Diss. ETH Nr. 11024

Optimizing Compilers for Structured Programming Languages

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH Zürich)

for the degree of
Doctor of Technical Sciences

presented by
Marc Michael Brandis
Dipl. Informatik-Ing. ETH
born January 7, 1966
citizen of Germany

accepted on the recommendation of
Prof. Dr. H. Mössenböck, examiner
Prof. Dr. N. Wirth, co-examiner

1995

Für meine Eltern

Acknowledgements

This work would not have been possible without the help and encouragement of many people. I am most indebted to Prof. H. Mössenböck for his continuous support and his insistence on clarity and simplicity. Without his constructive criticism, this project would not have achieved such favorable results. I also would like to thank Prof. N. Wirth for a liberal supervision of this project and for acting as my co-examiner. His way of thinking and problem solving had a significant impact on this work.

It is a pleasure to mention my colleagues of the Institute for Computer Systems; they established an agreeable and inspiring environment and many of them contributed to this work. Régis Crelier, Michael Franz, and Josef Templ participated in our initial effort to port Oberon to stock hardware. Robert Griesemer was a well-versed discussion partner in compiler-related topics as well as in programming methodology. Régis Crelier wrote the Oberon-2 front-end which became part of both Oberon-2 compilers I implemented. I also gratefully acknowledge all the other support I have received over the years from my colleagues at ETH, which is just too numerous to be mentioned in detail.

Cheryl Lins provided many helpful hints and literature pointers when I got started in the field of optimizing compilers. My internship with Dr. Peter Oden and Jonathan Brezin of the IBM TJ. Watson Research Laboratory resolved many of my questions about optimizing compilers. The chance to study their compiler was a tremendous help in starting my project. Dr. Martin Reiser of the IBM Research Laboratory in Rüschlikon supported my work by organizing the before mentioned internship and by making the right connections into IBM.

Several students contributed to this work in their diploma theses. Jürg Bolliger developed a Trace Scheduling algorithm for our optimizing compiler. Jakob Magun constructed a novel register allocator based on cyclic interval graphs. Thomas Nadig showed that single-pass generation of SSA-form is feasible for the whole programming language Oberon-2. In his semester project, David Posva implemented the algorithm to replace unstructured control-flow by structured code.

I also would like to thank Jürg Bolliger, Stephan Gehring, Martin Gitsels, Stefan Ludwig, Niklaus Mannhart, and Whitney de Vries for proofreading parts of this thesis. Their comments greatly helped me in improving the quality.

Contents

Abstract

Modern processor architectures rely on optimizing compilers to achieve high performance. Such architectures expose details of their hardware to the compiler, which has to deal with them in generating machine code. This development has led to complex and slow compilers, which are difficult to understand, implement, and maintain.

This thesis reports on methods to simultaneously reduce the complexity and the compile-time of optimizing compilers by more than a decimal order of magnitude. It presents a novel intermediate program representation, which integrates data- and control-flow into a single data-structure. This provides not just for simpler and faster optimization algorithms, but also for more powerful optimization techniques. The thesis also describes single-pass algorithms to construct this intermediate program representation from structured source code, as well as single-pass techniques to transform programs with restricted kinds of unstructured control-flow like in Oberon into structured form. The integration of these techniques with the parser allows to implement fast and compact front-ends for structured programming languages, that avoid the many auxiliary data structures other optimizing compilers require.

A description of several optimization algorithms and how they can be implemented on this intermediate program representation shows the feasibility of the approach. Most of these techniques have been implemented in a prototypical optimizing compiler translating a subset of the programming language Oberon for the PowerPC architecture. Measurements on this compiler prove that both the complexity and the compile-time of optimizing compilers can be reduced by an order of magnitude when translating a structured programming language and when using this novel intermediate representation and the associated algorithms. The thesis concludes with some feedback to the programming language designers, which language constructs cause undue complications in optimizing compilers and should therefore be omitted.

Kurzfassung

Moderne Prozessorarchitekturen erfordern optimierende Compiler um hohe Rechenleistung zu erreichen. Details der Hardware-Implementierung wurden für den Compiler sichtbar gemacht, und letzterer muss diese Hardwareeigenschaften bei der Codeerzeugung berücksichtigen. Diese Entwicklung führte zu komplexen und langsamen Compilern, die schwer zu verstehen, zu implementieren und zu warten sind.

Diese Dissertation beschreibt Methoden, die gleichzeitig die Komplexität und die Übersetzungszeit optimierender Compiler um mehr als eine dezimale Grössenordnung reduzieren. Sie präsentiert eine neuartige Zwischenrepräsentation für Programme, die Daten- und Kontrollfluss in eine einzige Datenstruktur zusammenfasst. Dies erlaubt nicht nur einfachere und schnellere Optimierungsalgorithmen, sondern auch leistungsstärkere Optimierungsverfahren. Die Dissertation beschreibt auch Ein-Pass-Algorithmen, um diese Zwischenrepräsentation aus strukturierten Quellprogrammen zu erzeugen, als auch Ein-Pass-Verfahren um Programme mit eingeschränkten Formen von unstrukturiertem Kontrollfluss wie in Oberon in strukturierte Form zu übersetzen. Die Integration dieser Verfahren mit dem Parser erlaubt es, schnelle und kompakte Front-Ends für strukturierte Programmiersprachen zu bauen, welche die vielen Hilfsdatenstrukturen anderer optimierender Compiler vermeiden.

Die Beschreibung einiger Optimierungsalgorithmen und deren Implementation auf dieser Zwischenrepräsentation belegt die Anwendbarkeit des Ansatzes. Die meisten dieser Techniken wurden in einem Prototyp-Compiler realisiert, der eine Untermenge der Sprache Oberon für die PowerPC Architektur übersetzt. Messungen an diesem Compiler beweisen, dass sowohl die Komplexität als auch die Übersetzungszeit optimierender Compiler um eine Grössenordnung reduziert werden können, wenn eine strukturierte Programmiersprache übersetzt wird, und wenn diese neuartige Zwischenrepräsentation und die dazugehörigen Algorithmen verwendet werden. Die Dissertation schliesst mit einigen Hinweisen an die Entwickler von Programmiersprachen, welche Sprachkonstrukte übermässige Schwierigkeiten bei Bau optimierender Compiler machen und deshalb vermieden werden sollten.

1 Introduction

1.1 Motivation

The motivation for the work presented in this thesis stems from the recent shift in the interface between compiler and processor architecture, that started with the introduction of RISC architectures. Instead of trying to make complex instructions run fast, these architectures provide simple instructions only, out of which more complex operations can be assembled. It is the task of the compiler to select fast code patterns and to customize the patterns for often-encountered special cases. Compilers doing so are called optimizing compilers.

On early RISC processors, the biggest source of improvement in code quality was to allocate variables in the large register set. The portable Oberon-2 compiler OP2 uses a simple heuristic to allocate local variables to registers. This technique alone allowed to achieve very competitive code quality on most RISC processors [BCFT92]. On the IBM RS/6000, however, code produced by the optimizing C-compiler ran twice as fast as Oberon code. This difference can mostly be attributed to the RS/6000 exploiting instruction-level parallelism in the form of parallel execution units and in the form of instructions performing multiple operations at once. The compiler must generate code that makes good use of these features.

While the optimizing C-compiler on the RS/6000 generates code of very high quality, it does so at a high cost. The compiler is a large and complex application that compiles about two orders of magnitude slower than the Oberon-2 compiler. The work presented here is a step towards achieving similar code quality at much lower cost. Even though the project started on the RS/6000, the results are also relevant to other RISC architectures, which make use of instruction-level parallelism in their latest incarnations and which rely on the compiler to generate code of high quality.

1.2 Contributions

At the heart of an optimizing compiler is the intermediate representation of the program, on which all code-improving algorithms operate. This thesis innovates on the intermediate program representation in several respects. It presents a novel representation called *guarded single-assignment (GSA) form* with very clean semantics, which makes all dependencies explicit. The intermediate representation provides completely position-independent semantics, and integrates data- and control-flow information into a single data-structure. This not only simplifies keeping the data-structure consistent, but also enables more powerful and simpler optimization algorithms.

Another contribution of this thesis are single-pass algorithms to construct GSA or similar intermediate representations for programs in structured programming languages. Previously known methods to generate intermediate representations of similar expressiveness require many passes and several auxiliary data structures. The thesis also presents a single-pass technique to automatically transform programs with restricted forms of unstructured control-flow into structured programs. This technique can be integrated with the algorithms to generate GSA form. It is thus possible to translate programs in the language Oberon-2 into GSA form in a single pass during parsing of the source code.

Using GSA form and these single-pass algorithms, we have built a prototypical optimizing compiler for a subset of Oberon, which is more than an order of magnitude smaller and faster than industrial optimizing compilers, but which still achieves competitive code quality.

Finally, this thesis provides some feedback to programming language designers, on which language features complicate the design of an optimizing compiler, and it presents an overview of the large field of optimizing compilers.

1.3 Overview

Chapter 2 introduces into the field by discussing what an optimizing compiler is, and what code improvements one can expect it to perform.

Chapter 3 presents the most important developments in computer architecture that affect compilers. The corresponding hardware features must be taken into account when generating fast code for modern machines.

Chapter 4 describes our target architecture, the PowerPC. It also discusses two hardware implementations in detail, and how they implement the techniques presented in Chapter 3.

Chapter 5 discusses different design options for intermediate program representations, and Chapter 6 describes our intermediate representation GSA form in detail.

Chapter 7 presents our single-pass algorithms to generate GSA form and how to deal with unstructured source programs. A discussion of alias analysis concludes the part about intermediate representations.

Chapter 8 describes several optimization algorithms on GSA form as we have implemented them in our prototype compiler. It also outlines several other optimizations, that may be beneficial to add in the future. Chapter 9 presents how the intermediate program representation can be mapped to the actual machine, namely by ordering instructions in a so-called instruction scheduler and by assigning registers to values in the register allocator.

Chapter 10 presents some measurements on our prototypical optimizing compiler OOC2. Chapter 11 concludes the thesis and presents some implications of this work on programming language design.

Instead of describing related work in one place, we have mentioned it wherever appropriate throughout the thesis. Due to the large variety of different aspects dealt with in this thesis, we consider this to be more easy to follow.

2 What is an Optimizing Compiler?

First of all, an optimizing compiler is a compiler, i.e. an application translating a program written in a high-level language into executable machine code for a given machine. Everything that we expect from a compiler also applies to an optimizing compiler, namely correctness of the translation, detection of errors, and compilation speed high enough to make it suitable for daily work.

What distinguishes an optimizing compiler from other compilers is that it uses techniques to improve the quality of the generated code. It is a common misunderstanding that such compilers generate optimal code. As most code optimization problems are NP-complete, optimizing compilers use heuristics to improve commonly found code patterns, but cannot guarantee any kind of optimality. Often they cannot even assure that the modified code is better than the original one. One can expect to get a measurable improvement on most programs, however.

These techniques necessarily make the compiler more complex and increase the compilation time. This is acceptable if the reward is higher code quality. The engineering decision is one of cost – increased compilation time and complexity – versus improvement. In some environments where compilation speed is important, one may only want to use very simple optimizations, while in other environments the quality of the generated code may be so important that even optimizations taking hours to complete and yielding only small improvements may be acceptable.

2.1 Structure of an Optimizing Compiler

An optimizing compiler is a complex software system. In order to ensure correctness and reliability, it is very important to structure it well, and to decouple the individual parts. Figure 2.1 shows the general structure of an optimizing compiler.

The front end parses the source text and generates an internal representation of the program. This internal representation contains all information about the source program that is necessary to generate correct object code. The back end translates this internal representation into executable machine code and writes it together with linkage information to an object file. Optionally, a set of optimization algorithms can be called before the back end is run. Each of these algorithms transforms the internal representation in a way that the semantics of the original program are preserved. It is advantageous to design a single internal representation common to all optimization algorithms rather than a set of representations specific to each optimization. Besides simplifying the design and the maintenance of the compiler, this allows optimization algorithms to be applied selectively or even iteratively without the need for additional phases to convert between representations.

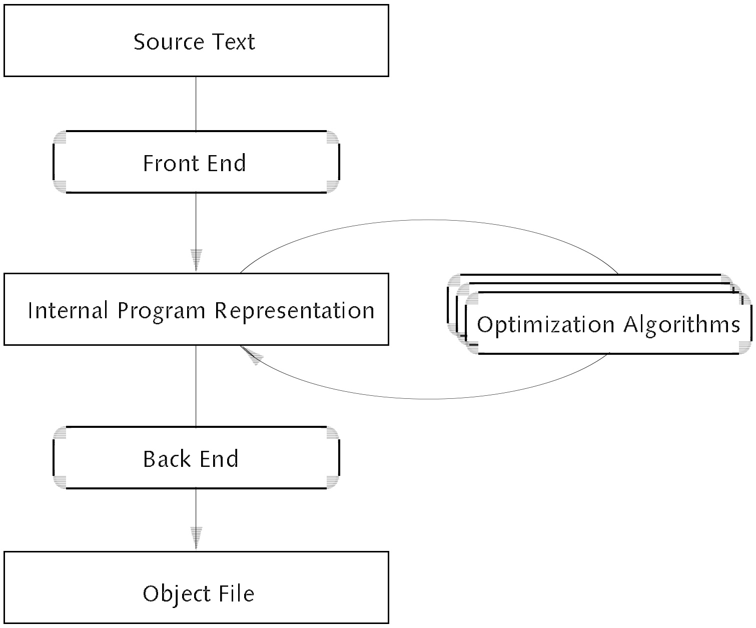


Figure 2.1: Structure of an optimizing compiler.

2.2 Optimization Algorithms

The goal of an optimization algorithm is to transform a program into another one, which is semantically equivalent, but according to some criteria better than the original program. Possible criteria are the execution speed or the size of the generated code.

Nowadays, optimizing compilers for workstations mainly optimize for execution speed. Usually, they even tolerate a certain amount of code size increase if the execution time can be reduced. However, even though main memories are large enough to store the code of very large programs, the negative effect of larger code on instruction cache performance requires to keep the code size within reasonable bounds.

When compiling code for embedded systems, keeping the memory requirements low is usually more important than pure execution speed. Compilers for such environments optimize for code size and for data size by packing data. We will not further explore these topics in this thesis.

Before discussing different kinds of optimization algorithms, a definition of semantic-preserving transformations is required.

Definition: The *global state* of a system is defined by all values stored in global variables and by the output found on the output devices.

Definition: Two programs *P* and *Q* are *semantically equivalent*, if executing them on the same global input state with the same input data will leave the same global output state after the program has terminated, or if the same set of exceptions have been raised.

This definition of semantically equivalent programs leaves a lot of freedom to reorder operations. It is only required that a correctly terminating program generates the same output state, but the order in which individual variables are written is left open. Furthermore, if program *P* does not terminate correctly but raises an exception, it is only required that *Q* does not terminate correctly either, and that it raises any of the exceptions that *P* may have raised. Nothing is said about the global state in case the program causes an exception. This corresponds to an imprecise exception model as found on many modern processors, e.g. on the DEC Alpha [DEC92] or to some extent on the PowerPC [IBM94a].

This definition of equivalence is sufficient but not required from the view of a programming language designer. Language definitions leave some behavior undefined in order to allow for different implementations. For example, Oberon [Wirth88] does not specify the order in which actual parameters of a procedure call are evaluated. However, some order is selected when translating to the intermediate representation. Enabling the optimizer to make use of the nondeterminism would require to keep more high-level information in the intermediate representation and probably would not offer significant benefits.

Definition: A *semantics-preserving optimization algorithm* is an algorithm, which transforms a program *P* into a semantically equivalent program *Q*.

We do not require *Q* to be better than *P* according to some criterion. It could very well be that the algorithm fails to improve a particular program, or in rare cases that it even degrades a program. Complicated interactions between different optimizations and the dynamic behavior of programs makes it very hard to tell whether a transformation yields an improvement in the final machine code in all cases. For example, some loop transformations may only be beneficial if the loop is executed a large number of times. Usually, compiler writers use an experimental approach: By running their optimizer over a large number of source programs, they find heuristics that deliver good results in these cases, and hope that other programs include similar patterns.

Guiding the optimizer by estimating the impact of a transformation on performance is a recently defined research topic. The most difficult problem is finding good estimates for execution frequencies affecting the execution time. Wang uses a symbolic representation of execution cost, avoiding to predict unknown behavior as long as possible [Wang94]. Wagner et. al define a statistical model over predicted branch frequencies in order to obtain an execution frequency estimate [WMGH94].

2.3 Scope of Optimization

Optimization algorithms can consider different scopes in which they search for optimizable patterns. Some compilers only optimize at the instruction level, that is, they only replace single instructions by others that achieve the same effect. As an example, multiplications by powers of two can be expressed as shifts. Others optimize at the level of simple statements, basic blocks, extended basic blocks, procedures, or the whole program. The larger the scope of an optimizing transformation, the better the result becomes, sometimes at tremendous costs in compile time. Some algorithms used in optimizing compilers have a complexity of $O(N^3)$ or $O(N^4)$, where N is the number of instructions in the considered scope. Using them on anything but a very small scope is impractical when compilation time is an issue. Whether a less-sophisticated algorithm running on a larger scope yields better results is a matter of empirical tests.

Historically, algorithms that work on basic blocks are called *local optimizers*, whereas algorithms considering whole procedures are called *global*. There have been attempts at optimizing across procedure boundaries in so-called *interprocedural* optimizers, but separate compilation imposes a natural barrier on this. Either the optimizer has to be run as part of the linking process like with the MIPS compilers at the highest optimization level, or pessimistic assumptions have to be made for external objects. This, the high complexity, and the large costs in run-time have so far hindered widespread use of such techniques.

2.4 Where Does the Improvement Come From?

According to Hennessy and Patterson [HePa90], the execution time of a program on a computer is determined by three factors: The number of instructions executed, the average number of cycles an instruction takes to execute, and the clock cycle time. A programmer or a compiler can influence the first two factors – the number of instructions executed, and the average number of cycles per instruction, i.e. the smooth flow of instructions through a processor pipeline. This is what an optimizing compiler tries to do in order to reduce the execution time.

The first factor translates into a simple set of rules: Reduce the number of instructions executed by computing as much as possible at compile-time, avoid recomputing values that have already been computed, and eliminate instructions that do not contribute to the final result. This can be implemented in an optimizing compiler using the following techniques [ASU86].

Constant Folding and Constant Propagation: If the operands of an instruction are all constant, so are the results of this instruction. Compute these constant results, and replace the instruction by them. We speak of constant propagation if such newly found constants are used to find further constant expressions.

Copy Propagation: Remove assignments of one object to another, and instead use the original object whenever possible.

Common Subexpression Elimination: Find groups of instructions that compute the same value, and replace some of them by the results of others, thus avoiding their recomputation.

Code Motion: Move operations to places where they are less frequently executed. The most useful optimization of this kind is the motion of loop-invariant code out of a loop.

Peephole Optimizations: Scan the code for patterns of instructions that could be expressed with fewer instructions on the target machine, and replace them. A prominent example of this are the floating-point multiply-and-add instructions of the PowerPC which replace a multiplication followed by an addition.

Dead Check Removal: Delete run-time checks which can be shown to never fail. Examples in Oberon could include index checks, NIL checks, or type guards.

Dead Code Elimination: Delete instructions that are never executed, or that do not contribute to the final result.

The second factor corresponds to avoiding code patterns that are expensive on a certain machine. The number of cycles per instruction can be reduced by selecting cheap instructions over expensive ones, by reordering instructions in order to avoid pipeline interlocks, or by reorganizing the data layout in order to remove cache misses. Note that all these optimizations are very closely related to a certain processor *implementation* and not just its architecture as the previous techniques.

Peephole Optimizations: Find expensive operations, and replace them by cheaper ones achieving the same effect.

Strength Reduction: Similarly to peephole optimizations, find repeated uses of expensive operations and replace them by cheaper ones. A typical example of this is to replace an index used to step through an array by a pointer, thus saving the cost of multiplying the index with the element size in each iteration.

Instruction Scheduling: Reorder instructions in order to reduce the number of pipeline interlocks on pipelined and superscalar processors.

Cache Optimizations: Reorder data and code in memory and reorder data accesses, so that locality of reference is improved and the number of cache misses is reduced.

2.5 Anatomy of an Optimization Algorithm

In order to get an idea on how an optimization algorithm works, and what important properties of an intermediate program representation are, we present an intuitive description of *common subexpression elimination (CSE)*. Common subexpression elimination is the optimization of finding operations that compute the same result – so-called common subexpressions – and instead of recomputing the result, keeping the original result available and deleting the second operation. We call instructions computing the same result *equivalent*. Equivalence is undecidable in general, so we base our common subexpression elimination algorithm on the weaker notions of congruence and availability.

Definition: Two computations yield *congruent* results, iff their opcodes are the same and all their corresponding operands are congruent. Constant operands are congruent, iff they are identical.

Congruence implies equivalence, that is, if two values are congruent, they are equivalent. However, there may be computations yielding equivalent results which are not congruent. For example, the expressions *a+b* and *b+a* are equivalent but not congruent.

If the same variable names are found as operands, the operands are not necessarily congruent. A variable name serves as a placeholder for the result of a computation, and during execution of a program, the same variable may refer to different computations. An optimizer has to keep track of which computations correspond to a variable at each point in the program, and only if they are congruent, the operands are congruent as well. This will be further discussed in the section on data-flow analysis.

Definition: The result of some computation *C* is *available* at point *P* in a program with starting point *S*, iff every path from *S* to *P* includes computation *C*.

Intuitively, requiring some result to be available at a point P means making sure that the computation has been performed, no matter which control-flow path to P has been taken. Note that availability is computed from control-flow, while congruence is computed from data-flow.

Optimization algorithms are based on *transformation rule*s of the following form: If precondition P is fulfilled, then perform transformation T. In common subexpression elimination, the preconditions are congruence and availability.

Transformation rule (CSE): Let C_0 and C_1 be two computations. If C_0 and C_1 are congruent and C_0 is available at the point of C_1, then replace all uses of C_1 by uses of C_0.

Note that C_1 will not have any use after the transformation. Such operations are called *dead code* and can be deleted. The removal is not strictly a part of common subexpression elimination, but can be performed at the same time.

In the following Oberon program, there are some common subexpressions.

```
VAR
   i, w, x, y, z: INTEGER;
BEGIN
   ...
   IF cond THEN w := i * 4 END;
   x := i * 4;
   y := i * 4;
   i := y + 10;
   z := i * 4
```

There are several computations of *i * 4*, which we will refer to by the variable name they are assigned to. The programmer would have been able to avoid the recomputations in this example, but this is not always possible. Such multiplications may have been generated by the compiler as part of the address computation for array accesses, and there would have been no way for the programmer to eliminate the common subexpressions by hand.

Applying the notion of congruence, it may be found that the multiplications of *i* by 4 in the computations *w*, *x*, and *y* are congruent. Even though the computation of *z* is lexically the same, it is not congruent with the others. The operand *i* has been changed in between and thus corresponds to a different value. That is, *i* at computation *y* and *i* at computation *z* are not congruent, hence *y* and *z* are not congruent either.

There is a path avoiding computation w, namely when *cond* evaluates to FALSE. In this case computation w will not be executed. Thus, w is not available at x, but x is available at y and later on. The transformation rule can be applied to x and y, yielding the code below. Note that a temporary variable t_0 has been introduced to keep the result, which will not be assigned elsewhere. There might be other assignments to x, preventing x from replacing uses of y.

```
IF cond THEN w := i * 4 END;
$t_0$ := i * 4; x := $t_0$;
y := $t_0$;
i := $t_0$ + 10;
z := i * 4
```

So far, one multiplication has been removed. So-called copy assignments of one variable to another as for x will be removed later in an optimization called copy propagation. By moving computation t_0 in front of the If-statement, its result would become available at w, and computation w would become superfluous. The preconditions required to perform this transformation will not be discussed in further detail here, but obviously they would have to be checked beforehand.

```
$t_0$ := i * 4;
IF cond THEN w := $t_0$ END;
x := $t_0$;
y := $t_0$;
i := $t_0$ + 10;
z := i * 4
```

Several important properties of an intermediate program representation can be derived from this example. As will be seen throughout this thesis, the requirements of common subexpression elimination are prototypical for a wide class of optimization algorithms.

- It should be simple to inspect operations and operands.
- It must be possible to distinguish between different values a variable may take on during program execution. More precisely, it should be easy to determine which assignments affect the value seen at a certain place, and how they affect it.
- It must be possible to find out which operations are executed under which conditions, and whether their results are available at a certain point.
- It should be simple to traverse all uses of a result, constant, or variable.

We will return to these properties in Chapters 5 and 6, in which different design options for intermediate representations will be explored.

3 Advances in Computer Architecture

Compilers are the link between the high-level programming language and the actual machine. As such, they are affected by advances in either programming language design or computer architecture. Optimizing compilers in particular are heavily affected by computer architecture, as they are expected to exploit the facilities of the machine. Understanding the common hardware techniques to improve processor performance allows to design compiler optimizations applicable to a large class of machines.

In this chapter, we will discuss the most prominent techniques used to speed up processors on the architectural side, namely pipelining, multiple execution units, and caching. Finally, it will be shown why RISC processors are well-positioned to exploit these methods, and what model they present to the compiler writer.

3.1 Pipelining

In a typical modern computer, instruction execution can be partitioned into five sequential phases: instruction fetch, decode, execute, complete, and result write. Pipelining is based on the simple observation that while an instruction is in some phase of execution, the hardware implementing the other phases is idle. Trying to use it for other instructions should yield a significant performance increase at very little additional cost. Ideally, while instruction *n* writes back its result, instruction *n+1* is in the completion phase, instruction *n+2* is in the execute phase, and so on. The hardware can be considered as

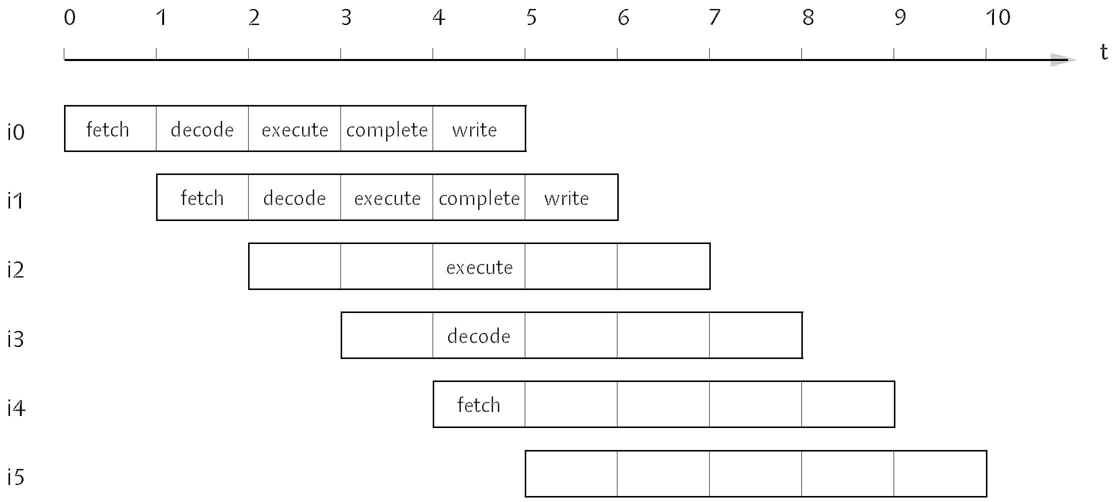


Figure 3.1: Instruction flow through a pipeline.

a *pipeline*, through which instructions flow and in which some operation is performed on the instruction at each *stage*. Figure 3.1 shows the instruction flow through such a pipeline.

Compared to performing the steps on each instruction sequentially, the throughput has been improved by a factor of five. There are some barriers to achieving this improvement in practice, however. The work done in each stage must be about the same, as the slowest stage determines the speed of the whole pipeline. In order not to mix up different instructions and to bridge timing gaps between stages, buffering has to be introduced between stages, increasing total execution time. Finally, some additional logic to control the flow of instructions through the pipeline and to detect *hazards* is required.

A *pipeline hazard* is a situation in which one or more instructions cannot be advanced to the subsequent stages. Consider the following code sequence.

```
(i1)  add   r1 := r2, r3
(i2)  add   r4 := r1, r5
```

Assume that in cycle *t*, the first instruction is in the execute stage, and the second in the decode stage. For the second instruction to enter the execute stage in cycle *t+1*, the source operands *r1* and *r5* must be available. However, *r1* will not be written by the first instruction till cycle *t+2*, and thus, the second instruction must be delayed until cycle *t+3* to enter the execute stage. Such delays are also called *pipeline bubbles*, as they can be considered as no-operations flowing down the pipeline. In this example, two bubbles would proceed through the pipeline, drastically reducing performance (Figure 3.2).

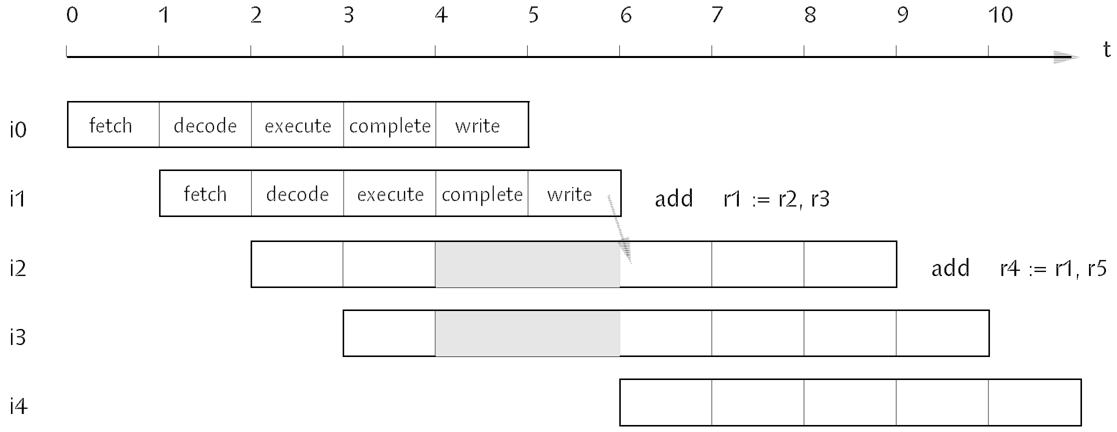


Figure 3.2: A hazard between i1 and i2 causes two pipeline bubbles.

As the above example is a very common case, and as the result of the first addition would be available at the end of the execute stage, pipelined computers implement a *forwarding path* for results (Figure 3.3). Results from the ends of the execute and completion stages can be fed directly into the execute stage, avoiding the above delay. For some instructions, e.g. for loads, the result is only available at the end of the completion stage, and a pipeline bubble cannot be avoided if the next instruction uses the result. We call the number of cycles between an instruction entering the execute stage and its result becoming available the *latency* of the instruction. In the above example, the latency of an add instruction is one, while for a load it is two.

Furthermore, when a branch is in the execute stage, the two subsequent instructions have already been fetched before the fetcher can be redirected to the branch target. The fetched instructions have to be discarded, causing two bubbles to flow down the pipeline. In order to reduce the adverse effect of branches on performance, branches are usually executed in the decode stage already. This requires the results of instructions setting condition codes to be forwarded to the decode stage, and still leaves one

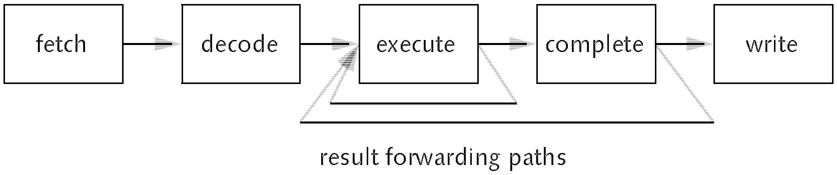


Figure 3.3: Forwarding paths.

instruction fetched after the branch, which is not used if the branch is taken. The idea to execute the fetched instruction anyway led to the concept of *delayed branches*, where the instruction following a branch is executed no matter whether the branch is taken or not. It is the compiler's task to place an instruction after the branch, which does useful work independently of the branch direction or that at least does not invalidate a required result. In the worst case, a no-operation has to be inserted after the branch.

By reordering instructions, such *branch-delay slots* can usually be filled with useful instructions, and other pipeline hazards like the delay between a load and uses of the loaded value can be avoided. This reordering performed in optimizing compilers is called *instruction scheduling* and will be discussed in detail in chapter 9.

We assumed that hardware detects when an instruction needs an operand which is not yet available, and delays the instruction appropriately. This feature is called *hardware interlocking*. One of the earliest RISC processors, the Stanford MIPS processor (*M*icroprocessor without *I*nterlocked *P*ipeline *S*tages) [Kane87] did not include such hardware and relied on the compiler to reorder instructions appropriately or to insert no-operations. On this processor, instruction scheduling was not just a question of achieving good performance, but also a matter of correctness of the generated code.

The number of stages in a pipeline does not have to be five, but can be any number. Splitting a computation into many small stages (*superpipelining*) allows the clock cycle time to be reduced as every stage has less work to perform, and thus increases the theoretical throughput. However, the additional buffering between stages does also increase the total time an instruction takes to execute, and many instructions may not have a latency of one any more. The latter will cause additional pipeline bubbles between subsequent dependent instructions and decrease performance, unless the increased latencies can be hidden by reordering instructions.

3.2 Superscalar Microarchitecture

One idea behind pipelining is to start executing a new instruction every clock cycle, as long as there are no hazards, and thus achieving a CPI (*cycles per instruction*) ratio close to one. Superscalar architectures are the answer to the innocent question of why to stop at a CPI of one, and not to go below that limit. They can execute more than one instruction each cycle through the use of multiple execution pipelines, also called *execution units*.

A superscalar processor fetches and decodes several instructions every clock cycle, and it dispatches the instructions to the appropriate execution units. Usually, there are different specialized execution units for different kinds of instructions. For example, the PowerPC 601 (Figure 3.4) contains an integer-unit for integer operations, a floating-point-unit for floating-point operations, and a branch-unit for branches [PPC601]. While the distinction between integer- and floating-point operations may seem natural, the separation of branches may not. Reconsider that branches were treated differently in the above pipeline, and that they work on different resources than other operations, namely condition codes and the PC register controlling the instruction fetcher. These properties call for the introduction of a separate branch-unit.

Ideally, one instruction is executed in each pipeline in every clock cycle, achieving a CPI of 1/3 for three units. This is only possible, if for each unit, there are exactly the same number of instructions and they are ordered appropriately. In practice, this CPI is only achieved for very small sections of code. Moreover, there can not only be hazards within one pipeline, but also hazards between pipelines. For

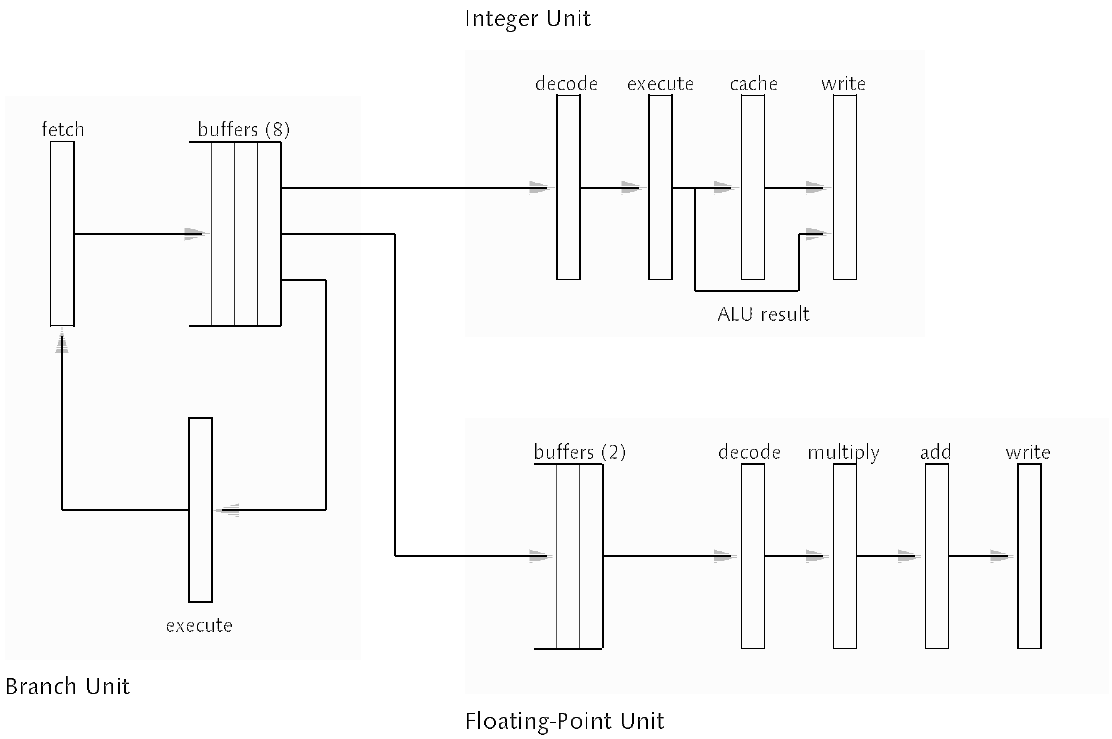


Figure 3.4: Pipeline structure of the superscalar PowerPC 601 microprocessor.

example, in the PowerPC 601, conditions are computed within the integer- and floating-point units, but conditional branches depending on them are executed in the branch-unit. The latency between an integer-compare starting execution and its result becoming available in the branch-unit is two cycles, and it is again the task of an instruction scheduling algorithm to avoid hazards by placing other instructions in between. The difference to scheduling for a single pipeline is that for each cycle, an instruction for every execution pipeline should be emitted instead of just for one pipeline.

A scalar processor must only check whether an instruction to be dispatched is dependent on other instructions currently being executed. A superscalar processor also has to check whether there are dependencies to other instructions to be dispatched in the same cycle. There are three kinds of dependencies to consider, as shown in Figure 3.5.

data	anti	output
add r1 := r3, r4	add r3 := r0, r4	add r0 := r3, r4
add r0 := r1, r2	add r0 := r1, r2	add r0 := r1, r2

Figure 3.5: Types of dependencies.

A *data-dependence* between two instructions corresponds to the first instruction writing a result that the second instruction needs. The second instruction must be executed after the first, and only after the result of the first has become available. There is an *anti-dependence* between two instructions if the first instruction reads an operand that the second instruction writes. The first instruction must read its operands before the second writes them. Whether this prohibits the two instructions from being executed in parallel depends on when these instructions read and write operands, but the second instruction cannot be executed before the first one. An *output-dependence* between two instructions corresponds to both instructions writing the same resources. As for anti-dependencies, this may or may not prevent the instructions from being executed in parallel. Hardware mechanisms to discard the first result in case of output-dependencies are usually not worth their cost. Note that in detecting pipeline hazards above, we have only considered data-dependencies, as the others play a minor role in single pipelines.

Data-dependencies are also referred to as *true dependencies*. The other dependencies are an artefact of allocating the same register to different values, and thus could be removed by using different registers, as exemplified in Figure 3.6.

anti	output
add r3 := r0, r4	add r0 := r3, r4
add *r5* := r1, r2	add *r5* := r1, r2

Figure 3.6: Removed dependencies through renaming.

Techniques to avoid such dependencies are called *renaming* techniques, and can be implemented either in hardware or in software, i.e. by the compiler. Many optimizing compilers include some sort of renaming.

In hardware implementations, renaming is either used in order to run code compiled for other implementations well, to hide the problems of small register files, or to simplify the implementation of *speculative execution*. Note that register renaming in hardware requires that there are more physical registers than visible to the programmer.

For example, the POWER-2 [IBM94b] implementation of the POWER architecture uses register renaming in order to make more instructions executable in parallel through its dual integer- and dual floating-point-pipelines, even for code optimized for the earlier POWER-1 implementation which only had one integer- and one floating-point-pipeline. The PowerPC 604 [PPC604] uses register renaming for the same reason, and additionally to provide a write location for speculative operations. When the CPU executes a conditional branch for which the condition is not yet available, it predicts the branch to be either taken or not taken, and continues to fetch and execute instructions along that path (see below). These instructions are called *speculative operations*, as their execution is based on the speculation that the branch takes a certain path; if the prediction turns out to be wrong and the branch proceeds along the other way, the effect of such speculative operations must be cancelled. This is simpler if the results have been stored in temporary rename registers.

The NexGen Nx586 and the Cyrix M1 [Half94], superscalar implementations of the Intel x86 architecture, use register renaming to overcome some of the problems of the small register set and the two-address architecture, besides allowing to run old code faster. Even when optimizing specifically for them, compiler-based register-renaming is heavily restricted by only six general purpose registers being available, and by the requirement that for two-address operations, one source operand must equal the destination operand. Besides changing the architecture, hardware register-renaming is the only way to decrease the dependencies between instructions and to enhance parallel execution.

In order not to stall the pipelines when a branch condition is not resolved, modern processors use some form of *branch prediction*, that is, they make a guess at whether the branch will be taken or not, and then continue execution along the corresponding path. Some use simple static schemes, like predicting backward-branches to be loop-closing branches and thus being taken, and forward branches being not-taken. Others allow the compiler to insert a *hint* on whether the branch is likely to be taken or not. Both schemes are called *static branch prediction*. By reordering the code, the compiler can still make use of static branch prediction, even if there is no special *hint* bit. According to Ball and Larus [BaLa93], between 80% and 90% of branches can be predicted correctly using static techniques. *Dynamic branch* *prediction* uses a cache to record for branches whether they were taken or not the last time, predicts them to take the same path, and achieves correct prediction rates well over 90%.

Note that the concept of delayed branches does not work well for superscalar machines. They require varying amounts of delay slots after a branch in order to fill their multiple pipelines. The number depends on both the actual machine and the code executed. Supporting variable amounts of branch delay slots would pose many implementation problems. This is why architectures with superscalar implementations in mind like the IBM POWER and PowerPC as well as the DEC Alpha architecture do not include delayed branches.

The drawback of a superscalar architecture is the additional hardware, which determines from a sequential instruction stream what to execute in parallel. The size of this hardware grows quadratically with the number of instructions to be dispatched every cycle [John90] and may have a negative impact on the cycle time of the processor. For relatively small numbers of instructions up to about ten this seems feasible.

If there are different types of units, the logic can be simplified by letting these units operate on different registers. For example, if there are integer units operating on integer registers, and floating-point units operating on floating-point registers, there can be no dependencies between integer and floating-point instructions. Therefore, it is not necessary to check for dependencies between such instructions.

Superscalar microarchitectures are currently being built for all major instruction set architectures. Like pipelining, it is an implementation method that can be used in any kind of processor to improve performance.

As for a single pipeline, dependencies can be either checked for in hardware and operations can then be delayed appropriately to satisfy constraints, or the hardware can run without interlocks, delegating the task of satisfying dependencies to the compiler. The former are called *superscalar* processors, as they still exhibit a purely sequential (scalar) execution model but achieve better performance. The latter are called *VLIW* (*V*ery *L*ong *I*nstruction *W*ord) processors, as they fetch one instruction for each execution unit every cycle without any kind of dependence checking. This block of instructions can be considered as one long instruction executed on the machine.

There are several drawbacks to a VLIW architecture. Since the long instruction word contains an operation for each unit, no-operations have to be filled in if there is nothing to perform in certain units, dramatically increasing the code size. Details about the number and types of units and the pipeline structure are reflected in the machine code, so that binary compatible processors with differing numbers of units or different pipeline structure cannot be implemented. The task of the compiler becomes more complex than for a superscalar processor, as a bad schedule does not only decrease performance but may also lead to wrong results. Finally, a VLIW processor can only combine instructions to be executed in one cycle in a static way, while a superscalar processor can combine them in a dynamic way. Consider the case of a conditional branch separating some operations. Whether the instructions after the branch are executed depends on the outcome of the branch, and a VLIW compiler will not be able to combine operations before and after the branch into one long instruction. If, on the other hand, the condition is known at run-time, a superscalar processor can execute instructions before and after the branch in the same cycle. If it is not known, the superscalar processor can still execute the instructions after the branch speculatively in the same cycle.

Since it is not possible to create binary compatible VLIW implementations with different numbers of units, the task of tailoring the code to a particular implementation may have to be delayed until the code is loaded into memory. *Portable object files* [Franz94] may offer a solution to this, although performing the very expensive steps of instruction scheduling and register allocation at load-time has a significant impact on load time. Moreover, some method has to be found to allow for the movement of instructions across conditional branches. *Predicated execution* [RYYT89], where each instruction specifies a condition controlling whether it is executed, would support this, but does have an impact on code size, as each instruction has to specify an additional condition. *Boosting* [SmLH90][SmHL92] allows to speculatively execute instructions into a shadow register file, and to commit speculative results as part of a conditional branch. This corresponds to an implementation of speculative execution for VLIW architectures.

Superscalar processors rely on optimizing compilers to achieve good performance, and compiler technology has to be improved in order to accommodate processors with many execution units. If at some point in time, the dependency checking logic for superscalar processors should become too complicated and the move to VLIW architectures unavoidable in order to further improve performance, even better compiler technology will be required to exploit these processors, with techniques that are yet to be discovered. Two early attempts to commercialize VLIW technology in the eighties have failed [Rau89][Mult93].

3.3 Caches

One of the fundamental rules holding for memory is: The larger the memory, the slower it can be accessed. Already in the early days of computing, CPU logic tended to be faster than memory. Most of the improvements in the field of logic have been dedicated to improving the speed, whereas most improvements in memories were related to increasing their size. This has widened the gap between CPU and memory speeds – a trend that is likely to continue. The fast SRAMs that exist are not suited to build large main memories due to their high cost and their low density.

According to Amdahl's law [Amda67], the speed of a computer is always bound by its slowest component, which has been memory for a long time. The speed of memory can only be improved by decreasing its size, which is not a viable option for main memories. Improving the speed of the computer and increasing the size of memory are therefore conflicting goals a hardware designer faces.

Memory is not accessed in a uniform way, however [Knuth71]. Within a certain amount of time, the same memory cells tend to be accessed over and over, e.g. when executing a loop iterating hundreds of times the same instructions are fetched repeatedly, or in procedures local variables are used over and over. This behavior is called *temporal locality*, as within a certain amount of time, only a certain localized amount of data is accessed, but this data is accessed many times. There is also the term *spatial locality*, which refers to consecutive items having a high chance of being accessed. For example, after an instruction has been fetched, there is a high probability that the next sequential instruction will be accessed as well. If some array element is accessed, there is a high chance that a neighboring element is required as well. The localized data accessed by a certain program over a certain period of time is called its *working set*. Note that the working set of a program is much smaller than all the data accessible to the program, if the program exhibits a certain amount of locality, which seems to be the case for almost all programs.

The principle of *locality of reference* can be exploited to improve speed by keeping the working set in a small and fast memory, and leaving the rest in slower memory. As long as data in the working set is accessed, the operation is fast. A slowdown is only encountered for accesses to objects outside the working set.

Such a fast memory for frequently accessed data is called a *cache*. Since programs should not have to be changed to exploit a cache, its operation must be transparent to them, that is, the way programs address memory must not be affected in any way. This is achieved by implementing the cache as associative memory storing the most recently accessed memory blocks.

The cache is organized as a number of *lines* of equal size, of which each corresponds to a block in the address space. Beside the data in that block, the cache stores the address to which this block belongs in so-called *tags*. Whenever memory is accessed by the CPU, the address will be compared against the addresses in the tags. If there is a match, the data is located in the cache. We say there is a *cache-hit*, and the access is fast. If on the other hand the data is not in the cache, we call the access a *cache-miss*. In this case, the data has to be fetched from slower main memory and must be brought into the cache, replacing some other data in the cache. Usually, the *least recently used* line is replaced, assuming that it has the lowest probability of being needed in the near future according to temporal locality.

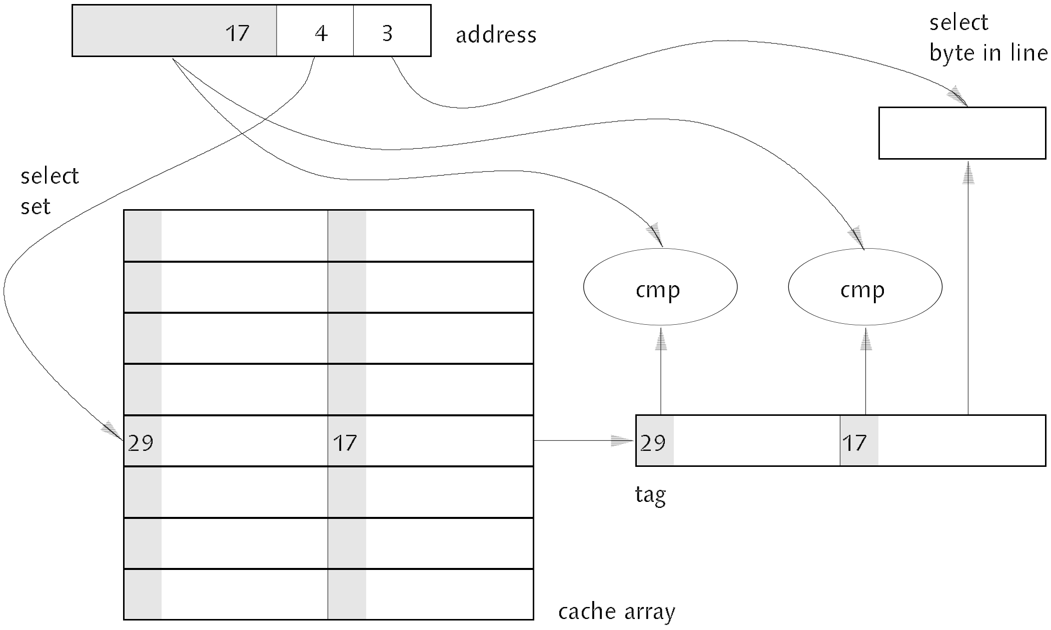


Figure 3.7: Access in a 2-way set-associative cache.

For caches with a large number of lines, searching all tags for a match as required by a fully associative cache, becomes impossible. Because of this, the cache is partitioned into a number of sets, in which the lookup is performed associatively. Some bits of the address determine which set is to be searched, that is, if these bits represent the number *N*, then the data can only reside in set *N*. The number of lines in a set is typically small, namely 1, 2, 4, or in rare cases 8. If there are *k* lines in a set, we call the cache *k-way set-associative*. Note that this places some restrictions on the pattern of the working set. If many blocks have the same set-selecting bits, they will all have to be kept in the same set, which may be too small, even though the cache as a whole would be large enough. Figure 3.7 shows how different parts of the address are used in an access to a 2-way set-associative cache.

As long as there are few cache-misses, the performance will be almost as good as if the whole memory were as fast as the cache. In case of large numbers of misses, performance will drop dramatically. There are several possible reasons for the latter to happen.

- *Capacity*: The size of the working set of the program is larger than the cache. One could also say that the temporal locality of the program is too small.
- *Line waste*: Memory is accessed in steps larger than the line size, thus a whole line is used for a single data item, while the neighboring items are brought into the cache and are not accessed. This can be considered as too little spatial locality.
- *Compulsion*: Too many items in the working set map onto the same cache set, so that each access replaces an item in the working set.

Programs can be designed to exhibit a large amount of locality, either temporal or spatial, and to avoid compulsory misses. This way, the performance on cached computers can be improved significantly.

3.4 Reduced Instruction Set Computers

Computer architecture in the late sixties was focused on closing the so-called *semantic gap*, the large difference between the level of abstraction that the machine language provided and the level at which programmers thought. By implementing more and more complex machine instructions, this gap could be closed to some extent. This direction of development culminated in the introduction of the DEC VAX, one of the most complex processor architectures ever designed.

At the same time, hardware designers found it more and more complicated to use high-performance implementation techniques like pipelining with complex instructions, as they exhibited complex pipeline usage patterns as well. Improvements in compiler technology showed that the semantic gap could also be closed by compilers instead of hardware. Moreover, some studies showed that compilers were much better in building complex operations out of simple instructions and in tailoring them to the needed context, than in using the complex machine instructions. Often, such tailored sequences of simple instructions ran in shorter time than the corresponding complex instructions.

The last point was largely due to caches becoming commonplace. When a memory access is slow, it makes perfect sense to try to execute as much as possible per memory access, and thus to create complex instructions. Such instructions encode more work than simpler ones. Thus it was possible to reduce the total number of instructions executed and to decrease the number of memory accesses, avoiding this bottleneck to better performance. Caches made memory accesses a lot faster, and caused the difficulties of pipelining complex instructions to become the bottleneck in high-performance computers.

It was John Cocke of the IBM T.J. Watson Research Lab who came to the conclusion that with a processor featuring simple instructions only, it should be much simpler to pipeline instructions. Together with a good compiler, it promised to offer better performance than other processors of their time. With his team he built the 801 computer [Radin82], the first RISC computer, and the optimizing PL.8 compiler [AuHo82] pioneering compiler optimizations. The 801 processor adhered to the following design principles.

- No instruction requires a pipeline stage multiple times.
- All computational instructions operate on registers only. Loads and stores are the only instructions accessing memory.
- Each instruction is 32 bits long, and the formats are simple to decode.
- There is a large register set providing ample room for temporary results, which will reduce the number of memory accesses.

Technologically, the 801 was a huge success, but it never became a commercial product. In the late seventies, the principles behind its design had been adopted by the group of Patterson at the University of California at Berkeley to build the Berkeley RISC-I and RISC-II processors and by Hennessy at Stanford University to build the Stanford MIPS. Patterson coined the term RISC as reduced instruction set computer. Shortly thereafter, computer companies became interested in the technology and started building commercial RISC processors.

Even though several RISC processors in use today do not provide a reduced set of instructions, they do provide a set of reduced instructions [Pren92]. That is, the instructions are reduced to operations that can be implemented in fast hardware. The above design principles still apply, with the exception that some instructions are allowed to use a pipeline stage several times. Complex instructions were added if it could be shown that they improved the performance significantly, that they did not complicate the hardware design too much, and that their impact on the achievable cycle time did not cost more performance than the instruction bought. As will be shown in the next chapter, the PowerPC architecture includes several relatively complex instructions that match commonly found program patterns very well. However, it is up to the compiler to find the code patterns in which they can be exploited.

While caching made RISC processors possible, pipelining made them a requirement, and superscalar implementation techniques do so even more. Even decoding multiple instructions every cycle becomes much more complicated if there is not a single instruction length, and dependence checking is almost impossible when many instructions access memory.

4 PowerPC Architecture

In this chapter, we will discuss the PowerPC architecture, which is the target of our optimizing compiler. It is a prominent RISC architecture and uses the techniques we discussed in the previous chapter. We will present unoptimized and optimized code patterns, and discuss their performance on different implementations.

4.1 Design Goals

After more than a decade of RISC research inside IBM, the IBM RISC System/6000 introduced in 1990 pioneered superscalar execution with its POWER architecture (*P*erformance *O*ptimization *W*ith *E*nhanced *R*ISC) [IBM90a][IBM90b][IBM90c]. From this architecture, the PowerPC architecture [IBM94] was derived in 1992 by cleaning up the design, namely by deleting some rarely used instructions and adding a few others. For a very large fraction of instructions, they are identical. [WeSm94] presents a good overview of the development of these architectures.

POWER was highly focused on achieving high performance at rather low clock speeds. The architecture defines three different kinds of execution units, namely branch unit BU, fixed-point unit FXU, and floating-point unit FPU. There is a minimal amount of shared resources between these units, so that they can operate largely without synchronization, thus simplifying superscalar execution.

Branch instructions execute in the BU, floating-point instructions in the FPU, and all remaining ones in the FXU. This is at the architectural level, for an actual implementation can still decide to support multiple such units with a single pipeline, or provide multiple pipelines for a single unit type. For example, the embedded controller IBM PPC403GA implements a single pipeline for all instructions, POWER-1 includes one BU, one FXU, and one FPU, whereas POWER-2 has one BU, two FXUs, and two FPUs. Besides simplifying superscalar execution, IBM also added a couple of more complex instructions that perform multiple operations at once, thus reducing the path-length of programs, i.e. the number of dynamically executed instructions. The decision what to add and what to leave out was based on program traces and the complexity involved in implementing these instructions.

The PowerPC took over the design goals from POWER and most of the architecture. However, it had been found that some instructions could cause trouble in superscalar processors with many execution units or at very high clock rates, and so they were dropped. Also, 64-bit extensions to the architecture were defined. The following overview only covers the 32-bit subset of the architecture.

4.2 PowerPC Architecture

The PowerPC architecture defines register to register operations for all computational instructions. They take source operands from registers or immediate-fields within the instruction and write the result into a register. The typical three-operand instruction format allows the specification of a target register different from the source registers, thus preserving the values of the source registers. When variables in memory are to be read or modified, they must be transferred between registers and memory using load or store instructions.

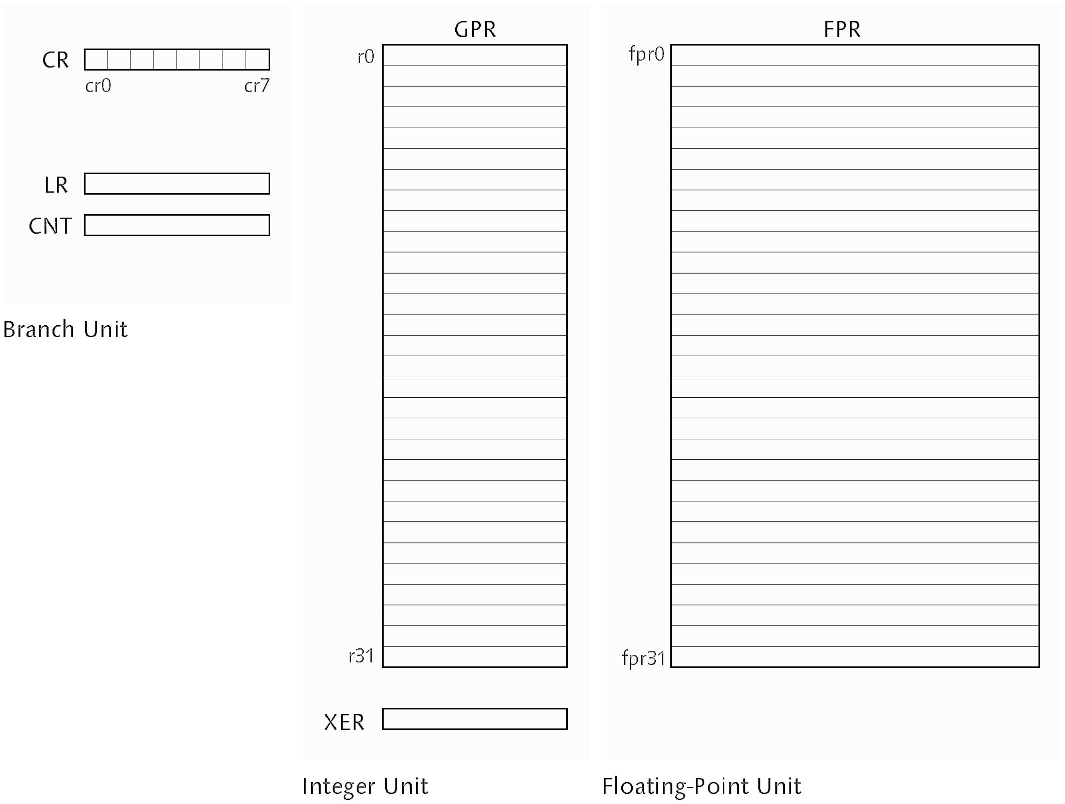


Figure 4.1: Registers in the PowerPC architecture.

The architecture defines thirty-two 32-bit *general purpose registers*, and thirty-two 64-bit *floating-point registers*. In addition, there is a *condition code register CR* holding 8 separate 4-bit condition fields, a *link-register LR*, a *count-register CNT*, and an *exception register XER*.

Each condition register field consists of 4 bits, specifying the conditions *less*, *greater*, *equal*, and *overflow* for integer or logical compares, whereas the last bit signals *unordered* for floating-point compares, as shown in Figure 4.2.

less	greater	equal	ovfl / unordered

Figure 4.2: Bits in a condition register field.

Table 4.1 explains the abbreviations used in subsequent instruction descriptions. Table 4.2 gives an overview of the arithmetic and logical operations.

rt, ra, rb	general purpose registers
frt, fra, frb, frc	floating point registers
crf	condition register field
crb	condition register bit
sb	general purpose register or sign-extended 16-bit immediate operand
imm	16-bit immediate operand
CY	carry bit
sc	general purpose register or unsigned 5-bit immediate operand
ub	general purpose register or zero-extended 16-bit immediate operand
mb	mask begin; unsigned 5-bit immediate operand
me	mask end; unsigned 5-bit immediate operand
cmask	5-bit mask

Table 4.1: Abbreviations.

Some of the instructions support the *overflow-option Of* – as indicated by an X in the Of-row in Table 4.2. This option allows to set the overflow bit in the XER-register. The *record-option Rc* supported by most instructions allows to set the condition code field *cr0* as part of the instruction.

While most of the instructions are self-explanatory, some require special mentioning. *rlwnm* rotates a word and then ands the word with a specified mask. This is useful for extracting bit-fields from a word, or for synthesizing shift-left-immediate and shift-right-immediate instructions.

rlwimi is similar to *rlwnm*, except for the rotated word not being anded with the mask but rather being inserted into the target register under control of the mask. That is, the target is computed as

rt := (rt AND ~mask) OR (ROT(ra, sc) AND mask).

The *cmask* field used in the trap-instruction *tw* allows to select any disjunctive combination of the conditions *equal*, *less-signed*, *greater-signed*, *less-unsigned*, and *greater-unsigned*. The operands *ra* and *sb* are compared, and if any of the specified conditions holds, a trap is raised.

For transferring values between memory and registers, load and store operations are provided. Table 4.3 gives an overview.

mnemonic	operation	Of	Rc
add rt, ra, sb	rt := ra + sb	X	X
addc rt, ra, sb	rt := ra + sb; set CY	X	X
adde rt, ra, rb	rt := ra + rb + CY; set CY	X	X
addis rt, ra, imm	rt := ra + LSH(imm, 16)		
subf rt, ra, rb	rt := rb − ra	X	X
subfc rt, ra, sb	rt := sb − ra; set CY	X	X
subfe rt, ra, rb	rt := rb − ra + CY; set CY	X	X
neg rt, ra	rt := −rb	X	X
mullw rt, ra, sb	rt := (ra ∗ sb) MOD WordSize	X	X
mulhw rt, ra, rb	rt := (ra ∗ sb) DIV WordSize		X
divw rt, ra, rb	rt := ra / rb (signed)	X	X
divwu rt, ra, rb	rt := ra / rb (unsigned)	X	X
slw rt, ra, rb	rt := LSH(ra, rb MOD 64)		X
srw rt, ra, rb	rt := LSH(ra, −(rb MOD 64))		X
sraw rt, ra, sc	rt := ASH(ra, −(sc MOD 64))		X
rlwnm rt, ra, sc, mb, me	rt := ROT(ra, sc) AND {mb..me}		X
rlwimi rt, ra, sc, mb, me	rt := INS(rt, ROT(ra, sc), {mb..me})		X
extsb rt, ra	rt := sign-extend-byte(ra)		X
extsh rt, ra	rt := sign-extend-halfword(ra)		X
cntlzw rt, ra	rt := count-leading-zeroes(ra)		X
cmp crf, ra, sb	crf := compare-signed(ra, sb)		
cmpl crf, ra, ub	crf := compare-unsigned(ra, ub)		
tw cmask, ra, sb	ASSERT(~cmask(ra, sb))		
and rt, ra, ub	rt := ra ∗ ub		X
andc rt, ra, rb	rt := ra − rb		X
andis rt, ra, imm	rt := ra ∗ LSH(imm, 16)		X
nand rt, ra, rb	rt := − (ra ∗ rb)		X
or rt, ra, ub	rt := ra + ub		X
orc rt, ra, rb	rt := ra + (−rb)		X
oris rt, ra, imm	rt := ra + LSH(imm, 16)		
nor rt, ra, rb	rt := − (ra + rb)		X

Table 4.2: Arithmetic and logical instructions.

mnemonic	operation	Ud
l*d* rt, ra, sb	rt := Mem[ra+sb]	X
st*d* rs, ra, sb	Mem[ra+sb] := rs	X
lmw rt, ra, imm	rt..r31 := Mem[ra+imm]	
stmw rs, ra, imm	Mem[ra+imm] := rs..r31	

Table 4.3: Load and store instructions.

d	data type
w	32-bit word
ha	16-bit word sign-extended
hz	16-bit word zero-extended
bz	byte zero-extended
fs	single-precision floating-point
fd	double-precision floating-point

Table 4.4: Data types for loads and stores.

As indicated by *d*, the instructions support different data types, where *d* can be any of the types in Table 4.4. The *update-option Ud* allows to write back the effective address computed into the base register *ra*. That is, the instruction

```
lwu    r3, r4, 20
```

computes

```
r4 := r4 + 20; r3 := Mem[r4].
```

The PowerPC architecture allows conditional branching based on any bit in the *CR* register being set or cleared and based on the state of the *CNT* register. Tables 4.5 and 4.6 present an overview. All branch instructions support the *link-option*, which causes the address of the instruction following the branch being written to the *LR* register. This can be used for subroutine linkage.

mnemonic	operation
b disp26	PC := PC+disp26
bc code, crb, disp16	IF cond THEN PC := PC+disp16 END
bcr code, crf, (LR | CNT)	IF cond THEN PC := (LR | CNT) END

Table 4.5: Branch instructions.

code	condition
bt	cond := crb
bf	cond := ~crb
dz	DEC(CNT); cond := CNT = 0
dnz	DEC(CNT); cond := CNT # 0
btdz	DEC(CNT); cond := (CNT = 0) & crb
bfdz	DEC(CNT); cond := (CNT = 0) & ~crb
btdnz	DEC(CNT); cond := (CNT # 0) & crb
bfdnz	DEC(CNT); cond := (CNT # 0) & ~crb

Table 4.6 : Conditional branch codes.

The PowerPC architecture supports static branch prediction. Backward-branches are predicted to be taken, and forward-branches to be not-taken. This prediction can be negated by a hint bit in conditional branch instructions.

All floating-point instructions are available both for single-precision and double-precision operands. Table 4.7 lists the floating-point instructions.

mnemonic	operation
fabs frt, fra	frt := ABS(fra)
fnabs frt, fra	frt := −ABS(fra)
fadd frt, fra, frb	frt := fra + frb
fsub frt, fra, frb	frt := fra − frb
fmul frt, fra, frb	frt := fra * frb
fdiv frt, fra, frb	frt := fra / frb
fmr frt, fra	frt := fra
fneg frt, fra	frt := −fra
fcmpo crf, fra, frb	crf := compare(fra, frb)
fmadd frt, fra, frb, frc	frt := fra*frb + frc
fmsub frt, fra, frb, frc	frt := fra*frb − frc
fnmadd frt, fra, frb, frc	frt := −fra*frb + frc
fnmsub frt, fra, frb, frc	frt := −fra*frb − frc
fctiw frt, fra	frt := INTEGER(fra)

Table 4.7: Floating-point instructions.

4.3 PowerPC Microarchitecture

In order to optimize code for a certain processor, knowing about the instruction set is not enough. The compiler also has to consider details of the pipeline structure, cache architecture and other features affecting performance. Since the introduction of the POWER architecture in 1990, IBM has delivered three largely different implementations of POWER and four implementations of PowerPC. Tables 4.8 and 4.9 present an overview. The issue width is the number of instructions that can be issued in one cycle.

	POWER-1	RSC	POWER-2
issue width	4	2	6
integer units	1	1	2
FP units	1	1	2
branch units	1	–	1
cache size kB (I/D)	8/32*	8	64/256*
cache associativity	2/8	2	2/4
branch prediction	none	none	none

Table 4.8: Overview of POWER implementations. Cache sizes marked with an asterisk indicate initial cache sizes, which were changed in later implementations.

The processors vary largely in the number of execution units provided, their cache architecture, and also largely in the performance they offer. In the following, we will discuss two designs in somewhat more detail, namely the PowerPC 601 and the POWER-2 implementation. Both are relatively recent implementations, are available in actual machines, and represent different price/performance-points.

	PPC 601	PPC 603	PPC 604	PPC 620
issue width	3	2+br	4	4
integer units	1	1	3	3
load/store units	–	1	1	1
FP units	1	1	1	1
branch units	1	1	1	1
cache size kB (I/D)	32	8/8	16/16	32/32
cache associativity	8	2/2	4/4	n/a
branch prediction	static	static	dynamic	dynamic

Table 4.9: Overview of PowerPC implementations.

PowerPC 601

The PowerPC 601 [PPC601] is the first implementation of the PowerPC architecture, and was introduced in 1993. There is an integer-unit which also performs loads and stores, a floating-point unit, and a branch unit. The processor can fetch up to eight instructions per clock cycle, and issue one instruction to each unit each cycle. The main pipeline hazards found in the PowerPC 601 are as follows.

- *Load-use*: A load has a latency of two, so its result cannot be used until two cycles after the load.
- *Floating-point operations*: Single-precision FP operations have a latency of two, and double precision operations a latency of three cycles. Double-precision operations occupy the multiply-stage for two cycles, and therefore a new double-precision instruction can only be issued every other cycle.
- *Compare-branch*: An integer compare has a latency of two cycles until its result is available in the branch unit. If the condition code is not available at the time the branch is issued, the branch unit will predict the outcome of the branch and continue execution along the predicted path. While a predicted branch is awaiting resolution, no other branch can be predicted.
- *Branches*: Even though the branch unit can execute a branch every cycle, two taken branches cannot be executed in subsequent cycles.

There are several other hazards related to the single cache port or to special purpose registers. Throughout this thesis, we will only consider those mentioned above.

The 601 implements the static branch prediction method of the PowerPC architecture.

POWER-2

POWER-2 [IBM94b] is the second major implementation of the POWER architecture, and was introduced in 1993 in high-end computers in the RS/6000 line. Compared to its predecessor POWER-1 (and the somewhat similar PowerPC 601), it basically doubled all features. It includes two integer-units, two floating-point units, and a branch-unit, which in some cases can perform two branches in one cycle. The instruction fetcher can fetch eight instructions and issue 2 integer, 2 FPU, and 2 branch instructions each cycle.

Two instructions can only be executed in the dual integer- or FP-units if they are independent of each other. There is one important exception supported by special hardware, however [MaEV92]. If there are two dependent additions, they can be executed in parallel by rewriting them as in Figure 4.10.

```
add   r5 := r3, r4                    add    r5 := r3, r4
add   r7 := r6, r5                    add3   r7 := r6, r3, r4
```

Figure 4.10: Dependent additions being rewritten.

add3 is an operation implemented by the second integer unit that adds three operands together. Note that the *add3* is not a machine instruction available to the programmer. The decode logic recognizes the case of two dependent additions and rewrites the second accordingly. This feature was added in order to get good performance out of code not optimized for dual integer pipelines.

This feature had an interesting impact on compilers. While compilers for the POWER architecture used *ori*-instructions to perform register-register-moves, this was changed with the introduction of POWER-2. From then on, they used *addi*-instructions, which could be executed in parallel with dependent additions.

Besides requiring subsequent instructions to be independent or of the above form in order to make use of the dual execution units, POWER-2 has pipeline hazards similar to those found in the PowerPC 601.

- *Load-use*: A load has a latency of two cycles.
- *Floating-point operations*: POWER-2 does not implement single-precision FP operations. Double-precision operations have a latency of two cycles.
- *Compare-branch*: An integer compare has a latency of two cycles until its result is available in the branch unit. If the condition code is not available at the time the branch is issued, the processor will speculatively execute along the sequential path.
- *Branches*: Even though the branch unit can execute up to two branches every cycle, two *taken* branches cannot be executed in parallel or in subsequent cycles.

POWER-2 does not implement branch prediction, but speculatively executes along the sequential (not-taken) path. The compiler has to rearrange the code so that the sequential path is the predicted path, as described by Bernstein and Rodeh [BeRo91].

4.4 Some Examples of Code Improvements

As discussed in Chapter 2, the compiler can reduce the execution time of a program by reducing the number of instructions executed, by selecting cheaper instructions over more expensive ones, and by ordering them so that pipeline stalls are rare. In the following section, we will look at some code samples and the PowerPC code produced by non-optimizing compilers. We will discuss possible improvements to this code and their performance impact on the PowerPC 601 and on POWER-2.

Initializing an Array

Array initialization code has the following typical form.

```
VAR
   a: ARRAY N OF LONGINT;

PROCEDURE InitArray;
   VAR i: LONGINT;
BEGIN
   i := 0;
   WHILE i < N DO a[i] := 0; INC(i) END
END InitArray;
```

Omitting procedure entry and exit code, this translates to the following PowerPC code with the non-optimizing Oberon-2 compiler.

```
            (1, 1)        li          Ri := 0                   ; i := 0
loop:
            (1, 1)        cmpi        CF1 := Ri, N              ; i < N
            (0/2, 0/2)    bc          ~CF1.LT,  end
            (1, 0)        twi         >=u, Ri, N                ; index check
            (1, 1)        rlwinm      R3 := Ri, 2, 0, 29        ; i*4
            (1, 1)        add         R4 := SB, R3              ; SB+i*4
            (1, 0)        li          R5 := 0
            (1, 1)        stw         offset-a(R4) := R5        ; a[i] := 0
            (1, 0)        addi        Ri, Ri, 1                 ; INC(i)
            (0, 0)        b           loop
end:
```

This loop executes 9 instructions for each array element initialized. Numbers in parentheses on the left indicate the predicted execution time for the PowerPC 601 and POWER-2, respectively. An execution time of zero cycles corresponds to the instruction being executed in parallel with the previous one. Conditional branches carry two numbers, the first one indicating the non-taken case, while the second one is for the branch-taken case.

Since the loop will execute N times, the execution time on the PowerPC 601 is 1+7*N+3 = 7*N+4 cycles, while for POWER-2, it is 1+4*N+3 = 4*N+4 cycles.

The code could be improved as follows. First of all, the repeated loading of the value 0 inside the loop could be moved out of the loop. Since the index i can be shown to be always within the bounds of the array, the index check can be dropped. Instead of running an index over the array, one could set a pointer to the array, and increment that pointer on each iteration, making use of the PowerPC store-and-update instruction. Finally, since the number of iterations of the loop is known, the PowerPC branch-and-count instruction can be used. This yields the following code.

```
            (1, 1)        li          R3 := N
            (1, 1)        mtspr       CTR := R3
            (1, 1)        li          R5 := 0
            (1, 0)        addi        R4 := SB, offset-a–4
loop:
            (1, 1)        stwu        4(R4) := R5
            (1, 1)        bdn         loop
```

This improved loop will execute in 2*N+4 cycles on the PowerPC 601 and 2*N+3 cycles on POWER-2. The loop body does not yet make good use of the parallel execution units, in particular, the branch-unit cannot keep up branching every cycle, and on POWER-2, no use is made of the dual fixed-point units executing stores. By executing four stores on each iteration, and reducing the number of iterations by a factor of four, the loop can be improved further for both implementations. The case where N is not a multiple of four has to be handled outside the loop, and has been omitted.

```
            (1, 1)      li        R3 := N / 4
            (1, 1)      mtspr     CTR := R3
            (1, 1)      li        R5 := 0
            (1, 0)      addi      R4 := SB, offset-a–4
    loop:
            (1, 1)      stw       4(R4) := R5
            (1, 0)      stw       8(R4) := R5
            (1, 1)      stw       12(R4) := R5
            (1, 0)      stwu      16(R4) := R5
            (0, 0)      bdn       loop
```

The theoretical execution time will be reduced to N+4 cycles for the PowerPC 601 and N/2+3 cycles for POWER-2, for an overall improvement by a factor of 7 and 8, respectively.

The PowerPC 601 uses a unified cache, i.e. a single cache for both instructions and data. Conflicts between instruction fetches and stores on that cache keep the PowerPC 601 from achieving the above number, however. The slightly different code pattern achieving 7/6*N+4 cycles is not discussed as the occurring interlocks are complicated.

Linear Search on an Array

The following loop is taken from the Quick-Sort algorithm as published in [Wirth86a].

```
    VAR
       a: ARRAY N OF LONGINT;

    PROCEDURE sort (L, R: LONGINT);
       VAR i, x: LONGINT;
    BEGIN
       ...
       WHILE a[i] < x DO i := i+1 END
       ...
    END sort;
```

The non-optimizing Oberon-2 compiler generates the following PowerPC code for this loop.

```
    loop:
            (1, 1)      twi       >=u, Ri, N             ; index check
            (1, 0)      rlwinm    R3 := Ri, 2, 0, 29     ; i*4
            (1, 1)      add       R4 := SB, R3           ; SB+i*4
            (1, 0)      lwz       R5 := offset-a(R4)     ; a[i]
            (2, 2)      cmp       CF1 := R5, Rx          ; a[i] < x
            (0/2, 0/2)  bc        ~CF1.LT, end
            (1, 0)      addi      Ri := Ri, 1            ; i := i+1
            (1, 2)      b         loop
    end:
```

The execution time is 8*(M–1)+8 = 8*M cycles for the PowerPC 601, if a matching element is found after M iterations. For POWER-2, the corresponding time is 6*(M–1)+6 = 6*M cycles. Note that there are pipeline interlocks between the load and the compare. Furthermore, the conditional branch requires speculation and the loop-closing branch at the end incurs a pipeline interlock until the conditional branch is resolved.

This loop can be improved in a similar way as the array initialization above. Instead of using an index to access the array, a pointer can be used. The value of the index can be derived from the pointer. The index check cannot be completely removed, but it can be propagated out of the loop by rewriting the loop into the following form.

```
ASSERT(i >= 0, index-check);
WHILE (i < N) & (a[i] < x) DO i := i+1 END ;
ASSERT(i < N, index-check)
```

Since the maximum number of iterations due to the limit of N can be precomputed, branch-and-count instructions with zero overhead can be used. The optimized loop looks as follows.

```
          (1, 1)         subfi      R0 := Ri, N              ; N–i
          (1, 0)         twi        <s, Ri, 0                ; ASSERT(i >= 0)
          (1, 1)         rlwinm     R3 := Ri, 2, 0, 29       ; i*4
          (1, 0)         add        R3 := SB, R3             ; SB+i*4
          (1, 1)         addi       R3 := R3, offset-a–4
          (1, 0)         mtspr      CTR := R0
loop:
          (1, 1)         lwzu       R4 := 4(R3)
          (2, 2)         cmp        CF0 := R4, Rx
          (0/2, 2/0)     bcdn       ~CF0.LT, loop
end:
          (1, 0)         subf       R3 := SB, R3
          (1, 1)         addi       R3 := R3, –offset-a
          (1, 1)         rlwinm     Ri := R3, 29, 2, 31      ; ptr –> i
          (1, 1)         twi        >=s, Ri, N               ; ASSERT(i < N)
```

This improved loop executes in 3*(M–1)+15 = 3*M+12 cycles on the PowerPC 601, and 5*(M–1)+9 = 5*M+4 cycles on POWER-2. On both implementations, pipeline interlocks prevent better performance, in particular POWER-2 loses a lot on the mispredicted loop-closing branch. By putting the branch at the top of the loop, and using an unconditional branch to close the loop, the execution time on POWER-2 could be reduced to 3*(M–1)+11 = 3*M+8 cycles. Each iteration would then take 4 cycles on the PowerPC 601.

```
          (1, 1)         crxor      CF0.LT, CF0.LT, CF0.LT   ; clear CF0.LT
loop:
          (1/2, 0/2)     bcdz       CF0.LT, end
          (1, 1)         lwzu       R4 := 4(R3)
          (2, 2)         cmp        CF0 := R4, Rx
          (0, 0)         b          loop
end:
```

Instead of pursuing this possibility further, we will discuss how overlapping of multiple iterations in so-called *software pipelining* can improve the execution time even more. The idea is to start a new iteration before the previous one has finished. Further elements of the array are loaded and compared against *x* before the conditional branch is executed. Note that array elements may be accessed that the original program did not access. We call such operations *speculative operations*, as their execution is based on the speculation that the result will be needed. It must be made sure that such speculative operations do not affect the final state of the program, since this would change the semantics. This prevents stores and operations that can raise exceptions from being executed speculatively.

```
                (1, 1)          lwzu        R4 := 4(R3)
                (1, 0)          lwzu        R5 := 4(R3)
                (1, 1)          lwzu        R6 := 4(R3)
                (1, 1)          cmp         CF0 := R4, Rx
                (1, 0)          lwzu        R4 := 4(R3)
                (1, 1)          cmp         CF1 := R5, Rx
    loop:
                (1, 1)          lwzu        R5 := 4(R3)
                (1, 0)          cmp         CF2 := R6, Rx
                (0/0, 0/0)      bcdz        CF0.LT, end
                (1, 1)          lwzu        R6 := 4(R3)
                (1, 0)          cmp         CF0 := R4, Rx
                (0/0, 0/0)      bcdz        CF1.LT, end
                (1, 1)          lwzu        R4 := 4(R3)
                (1, 0)          cmp         CF1 := R5, Rx
                (0/0, 0/0)      bcdn        ~CF2.LT, loop
    end:
                (1, 1)          addi        R3 := R3, –16
```

At the end, the pointer to the array has to be reduced by 16, as 4 additional array elements have been already loaded. The improved loop takes M*2+7 cycles on the PowerPC 601, and M+5 cycles on POWER-2, approaching a speedup of 4 and 6, respectively, when M is large.

Traversing a Linked List

A linked list may be traversed as follows searching for an element with a particular key.

```
    p := root;
    WHILE (p # NIL) & (p.key # key) DO p := p.next END
```

Using the non-optimizing Oberon-2 compiler, this translates into the following PowerPC code.

```
                (1, 1)          lwz         Rp, offset-root(SB)         ; p := root
    loop:
                (1, 1)          cmpi        CF1 := Rp, NIL              ; p # NIL
                (0/2, 0/2)      bc          CF1.EQ, end
                (1, 0)          lwz         R3 :=  4(Rp)                ; p.key
                (2, 2)          cmp         CF6 := R3, Rkey             ; p.key # key
                (0/2, 0/2)      bc          CF6.EQ, end
                (1, 0)          lwz         Rp := 0(Rp)                 ; p := p.next
                (1, 2)          b           loop
    end:
```

The execution time of this loop is 6*(M–1)+1+6 = 6*M+1 cycles on the PowerPC 601 if a match is found in the M'th element, and 5*(M–1)+1+5 = 5*M+1 cycles on POWER-2. No special instructions can be used to decrease the amount of code in the loop, but the pipeline interlocks can be removed by software pipelining. However, loading the pointer to the next element is the last operation in the loop body, and a new iteration cannot be started before that pointer is available. Moving that load instruction higher up in the body requires to move it above conditional branches. Since a load through a NIL-pointer does not raise an exception on current PowerPC operating systems, this can be done safely. The improved code after software pipelining looks as follows.

```
             (1, 1)         cmpi       CR0 := Rp1, NIL
             (1, 0)         lwz        Rp2 := 0(Rp1)
             (1, 1)         lwz        R8 := 4(Rp1)
             (1, 0)         cmpi       CR1 := Rp2, NIL
             (0/0, 0/1)     bc         CR0.EQ, end
             (1, 1)         lwz        Rp3 := 0(Rp2)
             (1, 0)         lwz        R9 := 4(Rp2)
             (1, 1)         cmp        CR6 := R8, Rkey
    loop:
             (1, 1)         cmpi       CR0 := Rp3, NIL
             (0/0, 0/0)     bc         CR1.EQ, exit-p2
             (1, 0)         lwz        Rp4 := 0(Rp3)
             (1, 1)         lwz        R8 := 4(Rp3)
             (1, 0)         cmp        CR7 := R9, Rkey
             (0/0, 0/0)     bc         CR6.EQ, end
             (1, 1)         cmpi       CR1 := Rp4, NIL
             (0/0, 0/0)     bc         CR0.EQ, exit-p3
             (1, 0)         lwz        Rp1 := 0(Rp4)
             (1, 1)         lwz        R9 := 4(Rp4)
             (1, 0)         cmp        CR6 := R8, Rkey
             (0/0, 0/0)     bc         CR7.EQ, exit-p2
             (1, 1)         cmpi       CR0 := Rp1, NIL
             (0/0, 0/0)     bc         CR1.EQ, exit-p4
             (1, 0)         lwz        Rp2 := 0(Rp1)
             (1, 1)         lwz        R8 := 4(Rp1)
             (1, 0)         cmp        CR7 := R9, Rkey
             (0/0, 0/0)     bc         CR6.EQ, exit-p3
             (1, 1)         cmpi       CR1 := Rp2, NIL
             (0/0, 0/0)     bc         CR0.EQ, end
             (1, 0)         lwz        Rp3 := 0(Rp2)
             (1, 1)         lwz        R9 := 4(Rp2)
             (1, 0)         cmp        CR6 := R8, Rkey
             (0/0, 0/0)     bc         ~CR7.EQ, loop
    exit-p4:
             (1, 1)         lr         Rp1 := Rp4
             (0, 0)         b          end
    exit-p3:
             (1, 1)         lr         Rp1 := Rp3
             (0, 0)         b          end
    exit-p2:
             (1, 1)         lr         Rp1 := Rp2
    end:
```

This code takes roughly 4*M cycles on the PowerPC 601 and 2*M cycles on POWER-2. Despite the loop being hard to optimize, this still presents a remarkable speedup of 1.5 and 2.5, respectively.

What Improvements Can Be Expected?

The above examples indicate that large improvements can be achieved with compiler optimizations, but also that they vary widely with the kind of code compiled. Large applications contain pieces of code that are improved a lot, and others that are not sped up at all. Comparing optimizing compilers with the code generation techniques used in the non-optimizing Oberon-2 compiler, overall performance gains for whole applications have been shown to be between factors of two and three on the PowerPC 601 and above three for POWER-2. For floating-point intensive code, even larger improvements around a factor of seven are found on POWER-2 [IBM94b]. Our simple optimizing Oberon-2 compiler OOC2 improves the execution speed of programs by about a factor of two.

5 Intermediate Program Representations

The single most critical design decision when building an optimizing compiler is the choice of the intermediate program representation, on which the individual optimization phases operate. Its structure remains the same throughout the entire compilation process, and thus affects all parts of the compiler.

There are three important mutually independent attributes of intermediate representations that we can distinguish: The level of abstraction at which the operations are described, the way results of computations are referenced, and the way control-flow is represented.

5.1 Abstraction Levels for Operations

The level of abstraction for operations can range from using operators of the high-level language directly, through using the target machine instructions, to using a sub-machine-language.

Source Language Operators

A high level of abstraction, i.e. using the operators of the source language, has the advantage of being easy to generate and easy to understand. It offers the highest amount of portability, as it can be translated into executable code for any machine on which the source language can be implemented. Optimizations also take place at this level and therefore become machine-independent as well. However, since many of the high-level operators will eventually be translated into several machine operations, this does not allow the compiler to optimize the actual machine instructions. For example, an access to an array element will consist of checking the index against the array bounds, multiplying the index by the size of the element, and then accessing the memory location. In the following example, two accesses to arrays of the same type with the same index are shown. A high-level representation of the array access using an index operator cannot express the common parts between them.

```
VAR
   a, b: ARRAY N OF INTEGER;
BEGIN
   ... a[i] + b[i]
```

The accesses will be represented as

```
index   a, i
index   b, i
```

and quite correctly are not congruent. However, at the machine level, *i* will be checked against the bounds 0 and *N* and multiplied by the size of an INTEGER twice. This fact could be discovered by optimization algorithms if the individual steps of checking and multiplying the index were expressed directly.

Sub-Machine Level Operators

The above fact is the reasoning behind using a sub-machine language for the operators, which is the other extreme in terms of abstraction-levels. Each operation is represented by atomic operations at the machine level of a very primitive hypothetical machine, allowing to optimize every single step that has to be performed for a given result. Since real machines tend to have instructions combining multiple such atomic operations, achieving good code quality requires patterns of operations to be found that may be implemented by a single target machine instruction. One example for such a low-level representation is the *register-transfer-level (RTL)* representation used in GNU CC [Stall94]. Note that such a representation also yields a rather portable compiler, as long as all primitive operations can be implemented on every target machine.

Machine Operators

A compromise would be to use machine instructions of the target machine in the intermediate representation. This still allows the compiler to optimize most constructs, but does not require a pattern matcher which finds operations to be combined in one machine instruction. Moreover this leads to a more compact intermediate representation. In fact, if all machine instructions have the same execution cost, optimizing the internals of an instruction as allowed by a lower-level representation does not yield a benefit. The size of the intermediate representation has a big impact on compilation speed. Using machine-level operations, the size of the intermediate representation is in between those of the other options, and so will be the compilation speed. Moreover, since actual machine instructions will eventually have to be generated for the operations, this is the only way to avoid changing the operation level within the compiler at some point. The disadvantage is that the intermediate representation becomes non-portable. With architectures becoming more and more similar, we believe this to be a minor problem, however. Affected the most by instruction-set differences are peephole optimizations, which are inherently non-portable.

Sometimes it is useful to keep high-level operations in the intermediate representation and lower them into real machine instructions later. Lowering them to the machine level means loss of information, which might have been useful in later optimization stages. For example, the code sequence implementing a record-assignment in Oberon may be much harder to optimize than an assignment operation itself. It may be found that the original record could be used instead of the assigned copy, avoiding the assignment altogether. Such a fact would be hard to determine from the machine code implementing the record assignment.

5.2 Abstraction Levels for Control-Flow

Control flow can also be represented at different levels. As for instructions, a high level corresponds to using the control structures of the source language directly in the intermediate representation, while a low level corresponds to exposing the branching structure of the program.

High-Level Control Structures

In a high-level programming language, a procedure contains a sequence of statements, some of which may be structured. Such structured statements contain one or more statement sequences, and conditions which control the execution of them. Applying this concept to an intermediate program representation, structured statements become special instructions which contain sequences of instructions implementing the condition evaluation and other sequences for the statement bodies. Figure 5.1 depicts an example.

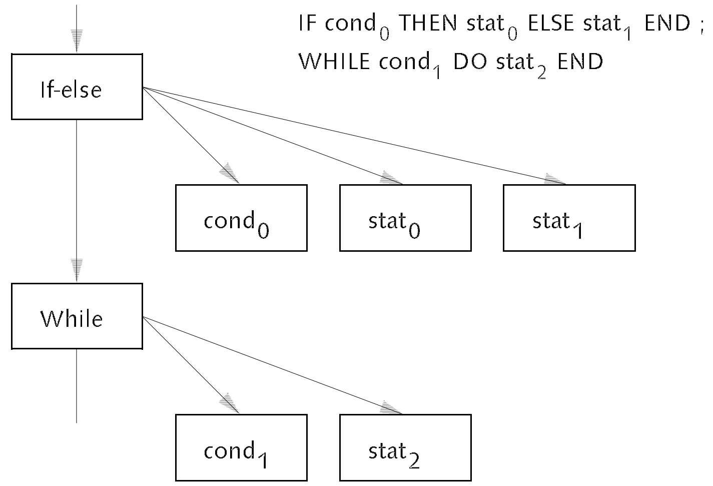

IF $cond_0$ THEN $stat_0$ ELSE $stat_1$ END ;
WHILE $cond_1$ DO $stat_2$ END
If-else
$cond_0$
$stat_0$
$stat_1$
While
$cond_1$
$stat_2$

Figure 5.1: High-level control structures as part of an intermediate program representation.

Using the control structures directly has several advantages. Since it directly reflects the high-level structure of the program, it is easy to generate and to understand. No useful information from the source language is lost. Many optimization algorithms require information about the presence and position of loops and about the nesting of control structures, which is readily available in this representation. No separate data structure is required to model control-flow besides nested instruction lists. Reconsidering the example of common subexpression elimination, a result is available in the sequence of instructions in which it is computed, including nested control structures. This is a very simple rule to determine availability.

The rule is slightly too pessimistic, however. Conditions are evaluated even when the body of the control structure is not executed, and the results of the instructions used in this evaluation would be available later on, both inside and outside the control structure. Even though in determining availability one could deal with these cases, the artificial separation of condition evaluation from other computations complicates things.

Another disadvantage is that branching is not explicitly represented and therefore cannot be optimized. Since everything has to be expressed in terms of high-level control structures, some legitimate transformations are not possible, as shown in the following example.

```
WHILE (p # NIL) & (p↑.key # x) DO p := p↑.next END ;
IF p # NIL THEN stat0 ELSE stat1 END
```

Assume it is determined that the comparison of *p* against NIL is congruent in both the While- and the If-statement. Then, the branch out of the loop in case *p* equals NIL could be redirected to $stat_1$, the other branch to $stat_0$, and the If-statement would become superfluous. Without using explicit branches in the intermediate representation, this cannot be expressed.

Using Loop- and Exit-statements, the above code could be optimized even in Oberon itself, as follows.

```
LOOP
   IF p = NIL THEN stat1; EXIT
   ELSIF p↑.key = x THEN stat0; EXIT
   ELSE p := p↑.next
   END
END
```

However, Exit-statements are explicit branches, and cause several problems when generating the intermediate program representation and analyzing the program. For example, assume that there would be Exit-instructions in the intermediate representation just like If- or While-instructions, closely reflecting the source program. Such an instruction would correspond to a branch behind the loop. At that position, however, it would not be possible to determine the available results without first finding all corresponding Exit-instructions, and analyzing the paths leading to them. Storing all Exit-instructions with the Loop-instruction they belong to would simplify this, but would also introduce complicated dependencies between different control structures, break the hierarchical nesting, and introduce another data structure which has to be maintained throughout optimizations.

Other problems of unstructured control-flow and possible approaches to dealing with them will be discussed in a later section.

High-level control structure representations can be generated directly while parsing the source program, with algorithms that have a time complexity linear in the size of the program.

Guarded Representations

Instead of representing branching implicitly only through high-level control structures, it could be made more explicit using guarded statements. This form is inspired by the *guarded command language* of E.W. Dijkstra [Dijk76], and provides very clean semantics, on which analysis can be based. We are the first to study it as the only representation of control flow in a compiler.

When translating an If-statement into guarded command language, a guard controlling the execution must be provided for each path. The guard represents a condition that causes the following statement sequence to be executed if the condition is true respectively to be skipped if it is false. If multiple guards evaluate to true, one of them is selected in a non-deterministic way, i.e. there is no defined order of evaluation for guards. This differs from the semantics of a simple If-statement in languages such as Oberon, where conditions are evaluated in source order and only if the previous conditions were false. However, by replicating previous conditions in negated form, this requirement can be expressed.

```
IF cond THEN $stat_0$ ELSE $stat_1$ END

|[   if   cond → $stat_0$   [ ]   ¬cond → $stat_1$   fi   ]|
```

In an intermediate program representation, the combination of a guard and the corresponding statement sequence can be treated like a guarded statement, without the evaluation of the condition, e.g. without the comparison of two expressions. The condition is computed separately using ordinary instructions and is referenced by the guards. In the following, guards are marked by a colon at the end and printed in italics for better readability.

```
(1) cond
(2) if-true:  (1)
        (3) $stat_0$
(4) if-false:  (1)
        (5) $stat_1$
```

Nesting of instructions corresponds to the instructions being controlled by the enclosing guard. In the above example, instruction 3 is part of the guarded statement represented as instruction 2. It will only be executed, if the parameter of the guard is true.

Oberon requires that conditions including Or- and And-operators are evaluated in *short-circuit* form. The elementary conditions are evaluated from left to right, and only as long as the complete result is not known. For example, if there are two conditions combined by an And-operator, and the first one evaluates to false, it is known that the combined result will be false and the second condition is not evaluated. In guarded command language, this can be expressed by nesting guards.

```
IF $cond_0$ & $cond_1$ THEN $stat_0$ ELSE $stat_1$ END

|[
    if    $cond_0 \rightarrow$
        |[    if    $cond_1 \rightarrow stat_0$    [ ]    $\neg cond_1 \rightarrow stat_1$    fi    ]|
    [ ]    $\neg cond_0 \rightarrow stat_1$
    fi
]|
```

Note that the statement sequence in the Else-path *$stat_1$* is replicated. In order to avoid this, an instruction to combine two guards by Or-ing them is introduced into the intermediate program representation. We call it a *merge*, as it merges two paths of control, and more precisely a *c-merge* as it combines conditions. The above example would then translate as follows.

```
(1)  $cond_0$
(2)  if-true:   (1)
          (3)  $cond_1$
          (4)  if-true:   (3)
                    (5)  $stat_0$
          (6)  if-false:   (3)
(7)  if-false:   (1)
(8)  c-merge:   (6), (7)
          (9)  $stat_1$
```

Now that there is a construct to combine different paths of control, this construct could be used in other situations as well, namely to merge the paths of an If-statement at the end with an *i-merge*, and to combine the path into a loop and the loop-closing branch using a *l-merge*. This way, no additional construct is needed to represent loops as shown in the following examples.

```
IF cond THEN $stat_0$ ELSE $stat_1$ END

(0)  true:
          (1)  cond
          (2)  if-true:   (1)
                    (3)  $stat_0$
          (4)  if-false:   (1)
                    (5)  $stat_1$
          (6)  i-merge:   (2), (4)
```

```
WHILE cond DO stat END

(0)  true:
          (1)  l-merge:   (0), (3)
                    (2)  cond
                    (3)  if-true:   (2)
                              (4)  stat
                    (5)  if-false:   (2)
```

Representations using guarded form still reflect the high-level control structure of the source program, but allow the compiler to optimize the branching structure. The set of predicates controlling the execution of an instruction can easily be determined as the conjunction of all guards in which the instruction is nested. Control-flow is also integrated with instruction lists and no separate data structure is needed to represent it. There is a single representation for loops. For common subexpression elimination, a result is available within the sequence of instructions in which it is computed, including nested ones. All advantages of using high-level control structures still apply, but additionally the branching structure of the program can be analyzed and optimized.

```
WHILE (p # NIL) & (p↑.key # x) DO p := p↑.next END ;
IF p # NIL THEN stat0 ELSE stat1 END
```

In this example, which we have already encountered in the section on high-level control-structures, the compiler would find both guards for *p # NIL* to be equivalent. It could move $stat_0$ into the loop-exit guard corresponding to *p = NIL*, and $stat_1$ into the exit guard corresponding to *p↑.key = x*. The If-statement would then become completely superfluous.

The fact that each instruction is guarded by a set of conditions offers very clean semantics and allows many powerful optimization algorithms, but also has one disadvantage. In the presence of unstructured control-flow, the predicates controlling execution of instructions are not directly available, and no clean nesting of guards can be given.

For programs with structured control-flow only, guarded forms can be generated while parsing the source program using algorithms of time complexity O(*N*), where *N* is the number of instructions in the program.

Control-Flow Graphs and Related Representations

A low-level representation of control flow directly exposes the branching structure of a program, like in an assembly language program, where conditional and unconditional branches are used to control the execution of individual instructions. Instead of using symbolic names or labels for branch targets, pointers are used to connect branches and targets, thus creating a directed graph called *control-flow graph (CFG)* [ASU86]. For a definition, the concept of a basic block is required.

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

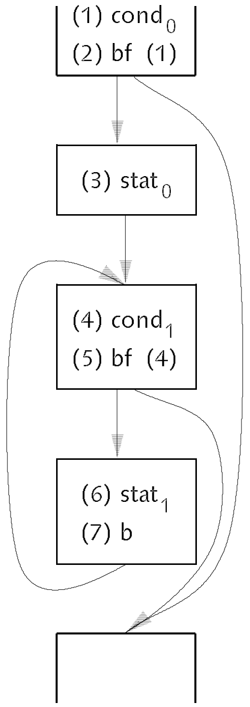
A basic block may only contain a branch instruction as the last instruction, and a label may only be at the very beginning. A basic block is either executed as a whole or not at all.

Definition: A *control-flow graph* is a directed graph whose nodes are basic blocks. There is a directed edge between nodes A and B if control may flow from basic block A directly to block B. This is the case if the last statement in A is a branch to B, or when B is on the fall-through path from A.

Control-flow graphs have the advantage that they can express arbitrary control-flow, but on the other hand they are not well–suited for optimizers. Information about the high-level control structures has been lost, and all information about what is executed under which conditions is only implicit. For example, loops are translated into cyclic subgraphs, and statements belonging to a single statement sequence are separated into different basic blocks when an intervening statement includes any kind of control-flow. Besides control structures, there are several operations in Oberon which translate into control-flow. For example the standard function ABS(x) compiles to a conditional statement on machines without an instruction to compute the absolute value of x, as does the arithmetic shift function ASH on machines which do not interpret the sign of the shift count. Even index-checks or NIL-checks are compiled into control-structures on several machines.

The control-flow graph of a program can be built while parsing the source program with algorithms that have linear time complexity in the size of the program.

In order to expose more optimization potential, many refinements to simple control-flow graphs have been proposed. They are built from the control-flow graph and find structural information by applying graph-theoretical algorithms.



```
IF $cond_0$ THEN
  $stat_0$;
  WHILE $cond_1$ DO $stat_1$ END
END
```

Figure 5.2: A control-flow graph.

Extended Basic Blocks

Consider how the property of availability could be determined in a control-flow graph. A result must have been computed before in the program in order to be available at some point P, no matter which path was taken to P. Obviously, a result is available within the basic block in which it is computed, but this is a very pessimistic solution. A better solution requires to analyze the branching structure of the program.

A simple refinement is based on the observation, that a result remains available in an instruction sequence after a branch. Only at a label paths join and availability has to be determined by considering all paths. This leads to the notion of an *extended basic block*.

Definition: An *extended basic block* is a sequence of consecutive statements, in which flow of control enters only at the beginning but may leave at several points.

More intuitively, we do allow branches out of an extended basic block, but we do not allow other entry points into the block than at the very beginning. This guarantees that at some instruction in an extended basic block, all preceding instructions in the same block have been executed, and thus their results are available.

Even though this delivers better results than just considering basic blocks, any structured statement will still make all results unavailable that have been computed before.

Dominator and Postdominator Trees

The flaws of not being able to determine availability over basic block or extended basic block boundaries can be avoided through the concept of *dominance*.

Definition: In a directed graph with start node S, we say that a node A *dominates* node B, iff for all paths P from S to B, A is a member of P. We call A a *dominator* of B, and write $A \overset{\cdot}{\rightarrow} B$. [ASU86]

Dominance is a relation on the nodes of the graph which is reflexive, transitive, and anti-symmetric.

Definition: We call A the *immediate dominator* of B, iff $A \overset{\cdot}{\rightarrow} B$, $A \# B$, and $\neg\exists C, A \overset{\cdot}{\rightarrow} C \wedge C \overset{\cdot}{\rightarrow} B$. We write $A \rightarrow B$. [ASU86]

Definition: The *dominator tree* of a directed graph G with start node S is the tree including the nodes of G, S being the root, and having an edge between nodes A and B iff $A \rightarrow B$.

Note that the dominator tree really is a tree, as each node has exactly one immediate dominator which becomes its parent in the tree. An exception is start node S that does not have any dominators by definition and represents the root of the tree. All nodes being predecessors of some node A are dominators of A. Applying the dominance theory to a control-flow graph, we can say that if a basic block A dominates basic block B, A is on every path from S to B, and thus the statements in A have always been executed when control reaches B.

A necessary condition for the result of an instruction at position A to be available at position B is that A dominates B, and that the operands cannot be overwritten on any path from A to B. Such paths have the property that they are also dominated by A.

Theorem: If $A \overset{\cdot}{\rightarrow} B$, all nodes on paths P from A to B either include A or are dominated by A.

Proof: Assume the contrary. Then, there would be a node C, $\neg A \overset{\cdot}{\rightarrow} C$, and a path from C to B not including A. Since $\neg A \overset{\cdot}{\rightarrow} C$, there would be a path from S to C not including A. By appending the path from S to C to the one from C to B, a path from S to B not including A can be constructed. This contradicts the premise $A \overset{\cdot}{\rightarrow} B$. ∎

By traversing the dominator tree in preorder and keeping track of which values are available at a certain dominating node, problems dealing with the availability property can often be resolved in a simple way.

For some purposes, also the concept of the *postdominator tree* is useful.

Definition: In a directed graph with stop node Z, we say that a node A *postdominates* node B, iff for all paths P from B to Z, A is a member of P. We call A a *postdominator* of B, and write $A \overset{\cdot}{\leftarrow} B$. [ASU86]

The definitions for *immediate postdominator* and the postdominator tree are analogous to the above definitions for the dominance relation. The postdominator tree can be computed as the dominator tree of the *reverse control-flow graph*, that is the graph having the stop node Z as root and connecting control-flow successors with their predecessors by a directed edge.

For the class of operations that define the global state, most notably store instructions, our common subexpression elimination algorithm must be modified. Replacing an instruction with a similar instruction storing to the same location available at that point would yield wrong results. Instead of using the instruction that dominates similar instructions, one has to use the instruction that postdominates other stores to the same location.

The dominator tree can be computed from the control-flow graph. The algorithm due to Lengauer and Tarjan [LeTa79] makes four passes over the CFG, and runs in time $O(N * \alpha(N))$, where N is the number of nodes in the CFG and α is the inverse of the Ackermann function.

Control-Equivalence and the Control-Dependence Graph

While the algorithm based on the dominance relation to determine availability delivers better results than algorithms for basic blocks or extended basic blocks, it still exhibits an asymmetry. Instructions preceeding a control structure can replace instructions within the control structure, but instructions succeeding the control structure cannot. This is obvious, since the result is not available within the control structure. However, by moving this succeeding instruction in front of the control structure, the semantics of the program would not be changed, and the result would become available. The movement changes only the order of evaluation, and not the conditions under which the instruction is executed. In fact, whenever the block preceeding the control structure is executed, the block succeeding it is known to be executed as well. We call these two basic blocks *control-equivalent*, as they are executed under the same control conditions. Instructions can be freely moved between control-equivalent blocks without altering the semantics of the program as long as no data dependencies are violated.

Definition: Two basic blocks A and B are *control-equivalent*, iff $A \dot{\rightarrow} B$ and $B \dot{\leftarrow} A$.

For control-equivalent nodes A and B, there must be a common dominator D including the conditional branch controlling execution of A and B. We say that A and B are *control-dependent* on D, as in D it is determined whether A and B are to be executed. A more formal definition due to [FeOW87] is as follows.

Definition: A node B is *control dependent* on node A, iff there exists a directed path P from A to B with any $Z \in P$ (excluding A and B) postdominated by B and A is not postdominated by B.

If B is control-dependent on A, then A must have multiple successors. Following one path from A results in B being executed, while taking others may result in B not being executed.

Definition: The *control-dependence graph (CDG)* over a control-flow graph G is the graph over all nodes of G, in which there is a directed edge between nodes A and B, iff B is control-dependent on A.

Note that control-equivalent nodes are control-dependent on the same node, and thus have the same parent node in the control-dependence graph. The CDG compactly encodes the required order of execution due to control. A node evaluating a condition on which the execution of other nodes depends has to be executed first. As long as this condition holds and data dependencies allow, nodes can be moved. Moreover, single instructions can be moved between control-equivalent nodes.

Definition: A node B is *transitively control dependent* on node A, iff there is a path from A to B in the CDG.

Reconsidering the example of common subexpression elimination, a result is available if it has been computed in a node on which the current node is transitively control-dependent or a control-equivalent node.

From the control-dependence graph, a possible control-flow graph can be reconstructed [BaHo92]. It can serve as a replacement for the control-flow graph in most situations. Note that it recovers some of the high-level information about the nesting of control structures that has been lost in the translation to a CFG. The control-dependence graph encodes, which blocks are executed under the same conditions in the source program, but does not make the conditions explicit like guarded forms. Note that it does not explicitly support combining conditional branches based on congruent comparisons. *If-conversion* [AKPW83] allows to construct guards from control-dependencies, but only for parts of the program which exhibit structured control-flow.

The control-dependence graph can be built from the control-flow graph and the postdominator tree using an algorithm with three passes and having a time complexity of $O(N^2)$, where N is the number of nodes in the CFG [FeOW87].

5.3 Abstraction Levels for Data-Flow

In order to optimize a program, it is not sufficient to know that a certain variable is used as an operand by some instruction. One needs to know which assignments to the variable affect the operand value, as the variable can take on different values when executing the program. In our CSE example, two computations of *a+b* are only equivalent, if both computations use the same value of *a* and *b*. It depends on the intermediate representation how easily different values of a variable can be identified.

Multi-Assignment Intermediate Languages

Traditional imperative high-level languages allow multiple assignments to the same variable, with the consequence that the same variable may take on different values during program execution. This is also the case for the source language in question, Oberon-2. Using an intermediate representation that also allows multiple assignments to the same variable is a natural match for them, but has an impact on the analysis methods used.

Traditional data-flow analysis techniques use *bit-vectors* to determine different properties of variables and anonymous computations. Anonymous computations receive temporary names. Often the names are reused in such a way, that lexically identical expressions receive the same temporary name [AuHo82], e.g. all computations *a+b* receive the temporary name t_1.

Thus, having two results with the same temporary name is a required but not sufficient precondition for congruence. It is also required that none of the operands can be modified in between two congruent computations. This precondition can be determined together with the precondition of availability of an expression, as the following algorithm for common subexpression elimination shows. It assumes to be run on a control-flow graph, but could be adapted to other control-flow representations.

The algorithm assumes that lexically-equivalent expressions have received identical temporary names. It uses a bit-vector with a bit for each temporary name, which indicates whether a certain expression is available at the current point. For each basic block, a bit-vector *avail-out* containing the expressions available at the end of the block is kept.

```
PROCEDURE CommonSubexpressionElimination (CFG: Graph);
   VAR avail: Bitvector; B, P: Block; instr: Instruction;
BEGIN
   FOR each basic block B in CFG DO
      B.avail-out := {}
   END;
   REPEAT
      FOR each basic block B in CFG DO
         IF B is start node THEN avail := {}
         ELSE
            avail := {0..n};
            FOR all predecessors P of B in CFG DO
               avail := avail ∧ P.avail-out
            END
         END;
         FOR each instruction instr in B in order DO
            IF instr.result IN avail THEN Delete(instr)
            ELSE
               remove all results dependent on instr.name from avail;
               INCL(instr.name, avail)
            END
         END
      END
   UNTIL done
END CommonSubexpressionElimination;
```

The algorithm at first makes the conservative assumption that nothing is available at the end of a block. It traverses each block, determining what is available at the top of the block by taking the intersection of *avail-out* of all predecessors. This corresponds to the rule that an expression is only available at the beginning of the block if it is available along all paths leading to this block. Then, all instructions in the block are traversed. If the result an instruction computes is already available, the instruction can be deleted. Otherwise, the newly computed value for some name *v* invalidates computations that used *v* as operand, and thus these computations can be considered as not being available anymore. We say that the newly computed value *kills* the previous value and thus all dependent expressions. Note that the transitive closure of such dependent computations has to be removed from the set of available expressions. Since the expression was just computed, it can then be added to the set of available results.

We have not specified under which conditions the algorithm terminates. In principle, it can be terminated at any point, with the consequence that some common subexpressions may not be found. In order to find all common subexpressions, and in the presence of loops, this algorithm has to be iterated several times over the CFG until a fixed point is reached. Finding that an expression *t* is available from a previous loop iteration can cause killing definitions of *t* to be deleted, and thus dependent expressions to become available at the end and the top of the loop. The upper limit for the number of iterations required to reach a fix-point for a loop is the number of instructions in the longest dependence chain of operations in the loop, which is at most the number of instructions in the loop. For nested loops, inner loops may have to be considered several times.

Relying on lexical equivalence has the disadvantage that common subexpressions like in the following example are not found.

(1) $t_0 := a + 1;$
(2) $b := a;$
(3) $t_1 := b + 1$

Even though *a* and *b* have the same value in this context, the incremented values receive different temporary names and thus will not be found to be congruent.

Static Single-Assignment Intermediate Languages

Static single-assignment programming languages like Sisal [BGOCF93] have not found widespread use among programmers. While their functional programming model offers a lot of interesting properties for parallel computing, it is sufficiently different to require a paradigm shift on the side of the programmers.

However, programs in imperative programming languages can be translated easily into static single-assignment (SSA) form. Compilers may use static single-assignment intermediate representations to benefit from its properties, with the source language adhering to a standard imperative programming paradigm.

Reconsider our example of common subexpression elimination, applied to straight-line code. If there are multiple assignments to some variable *a*, the optimizer has to keep track for each use of *a* by which assignment it received its value. One simple method to do so would be to give a unique number to each assignment, and to store the number along with the use of the variable. Figure 5.3 shows a piece of code with and without this numbering, where the numbers are appended as suffixes to the variables.

(1)	$a := 5$	(1)	$a_1 := 5$
(2)	$x := a + 1$	(2)	$x_2 := a_1 + 1$
(3)	$y := a + 1$	(3)	$y_3 := a_1 + 1$
(4)	$a := b * 2$	(4)	$a_4 := b_0 * 2$
(5)	$z := a + 1$	(5)	$z_5 := a_4 + 1$

Figure 5.3: Straight-line code and its static single-assignment representation.

Common subexpression elimination becomes very simple, as for two instructions only the opcodes and the operands with their suffixes have to be compared in order to determine congruence. If one of the operands were assigned a new value between its uses, the suffix would be different and thus the operands would not be equivalent. In the example, the computations for *y* and *z* are not congruent due to the differently numbered variable *a*.

Seen from a slightly different angle, each of the suffixed variables could be seen as a separate temporary copy of the original variable, with each copy being assigned only once. This is the main property of static single-assignment languages, namely that for each variable there is exactly one assignment in the program text. One can also say that each assignment in the program text receives a unique name, by which the result of the computation can be referenced.

The actual numbering does not play a role as long as each assignment can be uniquely identified, e.g. by a unique pair of variable name and number, or simply by a unique assignment number, thus dropping names altogether. Also, instead of storing assignment numbers, one could keep references directly to the assignments using pointers, which simplifies lookups.

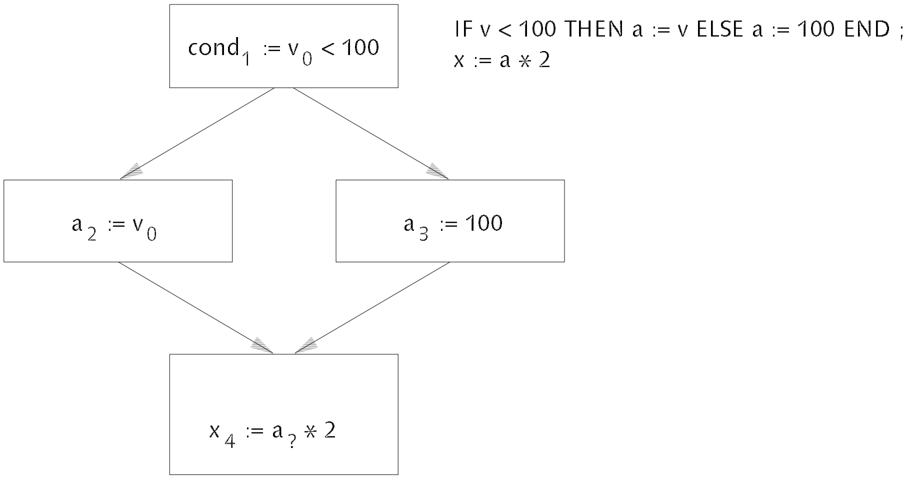
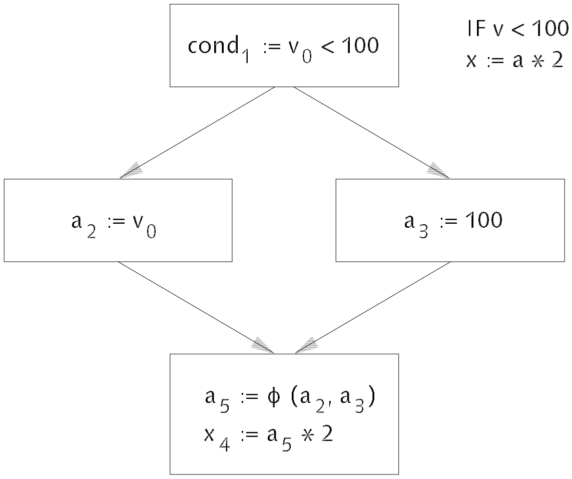


Figure 5.4: A static single-assignment program with control flow.

In the presence of control flow, there are sometimes multiple assignments that can affect the value of an operand at a certain point, as shown in Figure 5.4. Depending on the value of $cond_1$, either a_2 or a_3 has to be used in the computation of x_4.

Instead of keeping just a single assignment-reference for an operand, one could store references to all assignments that can affect the value. Doing so is known as using *def-chains* [ASU86]. That is, for each operand there is a chain of references to assignments possibly defining it. These chains can become long in the presence of complex control-flow, however, and maintaining and comparing them is complex and error-prone.

As an alternative to maintaining long def-chains for each operand, a placeholder for the chain may be generated that can then be referenced like an assignment [FeOW87]. When two control-flow paths join, and different instances of some variable *v* are available along these paths, a *ϕ-function* for *v* is generated referencing both instances, and serving as a placeholder for the chain of them. The ϕ-function is assigned to *v*, thus creating a new single instance of the variable that can be used thereafter (Figure 5.5).



```
IF v < 100 THEN a := v ELSE a := 100 END ;
x := a * 2
```

Figure 5.5: A static single-assignment program with control flow and ϕ-function.

As the name implies, ϕ-functions can not only be seen as placeholders for def-chains but also as functions. For a point at which n control-flow paths join, ϕ-functions have n operands, and implement a selection function. If control reaches the join point via the *m*-th path, the *m*-th operand represents the result of the function. An unfortunate property is that the selection criterion, namely by which path the join point is reached, is not an explicit parameter of the function. This prevents it from being treated as a function in the mathematical sense. A list of predicates specifying the controlling conditions for the individual paths would make them true functions. ϕ-functions with such predicates are named *gates*, and the resulting data-flow framework is called *gated single-assignment* form. Such predicates are not available in the presence of unstructured control-flow. As a conservative approximation, control dependencies or basic blocks can be used as parameters. In this case, each basic block is considered as a placeholder for an unspecified predicate controlling its execution, and all such predicates are treated as being different. Note that this reduces the optimization potential.

Unlike other instructions, ϕ-functions will not be translated into actual machine instructions. By allocating all operands and the result to the same storage location – including the same register – the function can be turned into a no-operation and deleted. Allocation of all operands to the same location can always be achieved, possibly by introducing move-operations on some paths to the gate.

Static single-assignment form has a couple of advantages over multi-assignment representations. The dependence between an instruction computing a value and another instruction using it, is explicit rather than implicit through variable name and position. Instructions can be freely moved without affecting semantics; they still use the same operands and produce operands for the same instructions. We can also say that instructions have a meaning independent of their position, and the meaning is completely defined by the opcode and the operands. Moreover, no result can be killed by intervening assignments, thus availability of a result at some position is only a matter of it being computed on all paths to that position.

Note that SSA form corresponds to complete renaming performed in software. All anti- and output-dependencies have been removed, and only true dependencies are represented. This is equivalent to a static data flow graph of the program.

Static single-assignment form can be computed for programs with arbitrary control-flow using three passes over the control-flow graph, the dominator tree, and the postdominator tree [CFRWZ91]. The time complexity of the algorithm is $O(N^3)$, where N is the size of the program. We have presented an algorithm which can generate SSA form for structured languages in a single pass while parsing the source program. It has the same time complexity $O(N^3)$ [BrMö94].

Dynamic Single-Assignment Intermediate Languages

Static single-assignment languages give a unique name to each statically occurring assignment in a program. In a loop, the variable may still take on different values for each iteration, and thus kills values from previous iterations. Dynamic single-assignment form avoids this by giving unique names to assignments dynamically, as defined by Rau [Rau91].

Definition: A program representation is said to be in *dynamic single-assignment (DSA)* form if the same virtual register is never assigned more than once on any dynamic execution path.

A virtual register in Rau's context corresponds to a renamed variable in ours. He adds that the static code may have multiple statements with the same virtual register on the left hand side, thus programs in dynamic single-assignment form are not necessarily in static single-assignment form. A definition for variable *v* in an earlier iteration can be referenced as *v[i]*, where *i* corresponds to the number of iterations executed in between. *v[0]* is the value defined in the current iteration and can be abbreviated as *v*.

```
WHILE i < N DO                            WHILE i[0] < N DO
                                              remap(a);
    a[i] := a[i–1] + a[i–2];                  a[0] := a[1] + a[2];
                                              remap(i);
    i := i + 1                                i[0] := i[1] + 1
END  END
```

Figure 5.6: An Oberon program and its DSA form.

In Figure 5.6, the *remap*-operations are inserted for a variable *v* before *v* is assigned, allocating a new register for that variable. Older instances of the same variable are appropriately renamed. The compiler will later move all remap operations to the top of the loop, so that the new registers can be allocated all at once.

The representation closely reflects the hardware architecture of the Cydra 5 VLIW computer [RYYT89], in which different iterations are executed in different windows of the register file. It uses a circular register file, with a based index according to which all register accesses are performed, and which is incremented once for each iteration, thus using new registers for new results, but still allowing the access to earlier results by other numbers.

Rau describes a framework for optimizing array accesses making use of DSA form. It is based on the idea that an array expression *a[i]* being available before a statement *i := i+1* could be made available as *a[i–1]* after the increment, as shown in Figure 5.6. However, this reasoning also applies to other frameworks, and has been exploited by others [CaCK90] to improve the register allocation of subscripted variables without using DSA form.

The main attractions of DSA form are that loops are expressed in their maximal parallel form, i.e. without any anti- or output-dependencies between iterations, and that values from previous iterations can be reused. In environments without special hardware support, however, DSA form must be translated to machine code by either replicating the loop body or by copying registers. By doing so, the same effects would be achievable with other representations as well. We believe that DSA form does not simplify analysis of the program in any form.

5.4 Combining Control- and Data-Flow

The above discussions have assumed that control- and data-flow are represented separately, that is in different kinds of data structures. This requires two data structures to be maintained and separate optimization algorithms for them to be designed. It is hard to make use of interactions between optimizations in the data-flow representation and the ones for control-flow. For example, control-flow could be optimized when common subexpression elimination determines that two conditional branches use congruent conditions. Control-flow optimizations change paths, and may make results available in places where they were not before, thus affecting common subexpression elimination. Finally, if SSA-form is used, a clean representation of control-flow within the data-flow framework would allow to give ϕ-functions well-defined semantics.

These advantages have been noted long time ago, and several approaches have been made to combine both.

Program Dependence Graphs

Program dependence graphs (PDG) [FeOW87] combine control dependencies and data dependencies into a common framework. Data and control dependencies are modeled as def-chains. Each instruction has references to the block it is control-dependent on, and to the instructions defining its operands. In order to reduce the number of defining instructions stored for each operand, so-called *merge nodes* are used, which have similar properties as ϕ-functions and combine the list of definitions for several operations. The presentation of the framework does not explicitly mention SSA form, but uses its concepts for scalar variables. However, references to structures (arrays or records) are still modeled in a multi-assignment style with anti-dependencies and output-dependencies. This and the missing control-flow information in merge-nodes make some operations still dependent on their position in the PDG.

Program Dependence Webs

The *program dependence web (PDW)* [BaMO90] is a refinement of the PDG. It is computed from the PDG, and replaces the data dependencies by gated single-assignment form. Since control-flow predicates are required for gates, it restricts the control-flow to reducible flow graphs [Aho] and spends a lot of effort on determining these predicates from the CDG. Program dependence webs offer very clean semantics and support powerful optimization algorithms, but are complex to generate. The presented algorithm starts from the PDG, and requires five passes to construct the PDW, with a time complexity of $O(N^3)$, where N is the size of the program.

Value Dependence Graphs

Value dependence graphs (VDG) [WCES94] use gated single-assignment form to represent data dependencies, and do not explicitly model control flow at all. Instead, the concept of *demand-dependencies* is introduced: If a value x is used as a gate operand, and if that operand corresponds to some condition C, then x is *demanded* under condition C. When control dependencies are reconstructed, a computation of x must be placed in the region corresponding to C or in a dominating region. Looping is represented using recursive function calls in the VDG.

While avoiding control flow altogether as in VDGs is an attractive idea, we are unable to assess all implications this might have on optimization algorithms and the back-end, which has to reconstruct valid control flow. However, we consider VDGs an interesting topic for future research.

Guarded Single-Assignment Form

Guarded single-assignment (GSA) form is the intermediate representation used in our optimizing Oberon-2 compiler OOC2. It combines a high-level representation of control-flow, namely guarded instructions, with instruction lists at the machine level and a static data flow graph. All information is combined in a single graph.

The representation has been inspired by the program dependence web, but includes the innovation of merge instructions to model confluence of control predicates, and has been simplified to serve the requirements of structured programming languages. It can be generated in one pass while parsing the source program and thus can replace all other intermediate representations used in optimizing compilers. The next chapter will discuss the GSA representation in detail and describe how it can be generated.

5.5 Discussion

Table 5.1 classifies several used or proposed intermediate representations according to the above mutually independent attributes. OP2 is the portable Oberon-2 compiler [Crel91] using an abstract syntax tree to achieve separation of front-end and back-end and thus portability of the front-end. Optimization algorithms were not considered in its design. GNU CC [Stall94] is based on traditional control-flow and data-flow frameworks, and spends a lot of effort to ensure portability, both to other source languages and other target machine architectures. OOC2 takes the opposite route in several respects, with the goal of reducing the complexity of optimizing compilers significantly. It combines the most recent results in optimizing compiler research and provides very clean semantics that makes all dependencies explicit. By restricting the source language to structured languages, it achieves clean semantics in the intermediate representation, and avoids complex graph algorithms in the front end. The machine-dependent intermediate representation allows omitting the translation from the intermediate level operators into real machine instructions.

	control flow	data flow	operator level
OP2 AST	control structures	multi-assignment	source language
GNU CC	CFG	multi-assignment	sub-machine language
PDG	CDG	multi-assignment / SSA	unspecified
PDW	CDG	gated single assignment	unspecified
OOC2	guards	guarded single assignment	machine language

Table 5.1: Classification of intermediate representations.

6 Guarded Single-Assignment Form

This chapter discusses what guarded single-assignment form as used in OOC2 looks like in detail. Besides presenting example programs in GSA form, it will be shown how instructions, control structures, data accesses, procedure calls, and aliasing effects are modeled.

Every procedure is translated into a list of instructions in GSA form, of which some may represent structured statements using guards and merges.

There are some points to know about how to read listings of GSA form. An instruction consists of an opcode followed by a list of operands. An operand can be a constant, a variable, a type, or a result of another instruction. Each instruction has a number identifying it, which can also be used to reference its result by writing it within parentheses. If there are multiple results, a second number separated by a colon is used to identify the result referenced. The first result has the number 1, and does not have to be explicitly specified.

Instructions that represent predicates controlling execution include a colon at the end of the opcode. For better identification, they are printed in italics. Instructions whose execution is controlled by such predicates are printed below them, indented appropriately.

If the result of an instruction is assigned to a variable in the source, the name of that variable will be printed on the left. Note, however, that this is only for readability purposes, and that this information is not used by the compiler in any way.

Instructions

Instructions are represented by an opcode node, operand nodes, and result nodes, all chained together (Figure 6.1). There are links to support the following important tasks.

- For an operand or result, find the instruction it belongs to. For an instruction, traverse all its operands or results.
- For an instruction, find the control-condition (*region*), under which it is executed. For a region, traverse all instructions that are executed under its control.
- For a region, traverse all regions that are nested in it.
- For an operand, find the place where it is defined, which can be either a result, a constant, or a variable. For a result, constant, or variable, traverse all operands where it is used (*use-chain*).

Furthermore, there are type- and location-attributes (see below) for all operands and results. Results additionally contain a general purpose field *info*, which can be used by optimization algorithms to store arbitrary attributes.

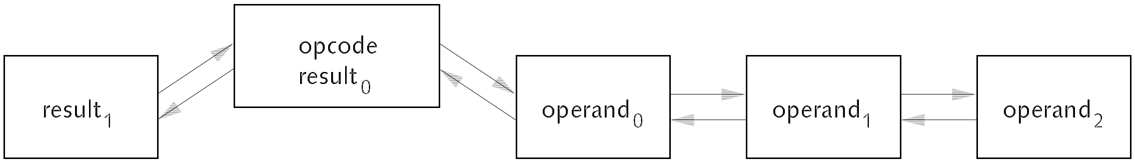


Figure 6.1: Representation of an instruction.

6.1 Control Structures

Guards and merge instructions are the building blocks for representing control structures. Both control the execution of a statement sequence, and are commonly referred to as *regions*. In addition, there is one global region *greg* representing the body of the procedure, which will always be executed. It contains the instructions at the outermost level. Note that regions are special instructions that participate in instruction lists just like ordinary instructions.

Guards

Guards represent predicates controlling the execution of statement sequences, and are simultaneously paths through the program. Which view is used depends on the context. In contrast to the presentation in Chapter 5, guards do not take a boolean value as operand and check it for being TRUE or FALSE, but rather take the result of a comparison (a *condition code*) and determine whether a certain condition holds. This more closely models the comparisons and conditional branches of the PowerPC architecture. Possible conditions are listed in Table 6.1.

condition	=	#	<	>=	>	<=
guard	if-eq:	if-neq:	if-less:	if-gteq:	if-gtr:	if-lseq:

Table 6.1: Mapping of conditions to guards.

Merges

Merges combine several predicates by Or-ing them. They are used to model short-circuit evaluation in Oberon, and to provide a list of predicates to gates, determining which operands are selected in the gate. The *n*-th operand of the merge corresponds to the *(n+1)*-th operand of the gate being returned as the result. For practical reasons, the compiler distinguishes between *c-merges* for Or-ing conditions, *i-merges* for terminating If-statements, and *l-merges* for closing loops.

If-Statements

As shown in Chapter 5, simple If-statements are modeled using one guard for the THEN- and ELSE-paths each, and an i-merge operation to combine the paths at the end. This i-merge operation serves as the predicate operand for succeeding gates. Note that the guards for the respective paths always specify exactly complementary conditions.

For a compound condition with And operators, the guards are nested according to the order of the operands from left to right, reflecting the short-circuit evaluation of Oberon. Or-ed conditions are modeled using c-merge instructions combining two paths.

In If-statements with ELSIF-clauses, two guards obviously cannot be sufficient. Furthermore, the semantics of Oberon require that the conditions are evaluated in source order. As soon as a condition evaluates to true, the corresponding statement sequence must be executed and the remainder of the If-statement be skipped.

```
IF $cond_0$ THEN $stat_0$
ELSIF $cond_1$ THEN $stat_1$
ELSE $stat_2$
END
```

If, in the above example, $cond_0$ would evaluate to TRUE, $stat_0$ would have to be executed and $cond_1$ as well as all the rest would have to be skipped. In other words, $cond_1$ must only be executed if $\sim cond_0$. This can be expressed by nesting the evaluation of $cond_1$ into the guard $\sim cond_0$, which in fact would even be a valid transformation at the source level.

```
IF $cond_0$ THEN $stat_0$
ELSE   (* $\sim cond_0$ *)
   IF $cond_1$ THEN $stat_1$
   ELSE $stat_2$   (* $\sim cond_1$ *)
   END
END
```

All complex If-statements are transformed into such a nesting of simple If-statements. Figure 6.2 shows a simple If-statement with a compound condition and its corresponding GSA form.

```
IF (0 < a) & (a < 100) THEN a := a * 10 ELSE a := a DIV b END

(1) greg:
       (2) cmp    0, a
       (3) if-less:     (2)                          ; 0 < a
              (4) cmp    a, 100
              (5) if-less:  (4)                      ; a < 100
                     (6) a := mul    a, 10
              (7) if-gteq:  (4)                      ; a >= 100
       (8) if-gteq:  (2)                             ; 0 >= a
       (9) c-merge:  (8), (7)                        ; (0 >= a) OR (a >= 100)
              (10) a := div    a, b
       (11) i-merge:  (5), (9)
       (12) a := gate    (11), (6), (10)
```

Figure 6.2: An Oberon If-statement and its corresponding GSA form.

While- and Repeat-Loops

Loops are represented with an l-merge node representing the loop header, which controls execution of the loop body. The l-merge combines the path leading to the loop and the path leading back from the end of the loop. The difference between a While-statement and a Repeat-statement is as follows. In a While-statement, the predicate is evaluated at the beginning, and it guards the execution of the statement body. In a Repeat-statement, the predicate is evaluated at the end and the statement body precedes it.

Figures 6.3 and 6.4 present two simple counting loops and their GSA form.

```
WHILE i < 100 DO i := i + 1 END

(1) greg:
          (2) l-merge:   (1), (5)
                    (3)  i := gate    (2), i, (6)
                    (4)  cmp    (3), 100
                    (5)  if-less:   (4)
                              (6)  i := add    (3), 1
                    (7)  if-gteq:   (4)
```

Figure 6.3: An Oberon While-statement and its corresponding GSA form.

```
REPEAT i := i + 1 UNTIL i >= 100

(1) greg:
          (2) l-merge:   (1), (6)
                    (3)  i := gate    (2), i, (4)
                    (4)  i := add    (3), 1
                    (5)  cmp    (4), 100
                    (6)  if-less:   (5)
                    (7)  if-gteq:   (5)
```

Figure 6.4: An Oberon Repeat-statement and its corresponding GSA form.

Other Control Structures

Even though we have not implemented Case-statements in OOC2, we would like to sketch how they may be represented. Note that a Case-statement has similar properties as an If-statement, but does not select a path according to a boolean value, but rather according to a numeric value serving as index of a list *L* of statement labels. We propose to introduce a special *case-guard* taking *E* and *L* as parameters, and causing execution of the nested instructions if $E \in L$. Since there can be more than two paths through a Case-statement, merges and gates with variable numbers of operands would be required as well.

Guarded single-assignment form does not directly support unstructured flow of control like in Exit- and Return-statements. For a more detailed discussion, see Section 7.4. Since Exit-statements are not supported in OOC2, Loop-statements were omitted as well.

6.2 Access to Structured Data

So far our GSA framework has only dealt with references to scalar variables. For structured variables, operations to access only a part of a variable are required. As proposed by [CFRWZ91], we use *access*- and *update*-instructions for this purpose.

Access-instructions allow the extraction of an item of the structured variable, which is a field in case of a record variable, or an array element in case of an array variable. The parameters are the variable to be accessed, the computed address of the item, and either the field offset of a record, or the indices used to access the array. The following example presents both kinds of accesses.

```
x := r.f                                  (1)  add    adr-r, offset-f
                                          (2)  x := access    r, (1), offset-f
```

```
y := a[i]                              (3)  trapw    >=u, i, LEN(a)
                                       (4)  mul    i, elsize-a
                                       (5)  add    adr-a, (4)
                                       (6)  y := access    a, (5), i
```

There is some redundancy in using both the computed address, the variable, and the offset or index values as operands. However, the variable is required in further analysis steps, and the offset or index values are helpful in distinguishing between accesses to different parts of the variable. On the other hand, the code to compute the address has to be emitted at some point in time, and if it is to be optimized as well, it should be present right from the beginning. Eventually access-instructions will be translated into machine loads, at which point superfluous offset- and index-parameters will be discarded.

Assignments to record fields or array elements are modeled in a similar way by update-instructions. In addition to the parameters of access-operations, they include an operand representing the value with which the field or element is to be updated. Conceptually, the result of an update-instruction is a new structured variable with the specified item updated, which is then assigned to the original variable.

```
r.f := x                               (1)  add    adr-r, offset-f
                                       (2)  r := update    r, (1), offset-f, x
```

Assignments to a structured variable are modeled as assignments of the whole variable. This is a conservative treatment, as without precise information about the item accessed, it is not known which accesses to the variable are affected by the assignment. Thus, the pessimistic assumption is made that the whole variable is modified. Note that subsequent accesses will use the updated value, as shown in the following example. For clarity, the address computations have been omitted.

```
x := a[i]                              (1)  x := access    a, adr1, i
a[j] := y                              (2)  a := update    a, adr2, j, y
z := a[i]                              (3)  z := access    (2), adr1, i
```

If instruction (3) would not use the updated array but the original one, a common subexpression elimination algorithm would find instructions (1) and (3) to be congruent, even though the accessed array element may have been changed by the update. Without further information about *i* and *j*, the whole array must be assumed to be updated. In fact, the redundant index- and offset-operands have been included to allow optimization algorithms to disambiguate such references.

Note that update-instructions will be translated into machine stores and do not really assign the whole structured variable.

If the structured variable is anonymous, such as in an access through a pointer, no variable can be specified as the first operand. In this case, the type of the structured variable is used as a placeholder. This maps all anonymous variables of the same type onto one parameter, which also is a conservative solution. If two pointers P and Q point to the same object, an assignment through P will also change the variable seen through Q. In the next access through Q, the updated value will be used as an operand, conservatively assuming that it has been changed.

In a programming language such as Oberon, there are also memory accesses to meta-information like type descriptors or type tags, and many implementations will include accesses to constants or module-linkage pointers in memory. Note that they do not belong to any variable or type, and that they are never written by the program. In access-instructions, special meta-variables are used as placeholders for the first parameter in such cases, one for each kind of meta-information. Common subexpression elimination will be able to remove redundant accesses.

6.3 Aliasing

Two designators are called *aliases*, if they are not identical but refer to the same storage location. In Oberon, aliases occur in conjunction with pointers and VAR parameters.

```
PROCEDURE p (VAR x, y: INTEGER);
BEGIN
  x := x+1; y := y+1
END p;
```

If the procedure is called as *p(a, a)*, *x* and *y* will refer to the same variable inside *p*, and the observed effect would be that *a* is incremented twice. We call multiple names referring to the same storage location *aliases*, i.e. *x* and *y* are aliases in this scenario. If *p* were called with two different parameters, no such aliasing would occur, and two different variables would be incremented.

When looking just at procedure *p*, it is unknown whether *x* and *y* are aliases or not, since this depends on the actual parameters passed to *p*. In this case, we call *x* and *y may-aliases*. They are *must-aliases* if they are known to be aliases or *no-aliases* if they are known not to be aliases, respectively.

Note that the outlined translation of *p* to GSA form would yield incorrect results in the case of *x* and *y* being aliases.

$x_1 := x_0 + 1;$
$y_1 := y_0 + 1$

In GSA form, the effects of aliasing have to be expressed using assignments. If a designator is assigned, all designators that are must-aliases have to be assigned the same value. If there are may-aliases, they may receive the new value or keep their original one, which is expressed using a selector function [CyGe93].

$x_1 := x_0 + 1;$
$y_1 := \mathrm{MayAlias}(x, y, x_1, y_0);$
$y_2 := y_1 + 1$

The meaning of the *MayAlias* function is: If the first two operands refer to the same memory location, select the third operand, otherwise the last operand. Note that the first two operands are the objects that may alias, and not the current values of these objects. The addresses of these objects determine whether the third or the fourth operand corresponds to the result.

At the machine level, the MayAlias function can be turned into a no-op by requiring the third and fourth operand being in their *home memory location*, the location into which a simple compiler would store them. If *x* and *y* are aliases, requiring x_1 and y_0 to be in memory guarantees that x_1 has overwritten y_0, and the fetch of y_1 will yield the correct value. Alternatively, the MayAlias function can be implemented by explicitly comparing the addresses and conditionally assigning the values, which may yield better code on some machines.

If there is an assignment to a potentially aliased variable *v*, MayAlias functions have to be generated for each variable that may be aliased to *v*. In programs making heavy use of reference parameters and pointers this can have a large impact on the size of the intermediate representation.

In OOC2, the MayAlias function is split into an *address-compare* function, and a *selection* function. This models more closely the translation of MayAlias functions using a compare and a conditional assignment. The address-comparison can often be propagated out of loops, or simplified in conjunction with optimizations on addressing code.

6.4 Locations

There are situations in which variables have to be in certain locations. For example, when a value is passed as a parameter to a procedure, it must be in the register or stack location that the calling convention dictates. If a variable is passed by reference, the most recent value computed for it must be stored at the address that is passed. Within a procedure, parameters are initially found at the location given by the calling convention. If MayAlias nodes are to be transformed into no-operations, both parameters and the result must be allocated in their home memory locations. Furthermore, there are certain instructions that deliver results in special registers; for example ALU-operations with the record-option return their condition code into CR0.

In order to model these requirements, all operands and results have a *location* attribute specifying allocation constraints. The constraint can be *none*, a certain register class, a certain register, or a memory location. The concept has been generalized to include other operations as well. For example, integer operations in the PowerPC architecture read their operands from integer registers and write their result to an integer register. Thus, the operands and the result are constrained to the class of integer registers.

The compiler includes an algorithm that traverses all uses of all values, introducing appropriate move-operations where constraints are not satisfied, and resolving conflicts where multiple values compete for the same location.

6.5 Procedure Calls

Procedure calls are modeled as *call*-instructions, which take as operands the procedure to be called and the parameters to be passed. For scalar value parameters the computed results will become the operands. For parameters passed by reference, that is VAR-parameters and structured value parameters, both the computed address and the actual variable are used as operands. The reasoning behind using both is similar as for modeling memory accesses. The address computation should be exposed to the optimizer and represents the data that will finally be passed to the procedure. The actual variable is required to guarantee the correct value flow, as exemplified in the next listing.

```
PROCEDURE P (VAR x: INTEGER);
END P;

PROCEDURE Q;
  VAR y: INTEGER;
BEGIN
  y := 10; ...
  y := 20;
  P(y)

GSA-listing for Q:
(1) greg:
     (2)  y := id    10
     ...
     (4)  y := id    20
     (5)  add    FP, offset-y
     (6)  call    P, (5), (4)
```

If instruction (4) were not used as an operand in the call-instruction, one of the following errors could happen. The optimizer may find instruction (4) to have no uses at all and delete it. Even if there were other uses of it, the optimizer may move it behind the call, thus passing the previous value of *y* to *P*. Finally, the compiler may decide to keep *y* in a register and to not update its home memory location – e.g. on the stack – which is the location accessed by *P*.

The last error cannot be avoided simply by using the actual variable as operand of the call. However, by constraining the operand to its home memory location, the optimizer will make sure that the correct value is actually stored before the procedure is called. The calling convention itself is satisfied by constraining all parameters to the dictated location, be it a register or a position on the stack. If auxiliary items like static links or static base pointers are to be passed, they are represented as ordinary parameters.

The effects of calling *P* on the variables seen in *Q* has not yet been modeled. *P* may modify *y* which has been passed as VAR-parameter, or assign variables that are accessible from both scopes. In GSA form, these modifications must also be expressed by assignments to the corresponding variables in *Q*.

For the following, assume that the set of variables assigned by *P* is known. For each variable *v* in this set, a new result node will be added to the call and assigned to *v*. Thus, if there are *N* variables modified in *P*, a call to *P* will become an instruction with *N* results. If *P* in the above example assigned to the variables *y* and *a*, the complete form of the call would be as follows. Note the added location attributes in its operands and results.

(6) y:loc$_y$, a:loc$_a$:= call P, (5):R3, (4):loc$_y$

In case the set of variables assigned by a procedure is not known, the pessimistic assumption must be made that everything could be assigned. In order to model this, a pseudo-variable *$mem* is introduced, which aliases with everything. Calls to such procedures assign to *$mem*, and cause MayAlias assignments to be introduced for all variables. Note that at assignments to *$mem*, there is a consistent view of memory, which is identical to the one a non-optimizing compiler would have generated.

The compiler must also make sure that variables accessed from both *P* and *Q* are in their home memory location before calling *P*. If the set of these variables is known, they can be appended as operands to the call-instruction, receiving an appropriate location-attribute. Again, if this set is not available, all variables must be put in their home memory location. Instead of adding them all as operands to the call, a consistent view of memory is generated, assigned to *$mem*, and *$mem* is used as an operand.

Generating a consistent view of memory requires to collect the last assignments for all variables since the last consistent view, that is the last assignment to *$mem*, and to store them into their home memory locations. This is achieved by a special *collect*-instruction, which includes all such assignments as operands attributed with appropriate locations. The result of this instruction is assigned to *$mem*.

The call to a parameter-less procedure *C*, of which neither the set of accessed nor the set of assigned variables is known would then be represented as follows.

```
(11)  $mem := collect   ...                    ; some variables
(12)  $mem := call   C, (11)
...                                            ; some MayAlias functions
```

The same mechanism is used to model the effects of the SYSTEM-procedures GET, PUT, MOVE, and BIT, which can access arbitrary locations and therefore also require a consistent view of memory.

6.6 Procedure Prolog and Epilog

In the previous section it has been discussed, how the calling convention of the target architecture is supported with location attributes on the side of the caller. A similar mechanism is used to model the calling convention on the side of the callee.

Each procedure starts with an *enter*-instruction and ends with a *return*-instruction. The enter-instruction includes a result for every item passed from the outside to the procedure, be it parameters, the return address, the stack pointer, static or dynamic links, and static base pointers. Every result has a location attribute corresponding to the position at which it will be found at procedure entry.

The return-instruction uses as operands everything that is passed back to the caller or that is used in the procedure epilogue. This includes the return-address and the dynamic link, as well as results returned. Again, these values have location attributes according to the calling convention. Furthermore, every variable outside of the procedure's scope that is assigned provides an operand. This way, all non-local effects of the procedure are summarized in the return-instruction. This is exactly the information needed in procedure calls to model their effect. Naturally, it is only available if the called procedure is statically bound, within the same module, and has been translated before. In all other cases, the compiler assumes that everything may be written by the call.

6.7 Ordering of Instructions

In intermediate program representations, instructions are usually ordered in the way a straightforward compiler would have generated them, which is a valid execution order. If a multi-assignment language is used, the meaning of an instruction even depends on its position, so the order is an (implicit) part of the semantics. All optimization algorithms must make sure that this part of the semantics is preserved.

In GSA form, however, all dependencies are explicitly expressed, and the semantics of an instruction are completely independent of its position relative to other instructions. Therefore, requiring the instructions to be in a particular order is unnecessary. Since the instruction scheduling algorithm will order instructions in a way that all dependencies are satisfied, placing any ordering constraints on other optimization algorithms is an unnecessary replication of concern. As an example, if some instruction is found to be *loop-invariant*, it can be moved to the region enclosing the loop. Where to insert it into the instruction list would require to determine its dependencies on other instructions, and introduce a significant overhead.

6.8 A Numbering Scheme for Fast Dominance Tests

There are several situations in which it must be tested, whether a region B is transitively nested in a region A, that is, whether $A \overset{*}{\rightarrow} B$. For example, if a result has been computed in region A, it is available in B if $A \overset{*}{\rightarrow} B$. Or if A and B are found to represent the same control condition, B will evaluate to true iff $A \overset{*}{\rightarrow} B$ and A # B.

The test could be performed by traversing the region nesting upwards from B until A or the root is reached, but for deep nestings, this may be impractical. In the following we propose a simple numbering scheme that allows us to determine dominance in a tree with two simple comparisons. The method is by no means restricted to our nesting of regions, but can be applied to any tree.

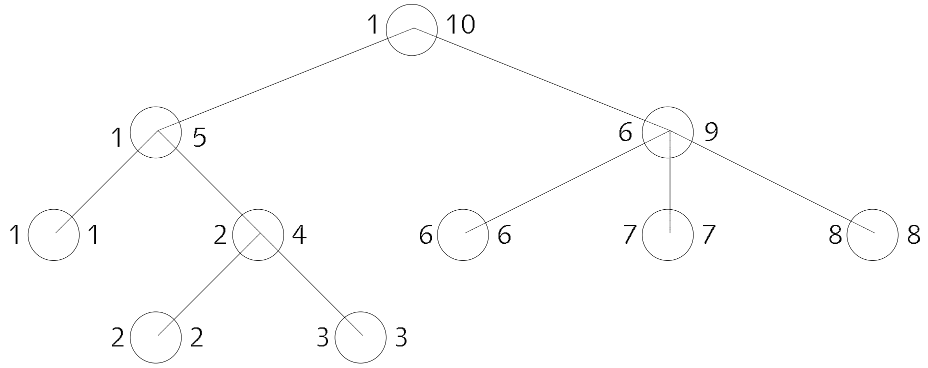


Figure 6.5: A numbering for a tree.

The idea is to map the half-order of dominance onto the half-order of subrange inclusion. A parent node is associated with a range of numbers that is the union of the ranges of all of its children plus a number for the parent node. We further require that nodes at the same depth of the tree receive distinct ranges (Figure 6.5). Then the following theorem holds.

Theorem: Let A and B be two nodes in a tree, with A.range and B.range corresponding to their respectively associated ranges. Then $A \overset{\cdot}{\rightarrow} B \Leftrightarrow A.range \supseteq B.range$.

Proof: By the first requirement for the associated ranges, $A \overset{\cdot}{\rightarrow} B \Rightarrow A.range \supseteq B.range$ holds. It must be shown that $A.range \supseteq B.range \Rightarrow A \overset{\cdot}{\rightarrow} B$ also holds. Assume the contrary. Then there would be nodes A and B, so that $A.range \supseteq B.range$, but not $A \overset{\cdot}{\rightarrow} B$. There must be some node Z, for which $Z \overset{\cdot}{\rightarrow} A \wedge Z \overset{\cdot}{\rightarrow} B$, and $Z.range \supseteq A.range \supseteq B.range$. Z has two children U and V, so that $U \overset{\cdot}{\rightarrow} A \wedge V \overset{\cdot}{\rightarrow} B$, $U.range \supseteq A.range \wedge V.range \supseteq B.range$. Since U and V are at the same depth of the tree, $U.range \cap V.range = \emptyset$. But also $U.range \supseteq A.range \supseteq B.range \wedge V.range \supseteq B.range$, $U.range \cap V.range \supseteq B.range$, contradicting our requirement for nodes at the same depth. ∎

Such a numbering can be generated by traversing the tree in postorder, giving monotonically increasing numbers N as ranges $N_i..N_j$ to leaf nodes, where the first node receives N_0, and the last node N_k. The parent node receives the range $N_0..N_{k+1}$. The numbering in the next subtree of the same level will start at N_{k+2}. The following procedure implements this numbering.

```
PROCEDURE NumberTree (node: Node; VAR n: INTEGER);
  VAR N: Node;
BEGIN
  node.rangeL := n;    (* store the lower bound *)
  FOR each successor N of node DO NumberTree (N, n) END;
  node.rangeH := n;
  (* store the higher bound, which is also the number associated with node *)
  INC(n)
END NumberTree;
```

After this numbering, a test whether a node is dominated by another translates into a simple subrange check. More precisely, since there are no overlapping ranges, a check of the upper bound identifying the node uniquely is sufficient.

```
PROCEDURE Dominates (X, Y: Node): BOOLEAN;
BEGIN
  (* X ˙→ Y ⇔ Y.rangeH ∈ X.range *)
  RETURN (X.rangeL <= Y.rangeH) & (Y.rangeH <= X.rangeH)
END Dominates;
```

This procedure corresponds to the reflexive definition of dominance. By excluding the upper bound from the range, one can also implement the non-reflexive variant where required.

```
PROCEDURE DominatesNR (X, Y: Node): BOOLEAN;
BEGIN
  (* X ˙→ Y ∧ X # Y ⇔ Y ∈ X.range–{X.rangeH} *)
  RETURN (X.rangeL <= Y.rangeH) & (Y.rangeH < X.rangeH)
END DominatesNR;
```

6.9 Implementation Issues

In this section, we give a short overview of the modularization of our prototype compiler OOC2 and of the data structure implementing guarded single-assignment form.

The compiler is based on the portable Oberon-2 compiler OP2 [Crel91] and shares its basic structure. The front-end constructs an abstract syntax tree of the program, and then calls the back-end to generate the code. The difference to other compilers based on OP2 as in [BCFT92] lies in the introduction of a *GSA phase*. Instead of directly emitting machine code, OOC2 generates a GSA representation for each procedure (GSA phase), calls optimization algorithms, and finally emits the code (back-end). Figure 6.6 shows this structure.

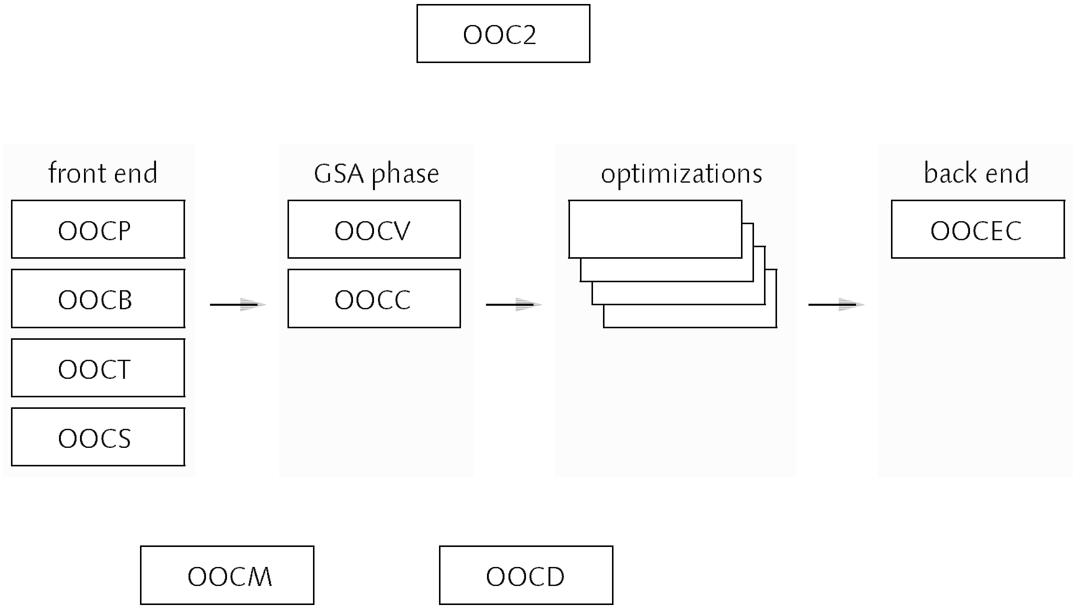


Figure 6.6: General structure of OOC2.

The front-end modules have not been changed: OOC*M* provides the host *machine* interface, OOC*S* and OOC*P* implement the *scanner* and *parser*, respectively, OOC*T* is the symbol *table* handler, and OOC*B* *builds* the abstract syntax tree. OOC*V* *traverses* the tree in order to generate the GSA representation by calls to the *code* generator OOCC. Module OOC*D* implements the abstract *data structure* for the intermediate program representation. Type extension has been used to model the structure. Figure 6.7 depicts the type hierarchy.

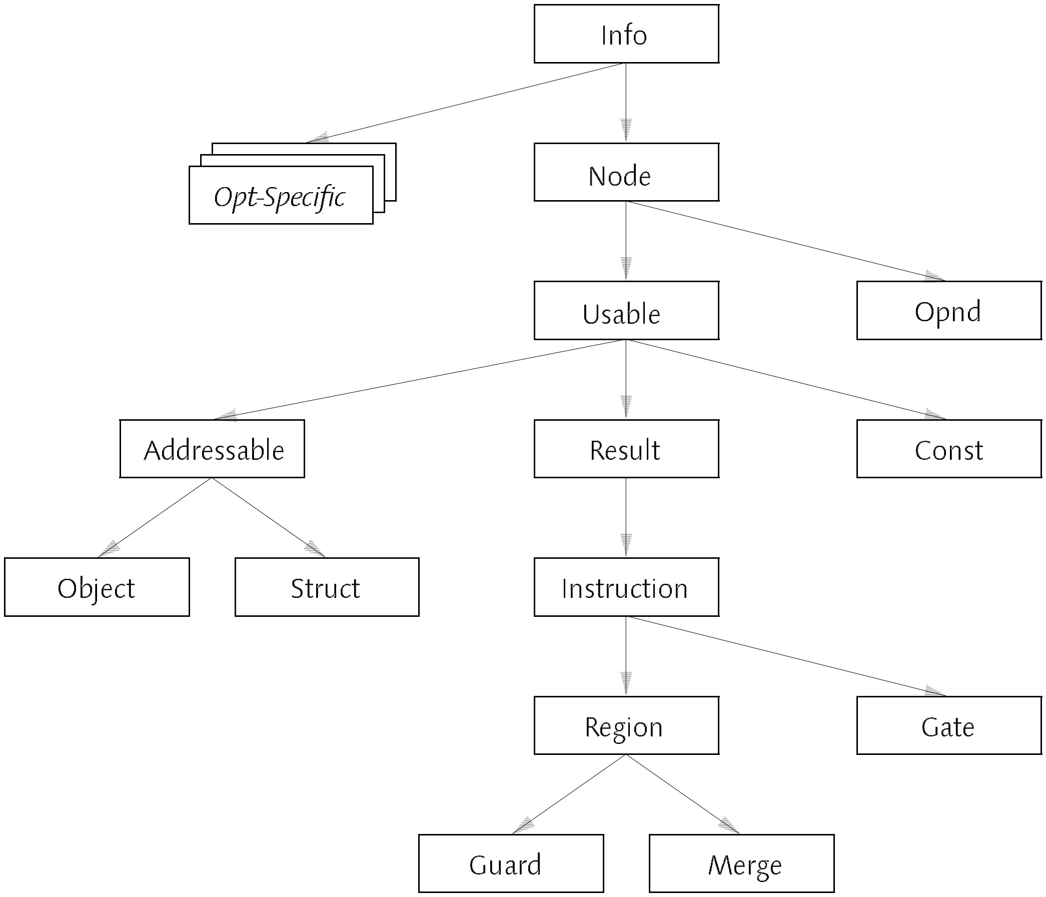


Figure 6.7: Type hierarchy in OOCD (an arrow from A to B corresponds to B extending A).

The hierarchy follows the *is-a* relation. For example, either constants (*Const*), results (*Result*), or something addressable (*Addressable*) can be used in computations, and thus represent *Usable*s. Regions are special instructions, and can be further specialized into guards and merges.

The *Info*-type at the root is empty and has been introduced to support optimization algorithms. *Result*s include an info-field of this type, which can point to any object of the data structure, or to optimization-specific attributes in types derived from *Info*.

As we have seen above, there are various lists maintained in the data structure. There is a list of instructions for each region, a list of operands and results for each instruction, and a list of uses for each *Usable*. In OOCD, all of the lists have been implemented as double-linked lists in order to allow for rapid traversal and rapid deletion. For the list of instructions and the list of uses, deletions or movements are frequent operations, which can be implemented efficiently only in double-linked lists, while for operands and results, a single-linked implementation may suffice.

OOCD provides procedures for all kinds of modifications to the data structure. The fields implementing the linked lists are exported to allow for quick traversal, but should be considered read-only outside of OOCD. The most important operations are shown in the following excerpt.

```
DEFINITION OOCD;

    (* operations on instructions *)
    PROCEDURE OpenInstruction (instr: Instruction);    (* initialize *)
    PROCEDURE OpenGate (gate: Gate);    (* initialize *)
    PROCEDURE Insert (region: Region; instr: Instruction);
    PROCEDURE Delete (instr: Instruction);
    PROCEDURE MoveInstruction (to: Region; instr: Instruction);

    (* operations on regions *)
    PROCEDURE OpenRegion (region, parent: Region);
    PROCEDURE CloseRegion (region: Region);
    PROCEDURE MergeRegions (region, tomerge: Region);
        (* combine two equivalent regions *)

    (* operations on operands *)
    PROCEDURE Operand (instr: Instruction; VAR x: Item);
        (* add the operand x, evaluating a variable to its last definition *)
    PROCEDURE OperandQ (instr: Instruction; VAR x: Item);
        (* add the operand x, do not evaluate variables to their last definition *)
    PROCEDURE DeleteOperand (opnd: Opnd);
    PROCEDURE ReplaceOperand (opnd: Opnd; VAR x: Item);

    (* miscellaneous *)
    PROCEDURE ReplaceUses (of, with: Usable);
    PROCEDURE GetConstant (VAR x: Item; val: LONGINT; type: Struct);
        (* find the Const node corresponding to val *)

END OOCD.
```

Many optimization algorithms both traverse and modify these lists at the same time, which can cause many problems. For example, when traversing a list of instructions and deleting some of them, it must be ensured that the instruction at the current position is not deleted before proceeding to the next. OOCD tries to catch such errors by resetting the links of deleted elements to NIL. However, all such problems cannot be caught at this level, and extreme care must be taken when simultaneously traversing and modifying a list.

7 Generating Guarded Single-Assignment Form

Guarded single-assignment form can be generated directly while parsing the source program if the program contains structured control flow only. For practical reasons, our compiler uses the portable Oberon-2 front-end OP2 [Crel91], and thus generates GSA form from the abstract syntax tree. However, this intermediate step is by no means required. This chapter describes the direct method.

It is assumed that the reader is familiar with single-pass parsing and code generation techniques as described in [Wirth86b] and [BCFT92]. In single-pass compilers, the parser directly calls code generation procedures when a source construct has been recognized. These code generation procedures write a machine code pattern matching the construct to a code array, which will finally be written to the object file. Only minimal amounts of context information are used to select the code pattern. The required information for operands, such as their type and their location, is passed in so-called *items*, which are records completely describing the operand.

Our optimizing compiler uses the same techniques to generate its intermediate data structure, but instead of emitting machine code to an array, it generates instructions into control regions. The selection of code patterns and the passing of attributes in items are similar. The differences are as follows.

- The region into which the instruction is to be placed is passed as a parameter to code-generation procedures. Since many different parts of the parser call the code generator, this information must also be passed between parser routines.
- Instead of branches, guards and merges are generated and used as regions into which instructions are to be placed.
- As required by GSA form, operands directly reference their defining result. Where necessary, gates are introduced. The following sections describe how this can be achieved.
- A separate graph is generated for each procedure.

In OOC2, module OOCC generates the data structure, relying on the operations provided by OOCD, as described in Chapter 6.

7.1 Translating Straight-Line Code

Generating intermediate code for expressions and other simple statements works like in single-pass compilers. Operands are evaluated and passed to operations as *items*, and instructions are generated and inserted into the current region.

The following listing shows how the parser recognizes simple expressions of the form

```
SimpleExpression = Term ( "+" | "–" ) Term .
```

and how the code generator routine *Add* creates the corresponding code pattern.

```
PROCEDURE SimpleExpression (region: Region; VAR x: Item);
   VAR y: Item; op: Symbol;
BEGIN
   Term(region, x);
   IF (sym = plus) OR (sym = minus) THEN op := sym; GetSym(sym)
   ELSE Error
   END ;
   Term(region, y);
   IF op = plus THEN Add(region, x, y)
   ELSIF op = ...
   ...
END SimpleExpression;

PROCEDURE Add (region: Region; VAR x, y: Item);     (* x := x + y *)
   VAR instr: Instruction;
BEGIN
   NEW(instr); OpenInstruction(instr);
   instr.op := add; Operand(instr, x); Operand(instr, y);
   Insert(region, instr);
   x.node := instr
END Add;
```

In order to get the program into single-assignment form, the compiler has to keep track of the most recent assignment to a variable. This information can conveniently be stored in the symbol table. Whenever a variable is assigned, this information is updated in the variable's object node in the symbol table. When a variable occurs as an operand, its most recent definition is fetched from the object node and used as the actual operand. A variable suffixed with its most recent definition is called the *current value* of the variable, or *value* for short. Figure 7.1 shows straight-line code, its single-assignment form, and the changes in the symbol table during compilation.

				a	*b*	*x*	*y*	*z*
(1)	a := 5	(1)	$a_1 := 5$	(1)	–	–	–	–
(2)	x := a + 1	(2)	$x_2 := (1) + 1$	(1)	–	(2)	–	–
(3)	y := a + 1	(3)	$y_3 := (1) + 1$	(1)	–	(2)	(3)	–
(4)	a := b ∗ 2	(4)	$a_4 := b_0 \ast 2$	(4)	–	(2)	(3)	–
(5)	z := a + 1	(5)	$z_5 := (4) + 1$	(4)	–	(2)	(3)	(5)

Figure 7.1: Straight-line code, its static single-assignment form, and changes in the symbol table during its translation.

7.2 Translating If-Statements

In GSA-form, a simple If-statement with no Elsif-clauses corresponds to two guarded regions for the Then- and Else-branch, and a merge-instruction combining the paths. After evaluating the condition, these regions can be generated. The statement-sequences of the Then- and Else-paths are placed into the regions corresponding to the condition being True and False, respectively. As described in Chapter 6, Elsif-clauses are translated into If-clauses in the Else-path.

```
PROCEDURE IfStatement (region: Region);
    VAR cond: Item; truePath, falsePath, merge: Region; nested: BOOLEAN;
BEGIN
    (* sym ∈ {if, elsif} *)
    nested := sym = elsif;
    GetSym(sym);
    Condition(region, cond);
    SplitPaths(cond, truePath, falsePath, merge);
    CheckSym(sym, then);
    StatementSequence(truePath);
    ResetIfMerge(merge);    (* see below *)
    IF sym = elsif THEN IfStatement(falsePath)
    ELSIF sym = else THEN GetSym(sym); StatementSequence(falsePath)
    END ;
    CommitIfMerge(merge);    (* see below *)
    IF ~nested THEN CheckSym(sym, end) END
END IfStatement;
```

Obtaining the single-assignment property for If-statements is based on the following observations: If the current value of variable v is v_k at the beginning of the If-statement, v_k will be the initial value on both paths through the If-statement, no matter what is assigned to v in the other paths. That is, the state of the symbol table is the same for the beginning of each path. Moreover, a rule concerning the placement of gates can be given.

Theorem: Let P and Q be the two paths through an If-statement, with corresponding merge node M. If P or Q contains an assignment to variable v, a gate for v will be needed at M.

Proof: Assume v_k is the current value of v before the If-statement, P contains an assignment to v leading to v_l, and Q contains no assignment to v. At M, there will be both v_k and v_l as current values of v, which have to be combined into one. On the other hand, if there is also an assignment to v in Q, a new current value v_m being different from v_k will reach M, thus also requiring a gate. The case where P contains no assignment to v while Q does is symmetric to the first case. ∎

Based on this theorem, a gate can be created as soon as an assignment to a variable is compiled in either path. If there already is a gate for the corresponding variable in the merge node, its operands are updated to reflect the most recent definition along that path. This functionality can be implemented in the code generation procedure for assignments, as shown below.

Since changes to the symbol table have to be undone after one path has been compiled, a place to store the old value is required. Instead of keeping a copy of the whole symbol table, the old value is stored in the gates of modified variables. Note that there is a gate for every modified variable. The following listing shows a simplified implementation of the method.

```
PROCEDURE Assign (region: Region; obj: Object; VAR x: Item);
    (* assign obj := x *)
    VAR gate: Gate;
BEGIN
    (* find the gate corresponding to obj in region.merge *)
    gate := FindGate (region.merge, obj);
    IF gate = NIL THEN gate := CreateGate (region.merge, obj) END ;
    (* replace the gate operand corresponding to region with x *)
    ReplaceGateOperand (gate, region, x);
    obj.current := x.node
END Assign;
```

```
PROCEDURE CreateGate (merge: Region; obj: Object): Gate;
    VAR gate: Gate;
BEGIN
    NEW(gate); OpenGate(gate);
    gate.op := gat; Operand(gate, merge);
    (* initial operands *)
    Operand(gate, obj.current); Operand(gate, obj.current);
    gate.obj := obj;    (* record which variable this gate corresponds to *)
    gate.oldValue := obj.current;    (* save the old value of obj *)
    Insert(merge, gate);
    RETURN gate
END CreateGate;
```

After the Then-path has been compiled, the symbol table has to be reset to the old values. This can be performed by traversing all gates in the merge node, as in procedure *ResetIfMerge.*

```
PROCEDURE ResetIfMerge (merge: Region);
BEGIN
    FOR all uses of merge in gates G DO G.obj.current := G.oldValue END
END ResetIfMerge;
```

When both paths have been compiled, the gates represent the new current values of assigned variables. By assigning the gates to the corresponding variables, gates for the same variable at outer merge nodes are correctly inserted or updated. This is the task of procedure *CommitIfMerge.*

```
PROCEDURE CommitIfMerge (merge: Region);
BEGIN
    FOR all uses of merge in gates G DO Assign(merge.region, G.obj, G) END
END CommitIfMerge;
```

Note that at the topmost level of the procedure, no gates can be inserted for assignments as there is no merge point with other paths. Instead, assignments to non-local objects are used in the *return*-node of the procedure, which summarizes all non-local effects.

Translating Case-statements would follow the same pattern. After each variant, the symbol table would have to be reset, and at the end, all gates would have to be committed. One important difference is that the arity of the merge instructions and gates is not known in advance, as the number of paths through the Case-statement is not known either. A method to increase the arity every time a new variant is compiled has to be implemented to deal with this.

7.3 Translating While- and Repeat-Statements

The translation of While-statements works very similarly to the translation of If-statements. At the beginning, a loop merge instruction is generated, into which the loop-controlling condition is evaluated. After splitting the paths, the true-path is made the second parameter of the loop merge, and all instructions generated for the loop body are inserted into it. The false-path makes up the exit path.

When variables are assigned in the loop, corresponding gates are generated in the loop merge. However, there are two important differences to the way gates are treated in If-statements.

- The loop merge combines the path leading into the loop and the backedge of the loop. Since all assignments in the loop are on the path to the backedge, and not on multiple different paths, it is never required to reset the symbol table.
- Not the gates in the loop header correspond to the most recent definition after the loop, but the last definition before the exit. This information can be stored in the gates.

- At the time the gate for variable *v* is generated, there may already be accesses to *v* within the loop. Initially, the referenced value is the last one defined outside of the loop, but now it has to be the gate for *v*. These uses have to be modified accordingly (see Figure 7.2).

```
WHILE i < 100 DO i := i + 1 END

(1) greg:                                   (1) greg:
     (2) l-merge:   (1), (4)                     (2) l-merge:   (1), (4)
                                                      (7) i := gate   (2), i, (6)
          (3) cmp    i, 100                           (3) cmp    (7), 100
          (4) if-less:   (3)                          (4) if-less:   (3)
               (6) add    i, 1                             (6) i := add    (7), 1
          (5) if-gteq:   (3)                          (5) if-gteq:   (3)
```

Figure 7.2: An Oberon While-statement and its corresponding GSA form; to the left before the assignment to i *has been compiled, to the right after the assignment. Changes are marked in bold.*

At the time the exit-path is split off, procedure *ExitLoop* saves the current value for all variables assigned so far. They will be made the current values again after the loop. For gates that are inserted later, the gate itself represents the current value after the loop. Assume a gate for variable *v* would be inserted after the exit path has been split off. In this case, there has been no previous assignment to *v* in the loop, in particular no assignment between the loop merge and the exit path. Thus, the gate for *v* corresponds to the last assignment to *v* before the exit, and represents the current value of *v* after the loop.

```
PROCEDURE ExitLoop (lmerge: Region);
BEGIN
   FOR all uses of lmerge in gates G DO G.exitdef := G.obj.current END
END ExitLoop;
```

The renaming of accesses to a variable within a loop after a gate has been inserted can be accomplished by procedure *FixupLoopGate*, which is called by *Assign*. Let the assigned variable be *v*, its value before the loop v_k, and the newly assigned value v_m. When the gate is created in the loop merge, its operands will be the merge, v_k and v_m, and its result will be v_n. All uses of v_k within the loop – that is, uses in regions that are dominated by the loop merge – will be replaced by uses of v_n, except for the use in the gate itself.

```
PROCEDURE FixupLoopGate (lmerge: Region; gate: Gate);
BEGIN
   old := second operand of gate;
   new := gate;
   FOR all uses U of old DO
      IF (U.instr # gate) & Dominates(lmerge, U.instr.region) THEN
         ReplaceOperand(U, new)
      END
   END
END FixupLoopGate;
```

Note that this procedure requires copy assignments to be left in the code. That is, if some variable *i* is assigned to *j*, a copy instruction has to be generated, instead of just letting both objects reference the same computation. Consider the following piece of code.

```
i := ...;
j := i;
WHILE cond DO ... i+j ...; i := i+1 END
```

If *i* and *j* would refer to the same computation *c* before the loop, the assignment to *i* in the loop would cause all uses of *c* to be renamed to the gate, which would erroneously rename the original use of *j*. For this reason, the use-chains of different variables have to be kept separate, namely by introducing a copy instruction which is referenced by the *j* object. These copy instructions will later be removed by an optimization step called *copy propagation*.

At the end of the loop, the values stored in the gates as exit values are copied into the corresponding objects as current values.

The translation of Repeat-statement works exactly the same and uses the same procedures. The only difference is that the exit path is determined at a different time.

7.4 Dealing with Unstructured Control-Flow

In Oberon, there are two forms of unstructured control-flow: Exit- and Return-statements. Since they correspond to a branch to the end of the loop or the end of the procedure, respectively, they cannot be directly expressed in GSA form. However, there is a simple method to rewrite such code into structured form automatically in the compiler, which we will outline in the following section. For the sake of simplicity, we discuss Exit-statements only, but exactly the same methods can be used to deal with Return-statements.

The basic idea is to introduce a boolean variable *$exit* for each Loop-statement, which indicates whether an exit from the loop has been performed. Obviously, *$exit* is initialized to False. Exit-statements are translated into assignments of True to this variable. All subsequent statements after an Exit will then be guarded by *$exit* being False, that is, they will only be executed if no Exit-statement has been executed. If a loop backedge is encountered, it also becomes guarded by this variable, which prepends *$exit* to the conditions controlling execution of the loop. Figure 7.3 shows an example for such a transformation in source form, with changes marked in boldface.

The translation includes some overhead in form of repeated checks of the *$exit* variable. However, much of this overhead can be eliminated by optimization algorithms. We have seen in Chapter 5 that repeated uses of the same condition will be found and the branching structure can be optimized. This optimization does not only help to improve the above code, but also makes it useless to write unstructured code with the intention of improving performance. The corresponding structured code will be optimized so that it executes as fast as the unstructured code.

```
                                      $exit := FALSE;
LOOP                                  REPEAT
  stat_0;                               stat_0;
  REPEAT                                REPEAT
    IF cond_0 THEN                        IF cond_0 THEN
      EXIT                                  $exit := TRUE
    END ;                                 END ;
    stat_1                                IF ~$exit THEN stat_1 END
  UNTIL cond_1;                         UNTIL $exit OR cond_1;
  stat_2                                IF ~$exit THEN stat_2 END
END                                   UNTIL $exit
```

Figure 7.3: A Loop-statement and its structured translation.

The translation can easily be added to the parser. Upon encountering a Loop-statement, a new *$exit* variable is generated and initialized to False. Exit-statements are translated into assignments to *$exit*. When a statement sequence is compiled, it is determined for each statement whether it is an Exit-statement or includes one. If it does, the remaining statements in the sequence are guarded by the *$exit* variable.

Note that the property of containing an Exit-statement propagates through the parser procedures as an *exited*-attribute. A simple statement sets *exited* if it was an Exit-statement, and a statement sequence sets the attribute if there was at least one statement that exited. If-statements exited if any path through them exited, as do Case-, While- and Repeat-statements. A Loop-statement never exited, as you cannot use Exit-statements to terminate multiple nested Loops. In all kinds of loops, *$exit* is prepended to the loop-controlling condition when their statement body exited.

Return-statements differ in a couple of small issues. There is only one *$return* variable for each procedure. If the procedure compiled is a function procedure, an additional variable is required to keep the function result. The *returned*-attribute is also propagated through Loop-statements up to the outermost region. Apart from this, the translation works the same.

The benefits of not having to deal with unstructured control-flow in the intermediate representation have also been noted in [HDEGSS92]. In [ErHe93] corresponding methods to automatically rewrite unstructured code into structured form have been described. Their approach differs from ours in that it can deal with arbitrary control-flow due to goto-statements, but it is inherently multi-pass and needs some representation that can express unstructured control-flow.

7.5 Alias Analysis

Aliasing effects are modeled by inserting *MayAlias* instructions into the code. When an assignment to a designator A is compiled, such instructions for all other designators within the current scope which may alias to A have to be generated. In the worst case, the number of MayAlias nodes is quadratic in the number of designators accessed. Thus, the cost to generate them is quadratic in the size of the program, assuming that the number of designators in a program is a linear function of its size.

Given such large numbers of nodes, and the negative impact they have on the optimization result, it is of utmost importance to find designators which cannot be aliases. The strong type system of Oberon and a couple of other properties of the language allow us to come up with a set of rules, telling when two designators *A* and *B* cannot be aliases.

- *A* and *B* can only reference the same memory cell when *A* or *B* is a VAR-parameter, or both *A* and *B* include a pointer dereferenciation.
- *A* and *B* must be *alias-compatible* in their types. Aliases are created by passing a designator to a VAR-parameter, or by assigning a pointer to another. For both, the type compatibility rules of Oberon must hold, and can be used in alias analysis.
- A VAR-parameter can only reference objects outside of its own scope – where we consider the heap being a scope outside of the global scope – and a pointer can only reference objects in the heap. Thus, a VAR-parameter does not alias to anything in its own or nested scopes, and only VAR-parameters or other pointers can alias to objects referenced through pointers. In particular, local variables of the currently compiled procedure are free of aliasing effects.

The notion of *alias-compatibility* is different from the one of type-compatibility. We say that two designators are alias-compatible iff there is a way how the designators could be made to reference the same memory location without violating type rules. However, due to the complex interaction between type extensions and parameter passing rules, a precise definition of when designators are alias-compatible is hard to give. For our purposes of avoiding superfluous MayAlias nodes, a pessimistic set of rules describing when two designators cannot be aliases is sufficient. Two designators *A* and *B* are not alias-compatible if any of the following points holds.

- *A* and *B* have different scalar types.
- *A* and *B* have structured types, and the type of *A* is not an extension of *B*'s type, and vice versa.
- *A* is a VAR-parameter of pointer type and *B* is not of the same type.
- *A* is a VAR-parameter and *B* is of a record type which does not include a field of *A*'s type.

This is only a small subset of all conditions under which two designators are not alias-compatible. It is a matter of future research to replace it by a proven notion of alias-compatibility.

Note that in languages such as C where pointers are addresses that may point anywhere, including at objects on the stack and in global storage, no matter whether their types are compatible or not, and in which pointers can even be made to point into objects, none of the above rules hold. The compiler has to start with the pessimistic assumption that objects accessed through pointers alias with everything, and then prove that the address of some variables is never taken, which prevents them from aliasing with pointers. In Fortran, it is not possible to determine at compile-time whether two variables in global storage are aliases, as the overlaying due to EQUIVALENCE statements happens at link-time.

In OOC2 alias analysis is performed as follows. Whenever an assignment to a non-local object *A* is compiled, all previously accessed non-local designators *B* are scanned and checked against the above rules. If it could not be determined that *A* and *B* cannot alias, a MayAlias node is generated and assigned to *B*. Note that this assignment to *B* does not trigger another scan for potentially aliased variables.

8 Optimizations

In this chapter, optimization algorithms on guarded single-assignment form as implemented in OOC2 will be discussed. After looking at some general properties of such algorithms, several optimization algorithms will be outlined.

8.1 General Properties

We have already seen in Chapter 2 that optimization algorithms are based on transformation rules. After certain properties of a piece of code have been determined, transformations are applied to it. This is also reflected in the structure of such algorithms. They include an *analysis* phase and a *transformation* phase.

Analysis Algorithms

The analysis phase finds attributes of computations or regions. An attribute describes some property of a value, e.g. the value being constant, or the value being available at a certain point. In Chapter 5, we have discussed the propagation of the availability-attribute in common subexpression elimination through programs in multi-assignment form. A collection of attributes – one for each computation in the considered program – is iteratively propagated through all paths of the program. This is a classical data-flow analysis algorithm as described by Allen and Cocke [AlCo76]. Figure 8.1 shows its general form.

A single pass of this algorithm over the program may not be able to establish all desired properties. When there are loops, there are also cyclic dependencies between computations. Whenever an attribute of an instruction in such a cycle changes, the whole cycle must be reconsidered.

```
PROCEDURE IterativeDataFlowAnalysis (P: Program);
  VAR v: AttributeVector;
BEGIN
  init attribute vector v;
  REPEAT
    propagate v over P
  UNTIL no more changes in v
END IterativeDataFlowAnalysis;
```

Figure 8.1: Iterative data-flow analysis.

Iterative data-flow algorithms can also be applied to programs in static single-assignment form. Iterative methods have several shortcomings, which can be avoided in SSA form. First, the attribute of every computation is propagated throughout the whole program, even though it may only affect a single use. If the analysis is performed until a fixpoint is reached, a single change in the attributes will result in another iteration of the analysis phase over the whole program. This makes the analysis unnecessarily inefficient. Second, attribute vectors often have to be stored for each basic block or region so that the results on different paths can be combined. These vectors require large amounts of memory, and the combination of attributes consumes a lot of run-time.

```
PROCEDURE SparseDataFlowAnalysis (P: SSAprogram);
  VAR worklist: LIST OF Instruction; instr: Instruction;
BEGIN
  init attributes of all instructions;
  add all instructions to worklist;
  WHILE worklist not empty DO
    fetch instruction instr from worklist;
    compute attributes of instr;
    IF attributes of instr were changed THEN
      add uses of instr to worklist
    END
  END
END SparseDataFlowAnalysis;
```

Figure 8.2: Sparse data-flow analysis.

Better algorithms for static single-assignment form are based on the following observation: When the attribute of some computation *C* changes, it is known that it can only directly affect the attributes in instructions that use *C*. Instead of reiterating over the whole program after an attribute change in *C*, only instructions which use *C* have to be reconsidered. This leads to the worklist-based algorithm in Figure 8.2.

Note that there is no longer an attribute vector, but one attribute associated with each computation. This makes use of the property of SSA-form, that a value and its attributes are independent of the position in the program. Thus, sparse data-flow analysis can only be applied to programs in SSA-form. The algorithm is called *sparse* because it only reconsiders small fractions of the program after an attribute change. It offers significantly better performance than iterative algorithms.

Pessimistic vs. Optimistic Algorithms

Assume some property *P* is the precondition for performing a transformation. It is desirable to find as many instructions as possible for which *P* holds, iterating over the instructions until a fixpoint is reached. Since programs can contain cyclic dependencies, the attributes of an instruction must be initialized properly before initiating the propagation.

Before discussing how to initialize the attributes, we would like to point out that the attribute values must change monotonically. That is, there must be an order-function on the attribute values, and it must be assured that changes are only made in one direction. As an example, consider the problem of constant propagation: The attribute values can be *constant* or *non-constant*. We only allow the attribute to change from *constant* to *non-constant*, but not in the other direction. Alternatively, we could allow it to change from *non-constant* to *constant* but not in the other direction. Without such a restriction, the attribute of an instruction could toggle between both values, so that a fixpoint would never be found and the above data-flow analysis algorithms would not terminate.

Obviously, the initialization value must be a value at one end of the attribute value ordering, and the direction of change must lead to the other end. If we initialize with the desired property, we call the algorithm *optimistic*. In constant propagation, this would correspond to setting all attributes to *constant* in the beginning, optimistically assuming everything to be constant. The propagation would then find instructions for which the optimistic assumption does not hold and change their attributes to *non-constant*. On the other hand, if we start with the assumption that the desired property does not hold – initializing to *non-constant* in constant propagation – we call the algorithm *pessimistic*.

Optimistic and pessimistic algorithms do not compute the same fixpoint in the presence of cyclic dependencies, as the following example shows.

```
(2) l-merge:  (1), (6)
          (3) i := gate   (2), 0, (5)
          (4) add   (3), 1
          (5) add   (4), –1
          (6) cond
          (7) if-true:  (6)
```

In this program fragment, 1 and –1 are added to *i*, so that *i* is not changed in the loop. If constant propagation starts with the pessimistic assumption that everything is non-constant, there is no way to find any of the instructions in the cycle (3)-(4)-(5) to be constant. On the other hand, if it starts with the assumption of everything being constant, it will be determined that (4) computes the constant 1, and (5) corresponds to the constant 0. The gate (3) then combines the constant 0 twice and thus is constant as well.

Note that the artificial nature of this example is due to its shortness and due to the simplicity of constant propagation. We will later meet more realistic examples with similar properties.

Pessimistic algorithms have the advantage that they start with a conservative and safe assumption. If they are not run until the fixpoint is reached, or if they miss some cases, the worst that can happen is that the property allowing the transformation is not determined in places where it would hold, thus leaving the code untransformed. Optimistic analysis procedures must always be run until a fixpoint is found and must handle all cases correctly. Otherwise, invalid transformations will be applied to the program.

8.2 Copy Propagation

The code generation phase of the compiler introduced copy-operations for assignments of constants or variables to other variables. Due to the single-assignment property, the assigned value cannot be overwritten, thus it can be used directly instead of the copy. *Copy propagation* replaces all uses of copy-instructions by uses of the respective operands on the right-hand side. This will not only remove some superfluous copy-instructions, but it will also improve the effectiveness of other optimization algorithms. In the following example, copy propagation enables a common subexpression elimination algorithm to derive that instructions (1) and (3) compute the same value.

```
x := i + 1; j := i; y := j + 1

(1) x := addi   i, 1
(2) j := copy   i
(3) y := addi   (2), 1
```

After copy propagation has been performed, the code looks as follows. Note that the copy instruction is still present but not used anymore. It will later be removed by dead code elimination.

```
(1) x := addi   i, 1
(2) j := copy   i
(3) y := addi   i, 1
```

Copy propagation is a simple algorithm that traverses all instructions, looking for copy instructions, and replacing their uses.

```
PROCEDURE CopyPropagation (P: Region);
   VAR R: Region; instr: Instruction;
BEGIN
   FOR all regions R in P DO
      FOR all instructions instr in R DO
         IF instr.op = copy THEN ReplaceUses(instr, instr.opnd) END
      END
   END
END CopyPropagation;
```

In OOC2, this algorithm is run once after GSA form has been generated. Due to the renaming of values after the insertion of gates in loops, copy instructions cannot be avoided altogether and hence this pass is required to cleanup the code. However, no new copy instructions will be introduced by later passes.

In contrast to this, optimizers based on multi-assignment intermediate representations often require copy assignments to temporary variables in order to avoid computed results to be overwritten. Consider the following example.

```
a := x+1;
....
a := ...
b := x+1
```

The result of computation *x+1* must be either recomputed for the assignment to *b*, or be assigned to a temporary variable *t* before the second assignment to *a* overwrites it, thus introducing a copy assignment between *a* and *t*. In such optimizers, copy propagation plays a crucial role in obtaining good code quality and is run several times. Moreover, the precondition of the transformation is much more complex in a multi-assignment framework, as all possibilities of overwriting a variable must be considered.

8.3 Procedure Inlining

Procedure inlining is the optimization of replacing the call to a procedure by the body of the procedure. This does not only avoid the cost of calling and returning from the procedure, but also allows one to optimize the inlined body together with the code of the original caller. Parameters do not have to be moved to a special parameter passing area, but can be accessed directly. To other optimization algorithms, the inlined operations are no different from the original operations in the procedure. For example, this allows finding common subexpressions between pieces of code that originally were in different procedures, or computing the result of operations on constant parameters at compile-time. It is mostly this customization of the inlined code to the call site that makes inlining profitable.

In contrast to common expectations, inlining can also increase run-time, as it tends to increase code size. For very small procedures, e.g. functions like MIN or MAX, inlined code is not larger than the code for passing of parameters. In such cases, inlining is always profitable. For larger procedures, no simple rule can be given, as the benefit depends on how much the inlined code will be improved, and how much the code size increase affects the run-time. Many compilers use size heuristics to decide which procedures to inline, e.g. IBM XLC [IBM90d] inlines procedures having less than 100 instructions in the intermediate code. We have decided to leave the decision to the programmer and to let him mark procedures to be inlined with a plus "+" sign. Size heuristics could easily be added, namely by marking small procedures as inline-procedures after their intermediate code has been generated.

In order to get the benefits of customizing a procedure body without actually inlining the procedure, *procedure cloning* has been implemented in some compilers [IBM94b]. Procedure cloning is an optimization which generates multiple versions of a procedure, each adapted to a certain subset of call sites. The IBM XLC compiler assumes that procedures being called with constant parameters would benefit from cloning, in particular when these parameters are used to guide the control flow inside the procedure. For such procedures, copies are generated with the parameter being replaced by the appropriate constant. The call sites including this constant parameter are modified to call the copy without that parameter. Other possible heuristics to select procedures for cloning could be derived from alias analysis, creating different versions of procedures which are sometimes called with aliased parameters and sometimes without, or from type analysis, when procedures are called with objects of different dynamic type. The ideas behind procedure cloning are similar to the ones of *partial evaluation*, which is a field of ongoing research [Sura93][Jones93][SeSo88].

In our compiler, inlining is performed while the caller is compiled. When a call to an inline procedure is generated, a deep copy of the procedure's body is inserted at the current position, parameters are connected to the uses in the copied body, and assignments to non-local variables modified by the inlined procedure are generated. Deep copying is implemented as a simple recursive algorithm. Since the copied data structure may include cycles, copying has to keep track of which objects have been copied already, so that the appropriate connections can be made. The *info*-field is used for this purpose, and initially is set to NIL. This field contains the reference to use in copies, or NIL if this reference still has to be determined.

```
PROCEDURE Inline (region, proc: Region);
BEGIN
    init info-fields of instructions in proc to NIL;
    copy := DeepCopy(proc);
    connect copy to region
END Inline;
```

Connecting the copy to the call site consists of putting the copy into the region of the call site, setting the operands of the enter-instruction to the passed parameters, and assigning the results of the return-instruction to the corresponding result variables.

In the following example, a function MIN returning the smaller of its parameters is to be inlined.

```
PROCEDURE+ MIN (a, b: INTEGER): INTEGER;
BEGIN
    IF a < b THEN RETURN a ELSE RETURN b END
END MIN;

...
x := MIN(x+1, 100);
... x * 2
```

Applying the above procedure to this example, the following intermediate code will be generated. Copied instructions are marked with an asterisk. Note that the numbering of instructions is not copied but generated in the context of the callee.

```
PROCEDURE MIN:
(1)  greg:
          (2)  a, b := enter    a, b
          (3)  cmp    (2), (2:1)
          (4)  if-less:   (3)
          (5)  if-gteq:   (3)
          (6)  i-merge:   (4), (5)
          (7)  $ret := gate    (6), (2), (2:1)
          (8)  return    (7)
```

```
PROCEDURE Caller:
(1)  greg:
          ...
          (20)  add    x, 1
          *(21)  greg:
                    *(22)  a, b := enter    (20), 100
                    *(23)  cmp    (22), (22:1)
                    *(24)  if-less:    (23)
                    *(25)  if-gteq:    (23)
                    *(26)  i-merge:    (24), (25)
                    *(27)  $ret := gate    (26), (22), (22:1)
                    *(28)  x := return    (27)
          (29)  mul    (28), 2
```

The inlined enter- and return-instructions are only used as copy assignments like copy-instructions. By applying copy propagation, they can be removed. Furthermore, instead of placing the instructions into an inlined global region, they could be put directly into the enclosing region, and making the copied region superfluous as well. As an improvement to the outlined algorithm, one can avoid copying the enter- and return-instructions and the global region node right from the beginning, and connect them directly by initializing the *info*-field appropriately.

```
PROCEDURE Inline (region, proc: Region);
   VAR res: Result; par: Node;
BEGIN
   init info-fields of instructions in proc to NIL;
   proc.info := region;     (* use region as 'copy' of proc *)
   FOR all results res of the enter-instruction
      and the corresponding parameters par DO
      res.info := par
   END;
   FOR all results res of the return-instruction
      and the corresponding parameters par DO
      Assign(res.defobj, DeepCopy(par))
   END
END Inline;
```

As opposed to the first version, not the top-level region of the procedure is taken as the root to be copied, but rather the operands of the return-instruction. Since the return-instruction summarizes all visible effects of the procedure, all required operations will still be copied. Dead code, however, will not be traversed and thus not be copied. The above example then looks as follows.

```
PROCEDURE Caller:
(1)  greg:
          ...
          (20)  add    x, 1
          *(23)  cmp    (20), 100
          *(24)  if-less:    (23)
          *(25)  if-gteq:    (23)
          *(26)  i-merge:    (24), (25)
          *(27)  x := gate    (26), (20), 100
          (29)  mul    (27), 2
```

Space for local variables of the inlined procedure is allocated in the stack frame of the caller, together with the space for its own local variables. In the case of open arrays passed as value parameters to the inlined routine, this would require to dynamically expand the stack frame. Due to the complexity involved in this, OOC2 does not allow to inline routines with value parameters of open array type.

The other kinds of procedures which cannot be easily inlined are recursive and exported ones. Inlining recursive routines would lead to infinite recursion in the inlining algorithm, unless the situation would be detected and the inlining process be cut off at a certain depth of inlined calls. Exported procedures may be inlined in the same module, but inlining them within another module would require to export the code to client modules, and to make them dependent on the actual implementation of the module. In OOC2, both recursive and exported procedures are not permitted as inline-procedures.

All differences between value- and VAR-parameters are modelled in the enter- and return-instructions of the inlined routine (see Sections 6.5 and 6.6). Inlining code does not change the most recent definition of variables in the caller, except if the they were passed as VAR-parameters or accessed as intermediate-level variables and if they were modified in the callee. In the latter case, the return-node includes a result for that variable, which is assigned and therefore updates the most recent definition of the variable.

8.4 Constant Propagation and Unreachable Code Elimination

In this section we describe an algorithm jointly finding constant computations and *unreachable* code. Code is unreachable if control will never pass to it, which in GSA form corresponds to a controlling guard evaluating to the constant FALSE. The algorithm presented here is Wegman and Zadeck's *Sparse Conditional Constant Propagation* [WeZa91], adapted to GSA form.

At first, this optimization looks superfluous, as the programmer should not introduce constant computations into the code and write unreachable code. However, if procedures are inlined, it can offer big benefits. Constant parameters are often encountered, and after inlining, instructions or guards using such constant parameters can be optimized with this algorithm.

The method is based on the sparse data flow analysis algorithm described before. It assigns a *lattice element* to each value in the program. As depicted in Figure 8.3, the lattice element can be of three types: The highest element is *top* $\top$, the lowest one is *bottom* $\perp$, and all elements in the middle are *constant*, ζ.

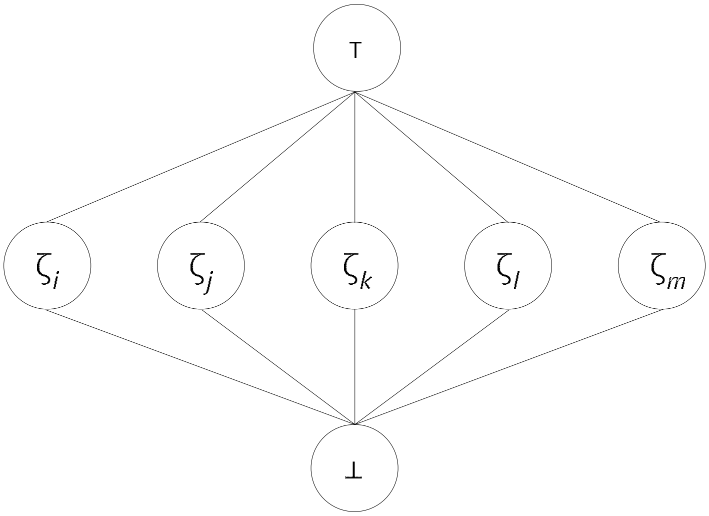


Figure 8.3: Lattice elements of constant propagation.

There is an infinite number of ζ_i lattice elements, each corresponding to a different constant *i*. There is no ordering among the constants, and there is no way how a computation attributed with element ζ_i will ever be attributed with a different ζ_j.

In the optimistic constant propagation algorithm of OOC2, the top element $\top$ corresponds to a computation being unclassified, and the bottom element $\perp$ to the value being non-constant. For regions, the meaning of the attributes is slightly different. A guard evaluating to FALSE controls unreachable code which does not have to be considered and which can be deleted later. Code guarded by TRUE will always be executed and can be moved to the region enclosing the guard. Merges can take on the constant attributes FALSE if both merged regions are FALSE, and LEFT or RIGHT if one of these regions is attributed FALSE.

Initially, computations are attributed as follows: Parameters, i.e. the result nodes of the enter-instruction, non-local variables, and accesses to data structures are assigned the bottom element, assuming that they do not evaluate to constants. Also, the top-level region *greg* is set to the bottom element. Constants are attributed with the corresponding lattice element ζ_i, and everything else, including regions, is initialized with the top element. Then, the instructions in the top-level region *greg* are put into the worklist, and the propagation starts.

```
PROCEDURE ConstantPropagation (P: SSAprogram);
  VAR
    worklist: LIST OF Instruction;
    newlattices: ARRAY OF Lattice;
    instr: Instruction; res: Result;
BEGIN
  initialize lattice elements;
  initialize lattice values of instructions in greg and put them into worklist;
  WHILE worklist is not empty DO
    fetch instruction instr from worklist;
    NewLatticeElements(instr, newlattices);
    FOR each result res of instr DO
      IF res.lattice # newlattices[res.no] THEN
        (* lattice element changed *)
        res.lattice := newlattices[res.no];
        add uses of res to worklist;
        IF (res IS Region) & (res.lattice # False) THEN
          initialize lattice values of instructions in region res
          and add them to worklist
        END
      END
    END
  END
END ConstantPropagation;
```

The algorithm does not add instructions to the worklist from regions which are either attributed FALSE or $\top$ for unclassified. In the latter case, the region can still be found to be FALSE, and the algorithm optimistically assumes so. Unreachable code thus does not affect the result of constant propagation in any way and does not consume compilation-time either.

In determining lattice values, the optimistic nature of the algorithm requires that values have to be considered constant as long as they have not been proven non-constant. In particular, yet unclassified values have to be taken as constants, of which the actual value is not yet known. This logic is built into the procedure computing the lattice elements of an instruction.

```
PROCEDURE NewLatticeElements (instr: Instruction; VAR lattices: ARRAY OF Lattice);
   VAR opnd: Operand; state: INTEGER; res: Result;
BEGIN
   IF instr IS Gate THEN LatticeOfGate(instr(Gate), lattices[0])
   ELSIF instr IS Merge THEN LatticeOfMerge(instr(Merge), lattice[0])
   ELSE
      state := $\zeta$;
      FOR each operand opnd of instr and WHILE state # $\bot$ DO
         IF opnd.lattice = $\top$ THEN state := $\top$
         ELSIF opnd.lattice = $\bot$ THEN state := $\bot$
         END
      END ;
      IF state = $\zeta$ THEN
         ComputeConstantResults (instr, lattices)
      ELSIF state = $\bot$ THEN
         FOR all results res of instr DO lattices[res.no] := $\bot$ END
      END
   END
END NewLatticeElements;
```

Gates and merges must be treated specially, as will be discussed below. An operand being attributed *bottom* $\bot$ makes the whole computation *bottom* $\bot$, and an operand being yet *top* $\top$ makes it *top* $\top$. If all operands are known constants, the procedure *ComputeConstantResults* is used to determine the constant values of the results. This procedure is an interpreter for the operations found in the intermediate code, and the only machine-dependent part of the constant propagation algorithm.

Merges receive an attribute value depending on which paths to it are executable. Figure 8.4 lists the function to compute it.

In computing the value of gates, only the operands corresponding to executable paths have to be considered. The merge operation contains the necessary information to select the operands. If both paths are executable, i.e. the lattice value of the merge is *bottom* $\bot$, and both value operands of the gate are not *top* $\top$, then the result of the gate is only constant if both value operands correspond to the same constant.

```
PROCEDURE LatticeOfGate (gate: Gate; VAR lattice: Lattice);
   VAR merge: Merge; src1, src2: Usable;
BEGIN
   merge := first operand of gate;
   src1, src2 := second and third operands of gate;
   IF (merge.lattice = $\top$) OR (merge.lattice = False) THEN    (* do nothing *)
   ELSIF merge.lattice = Left THEN lattice := src1.lattice
   ELSIF merge.lattice = Right THEN lattice := src2.lattice
   ELSIF (src1.lattice is $\zeta$) & (src1.lattice = src2.lattice) THEN
      lattice := src1.lattice    (* constant *)
   ELSE lattice := $\bot$
   END
END LatticeOfGate;
```

	⊤	False	*others*
⊤	⊤	⊤	Left
False	⊤	False	Left
others	Right	Right	⊥

Figure 8.4: Lattice element assignment for merges. Lattice values for the first operand are listed horizontally, those for the second operand vertically.

OOC2 performs constant propagation on integer operations only. Floating-point arithmetic is sensitive to the rounding mode set at run time, which is not known at compile time. The lattice elements are represented by extensions of type *Info*, containing a field for the lattice type and a field for the actual constant value if the type is constant.

The asymptotic complexity of the algorithm is O(*N*), where *N* is the number of instructions in the program. An instruction is added to the worklist when one of its operands lowers its lattice. Since every value can at most be lowered twice, every instruction can be added to the worklist at most *2*c* times, where *c* is the number of operands the instruction includes. Thus the upper bound on how often an instruction is visited is constant.

When the worklist has become empty, a fixpoint has been reached. A single pass over all instructions is sufficient to make the corresponding replacements.

```
PROCEDURE ReplaceByConstants (region: Region);
  VAR instr: Instruction; region0: Region; res: Result;
BEGIN
  FOR each instruction instr in region DO
    IF instr IS Region THEN
      IF instr.lattice = False THEN DeleteRegion(instr(Region))
      ELSIF instr.lattice = True THEN
        MoveInstructions(instr(Region), region)
      ELSIF instr.lattice = Left THEN
        region0 := first operand of merge instr;
        MoveInstructions(instr(Region), region0)
      ELSIF instr.lattice = Right THEN
        region0 := second operand of merge instr;
        MoveInstructions(instr(Region), region0)
      ELSE ReplaceByConstants(instr(Region))
      END
    ELSE
      FOR all results res of instr DO
        IF res.lattice is ζ THEN ReplaceUses(res, res.lattice) END
      END
    END
  END
END ReplaceByConstants;
```

This algorithm simultaneously traverses and modifies the intermediate representation, and great care must be taken in the implementation to assure its correctness. For example, if a region has the assigned lattice value TRUE, the enclosed instructions are moved into the currently traversed region. It must be made sure that the moved instructions are traversed as well.

8.5 Value Numbering

Value numbering is a technique to find instructions in a program computing the same value, allowing to remove some of these instructions. It is the most prominent technique to eliminate common subexpressions. Symbolic values are associated with computations in such a way that two computations receive the same value only if they are known to always compute the same result. The algorithms presented here build upon the notions of congruence and availability, as discussed in Chapter 2. There is a pessimistic and an optimistic value numbering technique, the latter of which has been implemented in OOC2.

As we have seen in Chapter 2, two instructions yield *congruent* results, iff their opcodes are the same and all their corresponding operands are congruent. Constant operands are congruent, iff they are identical. The *pessimistic value numbering* algorithm at first assumes that no two computations are congruent. It then computes *value numbers* for all instructions such that two instructions receive the same number if they are found to be congruent. Algorithms for this were first described by Cocke and Schwartz [CoSch70] and later adapted to SSA form [AlZa94].

A naive implementation of the algorithm would traverse all instructions, comparing operators and operands with the ones in all other instructions. If the instructions are traversed in a topologically sorted order, i.e. an instruction precedes all its uses, it will be known whether the operands of the instructions are congruent. A single pass over the program will be sufficient to reach a fixpoint in $O(N^2)$ time, where N is the number of instructions in the program. This ignores loop gates, however, for which the definition of the last operand has not yet been visited and which have to be considered as non-congruent with other values. If these effects have to be dealt with too, 2^k iterations will be required to reach a fixpoint in the worst case, where k is the deepest nesting level of loops.

A better implementation of the algorithm is based on hashing and has an expected running time of $O(N)$ for one iteration. The value number of some computation is determined using a hash function h over the operator and the value numbers of the operands. For a computation *f(A,B)*, the hash value is computed as *h(f,h(A),h(B))*. Hashing collisions can be resolved in any common way. The algorithm looks as follows.

```
PROCEDURE PessimisticValueNumbering (P: SSAprogram);
   VAR instr: Instruction;
BEGIN
   FOR all instructions instr in P DO
      initialize instr.valNum to a unique number outside of the hashing range
   END ;
   FOR all instructions instr in P DO instr.valNum := Hash(instr) END
END PessimisticValueNumbering;

PROCEDURE Hash (instr: Instruction): ValueNumber;
BEGIN
   IF instr.valNum is not in the hashing range THEN    (* not yet computed *)
      instr.valNum := $h$(instr.op, Hash(instr.$operand_0$), Hash(instr.$operand_1$), ...)
   END ;
   RETURN instr.valNum
END Hash;
```

Hash collisions are detected by comparing the operands of the function h, i.e. the opcodes and the value numbers of the operands. If they are the same, the instructions are congruent, otherwise there is a collision.

Loop gates have to be treated separately, as computing their hash value from the hash value of the last operand may lead to infinite recursion. The solution chosen here is to initialize all operations to a unique number in the beginning, and to use this number for loop gates instead of the hash value. The uniqueness of the number reflects the pessimistic initial assumption that no instructions are congruent.

After having found the congruent instructions, some instructions can be eliminated. If for a computation *A*, there is a congruent computation *B* that is available at *A*, all uses of *A* can be renamed to *B* and *A* can be deleted. In SSA form, *B* is available at *A* if *B* dominates *A*. The replacement phase can be implemented in different ways, e.g. by a top-down traversal of the dominator tree, in which a stack of lists keeps track of available instructions.

```
PROCEDURE Replace (R: Region; list: List);
  VAR instr, instr0: Instruction; R0: Region;
BEGIN
  FOR each instruction instr in R DO
    IF there is a congruent instruction instr0 in list THEN
      ReplaceUses(instr, instr0);
      Delete(instr)
    ELSE
      AddToList(list, instr)
    END
  END ;
  FOR each region R0 in R DO Replace(R0, list) END
END Replace;
```

Every instance of *Replace* has its own list, which is a copy of the one of its enclosing region, and to which it adds the encountered instructions. When the instance for region *R* returns, the old state is reset, making all instructions unavailable that have been added during traversal of *R*.

It is possible to combine the steps of determining congruence and replacing instructions into one pass, if instructions are topologically sorted within the regions, i.e. an instruction is traversed before all its uses (with the exception of loop gates, as before). In this case, the order of traversal in the replacement step is also a valid order for the traversal in the analysis step.

The *optimistic value numbering* algorithm has first been described in [AlWZ88] and [AlZa94]. It is only applicable to SSA form, while the pessimistic algorithm can also be made to work with programs in multi-assignment intermediate representations. Finite-state machine minimization [Hopcr71] is the basis for the optimistic method. It finds more values to be congruent than the pessimistic algorithm.

Optimistic value numbering assumes, until evidence to the contrary is discovered, that all computations that plausibly could be congruent are indeed congruent. Instructions considered to be congruent are placed in the same partition. Evidence that two computations are not congruent is that they have different operators, or that they have corresponding operands in different partitions.

If the *i*-th operand of an instruction in partition *P* belongs to partition *Q*, all instructions in *P* should have their *i*-th operand in *Q*. If for some instructions, the *i*-th operand is in a different partition than *Q*, these instructions should be taken out of *P*. We say that *Q* *splits* partition *P* into two subpartitions, namely $P \backslash Q$ containing the instructions with the *i*-th operand belonging to *Q*, and P/Q including the instructions with the *i*-th operand not being in *Q*.

Optimistic value numbering is based on splitting, which is repeated until a fixpoint is reached, i.e. until no further splitting is possible. Initially, all instructions with the same opcode are assumed to be congruent and placed in the same partition. Then, the algorithm uses the partitions to split other partitions, until no more changes are encountered. Splitting with partition *Q* can be implemented by traversing all uses of instructions in *Q*, moving the encountered instructions of partitions A_j with *i*-th operand being in *Q* into a new partition $(A_j \backslash Q)_i$. The original partition A_j is then equivalent to $(A_j/Q)_i$. Empty partitions can be discarded.

```
PROCEDURE OptimisticValueNumbering (P: SSAprogram);
   VAR
      Q: ARRAY OF Partition;
      worklist: LIST OF Partition;
      op: INTEGER; instr: Instruction; A: Partition;
BEGIN
   worklist := NIL;
   FOR each possible opcode op DO
      create a partition Q[op];
      add Q[op] to worklist
   END ;
   FOR all instructions instr in P DO
      place instr into partition Q[instr.op]
   END ;
   WHILE worklist # NIL DO
      fetch partition A from worklist;
      split by A;
      add new non-empty partitions to worklist, discard empty partitions
   END
END OptimisticValueNumbering;
```

This algorithm has a run-time complexity of $O(N^2)$, where N is the number of instructions in the program. The time to split by a partition is proportional to the number of traversed uses, which is proportional to N. When each instruction ends up in its own partition, the algorithm splits by N partitions, yielding a total time of $O(N^2)$. This can be reduced to $O(N \log N)$ by the following observation: Assume a partition P has already been used to split other partitions A, and is now split into P_0 and P_1. Neither P_0 nor P_1 will split A/P, as the i-th operand of instructions in A/P is not in P. Let $B = A \backslash P$, i.e. the i-th operand of all instructions in B is in P. This implies that the i-th operand of all instructions in B is either in P_0 or in P_1. Thus, $B \backslash P_0 = B/P_1$ and $B/P_0 = B \backslash P_1$. It is therefore sufficient to split by either P_0 or P_1 instead of both, and it makes sense to select the one with fewer uses. When choosing the smaller subpartition, a partition with N elements can at most be split $\log N$ times. Thus, at most $\log N$ partitions will be added to the worklist after the initial constant number of partitions, reducing the time complexity to $O(N \log N)$.

The optimistic algorithm finds congruences that the pessimistic misses, as in the following example.

```
i := 0; j := 0; s := 0;
WHILE i < 100 DO
   ... stat$_0$ ...;
   s := s+a[i];
   i := i+1; j := j+1
END ;
i := 0; t := 0;
WHILE i < 100 DO
   ... stat$_1$ ...;
   t := t+a[i];
   i := i+1
END
```

The optimistic algorithm would find i and j to be congruent in the first loop, and it would also find that the second loop is executed under the same conditions as the first one, and that s and t are congruent. Knowledge that the loops are executed under the same control conditions could be used for an optimization called *loop fusion*, which combines multiple loops into one, saving the loop control overhead. Such optimizations can be useful after inlining has been performed. The above example would then be reduced to the following.

```
i := 0; s := 0;
WHILE i < 100 DO
    ... stat_0 ...; ... stat_1 ...;
    s := s+a[i];
    i := i+1
END
```

The pessimistic algorithm would not discover these congruences, as the loop gates for *i* and *j* would be considered as different, thus the increments would not be found as congruent either. Due to the latter, the gates remain different, no matter how many iterations of pessimistic value numbering over the program are performed.

8.6 Dead Code Elimination

Dead code elimination is the optimization of deleting code that does not affect the final result. As for many optimization problems, there are pessimistic and optimistic algorithms for it, which we will both present.

According to our definition of semantic equivalence, we require that a program generates the same output state and generates the same exceptions after it has been optimized. From the view of a complete program, the final result is the state that it leaves in global storage and the operations it has performed on output-devices. If no interprocedural analysis is performed, one has to assume that every assignment to non-local variables – including all indirect assignments – could contribute to the final result. In our framework of intraprocedural analysis, we thus define the final result of a procedure as the combination of all non-local assignments and the exceptions raised. Note that the non-local assignments have been collected in the return-instruction of the procedure.

The pessimistic dead code elimination algorithm assumes all instructions to be alive, and then searches for instructions that can be proven to be dead. Instructions that have no uses, that cannot raise exceptions, and that do not assign non-local variables are dead, and thus can be deleted.

```
PROCEDURE DeadCodeElimination (region: Region);
    VAR instr: Instruction;
BEGIN
    FOR all instructions instr in region DO
        IF instr IS Region THEN
            DeadCodeElimination(instr(Region));
            IF instr(Region) contains no instructions THEN
                Delete(instr)
            END
        ELSIF instr has no uses & instr cannot raise exceptions
                & instr does not assign a non-local object THEN
            Delete(instr)
        END
    END
END DeadCodeElimination;
```

Since deleted instructions may be the only uses of other instructions, their removal can unveil other dead code. The algorithm should thus be iterated, for best results until a fixpoint is reached. Traversing the instructions in a topologically sorted order, so that uses of values are visited before their definitions reduces the number of iterations to reach a fixpoint. The number of iterations required corresponds to the longest chain of dead instructions being used by other dead instructions, and can be quite large.

This algorithm corresponds closely to a *reference-counting garbage collector*, where instructions having *zero* uses are considered being unreferenced and thus can be deleted.

The *optimistic dead code elimination* algorithm assumes all instructions to be dead, except for the ones that contribute to the final result. The return-node summarizes all non-local assignments and thus can be taken as the placeholder for the final result, and be marked as live. All instructions that transitively provide operands to the return-node also contribute to the result, and therefore are alive as well. They can be found by a recursive *MarkLive* procedure.

```
PROCEDURE MarkLive (instr: Instruction);
   VAR opnd: Operand;
BEGIN
   IF ~(live IN instr.attrib) THEN
      INCL(instr.attrib, live); MarkLive(instr.region);
      FOR all operands opnd of instr DO
         IF opnd.def IS Result THEN MarkLive(opnd.def(Result).instr) END
      END
   END
END MarkLive;
```

Instructions that raise exceptions have to be marked alive as well by a separate pass. It would be possible to represent the exception behavior of an instruction like an assignment to a global *exception* variable, and thus anchoring its effect in the return-node as well. We have not done so in OOC2 and leave the issue to future work.

Calling *MarkLive* to mark the return-instruction and instructions raising exceptions will mark all instructions contributing to the final result as alive. Everything not being marked is dead and can be deleted. Note that this algorithm corresponds to a *mark-and-sweep garbage collector*, where the return-instruction and exception-raising instructions provide the alive roots.

```
PROCEDURE DeadCodeElimination (P: GSAprogram);
   VAR instr: Instruction;
BEGIN
   initialize all instructions as dead;
   MarkLive(return-instruction);
   FOR all exception-raising instructions instr DO MarkLive(instr) END ;
   delete all instructions not marked live
END DeadCodeElimination;
```

Since both algorithms closely resemble garbage collection algorithms, we would assume that their individual strengths are similar to the ones of the garbage collectors. This is in fact the case. The pessimistic algorithm does not need to know all roots, but needs the effort to keep track of how often an instruction is used, and cannot delete cyclic structures. For example, the empty counting loop

```
i := 0;
WHILE i < N DO INC(i) END
```

would not be found as dead code by the pessimistic algorithm, as the value computed by the increment instruction is used in a gate for *i* at the top of the loop, the value of which is then used in the comparison and increment instructions. The optimistic algorithm would be able to remove the whole loop. Moreover, if a non-local variable *A* is assigned multiple times in a procedure, only the last assignment to *A* will be marked as alive by the optimistic method, while the pessimistic one will treat all assignments as live.

In our framework, the root for optimistic dead code elimination is readily available in form of the return-instruction summarizing the final result. OOC2 implements the optimistic approach. While programmers should not write dead code, such code is often generated as a by-product of other optimizations. When some constructs are replaced by others, the original code may become dead. Instead of keeping track of what is still referenced in all optimization algorithms and deleting instructions directly, it is easier to perform a dead code elimination step later.

8.7 Strength Reduction, Reassociation, and Loop Invariant Code Motion

Strength reduction is an optimization that replaces an expensive operation by a less-expensive one, usually a multiplication by an addition within a loop. *Reassociation* is a related transformation exploiting commutative, associative, and distributive laws to rewrite expressions, so that strength reduction becomes more profitable. *Loop invariant code motion* moves computations that are constant over all iterations of the loop to a place before the loop, and is performed as a by-product of the other transformations. The algorithms discussed here are those described by Markstein and Zadeck [MMZ94], adapted to GSA form.

Code amenable to strength reduction typically stems from array addressing within loops, as in the following example of matrix multiplication.

```
FOR i := 0 TO N–1 DO
  FOR j := 0 TO N–1 DO t := 0;
    FOR k := 0 TO N–1 DO t := t + a[i, k] * b[k, j] END ;
    c[i, j] := t
  END
END
```

The address of *a[i, k]* in the innermost loop is computed as follows.

```
ADR(a[i, k]) = ADR(a) + ((i * LEN(a)) + k) * SIZE(ElementType)
```

In the next iteration, array element *a[i, k+1]* will be accessed, the address of which is computed similarly.

```
ADR(a[i, k+1]) = ADR(a) + ((i * LEN(a)) + (k + 1)) * SIZE(ElementType)
```

By using the distributive and associative laws, this computation can be rewritten to a form reusing subexpressions from the computation of *a[i, k]*.

```
ADR(a[i, k+1]) = ADR(a) + ((i * LEN(a)) + k) * SIZE(ElementType) + 1 * SIZE(ElementType)
ADR(a[i, k+1]) = ADR(a[i, k]) + SIZE(ElementType)
```

The address of the element accessed in iteration *i+1* can thus be computed by a simple addition if the address from iteration *i* is known, avoiding 2 multiplications and 2 additions. More technically, we can think of a pointer being set to *a* before the loop, which is incremented to point to the next element in each iteration, instead of computing the element address from the index values over and over. The C programming language allows performing this optimization at the source level, which we consider disadvantageous from a programmer's point of view. Namely, readability suffers a lot, accesses cannot be checked against array bounds, and many more aliasing problems are introduced. We will now discuss how the compiler can generate the more efficient addressing code while still using the array notation at the source level.

Induction Variables

Strength reduction is centered around the notion of *induction variables*, which are variables that are changed in a predictable way when proceeding from one loop iteration to the next. The definitions used in compilers vary slightly, but all allow such variables to be incremented by a constant offset in each iteration. Our definition is more general.

Definition: An *induction variable V* is a variable that is computed by linear operations only, namely:

- V := I
- V := W + I
- V := W – I
- V := ϕ(merge, X, Y)

where *W* is an induction variable, *I* is a constant or a loop invariant, and *X* and *Y* are constants, loop invariants, or induction variables.

The goal of strength reduction as implemented in OOC2 is to rewrite expressions of the form $V*I+J$, where *V* is an induction variable, and *I* and *J* are constants or loop invariants. As a first step, induction variables and loop invariants have to be determined.

The loop-counting variables in For-statements are induction variables by definition in many programming languages, and thus are found easily. However, if the language explicitly allows to change the variable directly or through side-effects, the inductive nature is lost. The definition given in the Oberon-2 language report [MöWi91] specifies an equivalent While-statement allowing the programmer to modify the induction variable. It is not clear, whether the compiler is allowed to consider the counting variable to be inductive.

Classifying Computations

The same sparse data flow analysis algorithm used in constant propagation can also be used to classify computations in a loop into loop invariants, induction variables, and everything else. Instead of applying it to a whole procedure at once, it is run on single loops only, from innermost to outermost. The lattice elements are *loop-invariant LI*, which takes the role of *top* ⊤, *forward-inductive FI*, and *bottom* ⊥. Initially, all computations in the loop are marked as *LI*. The functions listed in Figure 8.5 are used to compute new lattice values during the propagation.

ϕ	LI	FI	⊥
LI	FI	FI	⊥
FI	FI	FI	⊥
⊥	⊥	⊥	⊥

+/–	LI	FI	⊥
LI	LI	FI	⊥
FI	FI	FI	⊥
⊥	⊥	⊥	⊥

others	LI	FI	⊥
LI	LI	⊥	⊥
FI	⊥	⊥	⊥
⊥	⊥	⊥	⊥

Figure 8.5: Combination functions for ϕ, +/– and others in determining loop invariants and inductive variables.

Loop-invariant computations can then be moved out of the loop, if their execution is not guarded by conditions within the loop. Otherwise, moving them out of the loop would also mean to move them out of their guarding region, and they may be executed in the transformed program even though they were not executed in the original program. In While-loops, the whole body is guarded by the loop condition, setting a barrier to move code out of such loops. This is one of the reasons why OOC2 transforms While-statements into Repeat-loops as follows.

```
                                   IF cond THEN
WHILE cond DO                        REPEAT
  stat                                 stat
END                                  UNTIL ~cond
                                   END
```

Alternatively, if the loop-invariant computations do not have side-effects, they could be *speculatively* moved out of their controlling guard, under the assumption that they are likely to be executed in the original program as well. The topic of such speculative code motions will again be considered in Chapter 9 on instruction scheduling. Cytron et al. [CyLZ86] describe an algorithm to move nested control structures with loop-invariant guards out of loops, which we have not considered for OOC2.

For the purpose of strength reduction, we not only need knowledge about inductive expressions, but also about *inductive cycles*. Computations on an inductive cycle deliver results to be used in subsequent iterations. Consider the following example.

```
(2) l-merge:   (1), (8)
     (3)  i := gate    (2), i, (4)
     (4)  add    (3), 1
     (5)  add    (3), 20
     (6)  sub    (5), 9
     (7)  cmp    (4), 100
     (8)  if-less:   (7)
```

In this loop, (3), (4), (5), and (6) are all inductive, but only (3) and (4) are on an inductive cycle. The others represent just offsets from elements of the cycle. The increment from one iteration to the next can be determined from the cycle. Since all cycles must include a gate at the top of the loop, the gates are a natural point to start the search for such cycles. A simple recursive algorithm finds inductive cycles and marks instructions belonging to it appropriately.

```
PROCEDURE FindInductiveCycles (loop: Region);
  VAR g: Gate; IC: IndCycleDesc; opnd: Operand; cyclic: BOOLEAN;
BEGIN
  FOR all loop-gates g in loop DO
    IF g is forward-inductive THEN
      create a new inductive cycle descriptor IC;
      opnd := last operand of g, corresponding to loop-backedge;
      cyclic := FALSE;
      SearchCycle (opnd.def, g, IC, cyclic);
      IF cyclic THEN g.info := IC END
    END
  END
END FindInductiveCycles;

PROCEDURE SearchCycle (def, header: Instruction; IC: IndCycleDesc; VAR cyclic: BOOLEAN);
  VAR opnd: Operand;
BEGIN
  IF def = header THEN
    cyclic := TRUE
  ELSIF def is forward-inductive THEN
    FOR all operands opnd of def DO
      SearchCycle (opnd.def, header, IC, cyclic)
    END ;
    IF cyclic THEN def.info := IC END
  END
END SearchCycle;
```

Each cycle receives its own descriptor which uniquely identifies it and to which all instructions on the cycle refer.

Reassociation

In the next step, expressions within the loop are symbolically rewritten to sums of products, using the commutative, associative, and distributive laws. More precisely, every computation is transformed into the form $(V*A + B + C + \mathit{others})$, where V is the header of an inductive cycle, i.e. a gate, A and B are loop-invariant, C is constant, and *others* is everything else. The expression is thus separated into a group of terms: A product of an induction variable with a loop-invariant, which can be reduced in strength; a term including all loop-invariants, which can be moved out of the loop; a constant term, which can be computed at compile time and likely be added for free as a displacement in a memory access; and a term including everything else, which cannot be optimized.

In the implementation, a descriptor for the four terms is associated with every computation. Procedure *Flatten* is called for all instructions to perform the actual rewriting.

```
PROCEDURE Flatten (instr: Instruction);
  VAR opnd1, opnd2: Operand;
BEGIN
  IF instr is not yet flattened THEN
    IF instr is the header of an inductive cycle THEN
      instr.sum := (instr*1, 0, 0, 0)
    ELSIF instr.op = "+" THEN
      opnd1, opnd2 := first and second operand of instr;
      Flatten(opnd1.def); Flatten(opnd2.def);
      instr.sum := opnd1.def.sum + opnd2.def.sum
      (* sum terms individually *)
      (*
        instr.sum.const := opnd1.def.const + opnd2.def.const;
        instr.sum.LI := opnd1.def.LI + opnd2.def.LI;
        ...
      *)
    ELSIF instr.op = "–" THEN
      opnd1, opnd2 := first and second operand of instr;
      Flatten(opnd1.def); Flatten(opnd2.def);
      instr.sum := opnd1.def.sum + (–1)*opnd2.def.sum
    ELSIF instr.op = "*" THEN
      opnd1, opnd2 := first and second operand of instr;
      Flatten(opnd1.def); Flatten(opnd2.def);
      instr.sum := opnd1.def.sum * opnd2.def.sum
      (* use distributive law *)
      (*
        instr.sum.const := opnd1.def.sum.const * opnd2.def.sum.const;
        instr.sum.LI := opnd1.def.sum.const * opnd2.def.sum.LI
          + opnd1.def.sum.LI * opnd2.def.sum.const
          + opnd1.def.sum.LI * opnd2.def.sum.LI;
        ...
      *)
    ELSE
      instr.sum := (0, 0, 0, instr)
    END
  END
END Flatten;
```

Note that the additions and multiplications in procedure *Flatten* are not performed at compile time, but correspond to operations performed at run time. In the compiler, they are inserted into expression trees combining the expression trees of their operands. All information required to perform strength reduction is then available in classified expression trees.

Strength Reduction

The algorithm traverses all instructions again, and generates new induction variables where useful. For an expression of the form $(V*A + B + C + others)$ with the above meanings for the variables, it performs the following steps.

- V corresponds to a loop gate $\phi(merge, X, Y)$. Introduce a new loop gate $V' = \phi(merge, X', Y')$, with $X' = X*A + B + C$ being computed outside of the loop. X' corresponds to the initial value of the expression $V*A + B + C$.
- Traverse the instructions V_n on the inductive cycle of V. For every instruction, add corresponding instructions that update V' to V'_n, as listed in Figure 8.6.
- Replace all occurrences of $V*A + B + C$ by V'.

The algorithm also has to keep track of which new inductive cycles have been generated already, so that for two occurrences of the same expression, only one inductive cycle is generated. The computations added as described in Figure 8.6 may include loop-invariant terms, which should be placed outside of the loop.

$V_n = V_m + d$	$V'_n = V'_m + d*A$
$V_n = V_m - d$	$V'_n = V'_m - d*A$
$V_n = d$	$V'_n = d*A + B + C$
$V_n = \phi(merge, V_m, V_k)$	$V'_n = \phi(merge, V'_m, V'_k)$

Figure 8.6: Added inductive operations when creating inductive cycle V' being a replacement for V∗A + B + C.

While this description is complete, it does not make good use of the addressing modes in the PowerPC architecture yet. Load- and store-instructions in the PowerPC architecture specify a base register and either a 16-bit signed offset or another register, which is added to the base to obtain the effective address. In the case where the *others* term of the expression was 0 – a common case according to our experience – the free addition performed as part of an address computation is not exploited. Moreover, there may be cases where two inductive expressions differ only in the constant term, and introducing new inductive cycles for each of them is superfluous, when the difference between the constant parts can be incorporated into the offset part of a memory access. The strength reduction algorithm in OOC2 also considers these cases, and adds the constant or loop-invariant terms only to the new inductive cycle if their addition cannot be performed as part of an address computation.

The PowerPC architecture also allows to perform an increment of a base register as a side effect of memory accesses with its load-and-update and store-and-update instructions. These instructions store the effective address computed into the base register, and thus can combine a memory access with the increment of the inductive cycle. In order to exploit this, the offset specified in one memory access must equal the increment of the inductive cycle. This can always be achieved by appropriately choosing the starting value of the inductive cycle, as shown in the following example.

```
(1) greg:
          (2) ldi    0
          (3) addr    a
          (4) l-merge:   (1), (9)
                    (5) gate    (4), (3), (7)
                    (6) stw    (5), 0, (2)
                    (7) add    (5), 4
                    (8) cond
                    (9) if-true:   (8)
```

After adapting offsets:

```
(1) greg:
          (2) ldi    0
          (3) addr    a
          (10) add   (3), –4
          (4) l-merge:   (1), (9)
                    (5) gate    (4), (10), (6:2)
                    (6) stwu    (5), 4, (2)                    ; store-and-update
                    (8) cond
                    (9) if-true:   (8)
```

Adding this feature to OOC2 is a matter of future work and should further improve performance on some loops.

After the uses of the original induction variables have been replaced by the pointer references, controlling the number of iterations of the loop is often their only purpose. In [MMZ94] an optimization called *linear test replacement* is described, which rewrites comparisons of old induction variables in terms of new ones. For example, assume a loop includes a guard of the form $V < X$, where V was an induction variable and X a loop-invariant. Uses of V in the form $V*A+B+C$ were rewritten to use a new inductive variable V', and the guard is the only remaining use of V. The guard may then be rewritten to $V' < X*A+B+C$, making V superfluous. However, we consider the transformation being unsafe. Without further knowledge about the possible ranges of V and X, the terms $V*A+B+C$ or $X*A+B+C$ may overflow, and the comparison may yield a different result than the original one.

We have developed another approach for OOC2, which makes use of the PowerPC *branch-and-count* instruction. Guards are reassociated to compare a value against zero, e.g. $V < X$ is rewritten to $X–V > 0$. This can be implemented as a simple addition to the *Flatten* procedure described above. The expression is also subject to strength reduction, and in our example, a new inductive variable V' representing $X–V$ is introduced, which is counted down to zero. This is precisely the semantics of the branch-and-count instruction, which decrements a value and compares it against zero.

If the value of the original inductive variable is used outside of the loop, it is still not possible to remove the associated computations. When the variable V counts up or down to a known limit L, an assignment $V := L$ right after the loop will make the inductive variable superfluous. If there are additional loop-controlling guards, so that the value after the loop cannot be precomputed, it may be possible to obtain the original value of V from some strength-reduced $V' = V*A+B+C$ by computing $V = ((V'–B–C)\ DIV\ A)$. However, as with linear test replacement, such a transformation is only valid if information about the range of V is available. Oberon has an advantage over languages such as C or Fortran in this respect, because index checks within loops allow to derive range information which is not available in languages without array index checking.

8.8 Peephole Optimizations

OOC2 makes use of the more complex instructions in the PowerPC architecture through peephole optimizations. The code is scanned for combinations of instructions that may be expressed with a single instruction, and appropriate replacements are made. Figure 8.7 presents some of these replacements.

```
(1) fmul    x, y
(2) fadd    (1), z                       (2) fmadd    x, y, z

(1) fabs    x
(2) fneg    (1)                          (2) fnabs    x

(1) rlwinm    x, s, a, b
(2) rlwinm    (1), t, c, d               (2) rlwinm    x, (s+t) MOD 32, e, f
                                         ; with {e..f} := {a..b} * ROT({c..d}, 31−t)

(1) muli    x, 2^n                       (1) rlwinm    x, n, 0, 31−n

(1) FXU-op    x, y                       (1) FXU-op.    x, y              ; Record-Option
(2) cmpi    (1), 0
```

Figure 8.7: Some transformation patterns for peephole optimization.

There are many more possible patterns, e.g. to exploit the floating-point multiply-and-subtract instructions, to use loads and stores with update, or to combine shifts and mask-operations in rotate-left-and-mask instructions.

When scanning the code, the algorithm searches for the last instruction in the pattern and then determines whether the operands match as well. For example, for a floating-point add, it is checked whether one of the operands is the result of a floating-point multiply. If it is, the addition is rewritten to a multiply-and-add. The multiplication will be left in the code, as there may be other uses of its result. If all uses could be consumed into a multiply-and-add instruction, the multiply instruction becomes dead code and will be removed later. Note that by allowing either operand of the addition to be computed in a multiply, we make use of the commutativity of addition. Similar arithmetic laws are used in other transformation patterns.

8.9 Other Possible Optimizations

The literature contains a large number of optimization algorithms, attacking very different types of programming languages, target architectures, and source programs. The compiler writer must select a set of algorithms that yield a benefit commensurable with the involved cost. In our view, the most important optimizations have been implemented in OOC2, but we would like to discuss several other algorithms which may be added in the future.

Redundancy Elimination and Partial Redundancy Elimination

Redundancy elimination is an optimization with similar goals as common subexpression elimination. It finds instructions which compute the same result and eliminates superfluous recomputations. Consider the example in Figure 8.8.

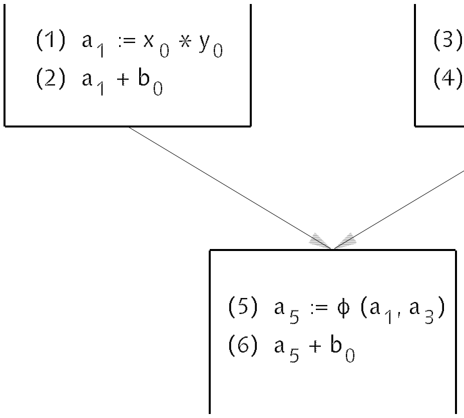


Figure 8.8: Redundant computations.

Instruction (6) computes the same result as (2) and (4), depending on which path was taken. Note that these instructions are not congruent, but include *redundancies*. By adding a gate combining the results of instructions (2) and (4), instruction (6) becomes superfluous. This optimization is called *redundancy elimination*.

Now assume that instruction (4) were not present. In this case, instruction (6) would be *partially redundant* to instruction (2), as on one path the value computed by instruction (6) is already available. By adding the computation to the other path, i.e. inserting instruction (4), instruction (6) becomes fully redundant and can be removed. The optimization of *eliminating partial redundancies* is always profitable, as it does not increase the number of instructions on any path executed and does not increase the code size either.

Classical redundancy elimination algorithms as the one by Morel and Renvoise [MoRe79] use lexical equivalence of computations to find redundancies, and thus are better suited to programs in multi-assignment intermediate representations. In [RoWZ88] an algorithm for SSA form is presented which can also detect redundancies between instructions that are not lexically equivalent, but which is rather complex. [BrCo94] describes an algorithm which uses reassociation to expose more redundant expressions and optimistic value numbering to detect the actual redundancies. We believe that this last algorithm would make for a nice addition to OOC2.

Elimination of Redundant Checks

Index checks can be removed when it is known that the checked value is within certain bounds. Information about the range of numbers that a value may take on can be derived from several sources: Induction variable analysis computes upper and lower bounds for induction variables, some operations like MOD return results within given bounds, and sometimes the range is restricted by the range of the type, e.g. a SHORTINT value is restricted to the range –128..127. By propagating such range information through the program, many index checks can be removed.

Our value numbering algorithm is capable of eliminating type tests or type guards for identical types. Corney and Gough [CoGo94] describes *type-flow analysis* propagating type information through the data-flow graph, which allows to eliminate redundant type checks even if the types are not identical but belong to the same type hierarchy. For example, if type T2 extends T1, and an object has already been determined to be of type T2, a later check for it being of type T1 can be omitted.

Lazy Evaluation and Partial Dead Code Elimination

A correct program requires that the definition of a value dominates all its uses, i.e. the value must have been computed before it is used, no matter which path is taken to this use. It is often encountered that a computation also dominates paths on which it is not used. By delaying the execution until its need is known (*lazy evaluation*), the execution time on some paths will be reduced. More technically, the computation should be moved to the least frequently executed region which still dominates all uses. Note that when a value is used inside a loop, this is not necessarily the closest dominator of all uses.

Partial dead code elimination [KnRS94] attacks the same problem with a slightly different terminology. A computation is considered partially dead if there are paths on which its result is not used. By moving instructions, partially dead code can be avoided.

Array Dependence Analysis

In OOC2, assignments to array elements are treated as assignments to the whole array, reducing the optimization potential. If two array references *a[i]* and *a[j]* are known to access different elements of the array, there are no dependencies between them. This allows them to be moved more freely and enables data-flow analysis to find more precise information. Consider the following example.

```
x := a[i];
a[j] := c;
y := a[i]
```

In this program fragment, *x* and *y* are equivalent if *j* refers to a different array element than *i*. If, on the other hand, *j* refers to the same element as *i*, *y* will be equivalent to *c*. Thus, knowing more about the relationship between *i* and *j* would be helpful, and the uncertainty prevents optimization of this fragment. *Array dependence analysis* tries to determine such information, in particular on induction variables of loops. A general solution requires to solve diophantine equations and is not feasible. However, practical methods to determine dependence or independence between array accesses for many common cases in numerical programs are known. Griesemer [Grie93] presents an overview of the topic, and Banerjee [Baner88] as well as Zima and Chapman [ZiCh90] discuss it thoroughly.

There are provisions in OOC2 to enable such analysis, namely the index values used to access an array are stored with the actual access. How the dependence and independence information could be represented within the GSA framework is a topic of future research.

Locality-Improving Transformations

Programs using large amounts of data sometimes encounter huge performance decreases due to cache misses. Consider the following program to multiply two matrices.

```
FOR i := 0 TO N–1 DO
   FOR j := 0 TO N–1 DO t := 0;
      FOR k := 0 TO N–1 DO t := t + a[i, k] * b[k, j] END ;
      c[i, j] := t
   END
END
```

Without loss of generality, we assume all matrices to be of size *N*N*. When computing the value of c_{ij}, the *i*-th row of *a* is multiplied by the *j*-th column of *b*. In order to compute the *i*-th row of *c*, the *i*-th row of *a* is multiplied with the whole matrix *b*, as shown in Figure 8.9.

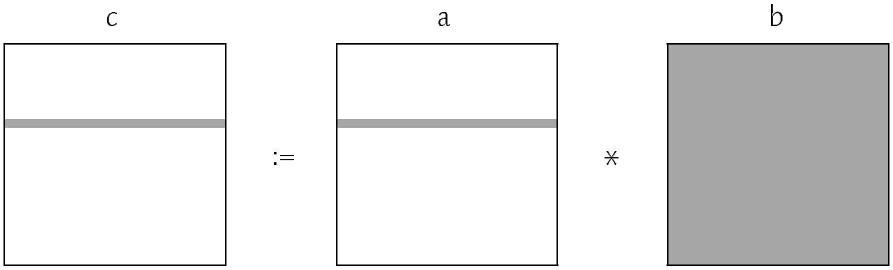
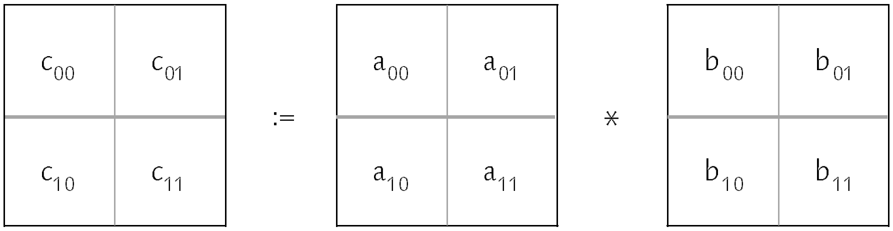


Figure 8.9: Data access patterns in matrix multiplication.

Assume that the code is executed on a machine with a 64 kBytes data cache with LRU-replacement, and none of the matrices are stored in the cache initially. If $N = 100$, and if the element type of the matrices is LONGREAL, one matrix will consume $N^2 * 8$ bytes = 80 kBytes. We will now compute how many array elements have to be fetched from main memory during matrix multiply.

In the computation of row c_0, row a_0 consisting of N elements and matrix b containing N^2 elements have to be read from main memory. Columns of b will be brought into the cache from left to right. After having fetched approximately 64 kBytes of b, the next column accessed will replace the leftmost column of b, which is the least-recently used one. At the end of the computation, the rightmost columns making up about 64 kBytes of data will be stored in the cache. In the computation of c_1, row a_1 and again the whole matrix b have to be accessed. When the first column of b is loaded for this, it replaces another column of b in the cache. Due to this, no column accessed will ever be found in the cache and the whole matrix has to be fetched from main memory again. As there are N columns of c to compute, the total number of array elements loaded from main memory during matrix multiply is $N * (N^2 + N) = N^3 + N^2$. Note the unfortunate property that no element brought into the cache is accessed more than once before being discarded again.

The number of main memory accesses can be dramatically reduced by a technique called *cache blocking*. Instead of multiplying a row of a by the whole matrix b, subblocks of a and b that fit into the cache are fetched, and as many operations on this data as possible are performed before replacing a subblock by another. The multiplication of large matrices can be partitioned into the multiplication of smaller matrices which are then combined, as shown in Figure 8.10.



$$c_{00} := a_{00} * b_{00} + a_{01} * b_{10}$$
$$c_{01} := a_{00} * b_{01} + a_{01} * b_{11}$$
$$c_{11} := a_{10} * b_{01} + a_{11} * b_{11}$$
$$c_{10} := a_{10} * b_{00} + a_{11} * b_{10}$$

Figure 8.10: Partitioning a large matrix multiplication into smaller ones.

One quarter of a matrix consumes 20 kBytes, so three such blocks can be kept in the cache at once. In the computation of c_{00}, four blocks are fetched into the cache, corresponding to N^2 elements. The blocks a_{00} and a_{01} can be reused in the computation of c_{01}, so only two blocks containing $N^2/2$ elements have to be loaded from main memory. Similar reuse is encountered in the other two computations, yielding a total number of $N^2 + 3 * N^2/2 = 5/2 * N^2$ elements to be accessed in main memory. Compared to the original code, the rewritten program needs approximately 40 times less main memory accesses.

The optimization does not only apply to matrix multiplication, but in general to programs dealing with large arrays and having regular access patterns, e.g. which traverse the array sequentially several times. In numerical applications, such patterns are often encountered. Array dependence analysis is a prerequisite for this optimization: The compiler must know about the order in which elements are accessed and it must determine, which accesses can be reordered.

According to Lam et al. [LaRW91], speedups by a factor of five are common when cache blocking numerical programs. Sarkar [Sark93] claims even higher factors around ten and predicts such transformations to become more important in the future. Wolf and Lam [WoLa91] describe algorithms to perform cache blocking in compilers and present some other cache-oriented optimizations.

9 Machine Code Generation

In this chapter we discuss *instruction scheduling* and *register allocation*. Instruction scheduling is the task of finding a valid execution order for the instructions of the program which makes good use of the processor pipelines. Register allocation assigns registers to operands and results, and inserts load- and store-operations where values cannot be kept in registers.

These two techniques are usually also considered to belong to the class of optimization algorithms. In our framework, however, they differ from other optimizations in that they cannot be omitted and in that they establish additional invariants of the intermediate representation: Instruction scheduling generates a valid execution order, whereas register allocation finds a possible register assignment.

9.1 Instruction Scheduling

Determining a valid execution order for instructions means topologically sorting the instructions according to their dependencies. We will discuss an algorithm that achieves this, and we will learn how it can be adapted to exploit the processor pipelines well.

Dependencies

We have introduced three kinds of instruction-dependencies in Chapter 3: data-, anti-, and output-dependencies. In GSA-form, a data-dependency between two instructions *A* and *B* corresponds to a result of *A* being used as an operand of *B*. Thus, *A* has to be executed before *B*. There are no anti- or output-dependencies between computations. There is one notable exception in which they are required, however, and have to be added before scheduling the code. Assignments to structured variables are fully renamed, i.e. each assignment generates a new instance of the variable. Depending on the order of accesses to these different instances, they may have to be allocated to different memory locations.

(1) a := upd a, adr_0, i, expr_0
(2) acc (1), adr_1, j
(3) a := upd (1), adr_2, k, expr_1

In our example, there is no dependency between instructions (2) and (3). Assume the scheduler would choose to execute instruction (3) before (2), and there would only be one copy of array *a* assigned by both (1) and (3). In this case, instruction (2) would access the wrong value if indices *j* and *k* were equal. Therefore, either (2) has to be executed before (3), or separate copies of the array have to be created.

Keeping separate copies of the array has the following disadvantages. First, update-instructions receive the semantics of an array-copy with one element changed, and thus have a high overhead. Second, the addressing code has to be adapted to the actual instance accessed, and to where it has been allocated. This is complicated if one attempts to avoid unnecessary copying. Third, the higher memory consumption of the program will also reduce cache performance. It is extremely rare that the benefit of having separate copies outweighs these disadvantages. In OOC2, we have decided to allocate only one memory location for the array and to introduce corresponding anti-dependencies. Note that the output-dependency between (1) and (3) is already modeled by the first parameter. By making (2) an operand of (3), the correct execution order is guaranteed. More general, in an update-instruction of the form

a_j := upd a_k, ...

all accesses to a_k on the path between the defining instructions of a_k and a_j become operands of the definition of a_j, so that they will be executed before.

All dependencies are then represented as definition-operand relationships. The instruction that computes a value has to be scheduled before all uses of that value. Consider the example in Figure 9.1.

```
(10) rliwnm   i, 2, 0, 29
(11) lwx   (2), (10)
(12) lwx   (3), (10)
(13) add   (11), (12)
(14) stwx   (5), (10), (13)
(15) i := add   i, 1
```

Figure 9.1: Example instructions in GSA form.

The dependencies can be visualized as a directed graph, with an arc between two instructions *A* and *B* if *A* computes a value that *B* uses (Figure 9.2).

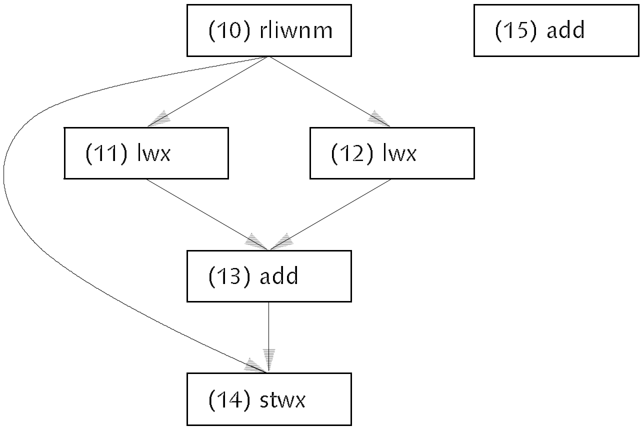


Figure 9.2: Dependence graph of the instructions in Figure 9.1.

List Scheduling

A scheduling algorithm traverses the instructions in the graph and emits them in an order that satisfies all dependencies, i.e. an instruction is only emitted after all instructions on which it is dependent have been emitted. We call an instruction *ready* if it is not dependent on other unscheduled instructions. Ready instructions are candidates for being emitted. Using this notion, a scheduling algorithm has the following form, as described by Hennessy and Gross [HeGr83].

```
PROCEDURE Schedule (g: Graph);
  VAR ready: LIST OF (*ready*) Instruction;
BEGIN
  determine ready in g;
  WHILE not everything emitted from g DO
    select instruction i from ready;
    emit i;
    update ready
  END
END Schedule;
```

Keeping track of ready instructions can be implemented by a field in each instruction counting the number of yet unavailable operands. When an instruction is emitted, the uses of all its results are traversed and the corresponding counters decremented. As soon as the counter of an instruction drops to zero, the instruction is ready.

The above algorithm will generate a valid ordering in any case. Finding an ordering that achieves high performance is a matter of implementing a good function selecting instructions from the set of ready instructions. Heuristics that immediately come to mind are as follows.

- Select instructions that do not cause interlocks, i.e. avoid instructions for which an operand or a resource is not available in the current cycle.
- Favor instructions on long dependence chains over ones on short chains. The longest dependence chain in the graph is a lower bound for the total execution time of the code, and thus should be scheduled with priority.
- Select instructions with many uses, thus making as many instructions ready as possible.
- Prefer instructions with many operands and few results, so that the number of registers will be reduced.

Similar heuristics have been described by Gibbons and Muchnick [GiMu86], and many others would be possible. We will now discuss how the first two heuristics could be implemented, as it has been done in OOC2.

With each result its latency is associated as an attribute. Each instruction stores the maximum length of the dependence chain from itself to the leaves of the dependence graph. These values can be determined in a depth-first traversal of the graph once at the beginning. Figure 9.3 shows the previous dependence graph with these attributes.

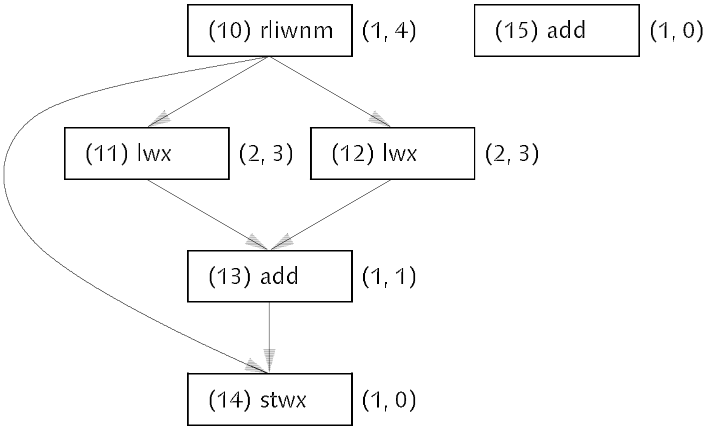


Figure 9.3: Dependence graph with (latency, dependence-length)-attributes.

For each instruction, the first cycle in which it could be executed without introducing interlocks is recorded. This is the maximum of issue-time plus latency for each instruction it depends upon. Note that this value is updated as the scheduling algorithm proceeds, selecting instructions for each issue-time.

The scheduler tries to emit an instruction for each execution unit in each cycle. Our selection function chooses an operation matching the execution unit, which can be issued in the current cycle, and which has the longest dependence chain. Note that the selection can be the empty set if there is no instruction to be executed in this cycle.

```
PROCEDURE Select (unit: Unit; cycle: INTEGER; VAR best: Instruction);
  VAR maxDep: INTEGER; instr: Instruction;
BEGIN
  maxDep := MIN(INTEGER); best := NIL;
  FOR each instruction instr in ready DO
    IF (instr executes on unit) & (instr.first <= cycle) THEN
      IF maxDep < instr.depLen THEN
        maxDep := instr.depLen; best := instr
      END
    END
  END
END Select;
```

After the selected instruction has been emitted, the ready set and dependent instructions have to be updated accordingly.

```
PROCEDURE UpdateReady (emitted: Instruction; cycle: INTEGER);
  VAR res: Result; u: Opnd; instr: Instruction; first: INTEGER;
BEGIN
  Remove(ready, emitted);
  FOR each result res of emitted DO
    first := cycle + res.latency;
    FOR each use u of res DO
      instr := u.instr;
      IF instr.first < first THEN instr.first := first END ;
      DEC(instr.depCount);
      IF instr.depCount = 0 THEN Add(ready, instr) END
    END
  END
END UpdateReady;
```

cycle	ready	emitted
0	$\{(10)_{(0,4)}, (15)_{(0,0)}\}$	(10)
1	$\{(15)_{(0,0)}, (11)_{(1,3)}, (12)_{(1,3)}\}$	(11)
2	$\{(15)_{(0,0)}, (12)_{(1,3)}\}$	(12)
3	$\{(15)_{(0,0)}, (13)_{(4,1)}\}$	(15)
4	$\{(13)_{(4,1)}\}$	(13)
5	$\{(14)_{(5,0)}\}$	(14)

Figure 9.4: Scheduling the graph of Figure 9.3. Suffixed numbers in parentheses correspond to the first and dependence-length attributes, respectively.

Figure 9.4 presents step by step how our algorithm would schedule the above piece of code for the PowerPC 601. Note that all instructions execute in the FXU, so this is an example of scheduling for a single pipeline.

As described, this algorithm can be used to schedule basic blocks or innermost non-cyclic regions (i.e. no loops). Schedulers operating on basic blocks only are called *local* schedulers. Reordering instructions across branches as performed in *global* schedulers is significantly more complex, since the control-conditions under which instructions are executed can change. We will return to this issue in the section on global scheduling algorithms below.

Note that the algorithm requires the dependence graph to be acyclic. Loops in GSA-form do not lead to acyclic definition-use relationships, however. Consider the following example.

```
(1) greg:
     (2) l-merge:  (1), (6)
          (3)  i := gate   (2), i, (4)
          (4)  i := add    (3), 1
          (5)  cmp    (4), 100
          (6)  if-less:  (5)
          (7)  if-gteq:  (5)
```

There is a cycle of dependencies between instructions (3) and (4). This cycle has to be broken somewhere, and since the schedule is generated for one iteration of the loop, the best place to do so is the back-edge of the loop. By ignoring the last parameter of loop-gates, a valid ordering would still be achieved. Note that every value being defined in one iteration and used in a later one goes through a loop-gate. In order not to schedule instructions with long latencies at the bottom of the loop when they are used at the top of the next iteration, their dependence-length could be set to their latency, even though they are leaves in the dependence graph. The scheduler will attempt to schedule them so that they terminate before the end of the current iteration.

Scheduling Nested Control-Structures

Nested control-structures are represented in GSA-form by regions participating in the instruction list. Assume that the instructions in the control-structure have already been scheduled and that the structure can be considered as one "large" instruction. Then, there are values – denoted by *in* – which are used inside the structure and defined outside, as well as values *out* being defined inside and used outside. This is similar to normal instructions, which use values and define new ones being used later. We thus say, the region uses the values *in* and defines the values *out*. Using this model, a control-structure can be scheduled just like other instructions. Note that this is already more aggressive than basic-block local scheduling, since it allows instructions to be moved over control-structures or even control-structures to be ordered in a different way than in the source. A technique achieving similar results at a much higher cost on control-flow graphs has been proposed by Bernstein and Rodeh [BeRo91].

As simple as it looks, there are several issues which have to be dealt with separately. An If-statement is not represented by a single region, but by guards and merges, which have to be scheduled all at once. The simplest solution is to make one of these regions the placeholder for all of them, i.e. it becomes ready when all regions belonging to the If-statement are ready. Since there is exactly one i-merge for each simple If-statement, it makes sense to make the i-merge this placeholder.

Another problem is that the timing over region boundaries is not well represented. First of all, an execution time for a control-structure can usually not be given. If it is an If-statement, it can be the execution time of either the Then- or the Else-path, or even larger numbers of different paths if there are c-merges or further nested structures involved. If, on the other hand, it is a loop, it depends on the number of iterations taken through the loop. For the purpose of scheduling, we always consider the shortest path in estimating the execution time of a control-structure. Assuming longer execution times would mask long latencies of instructions issued before the control-structure. For example, if a division with a latency of 20 cycles is issued before a control-structure with a minimum execution time of 3 cycles and a maximum one of 30 cycles, assuming the longest execution time would allow to schedule a use of the division result right after the control-structure. The program may often run through the short path and incur a 17 cycle interlock on the division result.

Conservatively, a control-structure is assumed to take all processor resources during its execution time, which in fact may not be the case. Instructions from the outer region could be scheduled into such free pipeline slots, but they would have to be replicated on all paths through the control-structure. We have not investigated this possibility further.

Finally, latencies for *in* and *out* values of a region are more complicated to obtain. Knowing that a value is used inside a control-structure or defined inside is not sufficient to determine its latency as seen in the outer region. For example, if a division with a latency of 20 cycles is scheduled as the last instruction in a region, the encountered latency for its result in the enclosing region is also 20 cycles. It must be known where within the control-structure the value is used or defined, on all possible paths through the structure.

Another idea we had but did not investigate further is to try to schedule nested regions at the very beginning or very end of their enclosing region. This would allow to optimize branching. For example, if an If-statement is the last statement in a loop, there is a branch from the end of the THEN-path to the end of the If-statement, where there is a branch to the loop header. The first branch can be redirected to the loop header, avoiding the second branch altogether. This is a topic for future research.

Forward vs. Backward Scheduling

The algorithm we have discussed so far generates the schedule forward, i.e. in the order in which the instructions will be executed. Unfortunately, the path-length heuristic tends to move instructions with long latencies too far apart from their uses, as it tries to make similar progress on all paths. In particular, load operations are usually placed at the top, ALU operations in the middle, and store instructions at the end, consuming large amounts of registers over large sections of the code. This effect is called *overscheduling*, as instructions and their uses are moved further apart than necessary. The problem is particularly pronounced in our framework, in which operations can be moved over large control structures and hence can increase register pressure significantly. Lam [Lam93] claims that by generating the schedule from the bottom, i.e. backwards, the danger of overscheduling can be reduced.

An intuitive explanation for this behavior is that in forward scheduling, instructions with long latencies are scheduled as early as *possible*, while in backward scheduling, they are scheduled as early as *needed* [Boll94]. For example, when scheduling forward, load instructions will be selected first due to their long latency, moving them to the top. In backward scheduling, a load instruction can be emitted as soon as the latency to all uses of the loaded value has been covered, which typically is much further down in the considered region.

We have neither found nor collected empirical data on this topic, but we have implemented both forward- and backward-schedulers for OOC2. The algorithm needs only minor adjustments, namely in that an instruction is ready when all its uses have been emitted, and in how the first-attribute is computed and interpreted.

Cycle vs. Operation Scheduling

The above algorithm performs so-called cycle scheduling, i.e. selects instructions for each cycle. Alternatively, one may also select cycles for operations as in operation scheduling. According to Lam [Lam93], operation scheduling offers the advantage that high-priority operations can be scheduled first. It requires that a *reservation table* is kept recording which processor resource is used in which cycle, and makes it harder to implement than cycle scheduling. Moreover, keeping track of which instruction can be issued in which cycles is more complicated. When instruction A is scheduled for cycle t, all instructions B_i computing operands for A must be issued to the latest in cycle $t{-}latency(B_i)$. This also places restrictions on operations computing operands for B_i which have to be tracked. Furthermore, an instruction which consumes A's result as an operand cannot be issued before $t{+}latency(A)$. The effort to keep track of all these restrictions is much higher than for a cycle-driven scheduler. Furthermore, for local schedulers, the concept of high-priority instructions is not well-defined. We do not believe operation scheduling to be superior in our framework, although we do not have empirical data on this topic.

Operation scheduling has benefits in scheduling loops, and we will return to this in the next section.

Direct Placement Scheduling

Griesemer [Grie92] describes a scheduling method for basic blocks that does not require the full dependence graph to be build. In this algorithm, instructions are immediately placed into a *reservation table* as they are generated. Assume that instruction *A* is just to be scheduled. All instructions B_i on which *A* is dependent have already been scheduled and their issue time $t(B_i)$ is known. The scheduler determines $max(t(B_i)+latency(B_i))$ of all B_i as the earliest possible issue time $t(A)$. Starting with this time, it scans forward in the reservation table for the next available pipeline slot and places *A* into it. After the whole basic block has been generated, it emits the reservation table in order, skipping over empty slots.

While the technique is simpler to integrate into a single-pass compiler, it would be harder to implement for GSA form. Note that the algorithm makes use of the order in which instructions are emitted by the code generator. This information is not available in GSA form, while the dependence graph is.

In terms of quality of the schedule, direct placement scheduling delivers inferior results, as it cannot use path-length heuristics and only considers the already generated schedule for its decisions.

9.2 Global Instruction Scheduling

Local scheduling methods suffer from a lack of overview, i.e. they do not determine latencies over region boundaries, and they are restricted by their inability to move instructions between regions. Usually, the number of instructions in a region is small, leaving only little freedom to rearrange code. Better schedules can be obtained by establishing a more global view and allowing to move instructions over region boundaries. Scheduling algorithms doing so are called *global*.

We distinguish between *acyclic* and *cyclic* global scheduling methods. Acyclic ones deal with control flow graphs which either include no cycles, or in which artificial scheduling barriers are introduced on loop back-edges. Thus, loop back-edges present no opportunity for optimization. In cyclic methods, instructions can also be moved across loop back-edges and the latencies along such edges are also considered for the schedule.

In the next section, we will discuss *trace scheduling* as the most prominent acyclic global scheduling algorithm, and *software pipelining* as a cyclic global scheduling method attacking innermost loops. Both methods deal with different kinds of code patterns and both have their place within an optimizing compiler.

9.3 Trace Scheduling

Trace scheduling was first proposed by Fisher [Fish81] as a method for global microcode compaction, and has later been adapted to VLIW machines for the Bulldog compiler by Ellis [Ellis85]. The central idea of the method is the concept of a *trace*, an acyclic path through the program, which might be taken during program execution. Instead of scheduling every basic block on this trace separately, the trace as a whole is scheduled at once as if it were a single basic block. By doing so, instructions are implicitly moved over branches and merges. If at execution time, the program takes this path, the results will be equivalent to those generated by the original program, but the code has a better schedule. However, if a different path is taken, the results would differ, and modifications have to be made to these *off-trace* paths in order to ensure semantic equivalence. Figure 9.5 shows a control-flow graph and traces on it shaded in grey.

Traces are selected and scheduled in decreasing likelihood of execution, i.e. the trace with the highest execution probability is selected and scheduled first. Then, among the remaining basic blocks in the

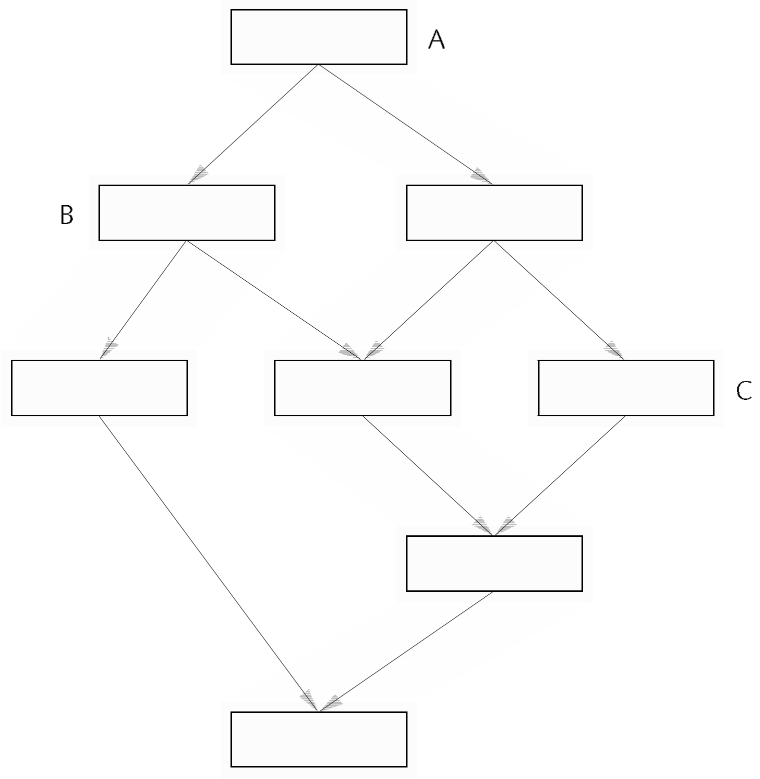


Figure 9.5: Control-flow graph with selected traces (A, B, C).

graph, the next likely path is selected and scheduled. This continues until all blocks have been scheduled. Note that traces do not overlap, i.e. every block belongs to exactly one trace.

Which path is likely to be executed can be determined using *static branch-prediction*. The root block of the graph is taken as the first block in the trace. The selection then follows branches to other blocks, and wherever multiple paths are possible, a prediction is made on which direction the corresponding branch is likely to take. Ball and Larus [BaLa93] have found a set of heuristics achieving correct prediction rates above 80%. Alternatively, *profiling* could be used to get even better results. In profiling, the code is instrumented to collect information on which paths are taken most frequently at execution time. The program is then run on a sample set of data, and the collected information is used to guide later compilations. Correct prediction rates above 90% are possible, but depend heavily on how realistic the sample data is.

The selection of a trace on a nested region graph as in OOC2 is no different from the selection on a control-flow graph. For pairs of guards, one is predicted as the likely path and added to the trace. The selection then continues within its nested instructions.

When scheduling a trace, implicit code motions over region boundaries are encountered. Figure 9.6 shows the two possible directions. If an instruction is moved into a region, a new condition *cond* is added to the guards controlling execution of this instruction. By making sure that the same instruction is also executed under the condition ¬*cond*, semantic equivalence can be ensured. More technically, a so-called *compensation copy* of the moved instruction has to be inserted in the region corresponding to the inverse condition. This inverse condition always represents an *off-trace* path. The results of both instructions have to be combined by a gate and some uses have to be renamed accordingly to reference the compensation copy or the gate. Keeping track of such movements and generating compensation instructions is called *bookkeeping* and is an essential activity in trace scheduling.

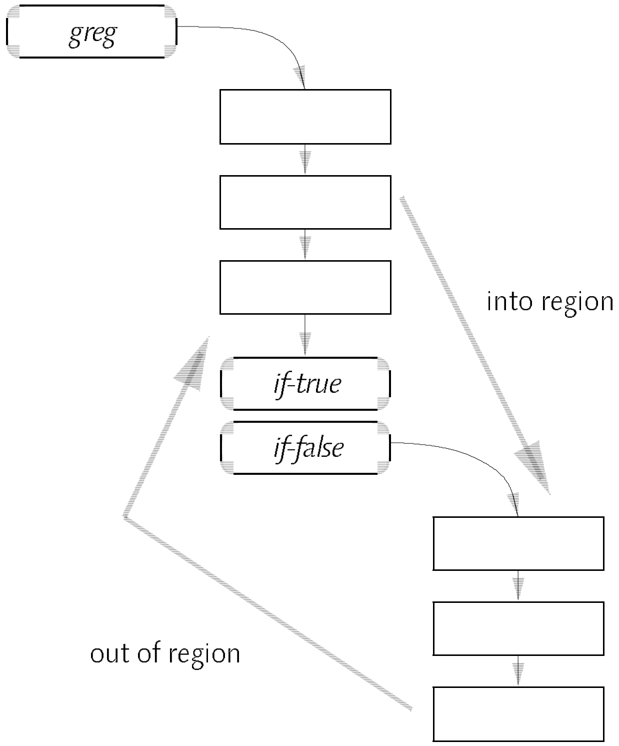


Figure 9.6: Code motions over region boundaries in trace scheduling.

On the other hand, when an instruction is moved out of a region, a condition guarding its execution is ignored. We say the instruction is executed *speculatively*. No compensation copies are required in this case. Note that instructions that may raise exceptions or change global data cannot be executed speculatively, as they change the output of the program. Motion of such instructions out of regions has to be prohibited, for instance by adding dependency edges between their enclosing region, them, and the next merge on the path. Alternatively, all instructions up to the speculative one can be moved back into the original region, and compensation copies can be generated where needed. This back-movement is called *rejoining*, as the position where paths are *joined* (*merged* in our terminology) is moved.

An instruction can be both executed speculatively and require compensation copies, namely when it has been moved from within one control structure into another control-structure not nested within the first. Keeping track of all cases is complicated and makes a trace scheduler hard to implement correctly.

For OOC2, we have developed an alternative approach to bookkeeping. Instead of keeping track of code motions, bookkeeping is implemented as an algorithm reestablishing an invariant of the intermediate program representation: The definition of a value must dominate all its uses, with the exception of gates, where the definition must dominate the path corresponding to the gate operand. When moving an instruction into a region, this invariant may not hold any more. If after scheduling a trace, a use is found which is not dominated by the corresponding definition, gates and compensation copies are added until all paths to the use contain a definition of the value. Figure 9.7 shows an example.

Instruction *X* in block *A* originally dominated its use in instruction *Y* in block *G*. *X* was moved into block *E* by the scheduler, and now does not dominate *Y* any more. The algorithm detects this, and places a gate in block *G* which will combine computations of *X*. Since the path through *B* does not contain a computation of *X*, a copy is inserted in *B*, and made the first operand of the gate in *G*. On the other path, it is found that a gate in block *F* is required, which combines the original computation *X* with the compensation copy in block *D*. After this, there is no path to *Y* which does not contain a computation of *X*.

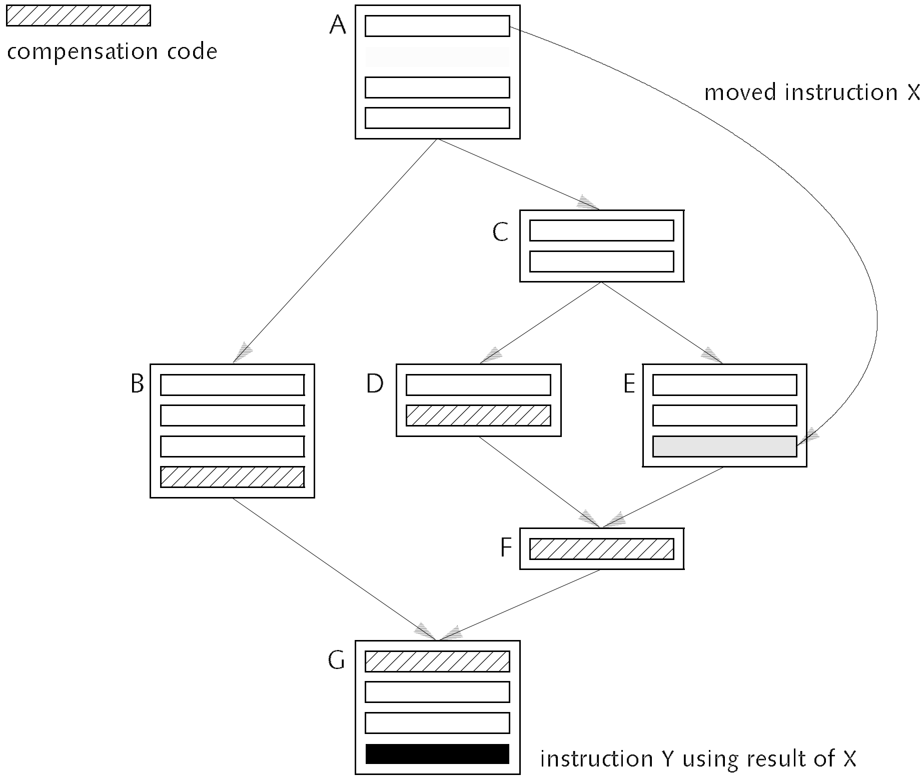


Figure 9.7: Bookkeeping by reestablishing dominance of definitions over uses.

This algorithm simultaneously achieves some of the effects of *partial dead code elimination* [KnRS94], which moves instructions to a position where they dominate all uses but no other paths. This corresponds to lazy evaluation, where a computation is only performed if its result is known to be needed. If an instruction is moved into a region by the scheduler, but still dominates all uses, no superfluous compensation copies will be generated as compared to the original bookkeeping algorithm.

The restriction of trace scheduling to acyclic graphs prevents it from significantly improving innermost loops, where most of the execution time is spent. If multiple iterations of the loop could be overlapped, much better schedules were possible. By first *unrolling* loops, i.e. repeating the original loop body several times, trace scheduling can be applied to it. However, trace scheduling already increases code size through its compensation copies, and loop unrolling continues this trend. Good heuristics are needed to keep the code size within reasonable limits.

For a more thorough discussion of trace scheduling and a description of how it has been implemented for OOC2, see the diploma thesis of Bolliger [Boll94].

9.4 Software Pipelining

As we have seen in the last section, trace scheduling must resort to loop unrolling in order to improve innermost loops. These loops are the section of code in which most of the execution time is spent. Because of this, special scheduling methods for innermost loops have been designed under the name *software pipelining*. Their goal is to overlap the execution of several loop iterations just like a processor pipeline overlaps the execution of several instructions. This is superior to loop unrolling followed by trace scheduling, as it allows a higher degree of overlap. Trace scheduling an unrolled loop with two iterations in its body allows to intermix the execution of iterations 1 and 2, 3 and 4, and so on. However, it does

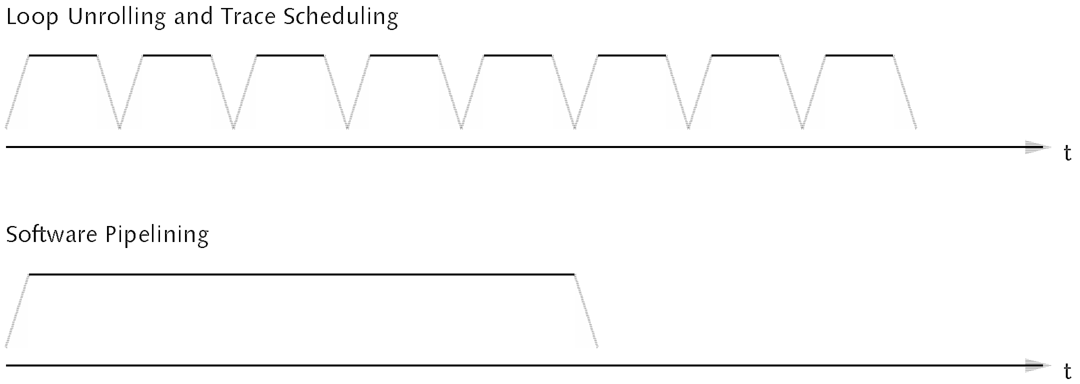


*Figure 9.8: Overlap of 8*2 iterations in trace scheduling and software pipelining. Pipeline startup and drain costs are shaded gray.*

not allow to overlap the execution of iterations 2 and 3. Thus, we encounter pipeline startup and pipeline drain effects at the beginning and the end of the trace scheduled unrolled loop, which are avoided in software pipelined loops (Figure 9.8).

Software pipelining achieves the effect of unlimited loop unrolling without the associated code size increase. Usually, it is not required to unroll the loop in order to obtain good overlap in the pipeline. If unrolling is required in order to obtain a good schedule, it can be computed by how much a loop should be unrolled, while with trace scheduling, one has to rely on heuristics.

In this section, we will discuss several methods for software pipelining.

Classic Software Pipelining

The fundamental idea of software pipelining – namely starting an iteration of a loop before the previous iterations have terminated – can be implemented with a simple modification to a local scheduler. Assume the scheduler is cycle-driven. When the selection function could not find an instruction to be issued for the current cycle, a new iteration of the loop is added to the set of instructions to be scheduled, increasing the amount of parallelism. Note that this simultaneously unrolls the loop. After adding the iteration, the selection function is called again, and hopefully will find an instruction to issue.

Figure 9.9 shows a simple code pattern for a block copy, and Figure 9.10 exemplifies how it would be software pipelined for the PowerPC 601. The decnz- and decz-guards correspond to a PowerPC branch-and-count instruction which simultaneously decrements and tests the count register.

```
(1) greg:
      (2) l-merge:   (1), (8)
      (3) gate    (2), src, (6:1)            ; address of src -4
      (4) gate    (2), dest, (7:1)           ; address of dest -4
      (5) gate    (2), count, (8:1)          ; loop counter
      (6) lwu    (3), 4
      (7) stwu    (4), 4, (6)
      (8) decnz:   (5)                       ; decrement and branch
      (9) decz:   (5)
```

Figure 9.9: Code pattern for a block move.

cycle	ready	emitted
0	$\{(2)_{(0,0)},(3)_{(0,0)},(4)_{(0,0)},(5)_{(0,0)},(6)_{(0,3)},(8)_{(0,0)}\}$	(2),(3),(4),(5),(6)
1	$\{(7)_{(2,1)},(8)_{(0,0)}\}$	(8),*add iteration*
1	$\{(7)_{(2,1)},(2)^1{}_{(0,0)},(3)^1{}_{(0,0)},(4)^1{}_{(0,0)},(5)^1{}_{(0,0)},(6)^1{}_{(0,3)},(8)^1{}_{(0,0)}\}$	$(2)^1,(3)^1,(4)^1,(5)^1,(6)^1$
2	$\{(7)_{(2,1)},(7)^1{}_{(3,1)},(8)^1{}_{(0,0)}\}$	(7)
3	$\{(7)^1{}_{(3,1)},(8)^1{}_{(0,0)}\}$	$(7)^1,(8)^1$

Figure 9.10: Software pipelining the loop of Figure 9.9.

There are several things to note in this example. Instructions are suffixed with the iteration they belong to, except for iteration 0. Merges and gates do not require any resources and have a latency of 0, thus they can be scheduled immediately after becoming ready. We distinguish between *branches* and *non-branches*, i.e. all other instructions. Loop-closing branches are usually only scheduled at the end of the body; a restriction which has to be lifted here.

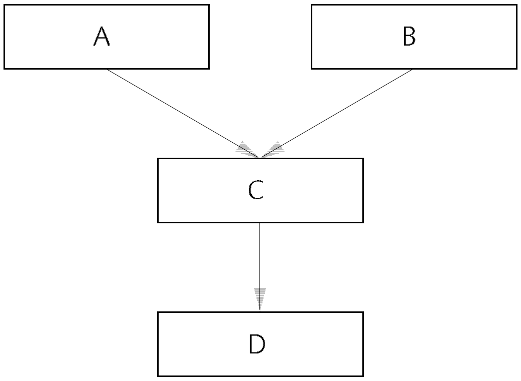
In cycle 1, the scheduler did not find a non-branch instruction to issue. It decides to add a new iteration, the instructions of which become only ready after their controlling branch has been scheduled, so the branch (8) is emitted in advance. Similarly to trace scheduling, this motion of the branch over the store instruction requires to copy the store into both subsequent paths. After this, the scheduler finds enough operations to fill all pipeline slots, resulting in the following final code.

```
(1) greg:
        (2) l-merge:  (1), $(8)^1$
        (3) gate   (2), src, $(6:1)^1$
        (4) gate   (2), dest, $(7:1)^1$
        (5) gate   (2), count, $(8:1)^1$
        (6) lwu    (3), 4
        (8) decnz:  (5)
                $(6)^1$  lwu   (6:1), 4
                (7)  stwu   (4), 4, $(6)^1$
                $(7)^1$  stwu   (7:1), 4, (6)
                $(8)^1$  decnz:  (8:1)
                $(9)^1$ decz:  (8:1)
        (9) decz:  (5)
                (7)'  stwu   (4), 4, (6)
        (10) c-merge:  (9), $(9)^1$                ; exit path
```

Since the addition of the second iteration corresponds to unrolling the loop, several operands had to be renamed. The instructions of the second iteration no longer reference the gates, but use the values generated in the first iteration – found in the last operand of the original loop gates. The gates have to be modified to use the values computed in the second iteration instead of the ones from the first.

As simple as the method looks, there are many problems with it, which have also been noted by Aiken and Nicolau [AiNi91]. In general, it is not easy to find out in which cycle a loop-closing branch can be placed. Consider the dependence graph of a loop body and the corresponding software pipelined code for a machine with two identical functional units in Figure 9.11.

There is no cycle in which all initiated iterations have terminated, and so there is no cycle in which a branch back to the very beginning could be placed. However, there is a certain pattern in the schedule, namely the schedules for cycles 1, 3, and 5 and for cycles 2, 4, and 6 are equal except for incremented loop indices. If there were a loop-closing branch from the end of cycle 2 to cycle 1 the schedule would be correct.



cycle	unit 1	unit 2
0	A^0	B^0
1	C^0	A^1
2	D^0	B^1
3	C^1	A^2
4	D^1	B^2
5	C^2	A^3
6	D^2	B^3
7	...	...

Figure 9.11: Dependence graph of a loop body and the corresponding software pipelined code.

The problem faced is finding such repeating patterns, which can only be solved if restrictions are made to the scheduler. Aiken and Nicolau [AiNi91] propose the scheduler to be deterministic, i.e. to be a function of the ready set. If done so, the ready set completely identifies a state of the scheduler. If a ready set is encountered that had already been scheduled, which is equal to the current ready set except for all iteration indices differing by the same constant *k*. Then both sets represent the same state, and a back-edge can be placed to the scheduled state. In the above example, we would find cycles 1 and 3 to have the ready sets $\{C^0, A^1, B^1\}$ and $\{C^1, A^2, B^2\}$ respectively. The iteration indices all differ by one – corresponding to one iteration having passed in between – and the sets represent the same state. Note that for a good schedule, not only the ready instructions have to be the same in the set, but also the remaining latencies until they can be issued.

The loop in which iterations are completely overlapped is called the *steady state* – corresponding to the pipeline being completely filled. Possibly there is also some code before it to fill the pipeline, which we call the *pipeline startup* code, as well as some code behind it to finish initiated iterations called *pipeline drain* code.

As is, the algorithm cannot be guaranteed to terminate. It is possible that an infinite number of different states would be generated, as exemplified in Figure 9.12. Assume the target machine would contain an integer-unit and a floating-point unit, and the loop body would include two instructions *A*, *B* for the integer-unit and an instruction *F* for the floating-point unit, all of which were independent of each other.

In cycle 1, there is no available instruction for the FPU, so a new iteration is added, which adds two integer and one floating-point instruction to the ready set. The same happens for all subsequent cycles. On the other hand, one floating-point instruction and only one integer instruction will be scheduled from the ready set every cycle, increasing the number of ready integer instructions by one every cycle, and thus leading to an infinite growth in the ready set.

cycle	ready	IU	FPU
0	$\{A^0, B^0, F^0\}$	A^0	F^0
1	$\{B^0, A^1, B^1, F^1\}$	B^0	F^1
2	$\{A^1, B^1, A^2, B^2, F^2\}$	A^1	F^2
3	$\{B^1, A^2, B^2, A^3, B^3, F^3\}$	B^1	F^3

Figure 9.12: Example generating an infinite number of ready states.

By restricting the number of iterations from which instructions can be ready simultaneously, the problem can be solved, as pointed out by Aiken and Nicolau [AiNi91]. Let N be this upper bound of iterations with instructions in the ready set. Then a new iteration M can only be added if there is no instruction X^{M-k} in the ready set with $k > N$.

In the presence of control flow within the loop body, the software pipelining algorithm follows each path, adding iterations as necessary and placing loop back-edges when possible. That is, if there is an If-statement in the loop, both its THEN- and its ELSE-path may include instructions from subsequent iterations and can have separate loop-closing branches. The scheme allows loops with arbitrary control flow to be software pipelined, but possibly at a very large code size increase.

The presented algorithm is elegant, allows to software pipeline arbitrary loops, and achieves very good schedules. However, it tends to result in very large code and requires large amounts of compile-time, both because a lot of code is generated and because it has to keep track of all states that the ready set took on. For further details on the technique, see [AiNi91].

Modulo Scheduling

Modulo scheduling as described by Lam [Lam88] takes a different approach. Instead of placing restrictions on the selection function and the number of iterations in progress at any point in time, it restricts the way iterations can be overlapped. Every iteration has exactly the same schedule, and the interval at which new iterations are initiated is fixed. This interval is called the *initiation interval II*, and lower and upper bounds for it can be precomputed. For best performance, the algorithm tries to make II as small as possible.

The upper bound for the initiation interval is the length of the scheduled loop body, corresponding to no overlap between iterations and completely sequential execution of the iterations. The lower bound depends on the following two restrictions.

- *Resource restrictions*: Let the target processor have n_t units of type t, and the loop body contain m_t instructions executed on units of type t. Then, $max_t(m_t/n_t)$ over all types t is a lower bound for II.
- *Cyclic dependencies*: If the value of some variable v in iteration n is used to compute the value of v in iteration $n+k$, there is a cyclic dependency between operations. Its length is the sum of all latencies on the cycle. The length l of the longest such cycle divided by k is a lower bound for II.

The first restriction can be substantiated as follows. In the steady state, k iterations will take $k * II$ cycles to execute. Assuming that each execution unit is available every cycle, there will be $k * II * n_t$ available resources for the execution of instructions of type t. On the other hand, $k * m_t$ instructions of type t have to be executed in this time. Since a resource cannot perform multiple computations at once, $k * m_t \leq k * II * n_t$, i.e. $m_t/n_t \leq II$ for all t.

Without loss of generality, assume that k in the second restriction is 1, i.e. the computation of v in iteration $n+1$ is dependent on the computation of v in iteration n. If v is computed by the first instruction in the schedule, a new iteration cannot be started before v is available after l cycles, thus placing a lower bound on the initiation interval. If v is determined by the j'th instruction in the schedule, it will be so in all iterations, thus the same bound l applies to the initiation interval.

```
(1) greg:
        (2) l-merge:   (1), (8)
        (3)  gate    (2), src, (6:1)               ; address of src –4
        (4)  gate    (2), dest, (7:1)              ; address of dest –4
        (5)  gate    (2), count, (8:1)             ; loop counter
        (6)  lwu    (3), 4
        (7)  stwu    (4), 4, (6)
        (8)  decnz:   (5)                          ; decrement and branch
        (9)  decz:   (5)
```

Figure 9.13: Code pattern for a block move.

Note that the initiation interval is an integer number. However, if the lower bound is a rational number of the form *p/q*, unrolling the loop *q* times will allow for an initiation interval bound of *p* cycles for *q* iterations. Alternatively, the lower bound can be rounded up to the next integer number.

The software pipelining algorithm precomputes this lower bound and then systematically constructs a schedule for the loop. Reconsider the above example for a block move (Figure 9.13).

For the sake of clarity, we omit the merges and gates in the following as they do not influence the schedule, leaving only the load, the store, and the branch to be scheduled. When scheduling for the PowerPC 601, the integer-unit will be required twice by the load and the store, placing a lower bound on *II* of 2. Figure 9.14 shows the generated schedule, where a new iteration is started every *II* cycles.

The store operation $(7)^0$ could not be scheduled in cycle 2, as the load of the next iteration had to be scheduled there. Remember that the schedule for every iteration must look the same and that after every *II* cycles, a new iteration has to be started. If the next iteration were added in cycle 4, cycles 4 and 5 would be the same as cycles 2 and 3, except for the incremented iteration indices. Thus, cycles 2 and 3 make up the steady state, i.e. there is a branch at the end of cycle 3 to cycle 2. Cycles 0 and 1 and cycles 4 and 5 represent the pipeline startup and pipeline drain sequences.

Instead of requiring a technique to determine when the steady state is found, *modulo scheduling* allows to directly generate the steady state. The fundamental idea behind it is the so-called *modulo reservation table*. If an instruction *A* is scheduled for cycle *t*, it also has to be scheduled for all cycles $t + II * k$, $k \geq 1$. Instead of repeating this placement information over and over, one could only store its relative position *t MOD II* within an interval once. This can be done in a reservation table with *II* rows, which is

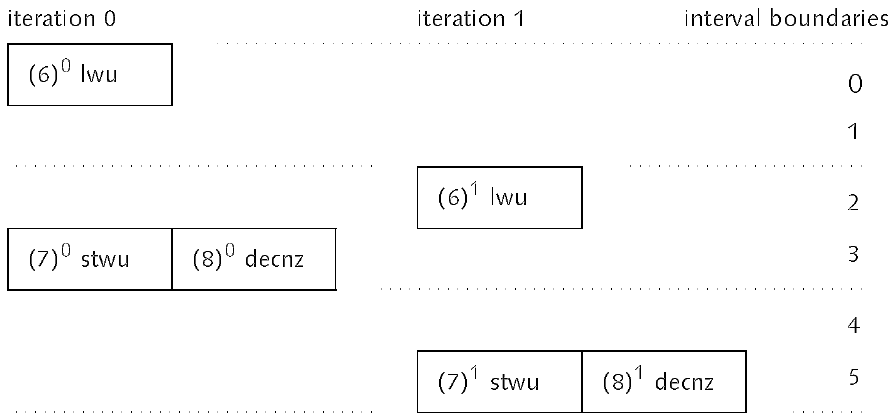


Figure 9.14: Software pipelined block move.

(6)^{0} lwu		0
(7)$^{-1}$ stwu	(8)$^{-1}$ decnz	1

Figure 9.15: Modulo reservation table for the block move.

called the *modulo reservation table*. Figure 9.15 shows the modulo reservation table for the previous example.

After scheduling the load instruction into the table, the scheduler does not proceed beyond cycle 1 but wraps around into cycle 0. After wrapping around once, instructions then scheduled belong to the iteration started in cycle $-1*II = -2$. Since the integer unit is already reserved in cycle 0, the store and the branch are placed into cycle 1. Note that these instructions are marked with the corresponding iteration −1.

(6)$^{-1}$ lwu		startup
▨	▨	

(6)^{0} lwu		steady state
(7)$^{-1}$ stwu	(8)$^{-1}$ decnz	

▨		drain
(7)^{0} stwu	(8)^{0} decnz	

Figure 9.16: Generating pipeline startup and drain code.

The schedule generated into the modulo reservation table forms the steady state of the software pipeline. The pipeline startup and drain code can be generated easily from the steady state, namely by copying parts of it. Note that before entering the steady state, the load instruction of iteration −1 is missing, while after exiting from the steady state, the store and the branch of iteration 0 are missing. This information can be obtained from the modulo reservation table, as shown in Figure 9.16.

A copy of the reservation table with appropriately renumbered entries is used as the startup code. The load belonging to iteration −1 is needed, but since there is no iteration −2, the store and branch numbered −2 can be omitted. This translates to the following procedure.

```
PROCEDURE GenerateStartup (table: ModuloResTable; nofIter: INTEGER);
   VAR iter, newIterIndex: INTEGER;
BEGIN
   iter := 1;
   WHILE iter < nofIter DO
      FOR each instruction instr in table DO
         newIterIndex := instr.iterIndex − iter;
         IF newIterIndex >= −nofIter THEN
```

```
            CopyIntoStartup(instr, newIterIndex)
          END
        END ;
        INC(iter)
      END
    END GenerateStartup;
```

nofIter represents the number of iterations overlapped in the reservation table, which is the number of times the scheduler wrapped around plus one. The instructions in the table have associated iteration indices in the range –(*nofIter*–1)..0. The algorithm determines for each instruction from the table which iteration index it would receive when copied into the startup code. If it belongs to an existing iteration, it is copied, otherwise it is skipped. The procedure to generate the pipeline drain code is similar, with the only difference being that the iteration count *iter* is added instead of subtracted to obtain the new iteration index, and that the check of whether it belongs to an existing iteration is a check of being less than or equal to zero.

In this software pipelining technique, a reservation table is used, and thus operation-driven scheduling is feasible. In the presence of cyclic dependencies, it makes a lot of sense to schedule the dependence cycles first, calling for such a scheduler. Assume a cyclic dependence of length *l*, which places the lower bound on *II*. In this case, every instruction on the dependence cycle must be issued as soon as it becomes ready to be issued, i.e. all latencies are fulfilled. If an instruction has to be delayed because of resource conflicts, the cycle lengthens and *II* increases as well. In the presence of dependence cycles, the algorithm can fail to produce a schedule with a given *II*, and then the process has to be started all over with a larger *II*. Lam [Lam88] recommends to increase *II* by 1 until the algorithm finds a schedule, and claims that the scheduler very often succeeds within the first three trials. Note that the algorithm must terminate when *II* reaches its upper bound.

When the lower bound is given by resource constraints, there is no transformation which can improve upon the bound. However, if a dependence cycle determines the lower bound, and the dependence cycle includes conditional branches, moving instructions over the branches may reduce the dependence length and thus allow to achieve a smaller *II*. Reconsider the list traversal loop we have seen in Chapter 4.

```
p := root;
WHILE (p # NIL) & (p.key # key) DO p := p.next END
```

The loop translates into the following GSA code.

```
(1) greg:
     (2) p := lw    root
     (3) l-merge:   (1), (9)
          (4) p := gate    (3), (2), (10)
          (5) cmp    (4), 0
          (6) if-neq:   (5)
               (7) lw    (4), 0                      ; p.key
               (8) cmp    (7), key
               (9) if-neq:   (8)
                    (10) p := lw    (4), 4           ; p.next
               (11) if-eq:   (8)
          (12) if-eq:   (5)
          (13) c-merge:   (11), (12)                 ; exit path
```

When scheduling for the PowerPC 601, there are two load and two compare instructions to be executed in the integer unit, placing a lower bound of four on *II* due to resource constraints. However, there is a long cyclic dependence chain which puts the lower bound to eight (Figure 9.17).

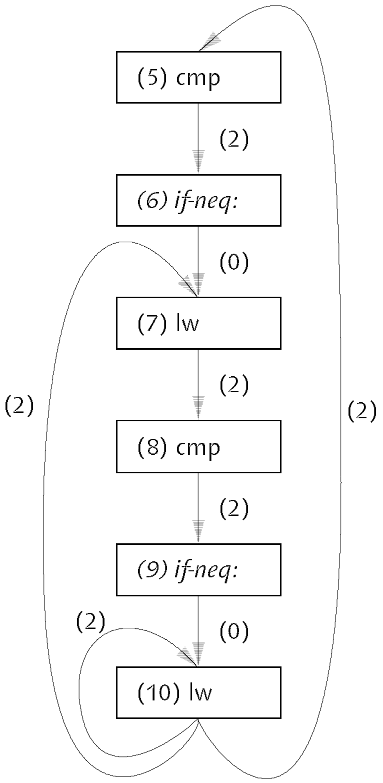


Figure 9.17: Cyclic dependence chains for the list traversal loop.

There are several cyclic dependence chains in the graph, the longest with a length of 8. However, many delays are caused by having to wait until conditions are resolved, and by speculatively executing some instructions, some dependence cycles can be broken. For example, by allowing all load and compare instructions to be moved over guards (6) and (9), the longest dependence cycle will then be a two-cycle one between successive accesses to p.next (the dependence cycle on instruction (10)). Figure 9.18 shows the modulo reservation table after allowing the speculative motions and obtaining an *II* of 4. The finally generated code for POWER-2 which even achieves an *II* of 2 is shown in Chapter 4.

We have not yet discussed the interplay of software pipelining and register allocation. Note that when there are overlapping loop iterations, each iteration typically requires its own set of registers so that values of other iterations are not overwritten. For example, in the reservation table for the block move (Figure 9.15), a new value is loaded before the previous one has been written. One could either save

$(5)^0$ cmp		0
$(7)^0$ lw	*(9)* $^{-1}$ *if-neq:*	1
$(10)^0$ lw	*(6)* ^{0} *if-neq:*	2
$(8)^0$ cmp		3

Figure 9.18: Modulo reservation table for the list traversal loop after speculative code motions.

the old value using a register-register move, which would destroy the carefully constructed schedule, or one could emit two copies of the steady state and appropriately rename registers. An upper bound for the number of copies required is the number of iterations in the modulo reservation table. With a detailed analysis of the live ranges as performed by register allocation a smaller number of copies may sometimes be found.

One problem we have not yet dealt with is how to perform modulo scheduling on control structures in a loop. Note that modulo scheduling is limited to innermost loops, so the only control structures in question are If-statements and possibly Case-statements.

Lam [Lam88] proposes *hierarchical reduction*, in which each path through the control structure is scheduled separately into a reservation table, and is then considered as one "large" instruction consuming many resources at once and being scheduled like simpler instructions. This approach is very closely reflected by GSA form, in which a guard can also be considered as a "large" placeholder instruction for all nested instructions.

Warter et al. [WHSB92] describe a different technique called *enhanced modulo scheduling*. Each instruction is annotated with a predicate under whose control the instruction is executed. Since every instruction can then be executed conditionally, the control structure completely vanishes. This model is called *predicated execution* [RYYT89] and requires special hardware support. In later work, Warter et al. [WMHR93][WaLH93] describe how the technique can be applied to conventional machines by first assuming that predicated execution is available, and then reconstructing a possible control-flow graph through replacing the predicates. Note that the model of predicated execution is also very well reflected in GSA form, as the controlling predicate for an instruction is readily available in its guard.

Other Algorithms for Software Pipelining

Circular scheduling is a simple software pipelining technique used in recent MIPS compilers and described by Jain [Jain91]. It only applies to loops consisting of a single basic block, i.e. with no branching within the loop body, and for which the number of iterations can be precomputed. Assume a loop is represented as $\{ABC\}^n$, where A, B, and C correspond to instructions and n is the number of iterations executed. The execution of this loop is equivalent to the execution of $A\{BCA\}^{n-1}BC$ or $AB\{CAB\}^{n-1}C$. Circular scheduling tries to find a good such ordering through heuristics deciding which instructions to *circle* from the top of the loop to the bottom. It schedules these orderings, and terminates as soon as an ordering with no pipeline stalls is found or no other reasonable ordering can be tried out. Jain also describes how register renaming helps to reduce dependencies and how it can be integrated into the algorithm.

The technique is restricted to overlapping at most two iterations and cannot create multiple copies of the loop body. Larger amounts of overlap can only be achieved through previous loop unrolling. Furthermore, due to its restriction to simple loops only, its effect on run-time is smaller than for the other algorithms.

Blainey [IBM94b] describes the software pipelining technique used in the IBM XL compilers, which is similar in spirit to circular scheduling. From the description, we are not able to tell in which respects it differs.

9.5 Register Allocation

So far, all our algorithms made the assumption, that an infinite number of registers is available. Every value computed is kept totally separate from others, and thus consumes its own *virtual register*. Since we are applying machine operations to values, they have to be stored in physical registers when used as operands in our machine model. If the number of values would be smaller than the amount of available physical registers, register allocation would be trivial: The compiler would just number all values uniquely. However, the usual case is that the number of values is larger than the amount of physical registers. Thus, some values must share a single physical register or must temporarily be kept in memory rather than in registers. The decision which values to store in which location is the task of a register allocator. In this section, we outline register allocation using *graph coloring*.

Live Ranges and the Interference Graph

Graph coloring register allocators are based on the notion of *live ranges* and the *interference graph* on them. A live range of some value *v* is the union of the intervals from the definition point of *v* to all uses of *v* in the scheduled code. Figure 9.19 shows a section of code and the live ranges in it.

```
(9)  adr   a
(10) l-merge:   (1), (24)
 (11) i := gate    (10), i, (16)
 (12) s := gate    (10), s, (23)
 (13) cmp    (12), 100
 (14) mul    (11), 4
 (15) lwx    (9), (14)
 (16) i := add    (11), 1
 (17) cmp    (16), 127
   (18) if-gtr:   (13)
       (19) s := subf    (15), (12)
   (20) if-lte:   (13)
       (21) s := add    (12), (15)
   (22) i-merge:   (18), (20)
 (23) s := gate    (22), (19), (21)
   (24) if-lte:   (17)
```

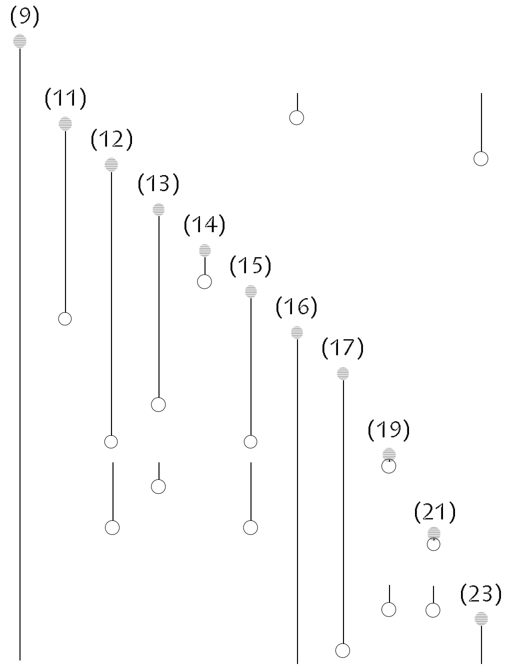


Figure 9.19: Code example with associated live ranges.

A live range may consist of several short intervals, all of which should be allocated to the same register. The idea is as follows: The defining computation evaluates the value into a register *R*, and all instructions using the value as an operand access it from *R*. When two live ranges overlap in the above graph – e.g. range (9) overlaps with all other ranges – we say the ranges *interfere*. Interfering ranges cannot be allocated to the same register *R*: Doing so would mean to ask *R* to store two values at once.

The information, which live ranges interfere and which do not, can be represented in an *interference graph* – a graph whose nodes represent live ranges. There is an edge between two nodes if they interfere. Figure 9.20 shows the interference graph of the example in Figure 9.19.

There are two disjoint graphs, one for general purpose registers and another one for condition code registers. If the code would access floating-point registers, they would be represented in a third graph. Since integer values and condition codes cannot be allocated to the same class of registers in the PowerPC architecture, they will be treated separately. That is, the GPR graph is mapped onto the available general purpose registers, while the condition code graph is mapped onto the available condition code registers.

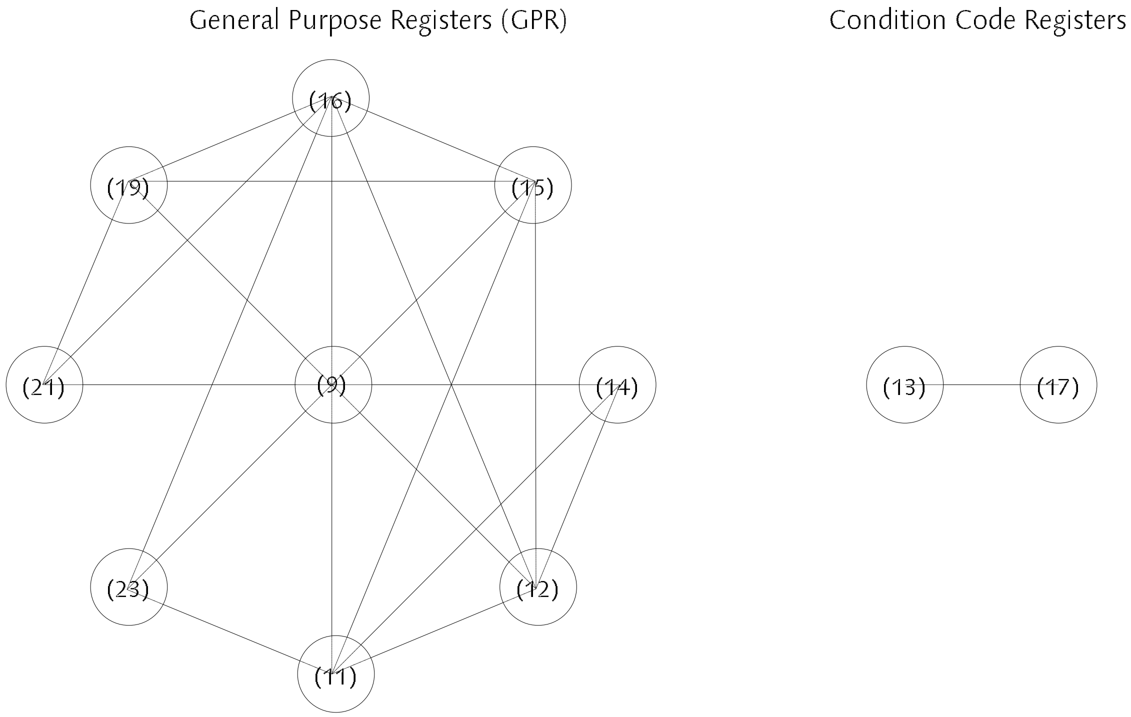


Figure 9.20: Interference graphs of the example in Figure 9.19.

Registers should be assigned to life ranges in a way, that all nodes connected by an edge get different register numbers. This problem is equivalent to the *graph coloring problem*, where two nodes cannot receive the same color if they are connected. This equivalence was first noted by Chaitin et al. [CACCHM81]. Since then, graph coloring has become the standard technique to allocate registers in optimizing compilers. Chow and Hennessy [ChHe90] have developed simplifications to the algorithm which allow a faster implementation, delivering only slightly inferior allocations. The techniques by Chaitin and Chow represent the two main directions in graph coloring register allocation research. In the following, we present the original algorithm by Chaitin.

Computing the Interference Graph

The interference graph is typically represented as a bit-matrix *b*, in which element b_{ij} is TRUE corresponds if live ranges *i* and *j* interfere. It can be constructed by propagating a vector of currently live values over the program. The propagation proceeds backwards over the program and iterates until a fixpoint is reached. Upon encountering the *end* of a live range *l*, i.e. value *l* is used at the current point but not yet in the live vector, *l* is added to the vector. If the defining computation of a value *m* in the vector is encountered, *m* is removed from the live vector, as it is not live before its definition.

Whenever two values are live at the same point in time, their live ranges interfere. A convenient method to record this is to add interferences between live range *l* and all ranges in the live vector at the time *l* is added to the live vector. Due to cyclic dependencies in case of loops, the algorithm has to be iterated until a fixpoint is reached. The method is optimistic in that it assumes live ranges not to interfere in the beginning.

```
PROCEDURE ComputeInterferenceGraph (P: SSAprogram; VAR b: Bitmatrix);
  VAR live: Bitvector; instr: Instruction; res: Result; opnd: Operand; R: Node;
BEGIN
  live := {}; b := {};
  REPEAT
    FOR each instruction instr in reverse order DO
      FOR all results res of instr DO EXCL(live, res) END ;
      FOR all operands opnd of instr DO
        IF ~(opnd IN live) THEN
          FOR all ranges R in live DO Interfere(b, opnd, R) END ;
          INCL(live, opnd)
        END
      END
    END
  UNTIL fixpoint reached
END ComputeInterferenceGraph;
```

Coalescing Live Ranges

Coalescing is the activity of combining two live ranges which are connected by a copy-instruction into one live range. In OOC2, there are no ordinary copy-instructions after copy propagation, but two instructions with similar properties. When a value must be allocated in a certain physical register *R*, e.g. in order to accommodate the calling conventions, the compiler inserts a *load-register-physical (lrp)* instruction. The instruction copies between the original value, which can be located anywhere, and the value constrained to *R*. Constraining the original value to *R* is not always possible, as shown in Figure 9.21.

```
(10)  add    i, 1
(11)  mul    j, i
(12)  call   p, (10):R3
(13)  call   q, (11):R3
```

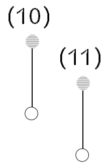


Figure 9.21: Two interfering live ranges constrained to the same physical register R3.

Both (10) and (11) need to be in R3, but since they interfere, they cannot be allocated to the same register. It would be a very difficult problem to find a position at which inserting a register-register move would resolve the conflict. The register allocator therefore inserts lrp-instructions right before all uses constrained to a physical register, and right after all definitions constrained to a register, making the rest of the live range unconstrained (Figure 9.22).

The compiler coalesces all live ranges constrained to the same physical register into one live range *P*, as they have to be allocated to the same register. It then searches for ranges connected by lrp-instructions, which do not interfere with *P* and which are not yet constrained to another physical register. If such a range is found, it is coalesced into range *P*, avoiding the register-register move a lrp-instruction corresponds to. In the above example, after coalescing (14) and (15), (10) does not interfere and can be combined as well, making instruction (14) superfluous. (11) interferes with (14)–(15), however, and can thus not be coalesced into the same range.

```
(10) add   i, 1
(11) mul   j, i
(14) lrp   R3, (10)
(12) call  p, (14):R3
(15) lrp   R3, (11)
(13) call  q, (15):R3
```

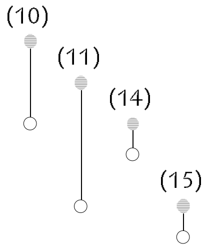


Figure 9.22: Avoiding the problem in Figure 9.21 by introducing lrp-instructions.

Another kind of register-register moves in OOC2 occurs in conjunction with gates. A gate combines two values and generates a third, which are all represented as separate live ranges. Consider the example in Figure 9.23.

The values (1) and (2) must be copied to the same register R_X, in which value (3) will be found. If (1) and (2) can be coalesced with (3), the copy assignments (4) and (5) can be omitted. Whether coalescing is possible is derived from the original interference graph.

Great care must be taken when inserting such copy assignments after the interference graph has been built. Consider an If-statement, in which in one path two variables *a* and *b* are swapped, while in the other path they are unchanged. Copy propagation will have removed the corresponding copy assignments. However, the original value of *a* will have to be combined with the value of *b* from the other path in the gate, and vice versa. Thus, the compiler has to generate a valid order of register assignments which performs a register swap. There are less obvious cases in which similar problems are encountered.

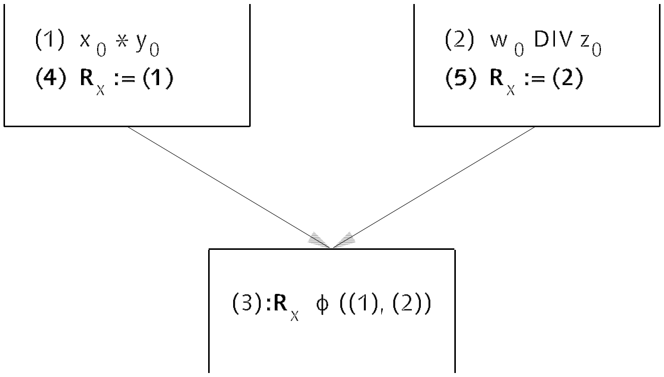


Figure 9.23: Translation of gates.

Coloring the Interference Graph

After live ranges have been coalesced, the register allocator tries to assign a color to each live range, so that no interfering ranges receive the same color. The number of available colors corresponds to the number of available registers. If there are *k* registers, we say that we search a *k*-coloring for the graph. If no coloring is found, the compiler selects some ranges for *spilling*, i.e. some values are kept in memory instead of registers. It then rebuilds the interference graph, and tries to find a *k*-coloring again. This process is repeated until a *k*-coloring is found.

Since the problem of finding a *k*-coloring for an arbitrary graph is NP-complete, we rely on non-optimal heuristic techniques. Chaitin's heuristic [CACCHM81] is based on the *degree* d_N of a node *N*, which is the number of neighbors of *N*, i.e. the number of edges between *N* and other nodes. His algorithm proceeds as follows.

```
PROCEDURE ColorGraph (g: Graph; k: INTEGER);
   VAR stack: Stack; N: Liverange; C: INTEGER;
BEGIN
   REPEAT
      empty stack;
      select a node N from g with degree d_N < k;
      WHILE such a node N is found DO
         remove N from g, lowering the degree of the neighbors of N;
         Push(stack, N);
         select a node N from g with degree d_N < k
      END ;
      IF ~(g is empty) THEN
         select node N to spill;
         Spill(N);
         rebuild interference graph g
      END
   UNTIL g is empty;
   (* success, all nodes could be removed and are now on the stack *)
   WHILE ~(stack is empty) DO
      Pop(stack, N);
      add N and all its previous edges to graph g;
      select a color C different from the colors of all neighbors of N;
      assign C to N
   END
END ColorGraph;
```

The algorithm works because of the following observation: If a node *N* has a degree $d_N < k$, d_N colors are used in its neighbors, and there must be another color for *N* out of the *k* available colors. That is, if a node *N* with degree d_N is successfully removed from the graph, it will also have d_N neighbors when it is added to the graph in the second step, and there will be a color for it. Figure 9.24 depicts the process on a simple graph taken from [Brig92], which is 3–colored.

The technique is sensitive to the order in which nodes are selected from the graph. A good heuristic is to select the node with the highest degree first, as it will lower the degree of the largest number of neighbors. Note that the algorithm may not find a *k*-coloring for a graph which is *k*-colorable. In practice, however, it delivers good results.

Spilling

When the coloring algorithm finds no node *N* of degree $d_N < k$, not all nodes can be kept in the *k* available registers. Some node must be selected to reside in memory, and so-called *spill code* to load and store the value must be inserted. After removing such a node from the interference graph, the degree of neighboring nodes is reduced.

Counting the number of accesses and weighting them by the loop nesting depth delivers a reasonable approximation of the cost of spilling a node, and can be used to choose the actual node to reside in memory. Note that spilling a node *N* does not correspond to *N* not consuming any register resources at all. When its value is computed, it resides in a register until it is stored, and before it can be used as an operand, it has to be loaded into a register. However, instead of one large live range for the value, there is a set of small live ranges which is more likely to be colorable.

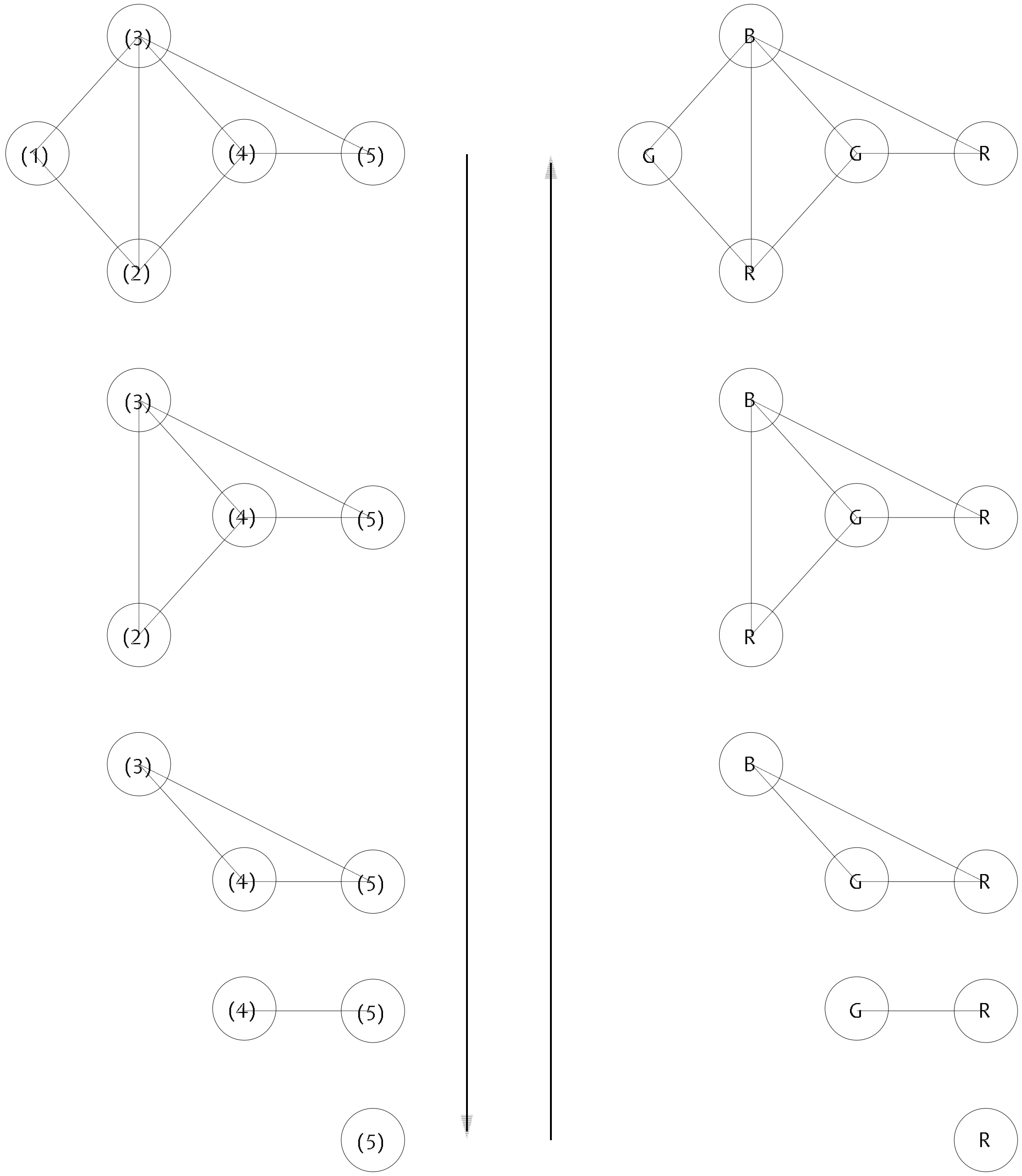


Figure 9.24: Coloring a simple graph with the colors R, G, B.

Problems with the Algorithm

The algorithm as described suffers from several problems. After one node has been spilled, the whole interference graph must be reconstructed and the coloring started from scratch. Briggs [Brig92] describes a modification to the algorithm which allows to spill several nodes at once, but which still cannot guarantee that a single pass will suffice to find an allocation.

Another problem is that the interference graph can be very large. Large procedures sometimes contain several thousand instructions. Their interference graph would be represented as a bitmatrix consuming several megabytes of memory. Such a large matrix does not fit into typical caches, causing performance degradations when allocating registers for large procedures.

Besides these two problems on the implementation side, there is also one problem with the allocation it generates. If a live range has to be spilled to memory in one place, it will reside in memory for the whole procedure. Long live ranges being defined at the beginning of the procedure and being used at the end can take up registers for the whole procedure, even though they are not accessed over large fractions of the code. Figure 9.25 shows an example.

```
a := ...; b := ...;
WHILE ..... DO
   ... a ...
   ... b ...
END ;
....
c := ...;
WHILE .... DO
   ... c ...
END ;
... a + b ...
```

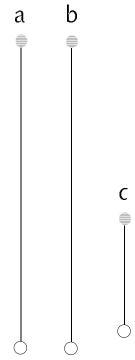


Figure 9.25: Two long live ranges (a, b) cause an often accessed live range (c) to be spilled inside a loop on a machine with 2 available registers.

In the example, *a* and *b* have larger numbers of uses than *c*, and thus will not be spilled, even though they are not accessed in the second loop. A better solution would be to allocate *a* and *b* to registers for the first loop, then spill *a* before the second loop and allocate the freed register to *c*. Doing so is called *live-range splitting*, as the range of *a* is split into two parts, with one part being allocated in a register and the other in memory. Briggs [Brig92] discusses several techniques to perform live-range splitting, with the main problem being to find which ranges should be split and where. *Hierarchical graph coloring* avoids these problems by considering the control structure of the program right from the beginning.

Hierarchical Graph Coloring

Callahan and Koblenz [CaKo91] reason that classical graph coloring register allocators do not consider the control structure of the program, and are not sensitive to local usage patterns. Moreover, they have no intelligent mechanism to place spill code, i.e. loads and stores to move values between memory and registers. Note that the goal of a register allocator is to minimize the number of dynamically executed memory accesses, so spill code should be inserted in locations where it is rarely executed. They propose *hierarchical graph coloring* as a method to split live ranges and to place spill code intelligently. This is also the method implemented in OOC2.

Instead of coloring an interference graph for a whole procedure at once, hierarchical graph coloring takes a bottom-up approach on the control structures of the procedure. It first builds interference graphs for the innermost regions, and colors them. After a coloring with *m* registers for region *R* has been found, *R* can be treated as one instruction requiring *m* registers at the next outer level. The algorithm proceeds up to the topmost region, integrating colorings for nested regions into the interference graphs of their respective enclosing regions.

This approach has several advantages. It does not need to build the interference graph of a whole procedure, but rather just interference graphs of individual regions, so the graphs become much smaller. When a value must be spilled, the algorithm has precise information about the associated costs instead of just estimates: The algorithm only decides whether a value should be kept in a register in the current region – and not in other regions – thus the number of memory accesses required when spilling the value denotes the cost precisely. In the previous algorithm, the cost is computed as the sum of all accesses in each region, weighted by an estimated execution frequency of the region. Finally, a value can reside in a register in some regions and in memory in others with hierarchical graph coloring.

When coloring the interference graph of a region, the algorithm distinguishes between *local* and *global* live ranges. *Local* live ranges are only accessed within the region, including nested regions, thus the whole live range is known in coloring. *Global* live ranges are also accessed in outer regions, and only a part of the whole live range is known. Global live ranges are never coerced into one register with other global live ranges, as two global ranges may interfere in the unknown part of their range, although they do not interfere within the current region. However, they may share a register with local ranges. This behavior can be implemented by temporarily adding interference edges between all global live ranges, and then coloring the graph. The coloring will deliver the following results: An assignment for each global live range to either memory or a register, a number *m* of registers used for local live ranges, and an interference graph encoding which ranges in registers may still be coerced into the same register. The latter is needed as we conservatively avoided to coerce global live ranges, even if they do not interfere in the colored region. Note that this register-interference graph is never larger than k^2 bits, where k is the number of available registers. There are no more than k registers to allocate, thus there cannot be more than k live ranges in registers for every region.

The result from region R is then integrated at the next outer level as follows: m live ranges for local ranges of R are introduced at the position of R, which interfere with everything being live across R. The interferences between global ranges from R are added to the interference graph. Then this graph is colored as well.

When selecting a range for spilling, the allocation in nested regions has to be considered as well. If a range is allocated to a register within a region R and should be spilled on the outside, the value has to be loaded before entering R and must be stored after exiting from R. On the other hand, when the value is kept in memory in R and should be allocated in a register on the outside, a store before R and a load after R must be added. This ignores the cases where the value is dead before or after R or when a valid copy of the value is already in memory. Some of these memory accesses can be omitted under these circumstances. The required memory accesses have to be considered when determining spill costs. Note that the allocator never changes the allocation in a nested region, but only adds spill code around that region if necessary. For loops, the spill code is added outside of R, so that it is executed only once. For guards, it is added on the inside, so that it is only executed if the region is actually entered.

Figure 9.26 depicts how this algorithm would allocate registers for the example of Figure 9.25.

```
a := ...; b := ...;
WHILE ..... DO
   ... a ...
   ... b ...
END ;
....
c := ...;
WHILE .... DO
   ... c ...
END ;
... a + b ...
```

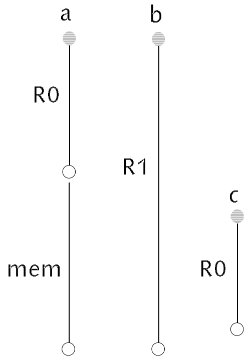


Figure 9.26: Register assignment after hierarchical graph coloring.

Register Allocation using Cyclic Interval Graphs

Another problem with graph coloring register allocators is that they do not necessarily coerce live ranges into registers so that a maximum amount of time is covered, i.e. that the register is put into use over as large a fraction of the code as possible. In an interference graph representation, information about the fraction of code covered by a live range is not available. Hendren et al. [HGAM92][HGAM93] propose a technique centered around *cyclic interval graphs*, in which this information is available, and present efficient algorithms to find sets of ranges covering maximal fractions of the code. The technique is inherently hierarchical in that it can only color one region at a time. Moreover, it requires live ranges to be a contiguous single interval. This latter aspect is an obstacle in representing the live ranges generated by arbitrary control-flow, and already causes problems with the conditional merges in OOC2.

Magun [Magun94] has implemented an experimental register allocator for OOC2 building upon cyclic interval graphs, but had to resort to interference graph representations when integrating the coloring of nested regions into the next outer region. The combination of two representations turned out to be rather complicated. Further experiments in this field are a topic of future research.

Supporting the PowerPC Calling Convention

In the PowerPC calling convention, parameters have to be passed in certain registers. Furthermore, there are so-called caller-saved registers, which are not preserved over calls, and callee-saved registers which are. The latter have to be saved by the callee if modified.

We have already discussed, how values can be forced into certain registers using location attributes and lrp-instructions. The destruction of the values in caller-saved registers over procedure calls can be modelled with the same mechanism. For every caller-saved register, a dummy result is added to the call-instruction, which is forced into the corresponding physical register. Therefore, all caller-saved registers are assigned to these dummy results at the position of the call. Ranges that live over the call will automatically be allocated to callee-saved registers.

The code to save and restore callee-saved registers that were assigned to life-ranges in a procedure is placed in the procedure prolog and epilog, respectively. In order to save as few registers as possible, the register allocator first assigns all caller-saved registers before proceeding to the callee-saved ones. It records which callee-saved registers were used, and inserts corresponding save- and restore-code after the allocation has succeeded.

10 Measurements

In this chapter we present some measurements on our optimizing Oberon-2 compiler OOC2. There are three different attributes that we consider: Size of the compiler, compilation speed, and quality of the generated code. As far as possible, we compare the numbers with those of the GNU C-Compiler 2.6.0 [Stall94] and those of IBM XLC 1.3 [IBM90d].

10.1 Compiler Size

One of our goals was to implement an optimizing compiler which is much smaller than other optimizing compilers, and not much larger than non-optimizing compilers. Table 10.1 lists some data about optimizing C-compilers for the PowerPC architecture.

Compiler	source files	source lines	source (bytes)	executables	code (bytes)
GNU CC 2.6	179	304842	8858307	gcc, cpp, cc1	2201752
IBM XLC 1.3	n.a.	n.a.	n.a.	xlc, xlcentry	3529600

Table 10.1: Optimizing C-compilers for the PowerPC architecture.

Compared to these compilers, Oberon-2 compilers are extremely small. The non-optimizing Oberon-2 compiler POP2 consists of nine modules, totaling slightly above eight thousand source lines and 150 kBytes object code (Table 10.2). POP2's source code is roughly thirty times smaller than the one of GNU CC, and its object code is 15 and 23 times shorter than the one of GNU CC and XLC, respectively. Note that each compiler has been compiled by itself, so theoptimizing compilers have benefited from the code size reduction due to their own optimizations.

The optimizing Oberon-2 compiler OOC2 shares the front-end with POP2, and thus there are no significant differences between the two compilers in the front-end. Table 10.3 shows the corresponding data for OOC2. The code emitter has been replaced by module OOCD implementing the GSA data structure, and the code generator has been rewritten to construct GSA form.

Module	function	source lines	source (bytes)	code (bytes)
POPM	Machine Interface	390	13936	4856
POPS	Scanner	298	10064	6480
POPT	Table Handler	872	35488	13296
POPB	Tree Builder	1478	52579	29312
POPP	Parser	1020	34685	21636
	front-end	*4058*	*146752*	*75580*
POPV	Tree Traversal	874	32100	13296
POPL	Code Emitter	870	28619	10908
POPC	Code Generator	2154	86994	48116
Compiler	Driver Program	186	6479	3576
	Total	**8142**	**300944**	**151476**

Table 10.2: The non-optimizing Oberon-2 compiler POP2.

Module	function	source lines	source (bytes)	code (bytes)
OOCM	Machine Interface	403	15230	4868
OOCS	Scanner	298	10006	6480
OOCT	Table Handler	840	35282	14892
OOCB	Tree Builder	1467	51730	33576
OOCP	Parser	1011	34281	22252
OOCV	Tree Traversal	615	24336	12720
OOCD	GSA Data Types	413	18349	5868
OOCC	GSA Code Generator	1588	74710	40340
OOC2	Driver Program	224	7837	5120
	Total	**6859**	**271761**	**146116**

Table 10.3: The front-end of the optimizing Oberon-2 compiler OOC2.

Considering that not all Oberon features have been implemented in OOC2 yet, the sizes of the two code generators are remarkably similar. The added complexity due to the algorithms generating GSA form is well consumed by simplifications in the code generator. OOC2's code generator does not have to deal with register allocation and with optimizing certain code patterns, making it easier to understand than the one of POP2.

OOC2 currently builds upon an abstract syntax tree, but as described in Chapter 7, this intermediate step is by no means required. The front-end could directly generate GSA form, omitting most of module OOCB and parts of OOCV.

Module	function	source lines	source (bytes)	code (bytes)
OOCI	Inlining	127	5007	3188
OOCCP	Constant Propagation	372	12798	6360
OOCVN	Value Numbering	256	8628	4128
OOCDC	Dead Code Elimination	98	3031	1312
OOCSR	Strength Reduction	873	35834	23008
OOCPH	Peephole Optimizations	62	2730	2044
	Total	**1788**	**68028**	**40040**

Table 10.4: Optimization algorithms in OOC2.

Table 10.4 lists the optimization algorithms of OOC2. Copy propagation has been integrated with the value numbering algorithm. Except for module OOCSR, the algorithms are remarkably small. This can be attributed to the properties of GSA form, which simplify both analysis and transformations of the code. Most of the complexity of the strength-reduction algorithm is due to dealing with machine properties, e.g. selecting which parts of a reassociated sum should be consumed into an addressing mode and which ones should be propagated out of the loop.

Data about a simple back-end for OOC2 is presented in Table 10.5. This is the back-end which is also used in further measurements. It consists of an algorithm to adapt the code to machine restrictions, i.e. sizes of immediate fields in instructions or to instructions which do not accept immediate operands at all, and a simple local backwards-scheduler in module OOCMC. Module OOCRA implements a simplistic hierarchical graph coloring register allocator, which cannot spill variables to memory. The code emitter generates the bit-patterns corresponding to the PowerPC instructions and writes the object file.

Combining the front-end, the optimization algorithms, and the simple back-end, OOC2 is slightly above 10500 lines of code or 415 kBytes of source code and 212 kBytes of object code (Table 10.6). The object code has been generated by POP2, and a significant reduction can be expected by compiling OOC2 with an optimizing compiler.

Module	function	source lines	source (bytes)	code (bytes)
OOCMC	Machine Code Generator	479	17864	7792
OOCRA	Register Allocator	662	25134	6176
OOCEC	Code Emitter	827	31932	11952
	Total	**1968**	**74930**	**25920**

Table 10.5: A simple back-end for OOC2.

subsystem	source lines	source (bytes)	code (bytes)
front-end	6859	271761	146116
optimizations	1788	68028	40040
back-end	1968	74930	25920
Total	**10615**	**414719**	**212076**

Table 10.6: OOC2 with the simple back-end.

Jürg Bolliger has implemented a trace scheduler for OOC2 [Boll94], about which some numbers are presented in Table 10.7. This scheduler has a much better functionality than the one mentioned above: It allows exchanging a machine model, delivers branch-prediction information which could be exploited when actually emitting code, and generates better schedules. However, its size being more than five times the one of the simple scheduler is an indication of how easily optimization algorithms can become overweight.

Jakob Magun developed a register allocator combining the concepts of cyclic interval graphs and hierarchical graph coloring [Magun94]. Like the trace scheduler, it offers better functionality than the simple back-end, but is also significantly larger (Table 10.8). The trace scheduler and this register allocator could replace modules OOCMC and OOCRA in the back-end and further improve the code quality achieved by OOC2.

Module	function	source lines	source (bytes)	code (bytes)
OOCSD	Scheduler Data	186	9110	2636
OOCRIOS1	RIOS-1 Machine Model	158	8550	2544
OOCBP	Branch Prediction	442	21909	8728
OOCFS	Forward Scheduler	642	33805	13676
OOCTS	Trace Scheduler	981	51324	22620
	Total	**2409**	**124698**	**50204**

Table 10.7: A trace scheduler for OOC2.

Module	function	source lines	source (bytes)	code (bytes)
OOCRAX	Register Allocator	**1922**	**92483**	**41996**

Table 10.8: An improved register allocator for OOC2.

10.2 Compilation Time

We have measured the compilation time of different compilers on a test program, which comprises several of the Hennessy benchmarks. The C-compilers have been run with both optimizations turned on and turned off. They have been instructed to just compile and generate an object file, so that the linking

Compiler	time (ms)	factor
IBM XLC 1.3	1200	5.94
IBM XLC 1.3 –O3 –Q	71200	352.48
GNU CC 2.6	3300	16.34
GNU CC 2.6 –O3 –finline–functions	7100	35.15
POP2	202	1.00
OOC2	871	4.31

Table 10.9: Compilation times of different compilers. The C-compilers are instructed to use all optimizations (–O3) and to perform inlining (–Q and finline–functions, respectively). OOC2 performs all optimizations by default.

time is not included in the numbers. Table 10.9 presents an overview of the compilation times, all measured on an IBM RS/6000 Model 250 (66 MHz PowerPC 601) running AIX 3.2.5. The compiled program is our benchmark of five algorithms, which consists of 283 lines of source code in Oberon and 291 lines in C. The times given correspond to user time as opposed to elapsed time, excluding disk access times and times related to context switching. This favors the C-compilers, which tend to perform more disk I/O, and the non-optimizing compilers over the optimizing ones, as optimizations do not perform I/O.

OOC2 is between four and five times slower than the non-optimizing Oberon-2 compiler POP2, but still faster than the C-compilers with all optimizations turned off. GNU CC seems to have a pretty inefficient front-end or back-end, so that the added optimizations only double the compilation time. The IBM XLC compiler adds the most optimizations and runs almost sixty times slower when all optimizations are turned on.

The compilation time of OOC2 can be further split up into the times spent in individual parts of the compiler. Table 10.10 shows these numbers.

Task	time (ms)	%
Front End	151	17
Generate GSA	142	16
Constant Propagation	59	7
Value Numbering	126	14
Dead Code Elimination	14	2
Strength Reduction	66	8
Peephole Optimizations	24	3
Instruction Scheduling	76	9
Register Allocation	147	17
Code Emission	66	8
Total	**871**	**100**

Table 10.10: Compilation time of OOC2.

The front-end time includes everything until the abstract syntax tree has been built, and is the same as in POP2. About the same amount of time is spent in constructing GSA form from the tree. We have found that about 60% of this time is spent in the memory allocator, which is called several times for every instruction. Reducing the number of distinct memory blocks in the GSA representation or implementing an improved memory allocator may reduce this time by a large margin.

Among the optimizations, value numbering, instruction scheduling, and most notably register allocation are the most expensive steps. The latter two cannot be omitted, thus only minor speed improvements can be expected by allowing to turn off optimizations. By turning off the other optimizations, more code would have to be scheduled and to be dealt with in the register allocator, so that the compilation time may even increase.

It should be noted, however, that the compiler has been compiled using POP2, and compiling it by an optimizing compiler will improve its speed. OOC2 does not yet support the full Oberon-2 language, so that it cannot compile itself. The missing features are Case- and Loop-statements, procedure variables and type-bound procedures, and spilling in the register allocator.

10.3 Code Quality

We have measured the code quality produced by the different compilers. There are two aspects of code quality: Code size and execution speed. In terms of code size, both Oberon compilers achieve remarkably small numbers, as listed in Table 10.11.

Compiler	code size	factor
IBM XLC 1.3	3072	1.15
IBM XLC 1.3 –O3 –Q	7448	2.79
GNU CC 2.6	3112	1.16
GNU CC 2.6 –O3 –finline–functions	2504	0.94
POP2	2672	1.00
OOC2	2336	0.87

Table 10.11: Code size achieved by the different compilers.

POP2 generates the smallest code among the non-optimizing compilers, on which OOC2 improves by 14%. The IBM XLC compiler more than doubles the amount of code when all optimizations are turned on. This is due to aggressive procedure inlining and loop unrolling. GCC also unrolls loops and inlines procedures, but too a smaller extent than XLC, and achieves a code size reduction by optimizing. OOC2 does not include any optimizations which increase the code size except for procedure inlining, which was only used on very small procedures.

Among the non-optimizing compilers, POP2 also generates the fastest code. This can be mostly attributed to POP2 allocating variables in registers, while the others only keep variables in registers for one basic block. The execution time of our sample programs on an IBM RS/6000 Model 250 (66 MHz PowerPC 601) is listed in Table 10.12. *Perm* is a heavily recursive program generating all permutations of a number with seven digits. *Intmm* multiplies two integer matrices, while *Mm* multiplies two REAL matrices. *Quick* and *Bubble* sort an integer array using Quicksort and Bubblesort, respectively.

Program	XLC 1.3	GNU CC 2.6	POP2
Perm	43	62	36
Intmm	44	59	35
Mm	50	66	31
Quick	36	53	25
Bubble	66	114	56

Table 10.12: Execution times for the non-optimized code (in ms).

With all optimizations turned on, these execution times decrease by large margins for the C-compilers. The effects are less dramatic in case of the optimizing Oberon compiler OOC2, as the code quality of the non-optimizing compiler is already good. However, OOC2 can keep up with GNU CC 2.6 in most respects, as shown in Table 10.13.

Program	XLC 1.3	GNU CC 2.6	OOC2
Perm	20	25	23
Intmm	12	17	17
Mm	13	19	18
Quick	15	18	20
Bubble	14	22	25

Table 10.13: Execution times for the optimized code (in ms).

The better execution times of GNU CC code compared to OOC2 code on *Quick* and *Bubble* can be attributed to loop unrolling. XLC improves the code further by yet larger amounts of loop unrolling, by making use of load-and-update as well as store-and-update instructions, and by using advanced instruction scheduling techniques. As we have seen, these optimizations come at a high cost of compilation, even though we are convinced that similar results can be achieved with smaller effort.

11 Summary and Conclusions

This last chapter summarizes what has been achieved, and outlines areas for further research, both in the field of compilers and in the field of programming languages.

11.1 Summary

We have discussed recent developments in computer architecture, and how they affect the execution time of programs. *RISC architectures* provide relatively simple instructions only, out of which more complex operations can be built. They rely on compilers to customize code patterns to the surrounding context, e.g. to avoid unneccessary recomputations, which were hidden and therefore seemed to be for free in more complex architectures. *Pipelining* and *superscalar microarchitectures* allow to overlap the execution of multiple instructions, but require the compiler to order instructions in such a way, that there are no dependencies between instructions to be executed concurrently. *Caches* provide a mechanism to hide the widening gap between processor clock speed and main memory speed, but rely on the program to exhibit large amounts of temporal and spatial locality. Again, the compiler can try to improve upon the locality in order to enhance performance on cached systems. Thus, compilers play an important role in achieving good performance on modern machines.

We have presented a framework in which a compiler can perform the corresponding code improvements, which are commonly referred to as *optimizations*. Our framework is based on our intermediate representation called *guarded single-assignment (GSA) form*, which combines the best features of high-level and low-level representations, and which allows close integration of data-flow and control-flow. It makes all dependencies between instructions explicit, and allows to implement efficient and powerful optimization algorithms. Aggressive instruction scheduling algorithms are well-supported by several means: Only essential dependencies between instructions are modeled, and control-flow is expressed using guarded statements. The latter allows control-structures to be treated like 'large' instructions to be scheduled, or to implement scheduling algorithms centered around predicated execution, which it resembles.

We have developed single-pass algorithms to generate GSA form for source programs in structured programming languages. The algorithms can be integrated with scanning and parsing, so that no other intermediate representation of the program has to be constructed by the compiler. The remaining forms of unstructured control-flow in languages like Oberon, namely the Exit- and the Return-statement, can be accommodated by rewriting them into structured form while parsing the source text.

Using these results, we have implemented a prototypical optimizing compiler for a subset of Oberon-2, which includes most common scalar optimization algorithms. It achieves a code quality competitive with state-of-the-art optimizing compilers, still it is not much more complex than single-pass compilers. In fact, the compiler *is more than an order of magnitude smaller than competing optimizing compilers*.

11.2 Implications on Programming Language Design

The semantics of the source program must be accurately reflected in the intermediate program representation and in the final object code. The more the definition of the source language is based on sound concepts, the simpler become the intermediate representation, the translation from the source program, and the transformation into executable machine code. Moreover, in order to allow for different implementation options, the source language should not overspecify semantics. Freedom in how a particular construct can be translated can also be exploited by optimizing compilers to choose a code pattern which fits the target machine well.

Oberon has proven to be a programming language which can be well accommodated in optimizing compilers. It avoids two problems present in languages like C and Fortran: (1) It does not contain a Goto-statement, which allows for arbitrary control-flow, and (2) its strong type system reduces the aliasing problems encountered. A lot of compiler research in the last decade has tried to circumvent these problems with Fortran and C, and could have been avoided by mending the language.

However, Oberon could still be improved in some respects. Unstructured control-flow in the form of Exit- and Return-statements must be accomodated by translating it into structured form. We have found unstructured control-flow to be rare in today's programming practice [Bran94]. Dropping it altogether would not cause any major inconveniences to the programmer. Unstructured control-flow is often encountered in places where the programmer tried to optimize the branching structure of the program. This task is better left to the compiler.

Aliasing of data in Oberon is found in conjunction with pointers or VAR-parameters. The latter allows to create a reference to arbitrary objects of a given type, even within other objects. For example, a VAR-parameter of type INTEGER can be an alias to all variables, record fields or array elements of type INTEGER in outer scopes, including the heap. The programmer usually does not want to create an alias when using a VAR-parameter, but rather provides a destination for a result or avoids copying a large data structure. We believe that the behaviour of programs is often unexpected or even wrong when actual aliasing occurs, i.e. when the same memory cell is accessed using two different names. We thus propose to omit VAR-parameters from future languages. The Ada programming language [Ada83] provides *in-*, *out-*, and *inout*-parameters but no VAR-parameters. The language definition explicitly allows *call-by-reference* or *copy-in-copy-out* as parameter-passing mechanisms, and programs whose result depends on the mechanism used are considered erroneous (sections 6.2.7 and 6.2.13). A similar mechanism may be used in future programming languges.

A lot of research has been directed towards extracting parallelism from sequential code. Our optimizing compiler extracts parallelism between scalar operations by building a data-flow graph, but is not able to extract parallelism between operations on a single array or on complex data structures with pointers. More parallelism could be found by adding array dependence analysis and techniques to analyze code on dynamic data structures. As an alternative, the programming language could allow the programmer to explicitly express parallelism, and to describe data structures and operations on it in a 'more parallel' way. As is, the Oberon programming language requires the programmer to specify a total ordering of operations, which often results in an overspecification. The work of Griesemer on his programming language Oberon-V [Grie93] could serve as a starting point in this direction.

11.3 Future Work

There are two major directions of future research based on our work. The principles developed can be applied to other environments, and some minor problems of OOC2 could be solved.

The set of optimizations implemented in OOC2 turned out to be well-supported by our intermediate representation. Adapting other optimization techniques to GSA form and evaluating their performance could yield interesting feedback to the designer of intermediate programming representations. Morever, OOC2 uses a machine-dependent intermediate representation in order to reduce the number of transformation steps and in order to allow optimizations to improve the actual machine code. We believe that due to the similarities between different RISC processors, the compiler should be easily portable to other RISC architectures. This claim has still to be proven by porting the compiler to a different target architecture. Finally, we believe that several of the principles we developed could also be applied to the translation of other programming languages, and doing so would be a rewarding project.

Aliasing turned out to be bigger a problem than we originally thought, and is probably the weakest point in OOC2. Our model of aliasing is an optimistic one, where the compiler assumes that variables do not alias and then adds *MayAlias* nodes where it cannot prove it. A pessimistic approach as described by Weise et al. [WCES94] may be more practical. It treats all references as being to one global store, and then selectively modifies references which can be shown to be free of aliasing effects. Implementing a pessimistic model and evaluating its benefits over the current model would yield interesting feedback to the compiler designer.

11.4 Conclusions

The overall conclusion which can be drawn from this work is that optimizing compilers do not have to be as large and complex as they used to be. By developing principles targeted towards programming languages with clean semantics, and by restricting the compiler to the essential task of an optimizing compiler – namely emitting code of high quality for one target architecture – the complexity of optimizing compilers can be reduced by an order of magnitude. At the same time, their run-time is reduced by an order of magnitude as well.

The reduction in complexity makes optimizing compilers not only more easily understandable and teachable, but also allows such compilers to be built by small workforces. Moreover, reducing complexity has a positive impact on the quality of any program, so we can expect more reliable compilers in the future.

References

[Ada83] Department of Defense, United States of America. *Military Standard, Ada Programming Language*. ANSI/MIL-STD-1815A. January 1983.

[AiNi91] A. Aiken and A. Nicolau. *A Realistic Resource-Constrained Software Pipelining Algorithm*. In *Languages and Compilers for Parallel Processing*. MIT Press. 1991.

[AKPW83] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. *Conversion of Control Dependence to Data Dependence*. In *Conference Record of the 10th Annual Symposium on Principles of Programming Languages*. Austin, Texas. January 1983.

[AlCo76] F. Allen and J. Cocke. *A Program Data Flow Analysis Procedure*. In *Communications of the ACM*. Vol. 19, No. 3. March 1976.

[AlWZ88] B. Alpern, M. Wegman, and F. Zadeck. *Detecting Equality of Variables in Programs*. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*. San Diego, California. January 1988.

[AlZa94] B. Alpern and K. Zadeck. *Value Numbering*. In *Optimization in Compilers*, edited by F. Allen, B. Rosen, and K. Zadeck. To be published by ACM Press.

[Amda67] G. Amdahl. *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. In *Proceedings of the AFIPS 1967 Spring Joint Computer Conference*. Atlantic City, New Jersey. April 1967.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1986. ISBN 0–201–10088–6.

[AuHo82] M. Auslander and M. Hopkins. *An Overview of the PL8 Compiler*. In *Proceedings of the ACM SIGPLAN '82 Conference on Programming Language Design and Implementation*. Boston, Massachusetts. June 1982.

[BaHo92] T. Ball and S. Horwitz. *Constructing Control Flow from Control Dependence*. Technical Report No. 1091, University of Wisconsin, Madison. June 1992.

[BaLa93] T. Ball and J. Larus. *Branch Prediction for Free*. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. Albuquerque, New Mexico. June 1993.

[BaMO90] R. Ballance, A. Maccabe, and K. Ottenstein. *The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages*. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*. White Plains, New York. June 1990.

[Baner88] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers. 1988. ISBN 0–89838–289–0.

[BCFT92] M. Brandis, R. Crelier, M. Franz, and J. Templ. *The Oberon System Family*. Technical Report 174, Departement Informatik, ETH Zürich. April 1992.

[BeRo91] D. Bernstein and M. Rodeh. *Global Instruction Scheduling for Superscalar Machines*. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*. Toronto, Canada. June 1991.

[BGOCF93] A. Böhm, D. Grit, R. Oldehoeft, D. Cann, and J. Feo. *SISAL Reference Manual, Language Version 2.0*. Computer Science Department, Colorado State University and Computing Research Group, Lawrence Livermore National Laboratory. 1993.

[Boll94] J. Bolliger. *Aggressive Instruction Scheduling*. ETH Informatik Diploma Thesis, Institut für Computersysteme, ETH Zürich. March 1994.

[Bran94] M. Brandis. *Building an Optimizing Compiler for Oberon: Implications on Programming Language Design*. In *Advances in Modular Languages*, Proceedings of the Joint Modular Languages Conference, Ulm, Germany. September 1994. Universitätsverlag Ulm, ISBN 3–89559–220–X.

[BrCo94] P. Briggs and K. Cooper. *Effective Partial Redundancy Elimination*. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. Orlando, Florida. June 1994.

[Brig92] P. Briggs. *Register Allocation via Graph Coloring*. Ph.D. thesis, Rice University, Houston, Texas. Available as Technical Report Rice COMP TR92–183. 1992.

[BrMö94] M. Brandis and H. Mössenböck. *Single-Pass Generation of Static Single Assignment Form for Structured Languages*. In *ACM Transactions on Programming Languages and Systems*; 1994.

[CACCHM81] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. *Register Allocation via Coloring*. In *Computer Languages*, Vol. 6, pp. 47–57. 1981.

[CaCK90] D. Callahan, S. Carr, and K. Kennedy. *Improving Register Allocation for Subscripted Variables*. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*. White Plains, New York. June 1990.

[CaKo91] D. Callahan and B. Koblenz. *Register Allocation via Hierarchical Graph Coloring*. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*. Toronto, Canada. June 1991.

[CFRWZ91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. In *ACM Transactions on Programming Languages and Systems*. Vol. 13, No. 4. October 1991.

[ChHe90] F. Chow and J. Hennessy. *The Priority-Based Coloring Approach to Register Allocation*. In *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 4. October 1990.

[CoGo94] D. Corney and J. Gough. *Type Test Elimination Using Typeflow Analysis*. In *Proceedings of the International Conference on Programming Languages and System Architectures*. Zurich, Switzerland. Published as Springer Lecture Notes in Computer Science No. 782. March 1994.

[CoSch70] J. Cocke and J. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University. April 1970.

[CyGe93] R. Cytron and R. Gershbein. *Efficient Accommodation of May-Alias Information in SSA Form*. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. Albuquerque, New Mexico. June 1993.

[Crel91] R. Crelier. *OP2: A Portable Oberon-2 Compiler*. In *Proceedings of the Second International Modula-2 Conference*. Loughborough, UK. 1991.

[CyLZ86] R. Cytron, A. Lowry, and F. Zadeck. *Code Motion of Control Structures in High-Level Languages*. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*; 1986.

[DEC92] Digital Equipment Corporation. *Alpha Architecture Handbook*. 1992.

[DHB89] J. Dehnert, P. Hsu, and J. Bratt. *Overlapped Loop Support in the Cydra 5*. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, ACM Press, New York. April 1989.

[Dijk76] E. Dijkstra. *A Discipline of Programming*. Prentice Hall 1976. ISBN 0–13–215871–X.

[Ellis85] J. Ellis. Bulldog: *A Compiler for VLIW Architectures*. MIT Press 1985. ISBN 0–262–05034–X.

[ErHe93] A. Erosa and L. Hendren. *Taming Control Flow: A Structured Approach to Eliminating Goto Statements*. ACAPS Technical Memo 76, School of Computer Science, McGill University. Montreal, Canada. September 1993.

[FeOW87] J. Ferrante, K. Ottenstein, and J. Warren. *The Program Dependence Graph and Its Use in Optimization*. In *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3. July 1987.

[Franz94] M. Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. Doctoral Thesis Diss. ETH No. 10497. Zurich, March 1994.

[GiMu86] P. Gibbons and S. Muchnick. *Efficient Instruction Scheduling for a Pipelined Architecture*. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*. Palo Alto, 1986.

[Grie92] R. Griesemer. *Scheduling Instructions by Direct Placement*. In *Proceedings of the 4th International Conference on Compiler Construction*. Lecture Notes in Computer Science 641, Springer. 1992.

[Grie93] R. Griesemer. *A Programming Language for Vector Computers*. Doctoral Thesis Diss. ETH No. 10277. Zurich, August 1993.

[Half94] T. Halfhill. *80x86 wars*. BYTE, June 1994.

[HDEGSS92] L. Hendren, C. Donawa, M. Emami, G. Gao, J. Sridharan, and B. Sridharan. *Designing the McCAT Compiler based on a Family of Structured Intermediate Representations*. In *Conference Record of the 5th Workshop on Languages and Compilers for Parallel Computing*. Department of Computer Science, Yale University. New Haven, Conneticut. August 1992.

[HePa90] J. Hennessy and D. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufman Publishers, San Mateo, California. 1990. ISBN 1–55860–069–8.

[HeGr83] J. Hennessy and T. Gross. *Postpass Code Optimization of Pipeline Constraints*. In *ACM Transactions on Programming Languages and Systems*. Vol. 5, No. 3. July 1983.

[HGAM92] L. Hendren, G. Gao, E. Altman, and C. Mukerji. *A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs*. In *Proceedings of the 4th International Conference on Compiler Construction*. Lecture Notes in Computer Science 641, Springer. 1992.

[HGAM93] L. Hendren, G. Gao, E. Altman, and C. Mukerji. *Register Allocation using Cyclic Interval Graphs: A New Approach to an Old Problem*. ACAPS Memo No. 33. Available from FTP-server *ftp.cs.mcgill.ca*. 1993.

[IBM90a] IBM. *IBM RISC System/6000 Technology*. IBM Order Number SA23–2619–00.

[IBM90b] IBM. *IBM Journal of Research and Development*. Vol. 34, No. 1. January 1990.

[IBM90c] IBM. *POWER Processor Architecture Version 1.52*. February 1990.

[IBM90d] IBM. *XL C User's Guide*. April 1990. IBM Order Number SC09–1259–00.

[IBM94a] IBM. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufman Publishers, San Mateo, California. 1994. ISBN 1–55860–316–6.

[IBM94b] IBM. *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*. IBM Order Number SA23–2737–00.

[Jain91] S. Jain. *Circular Scheduling: A new Technique to Perform Software Pipelining*. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*. Toronto, Canada. June 1991.

[John90] W. Johnson. *Super-Scalar Processor Design*. Technical Report CSL-TR-89-383. Computer Systems Laboratory, Stanford University, California. June 1989.

[Jones93] N. Jones (editor). *Special Issue on Partial Evaluation*. Journal of Functional Programming. Vol. 3, No 4. 1993.

[Kane87] G. Kane. *MIPS R2000 RISC Architecture*. Prentice Hall. 1987.

[Knuth71] D. Knuth. *An Empirical Study of FORTRAN Programs*. In *Software Practice and Experience*, Vol. 1, 105–133. 1971.

[KnRS94] J. Knoop, O. Rüthing, and B. Steffen. *Partial Dead Code Elimination*. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. Orlando, Florida. June 1994.

[Lam88] M. Lam. *Software Pipelining: An Effective Scheduling Technique for VLIW Machines*. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*. Atlanta, Georgia. June 1988.

[Lam93] M. Lam. *Instruction Scheduling*. Tutorial Notes, *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. Albuquerque, New Mexico. June 1993.

[LaRW91] M. Lam, E. Rothberg, and M. Wolf. *The Cache Performance and Optimizations of Blocked Algorithms*. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. Palo Alto, California. April 1991.

[LeTa79] T. Lengauer and R. Tarjan. *A Fast Algorithm for Finding Dominators in a Flowgraph*. In *ACM Transactions on Programming Languages and Systems*. Vol. 1, No. 1. July 1979.

[MaEV92] N. Malik, R. Eickemeyer, and S. Vassiliadis. *Interlock Collapsing ALU for Increased Instruction-Level Parallelism*. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*. Portland, Oregon. December 1992.

[Magun94] J. Magun. *Hierarchical Register Allocation with Cyclic Interval Graphs*. Diploma Thesis, Institut für Computersysteme, ETH Zürich. March 1994.

[MLCHB92] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. *Effective Compiler Support for Predicated Execution Using the Hyperblock*. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*. Portland, Oregon. December 1992.

[MMZ94] P. Markstein, V. Markstein, and K. Zadeck. *Strength Reduction*. In *Optimization in Compilers*, edited by F. Allen, B. Rosen, and K. Zadeck. To be published by ACM Press.

[Möss93] H. Mössenböck. *Objektorientierte Programmierung in Oberon-*2. Springer Verlag, 1993. ISBN 3–540–55690–7.

[MöWi91] H. Mössenböck and N. Wirth. *The Programming Language Oberon–2*. In *Structured Programming*, 12, 170–195. 1991.

[Mult93] *The Multiflow Trace Scheduling Compiler*. In *The Journal of Supercomputing*. Vol. 7, Pages 51–142. 1993.

[PPC601] IBM Microelectronics, Motorola. *PowerPC 601 RISC Microprocessor User's Manual*. IBM Order Number 52G7484, Motorola Order Number MPC601UM/AD Rev 1. 1993.

[PPC604] IBM Microelectronics, Motorola. *PowerPC 604 RISC Microprocessor Technical Summary.* IBM Order Number MPR604TSU-02, Motorola Order Number MPC604/D. 1994.

[Pren92] D. Prener. Personal communications. 1992.

[Radin82] G. Radin. *The 801 Minicomputer*. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, ACM Press, Palo Alto, California. March 1982. Also published in *IBM Journal of Research and Development*. Vol. 27, No. 3. May 1983.

[Rau91] B. Rau. *Data Flow and Dependence Analysis for Instruction Level Parallelism*. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing*. Santa Clara, California. August 1991.

[ReWi92] M. Reiser and N. Wirth. *Programming In Oberon – Steps Beyond Pascal and Modula-2*. Addison Wesley 1992. ISBN 0–201–56543–9.

[RoWZ88] B. Rosen, M. Wegman, and F. Zadeck. *Global Value Numbers and Redundant Computations*. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*. San Diego, California. January 1988.

[RYYT89] B. Rau, D. Yen, W. Yen, and R. Towle. *The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Tradeoffs*. In *IEEE Computer*, Vol. 22, No. 1. January 1989.

[SmLH90] M. Smith, M. Lam, and M. Horowitz. *Boosting Beyond Static Scheduling in a Superscalar Processor*. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, ACM Press, New York. Published as Computer Architecture News, Vol. 18, No. 2. June 1990.

[SmHL92] M. Smith, M. Horowitz, and M. Lam. *Efficient superscalar performance through boosting*. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, ACM Press, New York. Published as SIGPLAN Notices, Vol. 27, No. 9. September 1992.

[Sark93] V. Sarkar. *Advanced Optimizations for Memory Hierarchies*. Tutorial Notes, *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. Albuquerque, New Mexico. June 1993.

[SeSo88] P. Sestoft and H. Sondergaard. *A Bibliography on Partial Evaluation*. In *ACM Sigplan Notices*. Vol. 23, No 2. February 1988.

[Sura93] R. Surati. *A Parallelizing Compiler Based on Partial Evaluation*. Technical Report AITR-1377, Artifical Intelligence Laboratory, Massachusetts Institute of Technology. July 1993.

[Stall94] R. Stallman. *Porting GNU CC*. Online Documentation of GNU CC 2.6. Free Software Foundation. 1994.

[WaLH93] N. Warter, D. Lavery, and W. Hwu. *The Benefit of Predicated Execution for Software Pipelining*. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, Architecture Track. Hawaii. January 1993.

[Wang94] K. Wang. *Precise Compile-Time Performance Prediction for Superscalar-Based Computers*. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. Orlando, Florida. June 1994.

[WCES94] D. Weise, R. Crew, M. Ernst, and B. Steensgaard. *Value Dependence Graphs: Representation Without Taxation*. In *Conference Record of the 21th Annual Symposium on Principles of Programming Languages*. January 1994.

[WeSmi94] S. Weiss and J. Smith. *POWER and PowerPC: Principles, Architecture, Implementation*. Morgan Kaufman Publishers, San Mateo, California. 1994. ISBN 1–55860–279–8.

[WeZa91] M. Wegman and F. Zadeck. *Constant Propagation with Conditional Branches*. In *ACM Transactions on Programming Languages and Systems*. Vol. 13, No. 2. April 1991.

[WHSB92] N. Warter, G. Haab, K. Subramanian, and J. Bockhaus. *Enhanced Modulo Scheduling for Loops with Conditional Branches*. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*. Portland, Oregon. December 1992.

[Wirth86a] N. Wirth. *Algorithmen und Datenstrukturen mit Modula-2*. B. G. Teubner Verlag, Stuttgart. 1986. ISBN 3–519–02260–5.

[Wirth86b] N. Wirth. *Compilerbau*. B. G. Teubner Verlag, Stuttgart. 1986. ISBN 3–519–32338–9.

[Wirth88] N. Wirth. *The Programming Language Oberon*. In *Software – Practice and Experience*. Vol. 18, No. 7. July 1988.

[WMGH94] T. Wagner, V. Maverick, S. Graham, and M. Harrison. *Accurate Static Estimators for Program Optimization*. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. Orlando, Florida. June 1994.

[WMHR93] N. Warter, S. Mahlke, W. Hwu, and B. Rau. *Reverse If-Conversion*. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. Albuquerque, New Mexico. June 1993.

[WoLa91] M. Wolf and M. Lam. *A Data Locality Optimization Algorithm*. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*. Toronto, Canada. June 1991.

[ZiCh90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley. 1990. ISBN 0–201–17560–6.