Register Allocation
and
Phase Interactions
in
Retargetable Optimizing Compilers

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

at the

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Manuel Enrique Benitez

ABSTRACT

Retargetability is increasingly becoming a necessary property in optimizing compilers. Because register allocation is essential to high-quality code generation, compilers whose register allocators assume a particular register file organization sacrifice retargetability. Also, the strategies used to allocate registers on optimizing compilers that incorporate many code improvement phases tend to over-commit register resources and expect the register assigner to compensate for poor allocation decisions. State-of-the-art register assignment techniques can compensate to a point, but invariably emit poor code when confronted with small register sets or very high register demand. The research presented here will show that:

- when simplicity, retargetability, code quality and execution time are considered, a simple register allocation strategy that allocates registers on a first-come, first-served basis while preventing the code improvement phases from over-committing the registers is the most effective of the various retargetable register allocation techniques evaluated,
- phase iteration, a technique that re-attempts code improvements as additional opportunities to apply them arise, is an effective way of reducing interactions between the tasks performed by an optimizing compiler and simplify the ordering of these tasks,
- it is possible to integrate the code improvement and register allocation tasks performed by an optimizing compiler in order to improve the quality of the code produced by the compiler and increase its retargetability, and
- machine-level global code improvement techniques allow retargetable optimizing compilers to produce code whose quality meets or exceeds the quality of the code produced by traditional production compilers.

This dissertation also includes a detailed description of the experimental frameworks and the register deprivation measurement techniques that were developed to support the design, implementation and evaluation of the machine-level register allocation and code improvements algorithms presented here.

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy (Computer Science)

Author

This dissertation has been read and approved by the Examining Committee:

Dissertation Advisor

Committee Chairman

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering and Applied Science

ACKNOWLEDGMENTS

I would like to thank everyone who has, in some way, contributed to this work. This work would not have been possible without the support, guidance and encouragement of many.

First and foremost, I would like to thank my advisor, Jack Davidson, for the guidance and inspiration that he so willingly provided throughout this long endeavor. I am indebted to his insight and his expertise.

The members of my examining committee, Jack Davidson, William Wulf, Andrew Grimshaw, Barry Johnson and John Knight, provided many helpful comments and devoted much time to ensure that this work measures up to the standards of the University of Virginia and the scientific community. Robert Cook and Joseph Kent, although not part of my committee, also made valuable contributions.

Many of my fellow students deserve more than just the passing mention that they will get here. Mark Bailey provided many useful suggestions, never tired of reading my tired prose and pointed out many of the weaknesses in my logic. David Whalley's work with *ease* and some of his machine descriptions provided me with much needed building blocks. Ann Holler provided much needed encouragement and convinced me to make the most of my limited time as a graduate student. Sanjay Jinturkar, Bruce Childers and Michael Alexander also contributed in many ways.

This work would not have been possible without the help of the current and erstwhile staff at the department of computer science: Mark Smith, Gina Bull, Ray Lubinsky and Ann Bailey.

I would also like to thank my running partners, Jim French, Jeff Salowe and Bob Downer for going the extra mile(s) with me and making my last year here a memorable one.

I would also like to acknowledge the financial support of the research presented in this dissertation provided by the National Science Foundation's Grant number CCR-9214904.

Finally, I wish to thank my family for all their love and encouragement. It is to them that I dedicate this dissertation.

To my Mother

Make it as simple as possible—and no simpler.
—Nicklaus Wirth

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1

INTRODUCTION

Optimizing compilers capable of producing high-quality machine code are frequently used to help satisfy the ever-growing demand for computer system performance. Because of their complexity, the amount of time and effort required to design, implement and validate these optimizing compilers can be justified only if they serve a large number of users over a long period of time. At the current rate of progress, however, existing architectures are becoming obsolete and their users migrating to new systems in increasingly short periods of time. As this trend continues, it will become harder to justify the development of optimizing compilers that are not retargetable enough to migrate along with their users to new architectures.

Unfortunately, retargetability is not easily retrofitted into an optimizing compiler. The ability to operate across a broad spectrum of architectures must be incorporated not only into the parts of the compiler that generate machine code, but also those that perform code improvement transformations and manage resources. In essence, retargetability is a pervasive attribute that must be designed into the overall compilation strategy starting with the intermediate language, which is the internal representation of the code that the compiler performs code improvement transformations on.

Recent developments suggest that intermediate languages that can represent actual target machine instructions are particularly well-suited for use in high-quality retargetable optimizing compilers that need to exploit unique architectural features. Because the majority of existing retargetable compilers use intermediate languages that represent architecture-independent high-level code, however, little is known about how to structure code improvement algorithms that operate on low-level languages capable of representing machine code so as to maximize their effectiveness and their retargetability. Additionally, there remain some concerns as to whether it possible to extract sufficient high-level information from these intermediate representations to perform effective code improvements.

Good register allocation is essential to an optimizing compiler because most code improvement transformations require registers. The number and type of registers provided by an architecture, however, varies widely and is one of the major factors limiting the effectiveness of the allocator. In spite of this, most retargetable optimizing compilers use register allocation techniques that operate effectively only on a limited range of register sets and, therefore, produce high-quality code only for architectures that provide these kinds of register files. Designing a retargetable register allocation strategy that is effective across a wide range of architectures, however, is particularly challenging because strategies that provide good register allocation on architectures with small register files are quite different from strategies that perform effective allocation on architectures with large register sets.

The following sections expand on this brief introduction, present further evidence to support the notion that high-quality retargetable optimizing compilers should use machine-level intermediate languages and highly-retargetable register allocation algorithms. These sections are followed by a description of the scope and direction of the investigation needed to find answers to some of the unresolved issues associated with these compilation strategies.

1.1 Retargetable optimizing compilers

During the process of translating code written in a high-level programming language into a sequence of low-level machine instructions, a compiler must, at some point, generate machine code from a higher-level representation of the code and find suitable storage locations for the temporary values needed to construct arbitrarily complex expressions from a small set of basic instructions. In addition to performing these tasks, an optimizing compiler must also enhance the quality of the code by applying a comprehensive set of code improvement transformations[1]. Consequently, optimizing compilers are large, complex systems that perform competent code generation, register allocation and code improvement transformations. While the ability to generate correct, high-quality machine code is the main objective of an optimizing compiler, retargetability is gradually playing a greater role in the design of these systems.

1.1.1 What is retargetability?

Retargetability is a measure of how easily a compiler can be modified to produce code for a new architecture. To achieve a high degree of retargetability, the portions of the optimizing compiler that perform

[1] Code improvement transformations are commonly referred to as optimizations, which is a misnomer since code improvements rarely, if ever, result in “optimal” code.

code improvement transformations, allocate registers and generate machine code must be able to tolerate a certain amount of variability in the following architectural components:

- the functionality and the encoding of the instruction set,
- the size, partitioning and regularity of the register set,
- the number, capabilities and organization of the processing units,
- the data types supported and their formats,
- the addressing modes available,
- the width of the data paths,
- the degree of pipelining,
- the number of levels in the memory hierarchy,
- the size, relative speed and alignment requirements of the levels in the memory hierarchy and
- the mechanisms provided to support high-level languages.

This list is incomplete because many of the less common architectural elements have been omitted. Also, architectural innovations constantly add new items to this list. The impact of this architectural diversity on an optimizing compiler is so pervasive that retargetability cannot be easily retrofitted into the compiler, but must be incorporated into the overall compilation strategy.

1.1.2 Why is retargetability important?

As the quality of the code that optimizing compilers are expected to produce rises in response to the growing demand for system performance, the effort required to design, implement and validate an optimizing compiler increases. While it might seem unreasonable to further complicate this process by making the compiler highly retargetable, doing so is justified because:

- the effort spent in developing a retargetable optimizing compiler can be amortized over a longer period of time and a greater number of users than its less-retargetable counterparts,
- a longer life-span promotes the development of new code improvements and the enhancement of existing ones, which improves the overall quality of the code produced by the compiler,
- retargeting an existing compiler is a cost-effective way (and often the only cost-effective way) to produce high-quality machine code for processors with limited applications or few users and
- a retargetable compiler is a useful architectural design tool.

Rapid advances in VLSI technology, processor design and memory systems effectively reduce the useful life-span of existing architectures. Shorter life-span, in turn, make it increasingly difficult to justify the time and effort required to produce high-quality optimizing compilers that are bound to a single architecture. The life-span of a retargetable optimizing compiler, however, is independent of any one architecture and, thus, is more likely to outlive most, if not all, of its less retargetable contemporaries.

Another argument for the development of a retargetable optimizing compiler is that its user base can be quickly expanded with relatively little effort by retargeting it to a popular architecture. For this strategy to work, however, it is essential for the compiler to compare favorably against the other compilers available for that architecture. The current popularity of *gcc* [STAL89], for example, is due not only to the large number of systems for which it can produce code, but also to its low cost and the reasonable quality of the code that it produces.

The time saved moving a retargetable optimizing compiler to the next generation architecture instead of constructing an entirely new compiler for it can be spent enhancing existing code improvements or developing new ones. This not only improves the quality of the code retroactively across all of the architectures to which the compiler has targeted, but also makes the compiler more competitive against other compilers and subsequently increases its user base. Sometimes new code improvements are needed to utilize a feature provided by a new architecture but, because similar features are likely to appear in other architectures, the new transformation will improve the overall quality of the code produced by the optimizer across different architectures.

There is a substantial group of useful architectures, primarily intended to support embedded systems and signal processing, for which no compilers capable of producing high-quality machine code exist. One reason for this shortcoming is that the amount of code developed on these systems is too small to justify the development of a new optimizing compiler. The simpler task of adapting a highly-retargetable optimizing compiler to such a system, however, is more likely to be cost effective.

Retargetable optimizing compilers provide computer architects with the ability to gauge the impact of new architectural components and features on existing compiler technologies. Because a retargetable compiler can be targeted to the new architecture early in the design process, it can provide valuable feedback to the architect. This feedback can help to resolve important design issues such as how useful certain instructions, operations or addressing modes are, how many registers of each type should be provided and how many pipeline stages the processor should have.

1.1.3 Intermediate representations

Intermediate representation are used to isolate significant portions of the compilation process from the idiosyncrasies of the source language and the target architecture. This isolation is attained using the following strategy: the *front end* of the compiler transforms the source code into semantically equivalent intermediate code which is then improved by a series of code improvement transformations before it is transformed into machine code by the *back end*. The choice of an intermediate representation seriously affects the retargetability of an optimizing compiler [CHOW83]. The types of intermediate representations have been used in optimizing compilers are fixed, high-level intermediate representations and low-level representations.

Because fixed, high-level representations are independent of the target architecture, they require no modification to retarget. Likewise, the code improvement algorithms that operate on these representations remain unchanged across different architectures. Together, these two factors account for much of the retargetability that high-level representations provide. Unfortunately, high-level representations provide the code improvement algorithms with little control over all but the most common architectural features, thereby limiting the effectiveness of code improvement transformations [BENI94]. These limitations apply not only to code improvements like *peephole optimization* [MCKE65] and *instruction scheduling* [HENN83], which exploit specific architectural features, but also to code improvements like *loop-invariant code motion* [AHO86] and *evaluation order determination* [SETH70], whose broad applicability has fueled the myth that they can be effectively performed on high-level representations. The fallacy of this assumption is illustrated with a few simple examples. First, consider that after all loop-invariant items have been moved out of a high-level representation of a loop, the machine code produced for the loop could still contain part of a loop-invariant computation of a high-level constant expression that is too large to fit into an operand field on the target machine. Similarly, optimal evaluation orders cannot be produced on representations that obscure the register requirements of the machine code [MCKU84, DAVI86].

The factors that limit the effectiveness of code improvements transformations that operate on high-level representations can be avoided by using low-level representations. Unlike their high-level counterparts, low-level representations encode the semantics of a wide range of machine instructions. One of the most commonly used notations used to represent low-level code is the *register transfer* [BELL71]. Register transfers have been successfully used in a number retargetable optimizing compilers including PO [DAVI81], YC [DAVI84b], *vpo* [BENI89] and *gcc* [STAL89]. Regardless of the underlying notation, low-level representations allow the code improvement algorithms to operate at the machine level where all of the target machine nuances are exposed and all transformations, including those that exploit specific architectural features, can be performed effectively. This fact alone makes low-level representations more suitable for use in high-quality retargetable optimizing compilers than high-level representations.

While high-level representations shield code improvement algorithms from the target machine, low-level representations expose them to the target architecture. Because of this, significantly more foresight and planning is required to use these representations in a retargetable framework. Since architecture-specific issues are not hidden form low-level code improvement algorithms, retargetability entails isolating the architecture-specific portions of the algorithms behind an appropriate interface to limit the amount of code that the user must examine during the targeting process. Equally important is the need to extract enough high-level information from low-level code to support complex code improvement transformations. For example, a particular transformation may require extensive alias analysis to ensure that all of the references to a crucial variable are known. Failure to extract such high-level information limits the set of transformations that the compiler can safely apply and degrades the quality of the code produced.

1.1.4 Register allocation

Making effective use of the registers provided by each architecture, which is essential to producing high-quality code, is the job of the register allocator. The register allocation process is profoundly affected by the widely-varying specifics of the registers resources provided by the target architecture. Consequently, neglecting to account for the effects of architectural diversity on the register allocator can severely limit the retargetability of an optimizing compiler.

Most register allocation strategies provide the code improvement algorithms with an inexhaustible supply of registers, called *pseudo-registers*, to use when performing transformations. Under this strategy, the register allocation process consists of assigning actual machine registers to pseudo-registers. Because the set of target registers is finite, the assigner is sometimes unable to assign a target machine register to all of the pseudo-registers and is forced to *spill* some pseudo-register values to memory. Since it takes longer to access a value in memory than it does to access a value in a register, a good assigner strives to spill only values that have a minimal impact on the quality of the code. While this strategy works well when the target architecture has many registers[1], it produces an unacceptable number of spills on architectures that provide few registers. Furthermore, this register allocation strategy does not prevent code improvement transformations from over-committing registers and producing code whose quality is worse than that of the original, unimproved code. The term, *over-optimization*, has been used to describe this situation [AUSL82].

To prevent over-optimization, the compiler can abstain from using pseudo-registers by requiring code improvements transformations to obtain actual target machine registers. Unfortunately, this approach limits the internal representation to a finite set of machine registers and forces the compiler to reuse registers while performing code improvements, thereby introducing a few potential disadvantages. First, the register assignment decisions made by each code improvement algorithms constrain the allocation choices available to subsequent transformations. In compilers that perform code improvements using an infinite set of registers, this problem only occurs while applying low-level code improvements like instruction scheduling [GOOD88]. Second, the need to reuse registers complicates the task of determining which values have been previously computed since each individual expression value is no longer associated with a unique pseudo-register. Third, the process of extending the life of a register value is complicated by the fact that an appropriate register must be allocated from a limited set of registers. Finally, it is difficult to determine that a code transformation that releases at least as many registers as it consumes can be applied when all of the remaining registers are in use. When pseudo-registers are used, there is always a register available to perform these transformations.

1.2 The scope of this investigation

The goal of this research is to develop and evaluate low-level code improvement and register allocation algorithms that are both retargetable and conducive to high-quality code generation. The following sections define the scope and direction of this endeavor.

1.2.1 Architectures

While retargetability is an important attribute of the algorithms developed during the course of this investigation, practical considerations limit the set of architectures used to evaluate these algorithms. Proof of effectiveness will be limited to general-purpose and specialized (e.g., embedded systems, digital signal processing systems) uniprocessors both with and without pipelines, and providing either complex (CISC) or

[1] The exact boundary between many and few registers depends on numerous factors, although it is generally between ten and sixteen registers. More details will be provided in the following chapter.

reduced (RISC) instruction sets. VLIW and superscalar architectures are not specifically excluded and many of the algorithms presented here should perform well on these types of architectures with few modifications. On the other hand, vector, dataflow and parallel processors will not be considered in this dissertation.

With regards to the memory hierarchy, this dissertation is concerned primarily with architectures that provide an individually-addressable set of registers with a minimum of two registers for each basic data type on the processor. There is no upper-bound on the number of registers that the architecture may provide, although the scope over which the code improvement algorithms performed by the experimental framework operate is unlikely to effectively use more than 64 registers of any one basic data type. Although pure accumulator-based architectures and stack-based architectures are not excluded, no effort will be made to determine the effectiveness of any algorithm on these machines.

1.2.2 Code improvement transformations

Because high-quality code cannot be produced with only a few code improvement transformations, this investigation examines a comprehensive set of code improvements. This set consists of both *local* code improvements, whose scope is limited to a single *basic block*[1], and *global* code improvements, whose scope span an entire function. Improvements across function boundaries, called *inter-procedural* improvements, are omitted from this investigation although many of the techniques presented here apply to them.

Of the set of code improvements included in the investigation, the following are usually performed on high-level code:

- *constant folding*, which entails performing operations with constant operands at compile time,
- *dead variable elimination*, which eliminates assignments to values that will not be referenced in the future,
- *copy propagation*, which attempts to eliminate copies of an item by replacing them with a reference to the original instance of the item,
- *evaluation order determination*, which orders instruction sequences to reduce the number of registers needed to compute expressions,
- *common subexpression elimination*, which detects and eliminates redundant computations,
- *loop-invariant code motion*, which moves loop expressions that remain invariant across each iteration of the loop to a location outside the loop
- *loop strength reduction*, which replaces complex expressions that generate simple arithmetic sequences within a loop with simpler expressions that are less expensive to compute,

[1] A basic block consists of a sequence of straight-line code that can only be executed starting with the first instruction in the sequence and exited only through the last instruction [AHO86].

- *induction variable elimination*, which removes unneeded induction values from a loop,
- *dead code elimination*, which deletes code that will never be executed,
- *branch chain elimination*, which changes any branch that transfers control to another branch to branch directly to the destination of the second branch and
- *jump minimization*, which reduces the number of control transfers in a control flow graph (CGF) by performing a topological sort on the CFG.

In addition, *recurrence optimization* [BENI91b], which eliminates memory references by keeping values that will be used in subsequent loop iterations in registers, is included even though it is an application-specific improvement usually performed only by compilers specializing in compiling either scientific or digital processing codes. Finally, the architecture-specific code improvements called *peephole optimization* [MCKE65], *instruction scheduling* [HENN83] and *delay slot filling* [HENN90] are also included.

This dissertation presents newly-developed retargetable algorithms for performing evaluation order determination, loop-invariant code motion, loop strength reduction, induction variable elimination, recurrence optimization, instruction scheduling and delay slot filling on low-level code. These algorithms were developed using *llef* (which stands for *low-level experimental framework*), which was specifically built to support and measure low-level code generation, register allocation and code improvements, and are designed to exploit specific architectural features and isolate the portions of the algorithm that are affected by the target architecture. These algorithms are effective because they can handle special architectural features and retargetable because they are largely independent of the target architecture. To show that these algorithms are retargetable, they have been targeted, along with *llef*, to six different architectures. To show that they are effective, the quality of the code that they produce will be compared with the quality of the code produced by some of the optimizing compilers available for these architectures.

1.2.3 Register allocation

Because code improvement and code generation algorithms cannot produce high-quality code without a adequate supply of registers, developing register allocation strategies that operates effectively across a wide range of architectures is an important part of this investigation. An important feature of this strategy will be freedom from over-optimization. Techniques that can be used to minimize interactions between the register allocation decisions made during code generation and the subsequent code improvement transformations will also be explored.

To prevent over-optimization, code improvement transformations will be performed exclusively on machine code after a local-scope algorithm assigns target registers to the pseudo-registers created by the code generator to hold temporary expression values. The advantage of this strategy is that it allows the compiler to determine which machine registers are used not only after the initial code sequence is generated, but also as code improvement transformations are applied. A separate global-scope algorithm promotes user-defined variables to registers. The code improvement algorithms that perform loop-invariant code motion, loop-strength reduction, recurrence optimization and induction variable elimination use a register allocation algorithm whose scope is the loop that these improvements are transforming. A number of variations on this basic theme will be examined to determine how the register allocator can best ensure that scarce register resources are given to the transformations that will most improve the quality of the code.

The effectiveness of the register allocation algorithms presented in this dissertation will be measured with a *register deprivation* measurement technique that was developed specifically for this purpose. Using this technique, a single architecture can effectively simulate the register set environment presented by a substantial number of different architectures. This allows a pair of architectures, one CISC and one RISC, to be used to gauge the retargetability and the effectiveness of the register allocation strategies developed with the aid of *llef* across a broad spectrum of architectures.

1.3 Structure

The following two chapters give additional background information on code improvement and register allocation along with an overview of previous work that is closely related to the research presented in this dissertation. Chapter four presents the techniques used to perform register deprivation experiments which were developed specifically to measure the efficacy of the algorithms presented in this dissertation and have applications beyond their original purpose. The fifth chapter introduces *llef* the code improvement algorithms developed using it. The results of the measurements performed to determine the efficacy of this new framework are also included in this chapter. Chapter six discusses the algorithms that were developed to perform register allocation strategies that do not suffer from over-optimization even when the target architecture provides very few registers. In addition, this chapter shows the algorithms that were developed to overcome the problems associated with using target registers instead of pseudo-registers to perform code improvement transformations. This chapter closes with a description of an entirely new compilation strategy that is uniquely suited to low-level compilation and was developed using the experience gained in the course of this investigation. Finally, chapter seven concludes this dissertation with a brief summary.

CHAPTER 2

CODE IMPROVEMENTS

Code improvements play an important role in the generation of high-quality machine code. This chapter examines several different strategies for incorporating a substantial set of code improvements into a retargetable optimizing compiler framework. This chapter also presents a brief overview of some of the systems that have influenced the strategies used in the optimizing compiler framework developed to support this research.

2.1 The motivation for comprehensive code improvement suites

The impact of any single code improvement is not uniform throughout all of the code that a compiler generates. While the majority of code improvements substantially enhance the quality of the code regions that exhibit the particular properties that they are designed to detect and improve, they have little effect on the remaining code. Overall, the reduction in execution time produced by a single code improvement rarely exceeds two or three percent when measured on an adequately large set of benchmark programs. Since the magnitude of these improvements is quite modest, optimizing compilers must perform a comprehensive set of code improvements in order to produce high-quality object code. The set of improvements performed by an optimizing compiler is called its code improvement suite.

2.2 Code improvement phases

Implementing a comprehensive set of code improvements and orchestrating their function within a single compiler is a daunting task. To simplify the complexity of this task, individual code improvements are generally implemented as a separate, isolated *phases*. Each phase is invoked at a pre-determined time in the compilation process, performs its code improvement transformations on an intermediate representation of the source code, and returns control to the compiler. This modularization eliminates most of the communication paths and the potential interactions between the various code improvements, thus making the system easier to develop, implement, test and maintain. Isolating the compilation and code improvement process into a series of independent phases has proved so beneficial that it is used in most, if not all, optimizing compilers.

2.3 Phase interactions

Although modularization reduces the number of potential interactions between code improvement phases in optimizing compilers, even modularized systems interact in ways that make it difficult to determine the optimal order in which phases should be performed or preclude any such ordering altogether. Among the factors responsible for these interactions are:

- the competition for scarce machine resources (e.g. registers, easily-accessed memory locations, co-processors, etc.),
- the inability to track changes in the data-flow, control-flow or alias information caused by previous transformations,
- attempting to perform code improvements that are mutually exclusive,
- applying transformations that trigger other transformations and
- changing the representation of the code.

These factors complicate the ordering of the phases in a compiler and cause *phase ordering problems*. Unless steps are taken to minimize these factors, phase ordering problems can severely limit the effectiveness of a code improvement suite.

Phase ordering problems in retargetable compilers are exacerbated by the fact that architecture-specific issues such as the size of the register file and the complexity of the instruction set profoundly affect the optimal phase ordering. Consequently, frameworks that automatically determine effective phase orderings have the serendipitous advantage of simultaneously enhancing both the quality of the emitted code and the retargetability of the compiler. This is because failing to determine an appropriate phase ordering for the target architecture sacrifices some of the code quality that can be expected from a retargetable compiler while forcing the person retargeting the compiler to spend time performing this task reduces its retargetability.

2.4 Organizational structure

The concept of dividing a compiler into a front-end and a back-end was introduced in Section 1.1.3. The organizational structure resulting from this strategy is illustrated in Figure 1. While performing lexical and semantic analysis on the source code, the front-end not only ensures that the input conforms to the syntactic and semantic rules of the source language, but also produces a high-level intermediate representation of the code. Since high-level representations are independent of the target architecture, properly designed front-ends require little target-machine information.

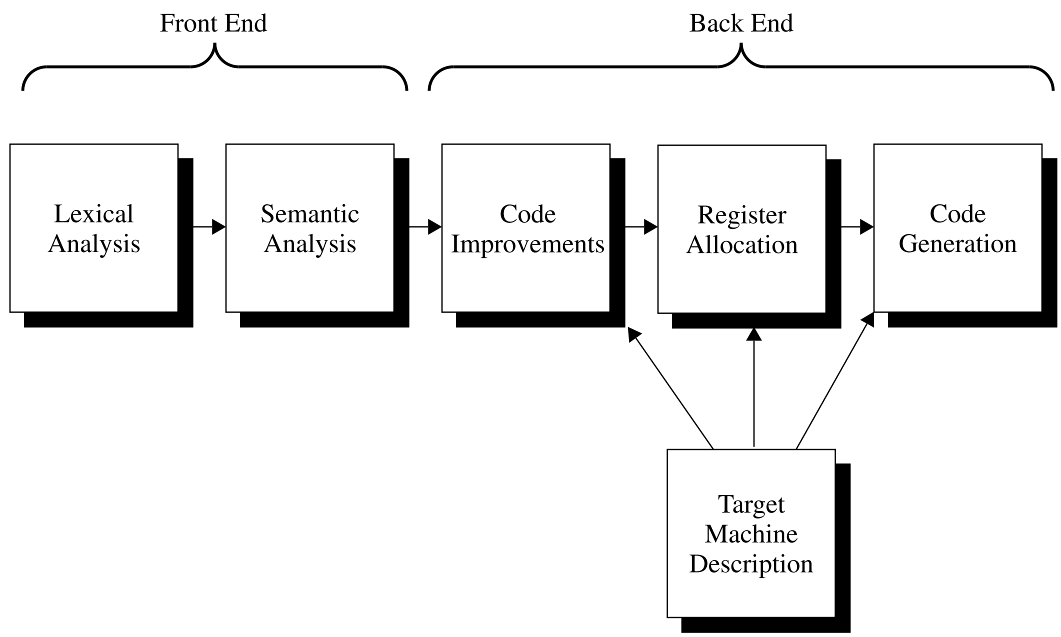


Figure 1: The front and back ends of an optimizing compiler

The back-end of the compiler accepts the high-level code produced by the front-end, applies a series of code improvements to it, determines how the register resources are to be utilized and produces code that the target machine can execute. Since the exact nature of these tasks depend on the target architecture, the back-end must incorporate information about the target machine. The method used to encode this information affects the retargetability of the compiler. Machine descriptions encoded in an *ad-hoc*, diffused fashion are common in systems that are hard to retarget. One the other hand, the target machine descriptions found in highly-retargetable systems are centralized and are accessed by the rest of the back-end through well-defined interfaces.

Retargetable optimizing compilers generally perform code improvements at the intermediate code level, because high-level intermediate code is much less dependent on the target architecture than machine code is. This strategy enhances retargetability by allowing code improvement phases to remain independent of the target architecture and eliminates the need to provide these phases access to detailed architecture-specific information. As a result, the machine description is smaller and the retargeting process is simplified.

As mentioned in Chapter 1, however, most code improvements are either architecture-specific or interact with those that are. Unfortunately, the semantic gap between the high-level intermediate code and the machine code prevents architecture-specific code improvements from being applied effectively on high-level intermediate code. Therefore, to accommodate the most essential architecture-specific code improvements, many back-ends invoke a set of *post-pass* phases after machine code is generated. The back-end structure resulting from this strategy is shown in Figure 2 and suffers from the following shortcomings:

- the majority of code improvements are performed prior to code generation and are therefore unable to fully exploit the capabilities of the target architecture,
- the opportunities created by the post-pass phases to perform code improvements that operate on high-level intermediate code are forfeited unless these improvements are duplicated at the machine code level,
- because of their highly architecture-specific nature, post-pass phases usually cater to specific architectures and are rarely retargetable and
- since the high-level context present in the source code is not as easily extracted from target machine code as it is from the intermediate code, post-pass improvements that require extensive data-flow, control-flow or alias analysis may lose much of their effectiveness.

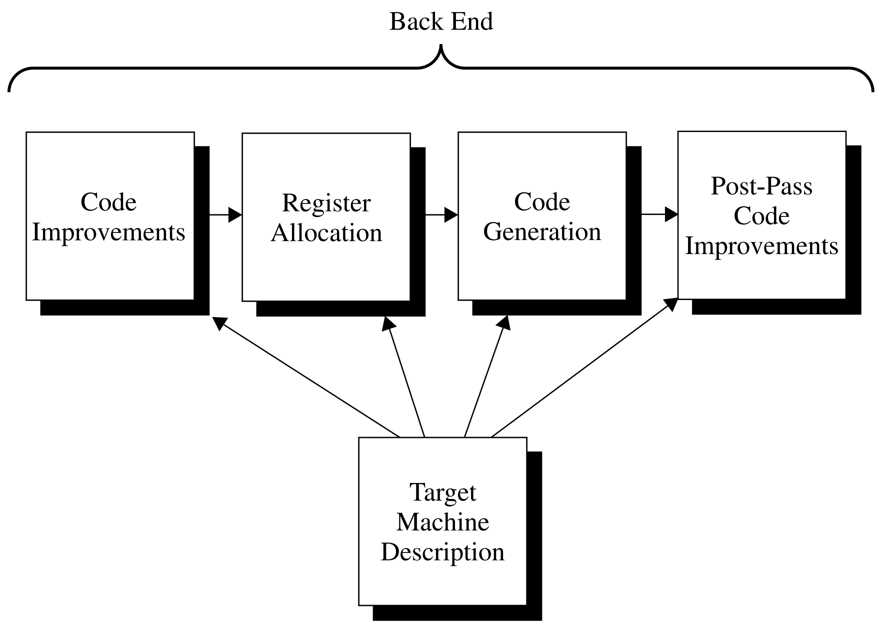


Figure 2: Back-end with post-pass code improvements

Some of the shortcomings of the structure shown in Figure 2 are absent in the organizational framework illustrated in Figure 3, which performs code improvements exclusively on machine code. As a result, the nuances of the target architecture are exposed to the entire code improvement suite, thus eliminating the inefficiencies caused by the semantic gap between the intermediate code and the machine code. Additionally, architecture-independent improvements are applied to machine code and take advantage of the opportunities provided by even the architecture-specific transformations to further improve the code.

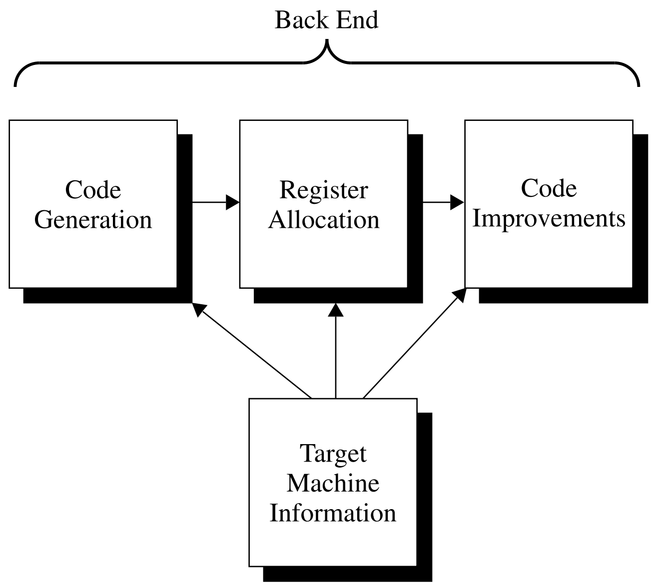


Figure 3: Structure of a back-end that operates exclusively on machine code

Although the framework in Figure 3 does not suffer from some of the disadvantages present in the framework in Figure 2, it is not inherently more retargetable and does not eliminate the need to extract high-level information from machine code. Furthermore, the framework introduces a new set of problems, including:

- since code improvements operate exclusively at the machine level, they usually require more target-architecture information than phases that operate on high-level, architecture-independent code do,
- phase interactions are more acute at the machine level, and
- the impact that code improvement transformations have on code generation and register allocation blurs the sharp boundaries between the back-end components shown in Figure 3.

Fortunately, all of these problems can be solved or significantly mitigated. A high level of retargetability can be obtained using an intermediate representation based on register transfers, which represent target machine instructions using an architecture-independent notation. Because an appropriately devised register transfer notation remains invariant across different architectures, large portions of the optimizing compiler, including the code improvement phases and the high-level information-extraction routines, can be made largely architecture-independent. Carefully executed, this strategy allows a simple interface to a small number of routines to provide all of the architecture-specific information needed to perform code improvements, code generation and register allocation at the machine-level. Finally, since all code improvements are performed on the same representation of the code, phase ordering problems can be reduced by using a *phase iteration* strategy to re-invoke code improvement phase whenever opportunities to perform even previously-applied code improvements arise.

2.5 Related work

The following sections introduce some of the retargetable optimizing compilers that have influenced the design of the optimizing compiler framework described in Chapter 5. These sections focus primarily on the overall structure of each compiler, the nature of the intermediate language on which the bulk of the code improvements are applied and the techniques used to minimize the interactions between the various code improvement phases.

2.5.1 Amsterdam Compiler Kit

The Amsterdam Compiler Kit (ACK) [TANE83] is a set of programs that simplify the task of developing a compiler. Like many modern retargetable compilers, ACK is based on the principle that a single, universal intermediate language, called an UNCOL [STEE61], allows *N* languages to be made available on *M* different architectures with only *N* front-ends plus *M* back-ends instead of *M*×*N* separate compilers. ACK includes front-ends for various languages, an intermediate language code improver that performs global-scope code improvements and peephole optimization on a high-level, machine-independent intermediate language called EM, a code generator, a post-pass code improver, a universal assembler and a linker. The structure of an ACK compiler is shown in Figure 4. EM succeed where previous UNCOLs failed by catering

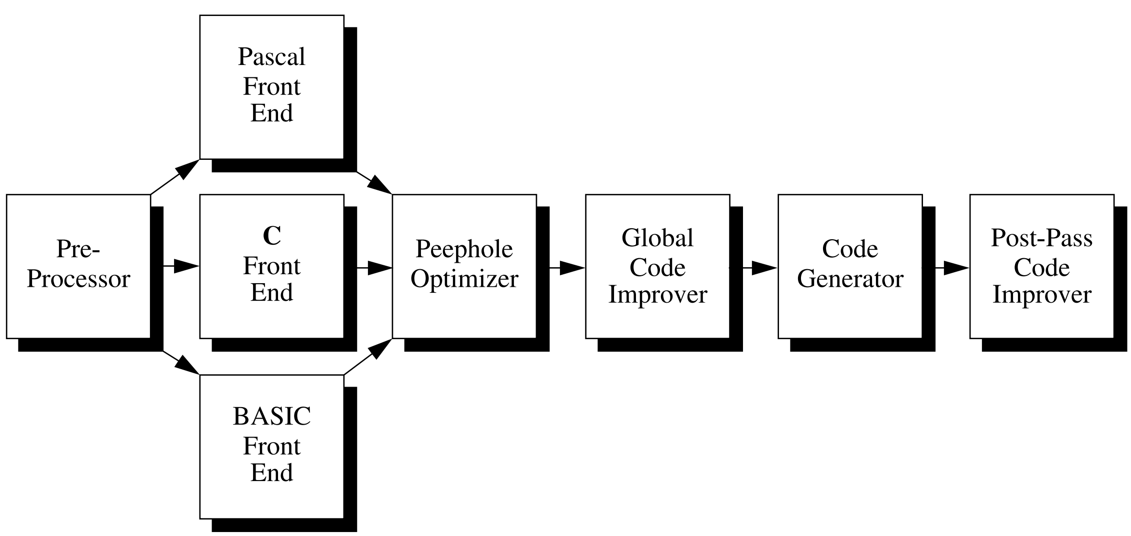


Figure 4: Structure of a compiler produced with the Amsterdam Compiler Kit

only to architectures with individually-addressed, 8-bit bytes. Much of the ACK's retargetability can be attributed to the fact that it applies code improvements almost exclusively on its machine-independent intermediate code. The back-end consists of a machine-independent code generator that uses a target machine description table to convert EM code sequences into assembly language instruction sequences for the target machine.

Although ACK has been used to produce compilers that generate good code for a number of different architectures, it suffers from several deficiencies. Most of these deficiencies are the result of performing code improvements almost exclusively on a high-level intermediate language. For example, to support peephole optimization, EM has been embellished with nearly 100 special-purpose instructions that are little more than abbreviations of primitive instruction sequences. While these special-purpose instructions allow ACK to exploit common machine-specific features like pre-decrement and post-increment addressing modes, there is no guarantee that the improvements made to the EM code will result in higher-quality machine code. In fact, most of the peephole transformations applied to the EM code do not improve the machine code produced on RISC architectures that lack complex addressing modes. Not only are some of the improvements made at the intermediate level useless, but some essential improvements are missed. Consequently, to produce good code, ACK must also perform peephole optimization at the machine level. This strategy complicates the compiler, the intermediate language and the retargeting process. Furthermore, in spite of these measures, ACK-based compilers are unable to produce high-quality RISC code.

2.5.2 PO

Davidson and Fraser developed a technique to performs peephole optimization via instruction selection in a retargetable mechanism [DAVI80]. This a peephole optimizer based on this technique, along with a common subexpression eliminator and a local-scope register allocator, comprise the retargetable PO optimizing compiler [DAVI84b]. The structure of the PO compiler is shown in Figure 5.

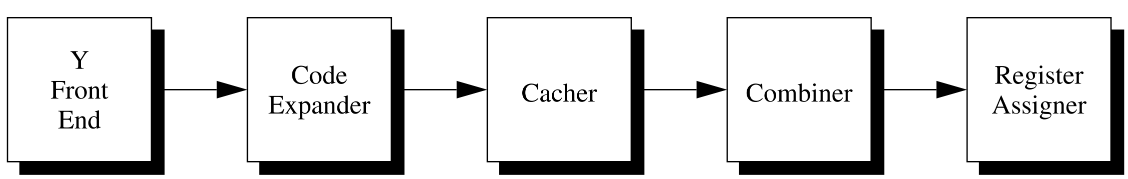


Figure 5: The structure of PO

PO was one of the first optimizing compilers to apply code improvement transformations exclusively on a low-level representation of the code. PO's front-end generates code for a high-level abstract *intersection machine* that provides a minimal set of computational primitives. Using an intersection machine enhances the retargetability of the compiler because it uses only a small set of operations that are likely to be available on most target machines. The *code expander* replaces each abstract machine instruction with a sequence of target machine instructions. Register transfers represent machine code using an architecture-independent notation that allows optimization phases to perform code improvement transformations that are largely independent of the target machine.

PO's *cacher* phase performs local-scope, machine-independent common subexpression elimination. Because all values, including those computed in address calculations, are exposed at the machine-level and assigned to pseudo-register values that are never re-assigned, *cacher* detects and eliminates all local common subexpressions [DAVI84a]. The *combiner* phase performs instruction selection by merging adjacent register transfers into singletons and consulting a machine description to determine if some instruction on the target machine has the effect described by the combination. This process transforms the naive code produced by the code expander into code sequences that fully utilize the instruction set and the addressing modes provided by the target machine. Before converting the resulting register transfers to assembly language code, a local register assigner is used to map pseudo-registers to appropriate target machine registers.

Although its lack of global code improvements prevents PO from producing very high quality object code, the code produced by PO is comparable to that produced by most of the production compilers that were available at the time it was developed. Unlike these contemporaneous systems, however, PO can be targeted to a new architecture is as little as three days. These results suggest that reasonable code quality can be obtained by performing even a few simple code improvements at the machine level, and that it is possible to structure such a system so that it is highly retargetable.

2.5.3 HP Precision Architecture

Shortly after the development of PO, a low-level, global-scope optimizing compiler was developed for the HP Precision Architecture [JOHN86]. One of the most notable features of this compiler is that most code improvement transformations are performed after code generation to fully exploit the processor pipeline, addressing modes and loop control instructions provided by the target architecture. The back-end of the compiler works with several front-ends which support a variety of source languages. Each front-end provides the back-end with aliasing information in the form of a list of pointer and variable references that actually or potentially access the same memory location. The code improvement algorithms use this information to perform transformations that change the definitions and uses of memory locations without disturbing the semantics of the original code. The front-ends are expected to perform improvements like constant-folding and evaluation order determination since these are not part of the back-end's repertoire.

The fixed order in which the code improvement phases are invoked was chosen to reduce the amount of time required to perform transformations and maintain the dataflow information. Code improvements that tend to eliminate instructions or require hard-to-update dataflow information are applied as early as possible in order to reduce the amount of work performed by subsequent transformations. The back-end also provides different levels of optimization to give the user the ability to trade compilation time for code quality and to produce debuggable code.

Measurements of the compiler's effectiveness indicate that the back-end produces code that is on average 37% smaller and 46% faster than the code produced directly by the front-end code generators. This is a substantial accomplishment, considering that the front-ends already produce good code. These results suggest that performing code improvement transformations on low-level code is an effective way to produce high-quality machine code.

2.5.4 VPO

The experience gained from the PO compiler was used to develop *vpo* [BENI89], which differs from its predecessor the following significant areas:

- it uses a **C** [KERN78] front end,
- the instruction recognizer uses a *yacc*-produced parser [JOHN78a],
- it performs global-scope register allocation and code improvements and
- register allocation precedes most code improvements.

Like PO, *vpo* uses a code expander to generate naive object code in the form of register transfers (RTLs) from the abstract intersection machine code emitted by the semantic analysis phase of the front-end. The code expander defers register allocation by using pseudo-registers to hold temporary expression values. All user variables, including those declared with the **register** class, reside in memory. The back-end performs instruction selection before local-scope register assignment, which maps the pseudo-registers used to contain temporary values to actual target-machine registers. The RTLs can be translated to assembly code at any time after the assignment phase, although a number of code improvement phases and a global-scope register allocator are usually invoked before the final translation phase, which converts the register transfers to assembly language. Figure 6 shows the structure of *vpo*.

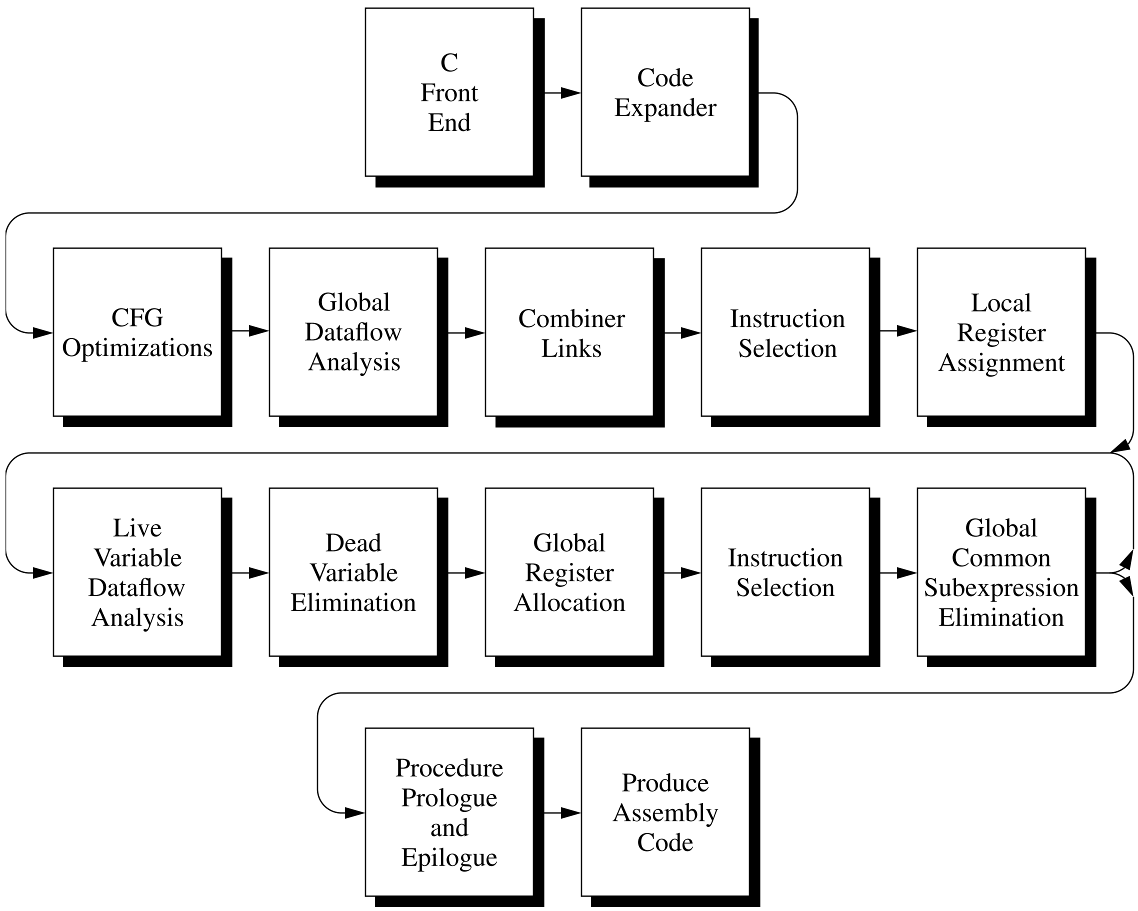


Figure 6: The structure of *vpo*

The first phase of the back-end builds a control-flow graph (CFG) by partitioning the code into *basic blocks* [AHO86] in order to perform branch chain elimination, jump minimization and dead code elimination. Because the instruction selection phase attempts to combine logically adjacent rather that physically adjacent instructions, local and global links are inserted to connect groups of instructions that can potentially merge into a singleton. The initial instruction selection phase uses these links to reduce the length of the naive code sequences produced by the code expander by a factor of 50% or more. This has the additional desired effect of decreasing the amount of work performed by the remaining phases.

Phase iteration is used to reduce the amount of effort required to determine an optimal phase ordering for each target architecture. This strategy ensures that all of the opportunities created by each transformation to perform additional improvements are automatically exploited. Phase iteration is facilitated by the fact that all code improvement phases operate on the same representation of the code [BENI88]. Each phase indicates whether it applied transformations to the code so that *vpo* can determine which phases should be re-invoked.

The global code improvements performed by *vpo* include dead variable elimination, local variable promotion and common subexpression elimination. Local variable promotion uses a very simple algorithm that assigns a single local or argument variable to an unused register for the entire life of a function. Comparisons performed on various CISC architectures show that *vpo* is capable of generating code of equal or somewhat better quality than the production optimizing compilers on these machines. The amount of time required to retarget *vpo* depends on the complexity of the target architecture and can range from two weeks to a month.

2.5.5 GCC

Like *vpo*, the GNU C compiler (*gcc*) [STAL89], performs code improvements exclusively on machine code represented using register transfers. Currently, *gcc* provides front-ends for **C**, **C++** and Objective **C** and its back-end supports 23 different target architectures. Register transfers are generated from the front-end's abstract expression trees using machine-specific templates. Various local and global code improvements and register allocation are applied to the register transfers in a pre-determined order. Some of these code improvement phases may be invoked more than once if the user explicitly requests it.

Because of its performance, retargetability and availability, *gcc* is available on most UNIX systems and, in a few cases, is provided and supported by the manufacturer as the standard compiler. Despite its size[1], *gcc* is relatively easy to retarget. Retargeting the system entails providing the machine-specific templates that produce register transfers from abstract syntax trees, indicate which register transfers denote valid instructions on the target machine and transform register transfers to assembly language code for the target machine. The compiler also requires a substantial number of parameters that control the code improvements

[1] A recent version of *gcc* contained over 432,000 lines of code.

to be appropriately specified and, in some cases, machine-specific routines to be written in order to exploit rare or unique architectural features. While it may only take several weeks to start producing correct code for a new architecture, determining which code improvement parameter values yield the best results and writing the additional improvement routines needed to produce the desired code quality may entail several additional months. This effort includes determining the best phase ordering for the target machine. While already large, *gcc* continues to grow as more target architectures are added to its repertoire. This suggests that the techniques used to provide target machine information and handle unique architectural features in *gcc* may not be as general or as retargetable as they could be.

2.5.6 Marion

Marion is a code generation system for pipelined RISC architectures that require quality instruction scheduling [BRAD91]. Marion integrates instruction scheduling, code generation and register allocation into a single phase. The instruction scheduler performs two separate instruction scheduling passes: the first to obtain schedule cost estimates and a second to produce a final schedule.

The experience with Marion indicates that integrating the scheduler, code generator and register allocator results in a system that produces code on pipelined RISC architectures that executes 12% faster than the code produced using separate components. Although this is an encouraging improvement, Marion does not support global-scope code improvements and its framework does not suggest how these improvements should be integrated. Also, despite the claim that Marion is retargetable, the emphasis on instruction scheduling over instruction selection limits its effectiveness to pipelined RISC architectures.

2.6 Summary

This chapter presented several compiler organization strategies along with evidence suggesting that performing code improvements exclusively at the machine-level is feasible and perhaps even desirable. The concepts introduced in this chapter that are used in the frameworks presented in Chapters 5 and 6 include:

- comprehensive code improvement suites are required to produce high-quality code,
- interactions between code improvement phases are affected by the target architecture,
- the choice of intermediate representation and the methods used to provide target architecture information to the compiler significantly affect the retargetability of the system,
- post-pass code improvements are essential to exploiting architectural nuances, and
- performing code improvements exclusively on a single representation of the code simplifies the task of re-invoking code improvement phases.

CHAPTER 3

REGISTER ALLOCATION

The purpose of register allocation in an optimizing compiler is to apportion a limited set of registers among the various parts of the compiler that need them. Good register allocation is essential to producing high-quality code because the allocator affects the ability of the code generator to produce efficient code and of the code improvements to perform effective transformations. While register allocation has often been considered a code improvement [HENN90], it is presented here not as a code improvement, but as a task whose importance in an optimizing compiler rivals that of code generation and actual code improvements.

3.1 The allocation process

Register allocation is one of the most studied yet least understood compilation tasks. In fact, the term, "register allocation," often means different things to different people. To avoid any misunderstandings regarding what the retargetable register allocation algorithms presented in this dissertation do, this section provides an overview of the allocation process and defines many of the terms used to refer to various aspects of the register allocation process throughout this dissertation.

The job of a register allocator is to determine where each *value* that is computed or used in a program will reside at each stage of the program's execution. A value is a typed quantity that can reside either in a register, in memory, or, when it can be computed by a sequence of instructions, in the code. Because storing a value as a sequence of instructions in a code segment is similar to keeping a value in a memory location in the data segment, code values will be treated like memory values. Values, which come from various sources, may be either *discretionary*, meaning that the allocator can veto their creation because the compiler is able to generate semantically correct code without them, or *mandatory*, which implies that the allocation process must locate a suitable storage location in which to keep them.

Figure 7 shows a diagram of the retargetable register allocation process presented in this dissertation. This process contains a number components that are absent from most other register allocators. Also, the components illustrated here interact in ways that differ from the ways that the components in other register

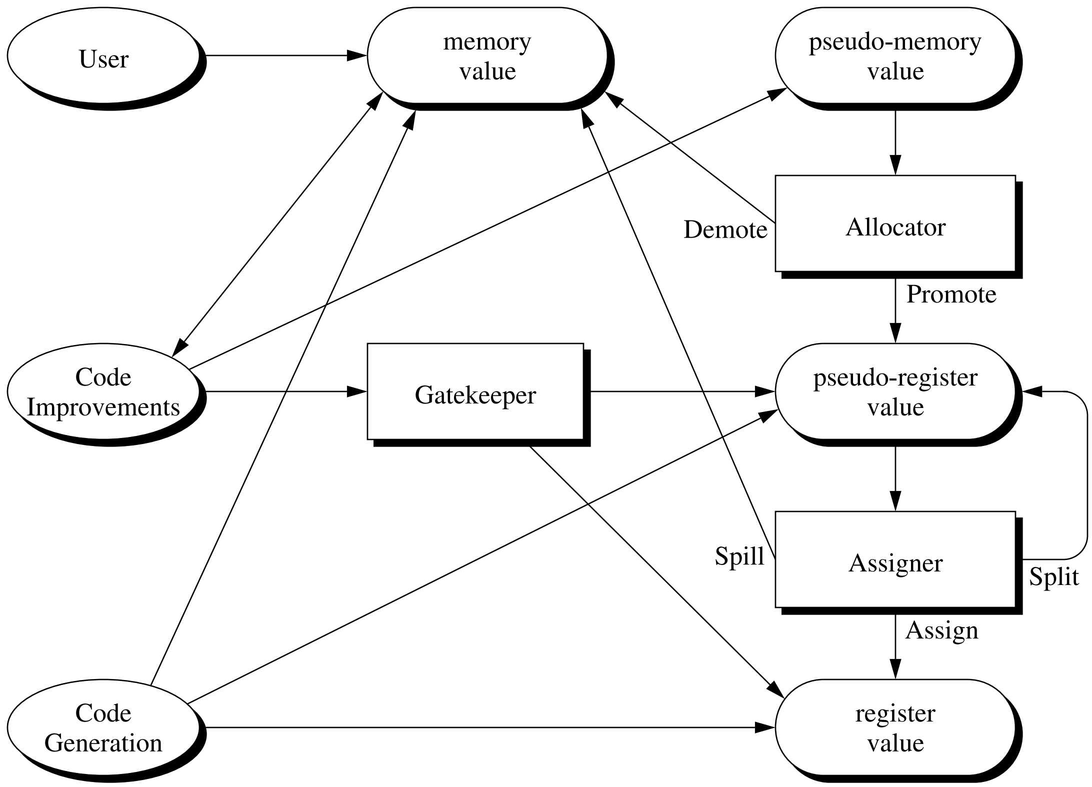


Figure 7: The retargetable allocation process

allocators do. These differences will become evident as the process shown here is compared to extant register allocators later in this chapter.

There are four different types of values: memory, pseudo-memory, pseudo-register and register. Of these, memory and register values can appear in the machine code and the two others exist only inside the compiler. Along the left-hand side of the diagram are the three sources of values: the user, the code improvements and the code generator. Also included in this diagram are the three active allocation agents: the gatekeeper, the allocator and the assigner. The following paragraphs explain how these agents operate.

The values assigned to the scalar and array variables declared by the user in the source code are mandatory. The compiler initially reserves storage space in memory to contain these values and generates intermediate to code accesses these values via load and store instructions if the target architecture is a RISC or operand effective addresses that reference memory if it is a CISC. As a result, these values are effectively assigned to memory.

The code generator creates mandatory memory and register values while forming complex expressions from a simple set of basic operations. The ability to create register values allows the code generator to produce machine code for architectures containing operations that implicitly reference specific registers and implement *calling conventions*[1] that pass function parameters in registers. Register values are bound to specific target registers and can appear only in operand fields that accept register items. This simplifies the task of translating the internal representation into machine code at the end of the compilation process.

The code generator can also create pseudo-register values, which are similar to register values except that they are not bound to specific target registers. Pseudo-registers allow the code generator to indicate that a value should reside in a register without having to determine exactly which target register the value should reside in or even if the target machine has enough registers to contain all of the pseudo-register values that it creates. Pseudo-registers make it possible to separate the register allocator from the code generator so that each of them can be specifically tailored to their respective tasks.

Because pseudo-registers are not bound, they can exist only in the intermediate code, and must be bound to specific target registers before the intermediate code is translated to machine code. This process, known as *assigning* the pseudo-register, is the task of the assigner, which is one of the three active agents in the allocation process. The assigner ensures that no contemporaneous pseudo-register values are assigned to the same register. This task is complicated by the fact that there may not be enough target registers to assign every pseudo-register value. The assigner overcomes this problem either by *spilling* or *splitting*. A pseudo-register value is spilled by transforming it into a memory value. When a pseudo-register value is spilled, the intermediate code is modified to reference the value in a memory location instead of a register. Spilling a value usually increases the amount of time and code required to reference it. For these reasons, good register assigners strive to spill the pseudo-register values that have the least negative impact on the quality of the machine code produced by the compiler.

Splitting entails dividing the life of a pseudo-register value into two or more disjoint partitions and replacing all of the references to the split pseudo-register value in each separate partition with a new pseudo-

---

[1] The calling convention is the set of rules that describe how function arguments and values are exchanged across function boundaries.

register value. Because the life-spans of the new values are shorter than that of the original value, they usually conflict with fewer other values and are more likely to be assigned to target registers. To ensure the continuity of the value across the partitions, *transfer code* is inserted at each partition boundary. Since transfer code takes time to execute, a good assigner will weigh the estimated increase in the execution time against the estimated advantages obtained from splitting the value and ensure that the net impact is beneficial before performing the split.

Code improvement transformations create values in order to reduce memory system traffic and eliminate redundant expression value computations. Unlike values produced by the user and the code generator, values created by code improvements are discretionary because the compiler can choose to forego the transformations that create them. To avoid over-optimization, the allocation agent known as the gatekeeper can cancel any transformation that attempts to use an unavailable register or create a pseudo-register value that cannot be assigned to a target machine register. Should the gatekeeper veto a transformation, the code improvement algorithm can perform it using a memory value instead of a pseudo-register value if the estimated benefits still outweigh the costs.

Memory values are not necessarily destined to stay in memory. For example, memory values that are not indirectly referenced could safely reside in a register. Such memory values are transformed into pseudo-memory values by the local variable promoter. Like memory values, pseudo-memory values are associated with a specific location on the target architecture's address space and are referenced by the intermediate code via load and store instructions. Unlike memory values, pseudo-memory values are either *promoted* by the allocator to a pseudo-register value, or *demoted* back to memory values. To prevent over-optimization, the allocator, like the gatekeeper, avoids promoting any pseudo-memory value that cannot be assigned to target register. A good allocator promotes the memory values that most improve the quality of the code and demotes only the values that would least benefit the code.

3.2 Retargetable register allocation

The need to perform effective register allocation in an optimizing compiler is not diminished by retargetability. Retargetability issues, however, complicate the allocation process and need to be addressed in the design of a retargetable compiler. The following sections describe how the size of the register set provided by the target architecture, which is the architectural component that most affects the allocation strategy and perhaps even the overall compilation strategy, impacts the retargetable optimizing compiler.

3.2.1 Very few registers

On architectures that provide only one, two or three registers, the register requirements of code generation are paramount. Any registers not required for code generation should be used to perform the transformations that will produce the greatest benefit. Since registers are extremely scarce, over-optimization is an acute problem.

An effective register allocation strategy for this class of architectures mandates a conservative gatekeeper to prevent over-optimization. Register releasing code improvements, especially those that perform transformations that reduce the number of registers required to calculate expressions and generate code, are essential. Only the most beneficial code improvement transformations will be able to create new pseudo-register and register values. Code generation should be performed as early as possible to ensure that it obtains the registers it needs and to determine how many registers remain for code improvements. Following this, only machine-level code improvement can be applied. Without the registers needed to perform substantial global-scope transformations, local-scope register allocation algorithms that produce near-optimal block-level allocation are usually sufficient and perhaps even desirable.

3.2.2 Few registers

On architectures providing between four and ten registers, there will always be some registers available to perform optimizations. Registers should be reused as much as possible to allow code generation and register-consuming code improvement transformations to share registers. With such few allocable registers, there will be a tendency to overcommit them—especially when the code provides many opportunities for improvement.

An effective register allocation strategy for these architectures requires a gatekeeper to prevent over-optimization in codes that contain iterative control-flow structures. Although a substantial number of code improvements can be supported, transformations that do not consume registers or reduce the number of registers used by the code are essential. An allocator that prioritizes promotions based on accurate cost and benefit estimates, followed by an assigner that aggressively reuses registers, should produce adequate results.

A global-scope register allocator or a local-global hybrid that produces little inter-block transfer code will be needed to exploit the ample opportunities to perform global transformations.

3.2.3 Many registers

On architectures containing between eleven and sixty-four registers, the primary challenge is to find suitable uses for the registers. The order in which to perform code improvements, also known as the *phase ordering*, will become a serious concern as more code improvements are added to the compiler in an attempt to fully employ the available registers. The register set is more likely to be underutilized than overcommitted on these architectures.

An effective register allocation strategy for these architectures emphasizes global-scope code improvements. Since over-optimization is not an overriding concern, the gatekeeper can be liberal in order to encourage register consuming transformations. The allocator will be able promote most pseudo-variables and, rather than trying to limit the creation of pseudo-registers, the best results will often come from slightly over-committing the register set and relying on the additional opportunities that this situation presents to register releasing transformations, and the ability of the assigner to reuse registers, to prevent the creation of spill code. Global-scope register allocation is required and inter-procedural register allocation preferred. Since inter-procedural register allocation is expensive, a more viable compromise is to use global register allocation and to minimize the overhead costs of the calling convention by using registers to pass values between functions.

3.3 Basic allocation strategies

There are at least as many different allocation strategies as there are compilers. Many of these strategies, however, are variations on a theme, and much can be learned about them by examining the few truly unique mechanisms that they use. A certain level of familiarity with these mechanisms is also essential to understanding the strengths and weaknesses of existing allocation strategies and of the strategies that were developed during the course of this research. For these reasons, the following sections introduce some of the most common local and global register allocation strategies.

3.3.1 Local-scope register allocation

The three most desirable attributes of good local-scope register allocation algorithms are simplicity, speed, and optimality within a basic block. Local-scope algorithms are rarely used alone in optimizing compilers, but they are often a part of hybrid register allocators that incorporate the advantages of local algorithms with the inter-block register allocation provided by global algorithms. The following sections survey the most commonly used local register allocation techniques.

3.3.1.1 Minimal cost path

The register allocation strategy developed for the pioneering FORTRAN I compiler [HORW66] is based on finding the minimal cost path in a directed acyclic graph (DAG). In this scheme, graph nodes represent the configuration of the values residing in the registers and the edges are annotated by cost values corresponding to the number of loads and stores required to transition between the configurations presented by each pair of connected nodes. The register allocator extends only the paths that have the potential to be the minimal cost path to avoid producing prohibitively large graphs.

The example shown in Figure 8 is based on an architecture with two allocable registers. A sample sequence of pseudo-variable references appears to the left of the graph. The entry node, which represents the register configuration at the start of the basic block, has useless values (x_5 and x_6) in both allocable registers. For each successive pseudo-variable reference, the graph shows the set of minimal change configurations that can be reached from the previous state. Register values that differ from their corresponding pseudo-memory value are annotated with an asterisk and must be put back in memory before the basic block ends. The cost of each edge is the number of instructions required to store back displaced annotated values and load the referenced pseudo-memory value into a register. Only transitions that displace no more than one value are considered. The total cost of the cheapest path from the entry node is given by the value immediately to the left of each node. Any path starting at the entry node and ending at any one of the least-cumulative-cost leaf nodes represents an optimal allocation. One such path has been highlighted.

As with all other optimal local algorithms, there is no obvious way to extend the minimal cost path graph to perform optimal global-scope register allocation. The minimal cost path technique is among the most complex and computationally intensive local register allocation algorithms used in production compilers. The following sections describe simpler optimal and near-optimal algorithms.

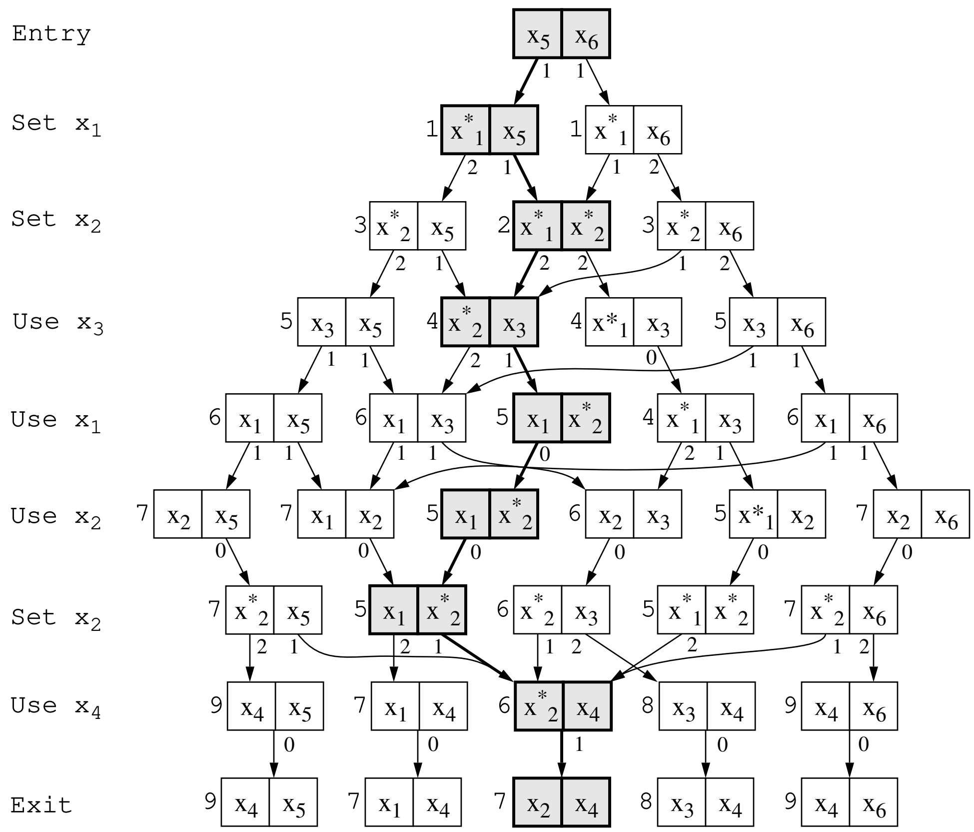


Figure 8: A minimal cost path graph

3.3.1.2 Page replacement

Belady's optimal page replacement algorithm, developed for operating systems, can also be used to perform optimal local register allocation [BELA66]. The strategy is simple: when a value must be loaded into a register and all of the registers contain live values, displace the register value with the most distant next use. This approach works because it displaces the value that is most likely to end up being displaced, anyway.

The page replacement algorithm is locally optimal and easier to understand than the minimal path cost algorithm. The drawback of using this strategy to perform register allocation is that it requires the ability to look ahead. While this can be easily accomplished in a multi-pass compiler, one-pass systems cannot do this and frequently resort to less accurate heuristics like displacing the least-recently-used or the least-recently-loaded register.

3.3.1.3 Use counts

Freiburghouse developed a system for performing local register allocation based on keeping track of the number of times each pseudo-register will be referenced in the future [FREI74]. This algorithm requires the first occurrence of each pseudo-register value to indicate how many times the value will be used. This count serves two purposes. First, by decrementing it after each use, the register allocator can automatically determine when the life of a pseudo-register ends and immediately release the register assigned to it. Second, when the register allocator is forced to displace a value, it selects the one with the smallest use count on the assumption that references to it are less likely to appear in the immediate future than references to values with higher use counts are. This strategy has been found to outperform algorithms that displace least-recently-used and least-recently-loaded values.

3.3.2 Global-scope register allocation

Global-scope register allocation is essential on systems that perform transformations that create new inter-block values. Unlike local-scope register allocators, global-scope register allocators must consider the control-flow structure of the function. The following sections discuss some of the global-scope allocation strategies that have been used in compilers.

3.3.2.1 Packing

Packing algorithms operate by filling a container that can hold objects. The dimensions of the container is proportional to the number allocable registers available and the number of instructions in the function being allocated. Register allocation entails packing this container with objects representing pseudo-register life-spans. A sample illustration of this process is shown in Figure 9.

The packing container for an arbitrary function on an architecture that provides four allocable registers is shown in Figure 9(a) and the objects representing the pseudo-register life-spans in Figure 9(b). Packing consists of placing these objects into the register slots of the container without changing their original vertical positions or overlapping them. One possible packing is shown in Figure 9(c).

There are many viable approaches to performing the packing process. One possible strategy is to place a single pseudo-register value in each register slot until either all of the slots have a pseudo-register in them or all of the pseudo-registers are packed. This approach is simple because it allows the allocator to use

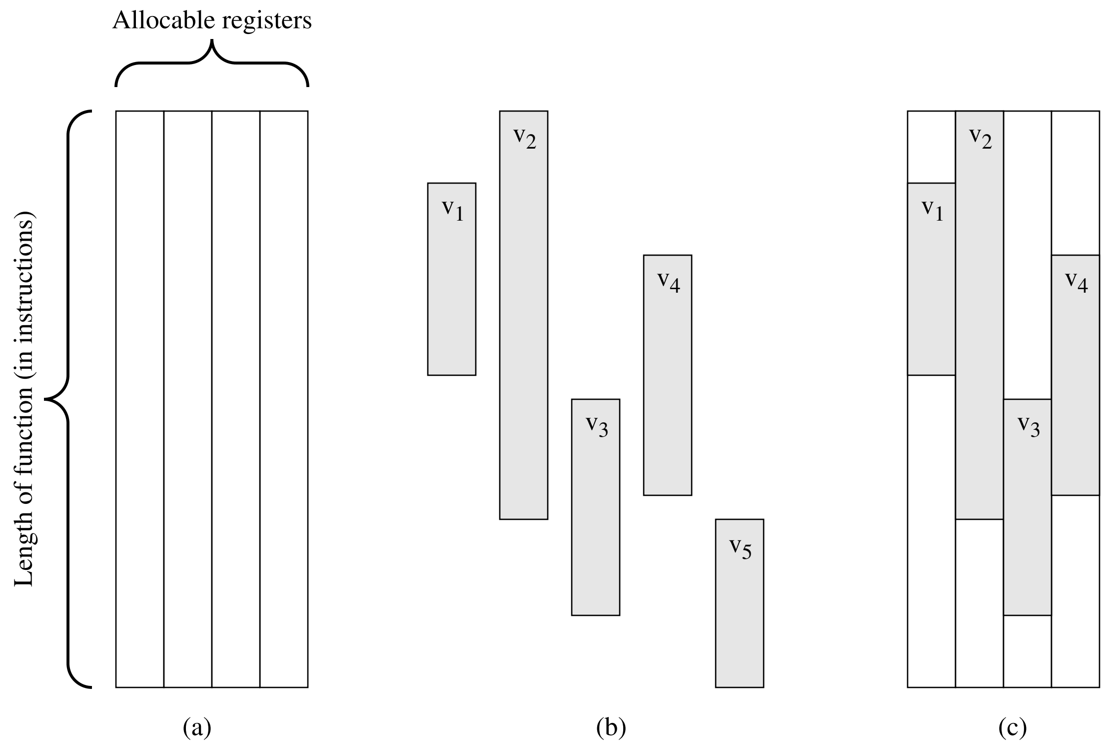


Figure 9: Register allocation via packing

a one-dimensional representation of the packing container. The disadvantage of this packing technique, however, is that it does not exploit opportunities to reuse registers by placing more than one pseudo-register in a slot. For example, pseudo-register value v_5 in Figure 9(b) can share the slot occupied by v_1 in Figure 9(c).

Sophisticated packing algorithms attempt to maximize the number of pseudo-register values packed into the container by exploiting opportunities to place multiple objects into a single slot. This approach has two advantages. First, every packed pseudo-register improves the quality of the code that references the value that it holds. Second, because each slot corresponding to a *non-volatile* register[1] incurs a use penalty by adding save and restore instructions to the prologue and epilogue of the function that maintain the register's external value, the packing algorithm can realize substantial savings by merely minimizing the number of register slots used. Because finding an optimal packing is an NP-complete problem [BAAS78], algorithms that can optimally pack a non-trivial set of pseudo-register values require unacceptable amounts of execution time. For this reason, compilers that use sophisticated packing allocators employ heuristics to obtain a good packing instead of an optimal one.

[1] Registers whose values potentially change across an external function call are called volatile registers. These registers are specified by the calling convention.

3.3.2.2 Coloring

Register assignment can be reduced to a graph coloring problem [CHAI81]. To perform register assignment via coloring, the assigner builds an *interference graph* where the nodes represent pseudo-register values and the edges connect values whose lifetimes overlap. An interference graph is shown in Figure 10.

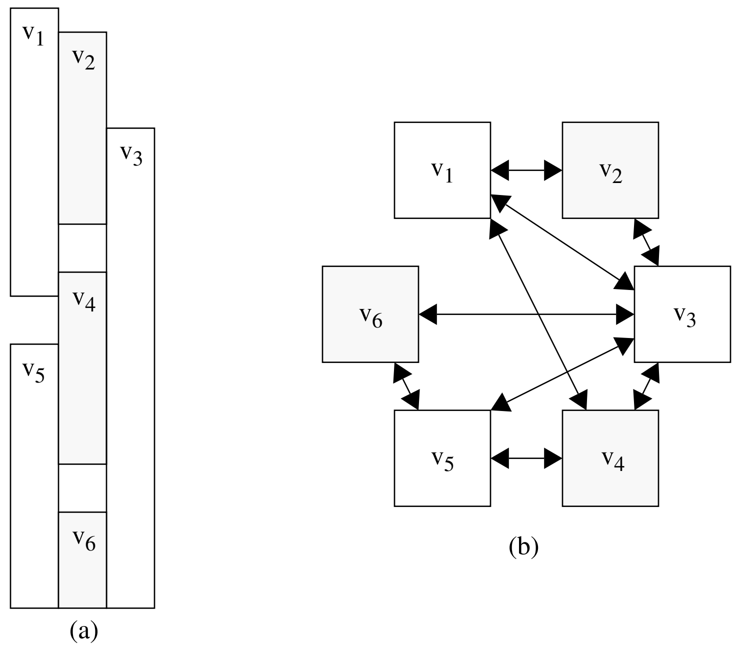


Figure 10: Interference graph

Figure 10(a) shows the packing of five pseudo-register values in a function and the corresponding interference graph for this packing is given in Figure 10(b). The interference graph yields the same information regarding the relative life-spans of the pseudo-register values that the packing container does. For example, the interference graph indicates that the life of pseudo-register value v_4 overlaps those of values v_1, v_3 and v_5 because the interference graph node representing v_4 has arcs connecting it to the nodes for v_1, v_3 and v_5. The interference graph makes the register assignment problem a graph coloring problem in the following sense: if node v_4 is colored with (or assigned to) a target register, then nodes v_1, v_3 and v_5, which are directly connected to v_4, cannot be colored with (or assigned to) the same target register.

Determining the coloring (or assignment) of an interference graph that requires the fewest number of registers is an NP-complete problem [BAAS78]. Like compilers that perform sophisticated register packing, those that use coloring register allocators employ heuristics to obtain a good coloring of the interference graph instead of an optimal one. Additionally complicating the coloring process is the need to handle interference graphs that require more registers to color than there are available. Compilers handle this problem by using a spill mechanism to remove nodes from the graph by demoting pseudo-register values until a coloring is obtained. Because this process can create spill code, it is essential to provide the allocator with a suitable strategy for coloring and demoting interference graph nodes. The popularity of coloring paradigms has produced a plethora of coloring heuristics and spill strategies [CHAI81, CHOW83, BRIG89, GUPT89, CALL91].

One useful aspect of register coloring is that architecture-specific register requirements can be incorporated into the interference graph. For example, if a pseudo-register value is live across an external function call, it can be connected to nodes representing the volatile registers. The coloring process will then automatically avoid assigning the pseudo-register to any of the volatile registers.

3.3.2.3 Probabilistic

Probabilistic register allocation, developed by Proebsting and Fischer [PROE92], is a hybrid strategy that performs both local and global scope allocation. The local allocation phase computes a set of Δ-probabilities for all of the pseudo-variable uses in a basic block. The global allocation phase uses these Δ-probabilities to direct the allocation process. The Δ-probability determination process is shown in Figure 11.

Inst #	Instruction	Live registers	Pseudo-variables					Comments
			v_1	v_2	v_3	v_4	v_5	
1	$r[1]=M[v_1]$	1	1^*	2/3	2/3	2/3	2/3	
2	$r[2]=M[v_2]$	2	$\underline{1}$	1^*	1/2	1/2	1/2	
3	$r[3]=r[1]+r[2]$	1	0	$\underline{1}$	1	1	1	v_1 and v_2 are dead
4	$r[4]=M[v_3]$	2	-	0	1^*	1	1	
5	$r[5]=M[v_4]$	3	-	-	$\underline{1}$	1	0	all registers in use
6	$r[6]=r[4]+r[5]$	2	-	-	1/2	1/2	-	reuse either r[4] or r[5]
7	$r[7]=r[3]+r[6]$	1	1^*	-	1	1	-	reuse r[3]
8	$M[v_1]=r[7]$	1	$\underline{1}$	-	1	1	-	
9	$r[8]=M[v_5]$	2	$\underline{1}$	-	1	1	1^*	reuse r[6]
10	$r[9]=r[7]+r[8]$	1	2/3	1^*	2/3	2/3	2/3	
11	$M[v2]=r[9]$	0	1	$\underline{1}$	1	1	1	
Register exit probability			2/3	1	1/3	1/3	2/3	

Figure 11: Δ-probabilities

The example in Figure 11 shows the Δ-probabilities of five pseudo-variables on a target architecture with three registers. A Δ-probability is calculated at every instruction for each live pseudo-variable. In the first instruction, pseudo-variable v_1 is explicitly loaded and the 1^* indicates that the pseudo-variable is guaranteed to be in a register. Since this displaces one of the three current register values, there is a 2/3 probability that each of the other pseudo-variables still reside in a register after the instruction executes. In the second instruction, the $\underline{1}$ value computed for v_1 indicates that it is purposefully retained in a register by the instruction. Since the second instruction also displaces one of the two remaining registers, the other pseudo-variables have a 1/2 chance of remaining in a register. At the third instruction, the register that held v_1 is used to receive the value of the operation, so the probability that v_1 is in a register after this instruction is zero. This process continues until the entire basic block is processed.

The probability that a pseudo-variable value resides in a register at the end of the basic block is shown in the bottom row of the table in Figure 11. These probabilities are determined by locating the last point where a pseudo-variable is loaded into a register and multiplying the Δ-probabilities from that point to the end of the basic block. Thus, at the end of the sample basic block, there is only a one-in-three chance that the value of v_3 is in a register while the value of v_2 is guaranteed to be in one of the registers.

Δ-probabilities are used to direct the global register allocation process. The global allocator operates on the basic blocks in the inner loops of the function first and proceeds outwards to the blocks that execute less frequently. Pseudo-variables with a high probability of residing in registers at the end of a basic block are given priority to the registers at the block boundaries. As these pseudo-variables are assigned to registers, loads and stores are removed and the Δ-probabilities are recomputed to reflect the new register state.

The probabilistic register allocation strategy ensures that the registers are used to hold the inter-block values that least impact the decisions made by the local allocator. This contrasts with strategies that force local register allocation to work around the decisions made at the global scope. The results obtained by Proebsting and Fischer show that this strategy degrades much more gracefully on small register sets than coloring allocators do. Unfortunately, the costs of calculating and updating Δ-probabilities as each new global allocation decisions is made are high.

3.4 Related work

This section presents compilers that have taken unique approaches to performing register allocation or pioneered the use of significant register allocation strategies. All of these compilers have influenced the research described in this dissertation. While not all of the allocation techniques discussed here were specifically designed to be retargetable, they all provide valuable insight into the register allocation problem.

3.4.1 SL/1

The SL/1 compiler was developed for the 256 general-purpose register CDC STAR-100 architecture and features a global-scope register allocation algorithm that uses cost values based on the estimated benefits of assigning values to registers as the basis for making register allocation decisions [DUNL78]. Costs are determined by estimating the difference in machine cycles between executing the program with a value assigned to a memory variable and with the value in a register. All values with a positive cost are candidates for register allocation. By sorting values according to their cost, items that yield the most benefits are considered by the packing allocator first and given priority to the registers. Both local variables, existing only within a single function, and global variables available throughout the program are considered for register allocation. The allocator assigns available registers to the pseudo-variables with the highest positive costs until all allocable registers or values with positive costs are exhausted.

Using the estimated benefit of placing a value in a register to direct the register allocation process makes the compiler responsive to the requirements of the target architecture. This mimics the allocation decisions made by assembly language programmers. Accurate cost analysis, however, is hampered by several factors. First, since costs are affected by the number of times that a value is referenced, the compiler must try to predict the dynamic behavior of the program from its static representation. Also, differences between the high-level intermediate representation and the object code obscure the impact of register allocation decisions on the target machine. Cost estimates based on the effect of register allocation decisions on a high-level intermediate representation lack the reponsiveness to the target architecture that is essential to fully exploiting the capabilities of this approach to register allocation.

Dunlap and Knight pointed out a number of problems that arose from their experience with register allocation. First, it is difficult to determine costs for values that can either be loaded from memory or recomputed. For example, a constant address expression may be used often enough to warrant keeping it in a register but when this value is displaced from a register it is cheaper to compute it from scratch than to save it to memory and reload it at a later point. Second, since register allocation decisions affect the number of registers needed to generate code, it is difficult to determine exactly how many registers will be used by the code generator and how many remain to hold other values. Their solution is to reserve a fixed number of registers for code generation, which means that some of the registers reserved for the worst-case code generation scenario remain unused even when the register allocator could have assigned values to them. Finally, they noted that interactions between register allocation, code generation and instruction scheduling complicate the process of finding optimal instruction schedules.

3.4.2 U-code

Sites developed a retargetable compiler incorporating a register and storage allocator that uses an abstract definition of the storage hierarchy on the target architecture [SITE79]. This allocator works on a high-level intermediate representation called U-code [MOTO85]. Each pseudo-variable in the intermediate representation is annotated with information describing its type, size and storage requirements. For each pseudo-variable, the storage allocator calculates a priority proportional to the number of references and inversely proportional to the length of it life. The storage allocator packs items starting at the top level of the machine hierarchy (registers) and continuing on to the lower levels of the hierarchy (memory). The packing process favors volatile registers that can be used without incurring save and restore costs as well as memory locations that can be accessed with cheap addressing modes. Pseudo-variables are not split and reside entirely within a single level of the hierarchy.

The U-code storage allocator is invoked prior to code generation and allocates a predetermined subset of the allocable registers. The remaining allocable registers are reserved in order to ensure that even the worst-case code generation register requirements can be satisfied. The disadvantage of this strategy is that the entire register set might have to be reserved for code generation on target architectures that provide very few registers. Hence, partitioning limits the retargetability of the storage allocator.

The life-spans of and the number of references to the set of pseudo-variable values shown in Figure 12(a) are used to illustrate the advantage of assigning a priority to each pseudo-variable before packing. On a packing container with a single register slot, assigning v_7, as shown in Figure 12(b), eliminates ten memory references and commits the remaining pseudo-variables to memory. This would be the preferred choice for a simple packing allocator, but not for a sophisticated packing algorithm that can assign multiple pseudo-variables to the same register. For example, selecting v_1, v_2 and v_3, as shown in Figure 12(c), would eliminate a total of eleven memory references. Because of this, Sites uses a heuristic to assign a priority to each pseudo-

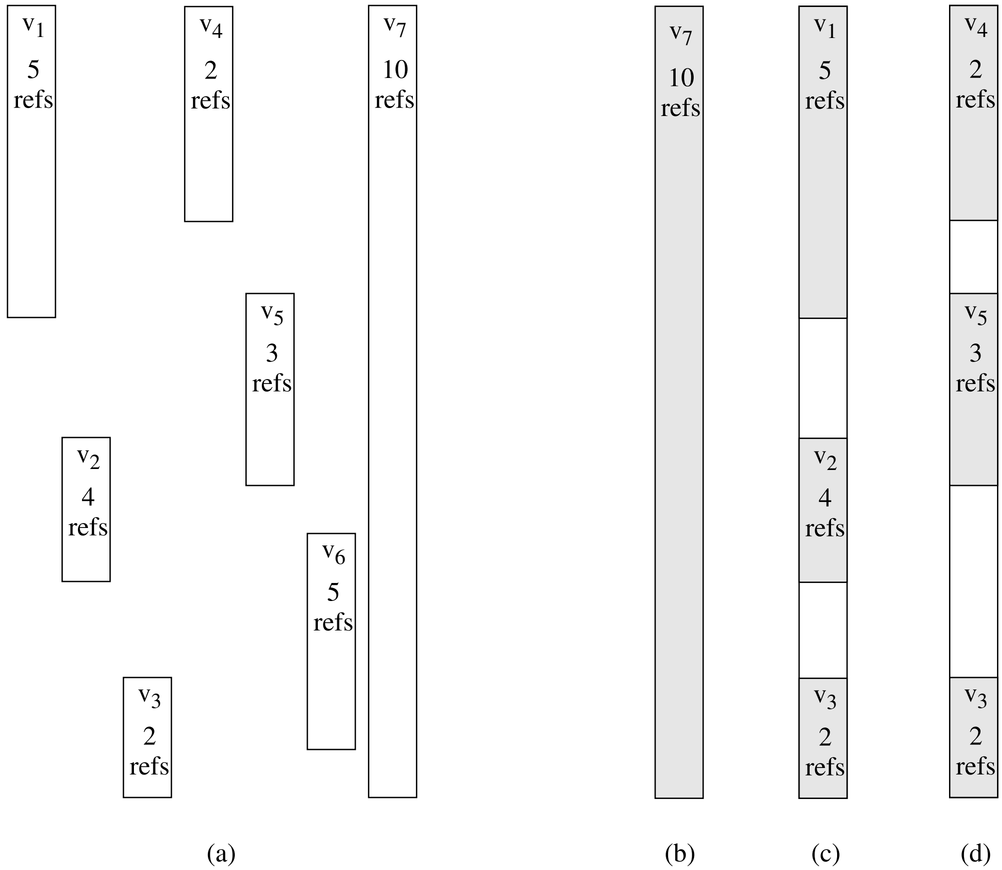


Figure 12: Priority-driven register packing

variable and then packs them based on their priority. The heuristic considers the number of references to each value, so that items with more references are given a higher priority. In addition, because long-lived pseudo-variables tend to overlap many competing pseudo-variables, the priority of a pseudo-variable is decreased proportionally to the length of its life-spans. While this technique might reduce the priority of v_7 enough to allow the optimal assignment to be chosen, it could also backfire and select v_3, v_4 and v_5. As with most heuristics, there is no guarantee against pathological cases.

An advantage of the U-code storage allocator over most coloring strategies is that the packing algorithm favors the least costly registers and memory locations. When assigning registers, it is desirable to use the volatile registers whenever possible, since these registers do not have to be saved in the procedure prologue and restored in the epilogue. Likewise, many target architectures offer a small set of memory locations that can be accessed using cheap addressing modes or fewer instructions.

Because it operates on a high-level intermediate representation, Site's storage allocator is hampered by its inability to obtain accurate cost and benefit estimates to guide its allocation decisions. For example, since the target machine may require multiple machine instructions to implement a high-level operation, a single pseudo-variable reference in U-code may translate to two or more memory references at the machine-level. Since packing priorities are based on U-code references, these discrepancies may cause the storage allocator to make poor decisions.

3.4.3 PQCC

Unlike the U-code storage allocator, the retargetable register allocator developed by Bruce Leverett for the Production-Quality Compiler-Compiler project [LEVE80] operates on a low-level representation and does not require the register set to be partitioned between the optimizer and the code generator [LEVE81]. Leverett's allocator integrates the task of assigning registers to the temporary values created by the code generator and the longer-lived values created by the user and the code improvement transformations. The register allocator attempts to assign registers so that the costs of accessing values, moving data, saving and restoring volatile registers and executing instructions are minimized.

The interaction between code generation and register allocation introduces a phase ordering problem that affects all optimizing compilers. Register allocation decisions determine which instructions the code generator selects. In turn, the instructions selected by the code generator influence the number of registers needed to hold temporary values and place restrictions on where some values must reside. To break this interdependence, the PQCC register allocator performs code selection twice. The first code selection pass makes preliminary decisions about which instructions to generate and creates short-lived register values, called temporary names (TNs) to satisfy code generation requirements. TNs are then packed along with the longer-lived TNs that represent the pseudo-variable values created by the code improvement transformations. The code generator selects instructions based on the results of the packing process and handles any of the discrepancies between the preliminary allocation decisions made during initial code selection pass and the final register allocation and code generation decisions.

Much of the power of this framework comes from the ability to perform register allocation decisions and instruction selection on a low-level representation whose architecture-independent structure is capable of reflecting target machine features. This representation provides more accurate cost and benefit estimates than high-level representation do and result in better register packing decisions. Leverett's work suggests that low-level representations can be used to make highly machine-specific tasks like register allocation retargetable without sacrificing their quality.

3.4.4 PL.8

The PL.8 compiler [AUSL82] pioneered the use of graph coloring register allocation. This compiler rigorously adheres to the strategy of dividing the compilation process into a series of simpler, independent problems. To support this strategy, register consuming transformations request pseudo-register values from a gatekeeper that always allows their creation. Consequently, all register allocation decisions are deferred to the assigner. An advantage of using this approach is that phase interactions caused by resource scarcity are eliminated because the allocator uses an unlimited set of pseudo-registers. As a result, register consuming transformations compete for registers on an equal basis. Transformations that initially consume registers but trigger other register-releasing transformations so that there is no net increase in the number of registers used are performed even when the demand for allocable registers temporarily exceeds the supply. The primary disadvantage of this approach, however, is that it often produces interference graphs that require more allocable registers to color than the target machine provides.

In order to perform register allocation via coloring in a reasonable amount of time, the PL.8 compiler uses a heuristic which does not attempt to find an optimal coloring. The assigner considers each node in the interference graph and counts the number of arcs connected to the node, which is the *degree* of the node. Nodes with degree equal to or smaller than number of allocable registers available are removed from the graph because, regardless of which registers are assigned to the interfering nodes, there will always be at least one register available to color them. When the interference graph is reduced to a set of nodes whose degree is greater than the number of allocable registers, the assigner randomly demotes one of them to memory. This process is repeated until the degree the remaining nodes is no greater than the number of allocable registers.

Auslander and Martin report that 95% of the functions compiled on a 32-register machine generate no spill code. On a sixteen register machine, this value drops to 50%. They also mention that:

> "If a target machine had many fewer than sixteen registers we suspect our compiler would over-optimize and produce unacceptable amounts of spill code."

Partly to blame for the lackluster performance of the PL.8 compiler on architectures with small register sets is its simplistic spill algorithm. Since the development of the PL.8 compiler, much effort has been devoted to improving spill mechanisms. Some of the latest advances have been reported by Gupta, Soffa and Steele [GUPT89], by Briggs et al. [BRIG89] and by Callahan and Koblenz [CALL91]. Neither improved spill mechanisms or coloring heuristics, however, can fully compensate for the over-optimization which prevents PL.8 style register allocation from producing high-quality code on machines with few registers.

3.4.5 UOPT

UOPT is a U-code optimizer developed by Chow which uses a priority-based coloring register assigner [CHOW83]. Like Site's storage allocator, UOPT's register allocator is invoked just prior to code generation. Unlike the PL.8 compiler, UOPT's code improvement phases create pseudo-variables instead or pseudo-registers so that code can be generated even if the allocator is bypassed. The assigner is never forced to spill variables because the allocator promotes only as many pseudo-variables can be assigned. Pseudo-variables promotions are ordered according to their priority. A variable's priority is determined by the sum of the references to it weighed by the estimated execution count of the basic block in which a reference takes place. The allocation process first employs a local-scope algorithm which makes promotion decisions based on priority and the assumption that pseudo-variables live at the entry point of the block will have to be loaded into a register and those live on exit and modified within the block will have to be saved back to memory. The process is completed by a global-scope assigner whose job it is to assign the pseudo-register items created by the local allocator in a way that eliminates most of the loads and stores that it produced.

Since UOPT's register allocator performs most of its work at the local level, it runs in linear time with respect to the number of basic blocks. UOPT also improves on the PL.8 coloring paradigm by limiting the creation of pseudo-registers so that the assigner is never forced to spill. Like Site's system, however, UOPT suffers from the need to reserve registers for code generation and from the significant differences between the high-level U-code and the target machine code.

3.4.6 Split allocation

McKusick developed a split register allocation approach where global-scope allocation is performed by the code improvement phases of the compiler and local-scope allocation is handled by the code generator

[MCKU84]. This allocator is used in a Graham-Glanville-based [GRAH82] optimizing compiler. McKusick's framework separates the register set into a dedicated partition and a temporary partition. The dedicated partition is managed exclusively by the global-scope allocator. These registers are used to hold values created by the user and the code improvement transformations. The global-scope allocator also generates the load and store instructions required to split pseudo-variables. The temporary register partition is used to evaluate expressions and generate code and is managed exclusively by the code generator.

According to McKusick, the advantages of splitting the register allocation task is that it allows global-scope register allocation to use the aliasing, data-flow and control-flow information provided by the semantic analysis phase of the compiler. The only architecture-specific information required by the global register allocator is the number and types of dedicated registers. The local register allocator is integrated with the code generator where information about the target architecture and the instruction set is readily available.

McKusick implemented two different variants of his approach. One uses a single-pass, local-scope algorithm to allocate temporary registers and the other performs multiple-pass coloring allocation to more fully utilize the registers in the temporary partition. The latter, more sophisticated approach alleviates register partitioning inefficiencies because it allows the code generator to use any excess temporary registers to hold some of the long-lived values created by the code improvement phases. This yields a 1% to 2% improvement on languages with uncontrolled pointers and a 10% improvement on languages with tightly-controlled pointers. His experience suggests that a modest improvement in the quality of the code produced by the compiler can be expected from an allocation strategy that allows code generation and code improvement transformations to share the allocable registers and from performing alias analysis.

3.5 Summary

This chapter has introduced the allocation process and described its components. It has also provided an overview of the basic allocation strategies and described how some of these strategies have been used in working compilers. In addition to providing the background information needed to examine the allocation process in detail, this chapter has introduced many of the basic concepts that have influenced the design and implementation of the new register allocation strategies presented in this dissertation. These concepts include:

- hybrid register allocators exploit the speed and optimality provided by local-scope register allocation algorithms without producing excessive transfer code to maintain inter-block values,

- accurate cost and benefit estimates are essential to ensuring that register are allocated to the values needed by the transformations that have the greatest potential for improving the code,
- items within the same storage hierarchy should be ordered to account for the differences in the usage costs caused by the calling convention or by architecture specific factors,
- register-consuming transformations should compete equally for the allocable registers and take advantage of transformations that release registers to maximize register utilization and
- the code generator and the code improvement transformations should be able to use registers from a single allocable register pool.

A number of new retargetable register allocation algorithms that incorporate these concepts will be described in Chapters 5 and 6. These chapters will also present evidence supporting the efficacy of these algorithms.

CHAPTER 4

REGISTER DEPRIVATION MEASUREMENTS

This chapter describes the register deprivation measurement techniques developed to gauge the efficacy of register allocation and code improvement strategies across a broad spectrum of register set sizes. The register deprivation experiments presented in this dissertation measure the impact that different register set sizes have on the register allocation and code improvement transformations used in the low-level experimental framework, or *llef*, which is introduced in the following chapter. In addition, this measurement technique was found to have uses beyond its original intent. The following sections introduce the register deprivation mechanism, describe some of its applications and show how the technique can be integrated into a language translation system.

4.1 Trials and probes

A register deprivation *trial* consists of a sequence of *probes*. A probe is a collection of data obtained by compiling and executing a suite of benchmark programs. Each successive probe is identical to the previous one in every respect except that the compiler is prohibited from using one or more of the registers available to the preceding probes.

The data collected during a trial can be any combination of measurements that give an indication of the quality of the code generated for each program in the benchmark suite. Potential measurements include wall-clock execution times, number of instructions and memory references executed, cache performance measurements, hardware monitor timing information and page replacement traces. The register deprivation results presented in this dissertation measure the number of instructions, the number of memory references and the amount of processor time required to execute a set of benchmark programs.

The number of probes that make up a complete register deprivation trial depends on the target architecture. In the first probe, the complete set of allocable registers is available to the compiler, while in the final probe, only the absolute minimum set of registers needed to successfully compile and execute the benchmark suite is used. The total number of probes is the difference between the number of allocable registers available to these two probes. Register deprivation experiments that vary only a subset of the registers and *register augmentation* measurements, which simulate the effects of having more registers than are actually available on the target machine, are also possible and will be mentioned later.

Figure 13 shows the instruction execution counts obtained from a sample register deprivation trial consisting of six probes taken on a mythical machine with a three program benchmark suite. These results

Benchmark	Number of allocable registers in the probe					
	4	5	6	7	8	9
prog1	158,507,316	135,235,161	114,342,566	103,482,410	99,286,348	98,637,175
prog2	97,258,149	88,617,097	80,234,234	77,982,381	76,321,685	75,795,145
prog3	63,196,050	56,069,413	54,453,682	53,297,184	52,943,230	52,109,342

Figure 13: Instruction execution counts from a register deprivation trial

indicate that the system used to compile the benchmark programs uses each additional allocable register available in a manner that reduces the number of instructions that the benchmark programs execute. Although these values indicate the overall trend of the impact that each additional register has on the number of instructions that the benchmark suite executes, it does not provide the information needed to determine how each individual code improvement phase or groups of phases in the compiler are affected. To obtain this information, a pair of register deprivation trials must be performed.

4.2 Comparative measurements

To separate the impact that each additional register has on the code improvement phases of the optimizer from any side effects that they might have on the code generated for each benchmark program, a *base* trial is performed in which the compiler produces code without performing any code improvement transformations. Unlike the optimized trial, the results of the base trial shown in Figure 14 reveal few

Benchmark	Number of allocable registers in probe					
	4	5	6	7	8	9
prog1	160,452,720	160,173,064	160,148,155	160,147,579	160,147,532	160,147,532
prog2	98,753,703	98,692,100	98,682,457	98,678,981	98,678,981	98,678,981
prog3	64,093,511	63,981,342	63,978,106	63,971,423	63,971,367	63,971,288

Figure 14: Instruction execution counts from a register deprivation base trial

improvements as new allocable registers are added. These slight improvements are the result of a reduction in the amount of spill code produced by the local register allocator.

To illustrate the relationship between the reductions in the number of instructions executed that are exclusively attributed to performing code improvement transformations and the number of allocable registers available, each result obtained in the first register deprivation trial is compared against its corresponding base trial result. This process eliminates the impact that the size of the allocable register set has on the components that are not being measured, regardless of whether it is beneficial or detrimental. The following formula, taken from Hennessy and Patterson [HENN90], is used to obtain the percent improvement of the initial optimized trial over the base trial:

$$\text{percent improvement} = \frac{\text{instructions executed}_{\text{base trial}} - \text{instructions executed}_{\text{optimized trial}}}{\text{instructions executed}_{\text{optimized trial}}} \times 100$$

Figure 15 shows the results obtained by applying this formula to each corresponding pair of values in the sample trials. These improvement values indicate that code improvements produce only a slight reduction in

Benchmark	Number of allocable registers in probe					
	4	5	6	7	8	9
prog1	1.23%	18.44%	40.06%	54.76%	61.30%	62.36%
prog2	1.54%	11.37%	22.99%	26.54%	29.29%	30.19%
prog3	1.42%	14.11%	17.49%	20.03%	20.83%	22.76%

Figure 15: Results of comparing two register deprivation trials

the number of instructions executed by the benchmark programs when the minimum number of allocable registers is used. As additional registers are made available, the code improvement phases are able to perform transformations that substantially reduce the number of instructions executed by the benchmark programs. The rate of improvement tapers off quickly after sufficient registers are provided to perform the most beneficial transformations. Once this point is reached, each successive register enables the compiler to make only modest improvements to the benchmark suite.

Instead of displaying the results of a comparative register deprivation measurement in tabular form, the measurement results presented in this dissertation are plotted with the number of allocable registers available to each probe on the X axis and the percent improvement value on the Y axis. These plots display all of the individual improvement values produced by each benchmark program along with a solid line that indicates the average performance improvement for each of the probes in a trial. The corresponding plot of the sample register deprivation experiment results is shown in Figure 16.

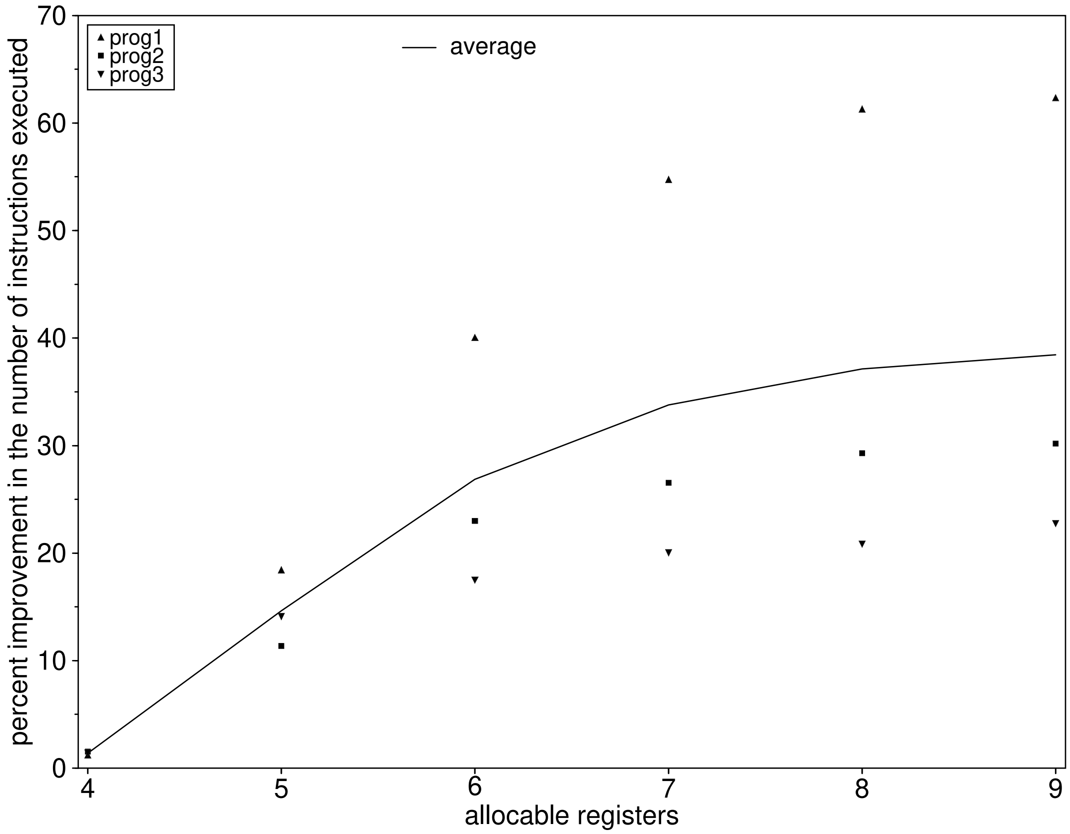


Figure 16: Results of a sample register deprivation experiment

4.3 Benchmark suite

The 13 benchmark suite described in Figure 17 is used to perform all of the register deprivation experiments presented in this dissertation. These benchmark programs were chosen to represent the kinds of codes that consume most of the cycles in a professional development and educational environment. Care has been taken to ensure that the benchmarks provide adequate opportunities to exercise all of the code-improvements performed by *llef*.

Name	Description	Source	Type	Lines of C code
cache	Cache simulation	User code	I/O, Integer	820
compact	Huffman coding file compression	UNIX utility	I/O, Integer	490
diff	Text file comparison	UNIX utility	I/O, Integer	1,800
eqntott	PLA optimizer	SPEC benchmark	CPU, Integer	2,830
espresso	Boolean expression translator	SPEC benchmark	CPU Integer	14,830
gcc	Optimizing compiler	SPEC benchmark	CPU, Integer	92,630
iir	Infinite impulse response filter	Kernel benchmark	CPU, Integer	50
li	LISP interpreter	SPEC benchmark	CPU, Integer	7,750
linpack	Floating-point benchmark	Synthetic benchmark	CPU, Floating-point	930
mincost	VLSI circuit partitioning	User code	CPU, Floating-point	500
nroff	Text formatting	UNIX utility	I/O, Integer	6,900
sort	File sorting and merging	UNIX utility	I/O, Integer	930
tsp	Traveling salesperson problem	User code	CPU, Integer	450

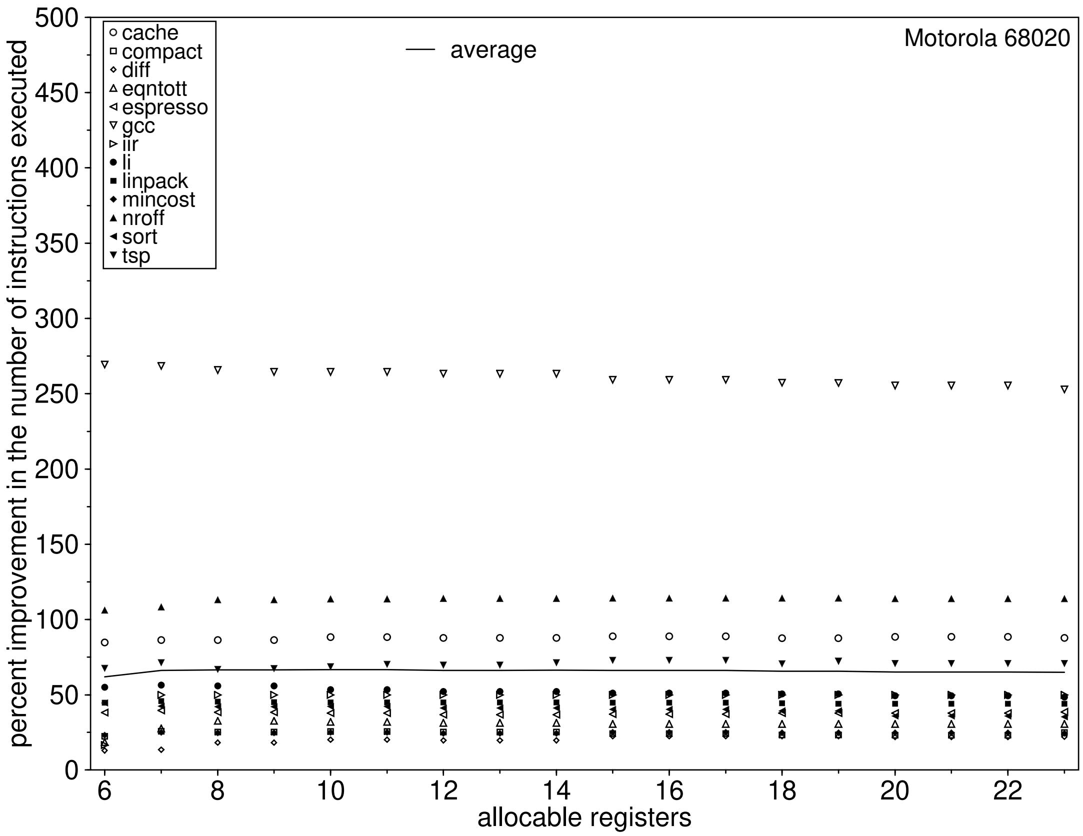
Figure 17: Benchmark suite

4.4 Applications

In addition to measuring the impact that the register set of an architecture has on each code improvement phase or groups of phases in an optimizing compiler, register deprivation measurements can be used to develop well-balanced benchmark suites, help computer architects determine which register set best complements the compilers that will be used to produce code for a new system and enhance the testability of existing validation suites. The following sections discuss how register deprivation measurements can be used to perform these various tasks.

4.4.1 System performance measurements

The process of comparing a register deprivation base trial during which the compiler foregoes all code improvements against a trial that applies the entire set of improvements can be used to determine the impact that the number of allocable registers available has on the entire language translation system. Figure 18 shows how the number of allocable registers affects the number of instructions executed by the object code produced by the *vpo* system presented in Section 2.5.4 for the Motorola 68020 [MOTO85] and the MIPS R3000 [KANE87] processors. These results show that *vpo* reduces the number of instructions executed by only a modest amount even when many registers are available. The reason for this is that *vpo* performs few global-scope code improvements. It is also interesting to note that the improvements obtained on the CISC processor are nearly constant while those on the RISC processor increased as additional registers were made available. This effect can be attributed to the fact that most of the code improvement transformations performed by *vpo* replace memory references with register references. On a CISC machine, the memory references are integrated into the instructions that use their values, so the total number of instructions remains fairly



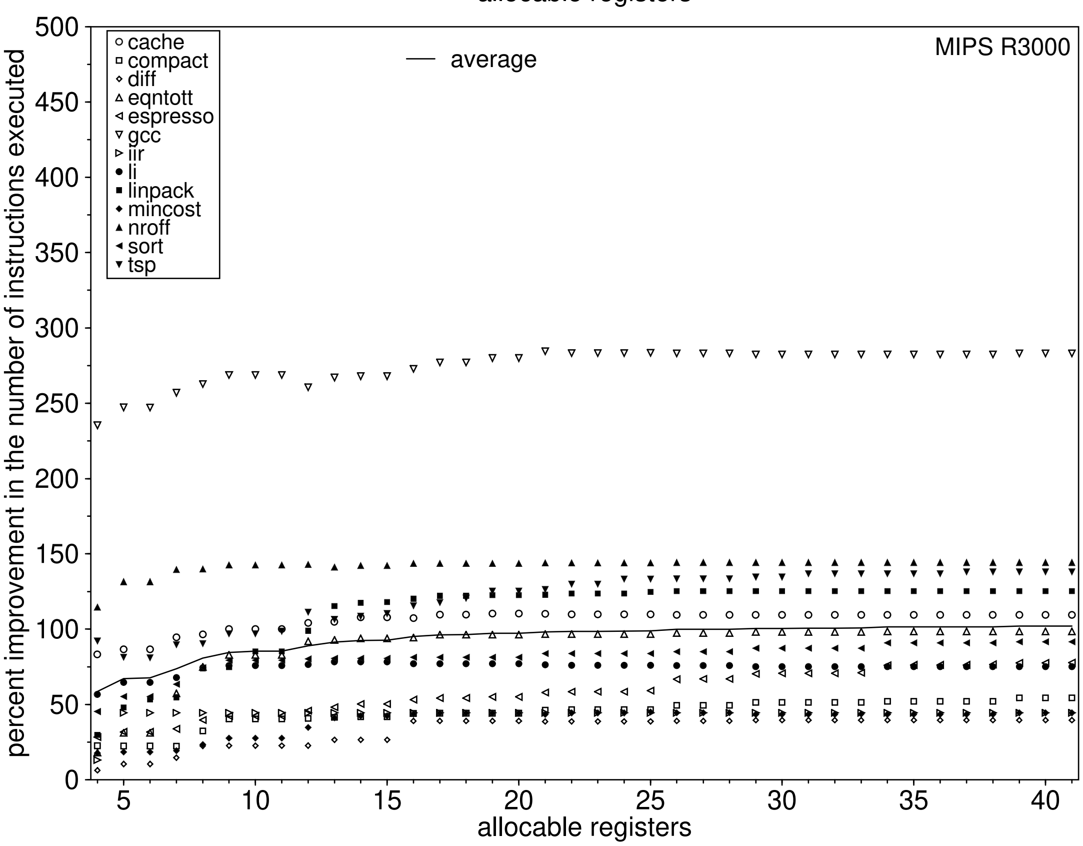


Figure 18: System performance results (instruction counts)

constant. On a RISC architecture, replacing a memory reference with a register reference eliminates a load or store instruction, and these reductions become evident when the number of instructions executed is measured.

It is important to emphasize that the results in Figure 18 show only the effect that the size register set has on the number of instructions executed, which is only one of the many components that affect the quality of the code. Attempting to draw general conclusions about code quality exclusively from this metric would be inappropriate. To obtain further insight on the impact that the number allocable registers available has on *vpo*, a pair of register deprivation trials that measured the number of memory references executed were performed. The results of comparing these two trials are given in Figure 19.

The memory reference comparison results suggest that the size of the register set has a significant impact on the number of memory references that the code produced by *vpo* performs. These results support the claim that the most effective code improvement transformations performed by *vpo* primarily replace memory references with register references, because they illustrate that each additional allocable register produces a substantial decrease in the number of memory references executed. This relation holds true for both CISC and RISC architectures.

While instruction and memory reference counts can provide useful insights concerning the impact that the register set has on the code produced by a compiler, the metric that is most indicative of code quality and overall system performance is execution time measurements. Given that the compiler generates correct code, most users are ultimately concerned with the speed at which the improved code executes. Since neither instruction execution nor memory reference counts alone or in concert can provide this metric, register deprivation trials must be performed to collect execution time information. The impact of varying the number of allocable registers on the overall execution time performance of the code generated by *vpo* is shown in Figure 20.

These results indicate that while there is some correlation between the instruction and memory reference counts and the execution time performance, that it is difficult to predict the magnitude of the execution time performance improvements from the execution time measurements. The instruction execution and memory reference counts do not, for example, indicate the magnitude of the impact that the code-improvements have on the performance of the memory system's cache or the processor's instruction pipeline.

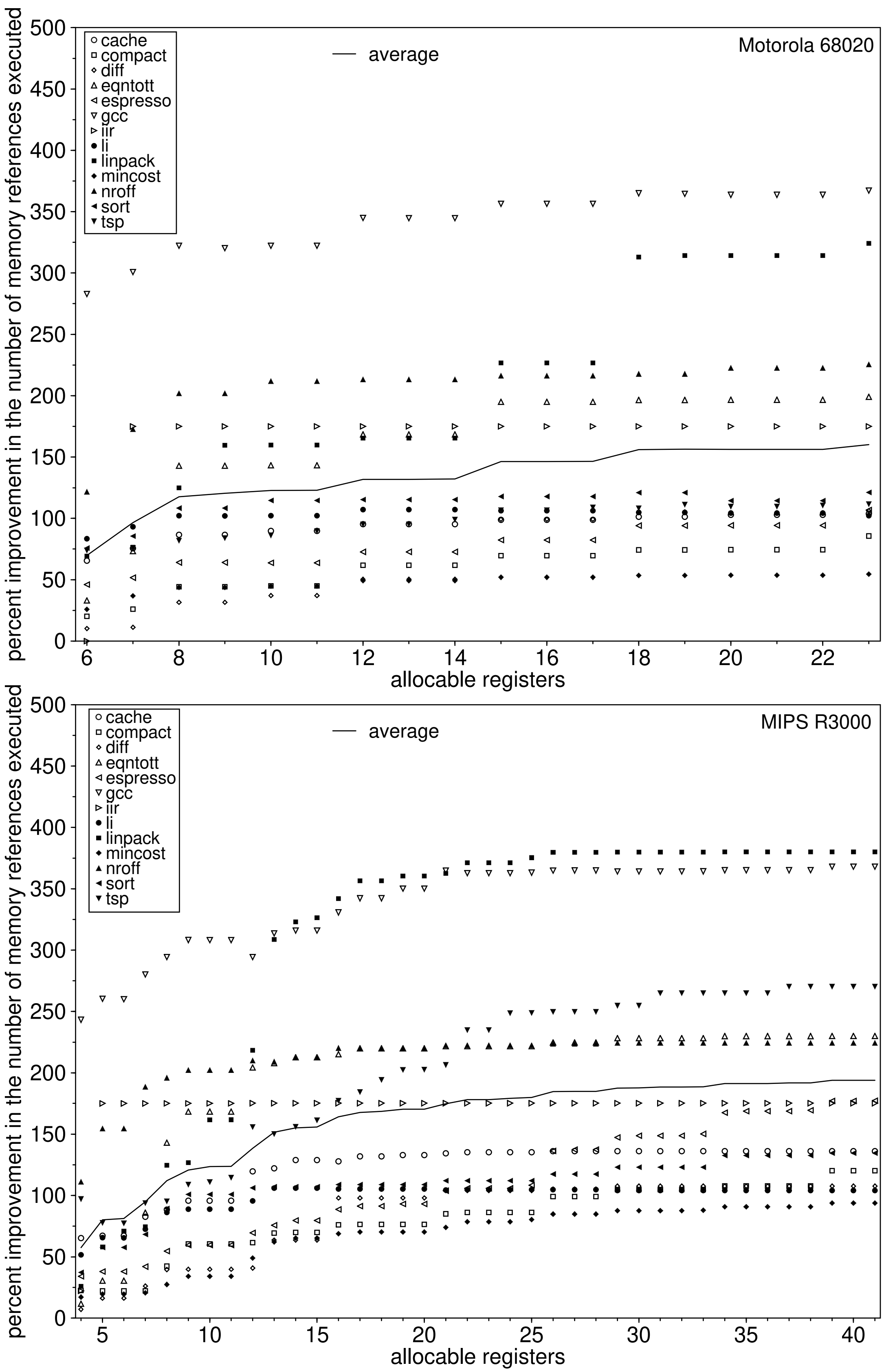


Figure 19: System performance results (memory reference counts)

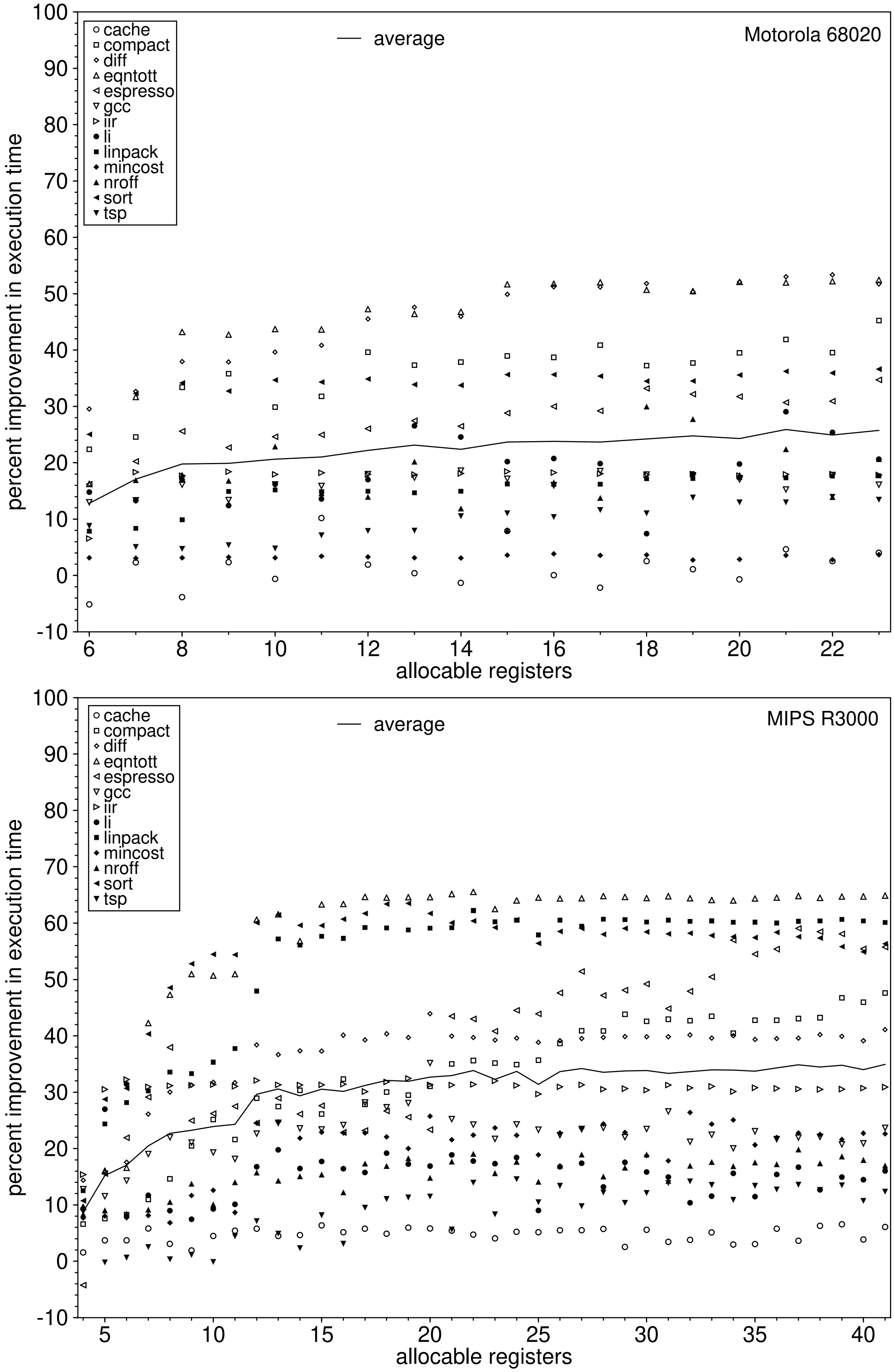


Figure 20: System performance results (execution times)

Also, the execution counts are unaffected by factors such as the amount of time spent performing I/O and the other essential tasks performed by the operating system on behalf of the program. In addition, instruction and memory reference counts are entirely reproducible while the execution time measurements, because of an interaction between the multi-tasking environment and the low resolution clock provided by the system on which they are obtained, have a substantial margin of error. To increase the level of confidence of the execution time register deprivation measurements presented in this dissertation, each individual measurement is repeated five times so that the highest and lowest execution time results can be discarded and the remaining three are averaged to provide the final execution time value. In spite of these precautions, it is not unusual for two identical execution time trials to vary by as much as a few of percent.

Despite the unfortunately wide margin of error, the execution time register deprivation measurements show that *vpo* is able to improve the quality of the code generated for both CISC and RISC architectures even when few allocable registers are available. These improvements are modest even when many registers are available, which is indicative of the fact that *vpo* performs few global-scope code improvement transformations. The positive average improvement obtained even when only the minimal number of registers are available suggests that *vpo* does not suffer from over-optimization.

4.4.2 Phase performance measurements

Register deprivation experiments can also be used to reveal the effect that the size of the register set has on the individual code improvement phases in a language translation system. This is done by comparing a register deprivation trial in which the phase whose impact is to be measured is invoked against a base trial in which it is not. This technique isolates the effect that the size of the register set has on the phase measured from the effect that the register set has on the other code improvement phases. It is important to note, however, that the interactions between the measured phase and the other phases influence the results of the comparison. In most cases, capturing the effect of these interactions along with the direct impact on the measured phase is desirable because it best represents the effect that the measured phase has on the optimizing compiler. In other instances, these interactions can be minimized by not invoking any of the other code improvement phases in both of the register deprivation trials performed.

Figure 21 shows how the size of the register set affects the reduction in the dynamic instruction execution counts produced by *vpo*'s local variable promotion phase. These results where obtained by comparing a base trial in which only control-flow optimizations were performed against a trial in which control-flow optimizations and local variable promotion were invoked. The figure shows that on a CISC architecture, the average improvement in the number of instruction executed was nearly constant for all but the smallest register set sizes. These results are consistent with the results shown in Figure 18 which indicated that replacing memory references with register references have little effect on the dynamic instruction execution counts on architectures that support complex addressing modes. For the MIPS R3000 architecture, the instruction execution counts steadily decreased as more allocable registers were made available and more load and store operations were removed from the code.

The effect of the number of allocable registers on the reduction in the dynamic memory reference counts caused the local variable promotion phase is shown in Figure 22. These results suggest that local variable promotion, even when performed using *vpo*'s unsophisticated technique of assigning a single register to a single user-variable over the lifetime of an entire function, effectively reduces the number of memory references executed by the emitted code. In fact, comparing the improvement values in Figure 22 with those shown in Figure 19 suggests that local variable promotion is responsible for the majority of memory references eliminated as a consequence of the optimizations performed by *vpo*.

4.4.3 Benchmark suite evaluation

Register deprivation measurements can be used to determine if a suite of test programs lacks sufficient amounts of some types of code. An interesting characteristic of the MIPS R3000 register deprivation results shown in Figure 18 is the presence the “flat” areas between probes 5 and 6, 9 and 11, and 14 and 15. These probes add only floating-point registers to the set of allocable registers available to the compiler and this fact suggests that either the benchmark suite does not contain enough floating-point codes or that the compiler cannot use of more than a very small number of floating-point registers effectively. Which of these hypotheses is correct can be determined by adding more floating-point codes to the benchmark suite. If these “flat” areas become less pronounced, one can conclude that the suite was heavily biased towards integer codes. The absence of flat areas in a register deprivation experiment does not necessarily indicate that the benchmark suite has an appropriate balance of integer and floating-point codes, but the ability to detect an imbalance under some circumstances is a useful property.

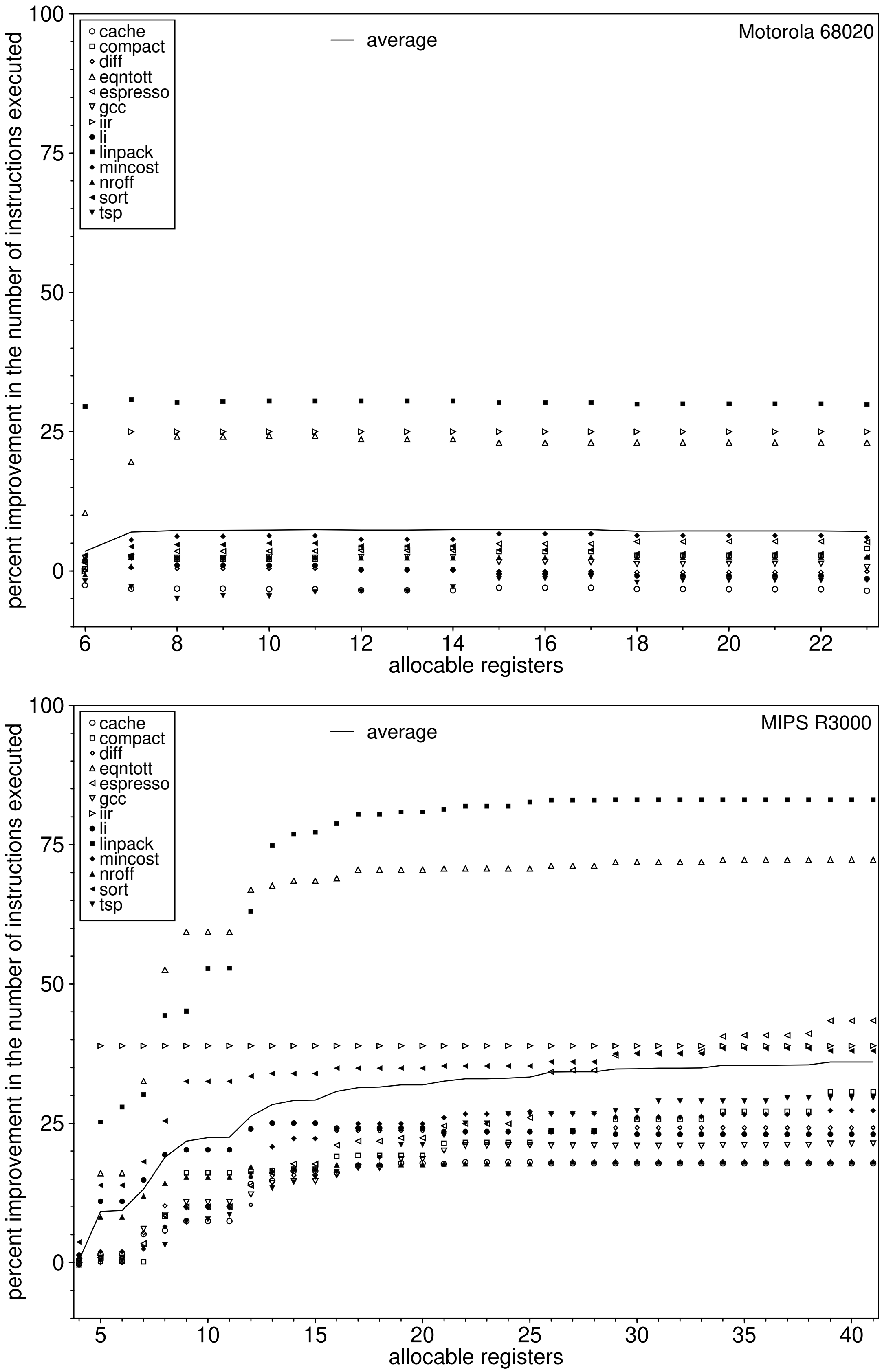


Figure 21: Local variable promotion performance results (instruction execution counts)

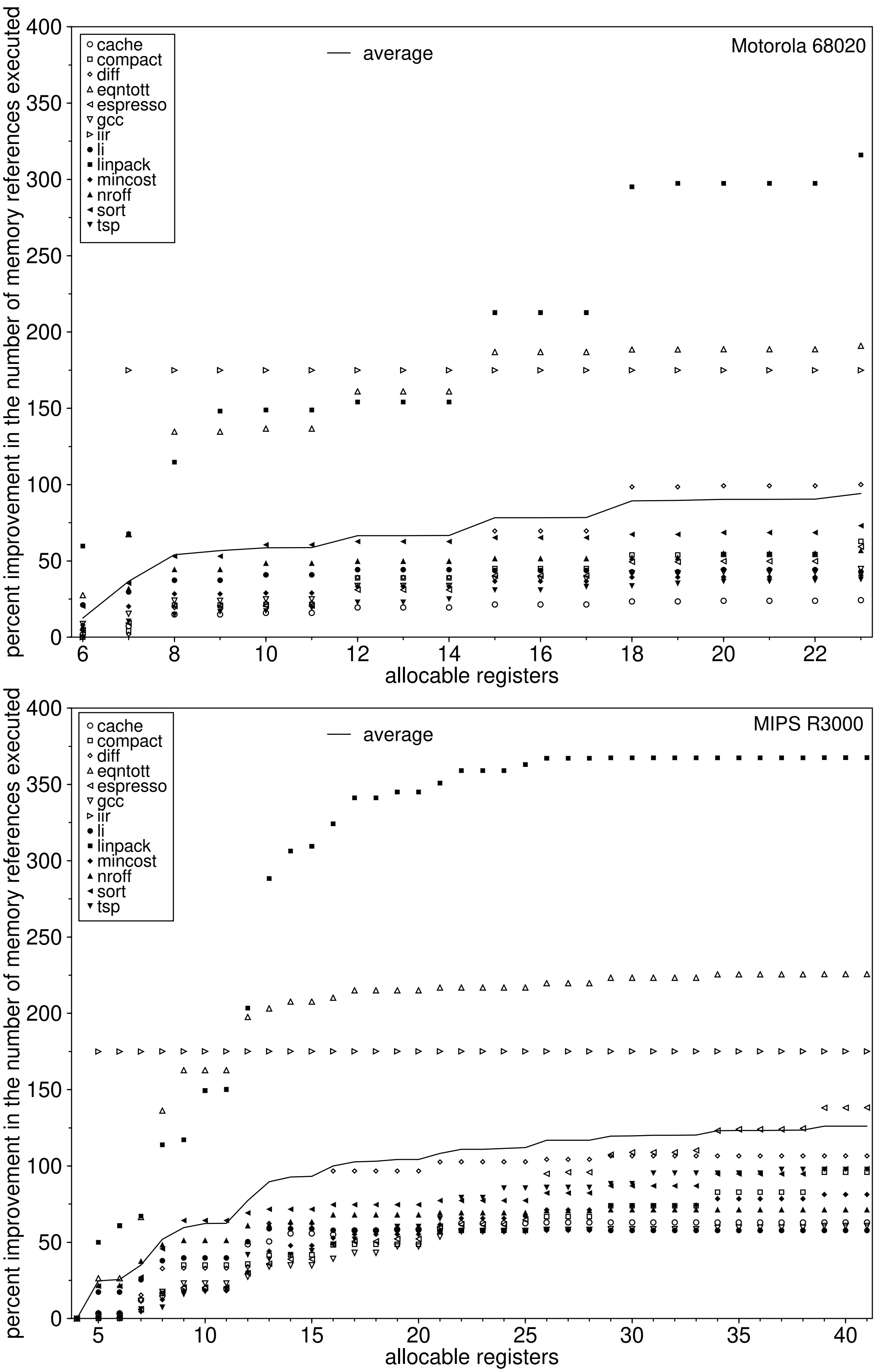


Figure 22: Local variable promotion performance results (memory reference counts)

4.4.4 Architecture design

Register deprivation measurements can be used to provide decisive architecture design information. For example, when attempting to determine whether to sacrifice potential instructions in favor of additional registers, a register deprivation experiment performed on a suitable simulator can be used to determine if the compiler will be able to use the additional registers effectively. This technique can also be employed to determine the number of pipeline stages, processing units and the amount of on-chip cache to provide.

An example of the kinds of insights that register deprivation measurements can help provide can be seen in the Motorola 68020 results shown in Figure 18. These results indicate that most significant memory reference reductions occur between probes 6 through 8, 11 and 12, 14 and 15, 17 and 18 and 22 and 23. Surprisingly, all of these probes increase the number of data registers available to the compiler. This information possibly suggests that the notion of partitioning the integer register set into registers that handle address expressions and registers that process all other integer expressions might not result in the best possible utilization of the register set.

4.4.5 Validation suite enhancement

The process of implementing and obtaining register deprivation measurements exposed deficiencies in parts of *vpo* and *llef* that were thought to be stable. In particular, the code responsible for inserting spill code in the local register assigner and transfer code in the global-scope coloring register allocator was more thoroughly exercised by the register deprivation process than by any of the substantially larger validation suites previously employed. In general, spill mechanisms are most thoroughly tested by the register deprivation probe that uses the fewest number of registers. This does not imply that all deficiencies will be found using only this probe. In some instances, problems that were not uncovered either in the probe with the maximum or minimum number of registers appeared in an intervening probe. One such example involved a problem with the aliasing mechanism used by the common subexpression elimination phase in *vpo*. As the number of available registers decreased, a common subexpression previously held in a register had to be re-computed and, because the re-computation was improperly handled by the aliasing mechanism, the effects of a memory update were misrepresented. This caused a register value to be incorrectly used in lieu of the updated memory value in a subsequent instruction. In later probes, the register containing the incorrect value was no longer available, so the correct value was fetched from the appropriate memory location instead.

Deficiencies in other parts of the optimizer were also uncovered, although these are not easy to characterize. The improved testing capability is derived from the fact that each probe in a register deprivation experiment effectively presents the optimization phases with a slightly mutated version of the original benchmark program. While not as effective as a completely different validation program, a mutated test has the ability to uncover problems that the original version may not. A mutated test also has the advantage of being quite similar to the original test, which is helpful when attempting to locate the point of failure. This capability allows the register deprivation technique to be used as an effective test tool. The nature of the deficiencies uncovered during register deprivation measurements suggest that register augmentation measurements, where the register set of a machine is artificially augmented, might provide similar testing capabilities.

4.5 Implementation

In order to obtain register deprivation measurements, a compiler must be specifically tailored to generate object code that uses only a subset of the registers available on a target machine. This usually entails more than just reducing the number of registers available to the register allocation algorithms. In some cases, modifications will also need to be made to the calling convention and to the code generator. Since the process of collecting register deprivation measurements can be lengthy and repetitive, it is most useful if it can be automated.

The following sections describe how register deprivation mechanism was implemented in *llef*. The compilers for the MIPS R3000 [KANE87] and the Motorola 68020 [MOTO85] processors were instrumented in order to measure their performance. These processors were chosen to ensure that both RISC and CISC systems, which place significantly different demands on the compiler, are represented.

4.5.1 Measurements

The object code produced by *llef* is instrumented to determine the number of instructions and memory references executed using *ease* [WHAL90]. When *ease* is employed, the object code produced by the compiler is instrumented with execution counters. After the instrumented code executes, the values of these counters are written to a file along with instruction information for each basic block. A report generator uses this information to provide a detailed execution profile that includes the total number of instructions and memory references executed by the object code.

To facilitate the task of obtaining execution time measurements, *ease* provides a facility for measuring the amount of processor time spent executing a program. This facility operates on the same principle as the */bin/time* utility provided with most UNIX systems, but allows execution time results to be directed into a separate file other than the execution log file so that they can be more easily read by the graph generation utility program. Because the execution time information provided by UNIX systems can be affected by factors that are not easily controlled, a number of precautions have been taken to minimize the impact of these factors. First, to minimize the errors introduced by the fairly long interval between system clock ticks, all benchmarks were invoked with input data sets so that they executed for at least 30 seconds. To eliminate clock drift and other external effects, each program was executed five times, the highest and lowest times were discarded and the remaining three times were averaged to provide a single time value. Finally, an effort was made to identify and reduce the external factors that affect the times. These efforts include running the time trials on lightly-loaded machines and avoiding the use of dynamic-link library functions that may be overwritten by the system and whose reload time may be partially charged to the user. While these measures increase the accuracy of the execution time values, the resolution provided cannot reveal performance differences that are less than one or two percent. In such cases, the instruction and memory execution counts become the only accurate measures of performance.

4.5.2 Registers

Although the number of allocable registers used by the register allocation algorithms presented in this dissertation are easily controlled by modifying a simple register description[1], careful consideration must be given to determining which registers to eliminate from each successive probe in a register deprivation experiment. This task is complicated by the fact that register sets commonly consist of more than one register type (e.g. general-purpose and floating-point registers). If the intent of the register deprivation experiment is to examine only integer codes or an optimization that operates only on loops that manipulate floating-point values, then registers of one type only need to be considered. Since this dissertation is interested in measuring

[1] The register description is discussed in Section 5.3.1.

the effects of register deprivation in general, the integer to floating-point register ratio is maintained as close to its original value as possible.

4.5.3 Calling convention

Because the relative size of the volatile register partition prescribed by the calling convention has a significant impact on the quality of the object code [DAVI91], the register deprivation measurements strive to maintain a volatile to non-volatile register ratio that is as close to the original ratio as possible. Many calling conventions pass actual parameter values in a subset of the volatile register set. Even though the number of registers reserved for parameter passing is usually small, register deprivation trials will eventually reach the point where insufficient registers remain to implement the original calling convention. Because of this, the ability to modify the calling convention is essential to performing a thorough register deprivation experiment. The register deprivation experiments performed on the MIPS R3000, which passes argument values across functions in registers, adjust the number of registers used to pass argument values so that the ratio of the number of argument passing registers to overall allocable registers remains fairly constant.

4.5.4 Probes

In order to maintain the register ratios discussed in the previous two sections, register deprivation trials start with the original set of allocable registers available to the compiler. This number is usually slightly less than the number of registers available on the machine, since some registers are reserved for special purposes (e.g., stack pointer, frame-pointer, etc.). To determine which register will be removed from each successive probe, a trial is performed where, for each type of register available, one of the volatile registers and one of the non-volatile registers are removed. The trial which results in the integer-to-floating-point and volatile-to-non-volatile ratios that are closest to the original values for these ratios is chosen. This process continues with the next successive probe until no additional registers can be withheld from the compiler. On most architectures, this point is usually reached when there are a total of two registers for each register type, which is the minimum number of registers needed to perform a binary operation. On machines where binary operations can reference memory locations directly, this limit may be lower.

Figure 23 shows which registers are available for each of the probes in the register deprivation experiments performed on a MIPS R3000. Of the forty-eight total general-purpose and floating-point

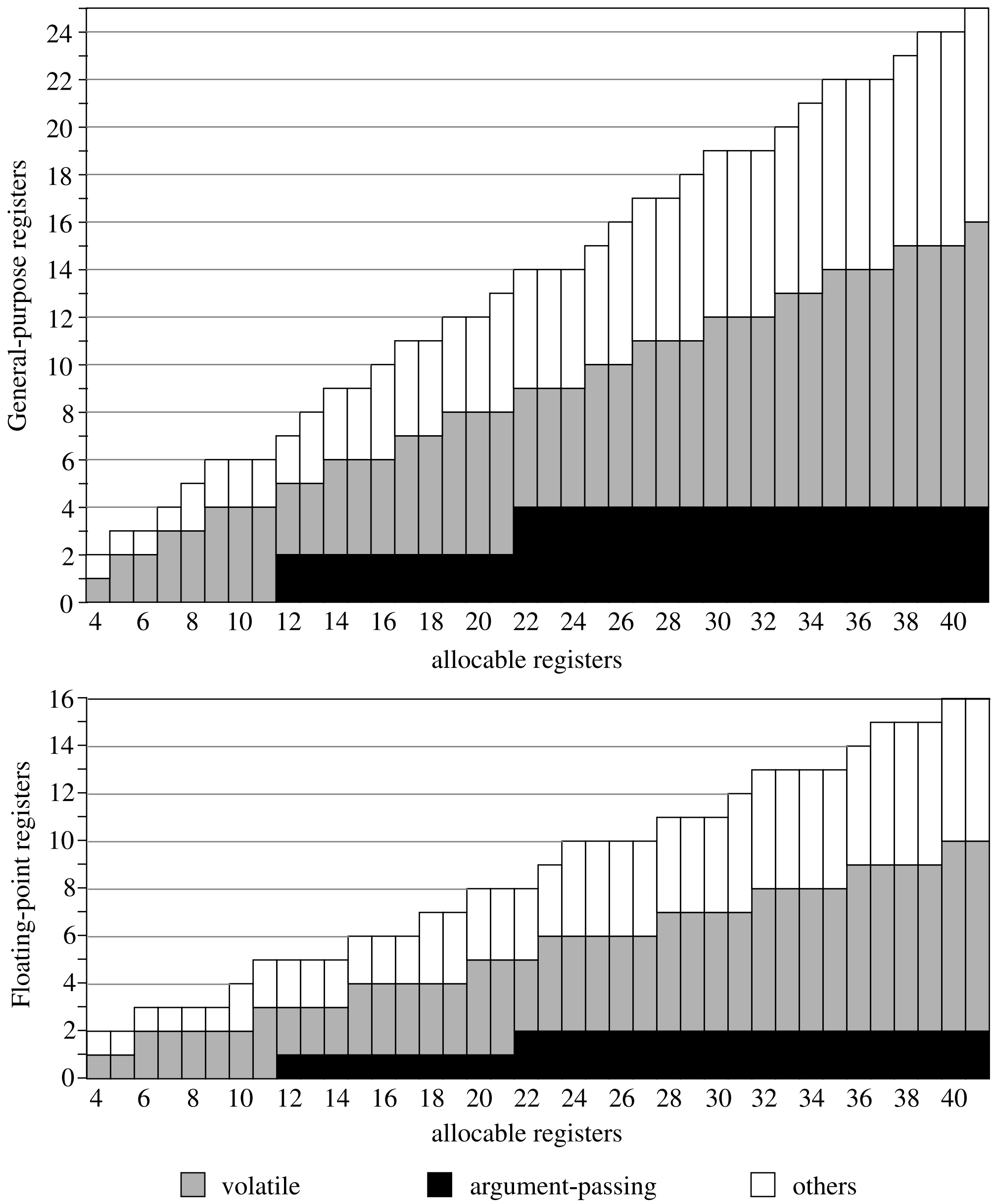


Figure 23: Registers available on each MIPS R3000 probe

registers, only the forty-one allocable registers are shown. The remaining registers are reserved by the assembler or the operating system, are special in that they always return a fixed constant value, or are used exclusively as argument pointers or frame pointers.

Figure 24 shows the registers available for each of the probes in the register deprivation experiments performed on the Motorola 68020. This architecture provides three different classes of registers: address, data, and floating-point. Of the twenty-four total registers in these three classes, one of the address registers

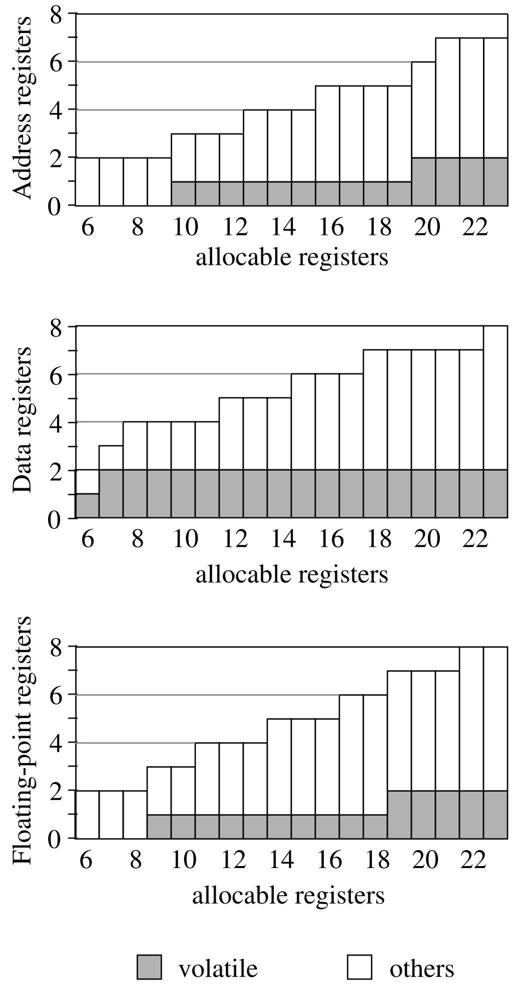


Figure 24: Registers available on each Motorola 68020 probe

is reserved to serve the dual role of stack pointer and frame pointer. Because a pair of registers is required from each of the classes is required to generate code, the minimum number of registers used is six. Unlike the volatile to non-volatile register ratios for the address and floating-point register classes, the ratios for the data register class are not kept fairly constant because the calling convention uses the volatile data registers to return function values. In the final six register probe, it is important to note that although there is only one allocable volatile data register, that functions that return floating-point values will actually use an additional volatile register in the return sequence. Because these code sequences make up a minute portion of the instructions executed in each probe, their effect on the outcome of the experiments are negligible.

4.5.5 Library Functions

On systems where the source code to the library functions is available, changes in the calling convention can be implemented by simply re-compiling the library. When the source code is either not available, or large portions are written in a language that does not allow the calling convention to be easily modified, a mechanism must be implemented to allow the use of the existing libraries. This mechanism entails generating special code before each external function call that passes actual parameters to a library function that expects argument values to arrive in specific registers. On some systems, it may not be possible to differentiate between calls to library functions and calls to user functions so that special interface code is needed before every function call. This special code introduces three problems:

- the code must be annotated so that it is not counted as part of the instructions or memory references that are executed by the program,
- the library code is not modified to reflect the code that would be generated if the register limits of the probe were fully enforced and
- the execution time of the programs is affected so that wall-clock execution times are not entirely accurate.

The first problem is trivially handled by *ease*. In order to prevent the second problem from affecting the results of instruction and memory reference count register deprivation test, none of the library codes are instrumented so that the number of instructions and memory references executed by the library functions are not counted in any of the probes in the register deprivation measurements. The last problem affects only the experiments that measure execution time, but this factor is not overwhelmingly significant because the benchmarks programs selected for these measurements spend only a small portion of their time executing library functions.

When generating code for a probe that limits the number of parameter passing registers, actual parameters that would normally be placed into these registers are placed in the memory locations normally reserved for actual parameter values that cannot be passed in registers. Special interface code, consisting of a series of uninstrumented load instructions to transfer actual parameter values to the registers prescribed by the standard calling convention, is generated between the point where these values are stored to memory and the external function call. The library function code is satisfied because it expects these parameter values to be in the registers that the special interface code has put them in. The register deprivation measurements that measure instruction and memory reference counts remain accurate because the execution counts are identical to those produced with a completely modified calling convention. The experiments that measure execution times are slightly affected by this approach.

Figure 25(a) shows a sample of the code that is generated on the MIPS R3000 for a probe with only four allocable registers. The source code consists of a simple call to a function that accepts two integer

The following C function call:

```
max(15, 19)
```

Produces the following assembly code:

```
li    $2,15
li    $16,19
sw    $2,0($sp)
sw    $16,4($sp)
lw    $4,0($sp)
lw    $5,4($sp)
jal   max
```

(a)

The following C function declaration:

```
int max(a, b)
int a, b;
{
}
```

Produces the following function prelude code:

```
max:  subu  $sp,$sp,8
      sw    $16,4($sp)
      sw    $4,8($sp)
      sw    $5,12($sp)
```

(b)

Figure 25: Calling convention interface code

parameter values. Normally, the calling convention would pass the first actual parameter value in general-purpose register four and the second actual parameter value in general-purpose register five. Since both of these registers are withheld from the compiler in this probe, the calling convention is modified so that these actual parameter values are passed in a special location on the caller's activation record. In the event that the function called is a library function still adhering to the original calling convention, the actual parameter values would not be passed properly. The special interface code, shown in italics, is generated to ensure that any function still following the original calling convention receives the actual parameter values correctly. Since the special interface code is not instrumented by *ease*, the execution counts generated when the code is executed reflects the instructions generated to satisfy the new calling convention. The execution time measurements, however, are not entirely accurate because they include the time required to execute the interface code.

In the C library, there are functions that call user-defined functions. An example of such a library function is *qsort()*. This function sorts an arbitrary array of data and one of the parameters passed to it is a pointer to a user-defined function that accepts a pair of pointers to elements in the array and returns a value that indicates which of the two items should be ordered first. If the library functions continue to rely on the original calling convention, the actual parameter values may be available in the registers specified by the original calling convention rather than in the memory locations from which the new code must load them. The solution once again is to introduce special interface code similar to the code required prior to an external function call. Figure 25(b) shows a sample of the interface code that is generated by the four register probe on the MIPS R3000. This code is placed at the entry point of functions that accept parameters in order to transfer the actual parameter values from the registers dictated by the original calling convention to memory to satisfy the requirements of the calling convention used by the probe.

4.6 Summary

This chapter has discussed register deprivation measurements and their uses. The register deprivation mechanism is used in this dissertation to measure the efficacy of the various register allocation and phase interaction reduction strategies presented in Chapter 6. This technique was developed primarily to gauge the effect that the size of the register set provided by the target architecture has on a language translation system and on the individual code-improvement transformations that it performs. By varying the number of allocable registers available to perform code generation and code-improvement transformations, register deprivation experiments effectively present the compiler with an entire set of virtual architectures. The differences in the way that the various phases of the compiler react to this set of architectures is illustrated by juxtaposing them on a single plot. In addition, the register deprivation mechanism enhances the ability of the benchmark suite to test the compiler, thus presenting the compiler with a more thorough validation suite.

CHAPTER 5

EXPERIMENTAL FRAMEWORK

This chapter describes the low-level experimental framework, *llef*, which was used to develop and evaluate register allocation and phase interaction reduction strategies that operate effectively across a wide range of architectures. This framework is uniquely suited to this task because it provides:

- a front-end and code generator that can produce un-optimized low-level code for various popular CISC and RISC architectures from C source code,
- a comprehensive set of architecture-independent optimization phases that perform both architecture-independent and architecture-specific code-improvement transformations directly on low-level code,
- a flexible phase ordering scheme that exploits the invariance of the low-level representation and the ability of the optimization phases to operate directly on it to change the order in which the phases are invoked and to re-invoke phases as needed,
- a description of the target architecture that recognizes low-level code sequences that represent valid target-architecture instructions and provides instruction cost and scheduling information,
- a set of register allocation and assignment primitives that are automatically generated from a simple description of the register resources provided by the target architecture and
- the ability to perform experiments using *ease* and register deprivation measurements.

Apart from its utility as an optimizer development tool, *llef* represents an important contribution to the field of optimization because it demonstrates that performing global code-improvement transformations on low-level code is not only feasible, but desirable for the following reasons:

- most optimizations are either architecture-specific, or interact with at least one architecture-specific optimization so that it is most effectively performed at the machine level,
- low-level code exposes all of the nuances of the target architecture, allowing the optimization phases to accurately gauge the impact of each transformation and to fully exploit the capabilities of the target architecture and
- because low-level code can be represented using an architecture-independent notation, the bulk of the optimizations can be performed by architecture-independent code, thus providing a very high degree of retargetability.

The execution time performance results shown in Figure 26 support the claim that *llef* is capable of generating production-quality object code across a substantial number of diverse architectures. This table presents the

Motorola 68020/68881	*eqntott*	*espresso*	*gcc*	*li*
gcc 1.37	288.0	0.0[1]	0.0[1]	1843.1
cc	356.5	878.6	418.5	2030.1
llef	322.7	825.0	414.9	1816.8
VAX-11 8600	*eqntott*	*espresso*	*gcc*	*li*
gcc 1.37	254.4	0.0[2]	368.5	1714.6
cc	417.4	761.2	400.1	1704.3
llef	277.7	599.3	395.6	1526.2
Intel 30386/30387	*eqntott*	*espresso*	*gcc*	*li*
gcc 1.40	278.0	510.8	221.4	1070.0
cc	341.5	584.3	278.2	914.3
llef	326.1	504.3	254.3	1073.0
SUN SPARC	*eqntott*	*espresso*	*gcc*	*li*
gcc 1.37	64.9	136.9	0.0[1]	399.0
cc	57.1	124.1	76.5	339.7
llef	48.6	124.4	79.5	343.6
MIPS R3000	*eqntott*	*espresso*	*gcc*	*li*
gcc 1.39	59.7	106.1	0.0[2]	285.6
cc	55.1	100.8	72.1	258.6
llef	50.9	108.1	76.5	275.1
Motorola 88100	*eqntott*	*espresso*	*gcc*	*li*
gcc 2.2.2	87.4	165.2	0.0[2]	414.7
Green Hills *C-88000* 1.8.4	131.0	191.4	110.5	446.5
llef	96.6	176.8	113.8	422.6

Figure 26: Execution time comparisons between *llef* and production compilers[3]

[1] Benchmark did not execute properly.
[2] Benchmark could not be compiled.
[3] All times given in seconds as reported by */bin/time.*

results obtained by directly comparing the execution time of the code generated by *llef* for the C component of the Standards Performance Evaluation Corporation (SPEC) benchmarks to the code generated by optimizing production compilers on six different architectures. These results show that *llef* is capable of generating production-quality code on architectures that provide both complex and simple instruction sets as well as architectures that provide both large general-purpose register sets and more modest and restricted register sets.

The code generation and instruction selection strategies used in *llef* were taken from the *vpo* system described in Section 2.5.4. Unlike its predecessor, however, *llef* is a high-quality language translation system that incorporates a comprehensive set of local and global optimizations. After a brief overview of *llef*, each section in this chapter is devoted either to discussing the enhancements made to the phases borrowed from *vpo* or to describing the algorithms used to perform the optimizations that were not included in *vpo*. In particular, this chapter presents:

- an improved strategy for performing global instruction selection,
- a register description that concisely provides all of the architecture-dependent register information that is needed to retarget *llef*'s register allocation and assignment primitives,
- a local register allocator that facilitates the use of argument-passing registers and instructions that reference specific registers implicitly,
- a retargetable instruction reorganizer that significantly reduces the amount of transfer code generated by the local register allocator when few allocable registers are available,
- a global-scope coloring register allocator that performs effective variable promotion without generating spill code,
- techniques for performing loop-invariant code-motion, loop strength reduction, recurrence optimization and induction variable elimination on a low-level representation and
- machine-independent algorithms that perform instruction scheduling and delay slot filling effectively.

For the sake of brevity, this chapter does not discuss the portions of *llef* that were taken from *vpo* and remain relatively unchanged. Details concerning the instruction selection and common subexpression elimination code improvement phases are omitted because they were described in a previous document [BENI89].

5.1 Overview

The front-end of *llef* consists of a *pcc*-based [JOHN78b] retargetable front-end which transforms traditional C [KERN78] into a high-level, stack-based intermediate representation and a machine-specific code generator which transforms this high-level code into same low-level, register transfer-based representation used by *vpo*. Because this machine-independent notation is used exclusively and all of the optimization phases perform code-improvement transformations directly on this representation, a large percentage of *llef* remains invariant across different target architectures. In addition, the ability to represent specific target machine instructions using a low-level representation allows the inclusion of machine-dependent optimization phases that are highly retargetable while remaining entirely effective.

The low-level intermediate language used by *llef* is the register transfer notation, which is based on the Instruction Set Processor (ISP) notation developed by Bell and Newell [BELL71]. Register transfers represent machine instructions by describing the effects that the instructions have on architecture's storage calls. While assembly syntax varies from machine to machine, the register transfer specification of an operation is identical across machines. For example, the following list shows register-to-register add instructions on various machines using their assembly language syntax:

```
a       r1=r1,r2            -- IBM RS/6000
ar      1,2                 -- IBM 370
add     1,2                 -- DecSystem 10
ix1     x1+x2               -- CDC 6600
add     %o2,%o1,%o1         -- SUN SPARC
add.l   d2,d1               -- Motorola 68020
addl2   r2,r1               -- VAX-11
addu    $1,$1,$2            -- MIPS R3000
```

Using the register transfer notation, each of these instructions is represented by the following register transfer:

```
r[1]=r[1]+r[2];
```

In contrast to the assembly language specification of the instruction, the register transfer unambiguously describes the action performed by the instruction. Thus, the above register transfer clearly indicates which register receives the result of the addition operation. Assembly language instructions, on the other hand, use conventions to designate source and destination operands.

Registers in the register transfer notation are represented using strings of the form *r*[*x*], where *r* is a lower-case alphabetic character representing the assignment type of the register (see Section 5.3.1) and *x* is an integer constant denoting the number of the register. Memory references are of the form *M*[*address*], where *M* is an upper-case alphabetic character and *address* is an arbitrarily complex expression that computes the value of the address to be referenced. Integer constants and operators are represented using the standard **C** programming language notation. Global and local identifiers are used to represent address locations and constant offset values, respectively. The set of identifiers consisting of two upper-case characters are reserved to denote special storage cells and complex operations. For example, the program counter and the integer condition codes are denoted by the identifiers `PC` and `CC`. Thus, an instruction that compares two registers and sets the condition code register accordingly is represented as:

```
CC=w[4]?w[5];
```

Likewise, a conditional branch instruction which branches to a labelled location if the condition code register indicates that the value in `w[4]` is less than the value in `w[5]` in the above comparison is represented as:

```
PC=CC<0,L54;
```

For historical reasons, the identifiers `ST` and `RT` represent the input and output ports of a calling stack, respectively. This stack, which might or might not exists on a target architecture is used to describe the instructions that call external functions and return from them. For example, an external function call is represented as:

`ST=`*address*`,`*arguments*`;`

where *address* is the address of the external function to call and *arguments* is a list of registers containing the values to pass to the external function and a return instruction is represented as:

```
PC=RT;
```

Complex operations such as extracting the high and low parts of a constant are also represented using reserved two-character identifiers. For example, the following pair of register transfers:

```
w[7]=HI[_global_id];
w[7]=w[7]+LO[_global_id];
```

would be used on some RISC architectures to describe the pair of instructions used to construct the value of the address denoted by `_global_id` into register `w[7]` by merging the high and low parts of the address together. Appendix A contains an informal specification of the register transfer notation.

All of the optimization phases operate directly on the register transfer representation and most of them can be invoked in any order and at any time in the optimization process. One advantage of this strategy is that each phase can specialize in performing only the transformations that are directly related to the optimization that it implements. Should these transformations produce new opportunities to perform other optimizations, *llef* automatically invokes or reinvokes the optimization phases that have the potential to further improve the code. This *phase iteration* strategy allows *llef* to automatically determine the phase ordering that best suits the needs not only of each target architecture, but also of each individual source function.

Figure 27 shows the procedure used to control the optimization process. Many of the optimization phases invoked by this procedure are described in this chapter. Although parts of the phase ordering are predetermined, the phase iteration mechanism dynamically determines the majority of the invocation sequence. Many optimization phases return a value indicating that transformations were performed on the code and that there may be new opportunities to apply additional transformations.

```
proc optimize is
    build_cfg()
    control_flow_optimizations()
    set_local_combiner_links()
    C = instruction_selection()
    live_variable_dataflow_analysis()
    build_ssa_form()
    if set_global_combiner_links() then
        C = instruction_selection()
        live_variable_dataflow_analysis()
    endif
    evaluation_order_determination()
    if local_register_assignment() then
        C = instruction_selection()
    endif
    if loop_determination_controlflow_analysis() then
        loop_branch_minimization()
        loop_inversion()
    endif
    repeat
        A = FALSE
        repeat
            C = FALSE
            live_variable_dataflow_analysis()
            A = dead_variable_elimination()
            if local_variable_promotion() then
                C = instruction_selection()
                A = TRUE
            endif
        until ¬C
        if A then
            C = common_subexpression_elimination()
            live_variable_dataflow_analysis()
            C = C ∨ dead_variable_elimination()
            C = C ∨ loop_optimizations()
            C = C ∨ instruction_selection()
        endif
    until ¬C
    control_flow_optimizations()
    insert_function_prologue_and_epilogue()
    instruction_scheduling()
    fill_branch_delay_slots()
endproc
```

Figure 27: Optimization driver procedure

5.2 Global-scope instruction selection

The global-scope instruction selection algorithm used by *vpo* calculates the reaching definitions dataflow information for each function in order to properly link logically adjacent instructions across basic blocks [BENI89]. To calculate reaching definitions as efficiently as possible, *vpo* creates and operates on bit vectors that reserve a single bit position for each instruction in the function [AHO86]. Because two of these vectors are needed to represent the reaching definitions at the entry and exit point of each basic block in the function and because single-instruction basic blocks are possible, this reaching definitions algorithm has a worst-case space utilization of $O(n^2)$, where n is the number of instructions in the function. Because of this, *vpo* requires substantial amounts memory to compile large functions. Also, since the length of the bit vectors determines the amount of time that it takes to perform the vector intersection and union operations that dominate the execution time of the algorithm, large functions also require an excessive amount of time to compile.

The purpose for calculating the reaching definition information is to determine how many different assignments can possibly reach each register reference. A recent development in data structures called the Static Single Assignment (SSA) form, can also be used to determine if how many different definitions reach a particular reference [CYTR91]. Although the algorithms used to construct the SSA form are more difficult to implement and understand than the iterative algorithm used to calculate reaching definitions, their storage space and execution time complexities are practically linear with respect to the size of the function.

The SSA form of a function is obtained by adding new assignments, called ϕ-*functions*, at each location in the CFG where multiple values of a variable converge. The ϕ-functions are placed so as to satisfy the requirement that every reference to a variable is reached by only one possible assignment in the function. To illustrate this process, consider the simple **if-then-else** statement in Figure 28(a). The value of `V` used in

```
1       if (P)
2           V = 4;
3       else
4           V = 6;
5       return(V);
```

(a)

```
1       if (P)
2           V1 = 4;
3       else
4           V2 = 6;
        V3 = ϕ(V1, V2);
5       return(V3);
```

(b)

Figure 28: The SSA form for an **if-then-else** construct

line 5 potentially comes from either the assignment in line 2 or the one in line 4. When the SSA form, which is shown in Figure 28(b), is constructed, a ϕ-function is introduced at the point where the two cases of the **if-then-else** statement merge to provide a new assignment to `V`. This ϕ-function represents the confluence of the values of V. Every use of V now references a value with only one potential source. Each reference can now be numbered to indicate which assignment value reaches it.

Instead of using the reaching definitions information, the SSA variant of the global combiner link insertion algorithm finds the assignment associated with each global reference and, if it is a ϕ-function, abstains from setting a global link because the value can come from more than one assignment. If the assignment is an instruction, then a global combiner link is set provided that the reference is the only possible use of the value. These two conditions satisfy all of the requirements for setting "safe" global combiner links [BENI89].

The SSA variant of the global combiner link phase was compared against the traditional reaching definition version using two large functions. The first function was a 1500 line parser for a compiler front-end. Using the reaching definition version of the algorithm, *vpo* requires 11,413,936 bytes of memory and 39.4 seconds to compile this function on a SUN Microsystems SPARC2 workstation, while the SSA variant used by *llef* requires only 3,367,592 bytes of memory and 16.1 seconds to produce the same code. The second function was a 2400 line machine-generated compiler validation test containing a complete cross product of binary operations and basic integer and floating-point types. This function required 29,086,592 bytes of memory and 61.7 seconds to compile using the reaching definitions version in *vpo* and only 5,343,872 bytes of memory and 17.8 seconds with the SSA variant in *llef*. For small functions, the amount of memory used was only slightly greater on average for the SSA version while constructing the SSA form for a module full of small functions rarely took over a few tenths of a second more time than computing the reaching definitions did. While it would have been possible to incorporate both versions of the global combiner link algorithms *into llef* and invoke them depending on the number of instructions and basic blocks present in each function, the amount of time saved by computing the reaching definition information instead of building the SSA form for small functions was deemed to be too small to justify the increased complexity and maintenance effort that using both algorithms would incur. For this reason, *llef* exclusively uses the SSA form to set global combiner links.

5.3 Local register assignment

The purpose of performing local-scope register assignment is to assign the pseudo registers created by the code generator to actual target machine registers. Few inter-block pseudo register values exist at this stage in the optimization process because no global-scope, register-consuming transformations have yet been performed. Under these conditions, a local-scope strategy results in a quick, near-optimal register assignment. After assignment, the actual register requirements of the code are fully exposed to subsequent optimization phases.

The advantage of postponing global optimizations until code generation and register assignment are performed is that registers do not have to be reserved to satisfy code generation requirements. Registers not used for code generation are available to perform optimizations and those that are used for code generation can be re-used in regions where they are not live. This eliminates the interaction between code generation and register allocation that plagues more traditional optimizers: it is difficult to determine how many registers are available to perform optimizations until code generation is performed and it is difficult to determine what code will be generated until all of the register-consuming optimizations are applied. Because code generation precedes optimizations, *llef* knows which registers are available to perform transformations before attempting any register-consuming optimizations.

The following sections present the aspects of the local register allocation process that are unique to *llef*. These include:

- the use of a simple register description to simplify the task of retargeting the register allocation algorithms used by the optimizer to a new target architecture,
- a mechanism that allows the code generator to explicitly use any target machine registers without precluding the local register assigner from using them and
- the re-ordering of expression calculation sequences to significantly reduce the amount of spill and transfer code generated by the local register allocator when allocable registers are scarce.

5.3.1 Register description

A simple description of the target architecture's registers provides all the information needed to retarget the register allocation and assignment primitives provided by *llef*. This register description provides information concerning the number of registers available on the target architecture, the types of data that they can hold, and how they are partitioned by the calling convention. The register description is processed by a utility program called *regtool* to generate a source file containing the declarations needed by the architecture-independent register allocation algorithms. The advantage of the register description is that it replaces roughly 300 lines of complex, architecture-dependent C source code with about 40 lines of register description text.

```
types BTREG, HTREG, WTREG, STREG, DTREG
class = general_purpose
      number = 32
      reserve = 0, 14, 15, 30, 31
      volatile = 8..13, 1..7
      type = WTREG
            alignment = 1
            size = 1
            invariant = 14, 30
      endtype
      type = BTREG, HTREG
            alignment = 1
            size = 1
      endtype
endclass
class = floating_point
      number = 32
      volatile = 0..31
      type = STREG
            alignment = 1
            size = 1
      endtype
      type = DTREG
            alignment = 2
            size = 2
      endtype
endclass
class = SPILL
      number = 32
      type = BTREG, HTREG, WTREG, STREG
            alignment = 1
            size = 1
      endtype
      type = DTREG
            alignment = 2
            size = 2
      endtype
endclass
```

Figure 29: Register description for the SPARC architecture

This approach simplifies the retargeting process and encourages experimentation with the calling convention and the order in which the target machine's registers should be assigned.

Figure 29 shows the entire register description for the SUN Microsystems SPARC architecture [SUNM87]. The first line of the description declares the basic assignment types. These types are used to denote the order in which registers are assigned and the constraints that govern their use. Because an architecture may have multiple rules pertaining to the use of the registers, a particular target machine register may belong to more than one assignment type. The five assignment types defined for the SPARC are used to assign registers to hold byte (8-bit) integer, half-word (16-bit) integer, word (32-bit) integer[1], single-precision floating-point and double-precision floating-point values. The notation used to denote registers and memory references in the register transfer notation explicitly indicates their assignment type. For example, the `WTREG` assignment type is used to assign a target machine register to pseudo-register `w[50]` and the `DTREG` assignment type is used to allocate a register for the pseudo-variable accessed by the memory reference `D[w[14]+a]`.

This description contains three register class declarations beginning at lines 2, 16 and 28. The first class describes the general-purpose registers, the second describes the floating-point registers and the last defines a special class that describes the temporary spill variables used to contain values in memory when there are insufficient registers to hold all of the pseudo register values created by the code generator. Each register class represents a separate, non-overlapping storage partition.

Lines 3, 17 and 29 declare the number of basic storage units available within each register class. Any one of these basic units can be reserved so that it is strictly off-limits to the register assigner. For the SPARC, line 4 reserves:

- general-purpose register zero is because it always contains the value zero,
- register 14, which is the frame pointer,
- register 15, which is used to receive the return value of an external function call,
- register 30, which is the argument pointer and
- register 31, which contains the return address.

Lines 5 and 18 indicate which registers belong to the volatile register partition defined by the calling convention for the general-purpose and floating-point register classes, respectively. This information is essential because it ensures that the register assigner will automatically favor the registers that can be used without inserting save and restore code in the function prologue and epilogue. For example, the volatile partition information given for the general-purpose registers in line 5 indicates that register 8 through 13 and then 1 through 7 should be selected for assignment before the remaining allocable integer registers are used. This technique is similar to that used in Site's U-code storage allocator discussed in Section 3.4.2.

[1] The number of bits are given only to clarify the type, this information is not provided by the register description because the register allocation primitives provided by *llef* deal with basic storage units instead of bits.

The assignment type information given for each register class provides the size and the alignment information needed to assign registers and temporary spill locations. The information in lines 6 through 14 indicates that a single general-purpose register can be used to hold any integer-type item regardless of its size (8-, 16- or 32-bits). Lines 19 through 22 state that a single floating-point register can contain a single-precision floating-point value and lines 23 through 26 indicate that an even/odd floating-point register pair is needed to hold a double-precision floating-point value. Line 9 indicates that general-purpose registers 14 and 30, which are the frame and argument pointers, contain word (32-bit) values that remain invariant throughout each function. This information is not needed to perform register allocation, but is provided for the loop-invariant code motion optimization phase.

5.3.2 Constrained references

To avoid the need to perform register allocation during code generation, *llef* allows the code generator to emit code that uses pseudo-registers to hold temporary expression values. Pseudo-registers, however, are not appropriate in instances where a function argument must be passed in a specific target machine register or the architecture contains instructions that implicitly operate on specific registers. The local register allocator in *vpo* is unable to assign registers that might be specifically referenced by the code generator, which are called *constrained references*, because it lacks the ability to determine when the life of a pseudo-register overlaps that of a constrained reference. This section describes the local register allocation algorithm used by *llef*, which is capable of using the full set of allocable registers while allowing the code generator to generate constrained register references on demand. One of the best properties of this mechanism is that it does not produce excessive amounts of transfer code.

The code shown in Figure 30, which is taken from an actual sequence of low-level register transfers produced on the SPARC architecture, are used to describe the local register assignment mechanism. This code sequence passes a single integer parameter to an external function that returns a double-precision, floating-point value. To further illustrate the complexity of the local register allocation task, the code requires the value of a floating-point pseudo-register to be maintained across the external function call despite the fact that all of the floating-point registers are volatile[1].

[1] A common term for this is *caller-saves*, meaning that the caller is responsible for saving and restoring the registers.

```
1              d[32]=D[_sum];
2              w[33]=W[w[14]+a];
3   {2}        w[8]=w[33];                             w[33]
4   {3}        ST=_foo,w[8]                            w[8]
use            w[1]..w[13]
spill          d[0]..d[31]
reserve        d[34]=d[0]
5   {1,4}      d[32]=d[32]+d[34];                      d[34]
6   {5}        D[_sum]=d[32];                          d[32]
```

Figure 30: A call sequence before local register assignment

The local-scope register assigner processes a basic block starting with the first instruction and visiting each successive instruction until the end of the block is reached. The assigner maintains the assignment status information that associates each pseudo-register with the location where its value resides. At the start of the basic block, no associations exist[1]. In instruction 1, pseudo-register `d[32]`[2] has no association, so its assignment type, `DTREG`, is used to find the most desirable floating-point class storage unit available. The allocator first searches over the entire lifetime of the pseudo-register for any constrained register references whose corresponding storage units overlap the storage units within the register class in which the `DTREG` assignment type was defined. The constraints found are a reference to `w[8]` in instruction 3 and to `w[1]` through `w[13]` in the `use` line. Because none of these items affect the storage units in the floating-point register class, they do not constrain the location to which the pseudo-register can be assigned. This search is essential to performing effective register allocation in the presence of constraints, because it prevents the allocator from assigning a pseudo-register value to any register that would guarantee the need to introduce a spill or transfer in the future. Since the size of `DTREG` items is two, storage units zero and one, which correspond to register `d[0]`, are assigned. The assigner associates pseudo-register `d[32]` with `d[0]`, reserves floating-point register class storage units zero and one and updates the first instruction as follows:

```
1              d[0]=D[_sum];
```

In instruction 2, the right-hand side contains a reference to `w[14]` which is left untouched because it is a reserved target machine register. The assignment to `w[33]` is handled like the previous assignment. There is no constraint with w[8] in instruction 3 because the life of the pseudo-register ends before the

---

[1] This is not always true, but it is a valid assumption for this example.
[2] It is a pseudo register because, as the register description in Figure 29 shows, there are only 32 registers (0 through 31) in each register class.

assignment takes place. The `WTREG` assignment type favors storage unit eight,[1] which is available, thus `w[8]` is associated with `w[33]` and instruction 2 is updated:

```
2          w[8]=W[w[14]+a];
```

The right-hand side of instruction 3 references `w[33]`, which is already associated with `w[8]`. The *dead list* associated with this instruction indicates that value `w[33]` will not be used again, so the association of `w[33]` and `w[8]` is removed and general-purpose register class storage unit eight is released. The left-hand side reference, `w[8]`, is a constrained register reference that does not need an association, but the corresponding general-purpose register class storage unit is reserved so that its value is not accidentally overwritten. The assignment for the instruction is now complete:

```
3   {2}    w[8]=w[8];
```

This instruction performs no useful function and will be removed by a subsequent optimization phase.

Instruction 4 represents the external function call and `w[8]` contains the value of the first argument passed to the function in accordance with the calling convention for the SPARC's C library functions. This information is not encoded in the machine-independent assignment algorithm. It was available during code generation and enough information is present in the code to allow the local register assigner to ensure that function argument values reside in the appropriate target machine registers at the point of the call. This illustrates the utility of allowing the code generator to emit constrained register references.

The storage unit corresponding to `w[8]` is released because the dead list associated with instruction 4 indicates that the value in this register is not referenced after the external function call. The `use`, `spill` and `reserve` lines are associated with the external function call and were produced by the code generator to provide the local register assigner with calling convention information. The `use` line indicates that the external function call potentially overwrites the values in registers `w[1]` through `w[13]`. The `spill` line indicates that the values in registers `d[0]` through `d[31]` are overwritten, but, unlike the `use` line, it does not constrain the assignment to these registers because doing so would prevent the assigner from assigning any `DTREG` type pseudo-register that is live across an external function call to a register. For example, if the `spill` line constrained the registers, instruction 1 would not have been able to assign `d[32]` to a register. Because `d[0]` is associated with a pseudo-register value, the assigner must transfer its value to a location

[1] This is indicated by the volatile information given in line 5 of the register description show in Figure 29.

that is not affected by the external function call. Since all of the registers associated with the `DTREG` allocation type are affected by the function call, the assigner reserves storage units zero and one in the spill class. Changing the association of `d[32]` from `d[0]` to the spill location requires new code to be inserted before the external function call:

```
7   {1}      D[w[14]+spill0]=d[0];              d[0]
4   {3}      ST=_foo,w[8]                        w[8]
```

The assigner adds `d[0]` to the dead list of instruction 7 to indicate that its value will not be referenced again. A *combiner link* is inserted from the spill instruction, which now represents the first use of the `d[0]` register, to instruction 1, which assigns the value to the register. A subsequent invocation of the instruction selection phase will attempt to merge these instructions and, even thought the result is not an instruction on the SPARC architecture, the link is added because there are architectures on which the spill would merge with the load in instruction 1. The `reserve` line indicates that the external function call returns a value in register `d[0]` that will subsequently be referenced as pseudo-register `d[33]`. To make this association, the assigner reserves floating-point register class storage units zero and one.

Except for cases where all of the registers in a register class are volatile, spills are extremely rare if the target machine provides three or more registers in each register class. Fraser and Hanson show that, with sufficient registers, even simple spill techniques produce good results [FRAS92]. The sophistication of the spill mechanism used by *llef*'s local scope register assigner is motivated by the desire to generate high quality code on architectures that provide very few registers.

In instruction 5, the right-hand side references `d[32]`, whose value is located in spill storage units zero and one. Before this value can be referenced as a register, it must be moved from its temporary spill location to an appropriate register. The assigner reserves register `d[2]` for this value because the storage units corresponding to `d[0]` are reserved by the value of pseudo-register `d[34]`. The following code, which is needed to transfer the value of the spill location to the assigned register, is inserted before instruction 5:

```
8            d[2]=D[w[14]+spill0];
```

The right-hand side of instruction 5 also references value `d[34]`, which is already associated with `d[0]`. The dead list indicates that `d[34]` will not be used in the future, so floating-point storage units zero and one are released. On the left-hand side, `d[32]` is already associated with `d[2]`. After the assigner is finished with instruction 5, it has been transformed to:

```
5   {4,8}     d[2]=d[2]+d[0];                        d[0]
```

Note that a combiner link has been inserted from instruction 5, which now contains the first use of `d[2]`, to the new instruction which assigns the register.

The last instruction references and releases the `d[32]` value. Floating-point storage units two and three, which were reserved for this value, are released. The assignment process is complete, leaving the code shown in Figure 31. This code, once it is given appropriate function prologue and epilogue code, can be translated into object code and executed. Before doing so, however, *llef*, can optionally perform additional local and global-scope code-improvement transformations on it.

```
1             d[0]=D[_sum];
2             w[8]=W[w[14]+a];
3   {2}       w[8]=w[8];
7   {1}       D[w[14]+spill0]=d[0];                  d[0]
4   {3}       ST=_foo,w[8]                           w[8]
8             d[2]=D[w[14]+spill0];
5   {4,8}     d[2]=d[2]+d[0];                        d[0]
6   {5}       D[_sum]=d[2];                          d[2]
```

Figure 31: A call sequence after local register assignment

5.3.3 Evaluation order determination

Because the local register allocator must satisfy the register requirements of code generation in all circumstances, spill code will be produced when insufficient registers are available on the target machine. These situations also increase the likelihood that inter-block pseudo-register values will migrate from their initial assignment locations and cause the local register assigner to produce inter-block transfer code. Thus, reducing the number of registers required to generate code beneficially impacts the quality of the object code produced by *llef* on architectures with small register sets.

The number of registers required to perform local register assignment on a sequence of code depends on the maximum number of simultaneously live pseudo-register and register values present at any point in the instruction sequence. A well-known technique for minimizing the number of temporary values required to compute an arbitrary arithmetic expression is presented in the Sethi and Ullman text [SETH70]. This technique does not fit well into a low-level framework because it is intended to operate on the expression trees generated by the front-end's semantic analysis phase. The technique, however, was modified by Davidson to work on, PO [DAVI86], and has been further extended to operate within *llef*.

The following expression is used to illustrate the algorithm that determines and produces optimal evaluation orders in *llef*:

```
a = (a + b) * (c * d + e * f)
```

Figure 32 shows the sequence of instructions used to calculate this expression on the SPARC architecture just prior to local register allocation. The algorithm used to determine the optimal evaluation order of an

```
1            w[33]=W[w[14]+a];
2            w[34]=W[w[14]+b];
3   {1,2}    w[33]=w[33]+w[34];                  w[34]
4            w[35]=W[w[14]+c];
5            w[36]=W[w[14]+d];
6   {4,5}    w[35]=w[35]*w[36];                  w[36]
7            w[37]=W[w[14]+e];
8            w[38]=W[w[14]+f];
9   {7,8}    w[37]=w[37]*w[38];                  w[38]
10  {6,9}    w[37]=w[37]+w[35];                  w[35]
11  {3,10}   w[33]=w[33]+w[37];                  w[37]
12  {11}     W[w[14]+a]=w[33];                   w[33]
```

Figure 32: Expression calculation code sequence

expression operates on this low-level representation as if it were an expression tree like the one shown in Figure 33. The nodes of this expression tree are the instructions and the edges represent the dependencies of the pseudo-register values that form the expression. Since these edges correspond to the combiner links that already exist to support instruction selection, no additional data structures are needed to represent the expression trees.

The evaluation order algorithm, shown in Figure 34, operates on each basic block individually by assigning an initial *carry* and *use* value to each instruction. The *carry* value is the number of unique storage units required to contain the values calculated by the instruction and subsequently used by other instructions. The *use* value is the number of storage units other than those needed to contain the values calculated by the subexpression that are used by the instruction. The algorithm calculates storage unit usage instead of the more traditional register usage because storage units more closely reflect the actual register resource requirements of expressions that use multiple register types. Additionally, the algorithm ignores the storage units associated

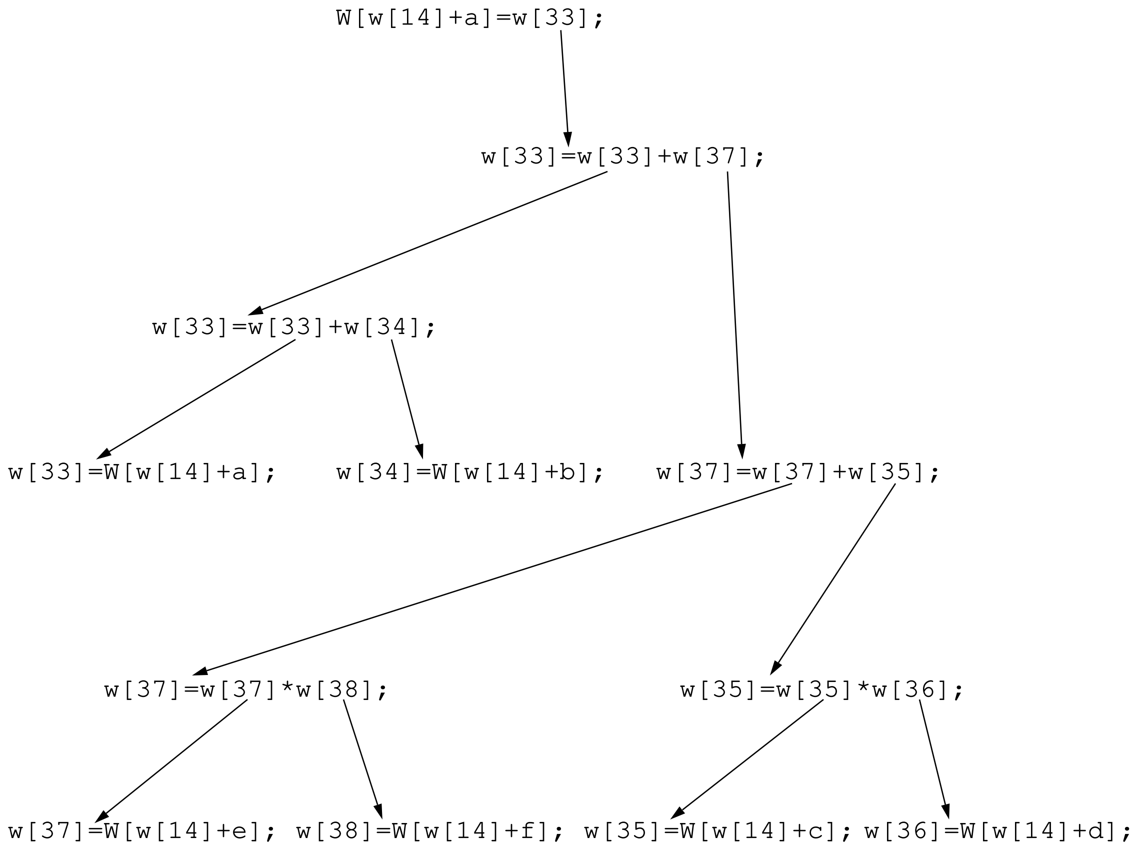


Figure 33: Evaluation order determination expression tree

with the frame, argument or stack pointer registers because the local register allocator keeps these storage units reserved regardless of the evaluation order. Figure 35 shows the initial *carry* and *use* values returned by the *result_storage_units_required* and *additional_storage_units* required functions for each instruction in the expression shown in Figure 32. All of the initial *use* values shown in this example are zero because none of the instructions write to any registers other than the ones that receive the values that are explicitly computed by the instructions.

After the initial *carry* and *use* values are determined for an instruction, the algorithm uses the combiner links associated with it to visit its immediate successors in the expression tree in order to determine how many total storage units are needed to assign the subexpression rooted there. The algorithm sorts the successors according to the value of their *use* field and then, starting with the successor that requires the greatest number of storage units to compute, the algorithm determines how many total storage units will be required to calculate the entire subexpression by adding up the total number of storage units required to hold

```
proc evaluation_order_determination is
    for each B where B ∈ CFG do
        I = B.first_instruction
        while I ≠ ∅ do
            I.carry = result_storage_units_required(I)
            I.use = additional_storage_units_required(I)
            N = 0
            for each L where L ∈ I.combiner_links do
                S_N.carry = L.instruction.carry
                S_N.use = L.instruction.use
                N = N + 1
            endfor
            if N > 0 then
                sort_by_use(S, N)
                C = S_N.carry
                U = S_N.use
                while N > 0 do
                    N = N - 1
                    C = C + S_N.carry
                    U = max(U - S_N.carry, 0)
                    U = max(U, S_N.use)
                endwhile
                I.use = max(I.use, C - I.carry + U)
            endif
            I = I.next_instruction
        endwhile
        I = B.last_instruction
        while I ≠ ∅ do
            if I.number_of_combiner_links > 1 then
                I = reorder_subexpressions(I)
            endif
            I = I.previous_instruction
        endwhile
    endfor
endproc
```

Figure 34: Evaluation order determination algorithm

all of the values referenced by the instruction. Since all of these values will have to be available when the instruction is assigned, this sum indicates how many storage units are needed to hold these values. The order in which subexpressions are visited allows the storage units used by the subexpressions calculated first to be re-used to hold the values of subsequently computed subexpressions, thus occupying the least number of total storage units at each point in the expression. When this process is complete, each instruction reflects the total number of storage units required to assign registers to the instruction sequence that calculates the subexpression in the expression tree that is rooted at that instruction. Figure 36 shows final *carry* and *use* values associated with each instruction at the end of the storage unit requirement determination stage.

Instruction				*carry*	*use*
1		`w[33]=W[w[14]+a];`		1	0
2		`w[34]=W[w[14]+b];`		1	0
3	{1,2}	`w[33]=w[33]+w[34];`	w[34]	1	0
4		`w[35]=W[w[14]+c];`		1	0
5		`w[36]=W[w[14]+d];`		1	0
6	{4,5}	`w[35]=w[35]*w[36];`	w[36]	1	0
7		`w[37]=W[w[14]+e];`		1	0
8		`w[38]=W[w[14]+f];`		1	0
9	{7,8}	`w[37]=w[37]*w[38];`	w[38]	1	0
10	{6,9}	`w[37]=w[37]+w[35];`	w[35]	1	0
11	{3,10}	`w[33]=w[33]+w[37];`	w[37]	1	0
12	{11}	`W[w[14]+a]=w[33];`	w[33]	0	0

Figure 35: Initial *carry* and *use* values

Instruction				*carry*	*use*
1		`w[33]=W[w[14]+a];`		1	0
2		`w[34]=W[w[14]+b];`		1	0
3	{1,2}	`w[33]=w[33]+w[34];`	w[34]	1	1
4		`w[35]=W[w[14]+c];`		1	0
5		`w[36]=W[w[14]+d];`		1	0
6	{4,5}	`w[35]=w[35]*w[36];`	w[36]	1	1
7		`w[37]=W[w[14]+e];`		1	0
8		`w[38]=W[w[14]+f];`		1	0
9	{7,8}	`w[37]=w[37]*w[38];`	w[38]	1	1
10	{6,9}	`w[37]=w[37]+w[35];`	w[35]	1	2
11	{3,10}	`w[33]=w[33]+w[37];`	w[37]	1	2
12	{11}	`W[w[14]+a]=w[33];`	w[33]	0	3

Figure 36: Final *carry* and *use* values

The final part of the evaluation order determination process uses the final *carry* and *use* values to order the subexpressions within each expression so that they use the fewest number of storage units possible during local register assignment. To produce this order, the instructions are rearranged so that the subexpressions requiring the most storage units to compute are evaluated first. The algorithm shown in Figure 34 does this by starting at the bottom of a basic block and moving subexpressions that use the fewest number of storage units closer to the instruction that uses their value. The *reorder_subexpression* function, which is invoked for each instruction that uses two or more subexpression values, is shown in Figure 37. This function orders the subexpressions used by the instruction according to the total number of storage units required to perform local register assignment on them and moves the subexpressions that use the fewest storage units as

```
func reorder_subexpression(I) is
    N = 0
    for each L where L ∈ I.combiner_links do
        S_N.cost = L.instruction.carry + L.instruction.use
        S_N.instruction = L.instruction
        N = N + 1
    endfor
    sort_by_cost(S, N)
    M = 0
    while M < N do
        I = move_closer(I, S_M.instruction)
        M = M + 1
    endwhile
    return I
endfunc
```

Figure 37: The *reorder_subexpression* function

close to the instruction as possible. This process has the desired effect of rearranging the code so that the subexpressions that need the most storage units are calculated first. The *move_closer* function, which moves subexpressions closer to the instruction that uses the value that they compute, is shown in Figure 38. This function first ensures that the subexpression resides in the same basic block that its value is used. This may

```
func move_closer(I, S) is
    if S.basic_block ≠ I.basic_block then
        return I
    endif
    if S.next_instruction ≠ I then
        if S.prev_instruction = ∅ then
            S.basic_block.first_instruction = S.next_instruction
        else
            S.prev_instruction.next_instruction = S.next_instruction
        endif
        S.next_instruction.prev_instruction = S.prev_instruction
        S.prev_instruction = I.prev_instruction
        S.next_instruction = I
        I.prev_instruction = S
        S.prev_instruction.next_instruction = S
    endif
    return reorder_subexpression(S)
endfunc
```

Figure 38: The *move_closer* function

not always be true in the presence of global combiner links. The function does not move code across basic blocks. Doing so without inadvertently changing the semantics of the code entails an extensive verification algorithm. Because there are so few actual opportunities to apply this type of transformation, the benefit of implementing the verification algorithm was not sufficient enough to justify the cost. If the subexpression is computed in the same basic block that its value is used, *move_closer* removes the last instruction in the subexpression from its location and places it immediately before the instruction indicated. The function then calls *reorder_subexpression* recursively on the newly-ordered subexpression to obtain an optimal ordering of the subexpression values that it uses before ordering the rest of the subexpressions whose values are used by the original instruction. The effects of applying this evaluation order determination algorithm on the code shown in Figure 32 are shown in Figure 39.

```
4            w[35]=W[w[14]+c];
5            w[36]=W[w[14]+d];
6  {4,5}     w[35]=w[35]*w[36];                 w[36]
7            w[37]=W[w[14]+e];
8            w[38]=W[w[14]+f];
9  {7,8}     w[37]=w[37]*w[38];                 w[38]
10 {6,9}     w[37]=w[37]+w[35];                 w[35]
1            w[33]=W[w[14]+a];
2            w[34]=W[w[14]+b];
3  {1,2}     w[33]=w[33]+w[34];                 w[34]
11 {3,10}    w[33]=w[33]+w[37];                 w[37]
12 {11}      W[w[14]+a]=w[33];                  w[33]
```

Figure 39: Expression code sequence after evaluation order determination

As Figure 40(a) shows, the local register assigner requires a minimum of four `WTREG` assignment type registers to assign the initial code sequence without spill code. In contrast, the ordered code sequence can be assigned with only three registers as shown in Figure 40(b). By producing code sequences that require fewer registers to allocate, ordering improves the quality of the code in two ways. First, ordering the code may reduce or eliminate the spill code produced by the local register allocator. Second, any registers that are saved as a result of ordering are used to perform other code improvement optimizations, thus increasing the overall quality of the code. In addition, the ordering phase is architecture-independent and highly retargetable because the register information that it uses is provided exclusively by the functions that are automatically constructed from the register description.

5.4 Variable promotion

The purpose of variable promotion is to allocate user-defined local variables to registers. A global-scope, graph coloring approach, similar to the PL.8 approach in that an interference graph is built to drive the coloring process, is used to perform this task. Unlike the PL.8 assigner, however, *llef* uses the interference graph to perform discretionary register allocation instead of mandatory register assignment. Because the

```
1           w[0]=W[w[14]+a];
2           w[1]=W[w[14]+b];
3  {1,2}    w[0]=w[0]+w[1];                     w[1]
4           w[1]=W[w[14]+c];
5           w[2]=W[w[14]+d];
6  {4,5}    w[1]=w[1]*w[2];                     w[2]
7           w[2]=W[w[14]+e];
8           w[3]=W[w[14]+f];
9  {7,8}    w[2]=w[2]*w[3];                     w[3]
10 {6,9}    w[2]=w[2]+w[1];                     w[1]
11 {3,10}   w[0]=w[0]+w[2];                     w[2]
12 {11}     W[w[14]+a]=w[0];                    w[0]
```

(a)

```
4           w[0]=W[w[14]+c];
5           w[1]=W[w[14]+d];
6  {4,5}    w[0]=w[0]*w[1];                     w[1]
7           w[1]=W[w[14]+e];
8           w[2]=W[w[14]+f];
9  {7,8}    w[1]=w[1]*w[2];                     w[2]
10 {6,9}    w[1]=w[1]+w[0];                     w[0]
1           w[2]=W[w[14]+a];
2           w[0]=W[w[14]+b];
3  {1,2}    w[2]=w[2]+w[0];                     w[0]
11 {3,10}   w[2]=w[2]+w[1];                     w[1]
12 {11}     W[w[14]+a]=w[2];                    w[2]
```

(b)

Figure 40: Local register allocation before and after evaluation order determination

coloring allocator works with pseudo-variables, forced spill code is never created and only as much allocation is done as the registers that are not used by the code generator or by other optimizations allow. This strategy prevents over-allocation even when very few registers are available on the target machine. The following sections will present the algorithms that build and use the interference graph.

5.4.1 Local interference graph

The interference graph is constructed by the local-scope algorithm shown in Figure 41. Before processing a basic block, the algorithm uses the live-variable dataflow information to determine which unaliased local user variables are live at the entry point of the block. A local interference graph node is created for each of these items and added to the list of nodes live on entry to the basic block. These nodes are made to interfere with each other to ensure that they are not assigned to the same register. They are also made to interfere with the set of registers live at the start of the basic block to ensure that they are not assigned to these registers.

```
proc build_local_interference_graph_nodes(B) is
    R = B.values_live_on_entry ∩ REGISTERS
    for each V where V ∈ (B.values_live_on_entry ∩ UNALIASED_LOCAL_VARIABLES) do
        V.current_node = new_interference_node(B, V)
        V.current_node.live_at_block_entry = TRUE
        V.current_node.register_conflicts = R
        B.nodes_live_on_entry = B.nodes_live_on_entry ∪ V.current_node
    endfor
    for each N where N ∈ B.nodes_live_on_entry do
        N.node_conflicts = B.nodes_live_on_entry - N
    endfor
    B.nodes_live_on_exit = B.nodes_live_on_entry
    I = B.first_instruction
    while I ≠ ∅ do
        for each V where V ∈ (I.used ∩ UNALIASED_LOCAL_VARIABLES) do
            V.current_node.used_first = V.current_node.used_first ∨ ¬V.current_node.is_set
            V.current_node.benefit = V.current_node.benefit + scaled_load_benefit(B)
        endfor
        for each V where V ∈ I.dead_list do
            B.nodes_live_on_exit = B.nodes_live_on_exit - V.current_node
            V.current_node.range_end = I
            V.current_node = ∅
            R = R - V
        endfor
        for each V where V ∈ I.assigned do
            if V ∈ UNALIASED_LOCAL_VARIABLES then
                if V.current_node = ∅ then
                    V.current_node = new_interference_node(B, V)
                    V.current_node.range_start = I
                    V.current_node.register_conflicts = R
                    V.current_node.node_conflicts = B.nodes_live_on_exit
                    for each M where M ∈ B.nodes_live_on_exit do
                        M.node_conflicts = M.node_conflicts ∪ V.current_node
                    endfor
                    B.nodes_live_on_exit = B.nodes_live_on_exit ∪ N
                endif
                V.current_node.is_set = TRUE
                V.current_node.benefit = V.current_node.benefit + scaled_store_benefit(B)
            else if V ∈ REGISTERS then
                R = R ∪ V
                for each N where N ∈ B.nodes_live_on_exit do
                    N.register_conflicts = N.register_conflicts ∪ V
                endfor
            endif
        endfor
        I = I.next_instruction
    endwhile
    for each V where V ∈ B.nodes_live_on_exit do
        V.current_value.live_at_block_exit = TRUE
        V.current_value = ∅
    endfor
endproc
```

Figure 41: Local interference graph construction algorithm

The algorithm copies the list of nodes live on entry to the basic block to the list of items live at the exit point. These two lists are obviously identical for empty basic blocks. If the basic block contains instructions, node items will be added and removed from the exit list as the algorithm progresses through the basic block so that, after the last instruction is processed, the exit list contains only nodes for the items live at the exit point of the block.

An interference graph node contains either a pointer to the instruction where the life of a variable begins or an indication that its life extends beyond the entry point of the basic block. Similarly, there is either a reference to the last instruction in the life range, or an indication that the range extends beyond the exit point of the basic block. To facilitate the coloring process, each node contains a list of the local user variables and registers whose lives overlap the life of the node at some point in the basic block. A benefit value, which is the savings anticipated from replacing the pseudo-variable with a register scaled by the loop nesting level of the basic block, is calculated for each node and used to prioritize the coloring process. Note that the functions called to determine these benefit values, *scaled_load_benefit* and *scaled_store_benefit*, are underlined to indicate that they are architecture-specific and must be examined while retargeting *llef*. This notation applies to all of the algorithms presented here.

Figure 42 shows the local interference graph nodes created for a simple basic block. Nodes 1 and 2 are created for variables `a` and `b` and placed in the list of nodes live on entry to the block because their lives start outside of the basic block. These nodes conflict with each other and with the frame and argument pointer registers, `w[14]` and `w[30]`, whose lives pass through the basic block. The benefit value assigned to these nodes is α, which represents the estimated savings of replacing a variable load with a register reference scaled by the loop nesting depth of the basic block. Node 2 also conflicts with register `w[8]` because the life of this register that starts at instruction 1 overlaps the life of variable `b`, which extends to instruction 2. Node 3 is created because a new life of variable `a` begins at instruction 4 and extends to instruction 5. Because this node represents a life of variable `a` completely separate from the life denoted by node 1, these nodes may be assigned to different registers. Node 3 has a benefit of α, which is contributed by the load, plus β, which is the corresponding scaled benefit value for the store. Node 4 represents the life of variable `c` which starts at instruction 7. The node is a member of the list of nodes live on exit because the life of the variable extends beyond the end of the basic block.

label: L49
values_live_on_entry: a,b,w[14],w[30]
values_live_on_exit: b,w[14],w[30]
nodes_live_on_entry: 1, 2
nodes_live_on_exit: 4

```
1          w[8]=W[w[14]+a];      a
2          w[9]=W[w[30]+b];      b
3 {1,2}    w[8]=w[8]+w[9];       w[9]
4 {3}      W[w[14]+a]=w[8];      w[8]
5          w[8]=W[w[14]+a];      a
6 {5}      w[8]=w[8]*24;
7 {6}      W[w[14]+c]=w[8];      w[8]
8          PC=L50;
```

node_number: 1
variable: a
block: L49
live_at_block_entry: TRUE
live_at_block_exit: FALSE
used_first: FALSE
is_set: FALSE
range_start: ∅
range_end: 1
register_conflicts: w[14],w[30]
node_conflicts: 2
benefit: α

node_number: 2
variable: b
block: L49
live_at_block_entry: TRUE
live_at_block_exit: FALSE
used_first: FALSE
is_set: TRUE
range_start: ∅
range_end: 2
register_conflicts: w[8],w[14],w[30]
node_conflicts: 1
benefit: α

node_number: 3
variable: a
block: L49
live_at_block_entry: FALSE
live_at_block_exit: FALSE
used_first: FALSE
is_set: TRUE
range_start: 4
range_end: 5
register_conflicts: w[14],w[30]
node_conflicts: ∅
benefit: $\alpha+\beta$

node_number: 4
variable: c
block: L49
live_at_block_entry: FALSE
live_at_block_exit: TRUE
used_first: FALSE
is_set: TRUE
range_start: 7
range_end: ∅
register_conflicts: w[14],w[30]
node_conflicts: ∅
benefit: β

Figure 42: Basic block and its associated local interference graph nodes

5.4.2 Live range reduction

Using live variable information to construct the local interference graph extends the life of local user variables that may be used before they are assigned to the entry point of the function. While this is usually true for function arguments, which are considered by *llef* to be a subset of the local user variables, there is no need to start the life of most arguments at the entry block, because the argument's value can be obtained from a memory location[1] at any point in the code. If the life starts at the entry block but is not used until the end of

[1] One notable exception are arguments which are passed to the function in a register. In such cases, the argument's value must be obtained from the register before the register is used for other purposes.

the function, a register might needlessly be reserved to contain the value of the argument over a large portion of the function.

llef uses an approach is similar to Chow's shrink wrapping technique [CHOW88] to determine the where the value of the argument can be loaded in order to reduce the lifetime of the argument as much as possible. The locations chosen by this algorithm ensure that every path from the entry block to an initial use of the argument value contains a load of the value. The local graph nodes associated with the argument are elided from the basic blocks where the argument does not have to be in a register. The graph nodes in the basic blocks where the argument value must be loaded are marked so that an appropriate load is created if the argument is assigned to a register.

An example of this process is taken from the local interference information in Figure 42. Variable `b` is an argument[1] whose value extends to the entry point of the function. The location of block L49 in the CFG is shown in Figure 43. The basic blocks in every path from the function entry block to L49 contain a node for

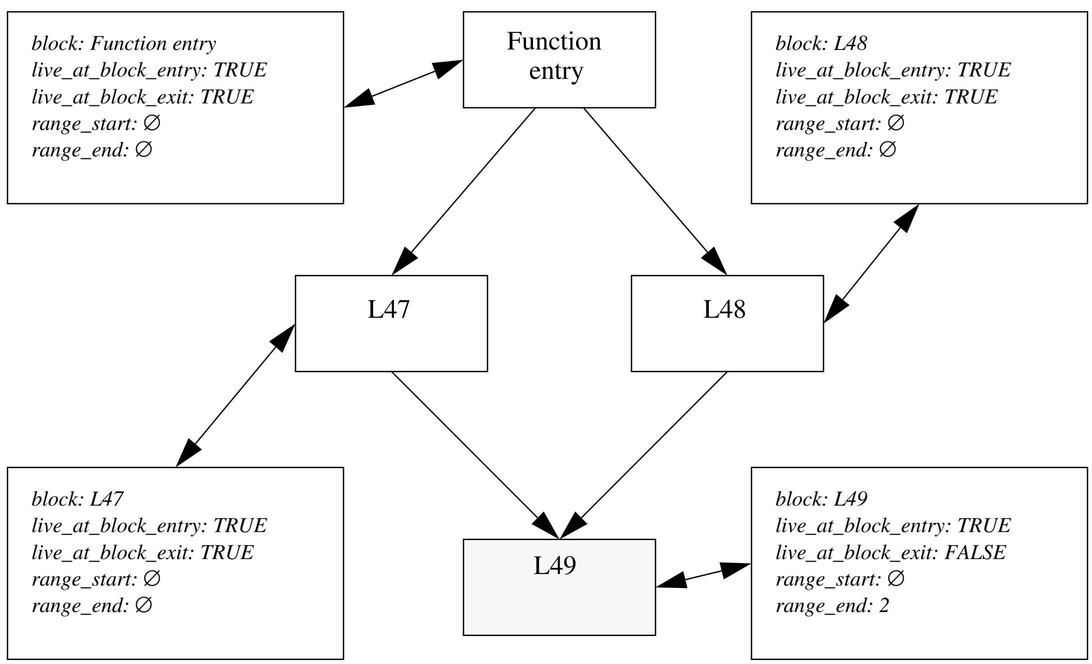


Figure 43: CFG and the local interference graph nodes for argument `b` before live range reduction

variable `b` which indicates that the value passes through the block. These nodes may contain conflicts with other variables and registers to prevent the argument from being assigned to any of the registers live in these

[1] Note that the code which references this variable uses the argument pointer register, `w[30]`, to form the address expression.

blocks. The live range reduction process determines that the value of `b` can be loaded directly from the caller's activation record in L49 and removes the nodes associated with the argument in the function entry block and in blocks L47 and L48. The node in block L49 is modified to indicate that the life of the argument starts at instruction 2 and the "*must_load*" field is set to indicate that the argument must be loaded before it is used if the node is assigned to a register. The node is also removed from the list of nodes live on entry to block L49.

5.4.3 Coloring

Prior to coloring, the algorithm shown in Figure 44 connects all of the local interference nodes associated with the same variable life together. The local interference graph nodes are then sorted in descending order based on the benefit value of the node. This ordering increases the probability that registers are assigned to the variables estimated to be referenced most frequently.

```
proc merge_local_interference_graph_nodes is
    for each B where B ∈ CFG do
        for each N where N ∈ B.nodes_live_on_exit do
            for each S where S ∈ B.successors ∧ S ≠ B do
                for each M where M ∈ S.nodes_live_on_entry ∧ M.variable = N.variable do
                    N.siblings = N.siblings ∪ M
                    M.siblings = M.siblings ∪ N
                endfor
            endfor
        endfor
    endfor
endproc
```

Figure 44: Graph node merge algorithm

The coloring process uses two different approaches to assign registers to the ordered list of interference graph nodes. The first approach, shown in Figure 45, performs global assignment by determining if there is any register that does not interfere with the entire lifetime of the variable. Given a local interference graph node for a variable, the global algorithm visits each local node comprising the global lifetime of the variable and determines the assignment type used to reference the variable, the sum of the benefit estimates, the set of registers that conflict with the variable and the cost of any load instructions that must be inserted if the variable is assigned. The assignment type is needed for two purposes: first, to allocate an appropriate register to hold the value of the variable and, second, to ensure that inconsistently referenced variables are not promoted to registers[1]. Adding the benefit estimates and the cost penalties for inserting load instructions ensure that the overhead costs of promoting the variable to a register are defrayed by the anticipated improvements. The assigner calls an external function to determine the overhead costs because they are machine-dependent. If the variable is referenced consistently, then an attempt is made to find a target machine register of the appropriate type that does not conflict with the variable. If such a register exists, the overhead costs of using the register[1] plus the cost of any other load instructions that have to be inserted are compared against the benefits of assigning the variable. If the benefits of assigning the variable outweigh the penalties, then the references to the variable are replaced with references to the assigned register. To ensure that no overlapping variables are assigned to the same register, it is added to the conflict list of all of the local nodes that conflict with the variable.

If a variable is not assigned to a register by the global coloring algorithm, the assigner will separately invoke a local coloring algorithm on each of the local nodes associated with the variable life. The benefit order determines which nodes are considered first by the local coloring algorithm shown in Figure 46. This algorithm differs from its global counterpart in that only the local lifetime of the variable is considered for assignment. Assignment penalties have to consider the cost of the instructions needed to load the variable from its memory location if it is used before it is assigned a value in the basic block and to update the memory location if an assignment is made to the variable in the basic block. These costs are added to the general overhead costs used to determine if it is worthwhile to assign a register for the lifetime of the variable in the basic block. The load and store instructions inserted by the local coloring assigner is transfer code produced by intentionally splitting the life of a variable after ensuring that the benefits outweigh the penalties. These are not forced spills caused by over-optimization.

The sample local interference nodes from Figure 42 can be used to illustrate the coloring process. Figure 47 shows the local interference graph nodes for block L49 along with the remaining nodes from the global interference graph. Nodes 1, 5 and 6 are siblings because they form the entire global life of the variable `a`, and nodes 4 and 7 are siblings representing the life of the variable `c`. These nodes are shown after having been ordered based on their benefit values.

---

[1] The code generator creates such a variable to convert integer values to floating-point on architectures that do not support register-to-register transfers across these types. This variable cannot be assigned to either an integer or a floating-point register. The user can also create this effect using a **union** type variable in C.

[1] The overhead cost of using a register is the cost of any load and store instructions that may be placed in the procedure's prologue and epilogue code to maintain non-volatile register values.

```
func coloring_global_register_assignment(N) is
    C = FALSE
    if ¬N.considered then
        N.considered = TRUE
        T = ∅
        for each I where I ∈ instructions_in_range(N.range_start, N.range_end) do
            T = assignment_type_compatibility(T, type_of_reference(N.variable, I))
        endfor
        B = N.benefit
        R = N.register_conflicts
        P = 0
        if N.must_load then
            P = P + scaled_load_cost(N.block)
        endif
        for each M where M ∈ N.siblings do
            M.considered = TRUE
            for each I where I ∈ instructions_in_range(M.range_start, M.range_end) do
                T = assignment_type_compatibility(T, type_of_reference(M.variable, I))
            endfor
            B = B + M.benefit
            R = R ∪ M.register_conflicts
            if M.must_load then
                P = P + scaled_load_cost(M.block)
            endif
        endfor
        if T ≠ NO_TYPE then
            A = assign_target_register(T, R)
            if A ≠ ∅ ∧ B > P + overhead_costs(A) then
                if N.must_load then
                    insert_variable_load(N.range_start, A, N.variable)
                endif
                transform_local_user_variable(N, A)
                N.colored = TRUE
                for each U where U ∈ N.node_conflicts do
                    U.register_conflicts = U.register_conflicts ∪ A
                endfor
                for each M where M ∈ N.siblings do
                    if M.must_load then
                        insert_variable_load(M.range_start, A, M.variable)
                    endif
                    transform_local_user_variable(M, A)
                    M.colored = TRUE
                    for each U where U ∈ M.node_conflicts do
                        U.register_conflicts = U.register_conflicts ∪ A
                    endfor
                endfor
                C = TRUE
            endif
        endif
    endif
    return C
endfunc
```

Figure 45: Global coloring assignment algorithm

```
func coloring_local_register_assignment(N) is
    C = FALSE
    if ¬N.colored then
        T = ∅
        for each I where I ∈ instructions_in_range(N.range_start, N.range_end) do
            T = assignment_type_compatibility(T, type_of_reference(N.variable, I))
        endfor
        if T ≠ NO_TYPE then
            A = assign_target_register(T, N.register_conflicts)
            if A ≠ ∅ then
                P = 0
                if N.used_first then
                    P = P + scaled_load_cost(N.block)
                endif
                if N.is_set then
                    P = P + scaled_store_cost(N.block)
                endif
                if N.benefit > P + overhead_costs(A) then
                    if N.used_first then
                        insert_variable_load(N.range_start, A, N.variable)
                    endif
                    transform_local_user_variable(N, A)
                    if N.is_set then
                        insert_variable_store(N.range_end, N.variable, A)
                    endif
                    N.colored = TRUE
                    for each U where U ∈ N.node_conflicts do
                        U.register_conflicts = U.register_conflicts ∪ A
                    endfor
                    C = TRUE
                endif
            endif
        endif
    endif
    return C
endfunc
```

Figure 46: Local coloring assignment algorithm

Node 7 has the largest benefit value, so it is selected for assignment first. The global coloring assigner determines that the global life of this variable interferes with registers `w[8]`, `w[14]` and `w[30]` and that it consistently references the variable using the `WTREG` allocation type. The volatile register `w[9]` is available for this variable and since the total benefit of the variable, $2\alpha+\beta$, is greater than the cost of using the register, the assignment is made. Since no nodes conflict with the life of this variable, no register conflicts are added to any of the other nodes.

The process is repeated for node 3, which is next in the ordered list. This node has no siblings, but it is considered by the global coloring algorithm because it represents an entire pseudo-variable life. Register

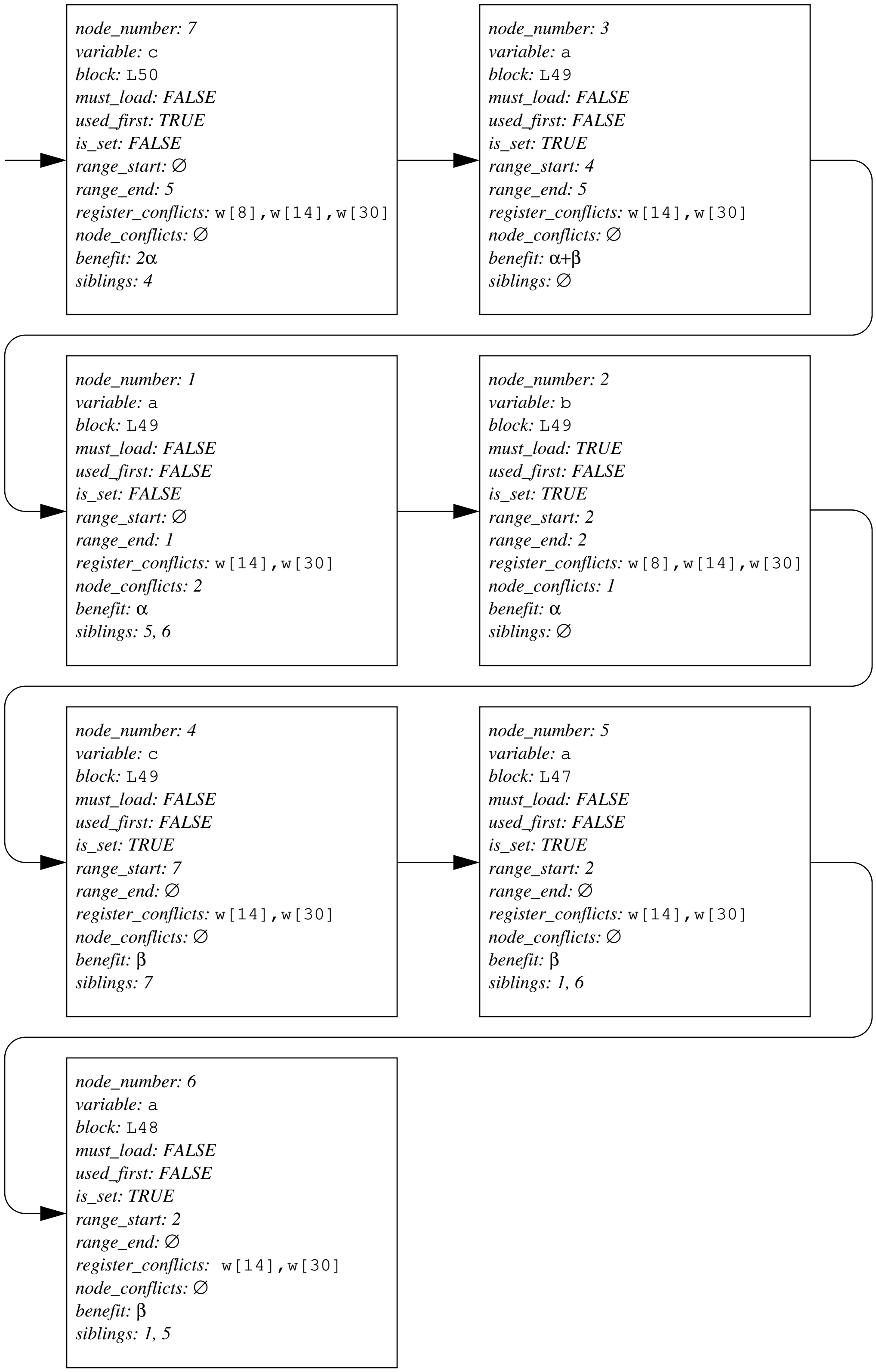


Figure 47: Global interference graph nodes

w[8] is assigned to this pseudo-variable. Node 1 is considered next by the global algorithm along with siblings 5 and 6. The assignment type of this item is WTREG and the total benefit is α+2β, so register w[8] is assigned to this variable. The next node in the list, node 7, is considered next. Because the "*must_load*" field of this node is set, the cost of a load instruction is added to the penalty value for this node. Despite the fact that a valid register can be assigned to the variable, the benefit estimate does not exceed the penalty, so the variable is not assigned to a register by either the global or the local algorithms. The remaining nodes are skipped because they were colored through their association to earlier nodes in the list.

After the coloring process is finished, basic block L49 contains the code shown in Figure 48. A subsequent instruction selection pass will then improve the code so that instructions 1, 4, 5 and 7 are removed. Although no special effort is made to ensure that locals are assigned to the registers that will most reduce the number of assignments required, the tendency to favor the same registers used by the local assigner creates very few extraneous register-to-register transfers.

```
1           w[8]=w[8];
2           w[9]=W[w[30]+b];              b
3   {1,2}   w[8]=w[8]+w[9];               w[9]
4   {3}     w[8]=w[8];
5           w[8]=w[8];
6   {5}     w[8]=w[8]*24;
7   {6}     w[8]=w[8];
8           PC=L50;
```

Figure 48: Code after coloring process

5.5 Loop optimizations

Loops are the focus of numerous effective optimizations because even a small improvement to a frequently-executed loop body can result in a large improvement in the execution time of the program. Consequently, some loop optimizations will insert substantial amounts of code outside of a loop to support a small improvement inside the loop. This is a common strategy even though it can sometimes backfire and increase the execution time of the code. The following sections introduce the loop optimizations performed by *llef*. While the underlying algorithms used to perform these transformations are well-known, the algorithms presented here are unique because they are both highly-retargetable and able to operate directly on low-level code.

5.5.1 Loop-invariant code motion

Any expression computed inside a loop that does not depend on any item changed by the loop is *loop-invariant* and can be moved so that it is only computed once prior to entering the loop. This transformation is called loop-invariant code motion and *llef* is unique in that it applies these transformations after having performed local register assignment and at least one invocation of variable promotion and common subexpression elimination. Loop-invariant code motion directly invokes the loop strength reduction, recurrence optimization and induction variable elimination phases, which use the loop-invariance and register usage information provided by the code motion phase.

In *llef*, loop-invariant code motion is performed individually on each loop starting with the most deeply nested loops in the code and proceeding outward to the outermost loops. This strategy increases the probability that registers will be used to perform the transformations that yield the most benefits. Ordering code motion based on loop nesting level also exploits the fact that code moved out of an inner loop is often loop-invariant in the enclosing loop and can be moved repeatedly until it eventually propagates entirely outside the loops.

Figure 49 shows the algorithm that determines which expressions are loop-invariant. It is similar to the algorithm presented by Aho, Sethi and Ullman [AHO86] to find loop-invariant expressions in high-level intermediate code. This algorithm examines the loop whose information structure is passed to it as an argument and locates the two different types of loop-invariant registers which may be exist in a loop. The first type of loop-invariant register is live on entry to the loop and has no assignments within the loop. The second type of loop-invariant register item is not live on entry to the loop, is assigned only once within the loop and the value assigned to it is made up entirely of literal constants and register or memory items that are themselves loop-invariant. The algorithm performs multiple passes over the loop in order to determine the invariance of registers that are assigned expression that reference registers whose invariance has not yet been determined. It ensures that all such cases are detected by iterating through the loop's code as long as it continues to discover new loop-invariant registers at the end of each pass.

The algorithm uses a simple heuristic to determine if memory references are loop-invariant. If none of the instructions within the loop write to any memory locations, then all of the memory reads in the loop are considered loop-invariant. Otherwise, all memory references are treated as if they were modified within

```
proc loop_invariant_items(L) is
    L.available_registers = REGISTERS
    L.memory_writes = FALSE
    for each R where R ∈ REGISTERS do
        if R.always_invariant then
            R.loop_invariant = YES
            L.available_registers = L.available_registers - R
        else if R ∈ L.entry_block then
            R.loop_invariant = MAYBE
            L.available_registers = L.available_registers - R
        else
            R.loop_invariant = UNKNOWN
        endif
        R.invariant_value = ⊥
    endfor
    F = TRUE
    repeat
        C = FALSE
        for each B where B ∈ L.blocks_in_loop do
            I = B.first_instruction
            while I ≠ ∅ do
                if ¬ L.memory_writes then
                    L.memory_writes = writes_to_memory(I)
                endif
                for each R where R ∈ I.assigned do
                    L.available_registers = L.available_registers - R
                    if R.loop_invariant = MAYBE then
                        R.loop_invariant = NEVER
                    else if R.not_loop_invariant = UNKNOWN then
                        if is_invariant_expression(I, R, L.memory_writes ∨ F) then
                            R.loop_invariant = MAYBE
                        else
                            R.loop_invariant = NO
                        endif
                        R.invariant_value = I
                    endif
                endfor
                I = I.next_instruction
            endwhile
        endfor
        for each R where R ∈ REGISTERS do
            if R.loop_invariant = MAYBE then
                R.loop_invariant = YES
                C = TRUE
            else if R.loop_invariant = NO then
                R.loop_invariant = UNKNOWN
            endif
        endfor
        F = FALSE
    until ¬C
endproc
```

Figure 49: Loop-invariance determination algorithm

the loop. The algorithm also determines which registers are live at any point in the loop so that new registers can be assigned to the variables created by register-consuming loop optimizations.

The sequence of low-level code shown in Figure 50 is used to illustrate the loop-invariant code motion algorithm. Registers `w[1]` and `w[2]` are live at the entry point of the loop and are marked by the loop-invariance detection algorithm as possible loop-invariant items with their "*invariant_value*" field is set to ⊥

```
   L50:
1           w[1]=w[1]<<1;
2           w[8]=HI[_global];
3  {2}      w[9]=w[8]+LO[_global];              w[8]
4  {3}      w[10]=W[w[9]];                      w[9]
5           w[2]=w[2]+1;
6  {4,5}    CC=w[2]?w[10];                      w[10]
7           PC=CC<0,L50;
```

Figure 50: Loop containing loop-invariant register assignments

to indicate that the instruction containing the assignment to these registers is outside the loop. The frame and argument pointer registers, `w[14]` and `w[30]`, are marked loop-invariant even though they are not used in this code sequence because the register description (see Section 5.3.1) indicates that they are always invariant. All of these registers are added to the register assignment set.

The assignment to `w[1]` in instruction 1 gives the register more than one possible value within the loop, so it is not loop-invariant. The expression assigned to `w[8]` in instruction 2 is loop-invariant[1], so the register is marked as a possible loop-invariant item. Instruction 3 contains the only assignment to register `w[9]`, but the expression assigned to it is not considered loop-invariant because `w[8]` has not yet been identified as a loop-invariant register, so `w[9]` is identified as probably not loop-invariant. The same process takes place with `w[10]` in instruction 4. Like `w[1]`, `w[2]` is found to not be loop-invariant in instruction 5.

Because `w[8]` remains possibly loop-invariant at the end of the first pass, it is identified as a loop-invariant register. This triggers a second pass through the loop. Prior to the second pass, the algorithm resets the loop-invariant status of registers `w[9]` and `w[10]`. On the second pass, the expression in instruction 3 is loop-invariant and register `w[9]` is identified as possibly loop-invariant. The expression in instruction 4, however, is not loop-invariant because `w[9]` is not known to be loop-invariant.

[1] Constant values, including local and global identifiers, are always loop-invariant.

After the second pass, `w[9]` is marked loop-invariant and a third pass is initiated. The status of register `w[10]` is set once more to its initial state. The expression in instruction 4 is found to be loop-invariant in the third pass because `w[9]` is loop-invariant and the memory reference is loop-invariant because none of the instructions in the loop modify any memory locations. At the end of the third pass, `w[10]` is marked loop-invariant. A fourth pass through the loop uncovers no new loop-invariant registers and the detection process ends. On exit, the register allocation set contains `w[1]`, `w[2]`, `w[8]`, `w[9]`, `w[10]`, `w[14]` and `w[30]`.

Loop-invariant register assignments can be moved out of the loop without changing the semantics of the loop. If any such assignments exist, a preheader block for the loop is inserted into the CFG[1] and all loop-invariant register assignments are removed from the loop and added to the preheader block. This process does not require new registers because, by definition, the life of a loop-invariant register can be safely extended to cover the entire loop and the end of the preheader block. Figure 51 shows the results of performing loop-invariant code motion on the loop shown in Figure 50.

```
2            w[8]=HI[_global];
3   {2}      w[9]=w[8]+LO[_global];              w[8]
4   {3}      w[10]=W[w[9]];                      w[9]
    L50:
1            w[1]=w[1]<<1;
5            w[2]=w[2]+1;
6   {4}      CC=w[2]?w[10];
7            PC=CC<0,L50;
```

Figure 51: Loop after loop-invariant code motion

5.5.2 Loop strength reduction

A *basic induction variable* is an item that is only incremented or decremented by a constant value within a loop. Loop strength reduction transforms *induced* expressions of the form *scale·i+displacement*, where *i* is the basic induction variable whose *family* the expression is part of and *scale* and *displacement* are loop-invariant values[2], by allocating a new register to hold a new induction variable and inserting code to compute the initial value of the induced expression in the loop's preheader. Whenever the basic induction variable is incremented by x[3], the value of the induced expression changes from *scale·i+displacement* to

---

[1] In many cases, a block that qualifies as a preheader will already exist in the CFG, and a new basic block will not have to be created.

[2] In [AHO86] these values are called *cee* and *dee*, respectively. The terms *scale* and *displacement* are used here because they are more descriptive.

[3] Note that x can also be a negative value.

scale·(*i*+*x*)+*displacement*, which is equivalent to adding the constant value *scale*·*x* to the new induction variable. Once this is done, each reference to the induced expression value is replaced with a reference to the new induction variable and the instructions in the loop that calculated the induced expression can be removed. Collectively, these transformations are called loop strength reduction because they replace relatively expensive multiplication instructions with cheaper addition or subtraction operations.

In *llef*, loop strength reduction is performed on the low-level representation of the code using the loop-invariant expression and register allocation information provided by the loop-invariance determination algorithm. The low-level representation is particularly amenable to loop strength reduction because array reference expressions, which are fully exposed at the low-level, provide many opportunities to perform loop strength reduction and because special architectural features, such as index, auto-increment and auto-decrement addressing modes, can be fully exploited at the low-level to perform more effective strength reduction transformations. The loop shown in Figure 52 is used to illustrate how the loop strength reduction algorithm operates on low-level code.

```
1               w[1]=0;
2               w[9]=w[14]+a;
   L32:
3               w[8]=w[1]*4;
4  {3}           W[w[8]+w[9]]=0;                 w[8]
5               w[1]=w[1]+1;
6               CC=w[1]?100;
7               PC=CC<0,L32;
```

Figure 52: Loop containing an induced expression

The first step in performing loop strength reduction is to find all of the basic induction variables in the loop. The algorithm used to locate the basic induction variables in a loop is shown in Figure 53. Given a loop, this algorithm visits each instruction in the loop and determines which registers are incremented by constant amounts within the loop. Upon exit, the "*induction*" field of the information data structure associated with each register is set to *BASIC* if the register is a basic induction variable in the loop. The algorithm uses a single architecture-specific function, *is_basic_induction*, which examines an instruction, determines if the instruction increments the indicated register by a constant amount and returns a structure containing the increment value if it does. A list of the instructions in which the basic induction variable is incremented is kept so that they can be revisited by the transformation algorithm when it inserts new induction variables to

```
proc find_basic_induction_variables(L) is
    for each R where R ∈ REGISTERS do
        R.induction = UNKNOWN
        R.increment = 0
        R.references = ∅
        R.increments = ∅
    endfor
    for each B where B ∈ L.blocks_in_loop do
        I = B.first_instruction
        while I ≠ ∅ do
            for each R where R ∈ I.used ∧ R.induction ≠ NOT do
                R.references = R.references ∪ I
            endfor
            for each R where R ∈ I.assigned ∧ R.induction ≠ NOT do
                R.references = R.references ∪ I
                N = is_basic_induction(I, R)
                if N ≠ ∅ then
                    R.induction = BASIC
                    R.increment = R.increment + N.increment
                    R.increments = R.increments ∪ I
                else
                    R.induction = NOT
                endif
            endfor
            I = I.next_instruction
        endwhile
    endfor
    for each R where R ∈ REGISTERS ∧ R.induction ≠ BASIC do
        R.induction = UNKNOWN
        R.references = ∅
    endfor
endproc
```

Figure 53: Basic induction variable detection algorithm

perform loop strength reduction. When this algorithm is invoked on the low-level code shown in Figure 52, register `w[1]` will be recognized as a basic induction variable because it is incremented only by a constant value in instruction 5.

The second step in the loop strength reduction process is to find regular induction variables. These are registers that are assigned only an induced expression value within a loop. The algorithm used to detect regular induction variables is shown in Figure 54. This algorithm makes successive passes over each instruction in a loop until it fails to locate new regular induction variable items. For every assignment to a register not already known to be a basic or a regular induction variable, the algorithm calls the architecture-specific function, *is_regular_induction*, which examines instructions to find expressions whose value is the product of an induction variable and a constant value, the sum of an induction variable and a loop-invariant

```
proc find_regular_induction_variables(L) is
    repeat
        N = FALSE
        for each B where B ∈ L.blocks_in_loop do
            I = B.first_instruction
            while I ≠ ∅ do
                for each R where R ∈ I.assigned ∧ R.induction ∈ (UNKNOWN ∪ MAYBE) do
                    T = is_regular_induction(R, value_assigned_to(R, I))
                    if T = ∅ ∨ I.inner_loop ≠ L then
                        R.induction = NOT
                    else if R.induction = UNKNOWN then
                        R.induction = MAYBE
                        R.family = T.family
                        R.scale = T.scale
                        R.displacement = T.displacement
                        R.references = R.references ∪ I
                    else if R.induction = MAYBE then
                        R.scale = R.scale · T.scale
                        R.displacement = R.displacement · T.scale + T.displacement
                        R.references = R.references ∪ I
                    endif
                endfor
                I = I.next_instruction
            endwhile
        endfor
        for each R where R ∈ REGISTERS ∧ R.induction = MAYBE do
            R.induction = REGULAR
            N = TRUE
        endfor
        for each R where R ∈ REGISTERS ∧ R.induction = NOT do
            R.induction = UNKNOWN
            R.references = ∅
        endfor
    until ¬ N
endproc
```

Figure 54: Regular induction variable detection algorithm

expression[1], or an expression that consists of both a product and a sum. The algorithm can detect induction variables that are produced either by a single instruction or any sequence of instructions that incrementally apply additional *scale* and *displacement* values to a simpler induced expression. This flexibility is essential because the capabilities of the low-level code are identical to those of the target architecture and, as discussed in Section 1.1.1, there is much variability in the way that different architectures may form a particular induced expression. When the regular induction variable detection algorithm completes, the register information structure of the regular induction variables in the loop will have their "*induction*" field set to *REGULAR*. In

[1] To do this, the function uses the loop-invariance information already collected by the loop-invariant code motion phase.

addition, the "*family*", "*scale*" and "*displacement*" fields describe the induced expression assigned to the regular induction register. When this algorithm is invoked on the low-level code shown in Figure 52, `w[8]` will be marked as a regular induction variable of `w[1]`'s family with a *scale* value of four and a *displacement* value of zero.

The third step in the loop strength reduction process is to examine the address expression of every memory reference in the loop and determine if it is an induced expression. This is done by the algorithm shown in Figure 55. The main purpose of this algorithm is to create an information structure for each memory

```
proc find_induced_address_expressions(L) is
    for all B where B ∈ L.blocks_in_loop do
        I = B.first_instruction
        while I ≠ ∅ do
            for each R where R ∈ I.memory_references do
                M.reference = R
                M.instruction = I
                A = get_address_expression(R)
                T = is_regular_induction(∅, expand_expression(L, I, A))
                if T ≠ ∅ then
                    M.induction = REGULAR
                    M.family = T.family
                    M.scale = T.scale
                    M.displacement = T.displacement
                else
                    M.induction = NOT
                endif
                L.memory_references = L.memory_references ∪ M
            endfor
            I = I.next_instruction
        endwhile
    endfor
endproc
```

Figure 55: Induced address expression detection algorithm

reference in the loop indicating whether the address expression of the reference is an induced expression. Like the regular induction variable detection algorithm, this algorithm uses *is_regular_induction* to determine if an address expression, which could be arbitrarily simple or complex depending on the target architecture, is an induced expression. This process uses the information collected by the basic and normal induction variable detection algorithms, as well as the loop-invariance information compiled by the loop-invariant code motion phase, to determine which family the induced expression is associated with. This information, along with the *scale* and *displacement* values of the induced expression, are stored in the information structure associated with the memory reference. The *expand_expression* function uses the combiner links for the instruction that contains the address expression to expand the expression so that any references to regular or basic induction variables are made explicit. This mechanism is required because many loops contain induced expressions that are too complex for the addressing modes that the target architecture supports to calculate entirely within a single memory reference. When this algorithm is invoked on the code shown in Figure 52, the address of the memory reference in instruction 4 is found to be an induced expression of `w[1]`'s family with scale and displacement values of four and `w[9]`, respectively.

The final step in the loop strength reduction process is to use the information collected by the previous three steps to replace induced address expressions with new induction variables. This is done by the algorithm shown in Figure 56. This algorithm considers each memory reference in the loop and, using the set of registers available for use in the loop determined by the loop-invariant code motion phase, attempts to allocate a new register to use as a new induction variable to replace the induced address expression. The architecture-specific function *do_strength_reduction* is consulted to determine if the address expression is complex enough to warrant strength reduction. The benefit of replacing an induced address expression with a new induction variable dependents on the target architecture because on some CISC architectures a single memory reference using a complex addressing mode can fully subsume a fairly complex induced expression that would require many instructions to compute on a RISC machine. For those memory references whose induced address expressions are complex enough to transform, the algorithm inserts code in the loop's preheader to calculate and assign the initial value of the induced expression to the new induction variable. Then, instructions are added to increment the new induction variable at each location where the basic induction variable in whose family the induced expression was part of is incremented. The increment value of the basic induction variable is multiplied by the *scale* value of the induced expression to produce the increment value of the new induction variable. Finally, the induced address expression is replaced by the new induction variable. The family, scale and displacement values of the induced expression are copied to the information structure of the register used to create the new induction variable in order to allow the induction variable elimination phase, which is discussed in Section 5.5.4, to remove basic induction variables from the loop.

```
func replace_induced_expressions(L) is
    C = FALSE
    for each M where M ∈ L.memory_references ∧ M.induction = REGULAR do
        if do_strength_reduction(M) then
            R = assign_target_register(address_register_type(), L.available_registers)
            if R ≠ ∅ then
                C = TRUE
                L.available_registers = L.available_registers - R
                R.induction = REDUCED
                R.family = M.family
                R.scale = M.scale
                R.displacement = M.displacement
                I = insert_assignment(L.preheader, R, “M.scale · M.family + M.displacement”)
                for each I where I ∈ M.family.increments do
                    N = M.scale · is_basic_induction(I, M.family)
                    I = insert_assignment(I, R, “R + N”)
                    R.references = R.references ∪ I
                    R.increments = R.increments ∪ I
                endfor
                M.induction = REDUCED
                for each I where I ∈ M.references do
                    replace_expression(I, M.reference, R)
                    R.references = R.references ∪ I
                endfor
            endif
        endif
    endfor
    return C
endfunc
```

Figure 56: Induced address expression replacement algorithm

When this algorithm is applied to the loop shown in Figure 52, the induced address expression located in instruction 4 is found to be worth reducing. The algorithm allocates register `w[10]` and inserts the code needed to assign the initial value of the induced expression, which is `4*w[1]+w[9]`, to this register in the preheader block. A new increment instruction is inserted following the increment of the original basic induction variable in instruction 5 to add the product of the original increment and the scale value of the induced expression to the new induction variable. All the instances in the loop where the induced expression value is used are replaced with the new induction variable register. The resulting code is shown in Figure 57.

This code is a good example of how loop strength reduction introduces additional opportunities to perform dead variable elimination, constant propagation, instruction selection and common subexpression elimination transformations. The loop strength reduction phase, like most of the other optimization phases in *llef*, does not perform any of these transformations, but relies on phase iteration to ensure that they are

```
1           w[1]=0;
2           w[9]=w[14]+a;
8           w[10]=w[1]*4;
9   {8}     w[10]=w[10]+w[9];
   L32:
3           w[8]=w[1]*4;
4           W[w[10]]=0;
5           w[1]=w[1]+1;
10          w[10]=w[10]+4;
6           CC=w[1]?100;
7           PC=CC<0,L32;
```

Figure 57: Loop after loop strength reduction

subsequently re-invoked at some point in the compilation process. For example, Figure 58 shows the code that would remain after *llef* re-invoked dead variable elimination to remove the orphaned expression left behind by the loop strength reduction transformations shown in Figure 57.

```
1           w[1]=0;
9           w[10]=w[14]+a;
   L32:
4           W[w[10]]=0;
5           w[1]=w[1]+1;
10          w[10]=w[10]+4;
6           CC=w[1]?100;
7           PC=CC<0,L32;
```

Figure 58: Loop after phase iteration

One advantage of performing code-improvement transformations on low-level code is that the capabilities of the target architecture's instruction set and addressing modes can be fully exploited. The source code shown in Figure 59 is used to illustrate this advantage. The loop shown here contains three induced expressions that are related to the same basic induction variable. The code that results from applying the loop strength reduction algorithm to the SPARC's version of the low-level code for this loop is shown in Figure

```
void vector_add(a, b, c, len)
int a[], b[], c[], len;
{
      int i;

      for (i = 0; i < len; i++)
         a[i] = b[i] + c[i];
}
```

Figure 59: Loop with three related induced expressions

```
1              w[1]=0;
2              w[10]=W[w[30]+a];
3              w[11]=W[w[30]+b];
4              w[12]=W[w[30]+c];
5              w[13]=W[w[30]+len];
   L14:
6              w[8]=W[w[11]];
7              w[9]=W[w[12]];
8   {6,7}      w[8]=w[8]+w[9];                     w[9]
9   {8}        W[w[10]]=w[8];                      w[8]
10             w[1]=w[1]+1;
11             w[10]=w[10]+4;
12             w[11]=w[11]+4;
13             w[12]=w[12]+4;
14             CC=w[1]?w[13];
15             PC=CC<0,L14;
```

Figure 60: Loop with three new induction variables

60. While this code is more efficient than the sequence of instructions that it replaces, better code can be produced if the target architecture supports an index addressing mode. Instead of introducing three new induction variables to perform loop strength reduction on the induced expressions, a single induction variable can be used by exploiting the fact that all three induced expressions are related to the same basic induction variable and have the same *scale* value. Figure 61 shows how this can be done. Instead of using all three of

```
1              w[1]=0;
2              w[10]=W[w[30]+a];
3              w[11]=W[w[30]+b];
4              w[11]=w[11]-w[10];
5              w[12]=W[w[30]+c];
6              w[12]=w[12]-w[10];
7              w[13]=W[w[30]+len];
   L14:
8              w[8]=W[w[10]+w[11]];
9              w[9]=W[w[10]+w[12]];
10  {8,9}      w[8]=w[8]+w[9];                     w[9]
11  {10}       W[w[10]]=w[8];                      w[8]
12             w[1]=w[1]+1;
13             w[10]=w[10]+4;
16             CC=w[1]?w[13];
17             PC=CC<0,L14;
```

Figure 61: Loop with one new induction variable

the newly-allocated registers to create new induction variables, this code calculates the difference between the first new induced expression added to the loop and the other related induced expressions and then adds it to the first induction variable using the index addressing mode. Thus, while the preheader code is less efficient, the loop, which now executes two fewer instructions during each iteration, will be able to make up for this deficit during the first iteration of the loop.

To produce the code shown in Figure 61, *llef*'s loop strength reduction algorithm invokes the *exploit_index_mode* procedure, shown in Figure 62, after each loop strength reduction transformation

```
proc exploit_index_mode(L, R) is
    for each M where M ∈ L.memory_references ∧ M.induction = REGULAR do
        if M.family = R.family ∧ M.scale = R.scale then
            D = assign_target_register(address_register_type(), L.available_registers)
            if D ≠ ∅ then
                D.references = ∅
                for each I where I ∈ M.references do
                    N = I
                    replace_expression(N, M.reference, "R+D")
                    if is_valid_instruction(N) then
                        I = N
                        D.references = D.references ∪ I
                        R.references = R.references ∪ I
                    endif
                endfor
                if D.references ≠ ∅ then
                    L.available_registers = L.available_registers - D
                    D.loop_invariant = YES
                    I = insert_assignment(L.preheader, D, "M.displacement - R.displacement")
                    D.invariant_value = I
                    M.induction = INDEX
                endif
            endif
        endif
    endfor
endproc
```

Figure 62: The *exploit_index_mode* procedure

performed by the *replace_induced_expressions* function in Figure 56. This procedure scans through all of the induced expressions that have not yet had loop strength reduction performed on them to determine if any of them contain the same basic induction variable and *scale* value as the new induction variable. If any such induced expression is found, the procedure attempts to replace all references to it with an index expression. The machine description is consulted to ensure that the target architecture supports the index addressing mode. Using this strategy, the algorithm can exploit a common architecture-specific feature in an architecture-independent way to produce more efficient code without increasing the amount of effort required to retarget *llef*.

5.5.3 Recurrence optimization

A *recurrence* is an instance of a value that is computed in a loop and used in a subsequent iteration of the same loop. If a value is used only in the next iteration of the loop, then the recurrence is called a first-order recurrence. When the value is used on the second iteration after it is computed, it is called a second-order recurrence, or a recurrence with an order of two. Recurrences can be optimized by using a register to hold the value across one or more iterations of the loop so that the subsequent references to the value can obtain it from the register instead of the memory location where it was stored. Figure 63 shows a simple loop that contains a first-order recurrence.

```
for (i = 1; i < 100; i++)
   a[i] = b[i] + (a[i - 1] * 5);
```

Figure 63: Loop containing a first-order recurrence

Because complex dependence analysis is required to perform recurrence detection, this transformation has been performed exclusively on source or high-level intermediate code. The recurrence detection and optimization algorithms developed for *llef*, however, operate on low-level code. Figure 64 shows the low-level code that would be produced on the SPARC for the loop in Figure 63 and is used to illustrate the recurrence detection and optimization algorithm. Note that variable promotion and loop-invariant code motion have already been performed on this code.

In order to perform recurrence optimization, *llef* examines a loop not only to detect recurrences, but also to ensure that no possible extraneous dependencies exist between the recurrent references and the other references in the loop. Consider, for example, the loop shown in Figure 65. While this loop contains two

```
for (i = n, j = m; i < m; i++, j--) {
   a[i] = b[i] * a[i - 1];
   a[j] = b[j] * a[j + 1];
   }
```

Figure 65: Loop containing possible extraneous dependences

separate recurrences, there exist values for `n` and `m` which produce dependences between the two recurrences. In such cases, using a register to hold the recurrent value of a previous iteration is inappropriate because the

```
                w[8]=1;
                w[9]=w[14]+a;
                w[10]=w[14]+b;
   L60:
                w[11]=w[8]-1;
   {4}          w[11]=w[11]*4;
   {5}          w[11]=W[w[11]+w[9]];
   {6}          w[11]=w[11]*5;
                w[12]=w[8]*4;
   {8}          w[12]=W[w[12]+w[10]];
   {7,9}        w[11]=w[11]+w[12];               w[12]
                w[12]=w[12]*4;
   {10,11}      W[w[12]+w[9]]=w[11];             w[11],w[12]
                w[8]=w[8]+1;
                CC=w[8]?100;
                PC=CC<0,L60;
```

Figure 64: Low-level loop with recurrence

value in the memory reference that a register has been assigned to represent is sometimes altered while the corresponding register value is not. Thus, excluding any additional information that might allow the compiler to statically determine that none of the possible values of `n` and `m` produce extraneous dependences or the insertion of special code to determine at run time if it is safe to execute an optimized instance of the loop, recurrence optimizations should not be performed on this loop.

The recurrence detection algorithm uses the induction variable information collected by the loop strength reduction phase to locate recurrences. The algorithm also gathers the additional information needed to ensure that no extraneous dependences exist. The first step of this process is determining which *partition* each memory access in the loop references. A partition is an area of storage that is disjoint from all other partitions and often corresponds to a user-declared variable. For example, each local and global user-variable, regardless of whether it is a scalar or an array, defines a partition.

Figure 66 shows the algorithm that determines which partitions are accessed by each of the memory references in a loop. The algorithm does this by expanding each address expression so that every induction variable and loop-invariant item is replaced with an with an expression representing its value at the start of the loop. The expanded address is scanned by the *get_partition* function to determine which memory partition is referenced. This task consists of searching for a *partition handle*, which is a local, global or label identifier that uniquely identifies a partition. The function will not return a partition item if no suitable handle is found

```
proc find_partitions(L) is
    X = ∅
    for each M where M ∈ L.memory_references do
        R = get_reference(M.instruction, M.reference)
        A = get_address_expresion(R)
        P = get_partition(expand_invariant_items(L, M.instruction, A))
        if P = ∅ then
            X = X ∪ M
        else
            L.partitions = L.partitions ∪ P
            if is_memory_write(M.instruction, M) then
                P.writes = P.writes ∪ M
            else
                P.reads = P.reads ∪ M
            endif
        endif
    endfor
    if X ≠ ∅ then
        if L.partitions = ∅ then
            L.partitions = new_partition(NONAME)
        endif
        for each P where P ∈ L.partitions do
            for each M where M ∈ X do
                if is_memory_write(M.instruction, M) then
                    P.writes = P.writes ∪ M
                else
                    P.reads = P.reads ∪ M
                endif
            endfor
        endfor
    endif
endproc
```

Figure 66: Partition determination algorithm

to signify that it is unable to narrow the location referenced by the address expression to a specific partition. After determining which partition a memory reference accesses, the reference is added to the set of items that reference the partition. If the partition accessed by the memory reference cannot be determined, the reference is added to all of the partitions referenced in the loop because it potentially refers to all of them.

When the partition determination algorithm is applied to the loop shown in Figure 64, it examines the address expressions of the three memory references that were located by the induced address expression detection algorithm. The address expression of the memory references in instruction 6, 9 and 11 are expanded to "`(1-1)*4+w[14]+a`", "`1*4+w[14]+b`" and "`1*4+w[14]+a`", respectively. The partition handles for these expressions are the identifier "`a`" for the first and last expression and "`b`" for the second expression.

Figure 67 illustrates the partition and memory reference information that would be collected for this loop by the induction variable and partition determination algorithms.

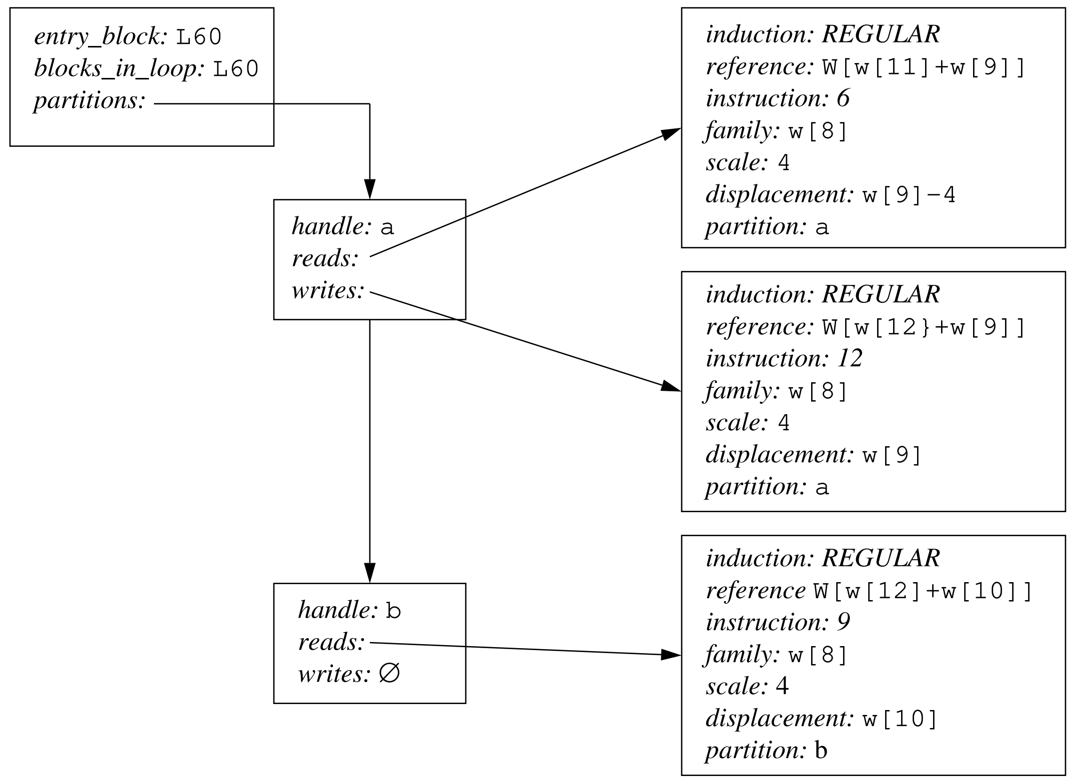


Figure 67: Memory reference and partition information

Sorting memory references into partitions serves two important purposes. First, it divides the references into smaller, related groups which facilitate the task of locating the recurrences. Because recurrences exist only when two or more memory references access the same location, the recurrence detection algorithm can limit its search to partitions that contain both read and write references. Second, it allows the recurrence detection algorithm to verify the absence of extraneous dependences that cause the recurrence transformations to alter the semantics of the loop.

Using the partition information and induction variable information of a loop, the algorithm shown Figure 68 determines the relative offsets between all of the address expressions associated with a partition. The algorithm examines each partition to ensure that every address expression is related to the same basic induction variable and uses the same scale factor as all the other address expressions in the partition. The relative offset value of each memory reference is determined by expanding its address expression and then

```
proc find_relative_offsets(L) is
    for each P where P ∈ L.partitions do
        P.safe = TRUE
        P.family = ∅
        for each M where M ∈ (P.reads ∪ P.writes) do
            if M.induction = INDUCTION then
                if P.family = ∅ then
                    P.family = M.family
                    P.scale = M.scale
                else if M.family ≠ P.family ∨ M.scale ≠ P.scale then
                    P.safe = FALSE
                endif
                R = get_reference(M.instruction, M.reference)
                A = get_address_expresion(R)
                E = expand_invariant_items(L, M.instruction, A))
                E = E - P.handle
                for each R where R ∈ REGISTERS ∧ R.always_invariant do
                    E = E - R
                endif
                M.offset = apply_algebraic_simplifications(E)
                if ¬ is_constant_literal(M.offset) then
                    P.safe = FALSE
                endif
            else
                P.safe = FALSE
            endif
        endfor
    endfor
    P.increment = P.scale · P.family.increment
endproc
```

Figure 68: Relative offset determination algorithm

removing the partition handle identifier and any invariant register values. After algebraic simplification, the remaining expression becomes the relative offset value for the memory reference. Any partitions containing memory references whose relative offset value cannot be algebraically simplified into a literal constant are marked unsafe to prevent recurrence optimizations from being performed on the partition. After verifying that every memory reference in the partition is related to the same basic induction variable and has the same scale factor, the increment value of the partition is obtained by multiplying the value added to the basic induction variable on every iteration of the loop by the common scale factor.

When the relative offset determination algorithm is invoked on the code shown in Figure 64, it collects the offset information shown in Figure 69. For partition `a`, the fully expanded address expression for the memory reference in instruction 6 is "`(1-1)*4+w[14]+a`". After the partition handle, `a`, and the invariant frame pointer register, `w[14]`, are subtracted and algebraic simplification is performed, the

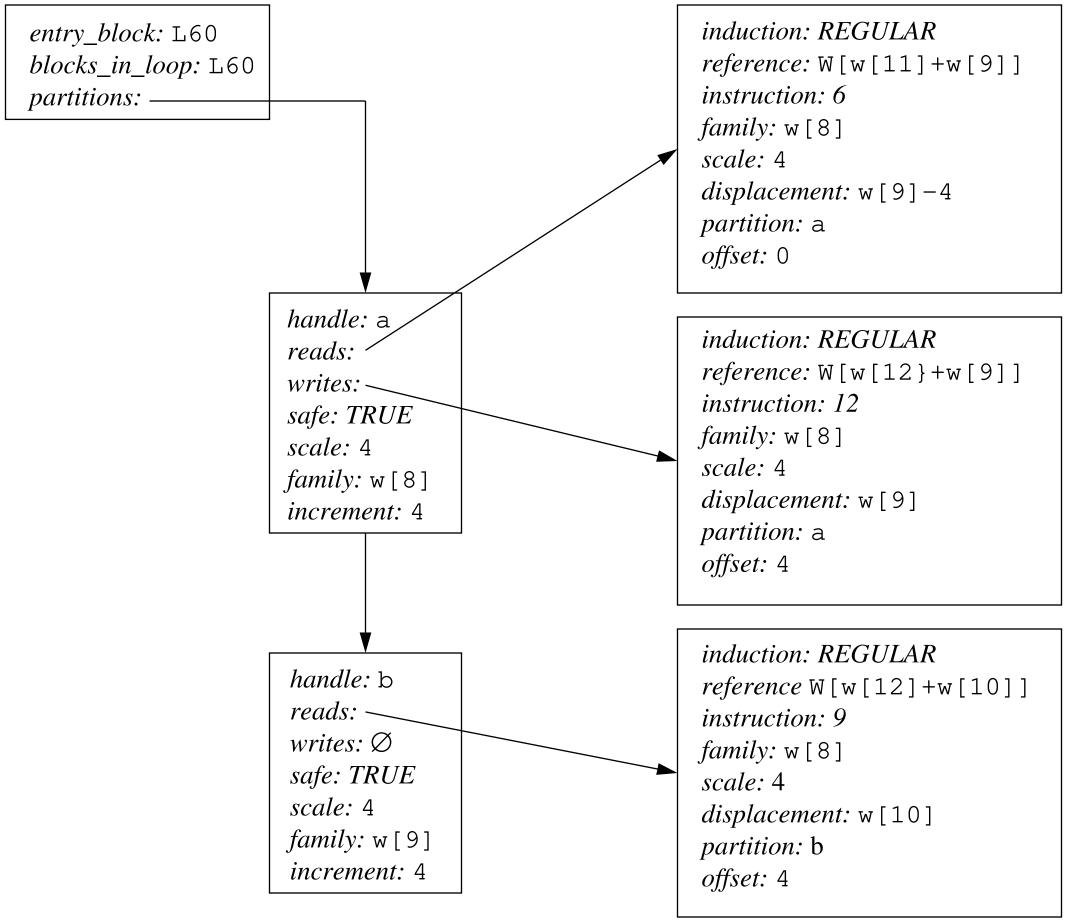


Figure 69: Final memory reference and partition information

resulting value, "0", becomes the displacement value for the reference. A similar process is used to produce the displacement value of "4" for the other two memory references. The basic induction variable shared by all of the memory references is `w[8]`. The increment value of this basic induction variable is 1, which is multiplied by the scale factor of each partition to produce the increment value of each partition.

The recurrence detection and optimization algorithm is shown in Figure 70. This algorithm operates on the principle that the increment value of a partition can be added to the offset value of each memory reference in the partition to determine the offset value of the reference on the next iteration of the loop. Thus, a partition that contains a write at offset 4, a read at offset 0 and an increment value of 4 has a recurrence because the value written at offset 4 on any given iteration will be read on the next iteration of the loop. The detection algorithm considers every pair of read and write references in a partition and determines the difference between their relative offsets. This difference is divided by the increment value of the partition and, if the result is a positive, non-zero integer, a recurrence exists whose order is equal to the quotient. Upon

```
func recurrence_optimization(L) is
    C = FALSE
    for each P where P ∈ L.partitions ∧ P.safe do
        for each W where W ∈ P.writes do
            for each R where R ∈ P.reads ∧ ¬ R.optimized do
                D = W.offset - R.offset
                O = D / P.increment
                if O > 0 ∧ O = int(O) then
                    E = assign_target_registers(O + 1, type_of_expression(W.reference),
                                                L.available_registers)
                    if E ≠ ∅ then
                        insert_initial_assignments(L.preheader, E, W.reference, L.increment)
                        load_write_value(E_0, W.reference)
                        for each I where I ∈ R.references do
                            replace_expression(I, R.reference, E_O-1)
                        endfor
                        while O > 0 do
                            insert_assignment(L.entry_block, E_O, E_O-1)
                            O = O + 1
                        endwhile
                        R.optimized = TRUE
                        C = TRUE
                    endif
                endif
            endfor
        endfor
    endfor
    return C
endfunc
```

Figure 70: Recurrence detection and optimization algorithm

detecting a recurrence, the algorithm attempts to allocate a set of registers whose size is equal to the order of the recurrence detected plus one to form a register pipeline capable of holding the values generated by each iteration of the loop that separates the write to the recurrence location from the read. If enough registers can be obtained, the algorithm calls the architecture-specific *insert_initial_assignments* function which inserts code to prime the register pipeline with the values needed during the initial iterations of the loop. Then, the memory write is altered so that the value stored into the recurrent memory location is also placed at the front of the pipeline. The instructions that read from the recurrence location are modified so that they obtain the value from the back of the register pipeline. Finally, a sequence of register-to-register transfers is inserted at the top of the loop to advance each value forward in the pipeline at the start of each loop iteration.

Figure 71 shows the result of applying the recurrence detection and elimination algorithm to the loop in Figure 64. The algorithm determines that the memory reference at instruction 6 reads the same location

```
1              w[8]=1;
2              w[9]=w[14]+a;
3              w[10]=w[14]+b;
16             w[11]=w[8]-1;
17 {16}        w[11]=w[11]*4;
18 {17}        w[13]=W[w[11]+w[9]];
   L60:
20             w[1]=w[13];                          w[13]
4              w[11]=w[8]-1;
5  {4}         w[11]=w[11]*4;
6  {5}         w[11]=w[1];
7  {6}         w[11]=w[11]*5;
8              w[12]=w[8]*4;
9  {8}         w[12]=W[w[12]+w[10]];
10 {7,9}       w[11]=w[11]+w[12];                   w[12]
11             w[12]=w[12]*4;
19 {10}        w[13]=w[11];                         w[11]
12 {11,19}     W[w[12]+w[9]]=w[13];                 w[12]
13             w[8]=w[8]+1;
14             CC=w[8]?100;
15             PC=CC<0,L60;
```

Figure 71: Loop after recurrence optimization

written by instruction 12 during the previous iteration of the loop. Since it is a first-order recurrence, the algorithm requests a pair of registers to form a register pipeline over the entire life of the loop. After obtaining a pair of registers, the algorithm inserts instructions 16, 17 and 18 in the loop's preheader to load the value that will be read from the pipeline in the first iteration of the loop. Instruction 19 is then inserted to continue filling the pipeline on each iteration of the loop. Instruction 12 is modified to allow subsequent invocations of the instruction selection and common subexpression elimination phases to integrate the calculation of the next recurrence value with the filling of the register pipeline. The memory read in instruction 6 of Figure 64 is replaced with a reference to the last register in the pipeline, thus eliminating a memory reference from the loop—which is the intent of the recurrence optimization. Finally, instruction 20 is added at the top of the loop to advance the values in the pipeline forward at the start of each new loop iteration. Like loop strength reduction transformations, recurrence optimizations often create opportunities to perform additional optimizations and rely on phase iteration to exploit them. Figure 72 shows the code that is eventually produced for this loop after performing recurrence optimization and re-invoking all of the other applicable optimization phases except for induction variable elimination, which is the subject of the next section.

```
1             w[8]=1;
2             w[2]=w[14]+a+4;
3             w[3]=w[14]+b+4;
21            w[3]=w[3]-w[2];
18            w[13]=W[w[14]+a];
   L60:
7             w[11]=w[13]*5;                  w[13]
9             w[12]=W[w[2]+w[3]];
10 {7,9}      w[13]=w[11]+w[12];              w[11],w[12]
12 {10}       W[w[2]+w[3]]=w[13];
13            w[8]=w[8]+1;
22            w[2]=w[2]+1;
14            CC=w[8]?100;
15            PC=CC<0,L60;
```

Figure 72: Loop containing recurrence after phase iteration

5.5.4 Induction variable elimination

Loop strength reduction transformations introduce new induction variables into loops that are related to pre-existing basic induction variables. These new induction variables reduce the number of references made to the original induction variable. In some instances, all references to the basic induction variable are eliminated and it can be removed from the loop. In most cases, as with the code shown in Figure 57, references to the basic induction variable remain.

Induction variable elimination attempts to remove all of the references to a basic induction variable by transforming them so that they reference a related induction variable instead. In *llef*, induction variable elimination operates in conjunction with loop strength reduction and uses the induction variable information gathered by that phase. Because the most common use of an induction variable after array indexing is to control the number of iterations that the loop performs, the induction variable elimination algorithm specializes in transforming compare instructions.

Figure 73 shows the induction variable elimination algorithm. This algorithm references the information produced by the loop strength reduction algorithm to determine which new induction variables were created and attempts to eliminate any remaining references to the basic induction variables related to them. The algorithm does this by calling the architecture-specific function, *is_a_compare_instruction*, to find instructions that compare these basic induction variables against loop-invariant expressions. When such comparisons are found, the algorithm transforms then by replacing the basic induction variables with a related induction variable and the comparison values with a corresponding scaled and displaced constant. Since the

```
func induction_variable_elimination is
    C = FALSE
    for each R where R ∈ REGISTERS ∧ R.induction = REDUCED do
        F = R.family
        if F.induction ≠ ELIMINATED then
            D = R.scale / F.scale
            if D > 0 then
                for each I where I ∈ F.references do
                    C = is_a_compare_instruction(I, F)
                    if C ≠ ∅ then
                        V = assign_target_register(type_of_register(F), L.available_registers)
                        if V ≠ ∅ then
                            S = insert_assignment(L.preheader, V, "C.value · D + R.displacement")
                            replace_expression(I, C.reference, R)
                            replace_expression(I, C.variable, V)
                            F.induction = ELIMINATED
                            F.references = F.references - I
                            C = TRUE
                        endif
                    endif
                endfor
            endif
        endif
    endfor
    for each R where R ∈ REGISTERS ∧ R.induction ∈ (BASIC ∪ ELIMINATED) do
        if R.increments = R.references do
            X = TRUE
            for each B where B ∈ L.exit_blocks do
                if R ∈ B.values_live_on_exit then
                    X = FALSE
                endif
            endfor
            if X then
                for each I ∈ R.increments do
                    remove_instruction(I)
                    R.increments = R.increments - I
                    R.references = R.references - I
                    C = TRUE
                endfor
            endif
        endif
    endfor
    return C
endfunc
```

Figure 73: Induction variable elimination algorithm

new constant is loop-invariant, the algorithm attempts to obtain a register so that the scaling operation can take place outside of the loop. The second part of the algorithm deletes any basic induction variables that will not be used outside the loop and are only incremented within the loop. These basic induction variables increments are specifically removed at this point because their circular self-reference prevents the normal dead variable elimination phase from detecting and deleting them.

When the induction variable elimination algorithm is invoked after the loop strength reduction algorithm has transformed the code shown in Figure 57, it further transforms the loop to produce the code shown in Figure 74. The algorithm examines basic induction variable `w[1]` because it is related to induction variable `w[10]`, which was created by the loop strength reduction phase. When the algorithm detects that this basic induction variable is compared to 100 in instruction 6, it inserts instructions 11, 12 and 13 to calculate the value that `w[10]` will contain when the value of `w[1]` is 100. Instruction 6 is transformed to refer to the new induction variable and to use the constant value that maintains the original loop's semantics. Because a reference to the basic induction variable remains in instruction 3, the increment of the basic induction variable in instruction 5 is not removed in the first iteration of the dead variable elimination phase.

```
1            w[1]=0;
2            w[9]=w[14]+a;
8            w[10]=w[1]*4;
9  {8}       w[10]=w[10]+w[9];
11           w[11]=100;
12 {11}      w[11]=w[11]*4;
13 {12}      w[11]=w[11]+w[9];
   L32:
3            w[8]=w[1]*4;
4            W[w[10]]=0;
5            w[1]=w[1]+1;
10           w[10]=w[10]+4;
6            CC=w[10]?w[11];
7            PC=CC<0,L32;
```

Figure 74: Loop after induction variable elimination

However, because instruction 3 is an orphan expression left behind by loop strength reduction, it will be removed by the next invocation of dead variable elimination and trigger another iteration of the loop optimization phases. On the second iteration of the induction variable elimination phase, instruction 5 will be removed from the loop. Figure 75 shows the final code produced for the loop after phase iteration finishes invoking all of the optimizations that improve the loop's code.

```
9           w[10]=w[14]+a;
13 {9}      w[11]=w[10]+400;
   L32:
4           W[w[10]]=0;
10          w[10]=w[10]+4;
6           CC=w[10]?w[11];
7           PC=CC<0,L32;
```

Figure 75: Loop after phase iteration

5.6 Architecture-dependent optimizations

Peephole optimization, which is the most common architecture-dependent optimization, is performed by the instruction selection mechanism, which is an integral and indispensable part of *llef*. Other architecture-dependent optimizations are incorporated into *llef* and performed on the same low-level representation used throughout the optimizer. An advantage of *llef* over other retargetable optimizing compilers is that, because all optimizations operate on a low-level representation, architecture-dependent optimizations are an integral part of the optimizer rather than separate post-passes. The following sections discuss the implementation of two popular architecture-dependent optimizations in *llef*, instruction scheduling and delay slot filling. Although these phases are incorporated into the architecture-independent part of *llef*, they are only invoked when generating code for architectures that benefit from the transformations that these phases perform.

5.6.1 Instruction scheduling

Many modern architectures use multiple-stage instruction pipelines to increase their performance. Each stage in an instruction pipeline specializes in performing a subset of the tasks required to execute an instruction and, because each stage operates independently, a pipelined processor can operate on multiple instructions simultaneously. Figure 76 shows a typical three-stage pipeline, where up to three instructions may be executing simultaneously. A common division of labor among a three-stage pipeline is:

- fetch and decode the instruction in the first stage,
- obtain the required operands in the second stage and
- perform the desired operation and store the results in the last stage.

Given the instruction sequence α, β and χ to execute, instruction α starts executing in the first cycle, but does not store its results until the end of the third cycle. If instruction β requires the value calculated by instruction α, then step β_b, which reads the operand values for instruction β, will have to wait until the fourth cycle, after

cycle 1	cycle 2	cycle 3	cycle 4	cycle 5
α_a	α_b	α_c		
	β_a	β_b	β_c	
		χ_a	χ_b	χ_c

Figure 76: Three-stage pipeline

stage α_c has completed, to obtain them. This condition is called a *data hazard*, and is just one of the various types of hazards that a pipelined processor can encounter [HENN90]. Some pipeline processors contain hardware to detect data hazards and stall the processor until all of the requested operands are available. This action is called a *pipeline stall*. Another solution, which requires additional hardware, is to use a *bypass* circuit to allow a value to be used by a previous stage of the pipeline even before it is written to its destination. Other processors perform no conflict checks, and rely on the compiler to ensure that sufficient instructions exist between the computation of a value and its use, even if the only effect of the intervening instructions is to consume processor cycles.

Memory system latency can produce a condition similar to a data hazard even on non-pipelined processors. After a load instruction is issued, many load/store architectures allow the processor to continue processing as long as the destination register specified in the load instruction is not referenced. If the register is referenced before the memory system has fetched the value, the processor must wait until the value arrives. Architectures exist that do not stall the processor if the destination register of a load instruction is referenced before the value arrives, thus forcing the compiler to ensure that sufficient instructions are placed between a load operation and the use of the requested value.

Detailed execution diagrams, such as the one shown in Figure 77 for the loop code in Figure 72, are used to determine how many processor cycles are needed to execute a sequence of instructions on a pipelined processor. Note that this sequence of instructions incurs three pipeline stalls for a total delay of four processor cycles. The first delay is caused by the second stage of instruction 10, which is delayed two processor cycles while the memory system fetches the value requested by instruction 9. The second delay occurs in instruction 12, which must stall for one processor cycle to obtain the results of the value computed by instruction 10.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
7_a	7_b	7_c											
	9_a	9_b	9_c										
		10_a	stall[1]	stall[1]	10_b	10_c							
					12_a	stall[2]	12_b	12_c					
							13_a	13_b	13_c				
								21_a	21_b	21_c			
									14_a	14_b	14_c		
										15_a	stall[3]	15_b	15_c

Figure 77: Pipeline execution diagram for loop body

[1] Processor waiting for the memory system to fetch the value loaded into `w[12]` by instruction 9.
[2] Processor waiting to obtain the value computed into `w[13]` by instruction 10.
[3] Processor waiting to obtain the value computed into the condition codes by instruction 14.

Finally, the conditional branch instruction at the end of the loop is delayed until the results of the comparison performed by instruction 14 are available.

The amount of time spent waiting on pipeline stalls and the memory system can be substantially reduced by performing instruction scheduling. This optimization rearranges instructions to interpose as much useful code as possible between the assignment of a value and it subsequent use. The local-scope instruction scheduling algorithm used by *llef* operates on the low-level code after all other optimizations except delay slot filling have been performed. The scheduler is mostly architecture-independent and relies on the information provided for each instruction by *llef*'s machine description along with a few small architecture-specific functions to perform effective instruction scheduling on the target architecture. The instruction scheduler can schedule code for processors that automatically stall the pipeline when a hazard is detected and for processors that require the compiler to produce code without hazards.

Figure 78 shows the architecture-independent top level of the instruction scheduling algorithm. This algorithm operates on each basic block individually. Before scheduling a basic block, all of the instructions in the block are placed in an *unissued set*. The next step is to determine the *leader set*, which is the set of unissued instructions that do not depend on any value computed by an unissued instruction. The final step entails selecting one of the instructions from the leader set and moving it to the *issued set*[1]. When more than one instruction in the leader set can be issued, the algorithm chooses the instruction that will least delay the processor. Then, any instructions remaining in the leader set are put back into the unissued set and the process

[1] This operation is also known as "issuing the instruction".

```
proc instruction_scheduling is
    for each B where B ∈ CFG do
        U = B.first_instruction
        B.first_instruction = ∅
        B.last_instruction = ∅
        L = calculate_leader_set(U)
        while R ≠ ∅ do
            I = choose_instruction(B.last_instruction, L, U)
            if E = ∅ then
                B.first_instruction = I
            else
                E.next_instruction = I
            endif
            I.next_instruction = ∅
            B.last_instruction = I
            U = merge_instructions(L, U)
            L = calculate_leader_set(U)
        endwhile
    endfor
endproc
```

Figure 78: Instruction scheduling algorithm

repeats until the unissued set is empty. The final order of the instructions in the basic block is the order in which instructions were issued.

The *calculate_leader_set* function, shown in Figure 79, uses an architecture-independent algorithm to determine which of the instructions in the unissued set do not depend on values calculated by unissued instructions. The function initializes the leader set with the first instruction in the unissued set. If the machine description has indicated through the "*alone*" field of the instruction that the instruction should be in the leader set by itself, a feature that is used to prevent control-flow instructions from issuing before the instructions that precede them, then no more instructions are added to the leader set. Otherwise, the function scans the remaining unissued instructions and adds to the leader set any instructions that do not conflict with any of the previously scanned instructions. Conflicts are determined by the contents of the "*does*" and "*blocks*" fields of the instructions and by the items that the instructions modify and use. The "*does*" and "*blocks*" fields are assigned by the machine description and are used to ensure that instructions that modify or use values that are not explicitly referenced in the low-level code do not lose their relative ordering. An example would be a pair of load and store instructions that reference potentially aliased memory locations. The "*does*" field for the load instruction and the "*blocks*" field for the store instruction would contain the *REF_UNK* item, which would ensure that the original order of the load and store would remain unchanged.

```
func calculate_leader_set(U) is
    L = U
    E = L
    U = L.next_instruction
    L.next_instruction = ∅
    T = ∅
    D = L.does
    X = L.blocks
    S = L.assigned
    R = L.used
    if L.alone then
        return L
    endif
    I = U
    while I do
        N = I.next_instruction
        if I.alone then
            return L
        endif
        if D ∩ I.blocks = ∅ ∧ X ∩ I.does = ∅ ∧
         S ∩ I.used = ∅ ∧ R ∩ I.assigned = ∅ ∧ S ∩ I.assigned then
            if T = ∅ then
                U = N
            else
                T.next_instruction = N
            endif
            I.next_instruction = ∅
            E.next_instruction = I
            E = I
        else
            T = I
        endif
        D = D ∪ I.does
        X = X ∪ I.blocks
        S = S ∪ I.assigned
        R = R ∪ I.used
        I = N
    endwhile
    return L
endfunc
```

Figure 79: The *calculate_leader_set* function

Because the *REF_UNK* item does not appear in the "*blocks*" field of a load, however, the relative order of two load instructions could change. The items that are explicitly modified and used by each instruction also create conflicts. In particular, it is important to ensure that no instruction that references a value is moved to the leader set where it might be issued ahead of the instruction that calculates the value. Also, an instruction that assigns to an item that contains a value needed by an unissued instruction must not be added to the leader set, because issuing it would improperly overwrite the value. Such a conflict is called a *storage-related dependency.*

Figure 80 shows the instructions that make up the body of the loop shown in Figure 72 along with the values of their "*assigned*", "*used*", "*does*", "*blocks*" and "*alone*" fields. If these instructions comprise the

#	Instruction	*assigned*	*used*	*does*	*blocks*	*alone*
7	`w[11]=w[13]*5;`	`w[11]`	`w[13]`	∅	∅	`FALSE`
9	`w[12]=W[w[2]+w[3]];`	`w[12]`	`w[2],w[3]`	`REF_UNK`	∅	`FALSE`
10	`w[13]=w[11]+w[12];`	`w[13]`	`w[11],w[12]`	∅	∅	`FALSE`
12	`W[w[2]+w[3]]=w[13];`	∅	`w[2],w[3],w[13]`	∅	`REF_UNK`	`FALSE`
13	`w[8]=w[8]+1;`	`w[8]`	`w[8]`	∅	∅	`FALSE`
21	`w[2]=w[2]+1;`	`w[2]`	`w[2]`	∅	∅	`FALSE`
14	`CC=w[8]?100;`	`CC`	`w[8]`	∅	∅	`FALSE`
15	`PC=CC<0,L60;`	`PC`	`CC`	∅	∅	`TRUE`

Figure 80: Loop code before instruction scheduling

unassigned set when the *calculate_leader_set* function is called, instruction 7 would be removed from the unassigned set to become the first member of the leader set. Since this instruction does not have to be alone, the function will attempt to add additional instructions to the leader set. Before doing so, it will set:

- `S` to `w[11]`, which is the set of items assigned by the instructions scanned,
- `R` to `w[13]`, which is the set of items whose values are needed by the instructions scanned,
- `D` to ∅, which is the set of items done by the instructions scanned, and
- `X` to ∅, which is the set of items blocked by the instructions scanned.

The function will add instruction 9 to the leader set because it does not conflict with any of the items assigned, used, done or blocked by the previously scanned instructions. The conflict sets are updated as follows:

- `w[12]` is added to `S` to yield `w[11]` and `w[12]`,
- `w[2]` and `w[3]` are added to `R` to yield `w[2]`, `w[3]` and `w[13]`, and
- `REF_UNK` is added to `D` to yield `REF_UNK`.

Unlike the previous two instructions, instruction 10 is not added to the leader set because its "*used*" field contains `w[11]` and `w[12]`, which appear in `S`. Clearly, it would be inappropriate to issue this instruction before either of the two instructions already in the leader set, since it uses the values that they calculate. Although instruction 10 is not added to the leader set, its "*assigned*", "*used*", "*does*" and "*blocks*" fields are added to the appropriate conflict items to prevent any of the remaining instructions that may conflict with them one from being added to the leader set. Instruction 12 is not added to the leader set because it uses `w[13]`, which is in `S`. Instruction 13 is added to the leader set because it does not conflict with any of the instructions examined so far. Instruction 21 is not moved into the leader set because it assigns a value to `w[2]`, which is in `R`. Instruction 14 is not placed in the leader set because it uses `w[8]`, which was added to `S` by instruction 13. Finally, instruction 15 cannot be added to the leader set at this time because the machine description has indicated that it must enter the leader set by itself.

The leader set contains all of the instructions that can be issued next without affecting the semantics of the original code sequence. Because the order in which instructions are issued affects the amount of time that a pipelined CPU will spend waiting on pipeline stalls and memory system latency, the scheduling algorithm calls the *choose_instruction* function, shown in Figure 81, to determine which instruction to issue next. Although the outcome of this determination depends on the target architecture, most of it can be

```
func choose_instruction(E, L, U) is
    I = L
    while I ≠ ∅ do
        A = 0
        D = U
        while D ≠ ∅ do
            if I.does ∩ D.blocks ≠ ∅ ∨ I.blocks ∩ D.does ≠ ∅ ∨
             I.assigned ∩ D.used ≠ ∅ ∨ I.used ∩ D.assigned ≠ ∅ ∨ I.assigned ∩ D.assigned ≠ ∅ then
                A = max(D.advantage, A)
            endif
            D = D.next_instruction
        endfor
        A = I.advantage + A ÷ 2
        if E ≠ ∅ then
            A = max(A - issue_penalty(I, E), 0)
        endif
        I.benefit = A
        I = I.next_instruction
    endwhile
    order_by_benefit(L)
    I = L
    while I ≠ ∅ do
        if can_issue_instruction(E, I) then
            return I
        endif
        I = I.next_instruction
    endwhile
    return NO_OP_INSTRUCTION
endfunc
```

Figure 81: The *choose_instruction* function

performed by an architecture-independent algorithm that uses the information provided by the machine description and a few simple architecture-dependent functions. The algorithm operates by assigning a benefit value to each instruction according to:

- the advantage value assigned to the instruction by the machine description,
- one-half of the maximum advantage of any of the instructions that are prevented from issuing by the instruction and
- a penalty value reflecting the cost of the pipeline stall or memory system wait that would result if the instruction were issued.

The advantage value reflects the fact that some instructions are more likely to cause delays later in the basic block than others. By assigning a higher advantage value to these instructions, the machine description increases the probability that they will be issued early, thus leaving more instructions in the unissued set to place between them and the instructions that may delay. For example, a load instruction may not necessarily delay, but it is usually better to issue it early so as to leave as many instructions as possible between it and the instruction that uses the value that it loads. Because an instruction that should be issued early could be prevented from doing so by an instruction that has a relatively low advantage value, a portion of the maximum advantage of any of the instructions that are waiting on the instruction to issue is added to the instruction's benefit value. Finally, the benefit value is substantially reduced if issuing the instruction would cause a stall. Since this cannot be determined without specific information regarding the target architecture's pipeline and memory system, an architecture-specific function called *issue_penalty* is used to determine this penalty value.

Once a benefit value is calculated for each of the instructions in the leader set, the set is ordered so that the instructions with the highest benefit is first. Following this order, the function then chooses the first instruction that can be issued. Again, an architecture-specific function is consulted to ensure that an instruction can be issued. If none of the instructions in the leader set can be issued, then the function returns a *no-op instruction*, which is an instruction that performs no useful function except to consume a processor cycle. This feature allows *llef* to generate code on architectures that do not contain the hardware needed detect hazards.

Before discussing the sequence of events that will transpire when the *choose_instruction* function is invoked with the leader set obtained from the code in Figure 80, the behavior of the machine description and architecture-specific functions that control the outcome of the scheduling algorithm must be presented. For this example the machine description assigns every instruction an advantage of 20 except for loads, which are assigned an advantage value of 30. Also, the *issue_penalty* function returns 20 if the previously issued instruction assigns to any of the register values used by the instruction it is called with and 10 if the instruction issued before the previously issued instruction does. Finally, since the target architecture detects hazards, the *can_issue_instruction* function always returns *TRUE*.

For instruction 7, the advantage value is 20 and instruction 10 is the only instruction that it prevents from moving into the leader set. One-half of the maximum advantage value of this single-element set is 10. Since no previous instructions have been issued for this basic block, there is no issue penalty, so the benefit value assigned to this instruction is 30. Instruction 9 is a load instruction with an advantage value of 30. It prevents instructions 10, 12 and 21 from joining the leader set, and one-half the maximum advantage value of these instructions is 10. Thus, with no penalty value, instruction 9 is assigned a benefit value of 40. Finally, instruction 13 has an advantage value of 20 and only keeps instruction 14 out of the leader set, so its benefit value is 30. Because instruction 9 has the highest benefit value, it will be considered first and, since the *can_issue_instruction* function does not prevent it from issuing, it will be returned by the *choose_instruction* function and issued.

Once instruction 9 is placed in the issue set, the instruction scheduler merges the instructions remaining in the leader set back into the unissued set and starts the process of choosing a new leader set over again. Figure 82 shows the state of the unissued set and the leader set, along with the benefit values assigned

Unissued set	Leader set(Benefit value)	Instruction issued
7, 9, 10, 12, 13, 21, 14, 15	7(30), 9(40), 13(30)	9
7, 10, 12, 13, 21, 14, 15	7(30), 13(30)	7
10, 12, 13, 21, 14, 15	10(10), 13(30)	13
10, 12, 21, 14, 15	10(20), 14(10)	10
12, 21, 14, 15	12(0), 14(20)	14
12, 21, 15	12(10)	12
21, 15	21(20)	21
15	15(20)	15

Figure 82: Instruction scheduling sequence

to each item in the leader set and the instruction issued for each stage of the entire scheduling process on the sample basic block. Figure 83 shows the rearranged code for the body of the loop after instruction scheduling is applied. The pipeline execution diagram for this code, shown in Figure 84, illustrates the benefits of

```
9           w[12]=W[w[2]+w[3]];
7           w[11]=w[13]*5;                  w[13]
13          w[8]=w[8]+1;
10 {7,9}    w[13]=w[11]+w[12];              w[11],w[12]
14          CC=w[8]?100;
12 {10}     W[w[2]+w[3]]=w[13];
21          w[2]=w[2]+1;
15          PC=CC<0,L60;
```

Figure 83: Loop after instruction scheduling

performing instruction scheduling on this architecture. Unlike the original instruction sequence, the new schedule does not cause the processor to stall at any point in the basic block. As a result, the loop body executes in four fewer processor cycles than the original code. The primary factor behind this improvement is that the instructions that use values have been separated from the instructions that calculate the values or request them from the memory system.

1	2	3	4	5	6	7	8	9	10
9_a	9_b	9_c							
	7_a	7_b	7_c						
		13_a	13_b	13_c					
			10_a	10_b	10_c				
				14_a	14_b	14_c			
					12_a	12_b	12_c		
						21_a	21_b	21_c	
							15_a	15_b	15_c

Figure 84: Pipeline execution diagram for scheduled loop

5.6.2 Filling branch delay slots

Branch delay slots are used in pipelined processors to take advantage of the processor cycles that are otherwise wasted when the pipeline executes a branch instruction. These wasted cycles result from the inability to determine the target of the branch until it reaches one of the later pipeline stages. Figure 85 shows the pipeline execution diagram illustrating the branch instruction from Figure 83. After the first stage of the

1	2	3	4	5	6	7	8
21_a	21_b	21_c					
	15_a	15_b	15_c				
		stall	stall	9_a	9_b	9_c	
					7_a	7_b	7_c

Figure 85: Branch-induced pipeline stall

pipeline fetches branch instruction 15, it stalls because, although it determined that the instruction was a branch, it does not know which instruction to fetch next. After the branch instruction exits the second stage of the pipeline, it has obtained the operands it needs, but has not used them to determine the target of the branch, so the first and second stages of the pipeline stall during the next processor cycle. Finally, when the branch instruction exits the third pipeline stage, the target of the branch is known and the first stage can fetch the instruction at that location during the next processor cycle. A branch delay slot reduces or eliminates the stalls caused by a branch because it allows the pipeline to continue fetching and executing the instructions immediately following the branch instruction while the processor determines the target of the branch. The compiler is expected to fill branch delay slots either with useful instructions, or with place-holding, no-op instructions if useful ones are unavailable. Some architectures provide less-efficient variants of control-flow instructions without delay slots to avoid wasting memory space holding useless instructions in unfilled slots.

Like instruction scheduling, filling branch delay slots is performed by reordering instructions. In *llef*, branch delay slots are filled after all other optimizations, including instruction scheduling, have completed. The same strategy used to implement instruction scheduling is applied to delay slot filling, so that the majority of the algorithm is architecture-independent while the machine description and a few small architecture-specific functions control the filling process.

Figure 86 shows the architecture-independent top level of the branch delay slot filler. This algorithm uses the branch delay slot information added to each branch instruction by the machine description to determine which instructions have branch delay slots and attempt to fill each slot with a nearby instruction depending on the type of the slot. The algorithm can fill three basic types of branch delay slots, although any single slot may be a combination of these types. The first and most common type is a branch delay slot in which the delayed instruction executes regardless of whether the branch is taken or not. In the second type of branch delay slot, delayed instructions are executed only if the branch is taken. In the last type of branch delay slot, delayed instructions are executed only when the branch is not taken. Most architectures provide only the first type of branch delay slots. The SPARC architecture provides both the first and second types of slots. For the sake of brevity, and because the strategy used to fill in the various types of branch delay slots are quite similar, this section will only discuss in detail a portion of the algorithm used to fill the first type of branch delay slots.

```
proc fill_branch_delay_slots is
    for each B where B ∈ CFG do
        I = B.first_instruction
        while I ≠ ∅ do
            for each S where S ∈ I.branch_delay_slots do
                if S.instruction = ∅ ∧ S.both_paths then
                    F = fill_both_paths(I)
                    if F ≠ ∅ then
                        S.instruction = F
                        F.fill_type = BOTH_PATHS
                    endif
                endif
                if S.instruction = ∅ ∧ S.taken_path then
                    F = fill_taken_path( I)
                    if F ≠ ∅ then
                        S.instruction = F
                        F.fill_type = TAKEN_PATHS
                    endif
                endif
                if S.instruction = ∅ ∧ S.untaken_path then
                    F = fill_untaken_path(I)
                    if F ≠ ∅ then
                        S.instruction = F
                        F.fill_type = UNTAKEN_PATHS
                    endif
                endif
            endfor
            I = I.next_instruction
        endwhile
    endfor
endproc
```

Figure 86: Branch delay slot filler algorithm

The algorithm examines every instruction while searching for branch delay slots. While it might seem sufficient to look for slots only on the last instruction in each basic block, which is the only place where branch instructions appear, it is necessary to look through the entire function because some of the instructions that have no effect on the control-flow of the function, such as external function calls, can have branch delay slots. If the target architecture does not use branch delay slots, then the search will end without changing the code.

If an instruction with a branch delay slot is found, then one of the functions whose task it is to locate an appropriate instruction to move into the branch delay slot is called. For branch delay slots that are executed regardless of which path the branch instruction takes, the *fill_both_paths* function is called. If an appropriate instruction is returned, it is moved into the branch delay slot with its "*fill_type*" field set to indicate that it occupies a branch delay slot. Unfilled delay slots are handled by the machine description when the low-level code is translated to assembly code. If the target architecture has a variant of the desired branch instruction that does not have a delay slot, then that variant is used. Otherwise, the machine description emits a no-op instruction to fill the slot.

Figure 87 shows the *fill_both_paths* function. This function searches back through the basic block containing the instruction with the branch delay slot in an attempt to find an appropriate instruction to put in the branch delay slot. The function must ensure that moving the instruction from its present location into the

```
func fill_both_paths(B) is
    D = B.does
    X = B.blocks
    S = B.assigned
    R = B.used
    I = B.prev_instruction
    while I ≠ ∅ do
        if I.fill_type = NONE ∧ D ∩ I.blocks = ∅ ∧ X ∩ I.does = ∅ ∧
         S ∩ I.used = ∅ ∧ R ∩ I.assigned = ∅ ∧ S ∩ I.assigned then
            if can_issue_instruction(I.prev_instruction, I.next_instruction) ∧
             can_issue_instruction(I, B) then
                return I
            endif
        endif
        D = D ∪ I.does
        X = X ∪ I.blocks
        S = S ∪ I.assigned
        R = R ∪ I.used
        I = I.prev_instruction
    endif
    return ∅
endfunc
```

Figure 87: The *fill_both_paths* function

delay slot will not alter the semantics of the code. The algorithm used to ensure this is similar to the one used by the instruction scheduling algorithm to determine which instructions belong in the leader set. In addition, instructions that are already part of a branch delay slot are bypassed and the function does not return an instruction before ensuring that the code sequence that would result after the instruction is moved is valid on architectures that do not check for hazards. The function returns the first instruction that meets these criteria.

In addition to obtaining an instruction from the same basic block that contains the instruction with the branch delay slot, it is also possible, although less desirable, to find possible candidates from the target basic block of the branch or from the fall through block. The reason that these locations are less desirable is that they only improve the performance of the code if the branch is taken or not taken, depending on the source of the instruction. These techniques have been presented in the literature [HENN90], and their corresponding implementations in *llef* are simple extensions of the techniques presented in this section.

The unique feature of the mechanisms used to schedule instructions and fill delay slots in *llef* is that they are largely architecture-independent despite the fact that they perform tasks that are very specific to the target architecture. The method used to order the phases that perform these tasks within the overall structure of the back-end is not unique and does little to mitigate the interactions between them and the rest of the back-end. A better mechanism for performing these code improvements would be more sensitive to the impact that the instructions used to fill delay slots have on the processor's pipeline and would be able to eliminate the false data hazards and storage-related dependencies caused by the local register assigner's tendency to reuse registers. Although these deficiencies merit further study and are closely related to the issues addressed in this treatise, they remain on the list of issues still to be resolved.

5.7 Summary

This chapter has presented some strategies and algorithms used by *llef* to perform a comprehensive set of local and global code-improvement transformations exclusively on low-level code. In some cases, the code-improvement transformations discussed are well-known and widely implemented. What is novel about these algorithms, and makes their inclusion in this dissertation appropriate, is that, in spite of their ability to operate on low-level code that represents the actual instructions supported by the target architecture, they are largely architecture-independent. While the low-level representation allows the code-improvement phases to fully exploit the power of the target architecture and perform architecture-dependent optimizations, the architecture-independent algorithms result in a highly-retargetable system. This strategy is further enhanced by using a carefully chosen set of functions to provide the architecture-specific information needed. This combination of attributes, along with the inclusion of *ease* and the ability to perform register deprivation experiments, makes *llef* a particularly effective tool to use in developing and evaluating strategies that perform register allocation and reduce phase interactions.

CHAPTER 6

IMPROVING THE LOW-LEVEL FRAMEWORK

This chapter describes several strategies that improve the code produced by *llef*. These strategies reduce the severity of the interactions between back-end phases of the compiler. Two of these strategies enhance the ability of the code improvement phases to detect opportunities to improve the code in the presence of register reuse. Another pair of strategies reduce phase interactions by modifying the behavior of the register assignment and allocation algorithms. The register deprivation measurements used to gauge the effectiveness of these strategies show that none of them significantly improve *llef*'s effectiveness. The experience gained from developing and measuring these strategies, however, suggests that significant improvements might be made to the low-level code improvement framework by using a unified approach that simultaneously reduces phase interactions by enhancing the ability of code improvement phases to detect transformations in the presence of register reuse and allocates registers to the set of transformations that are most likely to benefit the code. The impact of this approach on *llef* was so pervasive that the resulting implementation has effectively become an entirely new framework.

6.1 Code improvements

The following sections present a pair of strategies that were developed to improve the ability of two of *llef*'s global code improvement phases to detect opportunities to perform code improvements in the presence of register reuse. These sections show why the original framework failed to capitalize on these opportunities, describe the enhancements that allow the code improvement algorithms to take advantage of these opportunities and present the register deprivation measurements taken to gauge the effectiveness of these enhancements.

6.1.1 Common subexpression elimination

Code sequences often contain redundant expressions that can be removed by performing common subexpression elimination. *llef* uses a global variant of the common subexpression elimination technique used in PO [DAVI84b]. Unlike PO, however, *llef* performs local register assignment before common subexpression elimination. The advantage of this approach is that it avoids over-optimization. Its disadvantage is that it fails to eliminate all local common subexpressions.

Consider the unassigned code shown in Figure 88. Instruction 6 contains a redundant load of global variable `b` that could be eliminated by extending the life of pseudo-register `w[46]` to replace `w[50]` in instruction 7. Like PO and *vpo*, common subexpression elimination in *llef* uses an *equivalence table* to

```
1           w[46]=W[_a];
2   {1}     W[_b]=w[46];                 w[46]
3           w[47]=W[_c];
4           w[48]=W[_d];
5   {3,4}   w[49]=w[47]*w[48];           w[47]w[48]
6           w[50]=W[_b];
7   {5,6}   w[51]=w[49]+w[50];           w[49]w[50]
8   {7}     W[_e]=w[51];                 w[51]
```

Figure 88: Sample machine code sequence

maintain lists of equivalent string patterns. The equivalence table is constructed by examining each source item expression and replacing its components with their oldest equivalent pattern as shown in Figure 89. For example, replacing the components in the source item of instruction 5, "`w[47]*w[48]`", with their oldest equivalent pattern yields "`W[_c]*W[_d]`". This new pattern is the *canonical value* of the source item, and is entered into the equivalence table and associated with destination item, "`w[49]`", to indicate that they are equivalent.

Equivalence table construction and common subexpression elimination operate in unison. Consider the state of the equivalence table as the common subexpression eliminator begins to examine instruction 6. Because the source item of this instruction, "`W[_b]`", is in the equivalence table, the common subexpression eliminator attempts to replace it with all of its equivalent string patterns. The <u>*is_valid_instruction*</u> function is used to determine which of these substitutions are valid and how much they cost to execute on the target architecture. The source is then replaced with the pattern that produces the cheapest valid instruction. Thus, "`W[_b]`" is replaced by "`w[46]`", because it is equivalent and results in a less expensive instruction.

The impact of performing common subexpression elimination on the code sequence in Figure 88 is shown in Figure 90. At first glance, this new sequence appears to be preferable since it lacks the redundant load in the original code sequence. This assessment, however, is not universally true because more registers

```
1   w[46]=W[_a];          “W[_a]” = “w[46]”

2   W[_b}=w[46];          “W[_a]” = “w[46]” = “W[_b]”

3   w[47]=W[_c];          “W[_a]” = “w[46]” = “W[_b]”
                          “W[_c]” = “w[47]”

4   w[48]=W[_d];          “W[_a]” = “w[46]” = “W[_b]”
                          “W[_c]” = “w[47]”
                          “W[_d]” = “w[48]”

5   w[49]=w[47]*w[48];    “W[_a]” = “w[46]” = “W[_b]”
                          “W[_c]” = “w[47]”
                          “W[_d]” = “w[48]”
                          “W[_c]*W[_d]” = “w[49]”

6   w[50]=W[_b];          “W[_a]” = “w[46]” = “W[_b]” = “w[50]”
          ⇓               “W[_c]” = “w[47]”
    w[50]=w[46];          “W[_d]” = “w[48]”
                          “W[_c]*W[_d]” = “w[49]”

7   w[51]=w[49]+w[50];    “W[_a]” = “w[46]” = “W[_b]”
          ⇓               “W[_c]” = “w[47]”
    w[51]=w[49]+w[46];    “W[_d]” = “w[48]”
    Instruction 6 deleted “W[_c]*W[_d]” = “w[49]”
                          “W[_c]*W[_d]+W[_a]” = “w[51]”

8   W[_e]=w[51];          “W[_a]” = “w[46]” = “W[_b]”
                          “W[_c]” = “w[47]”
                          “W[_d]” = “w[48]”
                          “W[_c]*W[_d]” = “w[49]”
                          “W[_c]*W[_d]+W[_a]” = “w[51]” = “W[_e]”
```

Figure 89: Equivalence table construction

are required to perform local register assignment on the improved code sequence than on the original sequence. Figure 91 shows the impact of removing the redundant expression on a target architecture with only two allocable registers. While the original code sequence can be assigned with only two target machine registers as shown in Figure 91(a), the improved code sequence requires three registers to assign without spill

```
1              w[46]=W[_a];
2   {1}        W[_b]=w[46];
3              w[47]=W[_c];
4              w[48]=W[_d];
5   {3,4}      w[49]=w[47]*w[48];                w[47]w[48]
7   {5}        w[51]=w[49]+w[46];                w[49]w[46]
8   {7}        W[_e]=w[51];                      w[51]
```

Figure 90: Code after common subexpression elimination

code. Thus, for this particular target architecture, the local register assigner would insert spill code as shown in Figure 91(b). It is interesting to note that the quality of the code sequence in Figure 91(b) is not as good as the quality of the code produced without common subexpression elimination on most architectures. This is a good example of over-optimization.

```
1              w[0]=W[_a];
2   {1}        W[_b]=w[0];                        w[0]
3              w[0]=W[_c];
4              w[1]=W[_d];
5   {3,4}      w[0]=w[0]*w[1];                    w[1]
6              w[1]=W[_b];
7   {5,6}      w[0]=w[0]+w[1];                    w[1]
8   {7}        W[_e]=w[0];                        w[0]
```

(a)

```
1              w[0]=W[_a];
2   {1}        W[_b]=w[0];
3              w[1]=W[_c];
9              W[_spill]=w[0];                    w[0]
4              w[0]=W[_d];
5   {3,4}      w[1]=w[1]*w[0];                    w[0]
10             w[0]=W[_spill];
7   {5,10}     w[1]=w[1]+w[0];                    w[0]
8   {7}        W[_e]=w[1];                        w[1]
```

(b)

Figure 91: Code sequences after local register assignment

Common subexpression elimination can over-optimize code sequences by increasing the number of simultaneously live registers when it extends the life of expression values. *llef* prevents the common subexpression eliminator from over-optimizing by invoking it after local register assignment so that none of the registers containing redundant values are extended beyond the point where they are reused. Unfortunately, this strategy also has disadvantages. Figure 92 shows the impact that reusing registers has on the common subexpression eliminator. In instruction 3, the reuse of `w[0]` removes its equivalence to the values in `a` and `b`. When instruction 6 is processed, the value in `b` is not available in any registers, so variable `a` is used instead. Thus, performing local register assignment before common subexpression elimination prevents the removal of instruction 6 even if the target architecture provides enough registers to extend the life of the common subexpression value.

```
1   w[0]=W[_a];            "W[_a]" = "w[0]"

2   W[_b]=w[0];            "W[_a]" = "w[0]" = "W[_b]"

3   w[0]=W[_c];            "W[_a]" = "W[_b]"
                           "W[_c]" = "w[0]"

4   w[1]=W[_d];            "W[_a]" = "W[_b]"
                           "W[_c]" = "w[0]"
                           "W[_d]" = "w[1]"

5   w[0]=w[0]*w[1];        "W[_a]" = "W[_b]"
                           "W[_d]" = "w[1]"
                           "W[_c]*W[_d]" = "w[0]"

6   w[1]=W[_b];            "W[_a]" = "W[_b]" = "w[1]"
         ⇓                 "W[_c]*W[_d]" = "w[0]"
    w[1]=W[_a]

7   w[0]=w[0]+w[1];        "W[_a]" = "W[_b]" = "w[1]"
                           "W[_c]*W[_d]+W[_a]" = "w[0]"

8   W[_e]=w[0];            "W[_a]" = "W[_b]" = "w[1]"
                           "W[_c]*W[_d]+W[_a]" = "w[0]" = "W[_e]"
```

Figure 92: Common subexpression elimination after local register assignment

An unfortunate consequence of performing common subexpression elimination after local register assignment is that redundant expressions that are recomputed into the same register that contains the expression value are undetected. Consider the code sequence shown in Figure 93. Instructions 4 and 5 in this

```
1          w[3]=W[_a];
2   {1}    w[3]=w[3]*5;
3   {2}    W[_b]=w[3]+4;                   w[3]
4          w[3]=W[_a];
5   {4}    w[3]=w[3]*5;
6   {5}    W[_c]=w[3]-4;                   w[3]
```

Figure 93: Code sequence containing a recomputation

code sequence recompute a redundant expression value into `w[3]`. This pair of instructions can be removed without changing the semantics of the code. Unfortunately, the common subexpression elimination algorithm fails to detect this redundant expression because the association between the redundant value, "`W[_a]*5+4`", and `w[3]` is forgotten when instruction 4 is processed. This will be true of any redundant expression that requires two or more instructions to compute into the register that contained the redundant expression value.

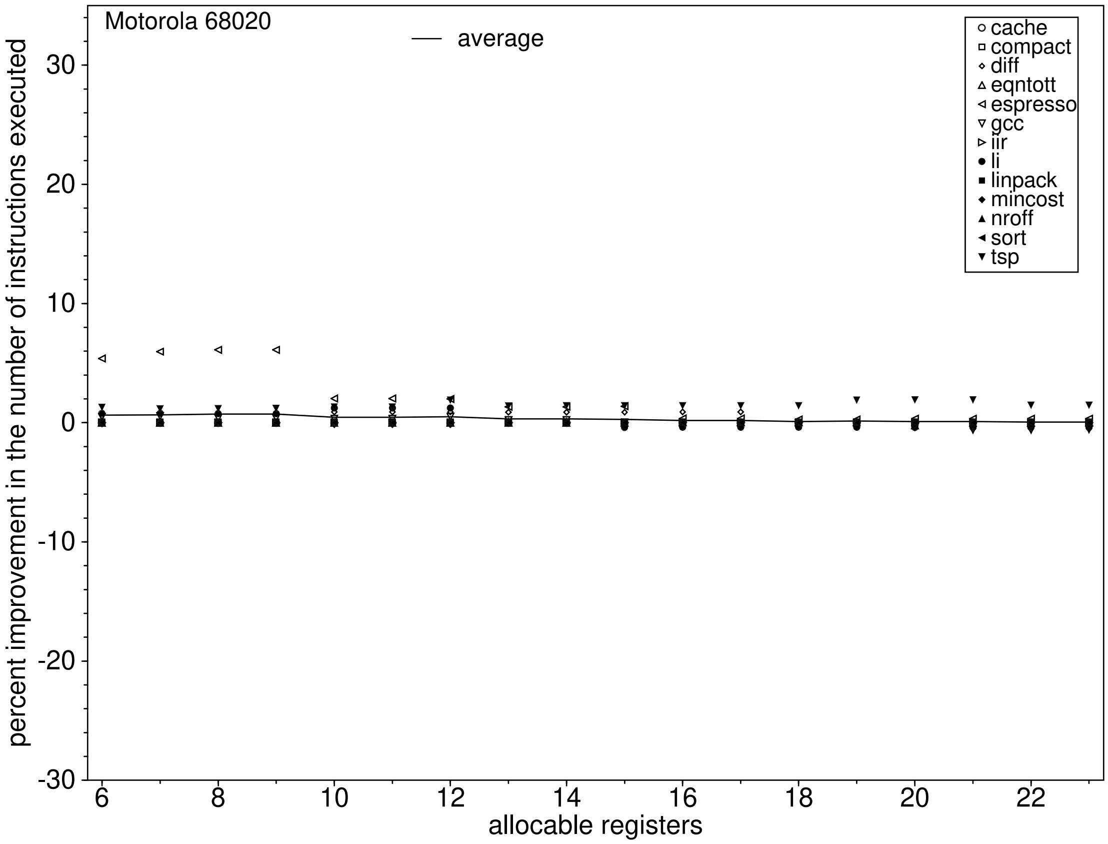
Because these recalculations occur frequently, the common subexpression mechanism was modified to remember the last value in a register when a new assignment is made to it. As shown in Figure 94, this value is retained as long as each successive instruction that modifies the register also references it.

```
1   w[3]=W[_a];              "W[_a]" = "w[3]"

2   w[3]=w[3]*5;             "W[_a]*5" = "w[3]"

3   W[_b]=w[3]+4;            "W[_a]*5" = "w[3]"
                             "W[_a]*5+4" = "W[_b]"

4   w[3]=W[_a];              "W[_a]*5" = "*w[3]"
                             "W[_a]*5+4" = "W[_b]"
                             "W[_a]" = "w[3]"

5   w[3]=w[3]*5;             "W[_a]*5" = "w[3]"
          ⇓                  "W[_a]*5+4" = "W[_b]"
    w[3]=*w[3]
    Instruction 4 deleted
    Instruction 5 deleted

6   W[_c]=w[3]-4;            "W[_a]*5" = "w[3]"
                             "W[_a]*5+4" = "W[_b]"
                             "W[_a]*5-4" = "W[_c]"
```

Figure 94: Recomputation elimination

While processing instruction 4, the special string "`*w[3]`" is added to the equivalence table to indicate that the expression value "`W[_a]*5`" would be available in `w[3]` should all of the instructions that compute its current value be deleted. Thus, when instruction 5 is processed, the common subexpression eliminator determines that a recomputation is taking place because the instruction is about to assign to "`w[3]`" an expression whose value is equivalent to "`*w[3]`". The common subexpression elimination phase is then able to eliminate the recalculation by removing instructions 4 and 5.

Figures 95 and 96 show the impact that recalculation elimination has on the code produced by *llef*. As Figure 95 shows, the impact of eliminating recalculations on the number of instructions executed is generally beneficial. There are, however, instances where eliminating recalculations increases the number of instructions executed. One such notable example is the *iir* benchmark compiled on the MIPS R3000 with



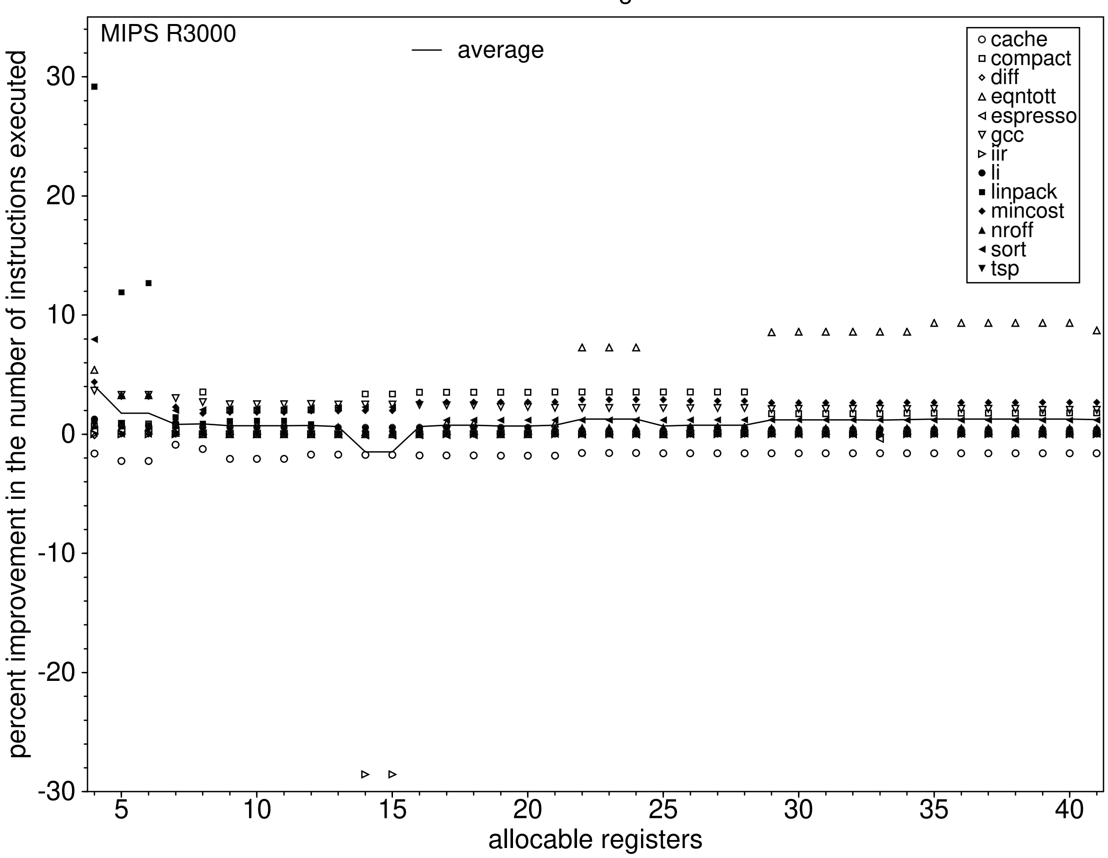
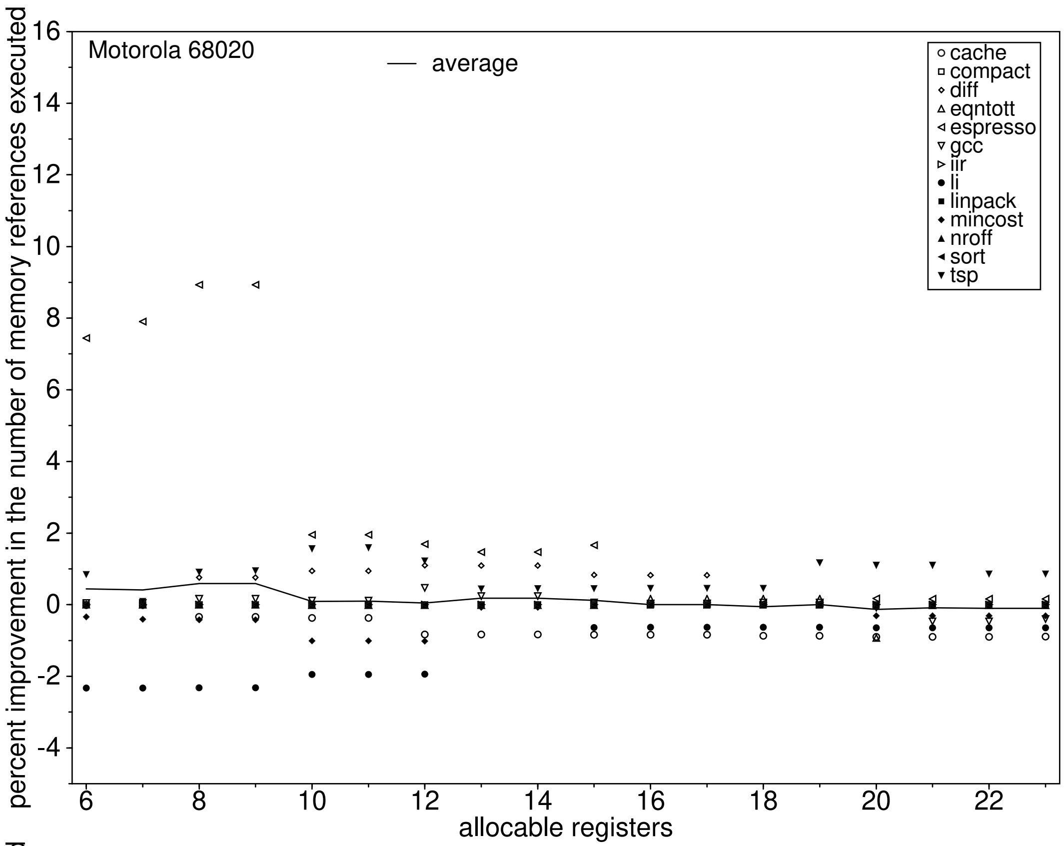


Figure 95: Impact of recalculation elimination (instruction execution counts)



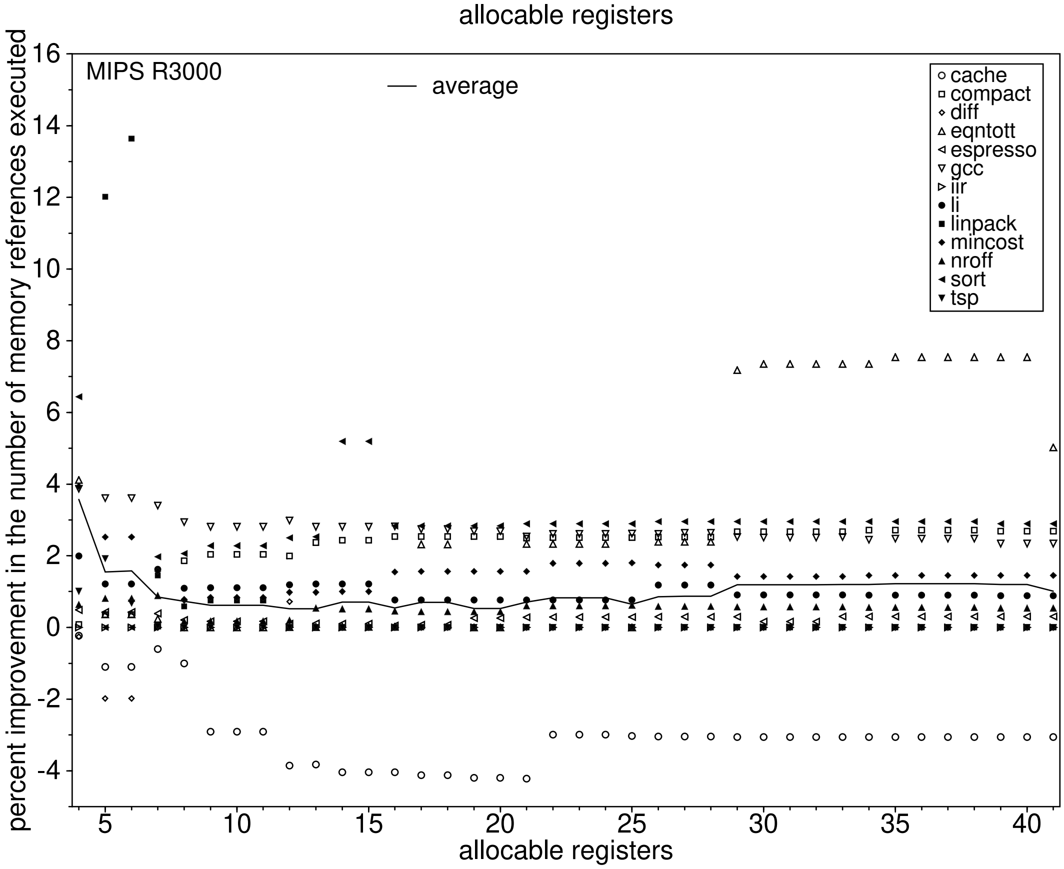


Figure 96: Impact of recalculation elimination (memory reference counts)

either 14 or 15 registers. In this case, eliminating recalculations produces code that executes nearly 30% more instructions. The reason for this unexpected behavior is that eliminating a recalculation extended the life of a register so that it not available for use by a more beneficial loop strength reduction transformation. This example illustrates the importance of allocating the register resources to the transformations that most benefit the quality of the code produced.

6.1.2 Loop-invariant code motion

Like common subexpression elimination, performing loop-invariant code motion can result in over-optimization by increasing the number of register values live in a function. Additionally, the tendency of the local register assigner to reuse registers can prevent the loop-invariant detection algorithm from discovering all of the loop-invariant expressions in a loop. To increase the effectiveness of loop-invariant code motion, an additional mechanism was developed to detect and move the loop-invariant expressions assigned to reused registers out of the loops.

Consider the loop code fragment shown in Figure 97. The loop-invariant register detection algorithm described in Section 5.5.1 does not identify `w[8]` as a loop-invariant register because it is assigned more than one value in the loop. Thus, after all of the detected loop-invariant register assignments are moved to the preheader, instructions 1 and 2 remain inside the loop. At this point, the algorithm shown in Figure 98 can be invoked to move additional loop-invariant expressions out of the loop.

```
    L87:
                ...
1               w[8]=HI[_global];
2   {1}         w[8]=W[w[8]+LO[_global]];
                ...
                PC=L87;
```

Figure 97: Loop containing a loop-invariant expression assigned to a reused register

This algorithm individually examines every expression computed by the loop. When a loop-invariant expression is found, the algorithm attempts to find an appropriate register to hold the expression value. Because a loop-invariant expression can appear more than once in a loop, an attempt is made to find a register that already contains the loop-invariant expression value. If the value is not available in a register, an attempt is made to allocate a new register that can be used across the entire loop. If an appropriate register is

```
func auxiliary_code_motion(L) is
    for each B where B ∈ L.blocks_in_loop do
        I = B.first_instruction
        while I ≠ ∅ do
            for each V where V ∈ I.values_assigned do
                if is_invariant_expression(I, V, L.memory_writes) then
                    E = get_expression(I, V)
                    R = get_appropriate_register(E, L.available_registers)
                    if R ≠ ∅ then
                        C = I
                        replace_expression(C, V, R)
                        if is_valid_instruction(C) ∧ C.cost ≤ I.cost then
                            I = C
                            if R.loop_invariant ≠ YES then
                                R.loop_invariant = YES
                                R.invariant_value = insert_assignment(L.preheader, R, E)
                            endif
                        else
                            L.available_registers = L.available_registers - R
                        endif
                    endif
                endif
            endfor
            I = I.next_instruction
        endwhile
    endfor
endproc

func get_appropriate_register(E, A) is
    for each R where R ∈ REGISTERS ∧ R.loop_invariant = YES ∧ R.invariant_value ≠ ⊥ do
        if get_expression(R.invariant_value, R) = E then
            return R
        endif
    endfor
    return assign_target_register(type_of_expression(E), A)
endfunc
```

Figure 98: Loop-invariant code motion enhancement algorithm

found, the algorithm allocates it, insert code in the loop's preheader to compute the loop-invariant expression value into it and replaces the reference to the loop-invariant expression value within the loop with a reference to the register.

When this algorithm is applied to the loop in Figure 97, the loop-invariant expression assigned to `w[8]` in instruction 1 is found to be loop-invariant and, as shown in Figure 99, register `w[10]` is allocated to contain the value of the expression over the life of the loop. Instruction 3 is added to the loop's preheader to compute the loop-invariant expression value into `w[10]` before entering the loop. The reference to the loop invariant expression value in instruction 1 is replaced with the newly allocated register item. Because

```
3             w[10]=HI[_global];
    L87:
              ...
1             w[8]=w[10];
2   {1}       w[8]=W[w[8]+LO[_global]];
              ...
```

Figure 99: Loop after the enhanced loop-invariant code motion algorithm is applied

these transformations do not produce the best possible code, *llef* relies on phase iteration to propagate `w[10]` throughout the loop and remove any orphaned expressions that were left behind by the loop-invariant code motion transformation. When phase iteration re-invokes the instruction selection phase, the code in Figure 99 will be further improved as shown in Figure 100.

```
3             w[10]=HI[_global];
    L87:
              ...
2             w[8]=W[w[10]+LO[_global]];
              ...
```

Figure 100: Loop after phase iteration

When the loop-invariant code motion phase is re-invoked, `w[8]` may be found to be a loop-invariant register[1] and its assignment in instruction 2 moved to the preheader. If it is loop-invariant, the enhanced loop-invariant code motion algorithm will allocate another register so that the loop-invariant expression is calculated outside the loop as shown in Figure 101. The code improvement process continues with another instruction selection, dead variable elimination and loop-invariant code motion pass until no new transformations are applied.

```
3             w[10]=HI[_global];
4             w[11]=W[w[10]+LO[_global]];
    L87:
              ...
2             w[8]=w[11];
              ...
```

Figure 101: Loop after the re-invocation of loop-invariant code motion

[1] The register will be loop-invariant only if there are no memory writes within the loop; otherwise, the memory reference in the source expression is not considered to be loop-invariant.

Because the algorithm only considers complete source expressions it may miss some loop-invariant items. Although the algorithm could be modified to examine parts of source expressions, this modification would greatly increase the execution time of the algorithm. Thus, the only individual items considered are memory reads on loops that contain no memory writes because the benefits of replacing memory reads with register references generally outweigh the additional processing time required.

Figures 102 and 103 show the impact of the enhanced loop-invariant code motion algorithm on the code produced by *llef*. These results reveal how profoundly some code improvements affect particular codes while having almost no effect on most other codes. For example, the memory reference counts in Figure 103 show that the transformations performed by the enhanced loop-invariant code motion algorithm substantially decrease the number of memory reference executed by the *eqntott* benchmark while having little effect on the remaining benchmarks except for the *iir* benchmark on the Motorola 68020 and the *nroff* benchmark. In the case of *eqntott*, the benefits come from moving a pair of loop-invariant memory references out of a loop that accounts for over 80% of the program's execution time. For the *iir* kernel on the CISC architecture, a relatively minor improvement made by the enhanced loop-invariant code motion transformation when 18 or more registers are available enables the recurrence elimination phase to make a significant improvement that would have otherwise gone undone. Because the registers in this benchmark are reused in a slightly different way on the RISC architecture, the basic loop-invariant code motion algorithm was able to perform this crucial intermediate step on the MIPS and prevented the kinds of improvements seen on the CISC architecture for this benchmark from occurring on the RISC architecture. Finally, enhanced loop-invariant code motion had a detrimental impact on *nroff* because it performed code motion transformations on the programs main input loop. Unfortunately, because this loop iterates only while skipping over comments and line breaks in the input file, the loop body rarely executes more than once whenever the loop is executed. Consequently, the transformations performed by the enhanced loop-invariant code motion algorithm have the undesired effect of increasing the number of instructions and memory references executed by *nroff*.

6.2 Register allocation

The following sections examine the various register allocation strategies integrated into *llef*. The purpose of these sections is to compare these allocation strategies in order to determine the feasibility of improving the effectiveness of retargetable register allocation using a few simple techniques.

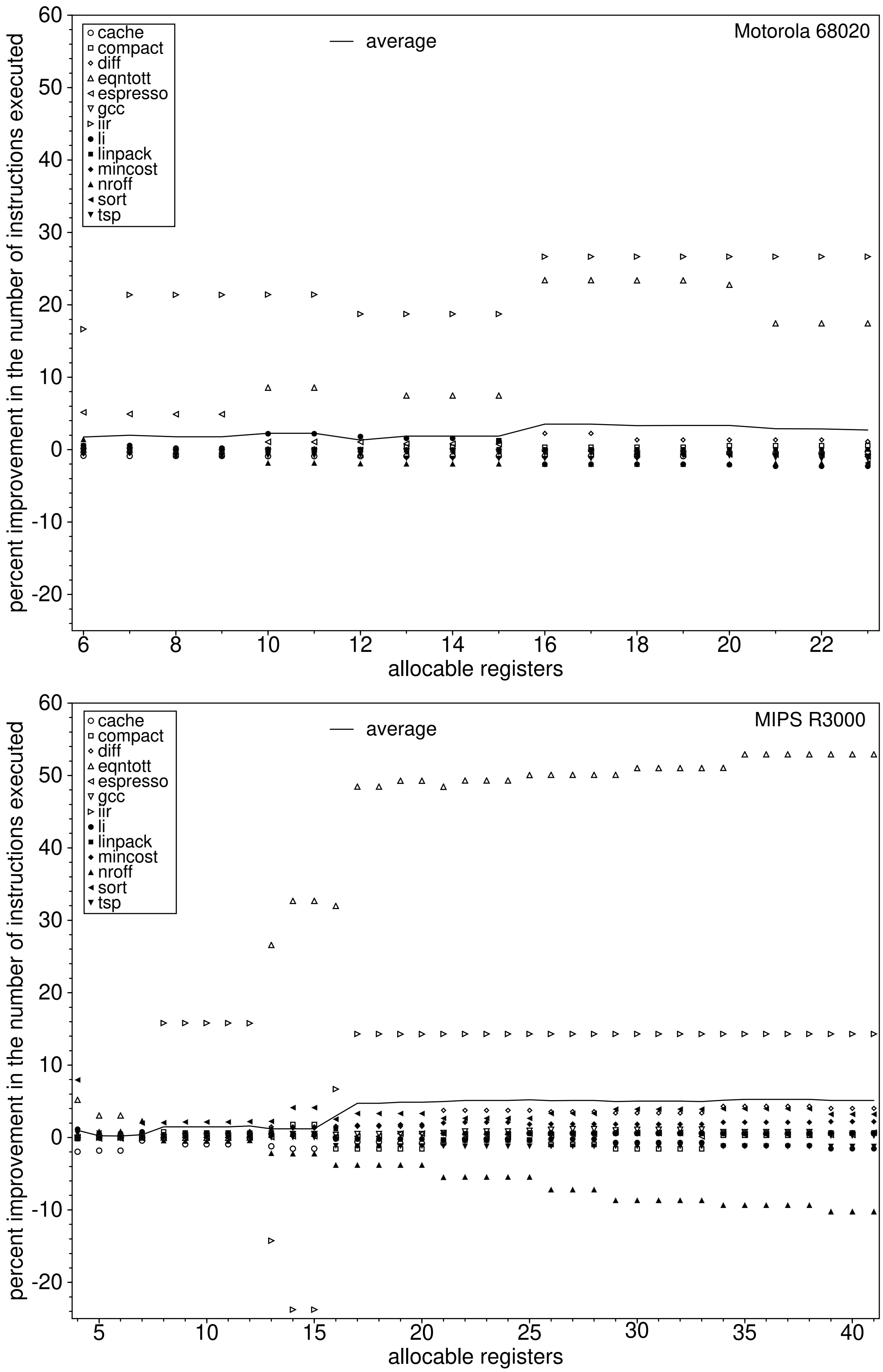


Figure 102: Enhanced loop-invariant code motion results (instruction execution counts)

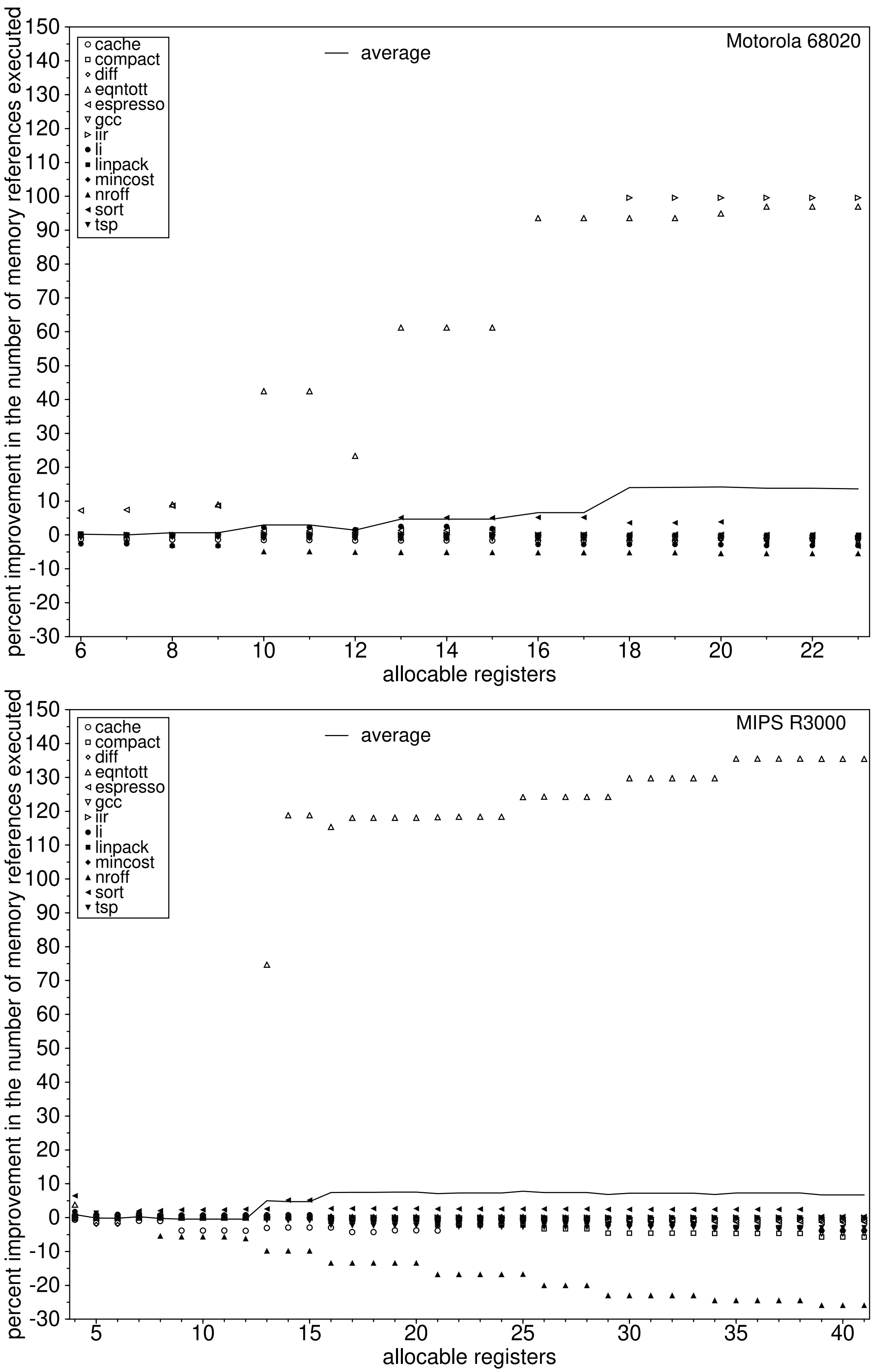


Figure 103: Enhanced loop-invariant code motion results (memory reference counts)

6.2.1 Free-for-all

The description of *llef* in Chapter 5 presented the standard register allocation strategy used by the framework. This strategy uses pseudo-registers only during the initial back-end phases, reuses registers during local register assignment to increase the number of registers available to the global code improvement phases, performs global-scope variable promotion using a graph-coloring register allocator and allocates registers to code improvement transformations on a first-come first-served basis. Because this strategy imposes no limits on the number of target architecture registers that any single code improvement phase can obtain, it is called the *free-for-all* strategy.

Even though the free-for-all strategy imposes few limits on the code improvements phases, it avoids over-optimization and easily outperforms the much simpler allocation strategy used by *vpo*. Figures 104 and 105 show how *llef*'s global-scope graph coloring register allocation compares to the simple register allocator used in *vpo*. The results indicate that the graph-coloring register allocation strategy used by *llef* does not suffer the effects of over-optimization typical of PL.8-style register allocators when few allocable registers are available.

6.2.2 Round-robin

Because register reuse significantly impacts *llef*'s common subexpression elimination and loop-invariant code motion algorithms, it may be advantageous to use a register allocation strategy that decreases the degree to which registers are reused. One such strategy, suggested by Hennesy and Gross, uses a *round-robin* allocation technique to cycle through the allocable registers and increase the distance between register reuses [HENN83]. Goodman and Hsu compared this approach against a traditional allocator to measure its impact on post-pass instruction scheduling. Their results show that the round-robin strategy usually allows better schedules to be produced, although there are cases where the schedules produced while using a traditional allocator outperform those produced with the round-robin allocator [GOOD88].

To illustrate how round-robin register allocation enhances the ability of the common subexpression eliminator to improve the code, consider the unassigned fragment of code shown in Figure 106. Instruction 4 of this code fragment computes a redundant subexpression that should be removed by the common subexpression eliminator. When the code is passed through a standard local register assigner that reuses registers as soon as they are released, the code fragment will be modified as shown in Figure 107.

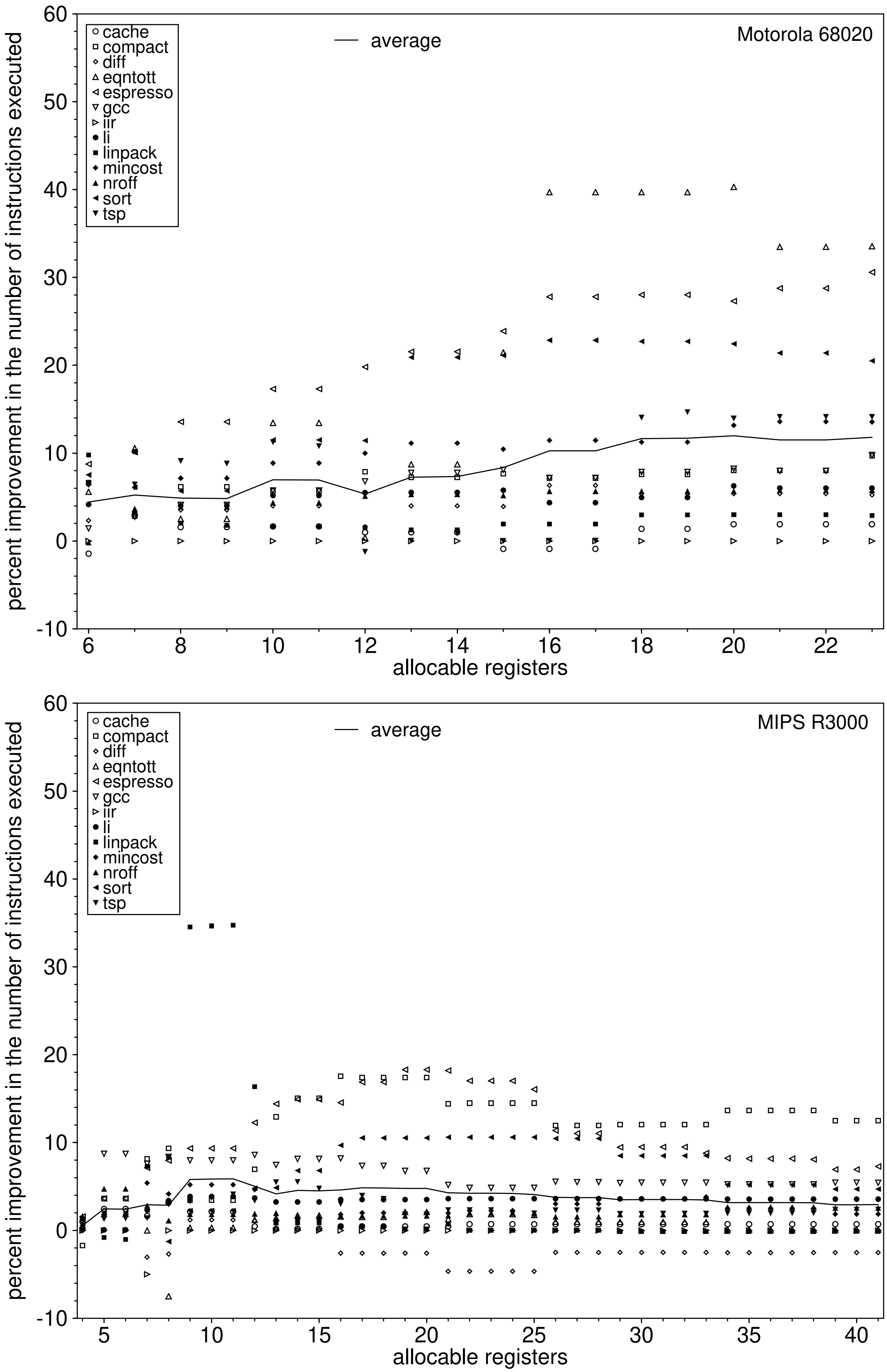


Figure 104: Performance of coloring vs. simple global register allocation (instruction counts)

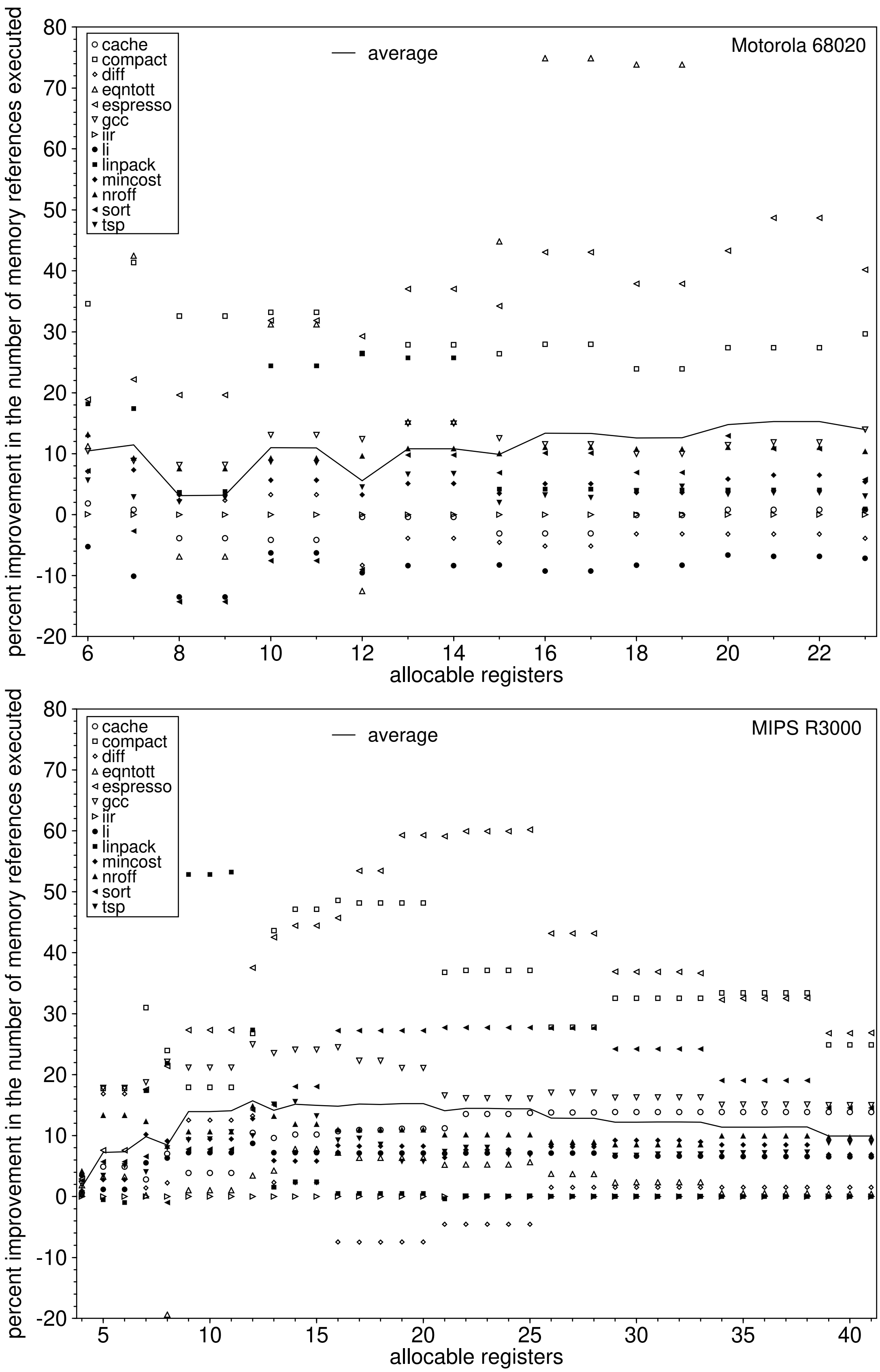


Figure 105: Performance of coloring vs. simple register allocation (memory reference counts)

```
1           w[35]=W[_i]*4;
2   {1}     w[36]=w[35]+_a;                w[35]
3   {2}     w[37]=W[w[36]];                w[36]
4           w[38]=W[_i]*4;
5   {4}     w[39]=w[38]+_a+4;              w[38]
6   {5}     w[40]=W[w[39]];                w[39]
7   {3,6}   W[w[37]]=w[40];                w[37],w[40]
```

Figure 106: Fragment of code before local register allocation

Unfortunately, *llef*'s common subexpression eliminator, even with the recalculation algorithm described in Section 6.1.1, will not remove the redundant subexpression because the equivalence table will not contain an item equivalent to "`W[_i]*4`" when instruction 4 is reached. If the code fragment is assigned using a round-

```
1           w[0]=W[_i]*4;
2   {1}     w[0]=w[0]+_a;
3   {2}     w[0]=W[w[0]];
4           w[1]=W[_i]*4;
5   {4}     w[1]=w[1]+_a+4;
6   {5}     w[1]=W[w[1]];
7   {3,6}   W[w[0]]=w[1];                  w[0],w[1]
```

Figure 107: Code fragment after traditional local register assignment

robin strategy that cycles through all of the volatile registers instead of immediately reusing freed registers, the resulting code fragment might resemble the code shown in Figure 108. In this example, the target machine provides four volatile registers and the round-robin allocator uses the entire set of volatile registers before attempting to reuse any particular register. Unlike the code shown in Figure 107, the common subexpression

```
1           w[0]=W[_i]*4;
2   {1}     w[1]=w[0]+_a;                  w[0]
3   {2}     w[2]=W[w[1]];                  w[1]
4           w[3]=W[_i]*4;
5   {4}     w[0]=w[3]+_a+4;                w[3]
6   {5}     w[1]=W[w[0]];                  w[0]
7   {3,6}   W[w[2]]=w[1];                  w[2],w[1]
```

Figure 108: Code fragment after round-robin local register assignment

eliminator is able to remove instruction 4 in Figure 108 because the equivalence table will indicate that `w[0]` contains the value computed by the source item in instruction 4. Thus, applying common subexpression elimination on the code fragment in Figure 108 produces the code shown in Figure 109, which contains one fewer instruction than the code in Figure 107.

```
1            w[0]=W[_i]*4;
2   {1}      w[1]=w[0]+_a;
3   {2}      w[2]=W[w[1]];                     w[1]
5            w[0]=w[0]+_a+4;
6   {5}      w[1]=W[w[0]];                     w[0]
7   {3,6}    W[w[2]]=w[1];                     w[2],w[1]
```

Figure 109: Code fragment after round-robin assignment and common subexpression elimination

Round-robin local register assignment has been implemented on *llef* to determine its impact on the back-end. The round-robin local register assigner remembers the last register assigned for each allocation type and attempts to allocate the next volatile register in the preference list instead of the first one. Because the round-robin assigner favors the volatile registers and assigns from the same set of registers as the traditional assigner, one would assume that, without global code improvements, the quality of the code produced with the round-robin allocator would be identical to that of the code produced using the traditional assigner. As the results in Figures 110 and 111 show, however, this is not the case. These figures compare the benefits of performing round-robin local register assignment with the traditional method without global code improvements. With few exceptions, the quality of the code produced using the round-robin algorithm is no better than the quality of the code produced by the traditional algorithm. Most of the differences on the CISC can be attributed to its two-address instruction set. For example, consider the instruction sequence shown in Figure 112(a). When local register assignment is performed on this code using the traditional local register

```
1            w[43]=w[41];                      w[41]
2            w[43]=w[43]+1;
```

(a)

```
1            w[0]=w[0];
2            w[0]=w[0]+1;
```

(b)

```
1            w[1]=w[0];                        w[1]
2            w[1]=w[1]+1;
```

(c)

Figure 112: The effect of round-robin allocation on a two-address instruction set

allocator, the same register is assigned to w[41] and w[43] as shown in Figure 112(b), which allows a subsequent code improvement phase to eliminate instruction 1. If a round-robin allocator is used, however,

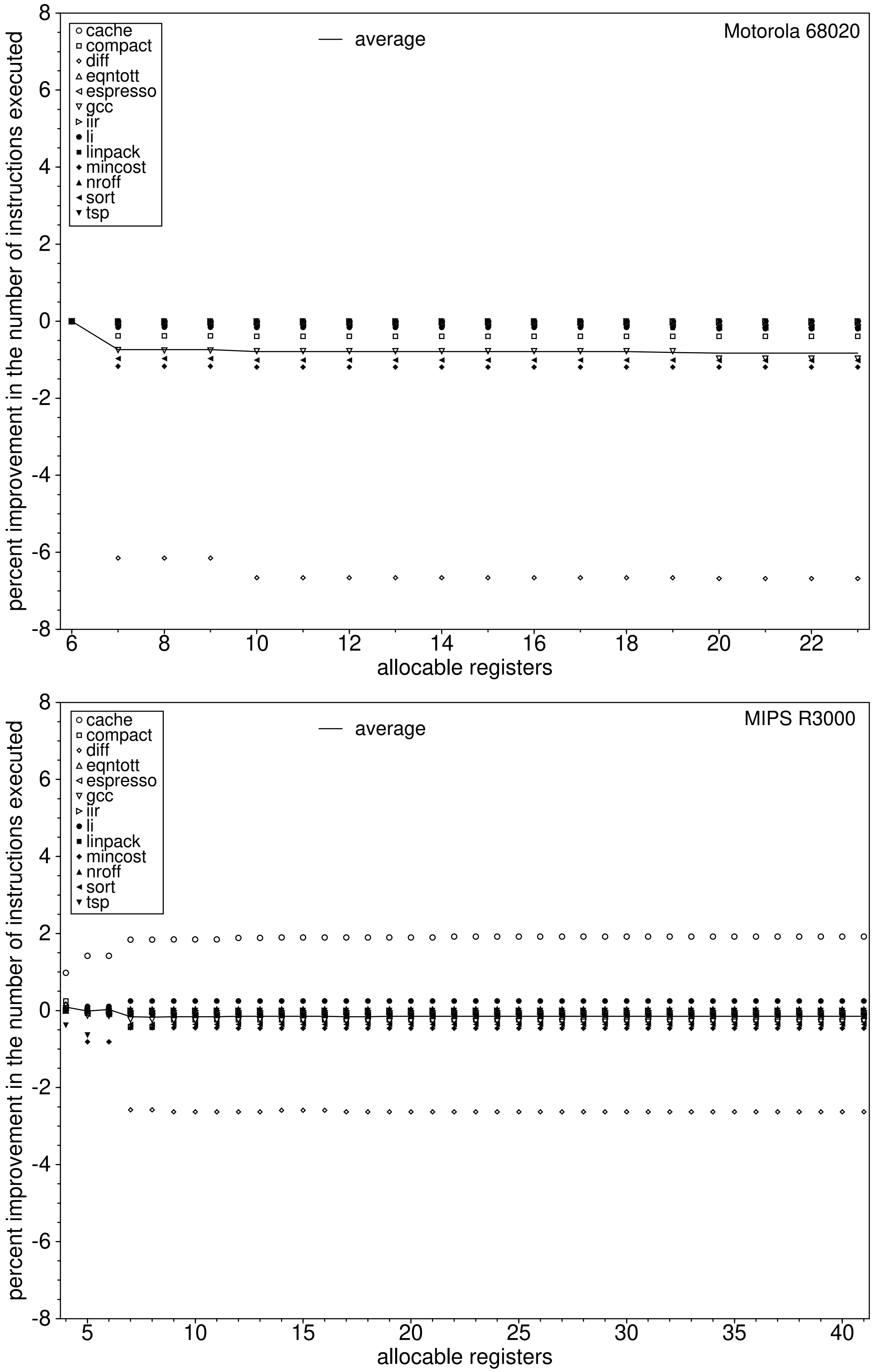


Figure 110: Round-robin allocation performance results (instruction execution counts)

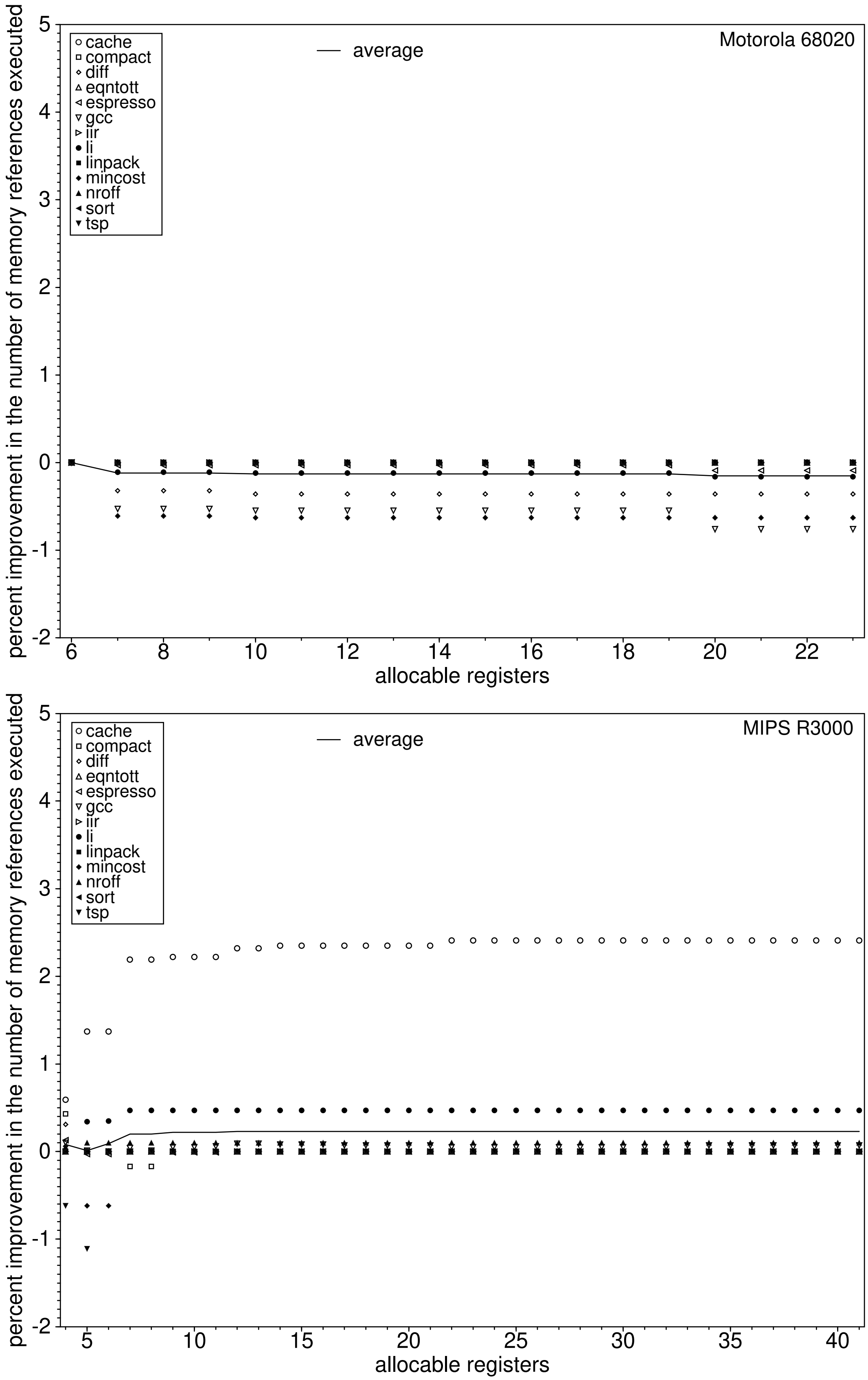


Figure 111: Round-robin allocation performance results (memory reference counts)

the code is assigned as shown in Figure 112(c), and instruction 1 cannot be merged with instruction 2 because the same register must be used in both the source and the destination of the two-address addition operation. On the RISC architecture, the differences are caused by the interaction between the local register allocator and the calling convention. Because registers are used to pass function parameters, the order in which registers are assigned affects the number of extra register-to-register transfer instructions required to ensure that parameter values reside in the appropriate register when an external function call takes place.

Round-robin local register assignment allows the common subexpression elimination and loop-invariant code motion phases to detect expressions that would otherwise be missed because of register reuse. Figures 113 and 114 show that the common subexpression eliminator is able to produce better code on average with round-robin allocation than with free-for-all allocation. Looking at these results alone, one might assume that the combined impact of round-robin register allocation and global code improvements would be positive. The register deprivation measurement results shown in Figures 115 and 116, however, show that the overall impact of the round-robin strategy is generally not beneficial. These results are caused by the interaction between the local round-robin allocator and the variable promotion, loop strength reduction and induction variable elimination code improvements. These code improvements are affected by the round-robin allocation strategy because cycling through the volatile registers increases the number of registers that will be used across any arbitrary span of the CFG. This effectively decreases the number of registers available to perform global code improvement transformations like variable promotion, loop strength reduction and induction variable elimination. Combined with the negative impact of local register assignment, these factors frequently overcome any of the added improvements obtained by the common subexpression elimination and the loop-invariant code motion phases.

6.2.3 Bounded register

A disadvantage of the free-for-all allocation strategy is that the code improvement phases invoked first can monopolize the allocable registers and prevent later phases from performing transformations. Thus, it is possible for the loop code motion phase to use all or most of the allocable registers to perform relatively unprofitable transformations to a loop and prevent a subsequent phase, like loop strength reduction, from applying a transformation that would significantly improve the loop. These frequency in which these scenarios occur increases as the number of allocable registers on the target architecture decreases.

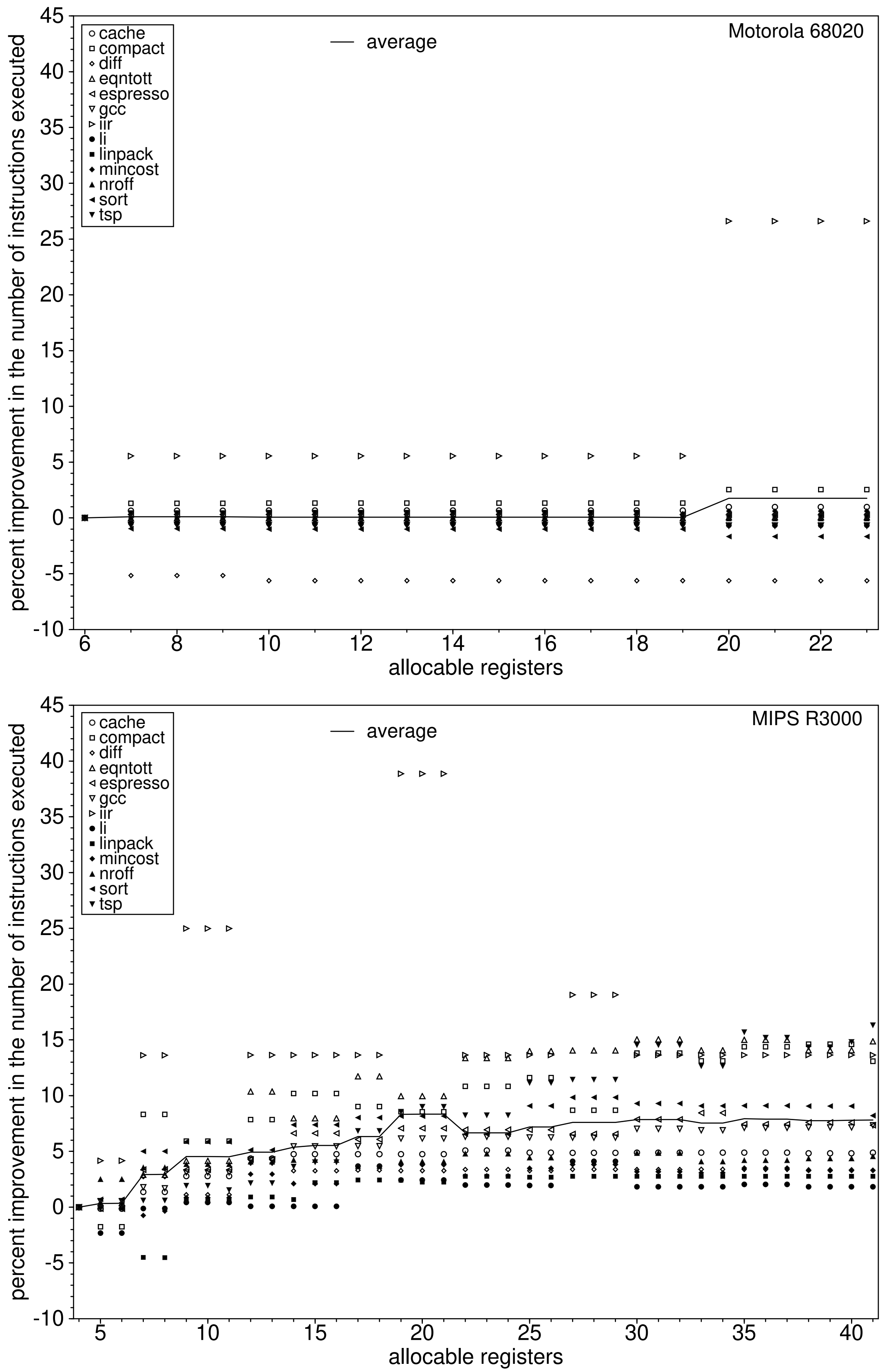
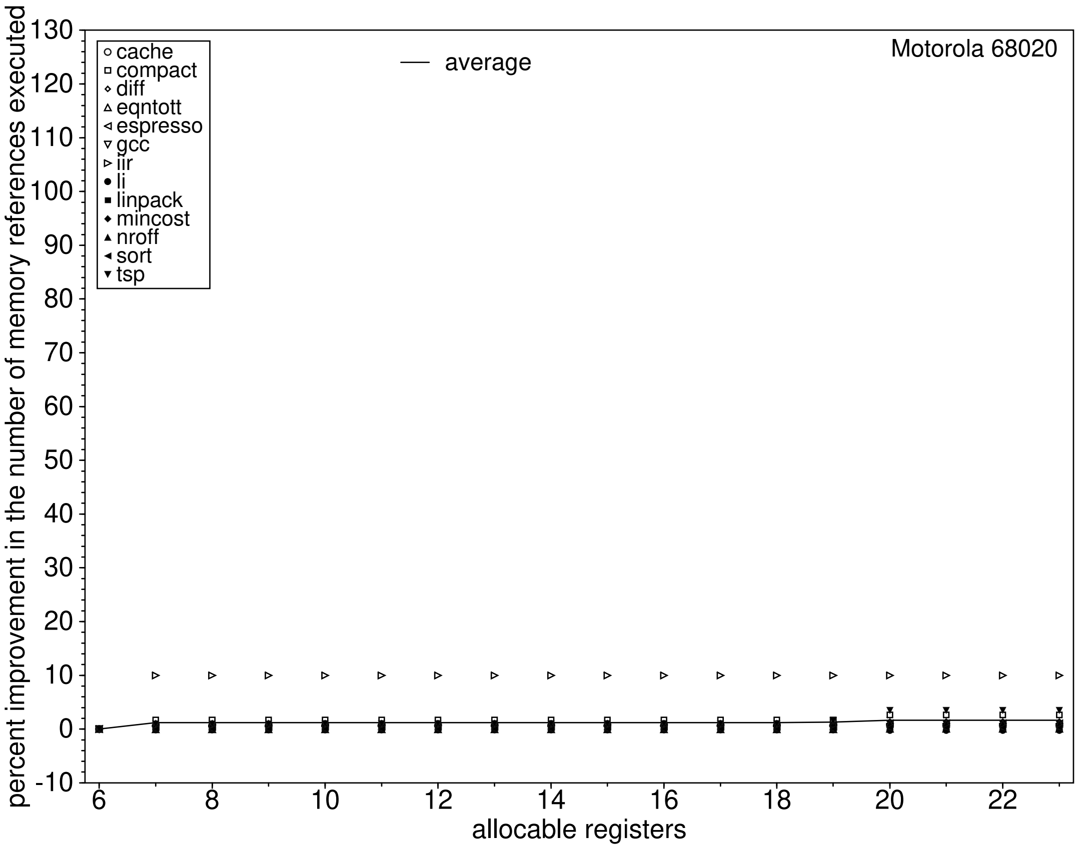


Figure 113: Effect of round-robin allocation on CSE (instruction execution counts)



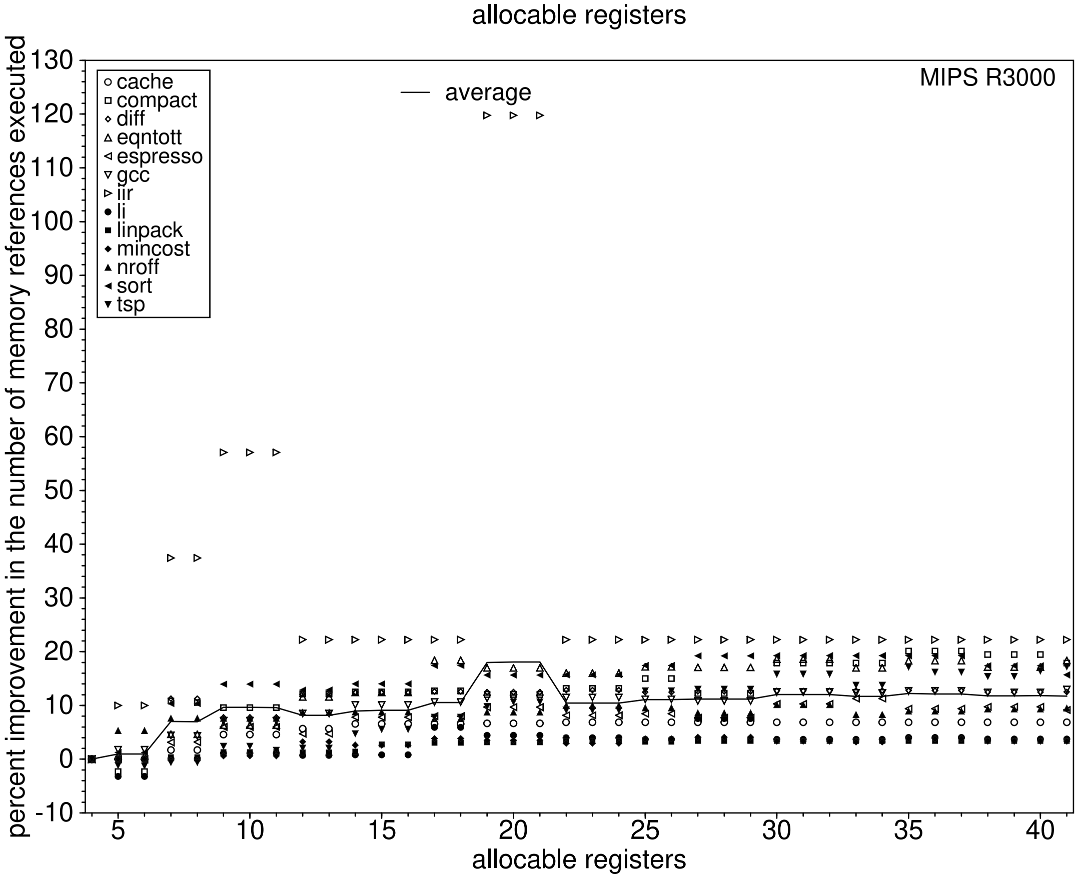
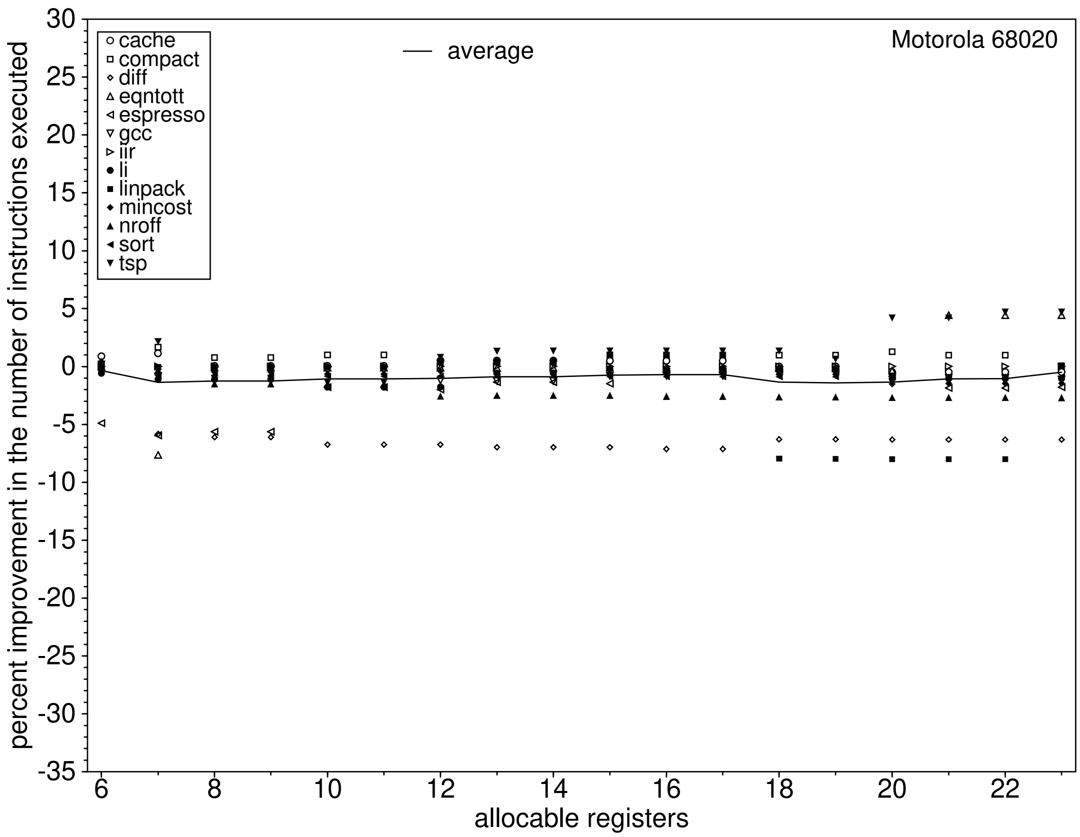


Figure 114: Effect of round-robin allocation on CSE (memory reference counts)



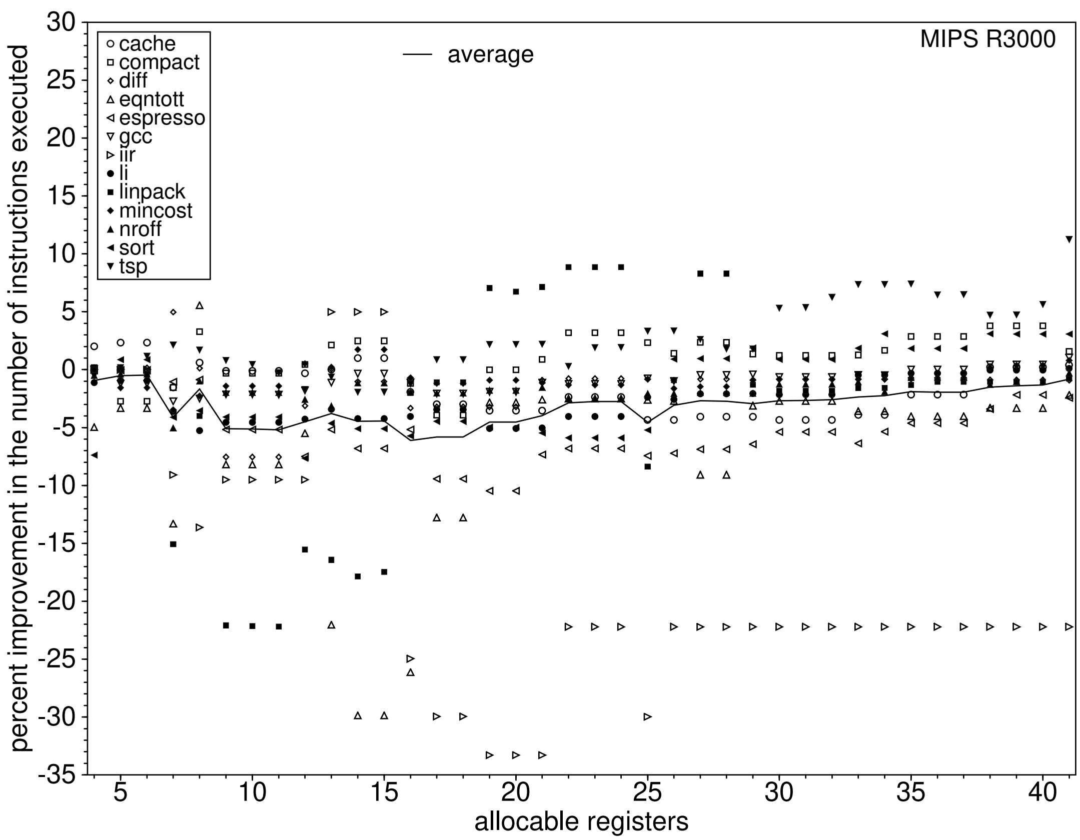
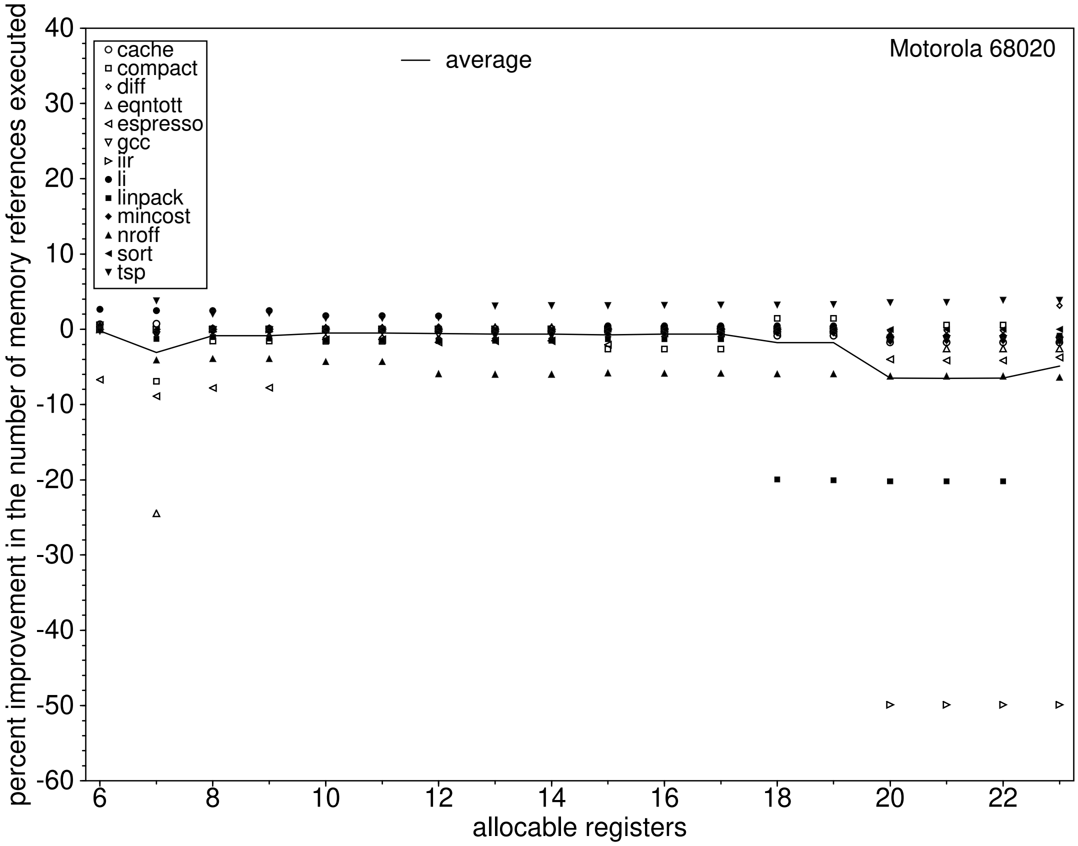


Figure 115: Round-robin allocation performance results (instruction execution counts)



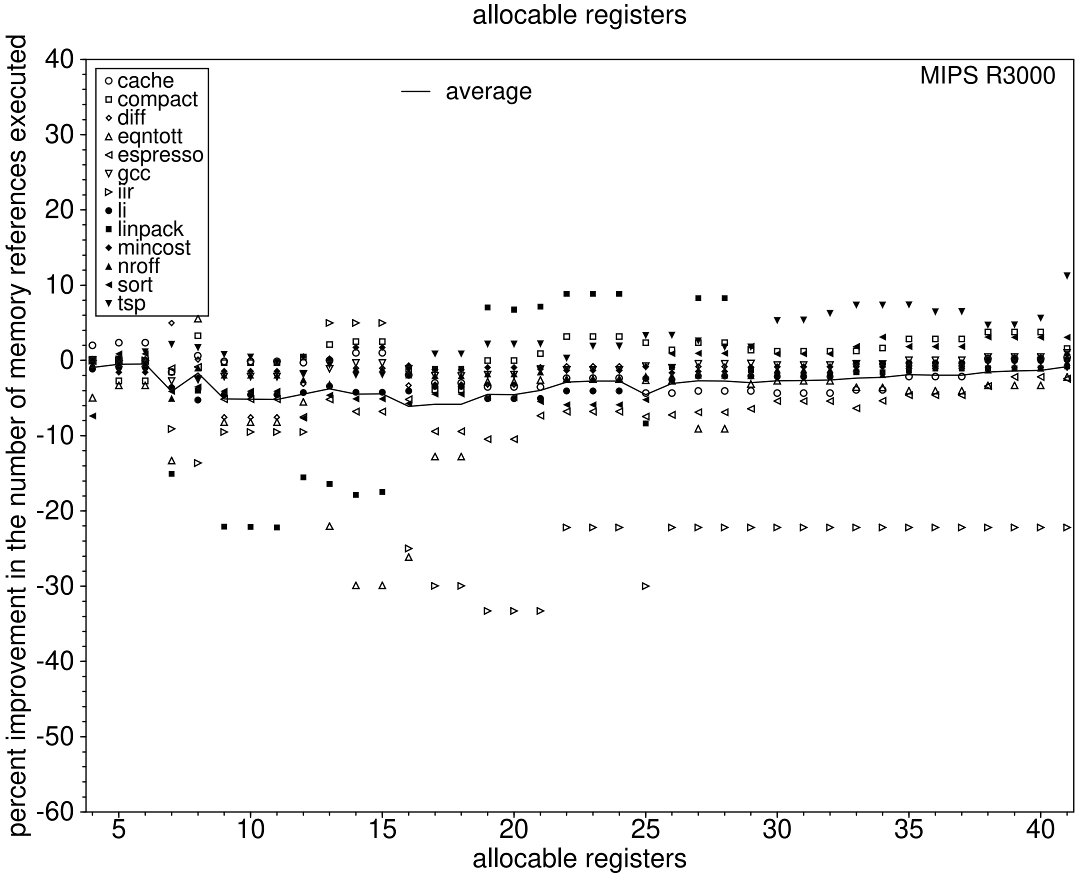


Figure 116: Round-robin allocation performance results (memory reference counts)

To prevent any single code improvement phase from monopolizing the registers, a limit can be placed on the number of registers that a phase can obtain during any single invocation. Combined with phase iteration, this *bounded-register* strategy increases the chances that each code improvement phase can obtain at least a few registers in its first invocation.

To measure the effect of the bounded-register allocation strategy on the quality of the code produced by *llef*, a bounded-register allocator has been incorporated into the framework. This allocator keeps track of the number of registers allocated by any of the code improvement phases to ensure that no code improvement phase obtains more than a predetermined number of registers each time it is invoked. To simplify the strategy, the same limit applies to all of the code improvement phases.

Using the bounded-register strategy is complicated by the fact that the most effective limit value can be found only through experimentation. There usually exists a limit that allows the bounded-register strategy to outperform the free-for-all strategy. Unfortunately, this limit value varies across architectures and can be affected by even small changes in the allocable register set. The substantial effort needed to determine the most appropriate limit value for any given architecture profoundly reduces the retargetability of the bounded-register strategy.

Figure 117 shows the average effectiveness of the bounded register algorithm relative to the free-for-all algorithm on the Motorola 68020 when the limit value is varied between two and eight, inclusively. The first number in each entry indicates the average percent improvement in the number of instructions executed. The second number is the average improvement in the number of memory references executed. The most effective limit value for each register deprivation trial is shaded. The memory reference counts values are weighed three times more heavily than the instruction execution counts when determining the most effective limit value for each trial. Figure 118 shows the results obtained for the MIPS R3000. On this architecture, all of the limit values tried for the 4 and 7 register trials produced worse code that the code produced with the free-for-all allocator.

Figures 120 and 121 show the register deprivation measurement results obtained by choosing the most effective limit value for each of the trials. In nearly all circumstances, the percent improvement provided by the bounded-register algorithm is modest. In several cases, the results show that limiting the number of

Registers	2	3	4	5	6	7	8
6	0.01 / 0.00	0.01 / 0.00	0.00 / 0.00	0.00 / 0.00	0.00 / 0.00	0.00 / 0.00	0.00 / 0.00
7	0.15 / 0.13	0.11 / 0.11	0.10 / 0.11	0.10 / 0.11	0.10 / 0.11	0.10 / 0.11	0.10 / 0.11
8	0.31 /-0.26	0.27 /-0.29	0.22 /-0.36	0.22 /-0.36	0.22 /-0.36	0.18 / 0.23	0.18 / 0.23
9	0.31 /-0.26	0.27 /-0.29	0.22 /-0.36	0.22 /-0.36	0.22 /-0.36	0.18 / 0.23	0.18 / 0.23
10	-0.30 /-2.12	-0.26 /-1.77	-0.31 /-1.85	-0.31 /-1.86	-0.31 /-1.86	0.26 / 0.38	0.26 / 0.38
11	-0.22 /-2.17	-0.26 /-1.78	-0.31 /-1.85	-0.31 /-1.86	-0.31 /-1.86	0.26 / 0.38	0.26 / 0.38
12	1.19 / 0.63	1.16 / 0.99	1.11 / 0.98	1.10 / 1.01	1.10 / 1.01	1.01 / 1.63	0.27 / 0.39
13	1.73 / 2.12	1.78 / 2.23	1.85 / 2.24	1.84 / 2.21	1.85 / 2.22	1.09 / 1.72	0.30 / 0.02
14	1.66 / 2.12	1.71 / 2.23	1.85 / 2.24	1.84 / 2.21	1.85 / 2.22	1.09 / 1.72	0.30 / 0.02
15	1.87 / 2.51	1.92 / 2.57	1.86 / 2.27	1.85 / 2.24	1.85 / 2.25	1.79 / 3.71	0.30 / 0.03
16	0.65 / 1.37	0.51 / 1.04	0.49 / 0.74	0.47 / 0.71	0.55 / 0.83	1.01 / 1.99	0.27 / 0.02
17	0.65 / 1.37	0.51 / 1.04	0.49 / 0.74	0.47 / 0.71	0.55 / 0.83	1.01 / 1.99	0.27 / 0.02
18	-0.18 /-4.52	-0.03 /-2.84	-0.13 /-1.04	-0.13 /-1.04	-0.08 /-0.99	0.37 / 0.19	0.28 / 0.03
19	-0.18 /-4.52	-0.03 /-2.84	-0.13 /-1.04	-0.13 /-1.04	-0.08 /-0.99	0.37 / 0.19	0.28 / 0.03
20	0.02 /-4.37	-0.01 /-3.33	0.08 /-0.93	0.37 /-0.19	0.01 /-1.02	0.46 / 0.21	0.29 / 0.03
21	0.45 /-3.79	0.34 /-2.72	0.38 /-0.36	0.70 / 0.27	0.33 /-0.43	0.37 / 0.69	0.32 / 0.51
22	0.45 /-3.79	0.34 /-2.72	0.34 /-0.36	0.70 / 0.27	0.33 /-0.43	0.33 / 0.69	0.28 / 0.51
23	0.50 /-3.58	0.48 /-2.54	0.48 /-0.16	0.57 /-0.25	0.47 /-0.24	0.47 / 0.87	0.40 / 0.67

Figure 117: Bounded-register results for the Motorola 68020

registers available to each code improvement phase enabled improvements that otherwise would have been missed. In general, however, the amount of effort required to determine the most effective limit values for each target machine is substantial enough to seriously impair the retargetability of the bounded-register algorithm. This effort would be multiplied even more if a different limit value had to be determined for each of the code improvement phases.

6.2.4 Comparisons

Figure 119 shows the relative effectiveness of the three different register allocation strategies implemented in *llef*. The categories shown are ease of implementation, compilation speed, code quality, retargetability and freedom from interactions. Ease of implementation indicates how much effort is needed to implement the strategy. Compilation speed is a measure of the average amount of time that the compiler requires to compile programs when performing each of the allocation strategies. Code quality is an aggregate measure distilled from the number of instructions and memory references executed by the code produced using each strategy. Retargetability is an indication of how many lines of source code must be examined to modify a strategy to produce code for a new target architecture and how much additional evaluation is needed

Registers	2	3	4	5	6	7	8
4	-0.94 /-0.85	-0.94 /-0.85	-0.94 /-0.85	-0.94 /-0.85	-0.94 /-0.85	-0.94 /-0.85	-0.94 /-0.85
5	-0.15 / 0.09	-0.14 / 0.09	-0.14 / 0.09	-0.14 / 0.09	-0.14 / 0.09	-0.14 / 0.09	-0.14 / 0.09
6	-0.10 / 0.18	-0.10 / 0.18	-0.10 / 0.18	-0.10 / 0.18	-0.10 / 0.18	-0.10 / 0.18	-0.10 / 0.18
7	-0.07 /-0.13	-0.08 / -0.13	-0.08 /-0.13	-0.16 /-0.23	-0.16 /-0.23	-0.16 /-0.23	-0.16 /-0.23
8	-0.02 /-0.01	-0.03 / 0.03	-0.05 / 0.03	-0.10 /-0.08	-0.10 /-0.08	-0.10 /-0.08	-0.17 /-0.21
9	-0.51 /-1.35	-0.42 /-1.53	0.01 / 0.17	-0.04 / 0.05	-0.04 / 0.05	-0.04 / 0.05	-0.11 /-0.08
10	-0.51 /-1.35	-0.42 /-1.53	0.01 / 0.17	-0.04 / 0.05	-0.04 / 0.05	-0.04 / 0.05	-0.11 /-0.08
11	-0.51 /-1.35	-0.42 /-1.53	0.01 / 0.17	-0.04 / 0.05	-0.04 / 0.05	-0.04 / 0.05	-0.11 /-0.08
12	0.10 / 0.51	0.08 / 0.49	0.04 / 0.48	-0.11 / 0.09	-0.14 / 0.00	-0.14 / 0.00	-0.14 / 0.00
13	1.00 / 2.52	0.48 / 1.42	0.16 / 0.66	0.00 / 0.27	0.00 / 0.27	-0.03 / 0.18	-0.10 / 0.03
14	4.14 / 2.02	3.66 / 0.96	3.24 / 0.39	3.05 / 0.00	3.06 / 0.01	3.06 / 0.01	2.99 /-0.14
15	4.14 / 2.02	3.66 / 0.96	3.24 / 0.39	3.05 / 0.00	3.06 / 0.01	3.06 / 0.01	2.99 /-0.14
16	0.73 /-2.68	-0.43 /-2.99	0.36 /-3.11	0.20 /0.35	0.19 / 0.33	0.19 / 0.33	0.04 / 0.04
17	-0.77 /-2.85	-1.11 /-3.36	-0.37 /-3.46	0.04 / 0.13	0.07 / 0.18	0.06 / 0.16	-0.08 /-0.13
18	-0.77 /-2.84	-1.11 /-3.36	-0.37 /-3.45	0.04 / 0.13	0.07 / 0.18	0.06 / 0.16	-0.08 /-0.13
19	0.07 /-2.55	-0.14 /-2.94	-0.35 /-3.14	0.14 / 0.35	0.13 / 0.35	0.14 / 0.35	-0.01 / 0.06
20	0.07 /-2.55	-0.14 /-2.94	-0.35 /-3.14	0.14 / 0.35	0.13 / 0.35	0.14 / 0.35	-0.01 / 0.06
21	0.42 /-2.28	0.27 /-2.86	0.05 /-3.19	0.14 / 0.31	0.14 / 0.30	0.14 / 0.30	0.00 / 0.01
22	0.43 /-2.14	0.40 /-2.56	-0.43 /-3.25	-0.33 / 0.20	-0.33 / 0.20	-0.33 / 0.20	-0.40 / 0.05
23	0.43 /-2.14	0.40 /-2.56	-0.43 /-3.25	-0.33 / 0.20	-0.33 / 0.20	-0.33 / 0.20	-0.40 / 0.05
24	0.43 /-2.14	0.40 /-2.56	-0.43 /-3.25	-0.33 / 0.20	-0.33 / 0.20	-0.33 / 0.20	-0.40 / 0.05
25	0.43 /-2.27	0.37 /-2.69	0.10 /-3.10	0.20 / 0.37	0.20 / 0.37	0.20 / 0.37	0.12 / 0.22
26	0.50 /-2.02	0.41 /-2.57	0.07 /-3.16	0.16 / 0.20	0.16 / 0.19	0.19 / 0.29	0.12 / 0.14
27	0.54 /-1.94	0.42 /-2.46	0.05 /-3.33	0.14 / 0.19	0.15 / 0.19	0.18 / 0.29	0.11 / 0.14
28	0.54 /-1.94	0.42 /-2.46	0.05 /-3.33	0.14 / 0.19	0.15 / 0.19	0.18 / 0.29	0.11 / 0.14
29	0.67 /-1.50	0.63 /-1.92	0.09 /-3.41	-0.44 /-0.36	-0.31 / 0.12	-0.32 / 0.10	-0.38 /-0.04
30	0.55 /-1.78	0.58 /-1.98	0.05 /-3.47	-0.44 /-0.35	-0.32 / 0.12	-0.32 / 0.10	-0.39 /-0.04
31	0.55 /-1.78	0.58 /-1.98	0.05 /-3.47	-0.41 /-0.24	-0.32 / 0.12	-0.32 / 0.10	-0.39 /-0.04
32	0.55 /-1.78	0.58 /-1.98	0.05 /-3.47	-0.41 /-0.24	-0.32 / 0.12	-0.32 / 0.10	-0.39 /-0.04
33	0.62 /-1.78	0.68 /-1.80	0.04 /-3.48	-0.38 /-0.26	-0.29 / 0.11	-0.30 / 0.09	-0.36 /-0.06
34	0.62 /-1.56	0.65 /-1.62	0.14 /-2.96	-0.35 / 0.10	-0.29 / 0.24	-0.34 / 0.07	-0.40 /-0.06
35	0.50 /-1.77	0.55 /-1.81	0.09 /-2.96	-0.39 / 0.09	-0.33 / 0.22	-0.39 / 0.05	-0.45 /-0.07
36	0.50 /-1.77	0.55 /-1.81	0.09 /-2.96	-0.39 / 0.09	-0.33 / 0.22	-0.39 / 0.05	-0.45 /-0.07
37	0.50 /-1.77	0.55 /-1.81	0.09 /-2.96	-0.42 / 0.09	-0.33 / 0.23	-0.42 / 0.05	-0.45 /-0.07
38	0.54 /-1.77	0.59 /-1.81	0.14 /-2.95	-0.42 / 0.09	-0.33 / 0.23	-0.41 / 0.05	-0.45 /-0.07
39	0.69 /-1.45	0.68 /-1.56	0.20 /-2.77	-0.44 / 0.10	-0.36 / 0.09	-0.41 / 0.00	-0.43 / 0.02
40	0.69 /-1.45	0.68 /-1.56	0.20 /-2.77	-0.44 / 0.10	-0.36 / 0.09	-0.41 / 0.00	-0.43 / 0.02
41	0.65 /-1.44	0.65 /-1.55	0.20 /-2.59	-0.40 / 0.27	-0.32 / 0.26	-0.38 / 0.18	-0.39 / 0.19

Figure 118: Bounded-register results for the MIPS R3000

to “fine tune” the strategy. Freedom from interactions is a measure of how strongly an allocation technique interacts with each of the code improvement phases.

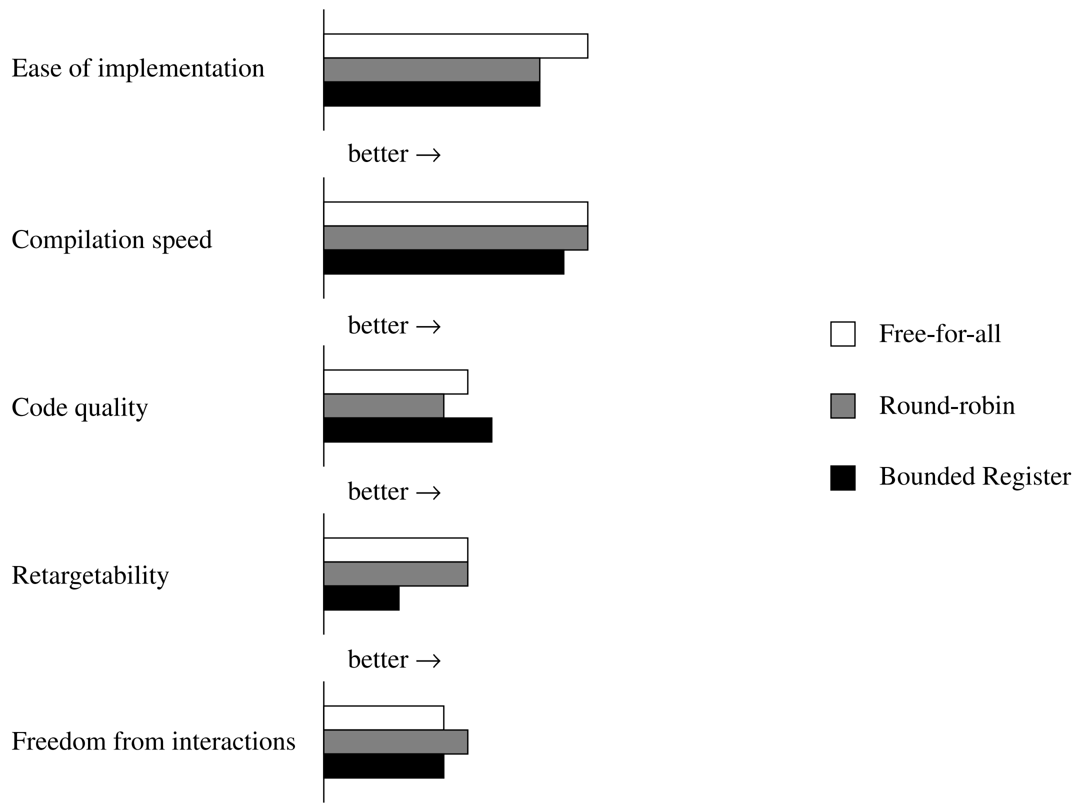


Figure 119: Relative effectiveness of the register allocation strategies

The comparison results indicate that the free-for-all strategy is the easiest to implement, executes at least as fast as the other techniques and is no less retargetable that the other strategies. Although the bounded register technique, when properly tuned, can produce somewhat better code and the round-robin strategy and interacts less with common subexpression elimination and loop code motion that the free-for-all strategy, the results suggest that, when all the factors shown here are considered, free-for-all performs better on average than the round-robin and the bounded register algorithms.

6.3 Unified approach

Although the previous sections show that the various attempts at alleviating *llef*'s deficiencies have yielded some worthwhile improvements, ample room remains for additional improvements. While performing machine-level code improvements and avoiding over-optimization are desirable strategies, the interactions caused by avoiding pseudo-registers and applying only code improvement transformations that are able to obtain registers they need limit the quality of the code produced by *llef*. The following sections present an approach that mitigates these deficiencies by unifying the code improvement and register allocation tasks to make them more responsive to each other. Doing so, however, entailed profound changes to *llef*. These changes resulted in a framework so different from *llef* and any of its predecessors, that it was given a new name: new technology retargetable optimizer, or *ntro*.

6.3.1 Overview

llef and *ntro* share the same front-end and much of the code expander that translates high-level intermediate language to machine code. The code expanders differ only in that RTLs are represented as strings inside *llef* and as trees in *ntro*. This change in representation was motivated by the need to maintain a persistent reference to the components that make up an instructions. With strings, for example, it is possible for a memory reference to be displaced when other parts of the instruction are modified. Using trees, however, individual elements stay in place when the instruction is modified.

The machine code produced by the code expander passes to *ntro*, whose initial phase builds a CFG for each source function and improves it by applying dead code elimination, jump minimization and branch chain elimination. Loops are also located at this time. Then, an instruction selection phase eliminates the naive code sequences produced by the code expander. Another phase is then invoked to perform global dead variable elimination. Finally, evaluation order determination is performed prior to local register assignment to reduce the number of hardware register needed to assign the pseudo-registers values generated by the code expander.

To this point, *ntro* has operated similarly to *llef*. The similarities, however, end after local register allocation is performed and the code is ready to be examined by the global code improvements. Instead of directly transforming the code, *ntro*'s common subexpression elimination, local variable promotion, loop-invariant code motion and induction variable elimination phases describe potential transformations in a transformation directory. After all potential transformations are described, a special phase, called the transformation dispatcher, performs the following:

- determines which of the transformations described in the directory would most improve code,
- attempts to find sufficient hardware registers to perform the transformation,
- applies the transformation,
- removes the transformation description from the directory and
- updates the descriptions of the remaining transformations as required.

The dispatcher continues to perform these steps repeatedly until:

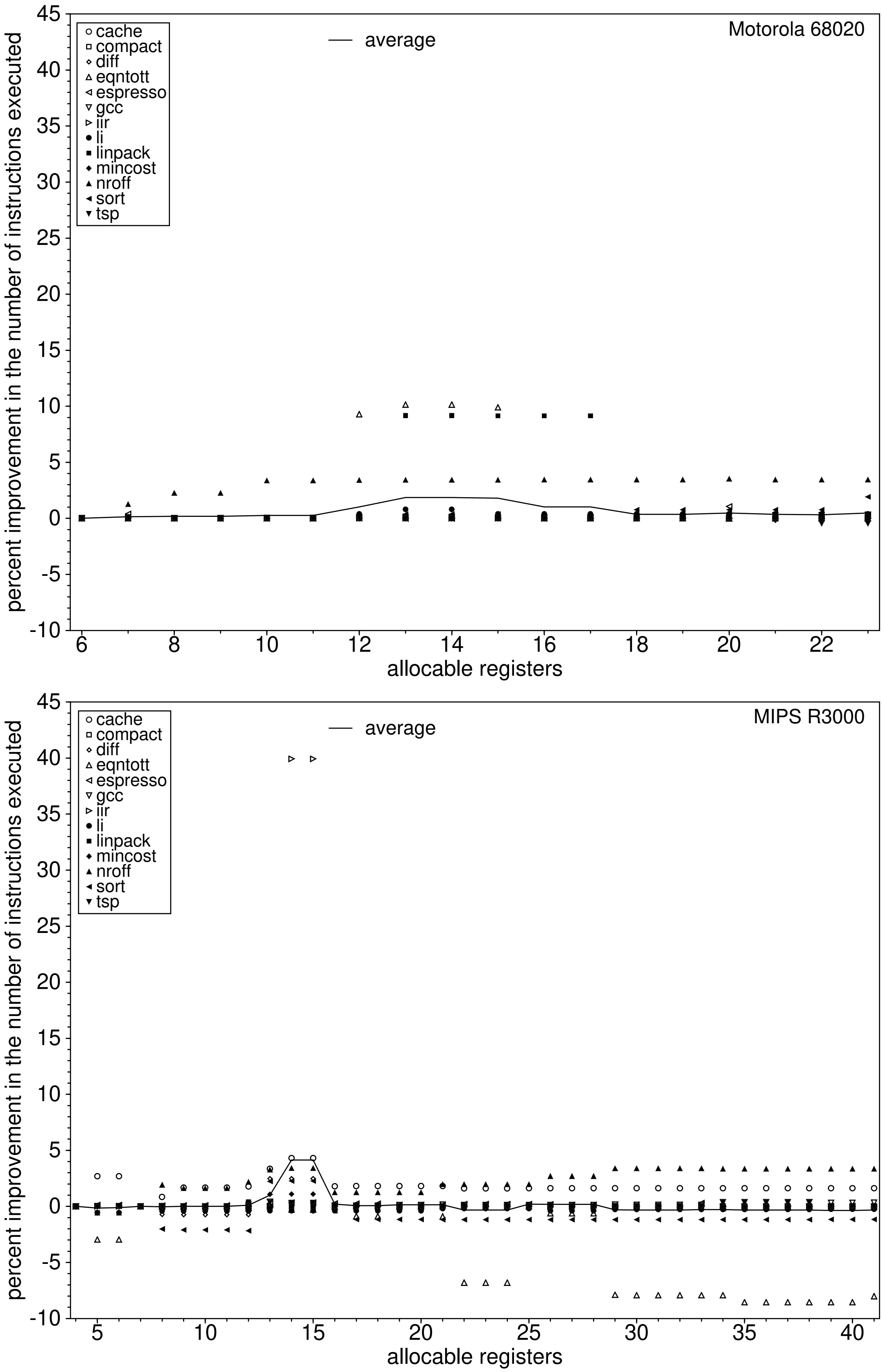


Figure 120: Bounded-register allocation performance results (instruction execution counts)

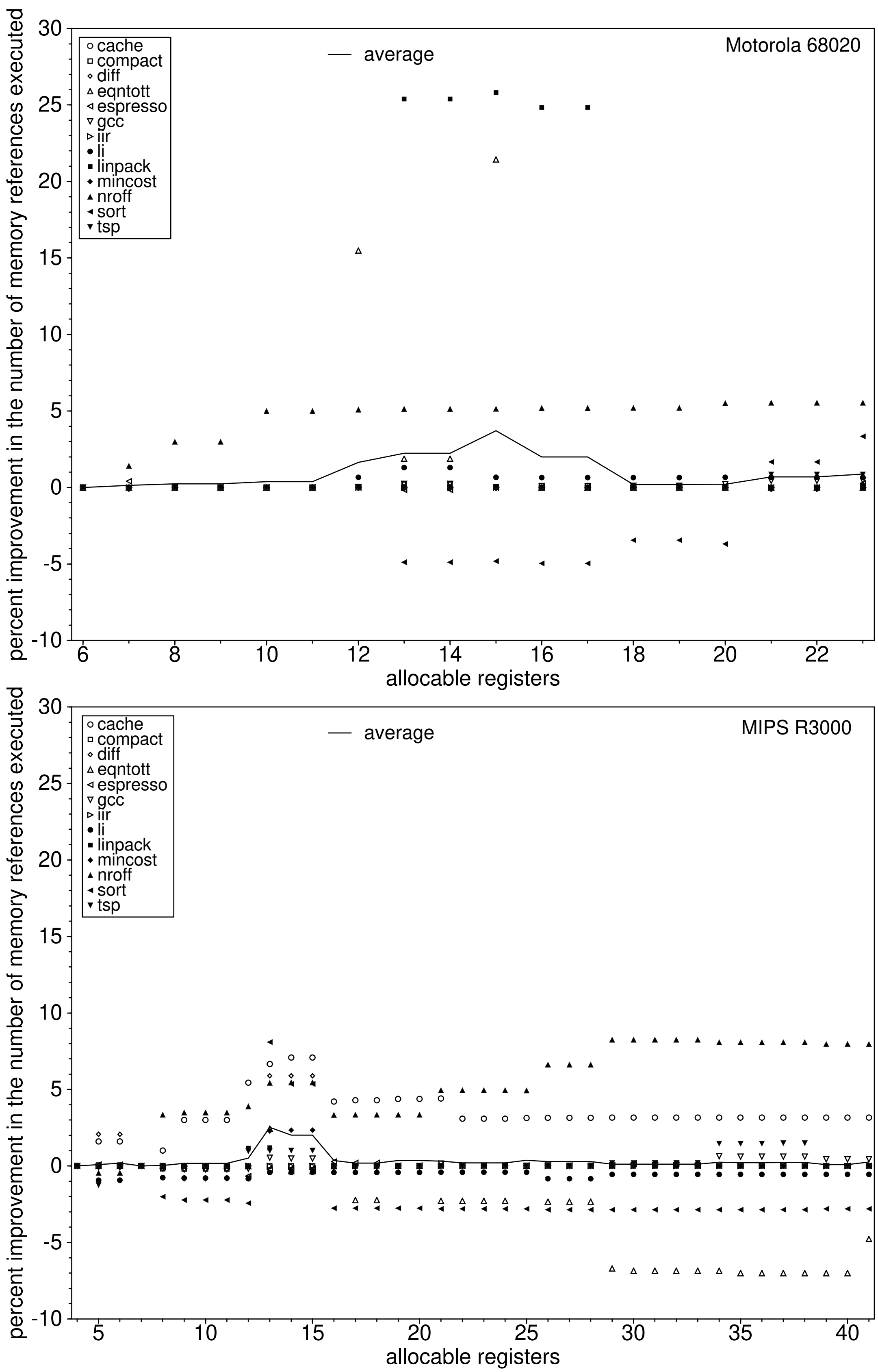


Figure 121: Bounded-register allocation performance results (memory reference counts)

• the transformation directory is empty,
• none of the transformations remaining in the directory would improve the code or
• sufficient registers cannot be found to perform any of the transformations in the directory.

Figure 122 shows an illustration of the structure of *ntro*.

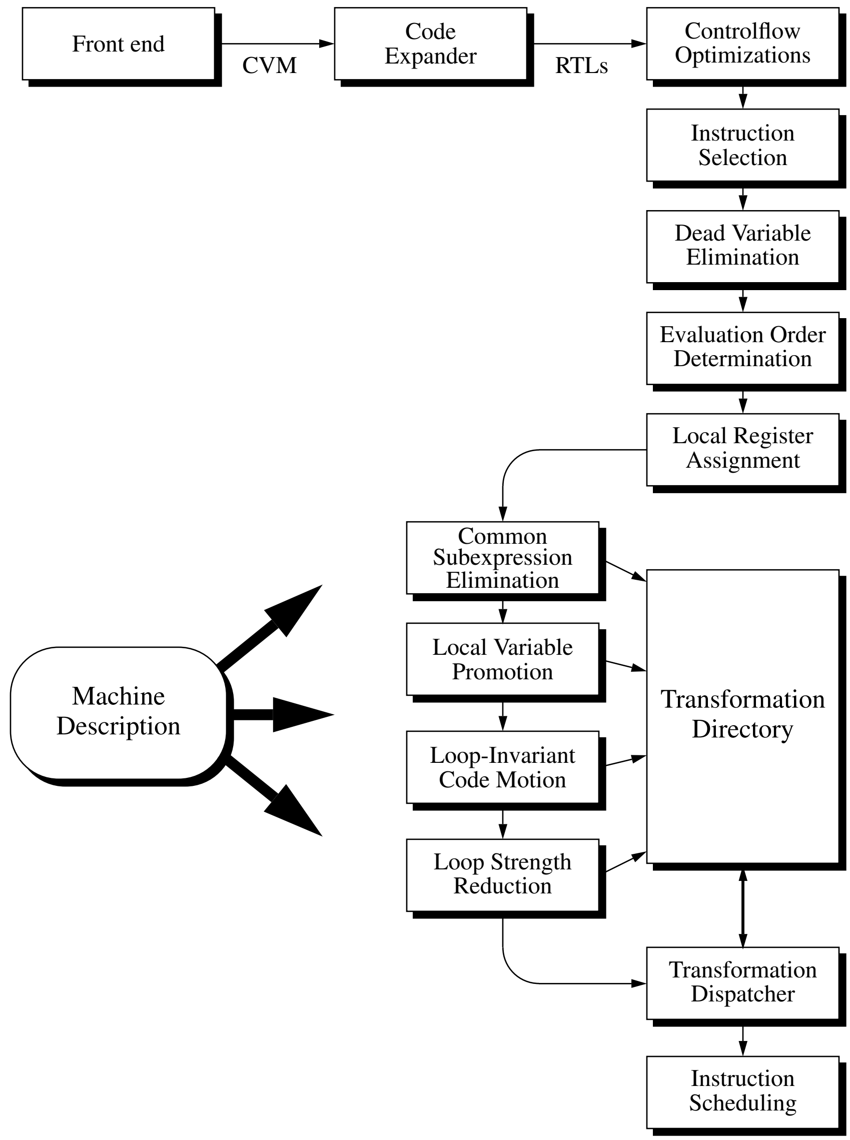


Figure 122: The structure of *ntro*

The transformation dispatcher enhances retargetability because it apportions registers well regardless of the size of the allocable register set on the target machine. The dispatcher also reduces phase ordering problems by using its knowledge of the target machine and the code being processed to determine the best ordering of not just phases, but of individual transformations, to best suit the needs of each function.

This completely eliminates the time and effort required to determine an appropriate phase ordering for each target machine.

Like *llef*, the *ntro* framework does not require the allocable register set to be partitioned between the code generator and the code improver. Unlike *llef*, however, describing all of the transformations before applying them allows *ntro*'s dispatcher to invest the register resources on the transformations that are most likely to improve the code. This strategy not only avoids overcommitting the register set, but is also less likely to under-utilize the register set by squandering the register resources on ineffective code improvements.

6.3.2 Implementation

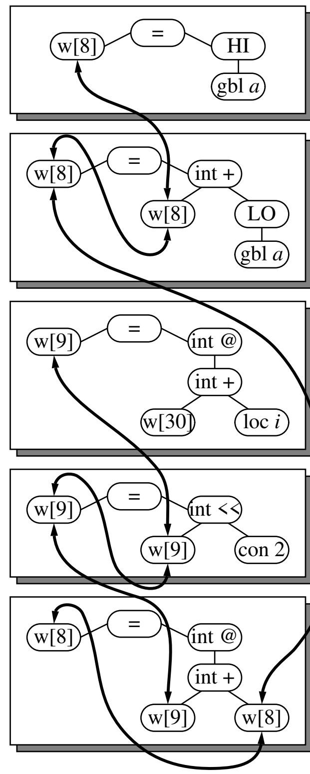
ntro internally represents RTLs as trees where each node may have zero, one or two children and contain information that is specific to its kind. A global define-use web connects related register and memory reference nodes to provide the define-use information needed to perform code improvements. Figure 123(b) shows the define-use web for the code sequence shown in Figure 123(a). Each register and dereference node whose aliases are completely known is connected to the node containing previous set or use of the item and to the node containing the next set or use of the item. Web links replace combiner links in the instruction selection process and are used to determine where an item is last referenced.

To connect definitions and uses across basic blocks, ϕ functions, which join merging values, and μ functions, which represent diverging values, are placed at the entry and exit points of some basic blocks. ϕ functions are the confluence operators used in the SSA form. μ functions, developed specifically to allow the web to point to all of the possible uses of an assignment, are essentially ϕ functions for the inverted CFG, where the function's exit block becomes its entry block and all control flow paths are reversed. Global web links can be used to combine instructions across basic blocks. An instruction pair is combined only if it contains an assignment that is the only possible source of a value and a use that is the only possible next use. These pairs are easily found because the RTL nodes representing the assignment and the use are directly connected without intervening nodes, or functions.

The web is updated as code improvement transformations are applied to allow instruction selection to be re-invoked at any point. This allows instruction selection to be used to determine if a potential transformation will enable new instructions to be selected. Updating the web also makes it possible to

```
1              w[8]=HI[_a];
2   {1}        w[8]=w[8]+LO[_a];
3              w[9]=W[w[30]+i.];
4   {3}        w[9]=w[9]<<2;
5   {2,4}      w[8]=W[w[9]+w[8]];                    w[9]
```

(a)



(b)

Figure 123: Define-use web links

eliminate the dead variables that are created as transformations are applied. The initial dead variable elimination pass uses the web to count the number of times that the value of each assignment is used and eliminates those that are never used. These use counts are then carefully updated as instructions are modified, inserted and removed so that they can automatically trigger the deletion of any expression orphaned by the code improvement process.

The values produced by each instruction and the relationship between these values are maintained in a *value graph*. This graph is used to perform common subexpression elimination, code motion and loop strength reduction. Even when the exact value of an item is not known, the relationships between values can enable useful transformations. Value graph nodes are indexed for quick access and to ensure that each node represents a unique type and value combination. Every RTL node points to the value graph node that represents its value.

The transformation directory is a collection of transformation descriptions, each describing a specific transformation that can be applied to the code. Figure 124 illustrates the descriptions of two simple code improvement transformations on a segment of SUN SPARC code. Although the directory would contain other transformations in addition to these two, space considerations prevent their inclusion. The complexity of the illustration warrants a bit of orientation to precede the explanation of the elements that make up the directory. Along the left side of the figure is the RTL tree representation of the code. The RTL items are enclosed by squares representing the basic blocks that make up the CFG. The transformation description items are the tall, narrow boxes to the right of the code. Within these boxes are the local transformation items, represented by the squares with the rounded corners and the instruction-level items, which are the smaller rectangles inside the local transformation items. The RTL trees at the bottom-right corner are RTL expression and value items.

A transformation description contains a list of local transformation items that indicate the area over which the registers required to perform the transformation would be live. The region covered by each local item is represented in Figure 124 by the pair of arrows originating from the local item and pointing to the basic blocks. Local items serve two purposes in the dispatch process. First, they tell the dispatcher which parts of the CGF must be examined to determine which hardware registers are available to perform the transformation. Second, each local item is connected to all of the other local items that it shares the region with. These connections form an interference graph that allows the dispatcher to determine which transformations cannot share the same registers. The dispatcher can use this information to find groups of transformations that could share a register and are together more beneficial than any single transformation is.

Associated with some of the local items are instruction-level items that indicate which RTL nodes are affected by the transformation, describe the specific change that would be made to those nodes and contain the estimated cost and benefit values associated with that change. Cost and benefit values are obtained from the machine description and multiplied by the loop nesting level of the code because loop code is usually executed more frequently than straight-line code. Each RTL node contains a list of the instruction-level transformation items linked to it, so that if any one of the transformations that can be applied to it is

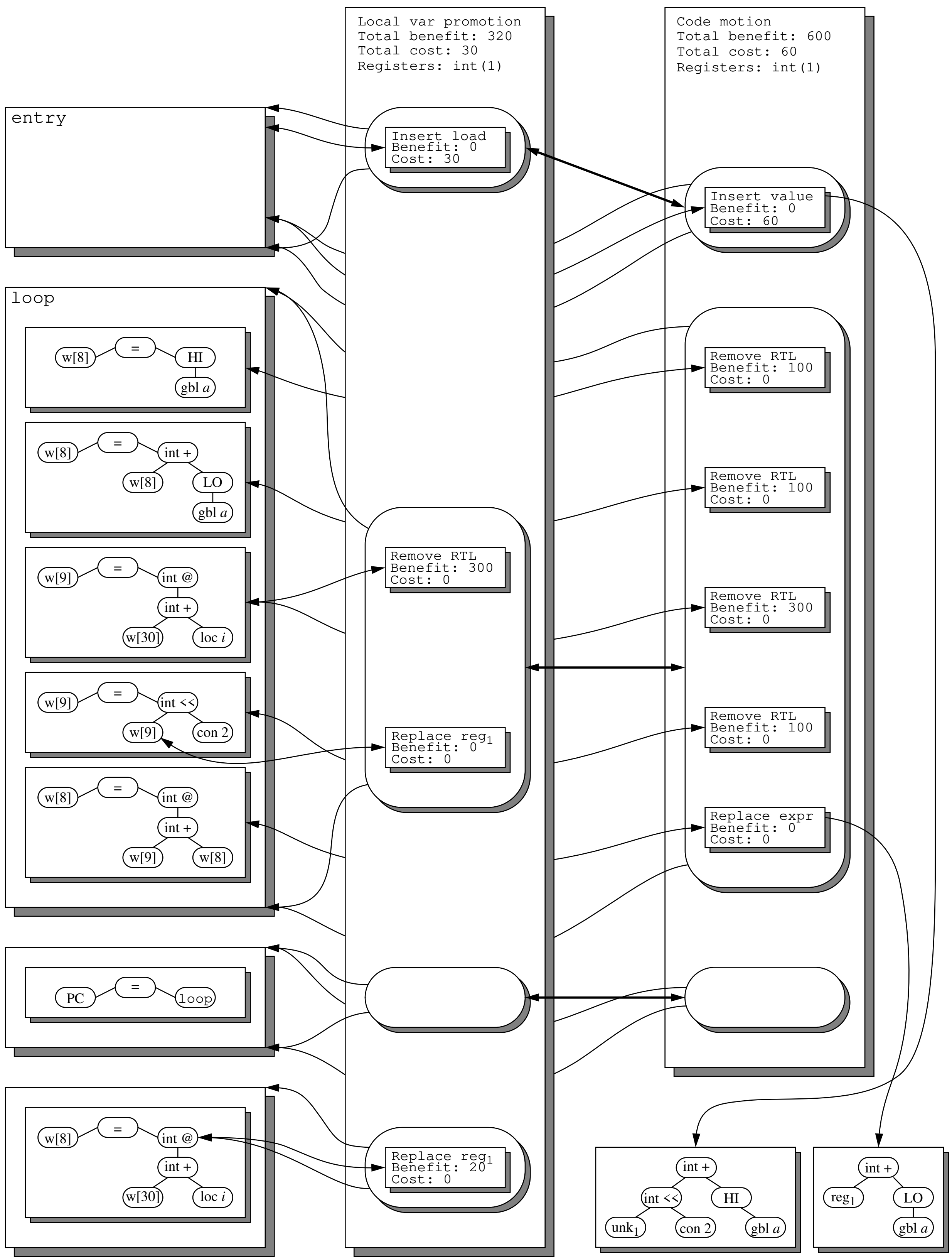


Figure 124: The transformation directory

performed, all of the other transformations that also reference it can be modified to reflect the impact of the transformation. For example, both of the transformations shown in Figure 124 affect the instruction that references variable *i* inside the loop. If the code motion transformation is applied, the entire RTL, along with the reference to the variable reference would be removed. In this situation, it would be necessary to locate the description of the local variable promotion transformation and update it. The update would entail removing the appropriate instruction-level transformation item from the local variable promotion description and updating the cost and benefit values to indicate that it is no longer as beneficial as it was prior to the code motion transformation.

The sum of the cost and benefit values of the items that make up a transformation reflect the global costs and benefits of performing the transformation. The dispatcher uses these sums to decide which transformation to apply next. As the illustration shows, the code motion transformation has a total cost of 60 and a benefit of 600, which means that it will be selected by the dispatcher over the local variable promotion transformation with a benefit of 320 and a cost of 30. On other architectures, these values might favor the local variable promotion transformation over code motion, instead.

Each description in the transformation directory indicates the number and type of registers needed to perform the transformation. The transformation dispatcher uses this information to attempt to allocate suitable registers just prior to attempting the transformation. This entails searching through the regions indicated by the local items to determine which hardware registers are not live within them. If insufficient hardware registers are available, the transformation is postponed until another transformation releases enough registers to allow it to proceed.

The code improvement phases can produce transformations that are variations of each other with different register requirements, so that if the most beneficial variant is postponed, the less beneficial variant can be applied instead. For example, consider a pair of related transformations: one promotes a local variable to a register over the entire life of the function and the other over a single loop. The former is desirable because it does not require the variable to be exchanged between its home location in memory and a register at the boundaries of the loop. Its disadvantage is that it ties up a register over a greater area than the second option does in return for a marginal benefit. If the demand for registers is low, the transformation dispatcher will successfully obtain the register required by the former transformation and the latter transformation will be automatically deleted from the directory by the dispatcher as a side effect of applying the first transformation. When there are few registers or the demand for them is high, the former transformation is postponed because no register is available for the entire life of the function, but the latter transformation might succeed, because it spans a smaller region.

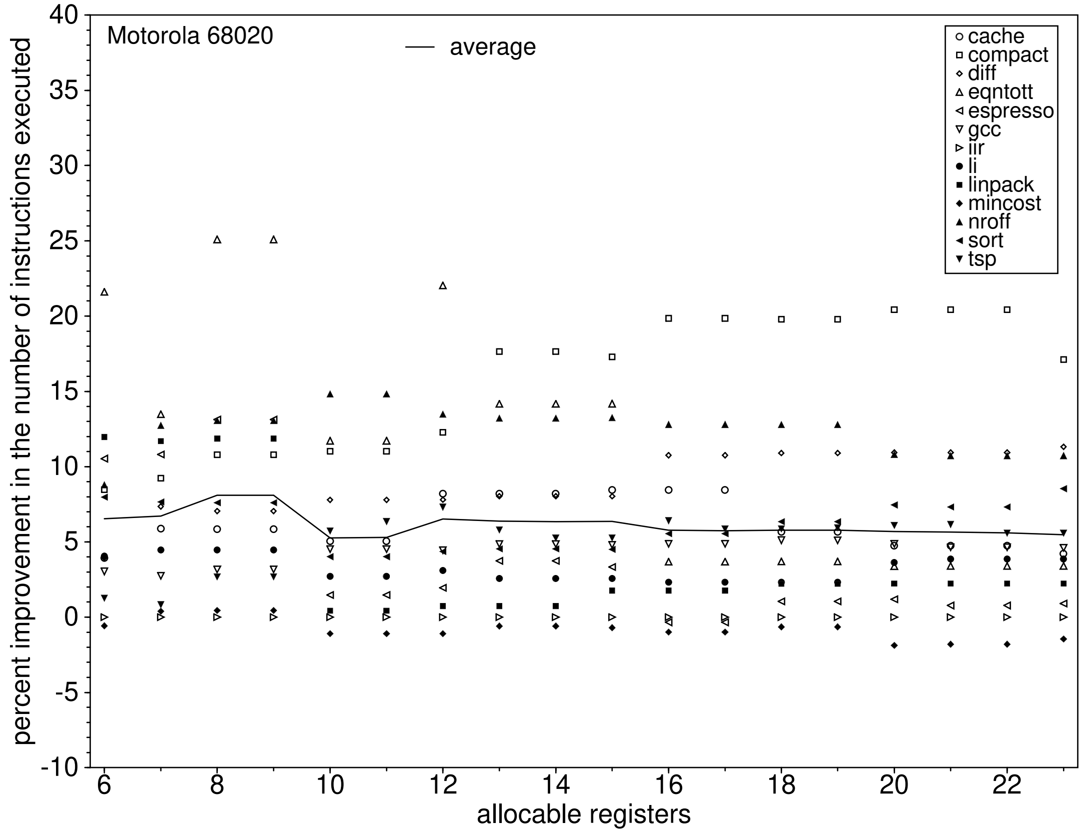
Some transformations initially consume registers, but trigger the elimination of a dead variable so that their net register consumption is zero or negative. On-demand register allocators like the one in *llef* prevent these transformations from happening unless there are enough free registers available to “prime” the transformation process. *ntro*, on the other hand, will find registers for these transformations without the additional free registers because it performs the transformation using pseudo-registers before determining which registers can be used to perform the transformation. The pseudo-registers are immediately replaced with the allocated hardware registers if the allocation process is successful. Otherwise, the trial transformation is retracted in its entirety.

Adding new code improvements to *ntro* is not as difficult as its integrated nature might suggest. The transformation directory structure provides most of the transformation primitives (i.e. replace an expression with a register, remove an RTL, add an expression that computes a value, etc.) needed to accommodate most code improvements, and new primitives could be added as required. The most important requirement is that code improvement phases be able to detect all of the potential transformations by examining the code before any of the other global code improvements have been applied.

6.3.3 Results

In the current implementation of *ntro*, only local variable promotion and loop-invariant code motion transformations are inserted into the transformation directory. Additional global code improvements such as common subexpression elimination, loop strength reduction and induction variable elimination could be added to the framework as described in the previous section. Two global code improvements, however, are sufficient for the purpose of measuring the effectiveness of the *ntro* framework.

Figures 125, 126 and 127 show the results of comparing *ntro* against *llef*’s free-for-all allocation strategy. To make the comparison fair, only the local variable promotion and loop-invariant code motion phases were invoked in *llef*. The results show that *ntro* produces, on average, better code for both CISC and



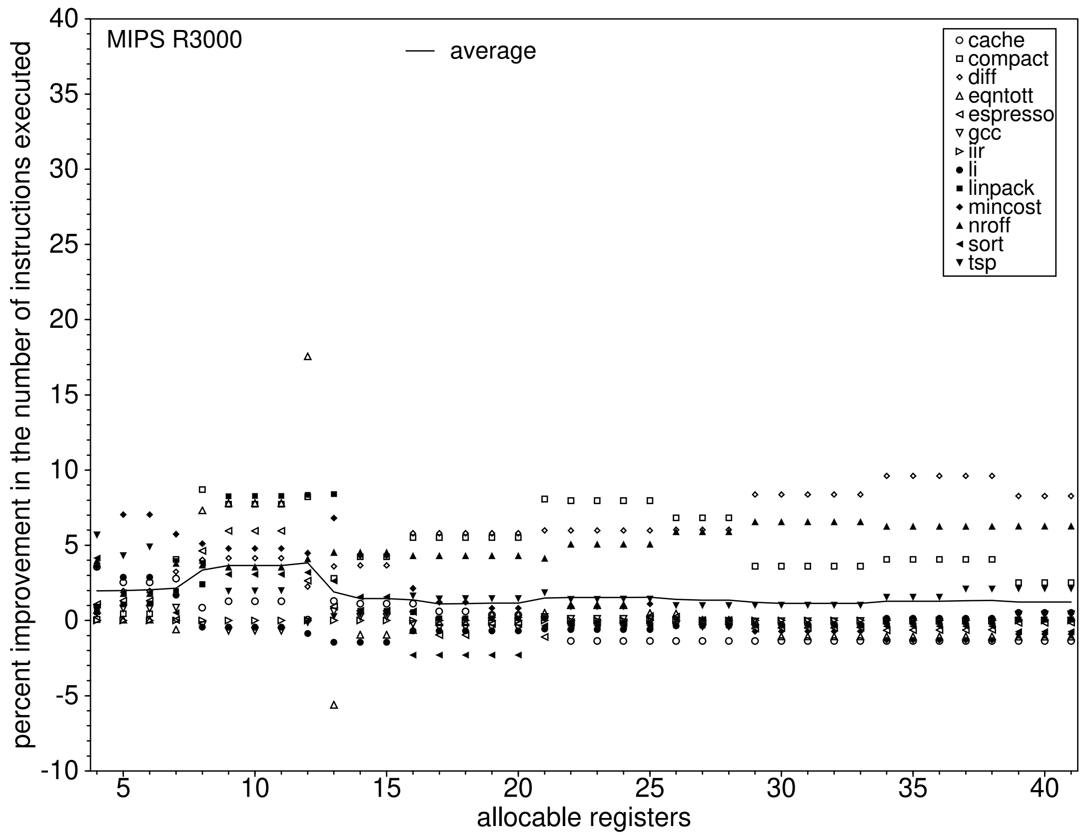


Figure 125: *ntro* performance results (instruction execution counts)

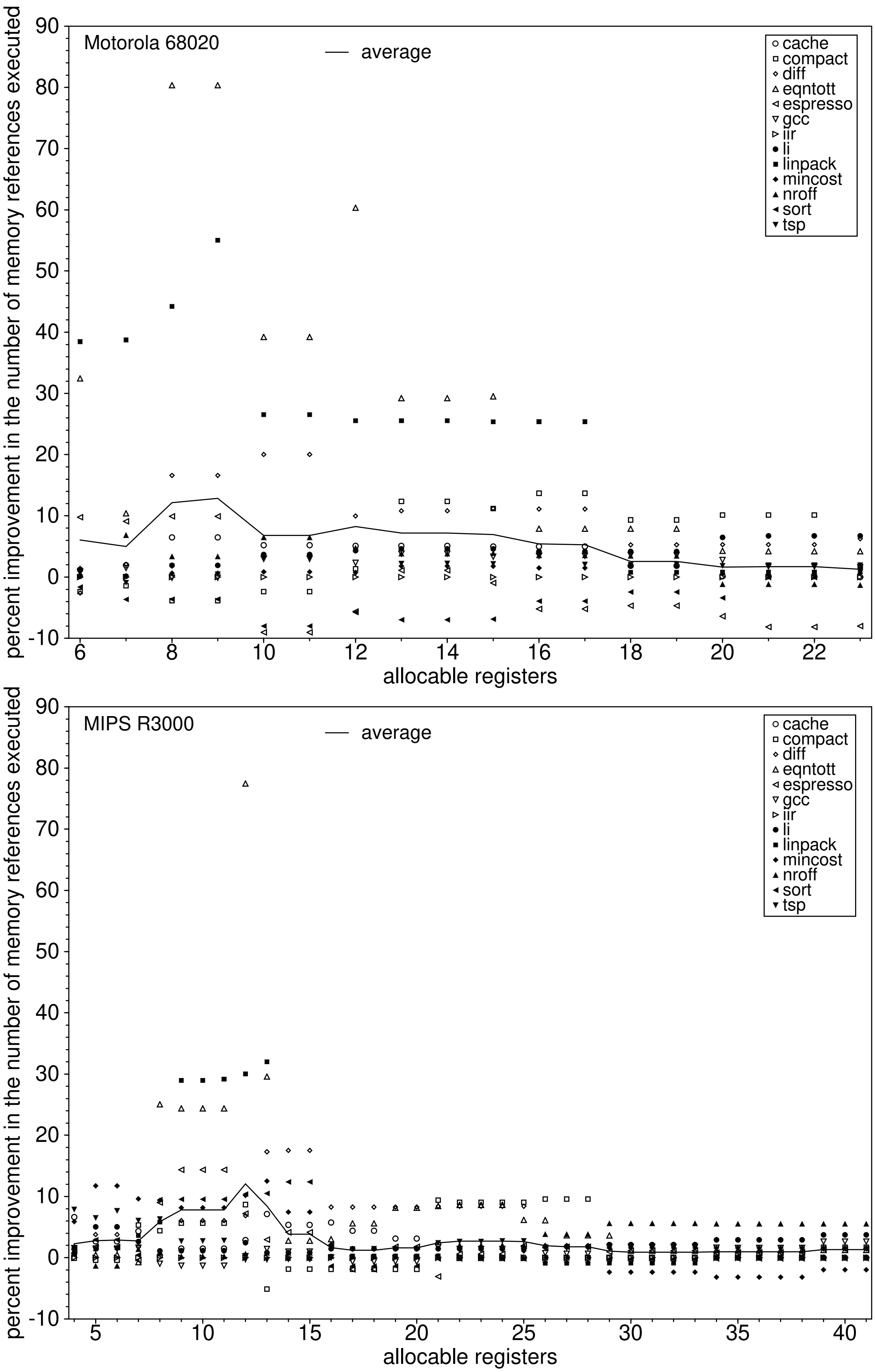


Figure 126: *ntro* performance results (memory reference counts)

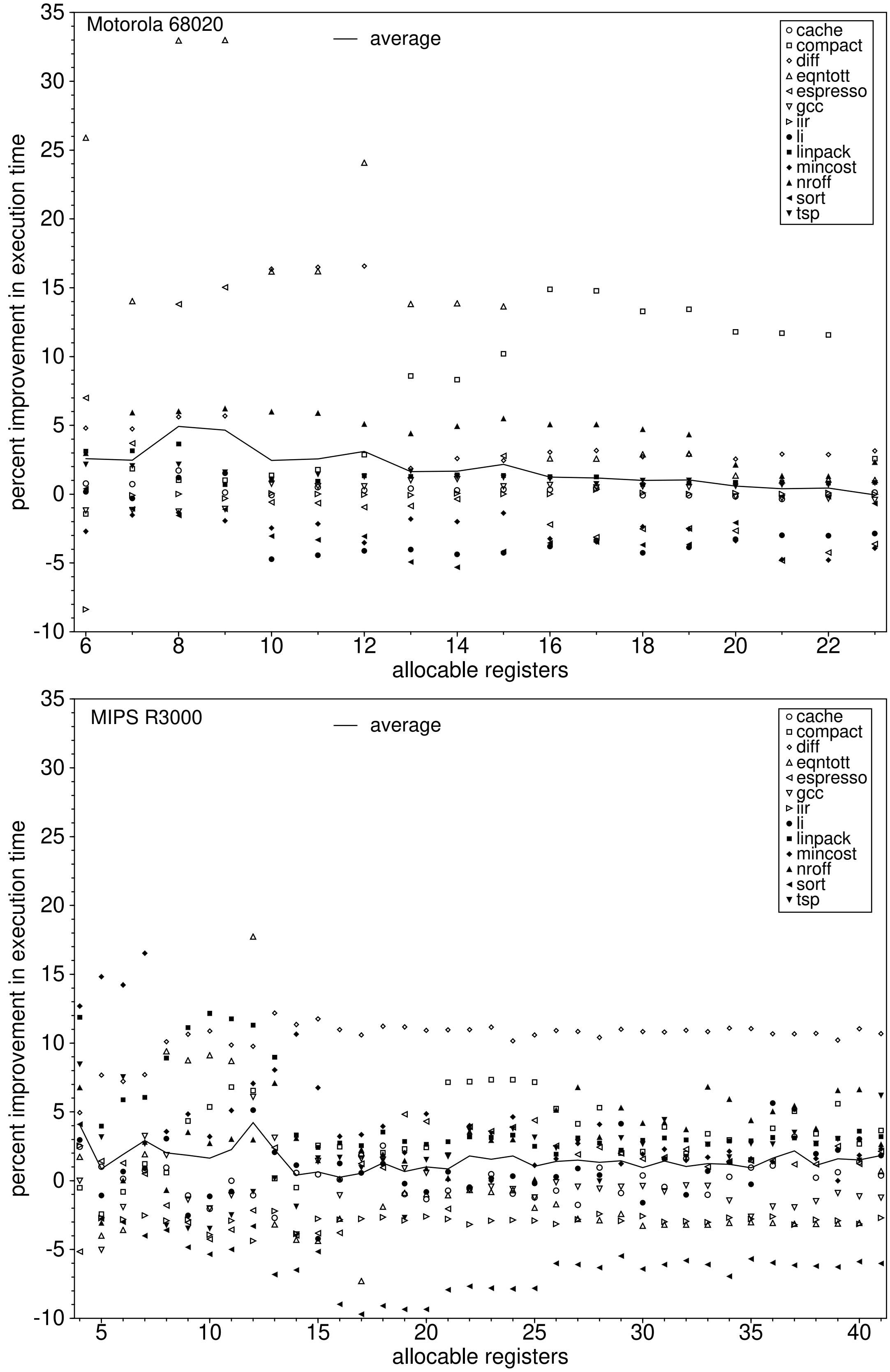


Figure 127: *ntro* performance results (execution times)

RISC machines over the entire range of different register set sizes than *llef*. The improvement, however, is more significant when there are enough registers to perform global code improvements but not enough to accommodate all potential improvements. Under these circumstances, *ntro* is better able to use scarce register resources to perform the most effective code improvements than *llef* is. When the number of registers available is high, *ntro* is only able to produce code that is only slightly better than the code produced by *llef*.

ntro is also easier to retarget than *llef*. As Figure 128 shows, *ntro*'s machine-dependent sections comprises an average of 50% fewer lines of source code than *llef*'s. Much of these savings can be attributed to the fact that special components and machine-specific patterns are easier to locate in tree RTLs than they are in string RTLs. Also, the experience obtained from developing *llef* was instrumental in the design of *ntro*'s more retargetable structure.

Architecture	*llef*	*ntro*
Motorola 68020	8052	7206
MIPS R3000	5522	3497
SUN SPARC	6738	3767

Figure 128: Number of machine-dependent source code lines

ntro's greatest disadvantage is compilation time. While *llef* requires only slightly more (5% to 20%) time than a production compiler needs to compile large functions (over 1500 lines), *ntro* will take more than ten times as long as *llef* to compile these functions. When compiling average-sized functions (from 20 to 200 lines), *ntro* is comparable to *llef* in terms of compilation time. The reason for these long compilation times is that the algorithms that *ntro* uses to incrementally update its global define-use web and live-variable dataflow information do not scale well to large control flow graphs.

6.3.4 Comparisons

Figure 129 compares *ntro* to the three register allocation techniques presented in Section 6.2. The results are added to the comparison results presented in Figure 119. These results reflect the fact that the *ntro* framework is much more difficult to implement that the free-for-all, round-robin and bounded-register algorithms and that it requires a substantially greater amount of time to execute. In terms of code quality, retargetability and freedom from interactions, however, *ntro* is clearly superior.

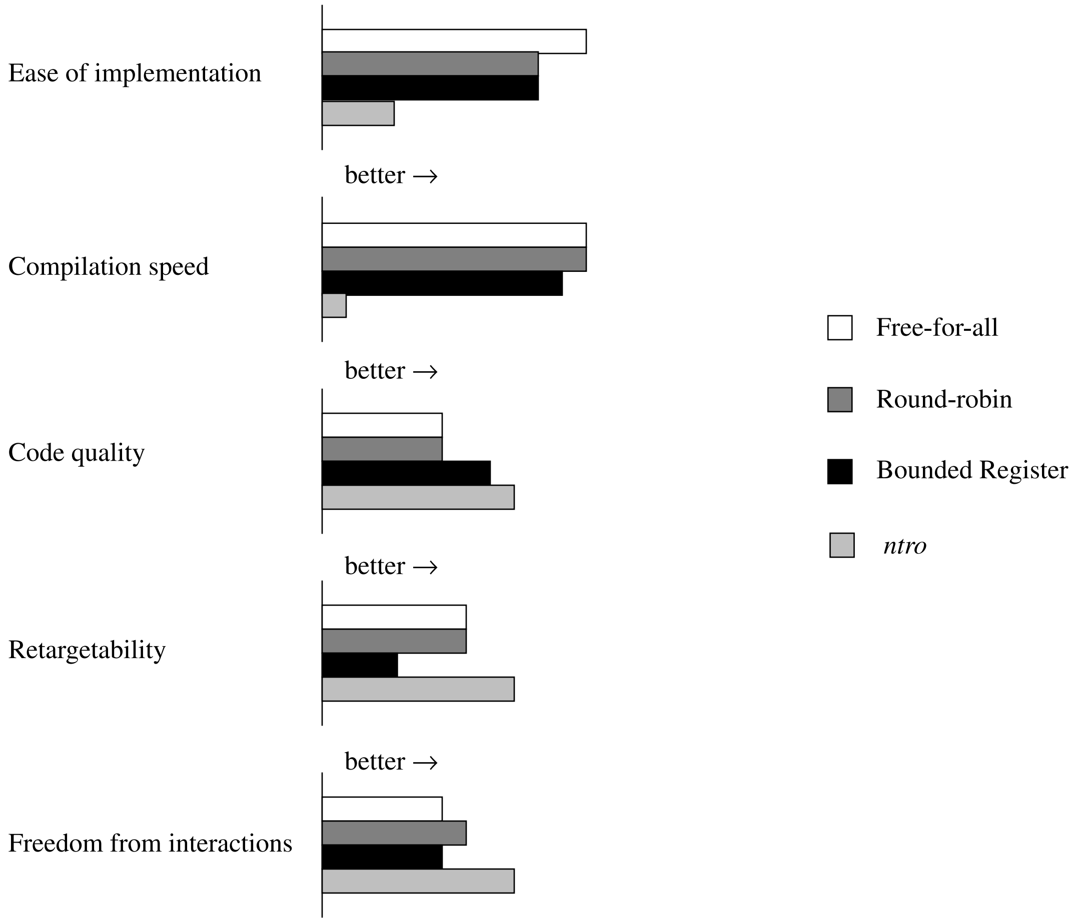


Figure 129: Relative performance of *ntro* and the allocation strategies in *llef*

6.4 Summary

This chapter has presented several different methods that can be used to improve the quality of the code produced by *llef*. Some of these improved the ability of the common subexpression elimination and the loop-invariant code motion phases to detect redundant or loop-invariant expressions in the presence of register reuse. Somewhat less successful were attempts to reduce the impact of register reuse and register misuse by modifying the behavior of the register allocator. These modified algorithms either had the unfortunate effect of actually reducing the average quality of the code produced by *llef*, or required so much additional effort to use that they significantly reduced the retargetability of the system.

These measurements suggested that a more promising approach to improving the quality of the code was to integrate code improvements and register allocation. This required the development of an entirely new framework called *ntro*. *ntro*'s code improvement phases use algorithms that are not affected by register reuse and a register allocation technique that apportions the registers to the code improvement transformations that are most likely to improve the quality of the code. This unified approach outperforms *llef* on both CISC and RISC architectures.

CHAPTER 7

CONCLUSIONS

The results of the research presented in this dissertation support the following conclusions:

- when simplicity, retargetability, code quality and execution time are considered, the simple *free-for-all* allocation strategy provides the best overall performance in retargetable, machine-level optimizing compilers,
- phase iteration reduces phase ordering problems, simplifies the task of each code improvement phase in the optimizing compiler and enhances the retargetability of the compiler,
- integrating the code improvement and register allocation tasks enhances the quality of the code emitted and improves the retargetability of the system without sacrificing simplicity,
- machine-level global code improvement techniques allow retargetable optimizing compiler to produce code whose quality meets or exceeds the quality of the code produced by traditional production compilers,
- the register transfer notation is an effective machine-level intermediate representation that can represent a wide range of instruction sets, and
- the register deprivation technique developed to gauge the effectiveness of the register allocation algorithms presented in this dissertation is an effective evaluation tool that has uses beyond its original application.

This dissertation describes the development and evaluation of several retargetable register allocation algorithms that were designed to avoid over-optimization and the need to partition the register set between code generation and code improvements. The most important characteristic of these algorithms is their ability to prevent the code improvement phases from performing more transformations than the register set on the target architecture permits. Other considerations include reducing interactions between the register allocator and the code improvement phases and allocating registers to the code improvement transformations that most benefit the estimated quality of the code produced by the compiler.

The register deprivation measurements presented in Chapter 6 show that when simplicity, retargetability, code quality and execution time are considered, the simple *free-for-all* register allocation algorithm is the best candidate for generating production-quality code over a wide range of architectures. The somewhat more sophisticated *round-robin* and *bounded-register* algorithms can improve slightly on the performance of the simple allocator, but often at the cost of simplicity or retargetability. Although the highest quality code was produced using a *unified* allocation strategy, which integrates the register allocation and code improvement tasks, the substantially greater amount of time needed to use this strategy limits its utility.

Another contribution of this work is the use of phase iteration as a technique that not only improves the quality of the code generated by the compiler, but simplifies the task of each phase in the optimizing compiler and enhances retargetability by eliminating the need for determining the most effective phase ordering on each target architecture. The phase iteration mechanism presented here is simple, yet able to determine when particular code improvement phases should be re-invoked. Most importantly, however, a properly implemented phase iteration strategy does not unduly increase the amount of time required to compile programs.

The search for effective techniques to improve register allocation and reduce phase interactions led to the development and evaluation of a framework that performed transformation dispatching, which is a technique that avoids phase ordering problems and reduces the impact of the interactions between the various phases of the compiler. Transformation dispatching does more than just order the phases, it orders the individual transformations applied by the optimizing compiler. As a result, the register resources provided by the target architecture are used more effectively. The measurements of the performance of this approach reveal that it is able to improve the execution time of the code produced by an optimizing compiler an average of two to five percent with just two global code improvements. This suggests that more significant improvements are possible with a larger code improvement suite. Under these circumstances, the competition for registers rises to a level where transformation dispatching clearly outperforms more traditional allocation methods even on architectures that provide large register sets.

The comparisons between *llef* and several production compilers on various different architectures prove that it is possible to build retargetable optimizing compilers that produce code whose quality equals or exceeds the quality of the code produced by traditional production compilers. The key to obtaining this level of retargetability and code quality from a retargetable optimizing compiler is performing code improvements and register allocation at the machine level. This is made possible by using the register transfer notation to represent target-machine instructions, carefully structuring code improvements to isolate architectural-dependencies and using register allocation algorithms that operate effectively across a wide range of register sets. These strategies have been successfully used in both the *llef* and the *ntro* optimizing compilers.

The success of *llef* and *ntro* shows that register transfers are an excellent intermediate representation for the application of code improvements. They support a wide range of architectures, yet they are an architecture-independent notation. These two factors make it possible for architecture-specific code improvements to be performed by algorithms that are largely architecture-independent and, thus, easy to retarget. *llef* shows how traditional, application-specific and architecture-specific global code improvements can be incorporated into a low-level framework that is both effective and retargetable.

The development of *llef* is an important by-product of this investigation. This experimental framework provided a flexible platform for the development and evaluation of the global code improvement and register allocation algorithms presented here. In addition, *llef*, is currently supporting the development and design of new techniques for describing calling conventions and architectural features, new run-time global code improvement algorithms and instruction scheduling algorithms for a new asynchronous, pipelined architecture.

The register deprivation measurement technique developed to gauge the retargetability of register allocation algorithms can also be used to improve compiler validation suites and determine the impact of register competition on code improvement algorithms. This technique greatly simplified the evaluation process used in this dissertation by allowing a pair of architectures, one CISC and one RISC, to simulate the impact that a wide range of different target architectures would have on the register allocation and code improvement phases of the compiler. Because the register deprivation process improves the testability of compiler validation suites, it is possible to claim that *llef* and *ntro*, both of which were subjected to substantial register deprivation trials, are more likely to produce correct code than average prototype systems are and, in all likelihood, are as reliable as most production compilers are.

The significant reduction in the number of architecture-specific lines between *llef* and *ntro* suggests that much more can be done to reduce the amount of code that a user must examine when retargeting a compiler to a new target architecture. One promising technique that can be used to streamline and simplify the retargeting process is the use of specialized descriptions that are processed by tools similar to *regtool*. In the case of register allocation, *regtool* allowed a short register description to replace more than ten times as many lines of complex **C** code. Using similar tools, it may be possible to reduce the amount of code that a user must examine during the retargeting process to a few hundred lines and reduce the amount of time required to retarget the entire optimizing compiler from a month to as little as a couple of days.

While transformation dispatching has many advantages, its use is currently restricted by the poor performance of the incremental update algorithms used to maintain the global define-use web and the live-variable dataflow information. The development of faster update algorithms is a prerequisite to incorporating transformation dispatching in production compilers.

Currently, *ntro* does not perform enough global code improvements to allow it to compete with existing production compilers. The comparison results presented in the previous chapter suggest that *ntro* should outperform traditional register allocation schemes in situations where the demand for registers is high. Since these situations are more likely to occur when comprehensive code improvement suites and inter-procedural code improvements are used, adding more code improvements to *ntro* will help strengthen the argument in favor of transformation dispatching.

Instruction scheduling is not currently incorporated into the *ntro* framework despite the fact that it interacts with register allocation. Integrating instruction scheduling into *ntro* would allow register allocation and code improvement algorithms to take their effect on the processor's pipeline into consideration. Such a framework would handle instances where a transformation's impact on the dynamic execution of the program affects its importance relative to the other potential transformations. Thus, scheduling information could be incorporated into the transformation directory to improve the accuracy of the benefits and penalties associated with each transformation and allow the transformation dispatcher to make better register allocation and code improvement decisions.

APPENDIX A

REGISTER TRANSFER NOTATION

The register transfer notation is a machine-independent intermediate representation used to describe a broad range of machine-specific instruction sets. Register transfers have their origins in the Instruction Set Processor (ISP) notation developed by Bell and Newell [BELL71]. The original notation was altered slightly to facilitate its use as a low-level intermediate language. The register transfer notation described here is similar to that presented by Davidson for PO [DAVI84b].

A register transfer describes the effect of a machine instruction. Most machine operations can be defined in terms of the data movement that they effect. For this reason, the register transfer notation is particularly suited to describing data transfers between registers and memory locations.

Registers

A register is represented using the following notation:

$$r[x]$$

where r is a lower-case alphabetic character denoting the assignment type of the register (see Section 5.3.1) and x is a decimal number indicating which of the registers belonging to that assignment type is being referenced. Depending on the register number, a register may represent either a target machine register or a pseudo-register. For example, if the target machine has 32 general-purpose registers, then register numbers 0 through 31 denote the target registers and numbers greater than 31 denote pseudo-registers.

Memory References

A memory reference is represented using the following notation:

$$M[address]$$

where M is an upper-case alphabetic character denoting the assignment type of the memory reference (see Section 5.3.1) and *address* is an expression indicating the address of the memory location being referenced.

Constants

Constants represent integer and floating-point values. An integer constant is represented using the following notation:

number

where *number* is a sequence of one or more numeric characters representing a decimal integer value. Integer constants are always positive. Negative integer values can be obtained by applying the unary negation operator to the constant.

A floating-point constant is represented using the following notation:

mantissa E*sign exponent*

where *mantissa* is a sequence of one or more numeric characters representing the decimal integer value of the mantissa and *exponent* is a sequence of one or more numeric characters representing the decimal integer value of the exponent. The base of the exponent is always ten. *sign* is either a '+' or a '-' character indicating the sign of the exponent. Floating-point constants are always positive. Negative floating-point values can be obtained by applying the unary negation operator to the constant.

Labels

Labels represent constant address values. Labels are represented using the following notation:

L*number*

where *number* is a string of one or more numeric characters. Labels are most commonly used to mark instructions in the code segment that branch instructions may transfer control to. Labels can also be used to denote address values that contain data much the same way as global identifiers do.

Global identifiers

Global identifiers represent constant address values or function entry points. Global identifiers are represented using the following notation:

identifier

where *identifier* is a string of one or more alphabetic characters, numeric characters and underscore '_' characters. The only restriction is that the first character cannot be a numeric character.

Local identifiers

Local identifiers represent constant offset values. Local identifiers are represented using the following notation:

identifier.

where *identifier* is a string of one or more alphabetic characters, numeric characters and underscore '_' characters. The only restriction is that the first character cannot be a numeric character. The only difference between a global identifier and a local identifier is that the latter ends with a period '.' character.

Operations

Register transfers permit logical and arithmetic operations to be performed on values. Operations may be either unary or binary and the symbols used to denote operations are usually overloaded so that the type of the operands determines which operation is performed. Table 1 lists the standard set of integer binary operators. Table 2 lists the standard set of floating-point operators. Note that it is not always the case that a

Operator	Function	Operator	Function	Operator	Function
' '	Unsigned left shift	'!'	Not equal	'"'	Unsigned right shift
'#'	Unsigned modulus	'$'	Sign extend	'%'	Modulus
'&'	Bitwise AND	'′'	Less than or equal	'*'	Multiplication
'+'	Addition	','	List separator	'-'	Subtraction
'/'	Division	':'	Equal	'<'	Less than
'='	Assignment	'>'	Greater than	'?'	Comparison
'@'	Unsigned multiply	'\'	Unsigned division	'^'	Bitwise XOR
'`'	Greater than or equal	'b'	Bitwise AND NOT	'g'	Unsigned greater-equal
'h'	Unsigned greater than	'l'	Unsigned less than	'o'	Bitwise OR NOT
's'	Unsigned less or equal	'u'	Unsigned comparison	'x'	Bitwise XOR NOT
'{'	Left shift	'|'	Bitwise OR	'}'	Right shift

Table 1: Integer binary operators

floating-point operator produces a floating-point value. For example, the floating-point comparison operator accepts two floating-point values, but produces an integer value. The unary integer operators are listed in Table 3. There is only one floating-point unary operator currently defined, which is the unary negate.

Operator	Function	Operator	Function	Operator	Function
'!'	Not equal	'′'	Less than or equal	'*'	Multiplication
'+'	Addition	'−'	Subtraction	'/'	Division
':'	Equal	'<'	Less than	'='	Assignment
'>'	Greater than	'?'	Comparison	'`'	Greater than or equal

Table 2: Floating-point binary operators

Operator	Function	Operator	Function
'−'	Negate	'd'	Auto-decrement
'i'	Auto-increment	'~'	Complement

Table 3: Integer unary operators

Macros

Macros serve two important purposes in the register transfer notation. First, they can be used to describe special storage locations within a target machine. Second, they can be used as functions that facilitate the task of describing complex operations that are not covered by the standard set of operators. A macro is represented by an identifier composed of upper-case alphabetic characters whose length is always two.

The `PC` macro represents the program counter on the target machine. This macro allows jump and branch instructions to be defined as assignments to the program counter. The `CC` macro is commonly used to represent the condition code register on machines that have a condition code register. This makes it possible to define instructions that modify and reference the condition codes. For historical reasons, the `ST` and `RT` macros define the input and output ports of a stack that is used to save the return addresses of the functions that perform external function calls.

Function macros are distinguished from non-function macros because they are followed by a list of expressions enclosed within square brackets and separated by commas. One of the most common uses of function macros is to perform type conversions. For example, the `IF`[] and `FI`[] function macros are commonly used to define integer-to-floating-point and floating-point-to-integer operations.

Expressions

Register transfer components can be combined with unary and binary operators to form expressions. Expressions containing binary operators use infix notation. No precedence is assigned to any of the operators and parentheses are used to indicate the evaluation order when two or more binary operations appear in the same expression. Unary operators usually precede their operand although, in the case of auto-increment and decrement, the placement of the operator before or after the register or memory location that it operates on indicates whether the operation happens before the value is obtained (pre-increment or pre-decrement) or after (post-increment or post-decrement).

Expressions can be used to form arbitrarily complex operations and addressing modes. Table 4 describes how address expressions are used to form several of the most commonly used addressing modes.

Register transfer	Addressing mode described
`M[_global_id]`	direct
`M[w[4]]`	register indirect
`M[dw[4]]`	pre-decrement
`M[w[4]i]`	post-increment
`M[w[4]+12]`	displacement
`M[w[4]*4]`	scaled
`M[(w[4]*4)+12]`	scaled displacement
`M[w[4]+w[7]]`	indexed
`M[w[4]+w[7]+12]`	indexed displacement
`M[(w[4]*4)+w[7]]`	scaled indexed
`M[M[_global_id]]`	memory indirect
`M[M[_global_id]+12]`	memory indirect displacement

Table 4: Sample address expressions

This table is not exhaustive since its intent is only to help the reader associate some common addressing modes with their corresponding register transfer representation and by doing so to illustrate how the basic elements can be combined to form any complex combination desired.

Instructions

Target machine instructions are described in terms of their effect on the storage locations on the target machine. An instruction in the register transfer notation is represented as a list of one or more assignments. Each assignment within an instruction is called an effect and is terminated with a semicolon. Consider a simple register-to-register move instruction. The following register transfer list (RTL) describes such an instruction:

```
w[4]=w[7];
```

This RTL describes an instruction that copies the value in register `w[7]` to register `w[4]`. A simple increment instruction is represented by:

```
w[4]=w[4]+1;
```

which describes an instruction that increments (adds one) the value in register `w[4]`. A three-address addition instruction is represented using the following RTL:

```
w[4]=w[5]+w[6];
```

which describes a instruction that adds the values in registers `w[5]` and `w[6]` and stores the results in register `w[4]`.

Instructions that change more than one storage location in the target machine are described using multiple effects. For example, if the add instruction described above also sets the condition code register, then the instruction would be described as follows:

```
w[4]=w[5]+w[6];CC=(w[5]+w[6]])?0;
```

Note that all of the expressions in the register transfer are evaluated before any assignments are made.

Branch instructions are described as an assignment of the program counter register. An unconditional jump instruction is represented by:

```
PC=L34;
```

which describes an unconditional transfer of control to the instruction labeled L34. Conditional branches are described as conditional assignments. The list operator is used to represent conditional assignment as follows:

```
PC=CC:0,L34;
```

In the above example, control is transferred to the instruction labeled L34 if the condition code register indicates that the last instruction that set the condition codes indicated that the result was a zero.

The external call stack, which is accessed through the `ST` and `RT` macros and might not exist on the target architecture, is used to describe the instructions that call external functions and return from them. For example, an external function call is represented as:

```
ST=_foo,w[4],w[5];
```

where `_foo` is the external function to call and registers `w[4]` and `w[5]` contain the values to pass to the external function and a return instruction is represented as:

```
PC=RT;
```

which describes a transfer of control back to the address currently residing on top of the calling stack.

Complex instructions that perform operations which are not likely to merge with other operations and form other valid machine instructions can be treated as "black boxes" by describing them as function macros. For example, the save instruction on the SUN SPARC [SUNM87], which makes a new register window available, is represented by:

```
w[14]=SV[w[14]+128];
```

In this example, the actual impact of the save instruction on the target machine is not clearly defined, but the register transfer is unique enough to distinguish it from any other instruction on the SPARC.

Floating-point to integer and floating-point to floating-point operations are complex enough to warrant the use of function macros. For example, the following register transfer list:

```
w[12]=FI[f[3]];
```

describes a floating-point to integer conversion operation. Other complex actions, such as extracting the high and low parts of a constant value, can also be handled by function macros. For example, the following sequence of RTLs:

```
w[7]=HI[_global_id];
w[7]=w[7]+LO[_global_id];
```

describe a pair of instructions that construct a 32-bit address by catenating its high and low parts together.

Internal representation

The register transfer notation examples given in the previous sections are presented in human-readable form. In actual practice, RTLs are encoded to reduce the amount of space needed to store them and the amount of time required to lexically scan them. A standard encoding notation has been defined for this purpose which encodes register, local identifiers and global identifiers into two character tokens using an extended 8-bit ASCII field [BENI91a].

BIBLIOGRAPHY

[AHO86] A. V. Aho, R.Sethi and J.D. Ullman, *Compilers—Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[AUSL82] M. A. Auslander and M. E. Hopkins, An Overview of the PL.8 Compiler, *Proceedings of the SIGPLAN Notices 1982 Symposium on Programming Language Design and Implementation*, Boston, MA, June 1982, 22-31.

[BAAS78] S. Baase, *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, Reading, MA, June 1978.

[BELA66] L. A. Belady, A Study of Replacement Algorithms for a Virtual-Storage Computer, *IBM Systems Journal*, *5*(2), April 1966, 78-101.

[BELL71] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

[BENI88] M. E. Benitez and J. W. Davidson, A Portable Global Optimizer and Linker, *Proceedings of the SIGPLAN Notices 1988 Symposium on Programming Language Design and Implementation*, Atlanta, GA, June 1988, 329-338.

[BENI89] M. E. Benitez, *A Global Object Code Optimizer*, MS Thesis, University of Virginia, Charlottesville, VA, January 1989.

[BENI91a] M. E. Benitez, *Register Transfer Standard*, Research Memorandum, University of Virginia, Charlottesville, VA, March 1991.

[BENI91b] M. E. Benitez and J. W. Davidson, Code Generation for Streaming: an Access/Execute Mechanism, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991, 132-141.

[BENI94] M. E. Benitez and J. W. Davidson, The Advantages of Machine-Dependent Global Optimization, *Proceedings of the International Conference on Programming Languages and System Architectures*, Zurich, Switzerland, March 1994, 105-124.

[BERN86] R. Bernstein, Multiplication by Integer Constants, *Software—Practice and Experience*, *16*(7), July 1986, 641-652.

[BRAD91] D. G. Bradlee, R. R. Henry and S. J. Eggers, The Marion System for Retargetable Instruction Scheduling, *Proceedings of the SIGPLAN Notices 1989 Symposium on Programming Language Design and Implementation*, Toronto, Ontario, June 1991, 229-240.

[BRIG89] P. Briggs, K. D. Cooper, K. Kennedy and L. Torczon, Coloring Heuristics for Register Allocation, *Proceedings of the SIGPLAN Notices 1989 Symposium on Programming Language Design and Implementation*, Portland, OR, June 1989, 275-284.

[CALL91] D. Callahan and B. Koblenz, Register Allocation via Hierarchical Graph Coloring, *Proceedings of the SIGPLAN Notices 1991 Symposium on Programming Language Design and Implementation*, Toronto, Ontario, June 1991, 192-203.

[CHAI81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein, Register allocation via Coloring, *Computer Languages*, *6*(1), January 1981, 47-57.

[CHOW83] F. C. Chow, *A Portable Machine-Independent Global Optimizer—Design and Measurements*, Ph.D. Dissertation, Stanford University, Stanford, CA, December 1983.

[CHOW88] F. C. Chow, Minimizing Register Usage Penalty at Procedure Calls, *Proceedings of the SIGPLAN Notices 1988 Symposium on Programming Language Design and Implementation*, Atlanta, GA, June 1988, 85-94.

[CYTR91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Transactions on Programming Languages and Systems*, *13*(4), October 1991, 451-490.

[DAVI80] J. W. Davidson and C. W. Fraser, The Design and Application of a Retargetable Peephole Optimizer, *ACM Transactions on Programming Languages and Systems*, *2*(2), April 1980, 191-202.

[DAVI81] J. W. Davidson, *Simplifying Code Generation through Peephole Optimization*, Ph.D. Dissertation, University of Arizona, Tucson, AZ, December 1981.

[DAVI84a] J. W. Davidson and C. W. Fraser, Register Allocation and Exhaustive Peephole Optimization, *Software—Practice and Experience*, *14*(9), September 1984, 857-865.

[DAVI84b] J. W. Davidson and C. W. Fraser, Code Selection through Object Code Optimization, *ACM Transactions on Programming Languages and Systems*, *6*(4), October 1984, 505-526.

[DAVI85] J. W. Davidson, *Simple Machine Description Grammars*, Computer Science Technical Report 85-22, University of Virginia, Charlottesville, VA, November 1985.

[DAVI86] J. W. Davidson, A Retargetable Instruction Reorganizer, *Proceedings of the SIGPLAN Notices 1986 Symposium on Programming Language Design and Implementation*, Palo Alto, CA, June 1986, 234-241.

[DAVI88] J. W. Davidson and A. M. Holler, A Study of a C Function Inliner, *Software—Practice and Experience*, *18*(8), August 1988, 775-790.

[DAVI91] J. W. Davidson and D. B. Whalley, Methods for Saving and Restoring Register Values across Function Calls, *Software—Practice and Experience*, *21*(2), February 1991, 149-165.

[DUNL78] D. D. Dunlop and J. C. Knight, Register Allocation in the SL/1 Compiler, *Proceedings of the 1978 LASL Workshop on Vector and Parallel Processors*, Los Alamos, New Mexico, September 1978, 205-211.

[FRAS92] C. W. Fraser and D. R. Hanson, Simple Register Spilling in a Retargetable Compiler, *Software—Practice and Experience*, *22*(1), January 1992, 85-99.

[FREI74] R. A. Freiburghouse, Register Allocation Via Usage Counts, *Communications of the Association for Computing Machinery*, *17*(11), November 1974, 638-642.

[GRAH82] S. L. Graham, R. R. Henry, R. A. Schulman, An Experiment in Table Driven Code Generation, *Proceedings of the SIGPLAN Notices 1982 Symposium on Compiler Construction*, Boston, MA, June 1982, 32-42.

[GOOD88] J. R. Goodman and W. C. Hsu, Code Scheduling and Register Allocation in Large Basic Blocks, *Proceedings of the 1988 International Conference on Supercomputing*, July 1988, 442-452.

[GUPT89] R. Gupta, M. L. Soffa and T. Steele, Register Allocation via Clique Separators, *Proceedings of the SIGPLAN Notices 1989 Symposium on Programming Language Design and Implementation*, Portland, OR, June 1989, 264-274.

[HENN83] J. L. Hennessy and T. R. Gross, Postpass Code Optimization of Pipeline Constraints, *ACM Transactions on Programming Languages and Systems*, *5*(3), July 1983, 422-448.

[HENN90] J. L. Hennessy and D. A. Patterson, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[HORW66] L.P. Horwitz, R. M. Karp, R. E. Miller and S. Winograd, Index Register Allocation, *Journal of the Association for Computing Machinery*, *13*(1), January 1966, 44-61.

[JOHN78a] S. C. Johnson, Yacc: Yet Another Compiler-Compiler, *Unix Programmer's Manual*, 2B, 19, July 1978, 1-34.

[JOHN78b] S. C. Johnson, A Portable Compiler: Theory and Practice, *Proceedings of the Fifth Association for Computing Machinery Symposium on Principles of Programming Language*, Tucson, AZ, January 1978, 97-104.

[JOHN86] M. S. Johnson and T. C. Miller, Effectiveness of a Machine-Level, Global Optimizer, *Proceedings of the SIGPLAN Notices 1986 Symposium on Programming Language Design and Implementation*, Palo Alto, CA, June 1986, 99-108.

[KANE87] G. Kane, *MIPS RISC Architecture*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[KERN78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[LAM88] M. S. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines, *Proceedings of the SIGPLAN Notices 1988 Symposium on Programming Language Design and Implementation*, Atlanta, GA, June 1988, 318-328.

[LEVE80] B. W. Leverett, R. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz and W. A. Wulf, An Overview of the Production-Quality Compiler-Compiler Project, *IEEE Computer*, *13*(8), August 1980, 38-49.

[LEVE81] B. W. Leverett, *Register Allocation in Optimizing Compilers*, Ph.D. Dissertation, Carnegie-Mellon University, Pittsburg, PA, February 1981.

[MCKE65] W. M. McKeeman, Peephole Optimization, *Communications of the Association for Computing Machinery*, *8*(7), July 1965, 443-444.

[MCKU84] M. K. McKusick, *Register Allocation and Data Conversion in Machine Independent Code Generators*, Ph.D. Dissertation, University of California, Berkeley, CA, December 1984.

[MOTO85] Motorola, *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[PERK79] D. R. Perkins and R. L. Sites, Machine-Independent Pascal Code Optimization, *Proceedings of the SIGPLAN Notices 1979 Symposium on Programming Language Design and Implementation*, Denver, CO, August 1979, 201-207.

[PROE92] T. A. Proebsting and C. N. Fischer, Probabilistic Register Allocation, *Proceedings of the SIGPLAN 1992 Symposium on Programming Language Design and Implementation*, San Francisco, CA, June 1992, 300-310.

[SETH70] R. Sethi and J. D. Ullman, The Generation of Optimal Code for Arithmetic Expressions, *Journal of the Association for Computing Machinery*, *17*(4), October 1970, 715-728.

[SITE79] R. L. Sites, Machine-Independent Register Allocation, *Proceedings of the SIGPLAN Notices 1979 Symposium on Programming Language Design and Implementation*, Denver, CO, August 1979, 221-225.

[STAL89] R. M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, Cambridge, MA, 1989.

[STEE61] T. B. Steel, A first version of UNCOL, *Western Joint Computer Conference Proceedings*, May 1961, 371-378.

[SUNM87] Sun Microsystems, *The SPARC Architecture Manual*, Version 7, Sun Microsystems Corporation, Mountain View, CA, 1987.

[TANE83] A. S. Tanebaum, H. V. Staveren, E. G. Keizer and J. W. Stevenson, A Practical Tool Kit for Making Portable Compilers, *Communications of the Association for Computing Machinery*, *26*(9), September 1983, 654-660.

[WALL86] D. W. Wall, Global Register Allocation at Link Time, *Proceedings of the SIGPLAN Notices 1986 Symposium on Programming Language Design and Implementation*, Palo Alto, CA, June 1986, 264-275.

[WHAL90] D. B. Whalley, *Ease: An Environment for Architecture Study and Experimentation*, Ph.D. Dissertation, University of Virginia, Charlottesville, VA, May 1990.

[WOLF91] M. E. Wolf and M. S. Lam, A Data Locality Optimizing Algorithm, *Proceedings of the SIGPLAN Notices 1991 Symposium on Programming Language Design and Implementation*, Toronto, Ontario, June 1991, 30-44.