpression, and does not need to be recomputed. This is because the fact that the

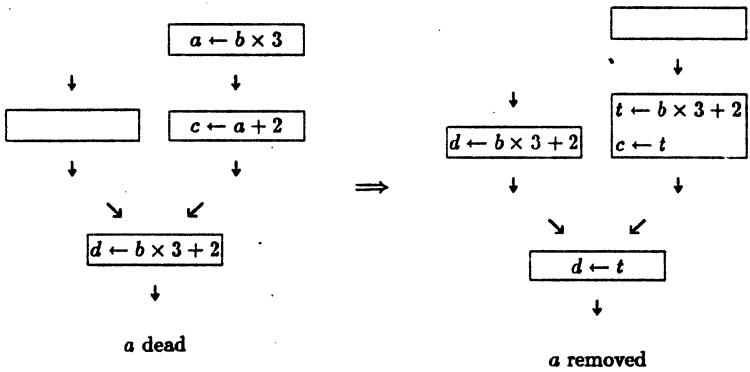


Fig. 3.4.2 Partial Redundancy in $b \times 3 + 2$ exposed by Copy Propagating through a

assignment $a \leftarrow \langle expr \rangle$ is $\texttt{AVIN}_i$ implies that $\langle expr \rangle$ is also AVIN, which is a sufficient condition that the expression $\langle expr \rangle$ is globally redundant. As a result, later redundant expression elimination and register allocation will replace $\langle expr \rangle$ by a load from a register in which the previously computed value of the expression is saved. In most cases, this is faster than a memory reference to the replaced variable a.

Apart from this, the other benefits of local copy propagation mentioned in Section 3.1.2 also apply in the global case. Since copy propagation is performed until no more copies can be made, variables and expressions are commonly mapped, and more common subexpressions can be exposed which would not otherwise be recognized. These common subexpressions can also be successively constructed across multiple basic blocks.

After replacing the variable a by the expression $\langle expr \rangle$, the assignment $a \leftarrow \langle expr \rangle$ can be made redundant. The elimination of these and other redundant assignments are done together in subsequent phases (Fig. 3.4.2).

3.5. Redundant Store Elimination

Redundant assignments are assignments to variables whose uses cannot be anticipated before the next assignments. In the case of local variables, assignments are also redundant if no more use of the variables occurs before procedure exit. In this case, the variables are called dead variables. A local variable is *dead* at a point if its value will not be used along any path in the procedure starting at that point.

(a) Redundant $a + b$ at node 3 (b) Redundant $a \leftarrow$ at node 1

Fig. 3.5.1 Duality between Redundant Expressions and Redundant Stores

Redundant assignments are traditionally found by solving for the liveness of variables appearing on the left-hand-sides of assignments. The assignment $a \leftarrow \langle \text{expr} \rangle$ is redundant if a is not live at the point of the assignment. However, this approach is complicated by the fact that a variable should still be regarded as live if there is an operation that may or may not change the value of the variable, as in function calls or indirect stores. If Eq. (3.3.4) in Section 3.3 were used in solving for partial anticipability or liveness, the resulting PAVOUT would not include variables that may or may not be live, and thus would not be applicable in finding store redundancies.

The approach to redundant store elimination in UOPT involves defining a set of new local attributes, which are applied to the uses of variables as the *L-values* (the assigned sides) in assignments. The same names are used for these new attributes, since they convey similar meanings, though in different contexts.

ANTLOC - The L-value of a variable is locally anticipated in a basic block if there is a simple assignment to the variable, and there is no effect on the execution result of the basic block by moving the assignment to the entry of the block, assuming the same value can be assigned. This means that in the code preceding the assignment, there is no use of the variable and no other indirect assignment that can potentially alter the value of the variable.

AVLOC - The L-value of a variable is locally available in a basic block if there is a simple assignment to the variable, and there is no effect on the execution result of the basic block by moving the assignment to the exit of the block, assuming the same value can be assigned. This means that in the code following the assignment, there is no use of the variable and no other indirect assignment or procedure call that can potentially alter the value of the variable.

ALTERED - The L-value of a variable is altered in a basic block if there is some reference to the variable, or some indirect assignment or procedure call that can potentially alter the value of the variable. Direct assignments to the variable are excluded from consideration.

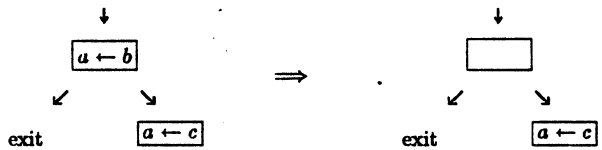
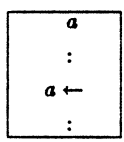


Fig. 3.5.2 Redundant Assignments (a local variable)

It is important to note the difference between the attributes for an assignment and the attributes for the L-value of a variable. The former refers to the assignment as an expression tree, whereas the latter refers the use of the variable on the left hand side of a direct assignment, even if different values are assigned at different times.

Example.



	$a \leftarrow$
ANTLOC	F
ALTERED	T
PAVLOC	T

By using these attributes, redundancies in assignments can be found by solving for the global anticipability attributes using Eq. (3.3.2).

THEOREM 3.5.1. *An assignment of the form $a \leftarrow \langle expr \rangle$ in basic block n is redundant if:*

(a) the local attribute AVLOC_n *for the L-value of a is true, and*

(b) the global attribute ANTOUT_n *for the L-value of a is true.*

PROOF. An assignment of the form $a \leftarrow \langle \text{expr} \rangle$ is redundant if additional assignments to a, of the form $a \leftarrow$, occur later regardless of the path taken, and in the intervening paths there is no potential reference or store to a. Condition (b) guarantees that assignments $a \leftarrow$ occur later, and in the intervening paths starting from the exit of block n, there is no potential reference or store to a. Condition (a) guarantees that in the region in the basic block n after the assignment, there is also no potential reference or store to a. □

The reason for our doing redundant assignment elimination different from traditional approach is because this method recognizes a duality that exists between redundant expressions and redundant assignments. The former refers to the computation of expressions, and the latter refers to the process of storing into a location. An expression which has been computed earlier

is redundant, while first stores into a location are redundant if they are followed later by other stores into the same location regardless of the stored values. The former is an availability problem, and the latter is an anticipability problem (Fig. 3.5.1). A major benefit of this approach is that this allows us to perform forward code motion involving assignments. This topic will be addressed later in Section 3.9.

This method also allows us to recognize redundant assignments to dead variables (Fig. 3.5.2). In the initializations to solve data flow Eq. (3.3.2) iteratively, ANTOUT$_i$ can be set to true for all exit blocks i and all variables which are local, and false otherwise. The effect is similar to inserting imaginary assignments to these variables just before the exits. Such a setup will enable the algorithm to expose the redundancies of assignments to dead variables.

3.6. Code Motion

Code motion optimization involves the backward movement of code from more frequently executed regions of the program to less frequently executed regions. The computations moved are usually invariant computations in strongly connected components of the program flow graph. To perform code motion, the loop-invariant computations must first be found. This requires the computation of *use-def* chains by data flow analysis. The use-def chains give the origins of the definitions that affect the variables inside the loops. After the loop-invariant computations are found, they are moved to the loop headers dominating all exit nodes in the loops involved. Finally, the invariant computations that are made redundant as a result of the insertions are deleted. All this analysis involves uncovering the loop structures embedded in the control flow graph using control flow analysis. The code motion is done loop by loop, and repeated passes over the same loop are often necessary to exhaust all possible code motion.

Morel and Renvoise [More79] have presented a method in which it is possible to perform code motion and the elimination of redundant expressions at the same time. They also generalize these optimizations to the suppression of partial redundancies. They view code motion as a program flow analysis problem in which positions to insert and delete code are determined once and for all by solving data flow equations. The resulting code movements are then from deleted positions to inserted positions. The algorithm does not require detailed analysis of the program control flow graph. The goal is to let flow analysis play the role of determining the profitability, correctness, origins and destinations of code movements, which were previously done by case analysis. This method of global partial redundancy suppression is adopted in UOPT with minor modifications. The approach has enabled us to acheive a concise, efficient and less costly implementation of the global optimizer.

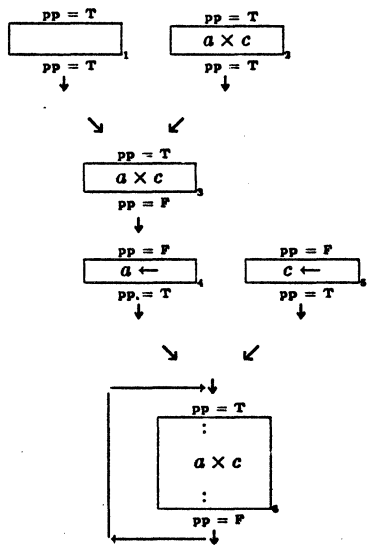


Fig. 3.6.1(a) The PP attribute for $a \times c$

Morel and Renvoise have pointed out that global redundant expression elimination and code motion are actually special cases in the global suppression of partial redundancies. A computation at a point is *redundant* if the computation is available at that point. A computation at a point is *partially redundant* if it is partially available at that point. The suppression of partial redundancies involves the determination of positions to insert computations that cause some partially redundant expressions to become redundant and be deleted, without introducing any new partial redundancy. Not all partial redundancies can be removed, but the method performs all code motion and removes all complete redundancies. We now present the steps that lead to the formulation of the partial redundancy suppression algorithm.

3.6.1. The Partial Redundancy Algorithm

Partial redundancy exists when an identical computation is performed more than once in a certain path in the program. The optimization transformation we are considering involves the insertion and deletion of computations at various points in the program. It is necessary that the transformation does not result in any path of the program flow graph containing more of the same computations than it contains before. This means that every insertion is at a point that the computation can be anticipated, and that all the anticipated first computations made after

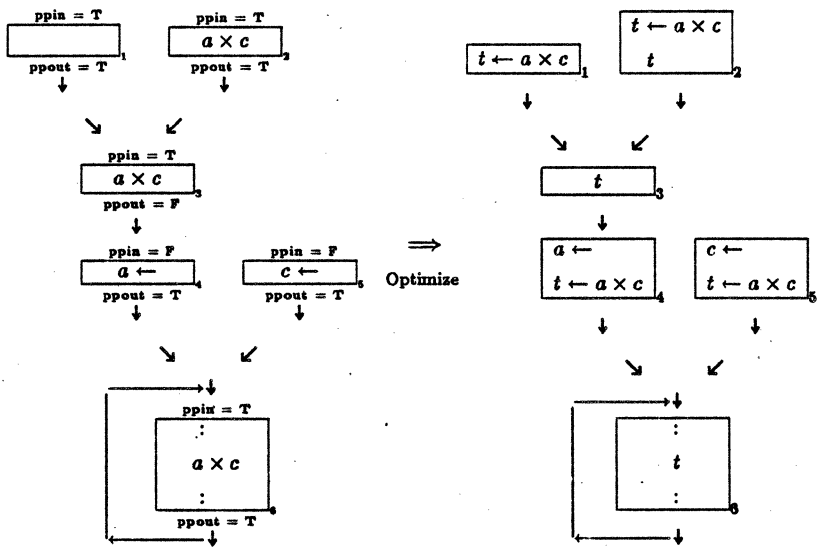


Fig. 3.6.1(b) Partial redundancy suppression for $a \times c$

that point are rendered completely redundant by the total effect of the insertions made. Global common subexpression is a special case in this optimization because it requires no insertion for the expression to become redundant. To establish positions to insert computations, we define a number of global attributes:

PP - (Placement Possible) A computation e is PP at a point p if it is anticipated at p and all the anticipated e's can be rendered redundant by zero or more insertions at that point and some other points in the procedure, and these insertions satisfy the conditions that the insertions are always at points that e is anticipated and the first anticipated e's after the insertions are rendered redundant (Fig. 3.6.1).

THEOREM 3.6.1. *If a computation e is* PP *at point p, then it is also* PP *at any point q on any path that leads from p to an anticipated e.*

PROOF. Since the computation e is anticipated at p and p leads to q before reaching e, e must be anticipated at q, and the set of occurrences of e anticipated at q must be a subset of those anticipated at p. Suppose p is established as PP by insertions at a set of points s. To establish that q is PP, apart from inserting at q, we can pick enough insertions from s until the e's

anticipated at q are all rendered redundant. □

For the sake of uniformity, we restrict all insertions to be at the end of basic blocks. This will have no effect on the optimizations that are to be performed. To generalize further, we also regard a computation to be placement possible when the computation is available, since no insertion is needed.

PPOUT - (Placement Possible on exit) A computation e is PPOUT at the exit of a basic block i if it is $\texttt{ANTOUT}_i$ and all the anticipated e's can be rendered redundant by insertions at the exits of block i and some other blocks in the procedure, and these insertions satisfy the conditions that the insertions are always at points that e is anticipated and the first anticipated e's after the insertions are rendered redundant; a computation e is also $\texttt{PPOUT}_i$ if it is $\texttt{AVOUT}_i$.

The purpose of the attribute PP or PPOUT is to determine the feasibility of insertions at particular points for the purpose of eliminating partial redundancies. To help solve for $\texttt{PPOUT}_i$, we also define PPIN for basic block entries:

PPIN - (Placement Possible on entry) A computation e is PPIN at the entry of a basic block i if it is $\texttt{ANTIN}_i$ and all the anticipated e's can be rendered redundant by insertions at the entry of block i and some other blocks in the procedure, and these insertions satisfy the same condition that the insertions are always at points that the e is anticipated and the first anticipated e's after the insertions are rendered redundant; a computation e is also $\texttt{PPIN}_i$ if it is $\texttt{AVIN}_i$.

As in the case of the other global attributes in Section 3.3.2, we can solve for PPIN and PPOUT by the following set of flow equations. The use of the $\prod$ operator in the second equation is implied by Theorem 3.6.1.

$$\texttt{PPIN}_i = \texttt{ANTIN}_i \cdot (\texttt{ANTLOC}_i + \neg\texttt{ALTERED}_i \cdot \texttt{PPOUT}_i).$$

$$\texttt{PPOUT}_i = \begin{cases} \texttt{FALSE} & \text{if } i \text{ is the exit block;} \\ \prod_{k \in \text{Succ}(i)} \texttt{PPIN}_k & \text{otherwise.} \end{cases} \tag{3.6.1}$$

The above solution for PPOUT does not give the best set of points for the final insertions. A necessary requirement to guarantee the profitability of the code transformation is that there must be no partial redundancy among the final set of insertions. We can partially satisfy this requirement by putting insertions at the earliest point in each simple path of consecutive blocks at which PPOUT is true. The insertion will then be available throughout the path. Thus, the condition to put insertion at a block exit, called INSERT, is:

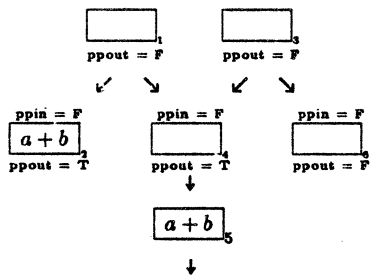


Fig. 3.6.2 PPIN and PPOUT of $a + b$

$$\texttt{INSERT}_i = \texttt{PPOUT}_i \cdot \Big(\sum_{j \in \texttt{Pred}(i)} (\neg \texttt{PPOUT}_j \cdot \neg \texttt{AVOUT}_j) + \texttt{ALTERED}_i \Big). \tag{3.6.2}$$

Eq. (3.6.2) indicates that we will put insertions at the exit of block i if it is PPOUT and at least one of the predecessors of i is not PPOUT and not AVOUT, or if the computation is altered in block i so that the insertion at the exit of block i will not be redundant. If all of the predecessors of block i are PPOUT, then the insertion at block i is redundant unless the computation is changed in that block.

After insertion at block i, we must prevent any insertion at the ancestors of block i that will become available at block i and thus would cause new partial redundancy with the computation inserted at block i. In other words, when the computation is not altered in block i, insertion at the exit of i should be prohibited if there is some insertions at some predecessors. This can occur only if the computation is PPOUT at the exits of some of the immediate predecessors of block i. Insertions should be put at block i only if the computation is not PPOUT at any of the immediate predecessor. Thus, we impose a stronger condition for insertions:

$$\texttt{INSERT}_i = \texttt{PPOUT}_i \cdot \Big(\prod_{j \in \texttt{Pred}(i)} (\neg \texttt{PPOUT}_j \cdot \neg \texttt{AVOUT}_j) + \texttt{ALTERED}_i \Big). \tag{3.6.3}$$

To use this formulation of INSERT, we also require that a computation be PPOUT at block i only if it is also PPOUT at all the predecessors of the successors of i. In Fig. 3.6.2, the expression $a + b$ is not PPOUT at block 1 because it is not PPOUT at block 3. We add the term $\prod_{j \in \texttt{Pred}(i)} (\texttt{PPOUT}_j + \texttt{AVOUT}_j)$ to Eq. (3.6.1) to get:

$$\texttt{PPIN}_i = \begin{cases} \texttt{FALSE} & \text{if } i \text{ is the entry block;} \\ \texttt{ANTIN}_i \cdot \prod_{j \in \texttt{Pred}(i)} (\texttt{PPOUT}_j + \texttt{AVOUT}_j) & \\ \quad \cdot (\texttt{ANTLOC}_i + \neg\texttt{ALTERED}_i \cdot \texttt{PPOUT}_i) & \text{otherwise.} \end{cases} \tag{3.6.4}$$

$$\texttt{PPOUT}_i = \begin{cases} \texttt{FALSE} & \text{if } i \text{ is the exit block;} \\ \prod_{k \in \texttt{Succ}(i)} \texttt{PPIN}_k & \text{otherwise.} \end{cases}$$

Eq. 3.6.3 can then be rewritten using PPIN as follows. The term AVOUT is added to exclude cases where the computation is available, when no insertion is needed:

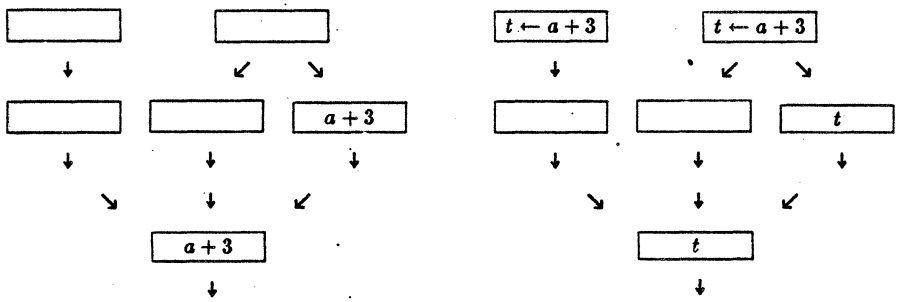
$$\texttt{INSERT}_i = \texttt{PPOUT}_i \cdot \neg\texttt{AVOUT}_i \cdot (\neg\texttt{PPIN}_i + \texttt{ALTERED}_i). \tag{3.6.5}$$

After the insertions, the computations that are anticipated at the points of insertions will be made redundant, and can be deleted. A computation at block n can be deleted if it is $\texttt{PPIN}_n$, since this implies that there have been some insertions at the ancestors of n which are available at n. The local attribute ANTLOC indicates whether the computation occurs in a basic block. Thus, the condition for deletions, designated by the term DELETE, can be computed as follows:

$$\texttt{DELETE}_i = \texttt{ANTLOC}_i \cdot \texttt{PPIN}_i. \tag{3.6.6}$$

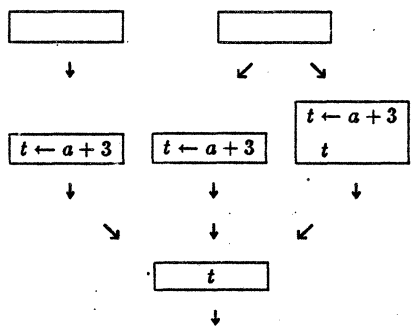
This deletion includes the case of redundant computations, when PPIN is true but no insertion is needed.

We can make an additional refinement to the above solution of PPOUT and PPIN. The application of the above partial redundancy elimination algorithm has the effect of moving computations upwards (or backwards) in the control flow graph so that some computations are computed earlier. Sometimes, this movement is a *code hoisting* optimization, but at other times, the same computation is unnecessarily duplicated. In all cases, the live ranges of expressions are increased (Fig. 3.6.3). Lengthened live ranges are undesirable because the variables in their extended points of occurrences may interfere with other code movements in later global optimization phases. Long live ranges also use up more register resources if allocated in registers. To limit the live ranges, insertions are desirable only at blocks at which the expression is originally partially available. It is possible to limit the expansion of live ranges by introducing the term



(a) Before optimization

(b) Large live ranges



(c) Smaller live ranges

Fig. 3.6.3 Effects on Live Ranges in Partial Redundancy Suppression

PAVIN in the solution for PPIN and PPOUT, without restricting the optimizations performed:

$$\texttt{PPIN}_i = \begin{cases} \texttt{FALSE} & \text{if } i \text{ is the entry block;} \\ \texttt{ANTIN}_i \cdot \texttt{PAVIN}_i \cdot \prod_{j \in \text{Pred}(i)} (\texttt{PPOUT}_j + \texttt{AVOUT}_j) & \\ \quad \cdot (\texttt{ANTLOC}_i + \neg\texttt{ALTERED}_i \cdot \texttt{PPOUT}_i) & \text{otherwise.} \end{cases} \tag{3.6.7}$$

$$\texttt{PPOUT}_i = \begin{cases} \texttt{FALSE} & \text{if } i \text{ is the exit block;} \\ \prod_{k \in \text{Succ}(i)} \texttt{PPIN}_k & \text{otherwise.} \end{cases}$$

Eq. (3.6.7), Eq. (3.6.5) and Eq. (3.6.6) are the actual data flow equations implemented in UOPT.

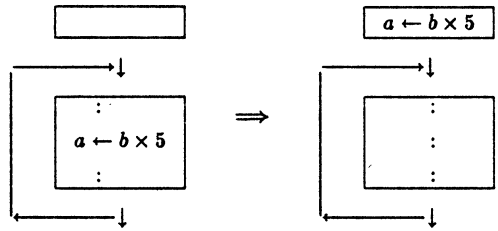


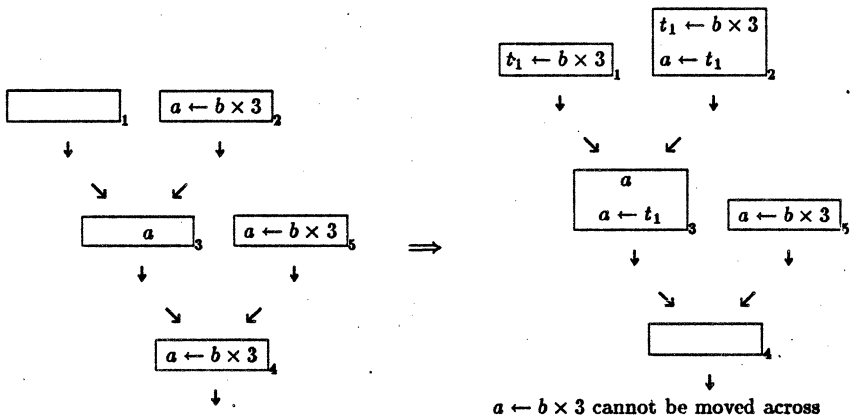
Fig. 3.6.4 Code Motion of Loop-invariant Assignment

3.6.2. Implementation Notes

The above optimization of partially redundant computations not only applies to expressions, but to assignments as well, by treating assignment as an operator (Fig. 3.6.4). To ensure the recognition of all redundancies and that all movements of assignments are legal, it is necessary that the global attributes are solved using the appropriate local attributes for assignments. The forward attributes AVIN, AVOUT, PAVIN and PAVOUT do not imply any code movement, so they can be solved using the PAVLOC and ABSALTERED local attributes for assignments. The backward attributes ANTIN and ANTOUT imply backward movements. It is incorrect to move an assignment across a block in which the assigned variable is used, since this changes the effective value of the variable at the time it is referenced. Thus, the ANTIN and ANTOUT attributes must be solved using the ANTLOC and ALTERED local attributes for assignments (Fig. 3.6.5).

Expressions are optimized individually, independent of any potential nesting. Each operator constitutes a computational item whose code motion is to be solved. In the case of nested expressions, some further attention is warranted. When $\texttt{DELETE}_i$ is true for an expression in basic block i, it must also be true for all its subexpressions, and only the value of the outermost expression needs to be saved in its prior computations. Thus, any deleted subexpression nested within another deleted expression must be flagged to indicate that its value is not needed in that basic block. The bit vector $\texttt{SUBDELETE}_i$ gives such expressions. It can be computed by checking whether a deleted expression occurs only as part of a larger deleted expression.

On the other hand, when INSERT is true for an expression, it may be false for some of its subexpressions. In such cases, their values are available at that point, and do not need to be recomputed at the point of insertion. Such expressions also have to be flagged to indicate that the values in their prior computations need to be saved up to that point. The bit vector SUBINSERT gives those expressions in a basic block which are not inserted but are part of inserted expressions (Fig. 3.6.6).



$a \leftarrow b \times 3$ cannot be moved across block 3 because of the use of a in that block, but $b \times 3$ can.

Fig. 3.6.5 Partial Redundancies in Assignments

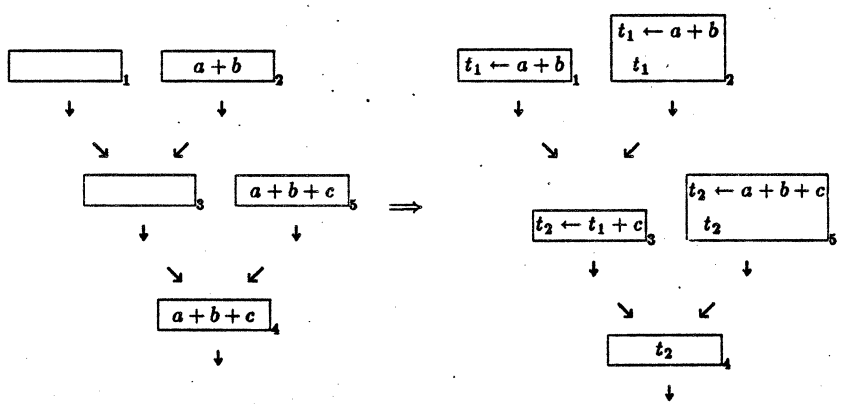


Fig. 3.6.6 Nested Partial Redundancies

After the partial redundancy optimizations, points of performing computations are changed. At a point of insertion, the inserted expression is computed and saved. At a point of deletion, the reference of the saved value of an earlier computation is made. Section 3.10 presents details

about determining the flow of saved computation results.

3.6.3. Observations

One elegant point about generalizing code motion to partial redundancy suppression is that additional cases of code motion out of loops are covered which would not otherwise be recognized in conventional code motion in which only loop-invariant computations are moved out of loops. Fig. 3.6.7 illustrates a case in which a computation is not loop-invariant because of a function call inside the loop. But because the computation is performed a second time in the loop after the function call, the first computation in the loop can be moved outside to the loop header.

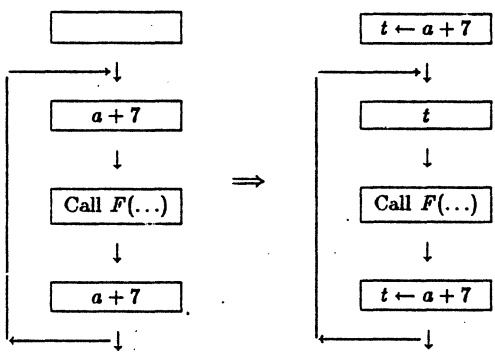


Fig. 3.6.7 Code motion of first occurrence of loop-variant $a + 7$ out of loop (t is temporary)

Although the term $\texttt{PAVIN}_i$ is introduced in Eq. (3.6.7) to prevent the unnecessary expansion of live ranges, not all useless code movement can be prevented. This over-movement can occur when the term $\texttt{PAVIN}_i$ is true due to the presence of a larger enclosing loop. Another situation occurs in the case of the `WHILE` loop, in which the loop termination conditional expression is unnecessarily moved and duplicated (Fig. 3.6.8). Appendix D contains notes on how the `WHILE` loop can be compiled by the front-end to allow for code motion, which also prevents this over-movement of the conditional expression.

A final point is that the copy propagation algorithm mentioned in Section 3.6 can enable more loop invariant computations to be detected in code motion optimization without additional effort. For example, if the statements $a \leftarrow b + c$ followed by $a + d$ occur inside the same loop, and b, c and d are loop invariant, then copy propagation will convert $a + d$ to $(b + c) + d$ which can then be recognized as loop invariant and moved out of the loop.

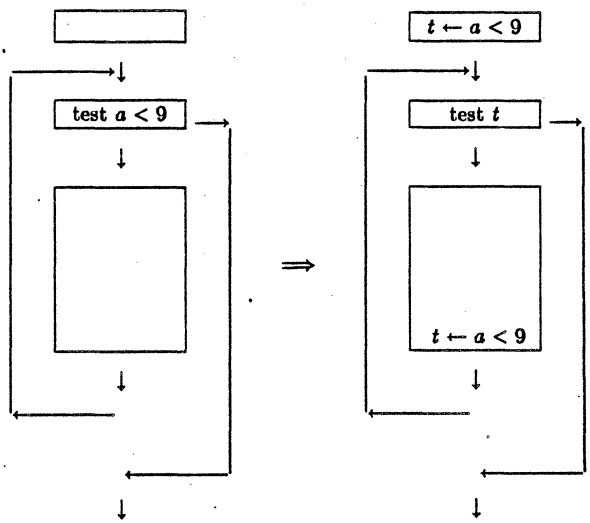


Fig. 3.6.8 Over-movement of the conditional expression in a WHILE loop

3.7. Reduction of Operator Strength

The purpose of the strength reduction optimization transformation is to replace complex operations by simpler ones. It is primarily associated with quantities that are linear functions of induction variables in loops. The process involves replacing multiplications between induction variables and constants (including region constants) by simple increments [Cock77] [Alle81]. Opportunities for strength reduction arise most often in subscripted array references. In multi-dimensional arrays, multiplications by constants are always necessary to compute offsets. Strength reduction optimization is especially important in machines with index registers, and fast instructions that increment or decrement these index registers.

Although strength reduction and code motion are different types of optimization problems, they are similar in a certain perspective, as illustrated in Fig. 3.7.1. The reduction candidate $i \times 3$ is to be replaced by a temporary t, which is to be properly initialized to 3 before loop entry, and properly incremented by 3 each time i is incremented by 1. It is possible to regard the whole process as movement of the induction expression $i \times 3$ to outside the loop. Although $i \times 3$ is not a loop constant expression, it is expensive to compute inside a loop. It is instead computed outside the loop as $i \times 3$, which is constant folded to $1 \times 3 = 3$, and stored in the temporary t. Because i is not a loop constant, but is an induction variable in the loop, t is

updated every time i is incremented inside the loop. Code motion is a special case because there is no induction expression to update each time through the loop.

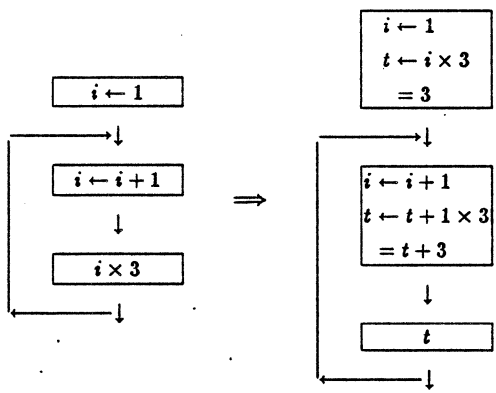


Fig. 3.7.1 Strength Reduction as Code Motion

This generalization can be further applied to more general strength reduction transformations involving products of induction variables. In Fig. 3.7.2(a), where a and b are region constants, applying the above process to the reduction candidate $i \times j$ transforms to Fig. 3.7.2(b). By targeting the newly formed $i \times b$ and $j \times a$ as reduction candidates, Fig. 3.7.2(b) is reduced to Fig. 3.7.2(c), which contains no more reduction candidate, although an additional pass is needed to move the loop-invariant expression $a \times b$ outside the loop by straight code motion.

Since code motion can be viewed as a special case of suppressing partial redundancy, as discussed in the Section 3.6, strength reduction can also be generalized in this respect and be included under the category of optimizations associated with partial redundancies. As a result of such a generalization, strength reduction is no longer limited to loops, but is possible in acyclic regions of flow graphs as well. Fig. 3.7.3 illustrates such a situation as compared with straight common subexpression. The reduction candidate $i \times 3$ can be regarded as a common subexpression, although there is an increment of i in between the two occurrences. In the optimization, the second multiplication is replaced by an increment of the temporary t. Fig. 3.8.2, in the next section, shows a case of combined strength reduction and partial redundancy.

The method of partial redundancy suppression of the Section 3.6 has the important characteristic that the code movements of all computations in the procedure are determined once, by the solution of the bit vectors INSERT and DELETE. The lengths of the bit vectors depend on

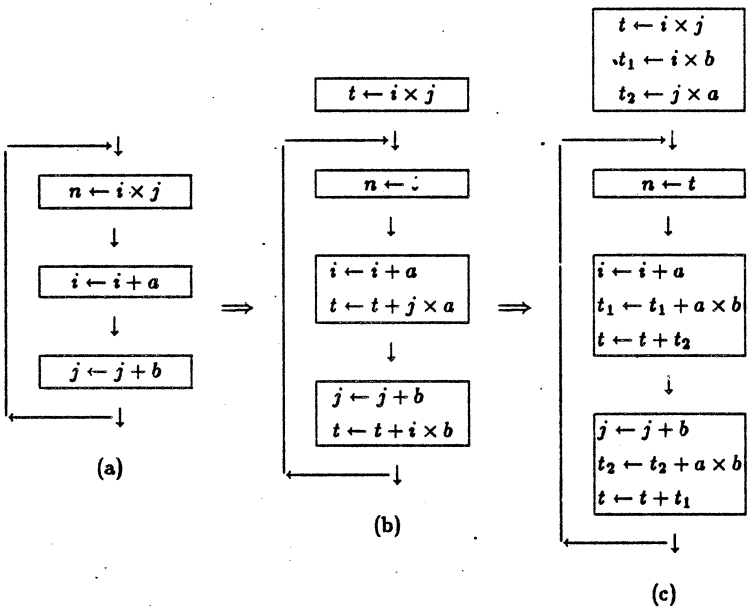


Fig. 3.7.2 Iterative Strength Reduction

the number of different computations in the procedure to be included in the optimization. Increasing the lengths of the bit vectors will increase the optimization time only marginally, since in the iterative solution of the data flow equations, the number of iterations is usually small, and depends more on the form of the control flow graph than on the contents of the bit vectors [Knut71]. Thus, by including strength reduction in the suppression of partial redundancies, we essentially get an additional optimization performed for free.

Before using the algorithm of Section 3.6 to perform strength reduction, it is necessary to determine the set of induction variables IV and strength reduction candidates CAND. In the current implementation, induction variables are limited to variables incremented by constant terms. As is the case in code motion, no analysis of program loop structure is needed. IV and CAND are local properties, and their determinations are limited to individual basic blocks. They are identified as follows:

IV - (Induction Variable) A variable v is IV in basic block i if it is defined in block i only

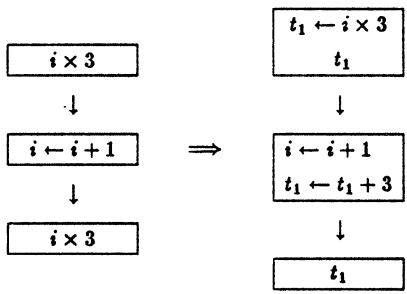


Fig. 3.7.3 Strength Reduction in Straight-line Code

by instructions of the form $v \leftarrow \langle \text{expr} \rangle$ where the expression $\langle \text{expr} \rangle$ consists only of the $+$ and $-$ operators, constants and the variable v itself which must occur at least once in $\langle \text{expr} \rangle$.

Candidacy for strength reduction is recursively defined. The expression itself does not have to occur in a basic block for it to be a strength reduction candidate. This is because in the subsequent transformation, it may be necessary to move the expression across the basic block, and this recognition is necessary to enable the code motion.

CAND - (Strength Reduction Candidates) An expression is CAND in basic block i if it is one of the following operations and satisfies the corresponding conditions:

(a) $+,-$: one of its operands is CAND_i and the other operand is either CAND_i or is invariant in block i.

(b) $\times$: one of its operands is a constant or region constant and the other operand is either CAND_i or is an expression consisting only of the $+$ and $-$ operators combining variables at least one of which is IV_i and the rest are either IV_i or are invariant in block i.

According to the above construction of CAND, the following are examples of induction expressions being recognized:

$i_1 \times k_1$
$i_1 \times a$
$(i_1 \times a + i_2 + k_1) \times b$
$(i_1 \times a + i_2 + k_1) \times b + c$
$(i_1 \times a + i_2 + k_1) \times b + k_2 \times i_3$

where i_1, i_2, etc. are induction variables,

k_1, k_2, etc. are constants and
a, b, etc. are region constants.

Note that expressions of the form $i_1 + k_1$ are excluded because they do not contain any complex operation to be simplified.

Strength reduction optimization is incorporated into the algorithm of the Section 3.6 by adjusting the local attributes using CAND. The result of the flow analysis will then automatically reflects the code motion of the strength reduction candidates. The local attributes are adjusted as follows:

$$\begin{aligned} \texttt{ALTERED}_i &= \texttt{ALTERED}_i - \texttt{CAND}_i \\ \texttt{ANTLOC}_i &= \texttt{ANTLOC}_i + \texttt{EXPOCCUR}_i \cdot \texttt{CAND}_i \\ \texttt{AVLOC}_i &= \texttt{AVLOC}_i + \texttt{EXPOCCUR}_i \cdot \texttt{CAND}_i \end{aligned} \tag{3.7.1}$$

In the above, the attribute EXPOCCUR gives whether an expression occurs in a basic block. The meaning of the first redefinition is that if an expression is a strength reduction candidate in block i, then block i should be made transparent to the expression so that the expression can move across the block. The second and third redefinitions say that if the expression occurs in the block, then it should be regarded as being locally anticipated and locally available.

The subsequent solutions for INSERT and DELETE will then determine the movements of the reduction candidates exactly as they do for partially redundant expressions. In the final code emission phase, in regions in which the reduction candidates are available and live, any increment or decrement of the induction variables will cause generation of the corresponding code to update the temporaries that contain the values of the induction expressions.

3.8. Induction Variable Elimination

After the strength reduction optimization of Section 3.7, additional opportunities for a different optimization are unfolded. If an induction variable is used only in strength reduction candidates that have been moved upward, and the variable is not live or will be assigned a new value, the variable can be eliminated in its loop-induction region. This means that the initialization and updates of the induction variable can be suppressed. Most often, the induction variable appears in the test for loop termination condition. In this case, linear function test replacement can be performed, which involves substituting the induction variable in the test by its induction expression. Such an operation further enhances the chance that the induction variable can be eliminated (Fig. 3.8.1). The algorithm we use, which relies on information gathered during code motion optimization, is applicable not only to strongly connected components of the flow graph,

but to all regions of the code. In addition, we do not limit test replacement to loop termination tests, but to any comparison operation which may be part of a boolean expression that can exist in any region of the program. Section 3.8.1 presents the linear function test replacement algorithm. Section 3.8.2 discusses the operations to eliminate induction variables.

3.8.1. Linear Function Test Replacement

Linear function test replacement is performed only for the purpose of enhancing the elimination of induction variables. If it does not result in making the replaced variable dead, then the test replacement should not be performed. The algorithm for linear function test replacement in UOPT finds and marks possible test replacement candidates. Subsequent to this, induction variable elimination is performed. This in turn results in establishing which test replacements are beneficial and which are not. A final pass over the test replacement candidates suppresses all those test replacements that are not desirable.

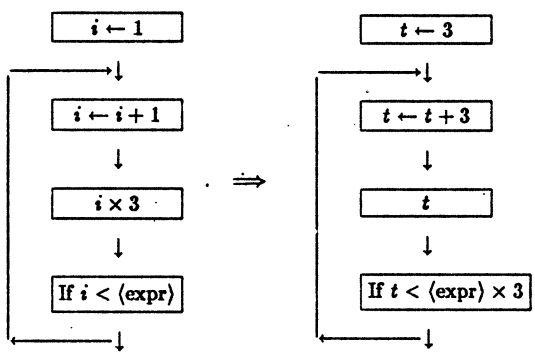


Fig. 3.8.1 Linear Function Test Replacement

The linear function test replacement algorithm is as follows:

Algorithm *TestReplace.*

For each comparison operation which occurs in block n in the program of the form i op k_1 where k_1 is a constant, if i is IV_n or is not ALTERED_n, then i can potentially be replaced by its induction expression. (The ALTERED attribute is the one that has been modified by Eq. (3.7.1).) Find an expression e in the program that satisfies the following condition:

1. e is an induction expression (see definition of CAND in Section 3.7);
2. e contains i as the only variable operand;

3. e is $\texttt{PPIN}_n$.

If the expression e can be found, then mark i as being replaceable by e. □

The purpose of condition 2 and the requirement that the test operation must be of the form i op k_1 is for ensuring that an equivalent test of the form e op k_2 can be obtained by transformation after the test replacement, where k_2 is formed by some constant arithmetic. If the form e op k_2 cannot be obtained, the transformation will slow down the program since the left or right sides of the comparison then contain additional computations.

Condition 3 makes sure that e is available at the point of replacement so that it does not have to be recomputed. The use of the PPIN attribute is more general than the AVIN attribute that applies before the code motion transformation. This is established by the following theorem:

THEOREM 3.8.1. *If a computation e is* PPIN *at block i, computed by Eq. (3.6.7), then it is available at the entry of block i after the insertions performed according to Eq. (3.6.5).*

PROOF. By Eq. (3.6.7), for $\texttt{PPIN}_i$ to be true, $\texttt{PPOUT}_j$ or $\texttt{AVOUT}_j$ must be true for all $j \in \text{Pred}(i)$. According to Eq. (3.6.5), one of the following cases must occur at block j:

(a) e is inserted at the exit of j ($\texttt{INSERT}_j$ = true);

(b) e is available at the exit of j ($\texttt{AVOUT}_j$ = true);

(c) e is $\texttt{PPIN}_j$ and not $\texttt{ALTERED}_j$ $((\neg\texttt{PPIN}_j + \texttt{ALTERED}_j) = \text{false})$.

In cases (a) and (b), e will be available at the exit of j. In case (c), the problem is reduced to finding whether the theorem is true for block j. We can apply the same reasoning to block j, and this process will eventually terminate since PPIN at the entry of the flow graph is false. The only situation where reasoning through case (c) will not terminate is when there is a cycle in which PPIN is true for all the nodes and e is not ALTERED in the cycle. But in this case, the fact that e is available at the exits of the headers to the cycle is sufficient to guarantee that e is available throughout the cycle. □

The above test replacement algorithm does not specifically require that the replaced variable i be an induction variable. One reason is that we do not recognize induction variables on a global basis. The induction variable attribute IV that we use is only a local attribute. A loop may contain more than one basic block, and a variable is an induction variable if it is IV in just one of the basic blocks. Also, the substituted expression e, although involved in code motion, may not have been a strength reduction candidate. But even under such situations, the test replacement performed is still an optimization. Thus, our approach to linear function test replacement is more general than the traditional approach.

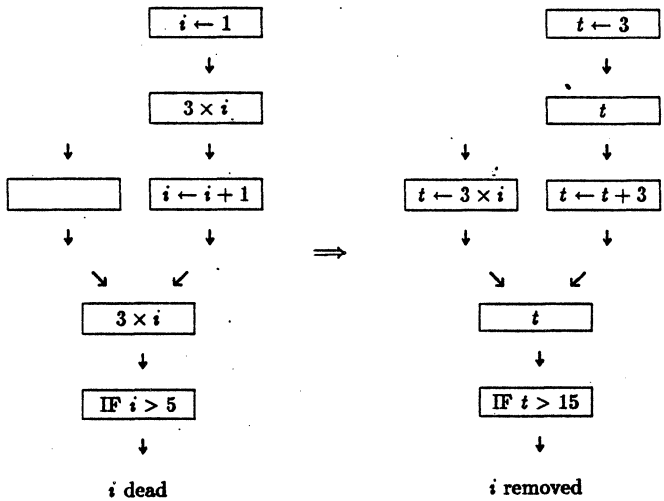


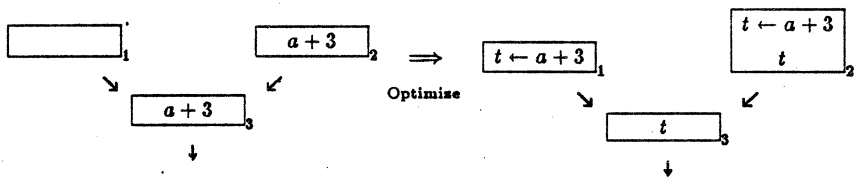
Fig. 3.8.2 Combined Strength Reduction and Partial Redundancy

3.8.2. Finding and Eliminating Redundant Induction Variables

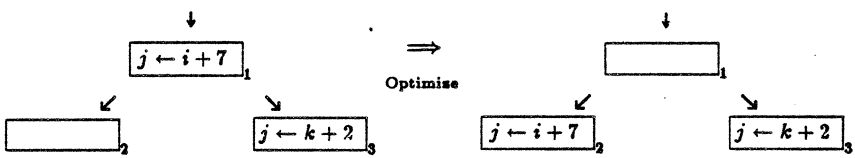
After the uses of the induction variables have been replaced, the elimination of these variables is actually equivalent to eliminating assignments to these variables which have now become redundant. These assignments consist only of increments to the induction variables. The same basic scheme of Section 3.5 can be used, which determines store redundancies by solving for anticipabilities of L-values. A different treatment is needed for induction variables, however. If a variable is an induction variable in a basic block, then its use in its increment statements must not be regarded as altering its L-value, in the definition of ALTERED of Section 3.5. The meaning of this is that all increments to induction variables are to be regarded as transparent. Thus, in a basic block in which an induction variable is only incremented, ANTLOC and ALTERED are both false. The earlier code motion and test replacement optimization also affect these attributes, and updating them is also needed.

After the computation of ANTIN and ANTOUT according to Eq. (3.3.2), an induction variable is redundant if its L-value is ANTOUT and not ALTERED in a basic block. In this case, all of its increments in that basic block are to be deleted.

Following the elimination of redundant induction variables, the test replacements performed by algorithm *TestReplace* have to be validated. This consists of checking, for each replaced variable i, whether ANTOUT_n just computed is true. If this is false, then variable i has not been



(a) Expression $a + 3$



(b) Assignment $j \leftarrow$

Fig. 3.9.1 Duality in Partial Redundancies between Expressions and Assignments

eliminated, and the test replacment for i is cancelled.

3.9. Optimization of Store Positions

The optimization of Section 3.5 involves only assignments that are completely redundant. As was noted in Section 3.5, a duality exists between redundant expressions and redundant assignments. The same is true when we generalize to partial redundancies. Fig. 3.9.1 illustrates this. Partial redundancy in expressions is a partial availability problem, and partial redundancy in stores is a partial anticipability problem. As partial redundancy in expressions can be removed by backward code motion, partial redundancy in stores can be removed by forward code motion. Fig. 3.9.2 shows a partial redundancy in stores occurring in a loop. The variable a is not referenced anywhere inside the loop. The resulting code motion moves the store to the exit of the loop, rather than the entry as is the case with expressions†.

Since partial redundancy in stores corresponds exactly to partial redundancy in expressions, provided that we reverse the direction of view from backward to forward (or from upward to downward), we can apply the same method of partial redundancy suppression to stores. The consequence is a scheme to optimize assignments that encompasses a greater scope, involving deletions from their original positions and forward movements to places where they are inserted.

† If the assigned value is loop-invariant, then the assignment will be treated as a loop-invariant computation and moved to the entry of the loop (see Section 3.6).

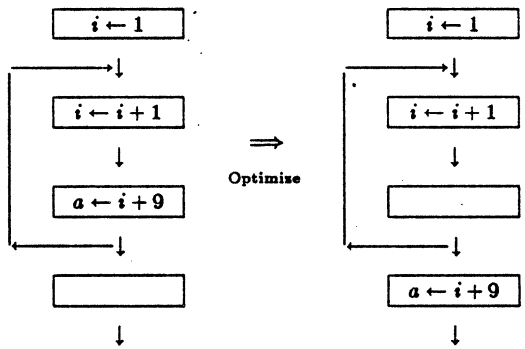


Fig. 3.9.2 Store Redundancy in Loop

Since the movements of the stores are only in the forward direction, an additional but important benefit that can be brought about is live range shrinkage.

The same methodology as in Section 3.6 is used in UOPT to suppress partial redundancies in stores. Instead of inserting at the exits of individual basic blocks, we now insert the stores at the entries. The L-value attributes of Section 3.5 are used as the starting local attributes in the flow analysis. The directions of all the parameters and attributes are reversed: OUT ⇔ IN, ANT ⇔ AV and Pred ⇔ Succ. The system of flow equations to solve for PPIN and PPOUT for stores, which correspond to Eq. (3.6.7), is as follows:

$$\texttt{PPOUT}_i = \begin{cases} \texttt{FALSE} & \text{if } i \text{ is the exit block;} \\ \texttt{AVOUT}_i \cdot \texttt{PANTOUT}_i \cdot \prod_{j \in \text{Succ}(i)} (\texttt{PPIN}_j + \texttt{ANTIN}_j) & \\ \quad \cdot(\texttt{AVLOC}_i + \neg\texttt{ALTERED}_i \cdot \texttt{PPIN}_i) & \text{otherwise.} \end{cases} \tag{3.9.1}$$

$$\texttt{PPIN}_i = \begin{cases} \texttt{FALSE} & \text{if } i \text{ is the entry block;} \\ \prod_{k \in \text{Pred}(i)} \texttt{PPOUT}_k & \text{otherwise.} \end{cases}$$

Using the resulting PPIN and PPOUT attributes, insertions and deletions of stores are determined by computing the attributes INSERTIN and DELETE. In this case, INSERTIN indicates insertion at the entry to a basic block rather than exit as was the case in backward code motion.

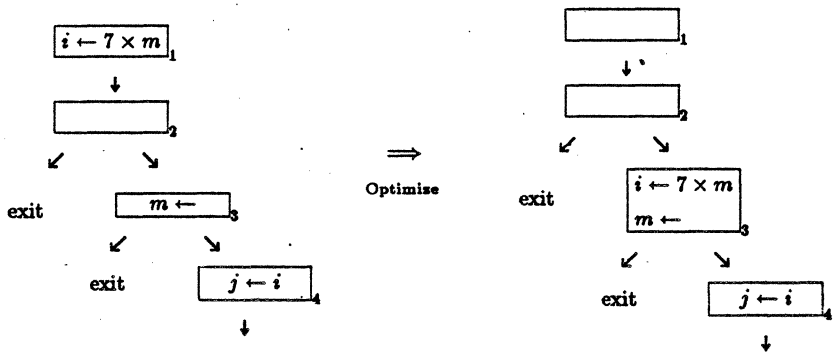


Fig. 3.9.3 Forward Code Movement to Eliminate Partial Redundancy in Store to i

$$\begin{aligned} \texttt{INSERTIN}_i &= \texttt{PPIN}_i \cdot \neg\texttt{ANTIN}_i \cdot (\neg\texttt{PPOUT}_i + \texttt{ALTERED}_i). \\ \texttt{DELETE}_i &= \texttt{AVLOC}_i \cdot \texttt{PPOUT}_i. \end{aligned} \tag{3.9.2}$$

As was remarked in Section 3.5, in solving for ANTIN, ANTOUT, PANTIN and PANTOUT, the initial values of ANTOUT and PANTOUT can be set to true if the variable is local, and false otherwise. This allows the recognition of paths in the program in which variables are dead. The result is that in the subsequent forward code movement, on reaching the entry to a path on which the assigned variable is dead, code insertion will be automatically inhibited (Fig. 3.9.3).

In the current optimization of store redundancies, no account is taken of the right-hand-sides of assignments. For an assignment $a \leftarrow \langle \text{expr} \rangle$, the content of $\langle \text{expr} \rangle$ does not affect the data flow analysis that results in computation of INSERTIN and DELETE. However, if a is assigned different values on different paths that converge, then the assignments to a cannot be moved to the point where the paths converge (Fig. 3.9.4). To take this into account, it is necessary to impose additional restrictions in the solution for PPIN and PPOUT in Eq. (3.9.1). In the initialization to solve for PPIN and PPOUT iteratively, the PPIN's for nodes which are confluences of more than one paths are to be set to false. In this way, these PPIN's will remain false throughout the iterations. The result is that the stores will not be moved across these nodes.

The attribute INSERTIN computed by Eq. (3.9.2) gives the stores to be inserted at the entry to a basic block by referring to the assigned variables, but gives no details about the assigned

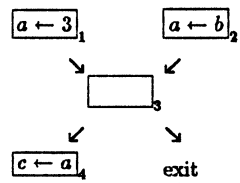


Fig. 3.9.4 Partial Redundancy in $a \leftarrow$ cannot be eliminated (a local variable)

expressions to be used. This is because all assignments of the form $a \leftarrow \langle \text{expr} \rangle$ are aggregately referred to as occurrences of the L-value of a. In performing the forward code motion specified by the INSERTIN attribute, it is necessary to determine the actual assigned expressions. Since the insertions are moved from the ancestral nodes in the flow graph, it is only necessary to search through the predecessors by taking an upward path starting with the immediate parent. Because of the restriction that stores cannot be moved from different paths that converge, a block in which a store insertion is indicated will not have more than one parent. The search must succeed, and the assignments found are deleted at their original basic blocks.

The content of the right-hand-side expression also affects the feasibility of the forward code movement in another way. It is possible that the value of the assigned expression $\langle \text{expr} \rangle$ in the assignment $a \leftarrow \langle \text{expr} \rangle$ is altered somewhere along the path that leads to the node where the store is inserted (Fig. 3.9.3). In such situations, the assignment should still be moved forward as far as possible, because even though the store partial redundancy cannot be fully suppressed, it can still be confined to the smallest region possible. The resulting insertion is at the entry to the node where the assigned expression $\langle \text{expr} \rangle$ is first altered.

It is to be noted that the optimization of Section 3.6 also removes partial redundancies in assignments, but in a different sense: the assignments are regarded as computations and are moved backwards in the flow graph instead of forward. (Compare Fig. 3.9.2 with Fig. 3.6.4.) The right-hand-sides of assignments are included in the data flow analysis, and the assignments $a \leftarrow \langle \text{expr}_1 \rangle$ and $a \leftarrow \langle \text{expr}_2 \rangle$ are regarded as different computations. There is no overlap between the current optimization and those performed in Section 3.6.

3.10. Global Optimization of Saves

The optimization of partial redundancy suppression for expressions (Section 3.6) requires that the values of expressions be saved at their points of computation and be made available for use later on at various points in the program. The saving of computed expression values constitutes a major portion of the new code introduced by the optimizer to the optimized

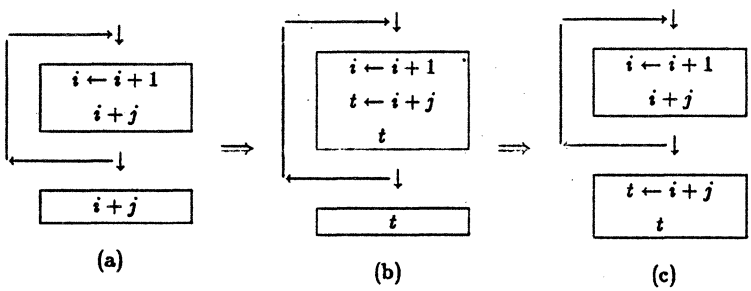


Fig. 3.10.1 Suppression of Undesired Common Subexpression Optimization

program. The optimizer has to make sure that these additional saves are optimally placed so that they do not cause deterioration in program performance. Fig. 3.10.1(b) shows a common subexpression optimization which actually results in slowing program execution; to avoid the one recomputation of $i+j$ outside the loop, the common subexpression is stored into the temporary t multiple times during the iterations of the loop. The optimization algorithm of Section 3.6 and many other redundancy elimination algorithms do not recognize such cases and do not avoid the optimization. This is because a computation is redundant whenever it occurs at a point where it is available. This availability condition applies even if the previous computation occurs inside a loop.

To enable the optimizer to avoid such undesirable optimization of redundant expressions, we address this problem in terms of the optimization of positions to save common subexpressions (Fig. 3.10.1(c)). The saving of expression values takes up execution time, and it is necessary to eliminate any redundancy in the save code. Since this redundancy in saves is of the same nature as the redundancy in stores discussed in Section 3.9 (both are memory store operations), this problem can be tackled using the same approach and with the same algorithm. Moreover, they can be performed at the same time, thus allowing us to obtain the effects of the optimization of temporary saves essentially for free.

3.10.1. Determination of Saved Computations

To apply the algorithm of Section 3.9 to the suppression of redundancies in temporary saves, some preliminary steps are needed after the code motion transformation of Section 3.6. It is necessary to look at all the places where computed values are saved and referenced across basic block boundaries. Then it will be possible to establish the local attributes for the temporary

saves with which we do flow analysis to suppress their partial redundancies. We call a node in which a computation is saved a *source* and a node in which a previously saved computation is referenced a *sink*. The reason for these names is because computations done at the sources are available and used at the sinks by virtue of the control flows. Our objective is to establish the bit vectors SOURCE and SINK for all basic blocks. If the bit position for an expression e in $\texttt{SOURCE}_n$ is true, then the expression must have been computed in block n, and the value of the last computation† in n is to be saved. If the bit position for e in $\texttt{SINK}_n$ is true, then a previously saved value of the computation of e is referenced in block n. SOURCE refers to the definitions of the temporaries and SINK refers to their references.

The bit vectors SOURCE and SINK can be computed by pure bit vector operations on attributes which are used in the previous optimizations. A computation e is saved in block n in one of the following two occasions:

1. The computation e occurs in the basic block n and is available at the block exit ($\texttt{AVLOC}_n$ is true). It is not redundant at the entry point of n (i.e. $\texttt{DELETE}_n$ is false) or it is altered earlier in block n ($\texttt{ALTERED}_n$ is true) so that its recomputation in n is needed.

2. The expression e has been inserted at the exit of basic block n in the code motion of Section 3.6 ($\texttt{INSERT}_n$ is true).

In both of the above cases, it is necessary that there is some partially anticipated sink, so that the computed value needs to be saved.

A previously saved computation e is referenced in block n under the following situations:

1. The expression e has a redundant occurrence in block n, and in this occurrence, it is not part of another redundant expression ($\texttt{DELETE}_n - \texttt{SUBDELETE}_n = \text{true}$).

2. The expression e is a subexpression of a larger expression inserted at the exit of block n in the earlier code motion, but e does not need to be inserted there because it is available at that point ($\texttt{SUBINSERT}_n - \texttt{AVLOC}_n = \text{true}$).

From the above, the bit vector SINK can be computed as follows:

$$\texttt{SINK}_i = (\texttt{DELETE}_i - \texttt{SUBDELETE}_i) + (\texttt{SUBINSERT}_i - \texttt{AVLOC}_i) \tag{3.10.1}$$

From the local attribute SINK, we can solve for its global partial anticipability by flow analysis. The resulting SINKPANTOUT bit vector is used in computing SOURCE:

$$\texttt{SOURCE}_i = [\texttt{AVLOC}_i \cdot (\neg\texttt{DELETE}_i + \texttt{ALTERS}_i) + \texttt{INSERT}_i] \cdot \texttt{SINKPANTOUT}_i \tag{3.10.2}$$

† There can be more than one computation of e in block n when all except the last are altered inside n.

3.10.2. Optimization of Saves by Flow Analysis

After the computation of the SOURCE and SINK attributes, we can transform them into the corresponding attributes which we use in the suppression of store partial redundancies in Section 3.9. We can then include temporary saves in the forward code motion algorithm. The transformation can be specified as follows:

$$\texttt{SOURCE}_i \Longrightarrow \texttt{AVLOC}_i$$

$$\texttt{SINK}_i \Longrightarrow \texttt{ALTERED}_i$$

$$\texttt{SOURCE}_i - \texttt{SINK}_i \Longrightarrow \texttt{ANTLOC}_i$$

The above transformation allows us to obtain the AVLOC, ANTLOC and ALTERED as defined in Section 3.5 applied to the temporaries that store the values of the expressions.

The iterations employed in Section 3.9 are used to solve for the basic blocks at the entries of which the saves to temporaries are to be inserted. At these points of insertion, the recomputation of the saved expressions are needed. At places where there are redundant stores to temporaries, the stores are inhibited.

3.11. Summary

In this Chapter, we have presented a framework of performing optimization that is comprehensive enough to include all the common and important optimization transformations. In Section 3.1, we present a set of local optimization techniques, most of which involve manipulations of the underlying data structures, which are used in various phases in the subsequent global optimizations, and according to which data flow information is gathered. In Section 3.3, we define the data flow attributes that form the basis for performing the various global optimizations.

A concise and more generalized method for performing copy propagation is introduced in Section 3.4. The method also includes global constant propagation as a special case. The copy propagation algorithm relies on the subsequent redundant expression and redundant store eliminations for its full benefits to be derived.

In Section 3.5, a method to perform redundant store elimination is presented. The method is based on the determination of whether a store is anticipated, as opposed to whether a variable is not live in the traditional approach. The dual relationship between redundant expressions and redundant assignments is introduced.

In Section 3.6, the partial redundancy algorithm to perform code motion and common subexpressions is formulated and a scheme for its usage is presented. In Section 3.7, we present a new method of performing strength reduction by regarding it as a generalization of code motion, thus enabling it to be performed at the same time as code motion in the partial redundancy algorithm.

In Section 3.8, we give a method to perform linear function test replacement. The method of Section 3.5 is adapted for use in the elimination of induction variables made redundant by previous optimizations.

In Section 3.9, the concept of partial redundancy in stores is derived using the duality first exposed in Section 3.5, and we propose the optimization of forward code motion as opposed to the standard backward code motion. The algorithm of Section 3.6 is modified to perform partial redundancy elimination in stores. This same algorithm is then re-applied to the optimization of temporary saves in Section 3.10. This completes the presentation of the sequence of optimization techniques that we use.

4. Register Allocation

Machines have different forms of memory organization and storage hierarchy. The memory storage elements that affect machine performance the most are the set of hardware registers — the fastest type of memory in most machines. Machine instruction sets are designed around the set of registers residing in the machines. Instructions involving registers are usually shorter and faster than those involving memory references. Therefore, efficient utilization of registers is very important in generating good object code.

Register management is a highly machine-dependent process. In many machines, specific operations are tied to specific registers. Many machine instructions limit one or more instruction operands to be among the hardware registers, since such a specification usually takes up a smaller number of bits in the instruction word. Index and base registers are commonly provided to access elements in arrays, or in indirect addressing. Many machines also offer the auto-increment and auto-decrement modes of addressing via index registers. Register management depends heavily on instruction selection at the lowest level of code generation, and is more appropriately done by the code-generating back-ends.

However, there is another aspect of register allocation which is less related to instruction selection, and can best be performed by the machine-independent optimizer so that the results can be used by all back-ends. This aspect of register allocation determines which quantities should reside in the limited number of registers during the course of execution of various program segments, and the optimization of the associated register-memory transfer operations. This global machine-independent register allocation, performed across entire procedures, is based on usage counts, and depends on the global control structure of the program and the availability of data flow information. Code generators usually gather only local information related to the instructions they are going to emit, and thus cannot be relied upon to perform this task in the global context. Global register allocation is best done in the global optimizer as the last phase, when the final structure of the code to be emitted has been determined. This chapter discusses the various aspects of machine-independent register allocation in UOPT.

4.1. Limitations

Register allocation at the intermediate code level has a number of limitations compared with register allocation done by the code generators. All of these limitations are due to the machine-independent nature of the intermediate code.

1. Only allocation of general-purpose registers is possible. Dedicated registers (e.g. stack pointers, displays, subroutine linkage registers) and registers restricted to specific operations

(e.g. multiplication in the Intel 8086) cannot be allocated, since these registers are invisible at the intermediate code level. Nevertheless, if the registers in the target machine are divided into classes, the optimizer can allocate variables of different data types to the different classes of registers according to the description given to it (e.g. the data registers and address registers in the MC68000, the general registers and floating-point registers in the IBM 360/370).

2. The requirements and effects of individual machine instructions pertaining to registers cannot be taken into account. Such uses of registers arising out of instruction selection by the code generators are not necessarily related to the register allocation decisions. When registers are globally allocated by the optimizer, intermixing of registers used by the optimizer and registers used by the code generator is not possible. Since the registers used by the code generator are not available to the optimizer, redundant register copies are sometimes introduced. For example, the optimizer cannot utilize the fact that an expression may already be residing in a register at the end of a sequence of machine instructions, unless it specifically tells the code generator to move the result there. Of course, no real move may be needed.

3. There are hidden register operations over which the optimizer has no control. For example, in U-Code, the computation stack is a storage area which is usually implemented using a set of registers in real machines. At a function call, it is necessary to save the items still exist on the computation stack — an operation that involves many register moves. At the intermediate code level, an item loaded on the stack is assumed to have been used even if it still resides on the stack. Since the home locations of the variables residing further down the stack may be changed by the call due to side effects, it is necessary to save the stack items in special temporary save areas. Another example is the passing of parameters in procedure calls. The actual mechanism may involve the use of registers, which is invisible at the intermediate code level.

4. The optimizer has to assume a fixed saving in execution cost for accessing a variable in register rather than from memory. This saving estimate, supplied to the optimizer in the machine description, is not in reality fixed for a given machine, since the execution times of individual machine instructions vary and are also dependent on the actual operand addressing modes used.

5. The optimizer employs usage counts of variables in the program to estimate the possible improvements when allocating variables in registers. The usage counts of variables in the intermediate code may differ from those in the object code, due to the availability of specialized instructions in the target machine. In most of these cases, a sequence of U-Code

instructions is collapsed into a single machine instruction. Examples include "increment and test", "increment pointer and load indirect".

The first two of these limitations are the most serious while the last three limitations are largely unavoidable and have minor impact. The unavailability of the detailed structure of the registers and the code sequence requirements introduce some inefficiency. However, we believe that such inefficiency is small and the more abstract model used in UOPT allows the same register allocation to be used across a wide variety of machines and code generators.

4.2. Assumptions and Overview

The purpose of register allocation in UOPT is to best utilize the limited number of general-purpose registers set aside for use by the optimizer in the code-generating back-ends. The register allocator should try to introduce as little register load and store code as possible. If the optimizer does not use up all the registers set aside for it, it conveys the information to the code generator so that the unused registers are available for use by the back-end. Since the input program is assumed executable without using the global optimizer, all program variables in the input are assumed to have been allocated in main memory. The optimizer does not attempt to change the stack frame composition or re-map variable addresses, since such transformations provide little improvement in execution speed. The optimizer also assumes no register allocation is present in the input program, since this interferes with its own register allocation. Temporaries generated by the previous phases of the optimizer are also assumed to have been allocated in main memory, and they are treated uniformly as variables. Due to these assumptions, it is not necessary to generate spill code for variables not allocated to registers. Instead, all objects have *home* memory locations and the optimizer attempts to re-map memory accesses to register accesses. This contrasts with the approach used in the PL.8 compiler project [Chai82] in which the register allocation phase attempts to map the unlimited number of symbolic registers assumed during earlier compilation and optimization phases into hardware registers. If this is unsuccessful, code is added to spill computations from registers to storage and later re-load them.

A precaution is taken due to *alias* and *equivalence*. Variables can be equivalenced to an array element. Non-local variables can also be altered or referenced by indirect assignments or loads. Such potentially aliased variables are not considered for assignment to registers since the indirect operations may alter or reference the home locations of these variables which have not been updated, resulting in incorrect program execution.

The general purpose registers used by UOPT are divided into classes, with each class being designated for specific data types and sizes. The division into classes is strict, and no overlap of

registers between the classes or more complex machine idiosyncrasy is currently handled. The registers within each class are assumed to be uniform.

The register allocation algorithm used is a combination of a local method based on usage counts and the global method based on the coloring algorithm, which also takes into account cost and saving estimates. The local phase allocates one block to a register each time. The global phase allocates one live range to a register each time. The local register allocation phase is inexpensive and near-optimal for straight-line code, but does little to contribute to the globally optimal solution. The global allocation phase is more computation-intensive and time-consuming. In our approach, the local allocation process is made to do as much allocation as possible so long as the allocation would not have any effect on the outcome of the global allocation phase. The algorithm is general enough to be applicable to all target machines.

The relative importance between the local and global phases can be varied by changing the maximum length of blocks allowed. The user can set the ZVREF option with a number, which imposes a limit on the maximum number of variable appearances allowed in a basic block. If this number is exceeded, the remaining code is made to belong to a new block. A default value for this option serves to guard against the presence of large blocks that can degrade the output of the register allocator. When blocks are small, the local phase will not be able to allocate as many items to registers based on its allocation criteria, and more work is left to the more expensive global phase. As the limit on block lengths becomes smaller and smaller, the overall allocation also approaches the optimal solution since registers can now be allocated across shorter segments to cater to any irregular clustering of accesses. The processing cost also increases correspondingly because of the larger number of blocks involved and the greater amount of work being performed by the global phase. Thus, the register allocation phase in UOPT has a large amount of built-in flexibility with respect to processing cost and quality of results. In practice, basic blocks are usually short, and most of the work is done by the global phase.

4.3. Cost and Saving Estimates

In determining the feasibility of assigning a variable to register, it is necessary to estimate the execution-time cost and saving due to the register assignments.

Assigning a variable to a register involves the loading of the variable from main memory to the assigned register prior to referencing the variable in a register in the subsequent code. If the value of the variable is changed in the intervening code where it resides in register, the home memory location of the variable has to be updated with the register content at the end of the code segment unless it is dead on exit. These extra move operations between registers

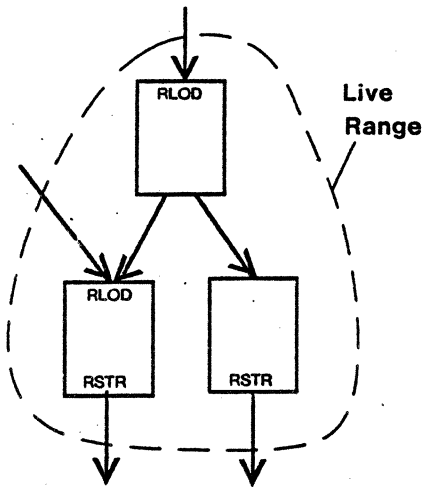


Fig. 4.3.1 Example of a live range with associated RLOD's and RSTR's

and memory represent the execution time cost of the register assignment. The execution time saving of the register assignment refers to how much the code segment is rendered faster due to the variable's residing in a register (Fig. 4.3.1). Thus, we define the following three parameters, which vary among target machines:

MOVCOST — The cost of a memory-to-register or register-to-memory move, which in practice is the execution time of the U-Code instructions RLOD and RSTR in the target machine.

LODSAVE — The amount of execution time saved for each reference of a variable residing in register compared with the corresponding memory reference that is replaced.

STRSAVE — The amount of execution time saved for each definition of a variable residing in register compared with the corresponding store to memory being replaced.

The parameters LODSAVE and STRSAVE may not be constant for all loads and stores for the same machine, since they depend on the actual machine instructions and addressing modes being used. For example, a machine instruction may directly specify an operand in main memory, or there may be loading of the operand into a register in a prior instruction before referencing the operand via the register. The addressing mechanisms used also depend on whether a given variable is local, global or an up-level reference. The actual addressing mechanisms may be via

displays or static links. Pipelining in the underlying architecture also affects the values. For machines that require operands to be in a register before any operation, LODSAVE is equal to MOVCOST. Otherwise, MOVCOST is larger than LODSAVE or STRSAVE. It is necessary to use average values for LODSAVE and STRSAVE for a given machine.

It is to be noted that LODSAVE and STRSAVE as defined above may not represent all the saving that comes from register assignments. The benefits of register allocation do not arise solely out of being able to reference an item in register instead of from memory. In many machines, having a register operand has the added benefit of allowing more freedom in the instruction selection process of the code generator. The saving that comes from enabling the code generator to use more efficient instructions is highly context-dependent, and cannot be easily paramaterized.

Only the relative values of the above three parameters are significant. A typical set of values for these parameters are 1.5 for MOVCOST and 1 for LODSAVE and REGSAVE. Section 5.7.2 discusses the effect of these parameters on the optimization results.

4.4. Local Register Allocation

Local register allocation in UOPT precedes the global register allocation phase. Local register allocation refers to allocation in a basic block, or a straight-line piece of code segment which may be part of a basic block. The allocation is based only on information available in each basic block. The solution to this problem using reference counts is well-established, inexpensive and can be easily implemented [Frei74]. Nevertheless, separate locally optimal solutions to the register allocation problem do not necessarily add up to the globally optimal solution. However, it is possible to determine a portion of register allocation locally that also belongs to the global solution, so that the work load of the subsequent, more expensive global allocation phase can be made smaller.

For each variable in the local code segment being considered, the local saving that can be achieved by assigning the variable to register is estimated. This is computed by:

$$\text{NETSAVE} = \text{LODSAVE} \times u + \text{STRSAVE} \times d - \text{MOVCOST} \times n \qquad (4.4.1)$$

where u is the number of uses of the variable,
d is the number of definitions of the variable and
n is either 0, 1 or 2.

n depends on whether a load of the variable to a register (RLOD) at the beginning of the code segment and a store from the register back to the variable's home location (RSTR) at the end of the code segment are to be inserted. If they are both needed, n is 2. If the first occurrence

of the variable is a store, then the initial RLOD is not needed. If the variable is not altered, or if the variable is not live at the end of the code segment, then the RSTR is not necessary.

If the local code segment is considered together with its preceding and subsequent code, the term involving MOVCOST represents the uncertainty in cost with regard to NETSAVE that may or may not contribute to the final global solution. This is because if the variable is also allocated to the same register in the surrounding code, then the RLOD and RSTR at the beginning and end of the current code segment are unnecessary, and the actual value of NETSAVE is increased. Thus, for each variable in the local code, we compute two separate quantities:

$$\text{MAXSAVE} = \text{LODSAVE} \times u + \text{STRSAVE} \times d \tag{4.4.2}$$

$$\text{MINSAVE} = \text{LODSAVE} \times u + \text{STRSAVE} \times d - \text{MOVCOST} \times n \tag{4.4.3}$$

The quantity MINSAVE represents the minimum saving in the local code segment gained by allocating the variable to register. The quantity MAXSAVE is the maximum possible saving. The actual saving after all register allocation is performed will range between MINSAVE and MAXSAVE. The parameters MAXSAVE and MINSAVE also apply to variables which do not occur in the code segment, when they are both 0; in such cases, the two parameters are used only in the later global allocation process.

When the surrounding blocks are considered together with the current block, the local allocation may displace some other variable which has been assigned to the same register in the adjacent blocks and which, if allowed to occupy the same register in the current block, would enable the elimination of the RSTR's at the ends of the preceding blocks and the RLOD's at the starts of the succeeding blocks. Thus, the absolute criterion for determining the local allocation of a variable in register can be given as:

$$\text{MINSAVE} > \text{MOVCOST} \times (p + s) \tag{4.4.4}$$

where p is the number of predecessors of the block,
s is the number of successors of the block.

When this condition is satisfied, the variable can be locally allocated in register with certainty regardless of the rest of the program. In computing the above condition, the frequency weights (see Section 4.5) of the adjacent blocks relative to the current code segment have to be taken into account.

In making local register allocations, if there are more variables satisfying the condition given by Eq. (4.4.3) than there are registers available, it is necessary to determine the priorities

among the variables. Priorities are assigned by imposing a partial ordering on the variables. Variable a is preferred over b if:

$$\texttt{MINSAVE}(a) > \texttt{MAXSAVE}(b) \tag{4.4.5}$$

Otherwise, the preference cannot be established absolutely.

The actual assignment of register number is not performed in the local allocation pass. It is done during node coloring in the global allocation phase, when the optimizer will look for opportunities to assign the same register to a variable over contiguous code segments to minimize the number of RLOD's and RSTR's.

4.5. Control and Data Flow Analysis

The overall register allocation process depends on the division of the input program flow graph into discrete code segments, each not longer than a basic block. A code segment is the smallest extent of program code over which a register is assigned to a variable. The smaller the code segments, the closer will the final solution be to the optimal allocation solution. However, the amount of processing time in global register allocation is potentially some exponential function of the number of program nodes. In UOPT, long basic blocks can be broken up into smaller segments based on the number of variable references already encountered. It is expected that as the limit on the sizes of the code segments becomes smaller and smaller, the usefulness of the local register allocation stage will diminish, since fewer and fewer variables can satisfy the condition given by Eq. (4.4.4).

The global register allocation solution also depends on estimates of the cost and saving of letting a variable reside in register across a certain region. In computing the cost and saving, it is necessary to take into account the loop structures of the program. This is because a register load or store outside a loop is preferred over one inside a loop, and a live range extending over a loop has greater priority to occupy a register than one not over a loop. Thus, each code segment is assigned a frequency weight poportional to how deep the segment is nested inside loops. The weight is arbitrarily increased by a factor of 10 each time a loop is entered. Thus, the frequency weight of a given code segment is 10 times its loop-nesting depth.

The loop structure of the program is detected by performing interval analysis on the control flow graph. The flow graph is partitioned into intervals, forming the *derived flow graph*. This process is performed iteratively until the *derived sequence* of the flow graph is obtained (Section 3.3 of [Hech77]). In the derived sequence $G_0, G_1, \ldots, G_k$, each G_{i+1} is the derived flow graph of G_i, and G_k is the limit flow graph. The degree of nesting of individual nodes in the original flow

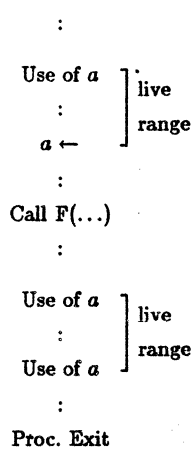


Fig. 4.5.1 The live ranges of a non-local variable a

graph is then found by going down the intervals starting with the limit flow graph G_k in the reverse order of the derived sequence until the nodes in the original flow graph G_0 are reached. In going from G_i to G_{i-1}, not more than one loop can be entered, and the loop must include an interval header in G_i.

A *live range* of a variable is an isolated and contiguous group of nodes in the control flow graph in which the variable is defined and referenced. No other definition of the variable reaches a reference point inside the live range. Also, the definitions of the variable inside the live range do not reach any other reference point outside the live range. Global register allocation assigns complete live ranges to registers, and if this is not possible, parts of live ranges are assigned. Computations for the separate live ranges of the program variables require processing and representation overhead. Since UOPT does not perform variable subsumption, computation of the separate live ranges is not strictly needed. Instead, one live range is assumed for each variable in a procedure at the beginning of the global register allocation phase. The optimizer can break each live range up into separate segments if necessitated by the register allocation process. In this respect, the live range of a variable in UOPT is the set of nodes in the program flow graph in which the variable needs be considered for allocation in register. This includes nodes in which the variable does not appear, because these nodes can serve as connecting links between definition nodes and reference nodes.

By virtue of the contiguity of the blocks in a live range, when the live range is assigned to a register, RLOD's are needed only at entry points to the live range and RSTR's are required

only at its exit points. UOPT supports both the caller-save and callee-save convention regarding registers in procedure calls. In the caller-save context, all registers need to be freed at a procedure call so that they can be used in the called procedure. Thus, live ranges are never allowed to extend over a procedure call. The optimizer is responsible for indicating which variable home locations are to be updated from registers before a procedure call, and which variables are to be re-loaded to registers after the call. Because of the occurrence of points that interrupt the contents of registers, our live ranges do not necessarily begin at definition points or end at reference points. When a procedure call is occurring later in the code, the live range of a variable should end at the last appearance of the variable before the call, regardless of whether that last appearance is a use or definition. Otherwise, it will needlessly occupy the register up to the procedure call when the register still has to be saved there. After a procedure call, the live range should begin at the first appearance of the variable, even though the procedure call may assign a value to it as a side effect. These remarks about live ranges bordering on procedure calls also apply to non-local variables near procedure boundaries: after the entry point to a procedure, the live range of a non-local variable begins at its first appearance; before the exit points of a procedure, the live range of a non-local variable ends at its last appearance. Fig. 4.5.1 gives an example of live range delimitation.

The live ranges of variables are computed by solving for the *live* and *reaching* attributes. A variable is *live* at block i if there is a direct reference of the variable at block i or at some point leading from block i not preceded by a re-definition or a procedure call. A variable is *reaching* block i if a definition or use of the variable reaches block i without passing through any procedure call. The live range of a variable is then the set of flow graph nodes in which the variable is both live and reaching.

In the case of the callee-save convention, live ranges are allowed to extend over procedure calls, and registers are allocated across the calls.

4.6. Global Register Allocation by Priority-based Coloring

The view of register allocation as a graph coloring problem has been well-established [Schw73] [Leve81] [Chai82]. A coloring of a graph is an assignment of a color to each node of the graph in such a manner that each two nodes connected by an edge do not have the same color. The interference graph is distinct from the program flow graph. Each node in the interference graph represents a program quantity that is a candidate for residing in a register. Two nodes in the graph are connected if the quantities interfere with each other. In our case, interference means there is overlap between their live ranges.

4.6. Global Register Allocation by Priority-based Coloring

After the building of the interference graph among the variables, the next stage is node coloring the interference graph. The number of colors used for coloring, r, is the number of registers available for use by the optimizer. The goal is to find the best way to assign the program variables to registers so that the execution time is minimized. Even if there are enough registers around, the best solution is not necessarily the one that allows all variables to reside in registers, because the cost of loading and updating the values of the variables have to be taken into account.

The standard coloring algorithm that determines whether a graph is r-colorable is NP-complete. It involves selecting nodes for which to guess colors, and backtracking if the guesses fail. The algorithm takes only linear time when the first trial succeeds. But if the graph is not r-colorable, or is in one of the borderline cases, an exponential amount of computation can be needed to prove that it is indeed so, since it is necessary to backtrack and attempt all possible coloring combinations before reaching the final conclusion. Thus, the standard coloring algorithm works well only when the target machines have a large number of registers. The standard coloring algorithm also does not take into account the cost and saving involved in allocating variables to registers. It always tries to allocate as many items in registers as possible, and does not consider the relative benefits of the individual variables, since they occur with different frequencies and with varying degrees of clustering. When it is found that an r-coloring is impossible, the decision regarding which variables to be excluded in the coloring (i.e. to be spilled) is difficult to make, since it is hard to predict the effect of spilling a certain variable on the outcomes of the subsequent coloring attempts. The loop-nesting depths of different parts of the program are also overlooked. In practice, variables occurring in frequently executed regions should be given greater preference for residing in registers. The algorithm also overlooks the fact that procedure calls affect register allocation. In the caller-save environment, the saving of registers before procedure calls and their reloading after the calls represent extra register allocation cost that has to be factered into the register allocation algorithm.

Because of the immense complexity of finding the optimal register allocation solution, most register allocators overcome the NP-completeness obstacle by aiming for a practical rather than the optimal solution†. Our philosophy regarding register allocation is the same. The emphasis is to do register allocation efficiently but still yield reasonable solutions for most input program configurations (with respect to the number of live ranges and the complexity of the interferences).

Our global register allocation algorithm is an adaptation of the standard coloring algo-

† To find the optimal solution also requires the use of very small code segments as the smallest allocation code range, and this also adds to the complexity of the allocation process.

rithm that enables us to overcome the problems in the standard algorithm mentioned above. By regarding all variables to have been assigned home locations before register allocation, we circumvent the problem of having to introduce spill code. Cost and saving estimates, which also include the effects of loop-nesting depths, are factored into the coloring decisions. This serves to prevent the over-allocation problem. The algorithm does not backtrack. Instead, it is benefit-driven. Allocation is ordered according to the cost and saving estimates. One live range is assigned to a register each iteration, each time picking the most promising live range according to the estimates of cost and execution time saving. It is hoped that this ordering procedure will allow the results of the allocation to be close to optimal. Our algorithm is also linear when an r-coloring can be found. Moreover, it does not deteriorate when r-coloring cannot be achieved. Thus, the algorithm works under any situation regarding register resources in the target machine — an attribute that is especially important in the machine-independent context.

Initially, we assume that one variable occupies a single live range, even though the live range may consist of non-adjacent parts. This allows us to avoid the cost of computing and representing separate live ranges prior to coloring. The interference graph is also made much simpler, and the processing cost associated with accessing, manipulating and updating the interference graph during coloring is also greatly reduced. In the course of performing coloring, when a variable cannot be assigned the same color throughout the procedure, its live program nodes will be separated into two or more groups, each group constituting a new live range. The new live ranges are treated the same way as variables as far as the coloring algorithm is concerned, and the interference graph is updated accordingly. Splitting is repeated until all the split live ranges can be colored or until all the split live ranges consist of single blocks. If a split-out live range is left uncolored at the termination of coloring, the effect is equivalent to spilling. In our case, no spill code needs be explicitly inserted, since register candidates are assumed to have been allocated in main memory either by the compiler front-end or earlier optimization phases. Live range splitting is performed with the emphasis on not creating small live range fragments unless warranted by the situation.

In the node coloring algorithm, variables which have a number of neighbors in the interference graph less than the original number of colors available are left uncolored until the very end, since it is certain that an unused color can be found for them. These are called *unconstrained* variables or live ranges. The rest of the variables live ranges are assigned colors by successive iterations of Step 2 of the algorithm. Each iteration selects a variable and assigns a color to it. New live ranges are formed out of splitting during the iterations, and if any of these are unconstrained, they are added to the unconstrained pool of variables. The iterations continue

until all constrained live ranges have been assigned a color, or there is no color left that can be assigned to any constrained variable in any code segment.

Algorithm *Priority-based Node Coloring.*

1. Find the live ranges whose number of neighbors in the interference graph is less than the number of colors available, and set them aside in the pool of unconstrained live ranges.

2. Repeat Steps a to c, each time assigning one color to a live range until all constrained live ranges have been assigned a color, or there is no register left that can be assigned to any live range in any code segment (taking into account registers allocated in the preceding local allocation phase).

   a. Perform Step (i) or (ii) for each live range lr until TOTALSAVE for all original or newly formed live ranges are computed:

   (i). If lr has a number of colored neighbors less than the total number of colors available, assume a color is assigned to it through all its live blocks. Then compute and record TOTALSAVE for the variable lr as follows:

   1. In each block i of the live range lr, determine whether register load and store is necessary based on whether the adjacent blocks in the flow graph belong to the same live range. Let the number of register loads and stores be n, which ranges from 0 to 2.
   2. Compute $\texttt{NETSAVE}_i$ as

$$\texttt{NETSAVE}_i = \texttt{LODSAVE} \times u + \texttt{STRSAVE} \times d - \texttt{MOVCOST} \times n$$

   where u is the number of uses of the live range variable and
   d is the number of definitions of the live range variable in block i.

   3. Let f_i denotes the frequency weight based on loop nesting of block i in the flow graph. Compute TOTALSAVE for the live range lr as:

$$\texttt{TOTALSAVE} = \sum_{i \in lr} (\texttt{NETSAVE}_i \times f_i).$$

   (ii). If the number of colored neighbors of lr is already equal to the number of colors available, then the live range lr has to be split. In performing live range splitting, attempts are made to split out as large live ranges as possible. A new live range lr_1 is split out from lr as follows:

   A new node in the interference graph is created for lr_1. A definition block from lr, preferably one at an entry point to lr, is first added to lr_1. Blocks adjacent to lr_1

that also belong to lr are successively added to lr_1, updating the neighbors in the interference graph until the number of colored neighbors of lr_1 in the interference graph is one less than the number of available colors. The motivation of this is to produce the largest possible live range that can still be colored. This is continued until no more adjacent block can be added to the new live range lr_1.

If the newly formed live range lr_1 has a number of neighbors in the interference graph less than the number of colors available, set it aside in the pool of unconstrained variables to be colored later. Otherwise, add it to the pool of candidates for estimation of TOTALSAVE.

As a result of the new node in the interference graph, some previously unconstrained live ranges may now become constrained. These have to be updated.

b. For each live range lr, compute ADJSAVE as

$$\text{ADJSAVE} = \frac{\text{TOTALSAVE}}{(\text{number of nodes in } lr)}.$$

(The quantities TOTALSAVE and ADJSAVE do not have to be recomputed if the live range has not changed since the previous iteration.)

c. Looking at the values of ADJSAVE computed for all the uncolored but constrained live ranges in Steps a and b, choose the live range with the highest value of ADJSAVE and assign a color to it.

3. Assign colors to the unconstrained live ranges, each time using a color that has not been assigned to one of their neighbors in the interference graph. □

Thus, the algorithm orders the assigning of colors according to which variable currently has the highest value of ADJSAVE (Step 2c). ADJSAVE can be visualized as the total number of occurrences of the variable in the live range, weighted by loop-nesting depths and normalized by the length of the live range. The adjustment by the live range length (the number of basic blocks belonging to the live range) is needed because a live range occupying a larger region of code takes up more register resource if allocated in register. In the local allocation phase, we have already taken pure occurrence frequencies into account. Thus, when entering the global allocation phase, all the variables that remain unallocated in each code segment have occurrence frequencies that do not differ widely, so the important consideration is whether the allocation enables the same register to be assigned across contiguous code segments so that register loads and stores can be minimized. The value of ADJSAVE comprises a measure of this connectedness. The more connected the code segments in the live ranges of a variable are, the more worthy is the variable to be allocated in register, and the more difficult it will be to find the same register

for it throughout; so, it is important to assign a color to it before other variables. The use of the ADJSAVE criterion is justified only if the local allocation phase precedes global allocation.

The determination of n in Step 2a(i) can make use of more information than previously possible in the local allocation phase of Section 4. If the first occurrence of the variable at an entry block is a store, then the RLOD is not needed. If all the predecessors of a block also belong to the live range, then the RLOD is also not necessary, unless any of the predecessor contains a procedure call in the case of caller-save environments, or the current block is an entry node (including the case of a goto-out-of-block target). An RSTR is necessary at the exit blocks of a live range only if the live range contains at least one assignment to the live range variable and the variable is not dead on exit. At blocks internal to live ranges, RSTR's are also generated if any successor node has an RLOD, or contains a procedure call in the case of caller-save environments.

The computation time complexity of the above algorithm can be estimated. We are mainly concerned with Step 2 of the algorithm, since this step takes a lot more time compared with Step 3 for the unconstrained live ranges. Let r be the number of registers. Let l be the number of live ranges, and assume that this stays fixed during the course of the algorithm. Also assume that each register is assigned to one and only one live range in the procedure, though in reality this is not always the case. Then there is r iterations for Step 2 of the algorithm. For the first iteration, a live range is to be chosen out of l live ranges. For the second iteration, the choice is to be made out of the $l-1$ live ranges remaining. Summing all the iterations, we get

$$l + (l-1) + \ldots + (l-r+1) = \frac{r(2l-r+1)}{2}.$$

Thus, the algorithm is $O(r(l-r))$. The time of the algorithm proportional to both the number of registers available and the number of candidates to reside in registers.

The above algorithm can easily extend to the case of multiple classes of registers. The interference graph will only give interferences between variables of the same class. The algorithm is repeated once for each class of register. In each case, the number of colors corresponds to the number of registers in the class being considered.

4.7. Optimization of Register-Memory Moves

To enhance the effectiveness of register allocation, the optimizer must optimize the register move operations it introduces as much as possible. In the previous register allocation phases, the optimizer takes into account the cost of the register move operations in determining register allocation. The RLOD's and RSTR's are assumed to be placed at the beginnings and ends of allocation code segments in the saving estimates.

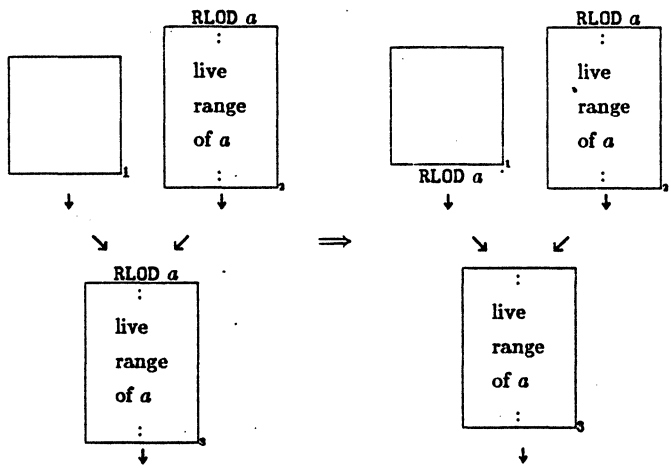


Fig. 4.7.1 Removing partial redundancy in RLOD

After register allocation has been completed, UOPT conducts one further pass to optimize the placements of RLOD's and RSTR's. This optimization can be viewed as a form of code motion, since the purpose is to move the register transfer instructions away from frequently executed regions. The algorithm of Section 3.6 can in principle be used, but in practice, a more simplified and condensed approach is possible. This is based on the fact that RLOD's are generated only at entry points to live ranges and RSTR's at their exits. Furthermore, the RLOD's and RSTR's are never moved across entire blocks, since this would alter the effective live ranges. RLOD's are only moved from the entry points of blocks to the exit points of their immediate predecessors, and RSTR's are only moved from the exit points of blocks to the entry points of their immediate successors. No data flow analysis is involved.

An RLOD for a variable a in block i is moved to the exits of the predecessors of i when the following conditions are satisfied:

(a) At least one predecessor of i belongs to the same live range of a;

(b) All the predecessors of i that do not belong to the live range have i as their only successor;

(c) i is not the target of a goto-out-of-block.

When the above conditions are satisfied, the RLOD is deleted from i and inserted at the exits of the predecessors of i which do not belong to the live range (Fig. 4.7.1). When one of the predecessors of i belonging to the same live range under condition (a) is also reachable from i, the result is the movement of the RLOD from the loop in which i is the loop entry block (Fig.

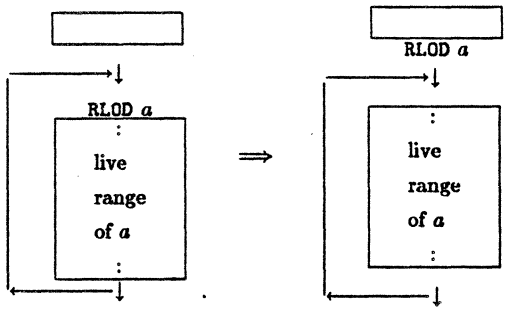


Fig. 4.7.2 Movement of RLOD out of loop

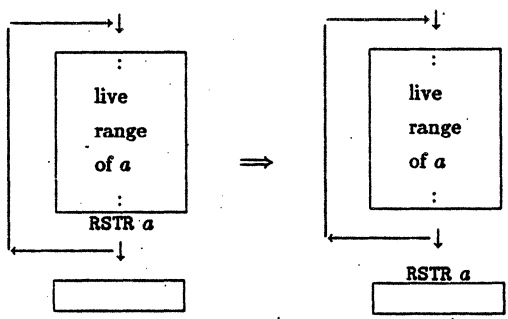


Fig. 4.7.3 Movement of RSTR out of loop

4.7.2).

An RSTR for a variable a in block i is moved to the entries of the successors of i when the following conditions are satisfied:

(a) At least one successor of i belongs to the same live range of a, and there is no RLOD of a at the entry point of that successor;

(b) For the successors of i which do not satisfy condition (a), they have i as their only predecessor, and are not the targets of gotos-out-of-block.

When the above conditions are satisfied, the RSTR is deleted from i and inserted at the entries of the successors of i that do not satisfy condition (a). As in the case of RLOD, forward movement out of loops (Fig. 4.7.3) is a special case of this transformation.

4.8. Summary

In this Chapter, we have introduced an integrated register allocation scheme that is suitable for use in the machine-independent context. The algorithm works for most configurations of general-purpose registers in the target machines up to and including the grouping into non-intersecting register classes. The performance and efficiency of the algorithm are not affected by the number of registers available. We introduced the parameterization of cost and saving in register allocation that enables our algorithm to cater to the different characteristics in machines regarding the benefits of register accesses over memory accesses.

The register allocation is divided into a local and a global phase. The local phase is employed to perform some initial allocation quickly that can reduce the work load of the subsequent global phase without affecting the final outcome. The local phase is useful only when there are long basic blocks. But when blocks are long, register allocation is unable to cater to the clustering of appearances within the blocks. The user can decrease the maximum size of the blocks used, thus increasing the number of discrete code segments and allowing the finding of more optimal register allocation solutions. The processing cost in register allocation will correspondingly increase, when more work is involved in the global phase.

The global register allocation scheme is an adaptation of the standard coloring algorithm. The standard algorithm handles insufficient registers by spilling variables into main memory. We have taken the different approach of assuming that all variables have been assigned home memory locations initially, and we handle the situation of insufficient registers by live-range splitting. This allows us to make the initial assumption of one live range for each variable throughout the whole procedure, which in turn enables us to avoid the processing and representation overhead of computing separate live ranges prior to performing the global allocation. The resulting smaller size of the interference graph also saves the processing cost associated with accessing and manipulating the interference graph during coloring. Our node coloring algorithm is priority-based. The allocation is ordered according to which variables have greater priority for residing in registers. By taking into account the cost of register transfer operations to and from memory, we can factor the effects of not allocating in register into the coloring decisions. We have weighted the cost and saving estimates by the loop-nesting depths of the regions concerned, and thus also take into account the control flow of the procedure concerned. Using the cost and saving estimates also makes it possible for us to take into account the effects of procedure calls in caller-save environments. The running time of our coloring algorithm is proportional to the number of registers and the number of live ranges to be colored. After the completion of register allocation, we conduct one more pass to optimize the positions of the register-memory transfer operations by suppressing partial redundancies among them.

5. Organization and Structure

In this Chapter, we look into the overall organization and structure of UOPT in implementing the optimization algorithms presented in Chapters 3 and 4. The interactions among the optimizations performed are addressed. A specific order for performing the optimizations is developed. Based on our implementation, the timings and efficiencies of the various optimization phases are studied. The data structures used in UOPT are described. The methods used in the collection of data flow information are examined. The interactions of UOPT with the procedure integrator PMERGE are also discussed.

5.1. Optimization Phase Structure

In performing optimization on an input program module, UOPT passes over the program code only once, when it reads in the code of the procedures. It optimizes procedures one at a time, writing out the optimized code before reading in the next procedure. In general, the contents of one procedure have no effect on the optimization of other procedures (i.e. no interprocedural analysis is done). The one exception to this is that UOPT does remember the levels of previously encountered procedures. By taking the static nesting of procedures into account, UOPT can determine whether side effects on variables in the current procedure are possible.

The input procedure code is separated into basic blocks while they are read. Basic blocks are delimited according to the set of op-codes that mark the ends of basic blocks, and U-Code labels that mark the starts of new basic blocks. As the code is read in, unreachable code is also removed by skipping until the next label if the previous basic block ends with an unconditional jump or a return. Some local optimizations are performed as part of the process of inputing program code, when data structures are built to represent the basic block code. After each basic block is completely read in, the remaining set of local optimization transformations are invoked (Section 3.1). Following local optimization, the local data flow attributes (Section 3.3.1) are collected. The reading of a basic block also causes a node to be added to the global control flow graph.

Once the whole procedure is read in, the global optimization phases begin. The initial step is analysis of the control flow graph. Unreachable flow graph nodes are identified, and the corresponding basic blocks are deleted. The control flow graph nodes are put into a depth-first ordering for maximizing speed in the subsequent data flow analyses. Additional data flow information is collected from the program code.

Using the global optimization approach presented in Chapter 3, we identify the following three underlying phases in global optimization:

Phase A — Copy propagation.

Phase B — Partial redundancy elimination for expressions (backward code motion).

Phase C — Partial redundancy elimination for stores to both program variables and optimizer-generated temporaries (forward code motion).

To the above, we add extra phases that perform the optimizations not yet included:

Phase D — Linear function test replacement.

Phase E — Induction variable elimination.

Phase F — Register allocation.

Induction variable elimination (Section 3.7) cannot be included in phase C because the data flow information used in solving for redundant induction variables has to be specially set up to disregard increments to induction variables.

For completeness, we list the local optimization phase here since new local optimization opportunities may be created by various code movements:

Phase G — Local optimization.

For maximum optimization efficiency, the different optimization phases should be performed only once. This, however, conflicts with the objective of achieving the most optimization, since further optimization opportunities can be uncovered by performing a given set of optimization transformations. Our objective is to develop a particular sequence in which the above optimization phases are applied or repeated that represents the best tradeoff between optimization efficiency and exhaustive optimization. We have to take into account the interactions between the various optimizations and the need to update the relevant data flow information after each optimization phase.

5.1.1. Underlying Principles

A program can be visualized as a sequence of points at which variables are alternately defined and referenced. Let d denotes a direct assignment to a variable a and u denotes a direct reference of a. Let u_i denotes an operation which may potentially reference the value of a, and d_i denotes an operation which may potentially alter the value of a. d_i and u_i occur in indirect loads (ILOD's) and indirect stores (ISTR's) respectively, and also in procedure calls due to side effects, and in the passing of address parameters to called procedures (see Section 5.4).

The optimizations of backward and forward code motion involve moving the u's, d's, u_i's and d_i's around, although procedure calls are considered stationary points and never moved.

5.1. Optimization Phase Structure

Embedded in the code motion algorithms of Chapter 3 are criteria for determining the movement of items from one point to another. One of the criteria is the rule that governs the legality of code movement: a u or u_i item cannot be moved across a d or d_i point, and a d or d_i item cannot be moved across a u, u_i, d or d_i point†. As an example, suppose the variable a has the following occurrences in a straight-line piece of code:

$$d_{i_1} \ldots u_1 \ldots d_2 \ldots d_3 \ldots u_2 \ldots u_{i_1} \ldots u_3.$$

Then a legal rearrangement of this piece of code is

$$d_{i_1} \ldots u_1 \ldots d_2 \ldots d_3 \ldots u_3 \ldots u_{i_1} \ldots u_2.$$

Whenever there are two consecutive occurrences of d's, the earlier occurrence is redundant. This transformation takes place in the redundant store elimination algorithm of Section 3.5. Thus, the above sequence of code can be reduced to

$$d_{i_1} \ldots u_1 \ldots d_3 \ldots u_3 \ldots u_{i_1} \ldots u_2.$$

The u's and d's also govern the availability of computations, which plays a major role in copy propagation and common subexpressions. An expression or assignment is no longer available after the occurrence of a d or d_i that changes the value of any of the variables in the expression or the value of the assigned variable.

In the code motion optimizations of phases B and C, the above d's and u's occurrences are what limit the code movement that can be attempted. Thus, in the code sequence

$$d_1 \ldots u_1 \ldots d_2,$$

if d_1 had been moved backward (to the left) or deleted, it would be possible to move u_1 backward past the original position of d_1. Similarly, if d_2 had been moved forward (to the right)or deleted, it would be possible to move u_1 forward past the original position of d_2. The same reasoning applies to the movement of a d in relation to the other u's, u_i's, d's and d_i's in its vicinity.

---

† When an item moved consists of multiple variables, the *use-def* of all the variables are taken into account.

5.1.2. Relationships among the Phases

We now study the interactions among the phases A to G we enumerated above in order to establish the best order of applying the various optimizations. The first observation we can make is that register allocation should be the last phase in the optimization sequence, because it has to take into account the appearances of all potential register-residing items, which are affected by all the other optimizations. Among the register-residing items are optimizer-generated temporaries whose associated optimizations are beneficial in terms of execution speed only if the temporaries are allocated in registers.

Linear function test replacement (phase D) has to be performed right after backward code motion (phase B), because it makes use of the availability information computed in that phase in finding expressions to replace a test variable. Induction variable elimination (phase E) depends on the test replacements performed, so phase E should occur after phase D.

Having taken care of register allocation, linear function test replacement and induction variable elimination, we are left with copy propagation (phase A), backward code motion (phase B), forward code motion (phase C) and local optimization (phase G). To study the interactions among these four different optimizations, we construct Table 5.1.1. In each entry of this table, we need to decide whether the optimization of the row entry affects the optimization of the column entry. Theoretically speaking, whenever an entry is yes, it is necessary to repeat the column entry's optimization after each application of the row entry's optimization in order to exhaust all optimization opportunities.

Entry I(a). According to our local optimization algorithms, the local optimization pass does all possible local optimizations within each basic block, and it is useless to repeat the local optimization pass on itself.

Entry I(b), I(c) and I(d). Local optimization can affect all other optimizations. We do not consider the optimization of local common subexpressions here, since it is a direct result of inputing the program code. Constant folding and stack height reduction change the structures of expressions. Expressions are mapped to their constant-folded and stack-height-reduced forms, and these locally optimized forms of the expressions are used in global data flow analyses. As we have mentioned in Section 3.1.2, local copy propagation enables more common subexpressions to be recognized, and also can create redundant assignments. Thus, we make the local optimization phase in UOPT precede all other optimizations.

Entry II(a). Copy propagation merges expressions from outside the basic block into expressions within the basic block. New local common subexpressions can be introduced. The large expressions may exhibit new opportunities for constant folding and stack height

	(a) Local Optimization	(b) Copy Propagation	(c) Backward Code Motion	(d) Forward Code Motion
I. Local Optimization	*No*	*Yes*	*Yes*	*Yes*
II. Copy Propagation	*Yes*	*No*	*Yes*	*Yes*
III. Backward Code Motion	*Yes*	*Yes*	*No*	*Yes*
IV. Forward Code Motion	*No*	*Yes*	*Yes*	*Yes*

Does the optimization of the row entry brings in new opportunities in the optimization of the column entry?

Fig. 5.1.1 Inter-relationships between the optimizations

reduction. In UOPT, local common subexpressions are recognized after copy propagation by re-hashing the newly formed expressions. Constant folding and stack height reduction are repeated in the final code re-emission phase.

Entry II(b). The copy propagation algorithm of Section 3.4 does all possible copy propagation for each basic block variable, and every time a new expression is merged into a basic block, the copy propagation algorithm is repeated recursively in the newly introduced expression. Thus, it is unnecessary to repeat the copy propagation pass on itself.

Entry II(c). As we have mentioned in Section 3.4, common subexpression recognition is a necessity after copy propagation for preventing the proliferation of copied expressions.

Entry II(d). One primary purpose of copy propagation is to create dead variables or redundant assignments. Thus, a redundant store elimination phase should always take place after copy propagation.

Entry III(a). Backward code motion involves the deletion and insertion of expressions at various points in the program. Expression structures are not altered. Any new local copy propagation that can possibly be resulted could have been globally accomplished in

global copy propagation. Any new local common subexpression that results from the insertions would have already been recognized as such by our backward code motion algorithm. Backward code motion cannot result in new store redundancy in the program variables. However, backward code motion does create new opportunities for local copy propagation, which we explain under entry III(b).

Entry III(b). Backward code motion does not usually create new opportunities for copy propagation, because the movement of the expressions does not alter the solution for the availability of assignments, represented by Eq. (3.4.1). An exception is in the case of induction expressions, where in the original code the increments to the induction variables prevent copy propagation from taking place. After an induction expression is moved to a loop header, an assignment to the induction variable may be available there so that new copy propagation can occur. For example, in Fig. 3.7.1, the induction expression $i \times 3$ is constant propagated to 1×3 and then folded to 3 after code motion has taken place.

Entry III(c). Backward code motion involves the movement of both the d's and the u's. In the code sequence

$$\ldots d_1 \ldots u_1 \ldots,$$

after d_1 has been moved to the left, u_1 can be moved to the left past the original position of d_1. This movement of u_1 past the original position of d_1 cannot be done concurrently in one pass of our backward code motion algorithm, since the deletion of d_1 from its original position is not done until the end of the pass. The movement of d_1 to the left must be due to some store partial redundancy in the variable. This means that some d occurs to the left of d_1, and the presence of this earlier d implies that the same form of partial redundancy that moves d_1 to the left cannot occur to u_1 after the movement of d_1. Thus, we conclude that, in the above code, if d_1 has been moved to the left, repetition of our code motion algorithm will never result in moving u_1 to the left past the original position of d_1. The same argument applies to the code sequence

$$\ldots u_1 \ldots d_1 \ldots.$$

Thus, there is nothing to gain by repeating the backward code motion pass on itself.

Entry III(d). Backward code motion involves the backward movement of u's, u_i's, d's and d_i's, and forward code motion involves their forward movement. Thus, the optimizations of these two phases are mutually restricting, and no new forward code motion optimization can be brought about by backward code motion. Even when some redundant expressions are deleted, no new redundancy in stores can be resulted, since the redundant expressions are

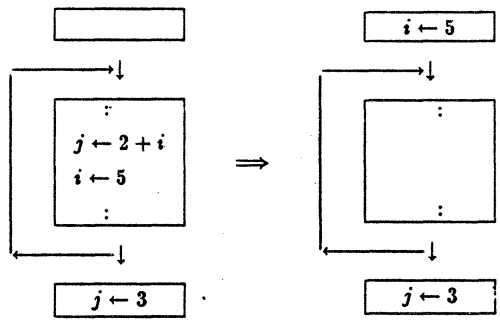


Fig. 5.1.1 Effects of redundant store elimination followed by backward code motion

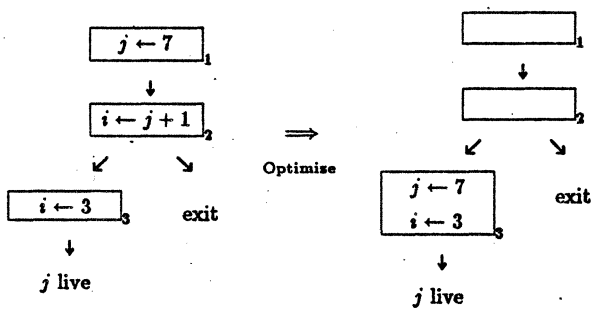


Fig. 5.1.2 Effects of redundant store elimination followed by further forward code motion

deleted at points where some earlier occurrences of the expressions are available. However, our forward code motion phase also eliminates redundancies in the saving of temporaries, and these temporaries are generated by the backward code motion phase. Thus, a forward code motion pass should always take place after the backward code motion phase.

Entry IV(a). Forward code motion is not likely to introduce new opportunities for local optimization. It involves the movement of assignments together with their assigned expressions. It does not alter expression structures. Any new local copy propagation could have been globally performed in the global copy propagation phase. Any new local common subexpression could have been recognized as global common subexpression in the backward code motion phase. Any new local redundant assignment could also have been globally suppressed earlier.

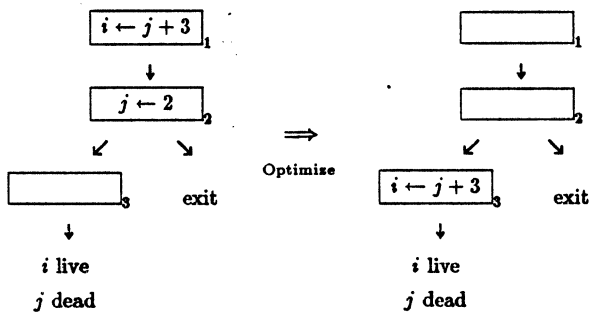


Fig. 5.1.3 Effects of redundant store elimination followed by further forward code motion

Entry IV(b). Forward code motion does not create new occasions for copy propagation, since the code motion does not influence the solution of Eq. (3.4.1). However, copy propagation for the temporaries created in other optimizations is possible after forward code motion. In Fig. 3.10.1(c), copy propagation of the temporary t is possible. This optimization is not performed in UOPT since such cases do not frequently occur and they do not considerably affect execution time.

Entry IV(c). As we have mentioned under entry III(c), the code movements in backward code motion and forward code motion are mutually restricting when no deletion is involved. However, when deletion of d's, d_i's, u's or u_i's takes place†, backward code motion can benefit because larger gaps for code movement are made possible. For example, in Fig. 5.1.1, the deletion of the redundant $j \leftarrow 2 + i$ (a u in i) enables $i \leftarrow 5$ (a d in i) to be moved out of the loop. The deletion of a d is possible only when the next occurrence is another d, so the backward movement of any u further down cannot be affected. But the example shows that a store redundancy elimination phase can create more opportunities for backward code motion.

Entry IV(d). The reasoning similar to that of entry III(c) is also applicable here. However, in the current case, deletions involve deleting both the redundant stores and the right-hand-side assigned expressions. The assigned expressions may contain u's that previously obstruct the forward movement of the corresponding d's. For example, in Fig. 5.1.2, the deletion

† The deletion of assignments are unique in that both the left and right hand sides are eliminated. The left hand side is a d for the assigned variable, and the right hand side may consists of u's for other variables. The data flow solution in forward code motion that leads to the deletion of stores is dependent only on the data flow attributes of the left hand sides of stores, and is independent of the data flow attributes of the right hand side expressions.

of $i \leftarrow j + 1$ (a u in j) enables the forward movement of $j \leftarrow 7$ (a d in j) to remove a store partial redundancy in j. The deleted store may also facilitate the forward movement of assignments whose assigned expressions contain uses of the variables whose stores are deleted. For example, in Fig. 5.1.3, the deletion of $j \leftarrow 2$ (a d in j) enables the forward movement of $i \leftarrow j + 3$ (a u in j) to remove a store partial redundancy in i. Thus, it is useful to repeat the forward code motion phase on itself.

5.1.3. The Actual Optimization Phases

We now construct a practical optimization sequence according to Table 5.1.1 and our discussions related to this table. As we have mentioned earlier, local optimization is applied while inputing each basic block. Because copy propagation and backward code motion can affect local optimization, we repeat local copy propagation and constant arithmetic in the final code re-emission phase.

Because copy propagation affects both backward code motion and forward code motion (entries II(c) and II(d)), it is best performed as the first global optimization phase. Under entry IV(c), we have concluded that a store redundancy elimination phase is beneficial for backward code motion. Thus, after copy propagation, we conduct a store redundancy elimination phase. This phase does not perform full forward code motion optimization since this is done in a later phase. Next, we perform backward code motion. Immediately following backward code motion is linear function test replacement and then induction variable elimination. The final global code optimization phase is forward code motion, which takes into account the expression temporaries generated in the backward code motion phase and the induction variables eliminated in the induction variable elimination phase. Register allocation concludes the global optimization phases.

We now list the complete sequence of events in the optimization of a procedure by UOPT:

Phase 1 — Input of the procedure code and performance of local optimization on a block by block basis.

Phase 2 — Collection and setting up of data flow information.

Phase 3 — Processing of the program control flow graph.

Phase 4 — Copy propagation.

Phase 5 — Elimination of redundant assignments.

Phase 6 — Partial redundancy elimination for expressions by backward code motion. (This

includes global common subexpressions elimination, loop-invariant expression removal and strength reduction.)

Phase 7 — Linear function test replacement.

Phase 8 — Induction variable elimination.

Phase 9 — Partial redundancy elimination for stores to both program variables and optimizer-generated temporaries by forward code motion.

Phase 10 — Global register allocation and assignments, and allocation of storage to temporaries not residing in registers.

Phase 11 — Emission of optimized code, with further local transformations applied to a few op-codes.

To recognize the relationship displayed in Table 5.1.1, our requirement is that, for each yes entry in the table, there must be an occurrence of the optimization of the corresponding column after the occurrence of the optimization of the corresponding row†. The above optimization sequence in UOPT obeys our requirement with the exception of entries III(b) and IV(b). Notice that the extra redundant assignment elimination pass of phase 5 has taken care of entries IV(c) and IV(d). Entries II(a) and III(a) are taken care of by the extra local optimizations performed during the final code emission phase. The ignorance of entry III(b) results in induction variable moved out of loops not being globally constant propagated. However, in most cases, the constant propagation of these induction variables is local in nature, and this is taken care of in the code emission phase.

Updates of all data flow information are needed for the code transformations done in phases 4 and 5. After phase 6, only the data flow information related to stores needs to be updated. Global data flow analysis is performed in phases 4, 5, 6, 8, 9 and 10. These different optimizations require different kinds of global data flow information. Also, since the global data flow attributes may be affected by each update, it is necessary to re-compute the relevant global data flow information each time prior to its use.

UOPT can potentially be re-invoked to conduct another optimization pass over its own optimization output to further exhaust the optimization opportunities. In such a second optimization pass, the new optimization opportunities that can be recognized will be very marginal, not only because most of them have already been performed, but also due to the numerous

† To fully implement Table 5.1.1, it is necessary to to apply this reasoning for the repetition passes also, but we regard this as overkill.

NSTR's introduced that prevent the construction of complete trees. The items allocated in registers also need to be remapped to regular storage. To re-run UOPT over its own optimized output, it is currently necessary to turn off the register allocation option in previous runs.

5.2. Timings of the Optimization Phases

The execution times of the various optimization phases in UOPT have been measured on a set of input benchmark programs. The approximate times spent in the different phases, expressed as percentages of the total optimization time, are as follows:

Phase 1:	5 – 10 %.
Phase 2:	25 – 30 %.
Phase 3:	negligible.
Phase 4:	5 – 7 %.
Phase 5:	2 %.
Phase 6:	10 – 15 %.
Phase 7:	negligible.
Phase 8:	2 – 3 %.
Phase 9:	10 – 15 %.
Phase 10:	20 – 25 %.
Phase 11:	5 – 10 %.

All the above optimization times are reasonable, except perhaps phase 2. The main reason why phase 2 is time-consuming is that, for each variable, expression or assignment that occurs in the procedure, it is necessary to check whether it is affected by the code of each individual basic block, regardless of whether it occurs in the basic block or not (Section 5.4). Thus, the complexity of this phase is of the order of the total number of variables, expressions and assignments in the procedure, which is the length of the bit vectors, times the total number of basic blocks.

The total amount of time spent in data flow analysis has also been measured. There are altogether 15 separate data flow analysis steps among all the phases. It is found that approximately 10 – 17 % of the total optimization time is spent in performing data flow analysis. The average number of iterations needed in performing each data flow analysis is 3.

Because quite a number of operations in the various phases are of the order of complexity of the total number of variables, expressions and assignments in the procedure times the total number of basic blocks, the time taken to optimize a procedure is approximately proportional to the square of the procedure size.

5.3. Data Structures

The data structures in UOPT are designed to represent the executable code of a complete procedure while performing optimizations. Since a procedure can be of arbitrary length, the data structures have to be space-efficient to accommodate large procedures. The data structures should also allow the various operations during optimization to be performed efficiently.

5.3.1. Data Structures for Global Optimization

For the purpose of recording program code, hash tables and linearly linked lists of statement nodes are used. A node in the linear list represents the equivalent of a statement in the source language. The order of appearance of the statements in the input program is preserved in the linked list. The hash tables are for representing expressions in the form of triples (op, l, r). Hashing of the table entries allows fast retrieval of the entry for a given expression in the construction of DAG's.

Two hash tables are used in UOPT. The *local hash table* contains all the expressions in the whole procedure. Apart from representing code, it also plays a crucial role in the recognition of local common subexpressions, since expressions exist in the table in the form of DAG's (Section 3.1.1). Each entry gives the basic block in which the item occurs. The same variables or expressions from different basic blocks are mapped to different table entries, so that any common subexpression recognized is limited to within the same basic block. Expressions in the local hash table are pointed to from the statement nodes that reference them. The list of statement nodes together with the local hash table gives the complete code of the procedure being optimized.

The *global hash table* is used to record the variables or expressions that exist in the procedure. One of its uses is to give the unique bit vector position assigned to each data flow item. Unlike the local hash table, each variable or expression is given a unique entry, regardless of where and how many times it occurs in the procedure. Although it is also in the form of DAG's, it is not used for recognizing common subexpressions. The DAG characteristic, which is due to the hashing nature of the table, also allows a smaller number of entries to be used in the case of tree expressions with commonly nested subtrees. An additional column in the global hash table gives the entry number of the item assigned to each bit position. Thus, from the assigned bit positions, the aggregate of all the variables and expressions that have appeared in the procedure can be accessed. An entry in the global hash table can be regarded as the image of many different entries in the local hash table, which are of the same variable or expression but belong to different basic blocks. Thus, the global hash table is of a fraction of the size of

the local table. Each entry in the local hash table has a pointer field that gives its image in the global hash table.

The control flow graph of a procedure is represented by a list of graph nodes. Each node corresponds to a basic block in the procedure, and has a list of predecessor nodes and a list of successor nodes. The predecessor and successor relationships together give the control structure of the procedure. The list of statement nodes for each basic block originates from the corresponding graph node in the control flow graph. Each basic block node also gives information about the state of register usages in that block.

Data flow analysis in UOPT is performed by bit vector operations. The total number of bits used depends on the number of different variables and expressions that exist in the procedure. Bit vectors are implemented by linked lists of sets in Pascal so that the lengths of bit vectors used are not restricted. As a result, a bit vector operation corresponds to multiple set operations for individual sets in the bit vector lists. The efficiency of bit vector operations depends on the maximum set length that the host machine can handle in a single machine operation. A bit vector gives information about a certain data flow attribute at a given basic block. To provide information about an attribute throughout the procedure, there has to be one bit vector per basic block for the data flow attribute. Since the bit vectors are used mainly in data flow analysis, they are closely related to the control flow graph. Thus, the bit vectors for the different data flow attributes also originate from the basic block nodes in the control flow graph.

The basic block nodes also have other bit vectors that give details about the changes to be made to the code in the basic blocks as the results of global optimizations. Computations to be inserted at the entry and exit of each basic block are indicated by two `INSERT` bit vectors, and computations to be deleted in the basic block are given by the `DELETE` bit vector (Section 3.5 and 3.10). The final code re-emission phase will generate the optimized output according to the contents of these bit vectors. While these bit vectors have been the direct results of our global optimization algorithms, this method of representation is also space-efficient, since bits rather than actual code-representing data structures are used. The overhead in manipulating the data structures in code insertions and deletions is also saved.

5.3.2. Data Structures for Register Allocation

Register allocation determines the items to reside in register at any point in the program code. The smallest segment of code over which an item is assigned to a register is a basic block in the control flow graph. Each basic block node contains information about the availability of the register resource for each register class in the basic block during and after register allocation.

Before register allocation begins, the live ranges of all potential register-residing items have to be determined by data flow analysis (Section 4.5). At the end of the data flow analysis, an ACTIVE bit vector in each basic block node indicates the variables and expressions whose live ranges cover that basic block. Inside a given basic block, only these active items need to be considered for possible assignment to registers. We refer to each pair of basic block and active item as a *live unit*.

Although the live ranges of the variables and expressions are already given in the ACTIVE bit vectors of the basic block nodes, such a representation is not adequate for supporting the various manipulations during register allocation. Additional data structures to represent individual live ranges and individual live units are necessary. A *live range node* represents a live range for a variable or expression. Each entry in the global hash table points to a list of the live range nodes corresponding to all its separate live ranges in the procedure. Since only one live range is assumed for each item initially (Section 4.6), only one live range node is created at the beginning of register allocation. As live ranges are split in the course of register allocation, new live range nodes are created and linked together in the lists. Before register allocation, each variable or expression is assigned a unique bit position. These bit position assignments are also used for the unsplit live ranges at the start of register allocation. As new live ranges are formed from splitting, they are assigned new, unique and unused bit positions to indicate that they are now considered separate from their parent live ranges.

Each live range node points to a list of *live unit nodes* that represents the individual live units belonging to the live range. Each live unit node contains register allocation information for the item in a basic block. This includes the number of local uses and assignments of the item, and information as to whether the first appearance is a store, and whether the item is dead on block exit. According to whether the predecessors and successors belong to the same live range, two flags also indicate whether RLOD and RSTR need to be generated at the block entry and exit respectively if the item is allocated in register. An additional field tells if the item has been locally allocated to register in the local register allocation phase.

Each live range node contains other information related to register allocation. All the basic blocks covered by the live range are given by a set of basic block numbers. The saving estimate that indicates the saving achieved if a register is assigned to the live range, which is computed in the node coloring algorithm (Section 4.6), is also given. A field also indicates if a color (register) has been assigned to the live range.

The interference graph is given using pointers among the live range nodes. Each live range node has a list of interference pointers that point to the live range nodes interfering with it. To see whether two live ranges interfere, it is only necessary to check whether they contain common

basic blocks. This can easily be found by computing the intersection of the two sets of basic block numbers in their live range nodes and checking whether the result is an empty set.

In implementing the node coloring algorithm of Section 4.6, bit vectors are used in separating all the live ranges into pools. For example, bit vectors are used to indicate the items that are candidates for each class of registers. Another bit vector gives the unconstrained versus the constrained live ranges. During the node coloring iterations, another bit vector gives the items that have so far been allocated to registers. This method of processing is storage-efficient, and also reduces the overhead in moving data structures around.

At the end of register allocation, the final register assignments are given in tables in the basic block nodes, with one table entry for each register in the target machine. The register tables also indicate whether RLOD's and RSTR's are necessary. In the final code re-emission phase, these tables are referenced to generate the appropriate register code for items residing in registers. Registers not used by the optimizer throughout the procedure are indicated in the output so that they may be used by the code-generating back-ends.

5.4. Collection of Data Flow Information

In Section 3.3, data flow information is classified into local attributes and global attributes. Local attributes are the data flow information that can be collected by looking at the code of a basic block. Global attributes are the data which have to be computed by data flow analysis. In this section, we focus on the collection of the local attributes.

Data flow information depends on the memory relationships among the storage items in a program, and the sequence in which the uses and stores of the items occur. In Section 5.1, we have referred to these appearances as u's, d's, u_i's and d_i's. We divide memory references into four categories:

(i) Direct loads of simple variables — This corresponds to the LOD instruction.

(ii) Indirect loads with known source range — This comes from the uses of the ILOD and the indirect comparison operators whose base addresses are given by the LDA instructions. The passing of a reference parameter is also regarded as an indirect reference, and so is included in this category. This corresponds to a PAR instruction with an address parameter based on an LDA instruction.

(iii) Indirect loads with unknown source range — This comes from the uses of the ILOD and the indirect comparison operators whose base addresses are loaded from locations in memory or are the results of function calls. The passing of a reference parameter whose address is formed the same way is also included.

(iv) Procedure calls — A called procedure can reference variables at the lexical levels that surround it (up-level references).

The indirect comparision operators are IEQU, INEQ, IGRT, IGEQ, ILES and ILEQ. Each of them involves two indirect references. Associated with the LDA instruction are two fields that specify the lower and upper limits of the address range within which the resultant address of any address computation that ensues can possibly lie. This information can easily be supplied by the compiler.

In a similar way, memory assignments are classified into four categories:

(a) Direct stores to simple variables — This corresponds to the STR instruction.

(b) Indirect stores with known target range — This occurs with the uses of the ISTR, INST, MOV and VMOV instructions whose base addresses are given by the LDA instruction. The passing of a reference parameter can also involve a potential store to the passed parameter in the called procedure, and is also included.

(c) Indirect stores with unknown source range — This comes from the uses of the ISTR, INST, MOV and VMOV and the indirect comparison operators whose base addresses are loaded from locations in memory or are the results of function calls. The passing of a reference parameter whose address is formed the same way is similarly included.

(d) Procedure calls — A called procedure can store to any variable at the lexical levels that surround it.

In collecting data flow information, we are concerned with whether a memory reference (categories (i), (ii) and (iii)) is affected by the memory assignments (categories (a) to (d)) in the region concerned, and whether an assignment (categories (a), (b) and (c)) is affected by the memory references and assignments (categories (i) to (iv) and (a) to (c)) in the region concerned. The kinds of operations to be taken into account depend on the actual definition of the local data attribute being considered (Section 3.3.1). In all cases, a memory reference affects (or kills) a memory assignment, and vice versa, if the two operations can possibly involve a common memory location.

Table 5.3.1 summarizes the rules for determining if a memory reference and a memory assignment can kill each other for each combination of reference and assignment categories. All available information is used in trying to effect as little kills as possible, since the killing operations restrict the optimization opportunities that can be unfolded. The explanations of the rules are as follows:

	(a) Direct Store	(b) Indirect Store (Range Known)	(c) Indirect Store (Range Unknown)	(d) Procedure Call
(i) Direct Load	Check Overlap	Check Overlap	Block no. of (i) $\neq$ Current Block	Level of (i) encloses Called Proc.
(ii) Indirect Load (Range Known)	Check Overlap	Check Overlap	Block no. of (ii) $\neq$ Current Block	Level of (ii) encloses Called Proc.
(iii) Indirect Load (Range Unknown)	Block no. of (a) $\neq$ Current Block	Block no. of (b) $\neq$ Current Block	Always Kill	Always Kill
(iv) Procedure Call	Level of (a) encloses Called Proc.	Level of (b) encloses Called Proc.	Always Kill	(Not Applicable)

Table 5.3.1 Rules for the killing between memory references and assignments

- When the source range of the memory reference and the target range of the assignment are known, it is only necessary to check whether the two ranges overlap. Entries (i-a), (i-b), (ii-a) and (ii-b) of the table fall under this rule.

- When either the source range of the memory reference or the target range of the assignment is unknown, the unknown range must not be from the local memory area of the current procedure. If the known range is from the local memory area, then it is certain that the two ranges do not overlap. Otherwise, it is possible that they overlap. This covers entries (i-c), (ii-c), (iii-a) and (iii-b).

- When both the source range of the memory reference and the target range of the assignment are unknown, they must both be outside the local memory area of the current procedure. No information is available to determine whether the source and target ranges overlap, so it has to be assumed that they kill each other. This covers entry (iii-c).

- When a source or target range is known, a called procedure can reference or alter a location only if the address is at a lexical level that encloses the called procedure. This fact is used in determining whether a procedure call can affect a memory reference or assignment in entries (i-d), (ii-d), (iv-a) and (iv-b).

- When a source or target range is unknown, then a procedure call is assumed to affect it.

	(a) Direct Store	(b) Indirect Store (Range Known)	(c) Indirect Store (Range Unknown)	(d) Procedure Call
(i) Direct Load	General	Equivalence	Alias	Side Effects
(ii) Indirect Load (Range Known)	Equivalence	General	Alias	Side Effects
(iii) Indirect Load (Range Unknown)	Alias	Alias	General	Side Effects
(iv) Procedure Call	Side Effects	Side Effects	Side Effects	(Not Applicable)

Table 5.3.2 Table of conditions for the occurrences of the killing relationships

This relates to entries (iii-d) and (iv-c).

Table 5.3.2 gives the circumstances that bring about the occurrences of the table entries. Entries (i-b) and (ii-a) occur when a simple variable is within the range of an array, which can only be brought about by equivalences. In entries (i-c), (ii-c), (iii-a) and (iii-b), we want to guard against the possibility that the same location is accessed both directly and indirectly, which happens in association with aliases. Killing due to procedure calls is necessary because of side effects. The other entries do not occur under specific circumstances.

The above rules apply only in the absence of inter-procedural data flow analysis. By taking into account possible candidates to be associated with the formal parameters and also the contents of called procedures, it is possible to eliminate many unnecessary kills among the memory references and assignments.

An additional data structure is used to represent the presence of the above memory references and assignments which affect data flow. Each basic block node points to a list of *kill-nodes* consisting of the u's, u_i's, d's and d_i's in their order of appearances in the code of the basic block. To determine whether an item is altered by the code of a basic block, it is only necessary to go through this list to check whether any element in the list kills the item. To determine whether a locally occurring item is anticipated at the basic block entry, it is only necessary to go

through the part of the kill-list that precedes the item in the basic block. To determine whether it is available at the block exit, the part of the kill-list that succeeds the item is used. These kill-nodes have to be updated on deletions and insertions in the course of optimization.

5.5. Effects of Procedure Integration

A procedure integrator, called PMERGE, has been implemented on U-Code at Stanford. Procedure integration is an optimization because it improves program running time by reducing the overhead in procedure calls, returns and the associated parameter passing. When invoked, PMERGE selects procedures in a program whose code is copied in-line at points at which they are called. With procedure integration, there is an associated cost in the increase in the total code size of the program. This cost does not apply for procedures that are called only once in the program.

We are mainly interested in how the procedure integrator affects the optimization performance of the global optimizer when they are used together. By invoking procedure integration as a pre-pass, the global optimization opportunities can be substantially increased, since the optimizations are performed one procedure at a time. It is expected that the total reduction in execution time will be greater than the sum of the two separate reductions when they work in isolation.

Procedure integration can bring in new global optimization opportunities in the following ways:

1. Since a procedure becomes larger, the optimizer can take into account a greater segment of code in looking for global optimization opportunities. All the optimizations performed can benefit.

2. By eliminating procedure calls, the optimizer can save the killing of many variables that arise out of the calls. Thus, computations can become available over a larger range. More redundant assignments and dead variables can be exposed. Computations can also be moved over greater distances since their movement is no longer hindered by the calls.

3. Copy propagation will dereference the parameters in the merged calls, so that more information is available when optimizing the code of the merged procedures.

4. Code in the merged procedures can be moved outside to the caller. This is especially beneficial when the merged call occurs in a loop and the merged procedure contains loop-invariant computations or strength reduction candidates.

5. The benefits of register allocation are substantially improved since the overhead of memory updates, the saving of registers before procedure calls and their re-loads after the calls can be eliminated. Registers can also be allocated over larger ranges of code that include the text of merged procedures.

The last point is particularly important in the case of common subexpressions occurring across procedure calls. Many common subexpressions can save execution time only if their values are saved and re-used in registers, because the cost of accessing main memory may exceed the cost of their re-computations. Procedure calls occurring between the common subexpressions can inhibit the use of registers to store their values, so that the full benefits of recognizing these common subexpressions cannot be derived.

There is a minor disadvantage that arises out of the use of the procedure merger with regard to optimization. When a procedure is integrated into the caller, its local variables are merged into the stack frame of the calling procedure. If the caller contains other procedure calls at some later points that cannot be merged, then these calls will prohibit the recognition of dead variables and redundant assignments in the merged procedure, which could have been recognized if the procedure is unmerged. In spite of this, the advantages of using a procedure integration pre-pass far outweigh this occasional disadvantage.

6. Performance Evaluation

In this Chapter, we study the performance of UOPT with respect to the optimizations performed and their effects on real machines. Using one machine as a main example and a set of benchmarks, the frequencies and contributions of the different optimization transformation are analyzed. The effects of some program and machine parameters on optimization performance are also examined. Then, we investigate the effects that the same machine-independent optimizations at the intermediate code level have on a variety of machines. The machines considered are the DEC 10 [Stan76], the 68000 [Moto80], the VAX [Digi81], the MIPS [Henn82c], the FOM [Bran82] and the S-1 [Hail79] [Livi83]. Using actual timing measurements, the differing improvements in the target machines are compared. We evaluate some machine characteristics and discuss how these characteristics interact with the different optimizations performed by UOPT and influence the ways that the optimizations are reflected in the underlying machine code. Finally, we give some general comments about the role played by machine-independent optimization and its relationship with all the other possible optimizations in real-world machines. Although we assume throughout that U-Code is the intermediate code, most of the remarks in this chapter also apply under more general compilation and machine-independent optimization environments.

6.1. Analysis of Optimization Performance

In this section, we study the contributions to overall performance of the different optimization phases in UOPT. A set of benchmark programs are run through the optimizer, and their optimized running times compared with their original running times. These benchmark programs are standard application programs, with minimal calls to un-optimizable external routines and runtimes. Inputs and outputs have been eliminated so that their execution is not affected by external devices. These studies are done on the DEC 10 target machine. The corresponding results for other machines are given at appropriate places to supplement the discussions.

Here is a brief description of the benchmark programs. All but the last two are in Pascal.

Perm — A program that computes permutations with recursions.

Tower — A program that solves the Tower of Hanoi problem. It is written in 120 lines of Pascal code.

Queen — A program that solves the Eight Queens problem. It contains a single recursive procedure.

Intmm — A program to compute the product of two integer matrices.

Mm — This program is identical to Intmm except that the matrices are in real numbers.

Puzzle — A compute-bound program that solves a puzzle about packing blocks into a cube. It contains 4 procedures and a main program in 160 lines of Pascal code. One of the procedures is recursive.

Quick — A program that performs the Quick Sort.

Bubble — A program that performs the Bubble Sort.

Tree — A program that performs the recursive Insertion Sort on a binary tree and checks the correctness of the insertions.

Fft — A program to perform the Fast Fourier Transformation. It is written in 250 lines of Pascal code.

Sieve — A program that compute the first n prime numbers using the Sieve of Erastosthenes. It contains only a main program with loop.

Quick2 — A second program that also performs the Quick Sort, but written in Fortran. There is no direct relation to the above Quick written in Pascal. In particular, it is not recursive.

Inverse — A program written in Fortran that computes the inverse of a matrix and verifies the result by multiplying back to form the unit matrix.

Table 6.1.1 shows the improvement in the running times of these benchmark programs on the DEC 10 using only PMERGE, only UOPT and a combination of the two. Some of the programs do not have procedures that can be integrated. Procedure integration is especially effective in reducing execution times in programs Perm, Tower, Bubble and Tree. In Perm and Tree, where the programs consist of mainly short procedures and numerous procedure calls, global optimization is not effective without procedure integration. The improvement in execution times shown in row 3 always exceeds the product of the improvement shown in rows 1 and 2.

The optimization in Mm is not as good as that in Intmm because constant arithmetic, linear function test replacement and strength reduction are not performed on real numbers, and the floating point operations have greater dominance of the running time.

6.1.1. Analysis by Statistical Counts

To analyze the usefulness of each optimization transformation, we have specifically set up, in UOPT, counts of the number of instances that each transformation is performed in the course of optimizing each program. Table 6.1.2 shows these statistics for the versions of the programs that have been procedure-integrated. Although the data shown are those for DEC 10 U-Code, they do not vary widely among different target machines. Due to the way we perform global optimizations, it is not possible to separate out the different kinds of optimizations in the way

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle	Quick
0. Original running time	13.77 (1.0)	2.48 (1.0)	3.95 (1.0)	1.30 (1.0)	1.43 (1.0)	5.32 (1.0)	1.94 (1.0)
1. Time using only Pmerge	9.62 (.70)	1.68 (.68)	3.95 (1.0)	1.29 (.99)	1.42 (.99)	5.22 (.98)	1.60 (.82)
2. Time using only Uopt	12.40 (.90)	2.10 (.84)	2.68 (.68)	.46 (.35)	.59 (.41)	2.58 (.49)	1.739 (.90)
3. Time using Pmerge and Uopt	7.44 (.54)	1.26 (.51)	2.68 (.68)	.42 (.32)	.56 (.39)	2.47 (.46)	1.30 (.67)

Program	Bubble	Tree	Fft	Sieve	Quick2	Inverse	Average
0. Original running time	5.06 (1.0)	1.22 (1.0)	2.85 (1.0)	5.09 (1.0)	.719 (1.0)	4.71 (1.0)	(1.0)
1. Time using only Pmerge	3.69 (.73)	1.01 (.83)	2.85 (1.0)	5.09 (1.0)	.719 (1.0)	4.71 (1.0)	(.90)
2. Time using only Uopt	3.80 (.75)	1.15 (.94)	1.08 (.38)	3.25 (.64)	.572 (.80)	2.35 (.50)	(.65)
3. Time using Pmerge and Uopt	2.34 (.46)	.93 (.77)	1.05 (.37)	3.25 (.64)	.572 (.80)	2.35 (.50)	(.55)

Running times in Seconds

(Normalized running times in parentheses)

Table 6.1.1 Optimized and un-optimized running times

they are generally visualized. The number of instances of code motion can be approximated as the number of insertions (row 5). However, these insertions are not only due to loop-invariant code motion, but to partial redundancy suppression as well. The number of redundant expressions can be taken as the number of deletions (row 6), but the deletions actually include those made redundant after the insertions. Also, we cannot directly count the number of strength reductions since they are performed as part of code motion. These same comments apply to estimating the number of optimizations related to stores.

From Table 6.1.2, it can be seen that, with the exceptions of local redundant assignment elimination (row 2) and linear function test replacement (row 7), all the optimization transformations occur quite frequently. Especially important are local and global common subexpressions, code motion and constant expression computation. Most of the constant expressions come from address collapsing in array offset computations. Common subexpressions, code motion and induction expressions also frequently occur in association with address expressions. Copy

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle	Quick
1. # of local common subexpr.	8	30	3	14	14	22	19
2. # of locally redundant assignments	0	0	0	0	0	1	0
3. # of constant arith.	4	11	13	21	21	83	21
4. # of global copy propagations	2	15	0	5	5	18	30
5. # of backward code motion insertions	4	10	6	20	21	42	5
6. # of backward code motion deletions	8	21	16	22	23	51	80
7. # of test replacements	2	0	1	4	4	2	0
8. # of globally redundant assignments	7	22	2	5	5	11	14
9. # of forward code motion insertions	0	4	2	0	0	3	4

Program	Bubble	Tree	Fft	Sieve	Quick2	Inverse	Total
1. # of local common subexpr.	4	3	92	2	4	15	230
2. # of locally redundant assignments	0	0	0	0	2	0	3
3. # of constant arith.	8	1	51	2	27	20	283
4. # of global copy propagation	3	18	18	1	1	1	117
5. # of backward code motion insertions	7	1	15	2	25	17	175
6. # of backward code motion deletions	11	15	18	4	27	25	321
7. # of test replacements	1	1	2	1	1	0	19
8. # of globally redundant assignments	4	11	17	1	1	1	101
9. # of forward code motion insertions	1	0	0	0	8	1	23

Table 6.1.2 Optimization statistics

propagation often occurs with the parameters of procedures that have been integrated into the callers. There is a strong correlation between the number of redundant assignments (row 8) and the number of copy propagations, since the latter transformation often gives rise to non-live

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle	Quick
% of var. references in registers	.65	.40	.76	.95	.95	.94	.67
% of var. assignments in registers	.70	.23	.72	.96	.96	.77	.77

Program	Bubble	Tree	Fft	Sieve	Quick2	Inverse	Average
% of var. references in registers	.91	.78	.87	.87	.62	.71	.77
% of var. assignments in registers	.92	.74	.80	.89	.62	.75	.76

Table 6.1.3(a) Static register allocation statistics in the DEC 10

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle	Quick
% of var. references in registers	.94	.72	.90	.96	.96	.95	.80
% of var. assignments in registers	.95	.58	1.0	.95	.95	.77	.80

Program	Bubble	Tree	Fft	Sieve	Quick2	Inverse	Average
% of var. references in registers	.90	.79	.93	.86	.74	.88	.87
% of var. assignments in registers	.91	.80	.83	.88	.77	.94	.86

Table 6.1.3(b) Static register allocation statistics in the 68000

variables.

Table 6.1.3 displays the register allocation statistics for the benchmark programs. It shows the percentages of variable references and the percentages of variable assignments that are in registers. The data are obtained by static counts in the optimized programs. The dynamic counts are expected to be better, since the register allocator in UOPT takes loop-nesting depths into account. Since all the program procedures are fairly small, the data may not be typical of those obtained in large procedures.

The DEC 10 uses the caller-save linkage convention, and the DEC 10 code generator allows UOPT to allocate up to 9 registers. Most of the programs do not use up all the registers. It

is the nature of the programs that dictates the percentages of variables allocated. Programs that have many procedure calls (e.g. Tower) tend to diminish the percentage allocated because the numerous instances of register saves and re-loads around procedure calls tend to increase the cost of the allocations. These calls are frequently standard function calls that cannot be merged.

The register allocation statistics for the 68000 is markedly different from that for the DEC 10, which is due to the use of the callee-save linkage convention in the 68000. The percentages of variable accesses allocated in registers in the 68000 are always greater than those in the DEC 10, since register saves and re-loads do not occur around procedure calls, so that the cost of allocating to registers does not increase due to procedure calls. Tables 6.1.3(a) and (b) show that the linkage convention concerning the handling of registers does affect register allocation. The 68000 code generator allows UOPT to use up to 6 data registers and 4 address registers, out of the 8 data registers and 8 address registers available.

6.1.2. Analysis by Partial Optimization

Another method we can use to study the effectiveness of individual optimizations is by applying each optimization separately and studying the resulting improvement in running times. It is also possible to get some ideas about the degree of correlation between the different optimizations by studying by how much the improvement from the completely optimized versions of the benchmarks exceeds the sum of the improvement from the partially optimized versions. Partial optimization is possible in UOPT according to the phase structure of the optimization process (Section 5.1.3). UOPT allows the user to control the extents of optimization by specifying options in his programs. In the following, we study the different degrees of improvement in program running times due to the selective applications of the various optimization phases.

Table 6.1.4 displays the running times of the benchmark programs on the DEC 10 for varying degrees of global optimization. Shown in row 0 are the running times for the un-optimized procedure-integrated versions. Row 1 shows the times when all the global optimization phases have been applied. Row 2 shows the running times with only local optimizations (phases 1, 3 and 11). Row 3 shows the running times with only local optimizations and register allocation (phases 1, 3, 10, 11). Row 4 shows the times when copy propagation (phase 4) is left out. Row 5 shows the times when backward code motion, redundant expression elimination and strength reduction (phase 6) are left out. Row 6 shows the times when no store optimization is performed, in which phases 5, 8 and 9 are left out. The last row shows the optimized running times when no register allocation (phase 10) is performed. The average column in the table shows that backward code motion and register allocation are the optimizations that reduce running time

the most. Next are store optimizations and copy propagation. Local optimization can only reduce running time by 5% on the average.

We now look more closely at the data for the individual programs: In all the programs, the times shown in rows 4 and 6 are always worse than the times shown in row 1. This shows that copy propagation and store optimizations always result in improvement in execution time, although the effect is not as substantial for copy propagation. Copy propagation is important in the program Tower, where there is a 14% deterioration in the optimized execution time when copy propagation is not performed.

Backward code motion is important in most of the programs. Comparing row 5 with row 1, it can be seen that the backward code motion phase is mainly responsible for the large running time improvement in Queen, Intmm, Mm, Puzzle, Fft and Inverse. In Perm, Tower and Tree, there are not many opportunities for code motion, and the numerous procedure calls tend to inhibit the saving of common subexpressions in registers. In contrast to the DEC 10, procedure calls do not affect register allocation to common subexpressions in the 68000, which explains why Table 6.1.5 shows that backward code motion always decreases the running time in the 68000. For Bubble and Quick2 running on the DEC 10, backward code motion actually has a negative effect on the optimization results. This is because most of the common subexpressions and induction expressions in these two programs are simple address expressions that can be collapsed into single instructions using special operand addressing modes in the DEC 10. The use of special operand addressing modes is facilitated when array indices have been allocated in registers, so that the common subexpressions are not really beneficial. In addition, there is an overhead in the saving and re-loading of these expressions. In the case of strength reduction, there is the additional overhead of incrementing the induction expressions every time through the loop. The effect of induction expression optimization is not pronounced when the induction expression does not involve multiplication, and the target machine can address operands using the indexed addressing mode. The good and bad effects of this backward code motion phase exist in all programs, and not necessarily all machines. It is our belief that any non-beneficial effect is marginal, but the gain is substantial enough in common programs to justify the use of this optimization phase in all machines. Appendix E contains the unoptimized and optimized object code for the inner loop of Bubble across a variety of machines.

Local optimization (phase 1) represents the minimal optimization that the user may specify when he invokes UOPT. Local optimization is most effect in Fft, where there are many array references and fields within arrays. In Perm, Queen, Puzzle, Bubble, Tree and Sieve, the resulting running times are worse. However, if register allocation is added (row 3), the running times are substantially improved. In fact, in Perm, Quick, Bubble, Tree, Sieve and Quick2, the

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle	Quick
0. No optimization†	9.62 (1.0)	1.68 (1.0)	3.95 (1.0)	1.29 (1.0)	1.42 (1.0)	5.22 (1.0)	1.60 (1.0)
1. Full global optimization	7.44 (.77)	1.27 (.75)	2.67 (.68)	.42 (.33)	.55 (.38)	2.47 (.47)	1.30 (.70)
2. Only local optimizations	10.92 (1.14)	1.68 (1.0)	4.22 (1.07)	1.10 (.85)	1.23 (.87)	5.24 (1.0)	1.42 (.89)
3. Only local optimizations, reg. alloc.	8.46 (.88)	1.39 (.83)	3.99 (1.01)	1.05 (.81)	1.19 (.84)	4.85 (.93)	1.25 (.78)
4. All except copy propagation	7.44 (.77)	1.44 (.86)	2.68 (.68)	.43 (.33)	.56 (.39)	2.46 (.47)	1.37 (.86)
5. All except backward code motion	8.00 (.83)	1.31 (.78)	3.99 (1.01)	1.22 (.95)	1.35 (.95)	4.83 (.93)	1.45 (.91)
6. All except store optimizations	8.94 (.93)	1.37 (.82)	2.68 (.68)	.43 (.33)	.56 (.39)	2.52 (.48)	1.36 (.85)
7. All except register alloc.	8.87 (.92)	1.36 (.81)	3.76 (.95)	.65 (.50)	.78 (.55)	3.74 (.72)	1.60 (1.0)

Program	Bubble	Tree	Fft	Sieve	Quick2	Inverse	Average	Cost
0. No optimization†	3.69 (1.0)	1.01 (1.0)	2.85 (1.0)	5.09 (1.0)	.719 (1.0)	4.71 (1.0)	(1.0)	0%
1. Full global optimization	2.33 (.63)	.93 (.93)	1.07 (.37)	3.52 (.69)	.572 (.80)	2.36 (.50)	(.61)	100%
2. Only local optimizations	3.79 (1.03)	1.05 (1.04)	1.82 (.64)	5.22 (1.03)	.703 (.98)	3.89 (.83)	(.95)	15%
3. Only local optimizations, reg. alloc.	2.04 (.55)	.91 (.90)	1.68 (.59)	3.30 (.65)	.487 (.68)	3.67 (.78)	(.79)	37%
4. All except copy propagation	2.34 (.63)	.91 (.90)	1.10 (.39)	3.57 (.70)	.572 (.80)	2.35 (.80)	(.65)	94%
5. All except backward code motion	1.98 (.54)	.87 (.86)	2.85 (1.0)	4.10 (.81)	.497 (.69)	3.67 (.78)	(.85)	87%
6. All except store optimizations	2.53 (.68)	.97 (.96)	1.07 (.37)	3.57 (.70)	.588 (.82)	2.37 (.50)	(.65)	83%
7. All except register alloc.	4.60 (1.25)	1.08 (1.07)	1.40 (.59)	5.86 (1.15)	.807 (1.12)	3.17 (.67)	(.87)	77%

† The times in this row correspond to the times in row 1 of Table 6.1.1

Running times in Seconds

(Ratio to un-optimized running times in parentheses)

Cost (last column) in % running time of full optimization by UOPT

Table 6.1.4 Running times for various extents of optimization (DEC 10)

Program	Perm	Tower	Queen	Intmm	Puzzle	Bubble	Tree	Sieve
No optimization	32.30 (1.0)	5.85 (1.0)	12.58 (1.0)	17.12 (1.0)	16.39 (1.0)	18.85 (1.0)	9.77 (1.0)	23.90. (1.0)
Full global optimization	27.90 (.86)	3.95 (.68)	6.54 (.52)	7.56 (.44)	6.33 (.38)	5.74 (.30)	9.44 (.97)	10.42 (.44)
All except backward code motion	28.80 (.89)	4.24 (.72)	8.93 (.71)	17.83 (1.04)	10.57 (.64)	10.05 (.53)	9.57 (.98)	13.91 (.60)

Running times in Seconds

(Ratio to un-optimized running times in parentheses)

Table 6.1.5 Effectiveness of backward code motion on the 68000

running times after only local optimization and register allocation approach or exceed the times after full optimization. The optimization cost in this case is only 37% of full optimization. Thus, it can be said that local optimization followed by register allocation is the most cost-efficient optimization choice if the user wants to compromise the needed performance of his programs with the associated optimization running-time cost.

Row 7 shows that, in order to bring across the full benefits of the various global optimizations, register allocation is a required concluding phase of the optimizations. Without register allocation, the programs Bubble, Tree, Sieve and Quick2 are worse in spite of all the global optimizations. Even more instructive is comparing the differences in improvement that row 3 has over row 2 and row 1 has over row 7. Rows 2 and 3 show between them the effects of adding the register allocation phase if the optimizer performs only the minimal local optimizations. Rows 1 and 7 show between them the effects of leaving out the register allocation phase when the optimizer performs its full set of optimization. In the average column, row 3 is .16 less than row 2, and row 1 is .26 less than row 7. This means that register allocation is a lot more effective when the optimizer performs other global optimizations. Without register allocation, the benefits of the other global optimizations cannot be fully exposed, because the cost of saving intermediate quantities in main memory is high enough in some cases to cancel out the benefits that can be derived from the optimizations.

The programs Quick and Sieve present an additional observation. In these two programs, copy propagation, backward code motion and store optimizations are all beneficial phases, since the running time is worse when each of them is left out (comparing row 1 with rows 4, 5 and 6 respectively). However, when all these three kinds of optimization are not performed, as is in row 3, the resulting running times are better instead. This means that these optimizations build

on each other. The benefits of a set of transformations can often be augmented if preceded or followed by other transformations. Thus, it is important not to leave out any of these phases when carrying out global optimization.

6.2. Effects of Optimization Parameters

In this section, we study the variation in optimization performance due to some parameters that influence optimization. The observations are explained and, in some cases, inferences are made regarding optimization in general. The studies are also done using the DEC 10 as the target machine.

6.2.1. Number of Registers Available the Optimizer

The DEC 10 has 14 general-purpose registers that can be used in code generation. Of these, the code generator set aside 9 registers for use by UOPT in allocating to program variables. The remaining registers are used by the code generator in generating machine instructions. We investigated the effects of allowing different numbers of registers to be allocated by UOPT. The results displayed in Table 6.2.1 show that the optimized running times always improve when a larger number of registers are available to UOPT. The 5 registers used by the code generator is enough for most practical purposes, and increasing the number for the code generator (i.e. decreasing the number used by UOPT) does not cause appreciable improvement in execution speed. Thus, we see that global machine-independent register allocation is an extremely useful optimization.

Another observation in Table 6.2.1 is that different programs require different numbers of registers for optimal register allocation. In the programs Perm, Tower and Tree, 4 registers seem to be all that are needed; for others, increasing the number further yields better execution speeds. In the programs Puzzle and Sieve, just 2 registers can dramatically improve the program running time. Different programs have different cut-off points regarding the number of registers they need for optimal register allocation. The cut-off number of registers required is related to the *chromatic numbers* of the interference graphs — the numbers of colors needed to color the graphs.

Combining the data of Table 6.2.1 with those in Table 6.1.3(a), in which 9 registers are used, we notice that in the programs Perm, Tower and Tree in which the percentages of variable accesses in registers are not high, the programs have not run out of registers. These programs actually have a large number of variable accesses that should not be put into registers. This supports our original conviction that the best allocation for some architectures is not necessarily the one that puts all variables into registers.

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle	Quick
0 registers	8.87 (1.0)	1.36 (1.0)	3.76 (1.0)	.65 (1.0)	.78 (1.0)	3.74 (1.0)	1.60 (1.0)
2 registers	8.25 (.93)	1.28 (.94)	3.62 (.96)	.64 (.98)	.78 (.99)	2.56 (.68)	1.48 (.93)
4 registers	7.44 (.84)	1.28 (.94)	3.29 (.88)	.58 (.89)	.71 (.91)	2.54 (.68)	1.42 (.89)
6 registers	7.44 (.84)	1.27 (.93)	2.68 (.71)	.43 (.66)	.56 (.72)	2.54 (.68)	1.42 (.89)
All 9 registers	7.44 (.84)	1.26 (.92)	2.68 (.71)	.42 (.65)	.55 (.71)	2.47 (.68)	1.30 (.81)

Program	Bubble	Tree	Fft	Sieve	Quick2	Inverse	Average
0 registers	4.60 (1.0)	1.08 (1.0)	1.40 (1.0)	5.86 (1.0)	.807 (1.0)	3.17 (1.0)	(1.0)
2 registers	3.71 (.81)	.96 (.89)	1.24 (.89)	4.02 (.69)	.724 (.90)	2.89 (.91)	(.88)
4 registers	3.84 (.83)	.93 (.86)	1.14 (.81)	3.99 (.68)	.669 (.83)	2.75 (.87)	(.84)
6 registers	2.33 (.51)	.93 (.86)	1.09 (.78)	3.53 (.60)	.621 (.77)	2.40 (.76)	(.75)
All 9 registers	2.33 (.51)	.93 (.86)	1.05 (.75)	3.25 (.55)	.572 (.71)	2.35 (.74)	(.73)

Running times in Seconds

(Normalized running times in parentheses)

Table 6.2.1 Effects of the number of registers available for register allocation (DEC 10)

6.2.2. Changing the Register Move-Cost

In Section 4.3, we discussed the cost and saving estimates that determine whether a variable should reside in a register. `MOVCOST` is the cost of a transfer operation between register and memory. `LODSAVE` and `STRSAVE` are the amounts of execution time saved for each reference and assignment of a variable respectively, due to the variable being in register at the time. The best values to use for these parameters vary among target machines. They are dependent on machine architectures and instruction characteristics.

Only the ratios of `MOVCOST` to `LODSAVE` and `STRSAVE` are important. We take both `LODSAVE` and `STRSAVE` to be 1. The value of `MOVCOST` is then a parameter in UOPT that can be set by the user in his program. When `MOVCOST` is 0, it implies that no execution time is sacrificed in

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle	Quick
MOVCOST = 0	7.67	1.21	2.61	.42	.55	2.53	1.49
MOVCOST = 1.0	7.44	1.24	2.58	.42	.55	2.48	1.30
MOVCOST = 1.5	7.44	1.27	2.67	.42	.55	2.47	1.30
MOVCOST = 2.0	7.44	1.28	3.61	.42	.55	2.47	1.29
MOVCOST = 3.0	7.55	1.28	3.61	.42	.55	2.49	1.29
MOVCOST = 4.0	8.08	1.28	3.61	.42	.55	2.49	1.28

Program	Bubble	Tree	Fft	Sieve	Quick2	Inverse
MOVCOST = 0	2.33	.94	1.07	3.25	.622	2.36
MOVCOST = 1.0	2.33	.93	1.07	3.25	.579	2.36
MOVCOST = 1.5	2.33	.93	1.07	3.25	.572	2.36
MOVCOST = 2.0	2.33	.93	1.07	3.24	.573	2.36
MOVCOST = 3.0	2.33	.93	1.07	3.24	.572	2.36
MOVCOST = 4.0	2.33	.95	1.07	3.24	.572	2.36

(Time shown is in Seconds of CPU time)

Table 6.2.2 Effects of the value of MOVCOST to optimized running times (DEC 10)

register-memory transfers. Since no cost is involved, UOPT will allocate as many variables in registers until all the registers are used up. Such a value of MOVCOST does not befit any machine in the real world. When MOVCOST is 1, it implies that, in the target machine, arithmetic and logic operations can only be performed on registers. If any computation involves a memory item as an operand, the item must first be brought into a register by a separate memory transfer instruction. The memory target to receive the value of a computation also has be stored into by a separate instruction. When MOVCOST is vary large, it means in the limiting case that the target machine can access memory as fast as it accesses the registers. This happens when the machine contains no fast memory elements, and all computations directly reference operands in memory (memory-to-memory architecture). In this case, UOPT will not allocate anything in register due

to the large value of MOVCOST. Thus, it can be seen that MOVCOST and the related LODSAVE and STRSAVE are indispensible parameters in the context of machine-independent register allocation.

Since the value of MOVCOST is machine-dependent, for each target machine, there must be an optimal value of this parameter at which the optimizer will yield the best register allocation. We studied the effects that varying the value of MOVCOST has on the optimized running times of the benchmark programs on the DEC 10. The results are tabulated in Table 6.2.2. The occurrences of the minimal running times in the table empirically determine MOVCOST.

From the table, it can be seen that the value of MOVCOST at which the optimized running times are best also vary among individual programs. This is because each program has different occurrence counts of individual machine instructions and addressing modes, which exhibit different fetch times. Also, register allocation can introduce an added degree of flexibility to the instruction selection process of the code generators that also affects the execution time.

The optimized running times displayed in Table 6.2.2 also have different degrees of dependence on the value of MOVCOST. The running times of Intmm, Mm, Bubble, Fft, Sieve and Inverse are somewhat unaffected by the variation in the value of MOVCOST, whereas Perm, Tower, Queen, Quick and Quick2 show higher dependence. This degree of dependence on MOVCOST is based on many factors. Programs that have only a few number of register-memory transfer operations (RLOD's and RSTR's), or whose such instructions are not nested inside loops, are relatively independent of the value of MOVCOST. This is because in our algorithm the cost of register allocation represented by MOVCOST arises directly from the RLOD and RSTR instructions. There are also different degrees of clustering of occurrences of the same variable. When a variable occurs very frequently in a block, the saving out of allocating the variable in register is great; MOVCOST will have to be made very large for UOPT to decide not to allocate the variable in register. When the target machine has many registers available for use by the optimizer, the results displayed in the row MOVCOST = 0 will worsen because the optimizer will allocate many items in registers even though their allocation is not profitable in terms of execution time.

In Table 6.2.2, the program Perm shows the best optimization when MOVCOST is 2; Queen shows the best time when MOVCOST is 1; Puzzle, Tree and Quick2 shows the best times when MOVCOST is from 1.5 to 2. We conclude that, for the DEC 10, the best value of MOVCOST is in the region 1.5 to 2. We have set MOVCOST to be 1.5 in the production optimizer.

6.2.3. Effects of Bounds Checking

Table 6.2.3 compares the optimization performance for versions of the programs with and without bounds-checking. Programs which have bounds-checking contain extra code that checks whether the ranges of subrange types or array subscripts are ever exceeded. Bounds-checked

versions always take longer to run than the corresponding versions without bounds-checking. UOPT does not perform any specific optimization on bounds-checking. We are comparing the percentage improvement that is achieved with respect to their un-optimized versions. The improvement shown in the table includes the effects of procedure integration.

The results show that programs without bounds-checking can be optimized more than the corresponding versions with bounds-checking. This is due to the fact that the bounds-checking instructions (CHKL, CHKR) cause changes in the tree structures of expressions that prohibit tree-restructuring in stack-height reduction. Address collapsing and strength reduction are affected, since they cannot easily be performed across a bounds-checked expression subtree. Bounds-checking also reduces the number of common subexpressions, since two expressions are the same only if their bounds-checking code is identical. It is possible to incorporate an extra bounds-checking optimization phase to further extend the optimization capability of UOPT.

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle	Quick
Unoptimized, without bounds-checks	13.77 (1.0)	2.48 (1.0)	3.95 (1.0)	1.30 (1.0)	1.43 (1.0)	5.32 (1.0)	1.94 (1.0)
Optimized, without bounds-checks	7.44 (.54)	1.26 (.51)	2.68 (.68)	.424 (.32)	.56 (.39)	2.47 (.46)	1.30 (.67)
Unoptimized, with bounds-checks	19.37 (1.0)	3.40 (1.0)	5.24 (1.0)	1.64 (1.0)	1.78 (1.0)	7.01 (1.0)	2.75 (1.0)
Optimized, with bounds-checks	12.36 (.64)	2.18 (.64)	4.86 (.93)	.86 (.53)	1.00 (.56)	4.28 (.61)	2.40 (.87)

Program	Bubble	Tree	Fft	Sieve	Quick2	Inverse	Average
Unoptimized, without bounds-checks	5.06 (1.0)	1.22 (1.0)	2.85 (1.0)	5.09 (1.0)	.719 (1.0)	4.71 (1.0)	(1.0)
Optimized, without bounds-checks	2.34 (.46)	.93 (.77)	1.05 (.37)	3.25 (.64)	.572 (.80)	2.35 (.50)	(.55)
Unoptimized, with bounds-checks	7.46 (1.0)	1.36 (1.0)	3.49 (1.0)	6.49 (1.0)	.924 (1.0)	6.51 (1.0)	(1.0)
Optimized, with bounds-checks	4.66 (.63)	1.11 (.81)	1.68 (.48)	5.21 (.80)	.789 (.85)	4.05 (.62)	(.69)

Running times in Seconds

(Normalized running times in parentheses)

Table 6.2.3 Comparison of optimization for versions with and without bounds-checking (DEC 10)

6.3. Characterization of Machines

In this section, we look at the machine characteristics that influence the ways the machines can benefit from the machine-independent optimizations we have addressed in the previous chapters. We are mostly concerned with the instruction sets, since they have the most to do with optimizability at the program code level. In Section 6.6, we shall summarize our findings about the relationships between the various machine-independent optimizations and machine characteristics.

Number of Addresses in Instructions

Most arithmetic operations reference two operands and yield a result. There are different ways in which machine instructions can specify these addresses:

1. Three-address instructions: This instruction format completely specifies the two operands and the address where the result of the operation is stored.

2. Two-address instructions: The two addresses specify the two operands in the case of binary operations, or the source and target in the case of data move operations. The result of an arithmetic operation is always left in one of the two addresses.

3. One-address instructions: Arithmetic operations are always carried out on a single register or accumulator. The results are always left on the accumulator. Since there is only one possible accumulator, the instructions do not need to specify it explicitly. They only specify the second operand in the case of binary operations, or the load and store targets in the case of transfers to and from the accumulator.

Addressing Modes

Operands can be specified in different ways in machine instructions:

1. Immediate addressing: The operand, which is a constant, is directly specified in the instruction.

2. Direct addressing: The instruction provides the absolute address of the operand in memory. A special case is register direct addressing, in which a register contains the operand.

3. Indirect addressing: The instruction gives the address of a memory location that in turn provides the address of the actual operand. A special case is register indirect addressing, in which the instruction selects a register that contains the address of the operand. Another variation of indirect addressing is indirect with autoincrement or autodecrement, in which the location containing the address is automatically incremented or decremented after or before the operand fetch.

4. Indexed addressing: The instruction specifies an offset and an index register. The address of the operand is found by adding the offset to the content of the index register. The actual base address can be either the offset or the content of the index register. When the base address is contained in a register, it is termed base addressing which can be used to implement program relocatability, for addressing within an activation record using a stack-frame pointer or in accessing array reference parameters.

On top of the number of addresses and the possible addressing modes for each field in the instructions, numerous restrictions or idiosyncracies may be present. This concerns the *orthogonality* of the instruction set. A machine with an orthogonal (symmetric or regular) instruction set provides uniform addressing capability for all op-codes. A machine with a non-orthogonal instruction set has different restrictions on addressing modes among the op-codes and the fields in each instruction. For each addressing mode, there can be other restrictions as well, such as limitation to a subset of the registers and the size of the constant or address specified.

An attribute often used to describe machines is the complexity of the instruction set, which has to do with the number and types of instructions provided and the lengths of the instructions. Complicated instruction sets often exist in machines that provide powerful operations and addressing modes, which require multiple instruction word lengths for their complete specifications (e.g. S–1). Reduced instruction-set computers (RISC's) have only a limited number of instructions that execute in single clock cycles and are of the same word size.

An important consideration regarding machine instruction sets is whether each address field in the instruction can address memory. The common situation is that not all the address fields can address memory, regardless of the number of addresses in the instruction. In such machines, individual arithmetic operations usually involve multiple instructions, the extra instructions being for transfering memory operands to registers. In machines with simple instruction sets, memory access is usually restricted only to the load and store commands (e.g. MIPS).

A additional attribute used to qualify machines is the level of the machine code. When many non-primitive operations are provided by the instruction set, the level of the instruction code is high. A special type of machines, the directly-executable language (DEL) processors, directly map language constructs into hardware. These machines are hardware interpreters for the source language statements. An example is the Fortran Optimized Machine (FOM) at IBM.

6.4. Optimization Results in Different Machines

Program	Perm	Tower	Queen	Intmm	Mm	Puzzle
Original DEC 10 running time	13.77 (1.0)	2.48 (1.0)	3.95 (1.0)	1.30 (1.0)	1.43 (1.0)	5.32 (1.0)
Optimized DEC 10 running time	7.44 (.54)	1.26 (.51)	2.68 (.68)	.424 (.32)	.56 (.39)	2.47 (.46)
Original 68000 running time	36.52 (1.0)	6.59 (1.0)	12.58 (1.0)	17.12 (1.0)	–†	16.56 (1.0)
Optimized 68000 running time	27.90 (.76)	3.95 (.60)	6.54 (.52)	7.56 (.44)	–†	6.15 (.37)
Original VAX running time	46.93 (1.0)	7.45 (1.0)	10.02 (1.0)	1.88 (1.0)	1.97 (1.0)	10.28 (1.0)
Optimized VAX running time	25.34 (.54)	3.42 (.46)	8.58 (.86)	.58 (.31)	.80 (.41)	4.53 (.44)
Original MIPS running time‡	(1.0)	(1.0)	(1.0)	(1.0)	–†	(1.0)
Optimized MIPS running time‡	(.52)	(.35)	(.46)	(.41)	–†	(.24)

Program	Quick	Bubble	Tree	Fft	Sieve	Average
Original DEC 10 running time	1.94 (1.0)	5.06 (1.0)	1.22 (1.0)	2.85 (1.0)	5.09 (1.0)	(1.0)
Optimized DEC 10 running time	1.30 (.67)	2.34 (.46)	.93 (.77)	1.05 (.37)	3.25 (.64)	(.53)
Original 68000 running time	23.59 (1.0)	20.54 (1.0)	10.47 (1.0)	–†	23.90 (1.0)	(1.0)
Optimized 68000 running time	17.75 (.75)	5.74 (.28)	9.44 (.90)	–†	10.42 (.44)	(.56)
Original VAX running time	6.07 (1.0)	16.23 (1.0)	9.32 (1.0)	4.32 (1.0)	13.60 (1.0)	(1.0)
Optimized VAX running time	4.17 (.69)	4.98 (.31)	8.70 (.94)	1.75 (.40)	8.32 (.61)	(.54)
Original MIPS running time‡	(1.0)	(1.0)	(1.0)	–†	(1.0)	(1.0)
Optimized MIPS running time‡	(.52)	(.33)	(.71)	–†	(.47)	(.45)

† Floating point instructions not yet available for running these programs.

‡ Real running times not available; programs are run using a simulator.

Running times in Seconds

(Normalized running times in parentheses)

Table 6.4.1 Optimization performance on different machines

To this date, the optimization output of UOPT has been used on 6 different machines — the DEC 10, the 68000, the VAX, the MIPS, the S–1 and the FOM. The code generators for these machines are all implemented at Stanford. The S–1 and the FOM are not capable of running real programs yet. In Table 6.4.1, we present the optimization results for the preceding benchmarks on the DEC 10, the 68000, the VAX and the MIPS.

Table 6.4.1 compares the original running times of the programs with their running times after procedure integration and global optimization. The data for the MIPS are based on counts of the number of instructions executed on the MIPS simulator. The 68000 currently implements multiplication using subroutines, and this may influence the comparison since more time is spent in performing multiplication. Also, the 68000 uses 32 bits for all non-boolean data even though it is not a true 32-bit machine, and the extra running time due to the use of 32-bit arithmetic cannot be optimized. Apart from the 68000, which uses the callee-save convention, and the FOM, which does not have conventional registers, all the machines use the caller-save linkage convention. All the code generators are implemented by different persons, so that there is a variety of code generating methodology used. The code generators perform machine-dependent peephole optimization, and the peepholing may duplicate some of the local optimizations performed in UOPT. The timing data for the unoptimized versions of the programs in the table include the effects of the machine-dependent optimization. Since the quality of the translated object code for a machine is highly dependent on the code generator, it is possible that a different code generator for the same machine may yield very different results in the Table 6.4.1.

6.5. Effects of the Optimizations on Machine Code

In this section, we look at the different types of optimizations performed in UOPT and consider how these optimizations at the intermediate code level can bring about differing effects on the object code of target machines. Since UOPT uses U-Code as the optimization medium, a machine that closely resembles the hypothetical U-Code machine is expected to exhibit the most direct and predictable benefits from the optimizations performed. The case in point is a stack machine whose instruction set closely resembles U-Code. We are mainly interested in how optimization in U-Code influences the object code of machines with other characteristics.

Among the optimizations performed, those that shorten code sequences will yield noticeable improvements in all machines, since the translated machine code will correspondingly be shortened. Thus, it can be certain that dead code elimination, redundant store and dead variable elimination are always beneficial; these optimizations result in the removal of useless code. Constant expression evaluation replaces a sequence of arithmetic operations by a single constant.

Since this cuts the code size of the computation to a fraction of its original length, the benefits of this optimization are also independent of the characteristics of the target machines.

Another class of optimizations moves program code from frequently executed regions of the program to less frequently executed regions. The transformation involves little or no change to the forms of the moved code, and thus their effects on the real machines are also independent of the machine characteristics. These optimizations include the various forms of code motion, which are related to either loop invariant expressions or partial redundancy suppression.

Next we consider the remaining types of optimizations whose effects on the underlying machines are not as obvious those just considered:

Constant propagation

Constant propagation replaces a memory operand by a constant. This allows the use of immediate addressing in the machine instruction, and saves the target machine a memory cycle to access the content of the memory location originally referenced. This does not necessarily result in the elimination of any machine instruction. However, if the constant is small, the immediate address occupies less space in the instruction. In two- or three-address machines in which only one operand can address memory, this can allow the code generator to squeeze the specification of an arithmetic operation into a single instruction.

Example. For the statement "I := I + J" where J is folded to 3, in 68000 code,

```
movl    pp$dat+580,d0          load J
addl    d0,pp$dat+576          add to I
```

can be reduced to:

```
addql   #3,pp$dat+576          add 3 to I
```

In the MIPS, an instruction to load a constant is also eliminated because the add instruction cannot address memory but can take an immediate operand:

```
ld FPinit+(-3),r1.             load J
ld FPinit+(-4),r2              load I
add r2,r1                      I + J
st r1,FPinit+(-4)              store to I
```

is reduced to:

```
ld FPinit+(-4),r1              load I
add #3,r1                      add 3 to I
st r1,FPinit+(-4)              store to I
```

In the DEC 10, however, there is no change in the number of instructions:

```
MOVE    2 ,PP$DAT+87           load I
ADD     2 ,PP$DAT+88           I + J
MOVEM   2 ,PP$DAT+87           store to I
```

is transformed to:

```
MOVE   2 ,PP$DAT+87          load I
ADDI   2 ,3                  I + 3
MOVEM  2 ,PP$DAT+87          store to I
```

□

Stack height reduction

Stack height reduction affects the target machine code in two different aspects:

1. The U-Code stack is usually implemented using general-purpose registers in the underlying machines. Stack height reduction reduces the number of registers needed to hold the items on the stack, thus freeing registers for other usages and reducing the chance that the code generators run out of registers, when spilling to main memory occurs with the associated spill code. When an item on the stack is an intermediate result of an earlier computation, a temporary register is always needed to keep its value. When the item on the stack is a variable, however, depending on the target machines, it may or may not need to reside in a register before being combined in the subsequent evaluation, since appropriate addressing modes may allow the arithmetic instruction to address one or both operands directly in memory. This optimization is especially important in machines that have only a small number of registers.

2. Stack height reduction can reduce the number of instructions in the target machine needed to evaluate the entire expression by eliminating extra load instructions. This is especially true in arithmetic instructions in which one and only one operand can address memory. In machines that provide memory-to-memory operations (e.g. S-1), no load instruction is needed; in machines in which all operands in expressions need to be loaded (e.g. MIPS), stack height reduction cannot reduce the number of load instructions.

Example. For the Fortran statement

```
I = (I + 5) + (J + K) + ((L + M) + (M + J)).
```

Original DEC 10 code:

```
MOVE   4 ,$MAIN.+33          load I
ADDI   4 ,5                  I + 5
MOVE   1 ,$MAIN.+34          load J
ADD    1 ,$MAIN.+35          J + K
ADD    4 ,1                  (I + 5) + (J + k)
MOVE   2 ,$MAIN.+36          load L
ADD    2 ,$MAIN.+37          L + M
MOVE   3 ,$MAIN.+37          load M
ADD    3 ,$MAIN.+34          M + J
ADD    2 ,3                  (L + M) + (M + J)
ADD    4 ,2                  (I + 5) + (J + K) + ((L + M) + (M + J))
MOVEM  4 ,$MAIN.+33          store to I
```

Stack-height reduced DEC 10 code:

```
MOVE   4 ,$MAIN.+33          load I
ADDI   4 ,5                  I + 5
ADD    4 ,$MAIN.+34          add J
ADD    4 ,$MAIN.+35          add K
ADD    4 ,$MAIN.+36          add L
ADD    4 ,$MAIN.+37          add M
ADD    4 ,$MAIN.+37          add M
ADD    4 ,$MAIN.+34          add J
MOVEM  4 ,$MAIN.+33          store to I
```

Original S-1 code:

```
Add.S     RTA,$MAIN.+116,#5              I + 5
Add.S     RTB,$MAIN.+120,$MAIN.+124      J + K
Add.S     RTA,RTB                        (I + 5) + (J + K)
Add.S     RTB,$MAIN.+128,$MAIN.+132      (L + M)
Mov.S.S   R1 ,$MAIN.+132                 load M
Add.S     R1 ,$MAIN.+120                 M + J
Add.S     RTB,R1                         (L + M) + (M + J)
Add.S     $MAIN.+116,RTA,RTB             I = (I + 5) + (J + K) + ((L + M) + (M + J))
```

Stack-height reduced S-1 code:

```
Add.S     RTA,$MAIN.+116,#5              I + 5
Add.S     RTA,$MAIN.+120                 add J
Add.S     RTA,$MAIN.+124                 add K
Add.S     RTA,$MAIN.+128                 add L
Add.S     RTA,$MAIN.+132                 add M
Add.S     RTA,$MAIN.+132                 add M
Add.S     $MAIN.+116,RTA,$MAIN.+120      add J and store to I
```

□

In the above examples, the DEC 10 instructions allow only one operand to address memory. Thus, the improvement in the optimized code is quite significant. The S-1 instructions allow both operands to address memory, and the effect of stack-height reduction is not as dramatic. In the MIPS, the arithmetic instructions cannot have memory operands, and all memory references require separate load instructions. Thus, the number of instructions in expression evaluation will not be affected by stack-height reduction. However, stack-height reduction still benefits the MIPS by reducing the number of registers required in expression evaluation.

Register allocation

Register allocation on the intermediate code level can affect the underlying machine code in many different ways:

1. By referencing variables in registers, it allows the use of the register direct addressing mode without the need of extra load instructions generated by the code generators. The same is true for stores to variables. The use of the register direct mode saves one memory cycle. The number of instructions may or may not be affected depending on the machine and the type of operation.

Example. In the DEC 10, the number of instructions is not changed in the case of addition because the code generator can use direct memory addressing. Register allocation only changes the addressing mode:

```
ADD    3 ,PP$DAT+86          add I to register 3
```

is changed to

```
ADD    3 ,4                  add I in register 4 to register 3
```

In the MIPS, register allocation is especially effective because the arithmetic instructions cannot reference operands in memory. The expressions A + B is translated to:

```
ld #FPinit-104,r4            load A to r4
ld #FPinit-100,r5            load B to r5
add r4,r5                    A[I]+B[I]
```

If variables A and B have been allocated to registers, the two load instructions can be eliminated. □

2. The positions of the load and store instructions to and from registers are optimized so that they do not occur at frequently executed program points. This optimization is effective regardless of the machine characteristics.

3. By allocating index variables in registers, it facilitates the code generators to use the indexed or base addressing modes instead of performing straight additions in address expressions. Each IXA operation in U-Code can be handled by the use of an indexed operand address.

Example. In accessing an array element A[I], the DEC 10 code before register allocation is:

```
MOVEI  2 ,PP$DAT+86          load adr(A)
ADD    2 ,PP$DAT+287         adr(A)+I
MOVE   4 ,0(2 )              load A[I]
```

After register allocation, the code becomes:

```
MOVE   4 ,PP$DAT+86(1)       load A[I] (I residing in register 1)
```

In the S-1, the code before register allocation for the statement "A[I] := B[J]" is:

```
Shfa.Lf.S    RTA,PP$DAT+300,#2          load I and shift it by 2 bits
Shfa.Lf.S    RTB,PP$DAT+296,#2          load J and shift it by 2 bits
Mov.S.S      PP$DAT+304[RTA],PP$DAT+340[RTB]   A[I] := B[J]
```

After register allocation, the entire statement can be handled by one instruction, with I residing in register R27 and J residing in register R28.

```
Mov.S.S      PP$DAT+304[R27]↑2,PP$DAT+340[R28]↑2
```

□

4. By allocating address variables in registers, it facilitates the use of the register-indirect addressing mode or base addressing, possibly in conjunction with an index register.

Example. In the 68000, the base address in the indirect or indexed addressing mode is always specified using an address register. The instructions to load addresses into the address registers prior to the uses of these addressing modes can be avoided if the address quantities have been allocated in registers by the optimizer. (See the examples on induction expressions below.)

The effects of 3 and 4 depend on the availability of the respective addressing modes in the machine. Since different machines provide different forms of addressing, the effects of register allocation on addressing can vary widely among machines.

Common subexpressions

Common subexpression optimization eliminates duplicate computations occurring in the program. Since redundant code is eliminated, this optimization is beneficial regardless of the machine characteristics. However, the values of the common subexpressions have to be saved at their points of computation and re-loaded at their subsequent occurrences. Since each redundant computation involves at least one memory reference†, the execution time saved is likely to be greater than the cost for the saving and re-loading. The net speed-up depends on how much the saved computation time exceeds the time for the saving and re-loading. If registers cannot be used to save the contents, the saving and re-loading to and from memory may exceed the computation time saved in the case of simple expressions. Thus, if the underlying machine does not provide many registers, common subexpression elimination may not be very effective. This saving and re-loading of the values of expressions also occurs in the case of the code motion of expressions, but in that case, it is the movement of computations out of loops that is mostly responsible for speeding up the execution time.

The optimizer detects redundancy among all expressions. In the case of address expressions, however, the recognition of redundancy may or may not be beneficial, depending on the machines. This is because it is possible to incorporate some address arithmetic into operand addresses using special addressing modes. Examples of address arithmetic that can be handled by special addressing modes are the indirect loads and indexing operations. In some cases, special addressing modes can represent the same computations as entire address expressions. If common subexpressions are recognized in address expressions that can be translated into special operand addresses, the saving into temporaries may be more expensive than the redundant address computations. If a common subexpression is nested inside a larger address expression, the saving operation also prevents the collapse of the larger address expression into a single operand address. Thus, common subexpression optimization in address expressions is not as effective in machines with advanced addressing modes. But since not all address expressions can

† Otherwise, constant arithmetic will be performed by the optimizer.

fit into single operand addresses, common subexpression optimization in address expressions is still beneficial in many situations.

Example. The array reference `A[I]` can be translated into a single operand address in the DEC 10:

```
MOVE  10,PP$DA+86(5)          load A[I], I in register 5
```

Even if the address computation of `A[I]` has been saved in an earlier occurrence, the re-use of the saved value would not result in better code because indexed addressing is just as fast as indirect addressing in the DEC 10:

```
MOVEI  8,PP$DA+86(5)          save adr(A[I])

. . .

ADD  10,0(8)                  load A[I] using address in register 8
```

□

Strength reduction

The optimization of strength reduction, associated with induction expressions in loops, can bring about the following effects on the underlying machines:

1. Expensive multiplication operations are replaced by additions, thus saving computation time.

2. The computation of address expressions is moved out of loops and incremented each time through the loop; this can be looked at as code motion of expressions that contain induction variables.

Example. Suppose the array reference `A[I,J]` occurs in a loop. The DEC 10 code for the address computation is:

```
MOVE    4 ,PP$DAT+20087          load I
IMULI   4 ,100                   I times 100
MOVE    1 ,PP$DAT+20088          load J
SOS     1 ,1                     decrement J
MOVEI   2 ,PP$DAT+-13(4 )        load adr(A)
ADD     2 ,1                     adr(A) + computed offset
MOVE    4 ,0(2 )                 load A[I,J]
```

After optimization, the induction expression that computes the address of `A[I,J]` are moved outside the loop. In the loop, the same array reference is replaced by:

```
MOVE    1 ,0(8 )
```

where register 8 contains the address of `A[I,J]`. Register 8 is incremented in the loop whenever the induction variables `I` and `J` are incremented. □

3. Registers are allocated to contain the address expressions that are moved, thus facilitating the use of special addressing modes, especially the register-indirect addressing mode and base addressing.

4. Because the optimizer introduces the increments of registers containing address expressions, it enables the code generators to make use of the autoincrement and autodecrement addressing modes in machines where these addressing capabilities are available.

Example. Pascal FOR loop:

```
FOR I:=1 TO 100 DO A[I]:=A[I]+B[I];
```

Original 68000 code:

```
        movl    #1,pp$dat+1376          I := 1
$2:
        movl    pp$dat+1376,d0          load I
        asll    #2,d0                   I times 4 to get offset in bytes
        movl    #pp$dat+572,a0          load adr(A)
        movl    pp$dat+1376,d1          load I
        asll    #2,d1                   I times 4 to get offset in bytes
        movl    #pp$dat+572,a1          load adr(A)
        movl    pp$dat+1376,d7          load I
        asll    #2,d7                   I times 4 to get offset in bytes
        movl    #pp$dat+972,a6          load adr(B)
        movl    a1@(0,d1:L),d1          load A[I]
        addl    a6@(0,d7:L),d1          A[I]+B[I]
        movl    d1,a0@(0,d0:L)          assign to A[I]
        addql   #1,pp$dat+1376          increment I
        cmpl    #100,pp$dat+1376        check for loop termination
        jle     $2
```

Optimized 68000 code:

```
        moveq   #1,d7                   I := 1
        movl    #pp$dat+576,a4          load adr(A)
        movl    #pp$dat+976,a5          load adr(B)
$2:
        movl    a5@+,d0                 load B[I] and increment adr(B[I])
        addl    d0,a4@+                 add B[I] to A[I] and increment adr(A[I])
        addql   #1,d7                   increment I
        cmpl    #100,d7                 check for loop termination
        jle     $2
```

□

The detection of the opportunities to use autoincrement or autodecrement addressing, as in the above example, is limited to information that can be gathered within one basic block, since code generators rarely do global analysis of the program. If the reference and increment of an address do not occur in the same basic block, the code generator may not be able to recognize the opportunity.

6.6. Relation to Machine Characteristics

We now summarize how the relevant machine qualifications we mentioned in Section 6.3 influence the ways machines can benefit from the optimizations of UOPT.

Three-address machines can completely specify an arithmetic operation in one instruction. Small common subexpression elimination may not be very useful to such machines, since a one-instruction computation may be less expensive than the saving and re-loading of an identical computation.

In one-address machines, stack height reduction is extremely beneficial, because the number of load instructions is minimized. In a stack height reduced, left-associative expression tree, only a single load instruction is needed; other operands are added to the accumulator directly from memory. In this case, the total number of instructions is equal to one plus the number of operations involved in the expression.

Machines without the immediate addressing mode cannot benefit from constant propagation, since constants have to be stored and referenced from memory. Machines with register indirect addressing benefit from the allocation of address quantities in registers. The use of the autoincrement and autodecrement modes are also made possible by strength reduction. Machines with indexed and base addressing also benefit from register allocation. In machines with multiple offset fields in these addressing modes, however, the optimization of address collapsing may not have direct benefits since the constants to be combined could have originally occupied the multiple offset fields.

Machines with non-orthogonal instruction sets usually exhibit a high degree of irregularity or unpredictability in the ways they can benefit from machine-independent optimizations.

Machines with complex and powerful instruction sets usually do not benefit as much from common subexpressions as reduced instruction-set machines. The primitive operations on the intermediate code level do not map easily into the operations at the machine instruction level.

For machines in which one and only one operand field in arithmetic instructions can access memory, stack height reduction is extremely useful, for the same reason as it is in the case of one-address machines above. Whenever there are some operand fields in instructions that cannot reference memory, register allocation is useful. For machines in which memory reference is limited to only the load and store instructions, register allocation is especially beneficial. These machines also benefit from stack height reduction because all variables that appear in expressions have to occupy registers; the chance of running out of registers is reduced, but the total number of instructions will not be changed.

The characteristics of directly-executable language (DEL) machines differ widely with respect to the nature of the languages that they support. In the case of FOM, the level of the machine code corresponds quite well with the level of U-Code. Since the level of U-Code is not low, we do not anticipate much difficulty for other DEL's to make use of optimizations in U-Code. The instruction sets of DEL's are usually quite orthogonal, and this enhances the usefulness of machine-independent optimizations to them.

6.7. Additional Remarks

From the comparison of optimization results on different machines in Section 6.4, and the discussion of the differing effects of the various optimizations on target machines in Section 6.5 and 6.6, we can reach an overall conclusion: the machine-independent optimizations performed by UOPT are beneficial for most real machines, but are slightly more effective in machines with simple instruction sets and addressing formats, although there are exceptions with respect to individual optimizations. To explain this, we introduce the concept of context-independent optimizations and context-dependent optimizations. Both these qualifications are applied to machine-independent optimizations. The optimizations mentioned in Section 6.5 that have differing effects on different machines are context-dependent optimizations, because their effectiveness depends on the details of the machine code. The rest of the optimizations (e.g. dead code elimination, constant arithmetic, code motion, etc.) are context-independent optimizations, because their effectiveness is independent of the machine characteristics. In machines with powerful and complicated instruction sets and addressing modes, the code generation process is more complex, because the code generator has to look for opportunities of using specific constructs in the instruction sets in order to fully utilize the capability provided by the machine. This peephole optimization is highly machine-dependent, and interferes with the context-dependent optimizations so that the latter's effects are not so directly felt in the final machine code.

To bring the above remarks into better perspectives, we group the set of all possible optimizations for a given machine according to whether they are machine-independent or machine-dependent. As shown in Fig. 6.6.1, the machine-independent optimizations are further divided into two subsets corresponding to the context-independent and context-dependent optimizations. The set of machine-dependent optimizations intersects with the context-dependent subset because the effects of the latter are masked by machine-dependent peephole optimizations. The set of machine-independent optimizations is always the same, but the set of machine-dependent optimizations varies among machines. A machine with a powerful instruction repertoire provides greater opportunities for peephole optimization, and the set of machine-dependent optimizations shown in Fig. 6.6.1 will correspondingly be larger; and when this set is larger, it is likely that

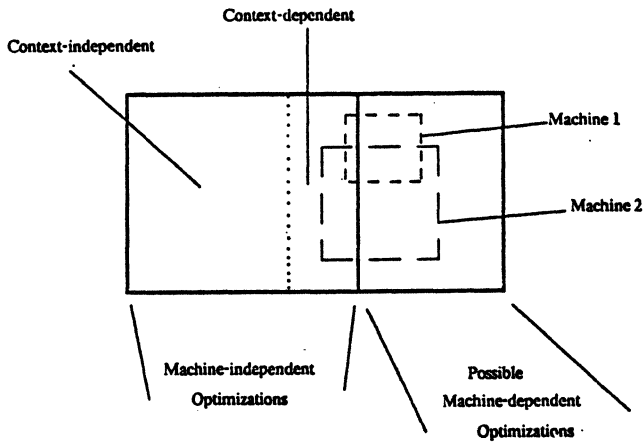


Fig. 6.6.1 Possible optimizations in real-world machines

its intersection with the context-dependent subset of machine-independent optimizations will increase. Because of this larger area of intersection, a larger portion of machine-independent optimizations is always performed in the code generation process, so that the impact of the machine-independent optimizations on the object code is not as strongly felt as in machines with simpler instruction sets.

Although a small part of the machine-independent optimizations can be obscured by the code generation process, the optimizations performed by UOPT can effectively reduce the running times of the object code in all the machines we have encountered. The preceding measurements and evaluations have allowed us to conclude that our approach of portable, machine-independent optimization is highly feasible in implementing production optimizers.

7. Conclusion

In this Chapter, we remark on the significance of this research, and put forth some suggestions for further related work.

7.1. Concluding Overviews

This thesis work has demonstrated that a separate, self-contained optimizer that exists independently of the front-ends and back-ends is both feasible and beneficial. The optimizer UOPT has a simple and clean interface with the front-ends and back-ends, and does not require significant changes to target it to new machines. It has been proven to be highly effective on a wide range of machines.

We believe that the intermediate code we used is a good compromise between completely machine-independent intermediate forms, which often restrict the extent of optimization that can be performed, and low-level pseudo-machine languages, which limit the types of machines that can benefit from the machine-independent optimizations. Although U-Code is slightly machine-dependent, this machine dependence does not limit the portability or the machine-independence of UOPT.

One of the greatest obstacles facing the prospective compiler writer is the need to implement various optimizations in his compiler. As a result of UOPT, an optimizer potentially exists for any machine, on the condition that the compilation process uses U-Code as the intermediate form.

Looking at the implementation aspect of UOPT, the novel global optimization framework introduced in this thesis makes it possible to systematize, simplify and speed up a full range of optimization processes. Some previously separate optimizations can now be performed concurrently. We have addressed the problem of sequencing the various optimization phases for maximal efficiency and best optimization results. All these are accomplished with an accompanying reduction in implementation complexities. The global optimization methodology can be followed by any other general-purpose optimizer.

In the area of register allocation, we have demonstrated that, using a few machine parameters, register allocation can be effectively and efficiently performed in the machine-independent context. Using a priority-based coloring algorithm, the traditional coloring problem can be approached practically and efficiently at the intermediate code level.

7.2. Suggestions for Further Work

One of the main limitations to UOPT's optimizations has been the need to assume the worst case at procedure calls regarding which variables are altered or referenced. Implementing inter-procedural flow analysis will allow UOPT to pin-point the exact variables affected by procedure calls. The analysis will involve an initial pass over the program that gathers and computes the effects of each procedure. The information made available by the inter-procedural flow analysis can then be supplied to UOPT when it performs global optimization.

Extensions and additions to the optimizations performed in UOPT are possible. Among these are the optimization of bounds-checking, optimizations aimed at reducing code size (e.g. code hoisting) and the capability to allow UOPT to change the control flow constructs of the programs. Since these interact with the optimizations already performed in UOPT, the conciseness and ease of maintenance of UOPT may have to be compromised.

Register allocation in UOPT also provides opportunities for further enhancement, perhaps at the expense of more optimization running time in the register allocation phase. Currently, code motion of the register-memory move instructions is performed after all register allocation has taken place. In the global coloring phase, register allocation priorities are computed by assuming that the register-memory move instructions are at their fixed positions, and no account is taken of the possibility that these move instructions can be transferred later to better positions to minimize the execution time cost. The algorithm could be made more exact if the possibility of code motion to reduce cost is factored into the priority ordering.

Procedure parameters are commonly passed in registers. Optimizing the use of registers in parameter passing is another possible extension to register allocation in UOPT. The primary purpose is to minimize the cost for the loading of parameters into registers before they are passed and the saving of parameters into home locations at the entries to callees. Register assignments to passed parameters should take into account the appearances of the parameters in the callees and before the points of call in the callers.

The possibility of overlapping registers of different sizes has not been treated in UOPT. Although such situations have not appeared in the machines to which UOPT has been applied, they do occur in other machines. It would be interesting to see how well the coloring algorithm in UOPT can be adapted to such situations.

In the systems aspect, there are many other opportunities related to UOPT that can be attempted. UOPT currently supports only Pascal and Fortran. It is possible to introduce additional programming languages that are compiled via U-Code. Extensions to U-Code should be minimized and reserved only for extreme cases. Specialized languages may display their

own commonly-occurring optimization opportunities, and these languages can have their own front-end optimizers that perform their own specialized optimization transformations and output U-Code; UOPT can still be used to advantage as the general-purpose global optimizer in the subsequent common optimization phase. Any extension to U-Code, or modification to its semantics, could entail changes in the optimizer itself. The extensions introduced should be such that they do not affect the optimizations already existing in UOPT.

To recognize the existence of other intermediate code for other programming languages and code generators, UOPT can be re-implemented on other intermediate code. Although the intermediate code may affect the optimizations that can be performed, the optimization methodogy in UOPT is somewhat independent of the intermediate code. Another possibility is to build translators between U-Code and other intermediate forms supported by other programming languages and code generators. This approach requires much less programming effort, although there is more overhead in the compilation and optimization processes due to the existence of multiple intermediate forms and the larger number of phases in translation.

Lastly, it is also possible for specific installations to incorporate UOPT as a built-in component in code generation. UOPT can be made the front part of a code generator. The code generator uses UOPT as the module that inputs the intermediate code. After global optimization, the code generator emits object code directly from the internal representations of UOPT. Such an arrangement serves to eliminate the input/output overhead inherent in multi-pass compiling systems and can render greater code generation efficiency without sacrificing modularity.

References

Aho72 A. V. Aho and J. D. Ullman, "Optimization of Straight Line Code," *SIAM J. Computing 1,* 1, pp. 1–19.

Aho77 A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, 1977.

Alle71 F. Allen and J. Cocke, "A Catalogue of Optimizing Transformations," pp. 1–30 of [Rust72].

Alle75 F. Allen, "Bibliography on Program Optimization," IBM Research Report No. RC 5767, Dec. 1975.

Alle76a F. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Comm. ACM 19,* 3, pp. 137–147.

Alle76b F. Allen, "An Annotated Bibliography of Selected Papers on Program Optimization," IBM Research Report No. RC 5889, March 1976.

Alle80 F. Allen et al., "The Experimental Compiling System," *IBM J. Res. Develop. 4,* 6 (Nov. 1980).

Alle81 F. Allen, J. Cocke and K. Kennedy, "Reduction of Operator Strength," pp. 79–101 of [Much81].

Amma75 U. Amman, K. Jensen, K. Nori, et al., "Pascal P Compiler Implementation Notes," ETH Zurich, 1975.

Ankl82 P. Anklam, D. Cutler, R. Heinen, Jr. and M. D. MacLaren, *Engineering a Compiler — VAX-11 Code Generation and Optimization,* Digital Press, 1982.

Arsa79 J. J. Arsac, "Syntactic Source to Source Transforms and Program Manipulation," *Comm. ACM 22,* 1 (January 1979).

Ausl82 M. Auslander and M. Hopkins, "An Overview of the PL.8 Compiler," *ACM SIGPLAN Notices, 17,* 6 (June 1982), *(Proceedings of the SIGPLAN'82 Symposium on Compiler Construction),* pp. 201–207.

Bagw70 J. T. Bagwell, "Local Optimizations," *SIGPLAN Notices 5,* 7 (July 1970).

Beat74 J. C. Beatty, "Register Assignment Algorithm for Generation of Highly Optimized Object Code," *IBM J. Res. Develop.,* Jan. 74.

Bran82 W. C. Brantley and J. Weiss, "FOM: Principles of Operations," IBM Research Report, August 1982.

Bush79 R. Bush, "UASMINT: A U-Code Assembler and Interpreter," S-1 Project Document, Computer System Lab, Stanford University, June 1979.

Cast79 F. Castaneda, F. Chow, P. Nye, D. Sleator and G. Wiederhold, "PCFORT: A Fortran to P-Code Translator," Computer System Lab Technical Report 160, Stanford University, January 1979.

Chai82 G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring," *ACM SIGPLAN Notices, 17,* 6 (June 1982), *(Proceedings of the SIGPLAN'82 Symposium on Compiler Construction),* pp. 201–207.

Chow80 F. Chow, P. Nye and G. Wiederhold, "UFORT: A Fortran to U-Code Translator," Computer System Lab Technical Report 168, Stanford University, January 1980.

Chow83a F. Chow and M. Ganapathi, "Intermediate Languages in Compiler Construction — A Bibliography," *ACM SIGPLAN Notices, 18,* 11 (Nov. 83), pp. 21–23.

Chow83b F. Chow, "Implemenation Manual for the U-Code Optimizer UOPT," Computer System Lab Technical Note, Stanford University, December 1983.

Cock70 J. Cocke and J. T. Schwartz, *Programming Languages and Their Compilers,* Courant Institute of Mathematical Sciences, New York University, April 1970.

Cock77 J. Cocke and K. Kennedy, "An Algorithm for the Reduction of Operator Strength," *Comm. ACM 20,* 11, Nov. 77.

Cock80 J. Cocke and P. Markstein, "Measurement of Program Improvement Algorithms," *Proc. IFIP Cong. '80,* (Tokyo, Japan, Oct. 6–9, Melbourne, Australia, Oct. 14–17, 1980).

Davi80 J. W. Davidson and C. W. Fraser, "The Design and Application of a Retargetable Peephole Optimizer," *ACM Tran. Prog. Lang. Syst.,* April 1980.

Digi81 *VAX Architecture Handbook,* Digital Equipment Corporation, 1981.

Frai79 D. J. Frailey, "An Intermediate Language for Source and Target Independent Code Optimization," *ACM SIGPLAN Notices 14,* 8 (August 1979), *(Proceedings of the SIGPLAN'79 Symposium on Compiler Construction),* pp. 188–200.

Frei74 R. A. Freiburghouse, "Register Allocation Via Usage Counts," *Comm. ACM 17,* 11, Nov. 74.

Gana80 M. Ganapathi, "Retargetable Code Generation and Optimization using Attribute Grammars," Ph.D. Thesis and Tech. Report 406, Computer Sciences Department, University of Wisconsin – Madison, 1980.

Gana82 M. Ganapathi, C. N. Fischer and J. L. Hennessy, "Retargetable Compiler code Generation," *ACM Computing Surveys, 14,* 4 (Dec. 1982).

Gana84 M. Ganapathi and C. N. Fischer, "Attributed Linear Intermediate Representations for Retargetable Code Generators," *Software – Practice and Experience 14,* 1, January 1984.

Gesc72 C. M. Geschke, "Global Program Optimizations," Ph.D. Thesis, Carnegie-Mellon University, October 1972.

Grah80 S. L. Graham, "Table-driven Code Generation," *IEEE Computer, 13,* 8 (August 80), pp. 25–34.

Gyll79 H. C. Gyllstrom, R. C. Knippel, L. C. Ragland, K. E. Spackman, "The Universal Compiling System," *ACM SIGPLAN Notices, 14,* 12 (Dec. 1979), pp. 64–70.

Hail79 B. Hailpern and B. Hitson, "S-1 Architecture Manual," Computer System Lab Technical Report 161, Stanford University, January 1979.

Harr75 W. Harrison, "A Class of Register Allocation Algorithms," IBM Research Report No. RC 5342, March 27, 1975.

Harr76 W. Harrison, "Formal Semantics of a Schematic Intermediate Language," IBM Research Report No. RC 6271, November 1976.

Hech73 M. S. Hecht and J. D. Ullman, "Analysis of a simple algorithm for global flow problems," *Conf. Record, ACM Symposium on Principles of Programming Languages,* Boston, Mass., Oct. 1973, pp. 207-217.

Hech77 M. S. Hecht, *Data Flow Analysis of Computer Programs,* American Elsevier, New York, New York.

Henn82a J. L. Hennessy, "Symbolic Debugging of Optimized Code," *ACM Tran. Prog. Lang. Syst.,* 1982.

Henn82b J. L. Hennessy, "Pascal*: A Pascal Based Systems Programming Language," Computer System Lab Technical Note 174, Stanford University, September 1982.

Henn82c J. L. Hennessy, et al., "The MIPS Machine," *Proc. Compcon,* IEEE, San Francisco, Feb. 1982, pp. 2–7.

Henn83 J. L. Hennessy, et al., "Design of a High Performance VLSI Processor," Technical Report 236, Computer System Lab, Stanford University, 1983.

Jens75 K. Jensen and N. Wirth, *Pascal User Manual and Report,* Springer Verlag, New York, 1975.

John75 R. K. Johnsson, "An Approach to Global Register Allocation," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, December 1975.

John77 S. C. Johnson, "A Tour through the Portable C Compiler," UNIX documentation, Bell Telephone Laboratories, Murray Hill, N. J., 1977.

Kenn76 K. Kennedy, "A Comparison of Two Algorithms for Global Data Flow Analysis," *SIAM J. Computing, 5,* 1 (March 76), pp. 158–180.

Kild73 G. A. Kildall, "A Unified Approach to Global Program Optimization," *Proc. ACM Symposium on Principles of Programming Languages,* 1973, pp. 194–206.

Kim78 J. Kim, "Spill Placement Optimization in Register Allocation for Compilers," IBM Research Report No. RC 7251, August 8, 1978.

Knut71 D. E. Knuth, "An Empirical Study of Fortran Programs," *Software - Practice and Experience 1,* 2, pp. 105–133.

Korn78 P. Kornerup, B. B. Kristensen and O. L. Madsen, "Interpretation and Code Generation based on Intermediate Languages," Report DAIMI PB-88, Matematisk Institut, Aarhus Universitet, Danmark, May 1978.

Leve79 B. W. Leverett, "An Overview of the Production-Quality Compiler-Compiler Project," Tech. Report CMU-CS-79-105, Carnegie-Mellon University, February 1979.

Leve81 B. W. Leverett, "Register Allocation in Optimizing Compilers," Ph.D. Thesis and Technical Report CMU CS-81-103, Carnegie-Mellon University, February 1981.

Livi83 *S-1 Uniprocessor Architecture,* Lawrence Livermore Laboratory, University of California, April 1983.

Love76 D. B. Loveman, "Program Improvement by Source to Source Transformation," *Conf. Record of the Third ACM Symposium on Principles of Programming Languages,* 1976.

Mads76 O. L. Madsen, B. B. Kristensen and J. Staunstrup, "Use of Design Criteria for Intermediate Languages," Report DAIMI PB-59, Matematisk Institut, Aarhus Universitet, Danmark, August 1976.

Mint79 R. J. Mintz, G. A. Fisher and M. Sharir, "The Design of a Global Optimizer," *ACM SIGPLAN Notices, 14,* 8 (August 1979), *(Proceedings of the SIGPLAN'79 Symposium on Compiler Construction),* pp. 226–234.

More79 E. Morel and C. Renvoise, "Global Optimization by Suppression of Partial Redundancies," *Comm. ACM 22,* 2 (February 1979).

More81 E. Morel and C. Renvoise, "Interprocedural Elimination of Partial Redundancies," pp. 160–188 of [Much81].

Moto80 *MC68000 16-bit Microprocessor User's Manual,* Motorola Inc., 1980.

Much81 S. S. Muchnick and N. D. Jones, *Program Flow Analysis,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

Nels79 P. A. Nelson, "A Comparison of Pascal Intermediate Languages," *ACM SIGPLAN Notices, 14,* 8 (August 1979), *(Proceedings of the SIGPLAN'79 Symposium on Compiler Construction),* pp. 208–213.

Nye81 P. Nye, "S-1 U-Code: An Intermediate Language for Pascal* and Fortran," S-1 Project Document PAIL-8, Computer System Lab, Stanford University, October 1981.

Nye83 P. Nye and F. Chow "A Transporter's Guide to the Stanford U-Code Compiler System," Technical Report, Computer System Lab, Stanford University, June 1983.

Palm75 R. C. Palm Jr., "A Portable Optimizer for the Language C," Master's Thesis, July 1975, Masachusetts Institute of Technology.

Perk79 D. Perkins and R. Sites, "Machine-independent Pascal Code Optimization," *ACM SIGPLAN Notices, 14,* 8 (August 1979), *(Proceedings of the SIGPLAN'79 Symposium on Compiler Construction),* pp. 201-207.

Rust72 R. Rustin (Editor), *Design and Optimization of Compilers,* Prentice-Hall, Englewood Cliffs, N. J., 1972.

Scha73 M. Schaefer, *A Mathematical Theory of Global Program Optimization,* Prentice-Hall, Englewood Cliffs, N. J.

Schn73 P. B. Schneck and E. Angel, "A Fortran to Fortran Optimizing Compiler," *Computer Journal, 16,* 4, pp. 353-354.

Schw73 J. T. Schwartz, "On Programming: An Interim Report on the SETL Project," Courant Institute of Math. Sciences, New York University, 1973.

Site79a R. L. Sites and D. R. Perkins, "Machine-independent Register Allocation," *ACM SIGPLAN Notices, Vol. 14, Number 8 (August 1979), (Proceedings of the SIGPLAN'79 Symposium on Compiler Construction),* pp. 221-225.

Site79b R. Sites et al., "Machine-independent Pascal Optimizer Project: Final Report," Technical Report UCSD/CS-79/038, University of California at San Diego, November 1979.

Stah76 T. A. Standish, D. C. Harriman, D. F. Kibler and J. M. Neighbors, *The Irvine Program Transformation Catalogue,* Dept. of Information and Computer Science, University of California, Irvine, 1976.

Stan76 "DEC System 10/20 Hardware Manual and FAIL," Stanford Artificial Intelligence Lab Operating Note 75, November 1976.

Stee61 T. B. Steel, Jr., "A First Version of UNCOL," *Proc. Western Joint Computer Conference, AFIPS,* 1961, pp. 371-378.

Tane82 A. S. Tanenbaum, H. V. Staveren and J. W. Stevenson, "Using Peephole Optimization on Intermediate Code," *ACM Tran. Prog. Lang. Syst. 4,* 1 (January 1982).

References

Tane83 A. S. Tanenbaum, H. V. Staveren, E. G. Keizer and J. W. Stevenson, "A Practical Tool Kit for Making Portable Compilers" *Comm. ACM 26,* 9 (September 1983), pp. 654–660.

Wilh81 R. Wilhelm, "Global Flow Analysis and Optimization in the MUG2 Compiler Generating System," pp. 132–159 of [Much81].

Wilk83 A. Wilk and W. Silverman, "OPTIMA – A Portable PCODE Optimizer," *Software – Practice and Experience, 13* (April 1983), pp. 323–354.

Wulf75 W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke, *The Design of an Optimizing Compiler,* American Elsevier, New York, N. Y.

Appendix A. Short Guide to U-Code

U-Code, a descendent of the P-Code intermediate language emitted by many Pascal compilers, exists in two different formats: a text format and a binary format. A U-Code instruction is represented in compiler programs as a record. In the binary format, these records are written directly into files. As a result, the read-write process is faster, and the binary format files occupy much less disk space.

U-Code can be thought of as the assembler language for a hypothetical U-machine. The U-machine has the following components:

1. A stack for use in all expression evaluation.
2. A read-only storage area where instructions and string constants are kept.
3. A static storage area (memory type S) where global variables and Fortran *own* variables are kept.
4. A set of registers (memory type R) where data items can be kept for fast accesses.
5. A memory stack divided into stack frames for the processing of procedure invocation. A stack frame (or activation record) is pushed on top of the memory stack whenever a procedure is invoked. The stack frame contains parameters, local variables and compiler-generated temporaries. Stack frame storage areas are either designated as memory type M for local storage or memory type P for parameters.
6. A heap for dynamic allocation of data objects at program execution time.

The Pascal and Fortran front-ends that output U-Code are both one-pass compilers. U-Code programs are organized into modules and procedures in the same order as they occur in the source programs. In the following, we group the complete U-Code instruction set into classes and give the syntax and a short description of each op-code. Information of particular use to the optimizer is noted. The readers are referred to [Nye81] for a more complete definition of the U-Code language.

1. Direct memory operations

⟨op⟩ ⟨data type⟩ ⟨memory type⟩ ⟨block number⟩ ⟨address⟩ ⟨length⟩

where ⟨op⟩ is:

LOD — load onto stack.

STR — store from stack into memory.

NSTR — same as STR but not popping item.

The length information is important in checking for storage interference in instructions that access memory. When the memory type is M or P, the variable is local if the block number is the same as the block number of the current procedure.

2. Constants

LDC ⟨data type⟩ ⟨length⟩ ⟨constant value⟩

LCA ⟨memory type⟩ ⟨length⟩ ⟨block number⟩ ⟨constant value⟩

LDA ⟨memory type⟩ ⟨block number⟩ ⟨address⟩ ⟨length⟩ ⟨base address⟩

LDP ⟨static level⟩ ⟨block number⟩ ⟨procedure name⟩

LDC pushes a constant value onto the stack. LCA pushes the address of a string constant onto the stack. LDA pushes a constant address onto the stack. LDP pushes a procedure descriptor onto the stack. In the LDA instruction, the base address together with the length field gives the range within which the result of the subsequent address computation will lie.

3. Unary operators

⟨op⟩

where ⟨op⟩ is:

CHKF — check if false.
CHKT — check if true.
CHKN — check if nil pointer.

⟨op⟩ ⟨data type⟩

where ⟨op⟩ is:

ABS — get absolute value.
CHKH — check upper bound.
CHKL — check lower bound.
NEG — negate.
NOT — boolean not.
ODD — check if odd number.
SQR — square root.

⟨op⟩ ⟨data type⟩ ⟨integer value⟩

where ⟨op⟩ is:

DEC — decrement.

INC — increment.

SGS — form singleton set.

⟨op⟩ ⟨resultant data type⟩ ⟨original data type⟩

where ⟨op⟩ is:

CVT — convert type of top of stack item.

CVT2 — convert type of second item on stack.

RND — round value of top of stack item.

ADJ ⟨data type⟩ ⟨offset⟩ ⟨length⟩

ADJ adjusts the size of a set.

4. Binary operators

⟨op⟩ ⟨data type⟩

where ⟨op⟩ is:

ADD — addition.

AND — boolean and.

DIF — set difference.

DIV — division.

EQU — equal.

GEQ — greater than or equal to.

GRT — greater than.

IOR — inclusive or.

LEQ — less than or equal to.

LES — less than.

MAX — maximum of two numbers.

MIN — minimum of two numbers.

MOD — remainder.

MPY — multiplication.

MUS — form a set of the elements in the given range.

NEQ — not equal.

SUB — subtraction.

XOR — exclusive or.

IXA ⟨data type⟩ ⟨unit size⟩

IXA computes the offset within an array by multiplying the subscript by the unit size of the array and adding to the base address.

⟨op⟩ ⟨data type⟩ ⟨length⟩

where ⟨op⟩ is:

INT — set intersection.

UNI — set union.

INN ⟨data type⟩ ⟨check flag⟩

INN checks set membership of an element.

5. Indirect memory operations

⟨op⟩ ⟨data type⟩ ⟨offset⟩ ⟨length⟩

where ⟨op⟩ is:

ILOD — load indirect.

ISTR — store indirect.

INST — non-destructive store indirect.

⟨op⟩ ⟨memory type⟩ ⟨length⟩

where ⟨op⟩ is:

MOV — block move.

IEQU — indirect equal.

IGRT — indirect greater than.

IGEQ — indirect greater than or equal to.

ILEQ — indirect less than or equal to.

ILES — indirect less than.

INEQ — indirect not equal.

In each of these instructions, the range of storage locations affected by the operation is given by the LDA instruction that loads the address argument.

6. Labels

⟨label⟩ LAB ⟨flag⟩

The flag indicates whether there is jump to the label from outside the current procedure. If there is such a jump, the block marked by the label is included as an entry point.

7. Jumps

⟨op⟩ ⟨label⟩

where ⟨op⟩ is:

FJP — jump if false.

TJP — jump if true.

UJP — unconditional jump.

GOOB ⟨static level⟩ ⟨label⟩

GOOB specifies a jump out of the current procedure to a nesting procedure.

RET

RET causes control to return to the calling procedure.

XJP ⟨data type⟩ ⟨case label⟩ ⟨default label⟩ ⟨lower bound⟩ ⟨upper bound⟩

⟨label⟩ CLAB ⟨length⟩

XJP and CLAB together implement the case statement.

8. Procedure calls

CUP ⟨data type⟩ ⟨block number⟩ ⟨name⟩ ⟨pop⟩ ⟨push⟩

ICUP ⟨data type⟩ ⟨pop⟩ ⟨push⟩

MST ⟨level⟩

PAR ⟨data type⟩ ⟨memory type⟩ ⟨block number⟩ ⟨address⟩ ⟨length⟩

CUP calls the procedure specified. ICUP calls the procedure whose descriptor is on top of the stack. MST marks the stack prior to parameter passing in procedure calls. PAR specifies the current item on the stack as a parameter to be passed in the upcoming call. In the CUP instruction, UOPT can determine the level of the called procedure by table look-up using the block number given in the instruction.

9. Special operators

DUP ⟨data type⟩

POP ⟨data type⟩

SWP ⟨top data type⟩ ⟨second data type⟩

DUP pushes an extra copy of the top item on the stack. POP pops the stack top item. SWP interchanges the top and second items on the stack.

10. Register operations

REGS ⟨action⟩ ⟨register class⟩ ⟨offset⟩ ⟨length⟩

REGS appears at the beginning of each procedure to reserve the registers used by UOPT in that procedure.

⟨op⟩ ⟨data type⟩ ⟨memory type⟩ ⟨block number⟩ ⟨register offset⟩ ⟨length⟩

where ⟨op⟩ is:

RLOD — load register item on the stack.
RSTR — store item from top of stack to register.

11. Non-executable instructions

BGN ⟨module name⟩ ⟨integer flag⟩

STP ⟨module name⟩

BGN and STP mark the beginning and end of a U-Code module. One module usually corresponds to a source program file.

⟨name⟩ ENT ⟨data type⟩ ⟨static level⟩ ⟨block number⟩ ⟨pop⟩ ⟨push⟩ ⟨external flag⟩

END ⟨name⟩

ENT and END mark the beginning and end of a procedure.

BGNB

ENDB

BGNB and ENDB together mark a range in the program code where the stack does not fall below its height at the positions of the BGNB and ENDB.

```
LEX ⟨level⟩ ⟨block number⟩
```

LEX specifies the static levels and block numbers of the procedures that enclose the current procedure. The nesting relationships among the procedures are determined according to the LEX instructions.

```
LOC ⟨page number⟩ ⟨line number⟩ ⟨character count⟩
```

LOC is used for reporting source program line numbers for debugging purposes.

```
COMM ⟨comment⟩
```

COMM is for putting in comments in U-Code files.

```
OPTN ⟨option name⟩ ⟨integer⟩
```

OPTN is for specifying a variety of compilation and optimization options.

```
⟨name⟩ EXPV ⟨data type⟩ ⟨memory type⟩ ⟨block number⟩ ⟨address⟩ ⟨length⟩
```

```
⟨name⟩ IMPV ⟨data type⟩ ⟨memory type⟩ ⟨block number⟩ ⟨address⟩ ⟨length⟩
```

EXPV and IMPV specify the export and import of variables.

```
⟨name⟩ DATA ⟨number⟩
```

```
DEF ⟨memory type⟩ ⟨length⟩
```

```
SDEF ⟨block number⟩ ⟨length⟩
```

DATA is for associating a name and a block number to a static data area. DEF defines the size of the M or P memory area of the current procedure. SDEF defines the size of a static memory block.

```
INIT ⟨data type⟩ ⟨memory type⟩ ⟨block number⟩ ⟨first offset⟩ ⟨last offset⟩ ⟨length⟩ ⟨value⟩
```

```
ZERO ⟨data type⟩ ⟨memory type⟩ ⟨block number⟩ ⟨address⟩ ⟨length⟩
```

INIT initializes the given storage area to the specified value. ZERO is for zeroing out the area indicated.

```
PLOD ⟨data type⟩ ⟨memory type⟩ ⟨block number⟩ ⟨address⟩ ⟨length⟩
```

```
PSTR ⟨data type⟩ ⟨memory type⟩ ⟨block number⟩ ⟨address⟩ ⟨length⟩
```

PLOD indicates the loading on the stack of a function result. PSTR indicates the storing of parameters from the stack to their assigned locations at the entry point of a procedure.

Appendix B. Notes on programming Data Flow Analysis

Data flow analysis plays a major role in the various global optimizations of UOPT. Since data flow analysis constitutes a non-trivial part of the processing in global optimization, it is necessary to do it as efficiently as possible.

The iterative algorithm is the simplest and most popular method to perform data flow analysis. The algorithm involves iterating through the nodes of the control flow graph applying the appropriate data flow equation until no more changes take place. When properly implemented, the average number of iterations in the outermost loop of the algorithm required to reach the final solution is around 3, and is seldom above 4 for well-structured programs. However, there are details of implementation not directly evident in the algorithm itself which, if not handled properly, can result in a substantial increase in the number of iterations required. These implementation details are dependent on the nature of the data flow analysis performed.

Most of the data flow analyses in UOPT involve the simultaneous solution of an IN attribute and an OUT attribute at the entries and exits respectively of the basic blocks. As an illustration, we take Eq. (3.3.1) from Chapter 3:

Availability System:

$$\mathtt{AVIN}_i = \begin{cases} \mathtt{FALSE} & \text{if } i \text{ is the entry block;} \\ \prod_{j \in \mathtt{Pred}(i)} \mathtt{AVOUT}_j & \text{otherwise.} \end{cases} \qquad (3.3.1)$$

$$\mathtt{AVOUT}_i = \mathtt{AVLOC}_i + \neg\mathtt{ALTERED}_i \cdot \mathtt{AVIN}_i.$$

The algorithm to compute AVIN and AVOUT is:

Algorithm *Global Availability.*

1. *changed* ← true;
2. WHILE *changed* DO
   a. *changed* ← false;
   b. i ← graph head;
   c. Repeat (i) - (vii) until i = last node;
      (i) *old* ← $\mathtt{AVIN}_i$;
      (ii) For each predecessor j of i do
      $$\mathtt{AVIN}_i \leftarrow \mathtt{AVIN}_i \cdot \mathtt{AVOUT}_j;$$
      (iii) IF *old* $\neq \mathtt{AVIN}_i$ THEN *changed* ← true;
      (iv) *old* ← $\mathtt{AVOUT}_i$;

(v) $\mathtt{AVOUT}_i \leftarrow \mathtt{AVLOC}_i + \neg\mathtt{ALTERED}_i \cdot \mathtt{AVIN}_i$;

(vi) IF *old* $\neq \mathtt{AVOUT}_i$ THEN *changed* $\leftarrow$ true;

(vii) $i \leftarrow$ next node. □

There are a number of ways to minimize the number of iterations in the above algorithm:

1. The graph nodes should be put in depth-first ordering prior to executing the above algorithm. This enables any change in the attribute of the current node to be immediately propagated to its adjacent nodes.

2. If not all the bits of the bit vector are used, masking the unused bits may also eliminate any extra iterations required to propagate information in the unused bits. When the conjunction operator is used, the unused bits should be initially set to 0. When the disjunction operator is used, they should be initially set to 1. Using such masking, the values of the unused bits will not change during the iterations.

3. If the propagation of information is in the forward direction, the loop of step 2c should start from the head of the graph. If the propagation of information is in the backward direction, this loop should start from the tail of the depth-first ordering. In the latter case, step 2c(vii) becomes

   (vii) $i \leftarrow$ previous node.

4. The relative positions of steps 2c(i)-(iii) and 2c(iv)-(vi) also depend on the direction of information propagation. When propagation is in the forward direction, the positions are as they appear above. When propagation is in the backward direction, steps 2c(iv)-(vi) should precede 2c(i)-(iii).

The arrangements of 1, 3 and 4 above speed up the algorithm by following the actual paths of information propagation as close as possible in performing the data flow operations. By propagating information downstream immediately, it is unnecessary to wait for the next iteration in the loop of step 2 whenever any change in the attribute of a node occurs.

Appendix C. Hints on Writing Programs that Cater to Optimization

Different programs exhibit different amount of optimization opportunities. While optimization opportunities are highly dependent on the nature of the programs, the ordinary programmer can enhance the optimizability of his programs by adhering to some guidelines. Here, we give a set of guidelines in writing Pascal and Fortran programs that can specifically enhance the optimizations performed by UOPT. Most of the following points are also applicable to other general-purpose optimizers. Some of these are due to the abscence of inter-procedural data flow analysis in UOPT.

1. **Variable declarations:** Variables should be declared locally and used locally as much as possible. This is because a pointer cannot point to a local variable. Also, local variables cannot be altered or accessed in calls to procedures not nested within the current one. In Fortran, only the common blocks are regarded as non-local storage.

2. **Storage relationships:** Storage overlaps caused by the use of equivalences (or variant records in Pascal) or commons should be minimized. Storage overlaps may cause unnecessary storage interferences that obstruct code movement and the recognition of redundancies. Equivalences also inhibit the allocation of variables in registers by UOPT.

3. **Memory accesses:** Up-level references and side effects (assignments to non-local variables) should be minimized. A pointer or a procedure call can interfere with an up-level memory access.

4. **Parameters:** Parameters should be passed by value whenever possible. This serves to suppress aliasing and side effects. An assignment to a reference parameter potentially kills many non-local variables. Values should be returned via the return values of functions. (This rule does not apply to Fortran programs, which only allow passing by reference.)

5. **Procedure declaration:** Procedures should be declared at the same level as much as possible. In Fortran, this means not using statement functions. When there are nestings among the procedures, procedures in down-level calls can access the local variables of the callers via up-level references and side effects.

6. **Pointers:** The use of pointers should be minimized. Pointer accesses kill all non-local variables, since the pointer can potentially point to any of them.

7. **Loops:** The programmers should stick to the use of standard loops provided in the programming languages. The compiler front-ends specifically compile the loops so that the

resultant control flow structures allow for code motion out of loops. Jumps into or out of loops should be avoided. The programmer should not construct his own loops using goto's.

Appendix D. What the Compiler Front-ends Should Do

UOPT assumes certain configurations in the input code that, if adhered to by the front-ends, can greatly enhance the optimization tasks.

D1. Pascal Front-end

1. **Order of procedures:** The order of the procedures in the U-Code file must correspond to the order in which they are declared in the source program. UOPT needs to know the level of the called procedures at the points of calls and, due to its one-pass nature, the level of a callee is recorded only if its body has been processed earlier.

2. **Identification of the main block:** The main program should be appropriately identified to the optimizer so that, when it is processing the body of the main block, it can treat global static variables as local variables. Currently, the main block is always assigned block number 1 by the Pascal front-end.

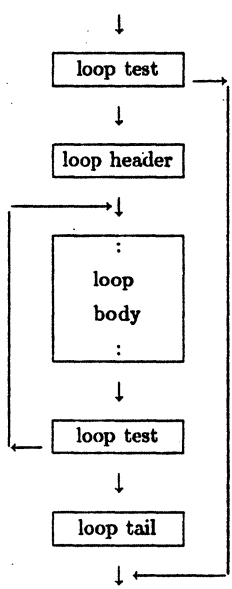


Fig. D.1 Recommended loop structure for `WHILE` and `FOR` loops

3. **WHILE and FOR loops:** Because UOPT does not alter the control flow structure of the program, the front-ends must compile loop statements in the source programs into forms that allow for code insertions in code motion. There have to be header nodes for the placement of loop-invariant expressions in backward code motion, and tail nodes for the placement of assignments moved forward and out of the loops. The more usual WHILE loop form of Fig. 3.6.8 does not accommodate code motion. For both WHILE loops and FOR loops, the compiled control flow structure should be as shown in Fig. D.1. In this configuration, the loop header and loop tail are formed by generating extra labels. In the unoptimized program, these two nodes do not contain any code. During optimization, backward code motion causes insertion of loop invariant expressions at the loop header, and forward code motion moves redundant stores in the loop forward and inserts them at the loop tail. Note that there is an increase in code space, since the loop termination condition appears twice. But constant propagation followed by constant arithmetic can often eliminate the loop entry test. (The REPEAT loop does not require any special treatment for the purpose of optimization.)

D2. Fortran Front-end

1. **Static memory:** All Fortran variables are *own* variables, and they must be allocated static storage. Also, variables within a program unit are not accessible from within other program units. Variables in the common areas are to be treated as global variables instead, since they are static and accessible from more than one program unit. Thus, the block in which non-common variables are allocated has to be identified to the optimizer so that the optimizer can treat the variables there as local variables. Currently, this static block is always assigned the block number 1 by the Fortran front-end.

2. **The levels of procedures:** There is no nesting of procedures in Fortran. However, statement functions can access variables within the program units in which the statement functions are declared. The optimizer has to be able to distinguish statement functions from other Fortran subroutines and functions because the static variables referenced within statement functions are non-local variables. The Fortran front-end UFORT declares all statement functions at static level 3. Except the main program unit, which is at level 1, all other subroutines and functions are declared at level 2.

Appendix E. Examples of Optimized Code

In this appendix, we use a piece of Pascal source code as example and compile this code for the 6 target machines. The U-Code both before and after optimization by UOPT are displayed. For each of the 6 target machines, we list the object code generated from the unoptimized U-Code followed by those generated from the optimized U-Code. This will serve to give a more complete view of the effects that the same optimizations on the intermediate code have on different target machine code.

The example we use is the full extent of the loop that does bubble sort, taken from the benchmark program Bubble used in Chapter 6. The Pascal source code is:

```
top := 70;
WHILE top > 1 DO
   BEGIN
   i := 1;
   WHILE i < top DO
      BEGIN
      IF list[i] > list[i+1]
         THEN Swap(list[i], list[i+1]);
      i := i+1
      END;
   top := top - 1
   END;
```

E1. U-Code

The DEC 10 versions of U-Code are given here. The procedure Swap has been copied in-line by the procedure integrator PMERGE earlier. Note the allocation of various quantities in registers in the optimized version.

Unoptimized

```
COMM  top := 70;

LOC        1       40 0
LDC  L 36 70
CVT  J L
STR  J S 1 8280 36
COMM  while top > 1 do
LOC        1       42 0
LOD  J S 1 8280 36
LDC  L 36 1
CVT  J L
GRT  J
FJP  L4
L5 LAB  0
```

Optimized

```
/ COMM ----BB 05
/ COMM  top := 70;
/ LOC        1       40 0
/ COMM  while top > 1 do
/
/
/
/ LOC        1       42 0
/ COMM ----BB 06
/L5 LAB  0
/ LDC  J 36 70
/ STR  J R 0 72 36
/ COMM ----BB 07
/L6 LAB  0
```

```
L6 LAB  0                                / COMM     begin i := 1;
 COMM     begin i := 1;                  /
 LOC        1       43 0                 / LOC        1       43 0
 LDC  L 36 1                             / COMM    while i < top do
 STR  L S 1 8244 36                      /
 COMM    while i < top do                /
 LOC        1       44 0                 / LOC        1       44 0
 LOD  L S 1 8244 36                      / LDC  J 36 1
 LOD  J S 1 8280 36                      / LOD  J R 0 72 36
 CVT2 J L                                / LES  J
 LES  J                                  / FJP  L7
 FJP  L7                                 / COMM ----BB 08
L8 LAB  0                                /L8 LAB  0
L9 LAB  0                                / LDC  L 36 1
 COMM       begin if list[i] > list[i+1] / STR  L R 0 0 36
                                         / LDA  S 1 3096 2520 3132
                                         / LOD  L R 0 0 36
                                         / IXA  L 36
                                         / STR  A R 0 36 36
                                         / LDA  S 1 3132 2520 3132
                                         / LOD  L R 0 0 36
                                         / IXA  L 36
                                         / STR  A R 0 144 36
                                         / COMM ----BB 09
                                         /L9 LAB  0
                                         / COMM   begin if list[i] > list[i+1] then
 LOC        1       45 0                 / LOC        1       45 0
 LDA  S 1 3096 2520 3132                 / LOD  A R 0 36 36
 LOD  L S 1 8244 36                      / ILOD J 0 36
 IXA  L 36                               / NSTR J R 0 108 36
 LOD  L S 1 8244 36                      / LOD  A R 0 144 36
 LDC  L 36 1                             / ILOD J 0 36
 ADD  L                                  / NSTR J R 0 180 36
 LDA  S 1 3096 2520 3132                 / GRT  J
 SWP  A L                                / FJP  L10
 IXA  L 36                               / COMM ----BB 10
 ILOD J 0 36                             / COMM           swap(list[i], list[i+1]);
 SWP  J A                                /
 ILOD J 0 36                             /
 SWP  J J                                /
 GRT  J                                  /
 FJP  L10                                /
 COMM          swap(list[i], list[i+1]) /
 LOC        1       46 0                 / LOC        1       46 0
 COMM starting merge of call to       BU / COMM begin t := x;
 LDA  S 1 3096 2520 3132                 /
 LOD  L S 1 8244 36                      /
 IXA  L 36                               /
 STR  A M 1 0 36                         /
 LOD  L S 1 8244 36                      /
 LDC  L 36 1                             /
 ADD  L                                  /
 LDA  S 1 3096 2520 3132                 /
 SWP  A L                                /
 IXA  L 36                               /
 STR  A M 1 36 36                        /
 COMM code start for  BUB$01             /
 COMM SWAP                               /
 OPTN TSOURCELOC 385                     /
 COMM begin t := x;                      /
 LOC        1       22 0                 / LOC        1       22 0
 LOD  A M 1 0 36                         / LOD  J R 0 108 36
 ILOD J 0 36                             / STR  J R 0 216 36
 STR  J S 1 8352 36                      / COMM  x := y;
```

```
 COMM  x := y;                          /
 LOC        1       23 0                / LOC        1       23 0
 LOD  A M 1 36 36                       / LOD  J R 0 180 36
 ILOD J 0 36                            / LOD  A R 0 36 36
 LOD  A M 1 0 36                        / SWP  A J
 SWP  A J                               / ISTR J 0 36
 ISTR J 0 36                            / COMM  y := t
 COMM  y := t                           /
 LOC        1       24 0                / LOC        1       24 0
 LOD  J S 1 8352 36                     / LOD  J R 0 216 36
 LOD  A M 1 36 36                       / LOD  A R 0 144 36
 SWP  A J                               / SWP  A J
 ISTR J 0 36                            / ISTR J 0 36
 OPTN TSYMLOC 0                         / COMM end;
 COMM end;                              /
 LOC        1       25 0                / LOC        1       25 0
 COMM end of merged call to          BU / COMM ----BB 11
L10 LAB  0                              /L10 LAB  0
 COMM            i := i+1               / COMM            i := i+1
 LOC        1       50 0                / LOC        1       50 0
 LOD  L S 1 8244 36                     / LOD  A R 0 36 36
 LDC  L 36 1                            / LDC  J 36 1
 ADD  L                                 / CVT  A J
 STR  L S 1 8244 36                     / ADD  A
 LOD  L S 1 8244 36                     / STR  A R 0 36 36
 LOD  J S 1 8280 36                     / LOD  A R 0 144 36
 CVT2 J L                               / LDC  J 36 1
 LES  J                                 / CVT  A J
 TJP  L9                                / ADD  A
L11 LAB  0                              / STR  A R 0 144 36
L7 LAB  0                               / LOD  L R 0 0 36
 COMM    top := top - 1                 / LDC  L 36 1
                                        / ADD  L
                                        / STR  L R 0 0 36
                                        / LOD  L R 0 0 36
                                        / CVT  J L
                                        / LOD  J R 0 72 36
                                        / LES  J
                                        / TJP  L9
                                        / COMM ----BB 12
                                        /L11 LAB  0
                                        / COMM ----BB 13
                                        /L7 LAB  0
                                        / COMM    top := top - 1
 LOC        1       52 0                / LOC        1       52 0
 LOD  J S 1 8280 36                     / LOD  J R 0 72 36
 LDC  L 36 1                            / LDC  J 36 1
 CVT  J L                               / SUB  J
 SUB  J                                 / STR  J R 0 72 36
 STR  J S 1 8280 36                     / LOD  J R 0 72 36
 LOD  J S 1 8280 36                     / LDC  J 36 1
 LDC  L 36 1                            / GRT  J
 CVT  J L                               / TJP  L6
 GRT  J                                 / COMM ----BB 14
 TJP  L6                                /L12 LAB  0
L12 LAB  0                              / COMM ----BB 15
L4 LAB  0                               /L4 LAB  0
```

E2. DEC 10

Unoptimized DEC 10 Code

```
; -- 40/1   top := 70;      MOVEI  1 ,70
   MOVEM  1 ,BUB$DA+230
; -- 42/1   while top > 1 do
   CAIG   1 ,1
   JRST   $4
$5     :
$6     :
; -- 43/1       begin i := 1;
   MOVEI  1 ,1
   MOVEM  1 ,BUB$DA+229
; -- 44/1    while i < top do
   CAML   1 ,BUB$DA+230
   JRST   $7
$8     :
$9     :
; -- 45/1       begin if list[i] > list[i+1] then
   MOVEI  1 ,BUB$DA+86
   ADD    1 ,BUB$DA+229
   MOVE   2 ,BUB$DA+229
   AOS    2 ,2
   MOVE   3 ,0(1 )
   CAMG   3 ,BUB$DA+86(2 )
   JRST   $10
;starting merge of call to       BUB$01
; -- 46/1         swap(list[i], list[i+1]);
   MOVEI  4 ,BUB$DA+86
   ADD    4 ,BUB$DA+229
   MOVEM  4 ,2(FP)
   MOVE   1 ,BUB$DA+229
   AOS    1 ,1
;code start for  BUB$01
;SWAP
   MOVEI  2 ,BUB$DA+86(1 )
   MOVEM  2 ,3(FP)
; -- 22/1                          t := x;
   MOVE   1 ,0(4 )
   MOVEM  1 ,BUB$DA+232
; -- 23/1                          x := y;
   MOVE   1 ,0(2 )
   MOVEM  1 ,0(4 )
; -- 24/1                          y := t;
   MOVE   2 ,BUB$DA+232
   MOVEM  2 ,3(FP)@
;end of merged call to           BUB$01
; -- 25/1
$10    :
; -- 50/1           i := i+1
   AOS    0 ,BUB$DA+229
   MOVE   1 ,BUB$DA+229
   CAMGE  1 ,BUB$DA+230
   JRST   $9
$11    :
$7     :
; -- 52/1    top := top - 1
   SOS    0 ,BUB$DA+230
   MOVE   1 ,BUB$DA+230
   CAILE  1 ,1
   JRST   $6
$12    :
$4     :
```

Optimized DEC 10 Code

```
; -- 40/1   top := 70;
; -- 42/1   while top > 1 do
$5     :
   MOVEI  7 ,70
$6     :
; -- 43/1      begin i := 1;
; -- 44/1    while i < top do
   CAIG   7 ,1
   JRST   $7
$8     :
   MOVEI  6 ,1
   MOVEI  6 ,BUB$DA+86(6 )
   MOVEI  9 ,BUB$DA+87(5 )
$9     :
; -- 45/1       begin if list[i] > list[i+1] then
   MOVE   8 ,0(6 )
   MOVE   1 ,0(6 )
   MOVE   10,0(9 )
   CAMG   1 ,0(9 )
   JRST   $10
; -- 46/1         swap(list[i], list[i+1]);
; -- 22/1                          t := x;
   MOVE   11,8
; -- 23/1                          x := y;
   MOVEM  10,0(6 )
; -- 24/1                          y := t;
   MOVEM  11,0(9 )
; -- 25/1
$10    :
; -- 50/1           i := i+1
   MOVEI  6 ,1(6 )
   MOVEI  9 ,1(9 )
   AOS    0 ,6
   CAMGE  6 ,7
   JRST   $9
$11    :
$7     :
; -- 52/1    top := top - 1
   SOS    0 ,7
   CAILE  7 ,1
   JRST   $6
$12    :
$4     :
```

E3. 68000

Unoptimized 68000 Code

```
|  40  top := 70;
        movl    #70,bubblesort$dat+1148
|  42  while top > 1 do
        cmpl    #1,bubblesort$dat+1148
        jle     $4
$5:
$6:
|  43     begin i := 1;
        movl    #1,bubblesort$dat+1144
|  44   while i < top do
        movl    bubblesort$dat+1144,d0
        cmpl    bubblesort$dat+1148,d0
```

```
        jge     $7
$8:
$9:
|  45     begin if list[i] > list[i+1] then
        movl    bubblesort$dat+1144,d0
        asll    #2,d0
        movl    #bubblesort$dat+572,a0
        movl    bubblesort$dat+1144,d1
        addql   #1,d1
        asll    #2,d1
        movl    #bubblesort$dat+572,a1
        movl    a0@(0,d0:L),d0
        cmpl    a1@(0,d1:L),d0
        jle     $10
|  46         swap(list[i], list[i+1]);
        movl    bubblesort$dat+1144,d1
        asll    #2,d1
        addl    d1,a0
        movl    a0,a6@(-4)
        movl    bubblesort$dat+1144,d0
        addql   #1,d0
        asll    #2,d0
        addl    d0,a1
        movl    a1,a6@(-8)
|  22
        movl    a6@(-4),a0
        movl    a0@,bubblesort$dat+1156
|  23
        movl    a6@(-8),a1
        movl    a1@,a0@
|  24
        movl    a6@(-8),a0
        movl    bubblesort$dat+1156,a0@
|  26
$10:
|  50          i := i+1
        addql   #1,bubblesort$dat+1144
        movl    bubblesort$dat+1144,d0
        cmpl    bubblesort$dat+1148,d0
        jlt     $9
$11:
$7:
|  52   top := top - 1
        subql   #1,bubblesort$dat+1148
        cmpl    #1,bubblesort$dat+1148
        jgt     $6
$12:
$4:
```

Optimized 68000 Code

```
|  40  top := 70;
|  42  while top > 1 do
$5:
        moveq   #70,d4
$6:
|  43     begin i := 1;
|  44   while i < top do
        cmpl    #1,d4
        jle     $7
$8:
        moveq   #1,d3
        movl    d3,d0
```

```
        asll    #2,d0
        movl    #bubblesort$dat+572,a0
        lea     a0@(0,d0:L),a4
        movl    d3,d1
        asll    #2,d1
        movl    #bubblesort$dat+576,a1
        lea     a1@(0,d1:L),a5
$9:
|  45     begin if list[i] > list[i+1] then
        movl    a4@,d5
        movl    a5@,d6
        cmpl    d6,d5
        jle     $10
|  46         swap(list[i], list[i+1]);
|  22
        movl    d5,d7
|  23
        movl    d6,a4@
|  24
        movl    d7,a5@
|  25
$10:
|  50          i := i+1
        addql   #4,a4
        addql   #4,a5
        addql   #1,d3
        cmpl    d4,d3
        jlt     $9
$11:
$7:
|  52   top := top - 1
        subql   #1,d4
        cmpl    #1,d4
        jgt     $6
$12:
$4:
```

E4. VAX

Unoptimized VAX Code

```
# -- 40/1    top := 70;
   movl    $70,bubblesort_dat+1148
# -- 42/1    while top > 1 do
   cmpl    bubblesort_dat+1148,$1
   jleq    _4
_5:
_6:
# -- 43/1        begin i := 1;
   movl    $1,bubblesort_dat+1144
# -- 44/1      while i < top do
   movl    bubblesort_dat+1144,r10
   cmpl    r10,bubblesort_dat+1148
   jgeq    _7
_8:
_9:
# -- 45/1          begin if list[i] > list[i+1] then
   addl3   bubblesort_dat+1144,bubblesort_dat+1144,r10
   addl2   r10,r10
   addl2   $bubblesort_dat+572,r10
   addl3   $1,bubblesort_dat+1144,r9
   addl2   r9 ,r9
```

```
    addl2   r9 ,r9
    addl2   $bubblesort_dat+572,r9
    movl    0(r10),r8
    movl    0(r9 ),r7
    cmpl    r8 ,r7
    jleq    _10
#starting merge of call to        BUBBLESORT$01
# -- 46/1          swap(list[i], list[i+1]);
    addl3   bubblesort_dat+1144,bubblesort_dat+1144,r6
    addl2   r6 ,r6
    addl2   $bubblesort_dat+572,r6
    movl    r6 ,-4(fp )
    addl3   $1,bubblesort_dat+1144,r5
    addl2   r5 ,r5
    addl2   r5 ,r5
    addl2   $bubblesort_dat+572,r5
#code start for  BUBBLESORT$01
#SWAP
    movl    r5 ,-8(fp )
# -- 22/1
    movl    *-4(fp ),bubblesort_dat+1156
# -- 23/1
    movl    *-8(fp ),*-4(fp )
# -- 24/1
    movl    bubblesort_dat+1156,*-8(fp )
#end of merged call to             BUBBLESORT$01
# -- 25/1
_10:
# -- 50/1            i := i+1
    incl    bubblesort_dat+1144
    movl    bubblesort_dat+1144,r10
    cmpl    r10,bubblesort_dat+1148
    jlss    _9
_11:
_7:
# -- 52/1     top := top - 1
    decl    bubblesort_dat+1148
    cmpl    bubblesort_dat+1148,$1
    jgtr    _6
_12:
_4:
```

Optimized VAX Code

```
# -- 40/1   top := 70;
# -- 42/1   while top > 1 do
_5:
    movl    $70,r7
_6:
# -- 43/1       begin i := 1;
# -- 44/1     while i < top do
    cmpl    $1,r7
    jgeq    _7
_8:
    movl    $1,r5
    addl3   r5 ,r5 ,r4
    addl2   r4 ,r4
    addl2   $bubblesort_dat+572,r4
    movl    r4 ,r6
    addl3   r5 ,r5 ,r3
    addl2   r3 ,r3
    addl2   $bubblesort_dat+570,r3
    movl    r3 ,r9
```

```
_9:
# -- 45/1       begin if list[i] > list[i+1] then
   movl   0(r6 ),r8
   movl   0(r9 ),r10
   movl   0(r6 ),r4
   movl   0(r9 ),r3
   cmpl   r4 ,r3
   jleq   _10
# -- 46/1         swap(list[i], list[i+1]);
# -- 22/1
   movl   r8 ,r11
# -- 23/1
   movl   r10,0(r6 )
# -- 24/1
   movl   r11,0(r9 )
# -- 25/1
_10:   # -- 50/1           i := i+1
   addl2  $4,r6
   addl2  $4,r9
   incl   r6
   movl   r6 ,r4
   cmpl   r4 ,r7
   jlss   _9
_11:
_7:
# -- 52/1    top := top - 1
   decl   r7
   cmpl   r7 ,$1
   jgtr   _6
_12:
_4:
```

E5. MIPS

Note that the MIPS code generator incorporates the local optimization portion of UOPT, so that local optimization is always performed.

Unoptimized MIPS Code

```
# -- 40/1   top := 70;
# -- 42/1   while top > 1 do
        mov #70,r1
        st r1,FPinit+(-4)
LBUB85:
LBUB86:
# -- 43/1      begin i := 1;
        mov #1,r1
        st r1,FPinit+(-5)
# -- 44/1    while i < top do
        ld FPinit+(-4),r2
        bge #1,r2,LBUB87
LBUB88:
LBUB89:
# -- 45/1       begin if list[i] > list[i+1] then
        ld #FPinit-147,r1
        ld FPinit+(-5),r2
        ld [r1+r2],r3
        ld #FPinit-148,r4
        ld [r4+r2],r5
        ble r5,r3,LBUB810
# -- 46/1         swap(list[i], list[i+1]);
```

```
        add r4,r2,r6
        st r6,-306[r15]
        add r1,r2
        st r2,-305[r15]
        st r5,FPinit+(-2)
        st r3,0[r6]
        st r5,0[r2]
LBUB810:
# -- 50/1          i := i+1
        ld FPinit+(-5),r1
        add #1,r1
        st r1,FPinit+(-5)
        ld FPinit+(-4),r2
        blt r1,r2,LBUB89
LBUB811:
LBUB87:
# -- 52/1    top := top - 1
        ld FPinit+(-4),r1
        sub #1,r1
        st r1,FPinit+(-4)
        blt #1,r1,LBUB86
LBUB812:
LBUB84:
```

Optimized MIPS Code

```
# -- 40/1   top := 70;
# -- 42/1   while top > 1 do
LBUB85:
        mov #70,r12
LBUB86:
# -- 43/1      begin i := 1;
# -- 44/1    while i < top do
        bge #1,r12,LBUB87
LBUB88:
        mov #1,r14
        ld #FPinit-148,r1
        add r1,r14,r13
        ld #FPinit-147,r2
        add r2,r14,r10
LBUB89:
# -- 45/1       begin if list[i] > list[i+1] then
        ld 0[r13],r11
        ld 0[r10],r9
        ble r11,r9,LBUB810
# -- 46/1         swap(list[i], list[i+1]);
        mov r11,r8
        st r9,0[r13]
        st r8,0[r10]
LBUB810:
# -- 50/1          i := i+1
        add #1,r13
        add #1,r10
        add #1,r14
        blt r14,r12,LBUB89
LBUB811:
LBUB87:
# -- 52/1    top := top - 1
        sub #1,r12
        blt #1,r12,LBUB86
LBUB812:
LBUB84:
```

E6. FOM

Unoptimized FOM Code

```
; -- 40/1   top := 70;
       AddI     kaa, 0, $C70, TOP
; -- 42/1   while top > 1 do
       GtI      aka, TOP, 1, $T6
       IfLF     aa., $T6, L$4
       Nop
       Nop
       Label    L$5
       Label    L$6
; -- 43/1      begin i := 1;
       AddI     kka, 0, 1, I
; -- 44/1    while i < top do
       LtI      aaa, I, TOP, $T7
       IfLF     aa., $T7, L$7
       Nop
       Nop
       Label    L$8
       Label    L$9
; -- 46/1       begin if list[i] > list[i+1] then
       AddI     kas, -1, LIST,
       AddI     sas, , I,
       LoadI    sk., , 0
       AddI     kaa, -1, LIST, $T9
       AddI     aka, I, 1, $T10
       AddI     aaa, $T9, $T10, $T9
       LoadI    ak., $T9, 0
       GtI      qqa, Load        IfLF     aa., $T11, L$10
; -- 46/1        swap(list[i], list[i+1]);
       Nop
Nop
       AddI     kas, -1, LIST,
       AddI     sas, , I,
       AddI     ksa, 0, , T$1
       AddI     kas, -1, LIST,
       AddI     aka, I, 1, $T14
       AddI     sas, , $T14,
       AddI     ksa, 0, , T$2
       LoadI    ak., T$1, 0
       AddI     kqa, 0, Load         LoadI    ak., T$2, 0
       AddI     kqa, 0, Load         StoI     ak., T$1, 0
       AddI     kaa, 0, T, T
       StoI     ak., T$2, 0
       Label    L$10
; -- 50/1           i := i+1
       AddI     aks, I, 1,
       AddI     ksa, 0, , I
       LtI      aaa, I, TOP, $T17
       IfLT     aa., $T17, L$9
       Nop
       Nop
       Label    L$11
       Label    L$7
; -- 52/1    top := top - 1
       SubI     aks, TOP, 1,
       AddI     ksa, 0, , TOP
       GtI      aka, TOP, 1, $T19
       IfLT     aa., $T19, L$6
       Nop
       Nop
```

```
        Label    L$12
        Label    L$4
```

Optimized FOM Code

```
; -- 40/1    top := 70;
; -- 42/1    while top > 1 do
        Label    L$5
        AddI     kaa, 0, $C70, TOP
        Label    L$6
; -- 43/1       begin i := 1;
; -- 44/1     while i < top do
        LtI      kaa, 1, TOP, $T5
        IfLF     aa., $T5, L$7
        Nop
        Nop
        Label    L$8
        AddI     kka, 0, 1, I
        AddI     kas, -1, LIST,
        AddI     sas, , I,
        AddI     ksa, 0, , T$1
        AddI     aas, LIST, I,
        AddI     ksa, 0, , T$3
        Label    L$9
; -- 45/1       begin if list[i] > list[i+1] then
        LoadI    ak., T$1, 0
        AddI     kqa, 0, Load        LoadI    ak., T$3, 0
        AddI     kqa, 0, Load        GtI      aaa, T$2, T$4, $T8
        IfLF     aa., $T8, L$10
; -- 46/1        swap(list[i], list[i+1]);
        Nop
        Nop
        AddI     kaa, 0, T$2, T
        AddI     kaa, 0, T$4, T$4
        StoI     ak., T$1, 0
        AddI     kaa, 0, T, T
        StoI     ak., T$3, 0
        Label    L$10
; -- 50/1           i := i+1
        AddI     aks, T$1, 1,
        AddI     ksa, 0, , T$1
        AddI     aks, T$3, 1,
        AddI     ksa, 0, , T$3
        AddI     aks, I, 1,
        AddI     ksa, 0, , I
        LtI      aaa, I, TOP, $T12
        IfLT     aa., $T12, L$9
        Nop
        Nop
        Label    L$11
        Label    L$7
; -- 52/1    top := top - 1
        SubI     aks, TOP, 1,
        AddI     ksa, 0, , TOP
        GtI      aka, TOP, 1, $T14
        IfLT     aa., $T14, L$6
        Nop
        Nop
        Label    L$12
        Label    L$4
```

E7. S-1

Unoptimized S-1 Code

```
; -- 40/1    top := 70;
   Mov.S.S       BUBBL$DA+872,#70
; -- 42/1    while top > 1 do
   Skp.Gtr.S     BUBBL$DA+872,#1
   SJmp          $4
$5:
$6:
; -- 43/1       begin i := 1;
   Mov.S.S       BUBBL$DA+868,#1
; -- 44/1     while i < top do
   Skp.Lss.S     BUBBL$DA+868,BUBBL$DA+872
   SJmp          $7
$8:
$9:
; -- 45/1          begin if list[i] > list[i+1] then
   Shfa.Lf.S     RTA,BUBBL$DA+868,#2
   Inc.s         RTB,BUBBL$DA+868
   Skp.Gtr.S     BUBBL$DA+296[RTA],BUBBL$DA+296[RTB]↑2
   SJmp          $10
; -- 46/1            swap(list[i], list[i+1]);
   Shfa.Lf.S     RTA,BUBBL$DA+868,#2
   Movp.P.A      (FP )0,BUBBL$DA+296[RTA]
   Inc.s         RTA,BUBBL$DA+868
   Movp.P.A      (FP )4,BUBBL$DA+296[RTA]↑2
; -- 22/1
   Mov.S.S       BUBBL$DA+880,(FP )0@
; -- 23/1
   Mov.S.S       (FP )0@,(FP )4@
; -- 24/1
   Mov.S.S       (FP )4@,BUBBL$DA+880
; -- 25/1
$10:
; -- 50/1             i := i+1
   Inc.s         BUBBL$DA+868,BUBBL$DA+868
   Skp.Geq.S     BUBBL$DA+868,BUBBL$DA+872
   SJmp          $9
$11:
$7:
; -- 52/1     top := top - 1
   Dec.s         BUBBL$DA+872,BUBBL$DA+872
   Skp.Leq.S     BUBBL$DA+872,#1
   SJmp          $6
$12:
$4:
```

Optimized S-1 Code

```
; -- 40/1    top := 70;
; -- 42/1    while top > 1 do
$5:
   Mov.S.S       R24,#70
$6:
; -- 43/1       begin i := 1;
; -- 44/1     while i < top do
   Skp.Lss.S     #1,R24
   SJmp          $7
$8:
   Mov.S.S       R22,#1
   Movp.P.A      R23,BUBBL$DA+296[R22]↑2
```

```
   Movp.P.A     R26,BUBBL$DA+300[R22]↑2
$9:
; -- 45/1       begin if list[i] > list[i+1] then
   Mov.S.S      R25,(R23)0
   Mov.S.S      RTA,(R23)0
   Mov.S.S      R27,(R26)0
   Skp.Gtr.S    RTA,(R26)0
   SJmp         $10
; -- 46/1         swap(list[i], list[i+1]);
; -- 22/1
   Mov.S.S      R28,R25
; -- 23/1
   Mov.S.S      (R23)0,R27
; -- 24/1
   Mov.S.S      (R26)0,R28
; -- 25/1
$10:
; -- 50/1           i := i+1
   Mov.S.S      RTA,(R26)0
   Movp.P.A     R23,(R23)0[RTA]
   Movp.P.A     R26,(R23)0[R26]
   Inc.s        R22,R22
   Skp.Geq.S    R22,R24
   SJmp         $9
$11:
$7:
; -- 52/1    top := top - 1
   Dec.s        R24,R24
   Skp.Leq.S    R24,#1
   SJmp         $6
$12:
$4:
```