Code Generation Techniques for Irregular Architectures

Steven Bashford

Lehrstuhl Informatik XII
University of Dortmund

Report No. 596

November 1995

Code Generation Techniques for Irregular Architectures

Steven Bashford Lehrstuhl Informatik XII
University of Dortmund

Report No. 596

November 1995

Abstract

The fast development of many different ASIPs make demands of rapid availability of dedicated compilers. Fast retargeting is a major aspect, while fast compilation times are of minor importance. There are also new demands in the quality of the generated code. Irregular properties together with fine–grain parallelism given by a target architecture have to be effectively supported by the compiler. This report is focused on the traditional tasks of code generation — *code selection*, *register allocation*, and *instruction scheduling*. The major subject is to expose the tendencies of research of code generation techniques in recent years, and survey their features with regards to *support for irregular architectures*, *fine–grain parallelism*, *retargetability*, and *phase coupling*. The report outlines the preferable techniques involved in code generators. Features of irregular architectures being sufficiently supported by these techniques are examined. The insufficiencies with regards to irregular architectures are described and approaches to overcome them are described. The essential problems arising are due to mutual dependencies among the tasks of code generation. Thus, phase ordering problems and *phase coupling* approaches are a very important issue of the report. Retargeting is discussed with regards to retargetability of the described techniques, but also with regards to the quality of the generated code. Relations of structural and behavioral models are exposed, addressing the issue of supporting both, the design process of the target architecture and effective retargeting of all tasks of code generation.

Contents

Chapter 1

Introduction

Application specific integrated circuits (ASICs) were developed for giving highly specialized, effective hardware support for certain applications, e.g., audio and video applications. Application specific instruction set processors (ASIPs) are a trade–off between ASICs and general purpose processors, with some specific hardware support but still being programmable. Thus, developed for the effective support of specific applications, ASIPs contain a certain degree of flexibility, allowing late changes, error corrections, and readjustments to related applications. Hence, the increasing usage of ASIPs is motivated by advantages as late design modifications, design error correction, and reuseability. Optimizing compilers are demanded for the reuseability of the software developed for certain ASIPs. But as the development of ASIPs is getting faster, due to the support of sophisticated CAD–tools, such compilers must be rapidly readjustable to new target architectures. Only few algorithms are implemented on a certain ASIP, thus, the costs for the development of a dedicated compiler must be in relationship to its effective usage. In this context the importance of retargetable compilers is increasing. Specification models for utilizing the design process are of interest. I.e., description techniques are required, that support both, the effective retargeting of the compiler and the adaptation to the design and synthesis of the hardware. There are new demands in the quality of the generated code. High quality code is important for aspects like size of the hardware, power–usage, and in the context of timing constraints predicting a specific response time behavior for an application. Therefore, properties given by the target architecture should be effectively supported by the compiler. Irregular architectures with fine–grain parallelism lead to very strong mutual dependencies of the subtasks of the compiler. A strict ordering of these subtasks often restricts the quality of the produced code. Thus, an integration (phase coupling) of the tasks is another issue of interest. High compilation times are of minor concern in this contect.

1.1 Scope of the Report

The report is concerned with retargetable code generation with the aim of high quality code generation. It constitutes an outline of recent code improving techniques. Thereby, it is focused on the traditional tasks of code generation: *code selection*, *register allocation*, and *instruction scheduling*. The primary goal is to expose the following aspects:

- Describe the **basic concepts**, **preferable techniques** (incorporated in code generation), and the **tendencies of research**, with regards to code selection, register allocation, and instruction scheduling.

- Analysis of techniques with regards to supporting **irregular architectures with fine–grain parallelism**. Thereby, exploitation of the following provided features of the target machine are of major concern:

  – *irregular register sets* and register classes;

  – *complex data pathes* with restricted interconnection; i.e., not all register sets are connected with each other, and functional units do not have general access to all register sets;

  – *instruction level parallelism*.

  Issues of secondary concern, but also incorporated in this report, are:

  – timing constraints given by the target architecture;

  – exploitation of features provided by autoincrement/decrement registers.

  The basic problems, insufficiencies, and superimposed solutions to solve the problems are described.

- The essential problems arising are due to mutual dependencies among the tasks of code generation. Thus, **phase ordering problems** and **phase coupling** approaches are a very important issue of the report.

- **Retargeting** of code generation was mentioned to be an important subject; this item is discussed with regards to:

  – retargetability of the described techniques;

  – quality of the generated code;

  – relations of essential entities of the hardware and specification models.

  The last item addresses the issue of supporting the design process of the target architecture. This requires specification models that enable both

  – utilizing the design of hardware together with the synthesis based on the specification;

  – effective retargeting of all tasks of code generation.

  The relations of structural and behavioral (instruction set based) models are discussed.

- Aspects of **timing constraints**, enforcing a certain timing behavior of the algorithms (to be compiled), are itemized.

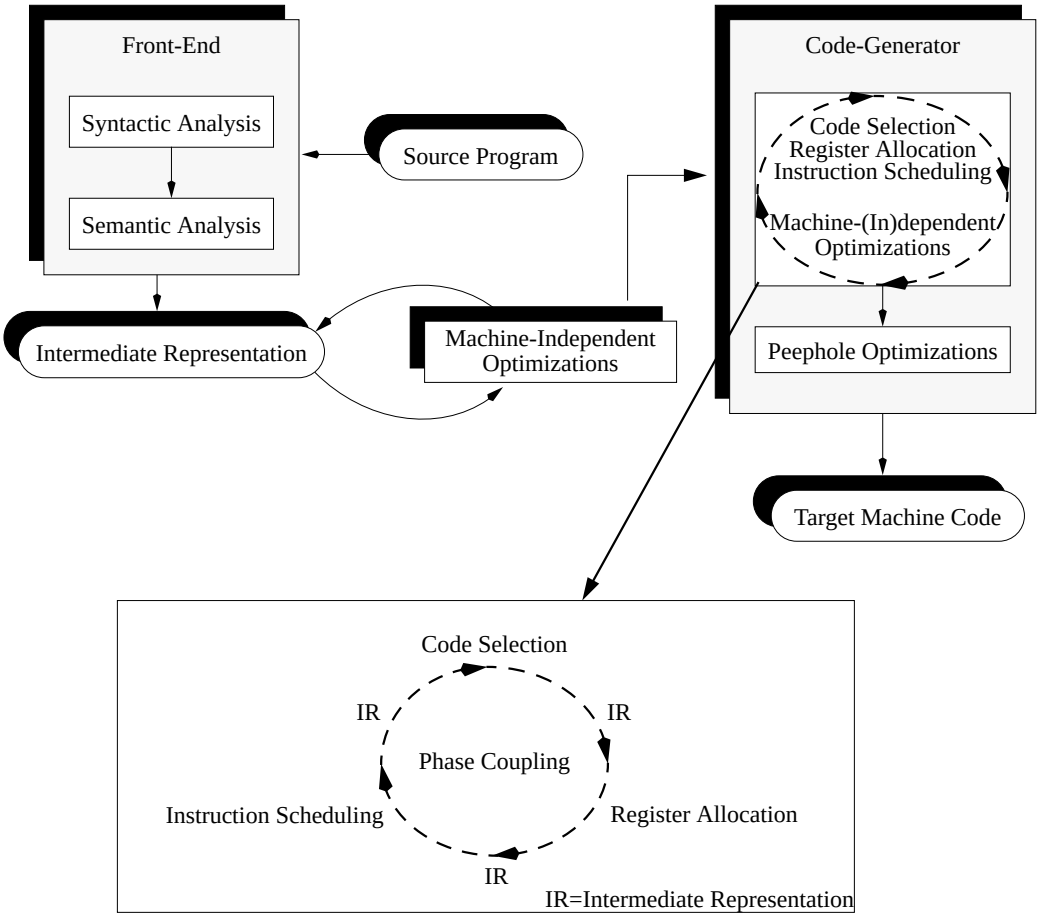


Figure 1.1: Phases of a Compiler

The report is not intended as an introduction to code generation. Knowledge about the basic foundations of code generation techniques is assumed (e.g., [ASU86]). However, introductions to the basic notions and concepts are given in each section. It enables the reader to class notions and concepts, and constitutes a basis for further detailed investigations.

This report constitutes no criticism on code generation techniques. All techniques examined were basically developed with certain classes of architectures in mind. Certainly, they produce very sophisticated results with regards to their suppositions. It is tried to emphasize the requirements and problems when irregular architectures with fine–grain parallelism are incorporated.

1.2 Structure of the Report

In figure 1.1 an overview of the compilation process is given. The front–end performs the syntactic and semantic analysis of the source program resulting in an intermediate representation of the source program. Some machine independent optimizations may follow, generating the final input to the back–end of the compiler. The code generator takes this input and generates *target machine code*, optionally optimized by a succeeding peephole optimization. The report

is organized as follows: chapter 2 is concerned with an outline of intermediate representations and their impact on code generation; a short summary of basic compiler optimizations is included. In chapter 3 the task of retargetable code generation is described. It introduces the basic terminology and points out the necessary issues of retargeting. Chapters 4 – 6 are related with the traditional code generation phases *code selection*, *register allocation*, and *instruction scheduling*. These sections are organized as follows: a short introduction of basic concepts is given; the *preferable techniques* developed in recent years are described; their *abilities/insufficiencies* in supporting features of irregular architectures and fine–grain parallelism are outlined; *approaches* to overcome the basic problems and drawbacks are described; the retargetability of techniques is discussed. Chapter 7 will describe the phase ordering problems in the context of irregular architectures and instruction level parallelism. It is concerned with drawbacks of the code generation tasks arising from strict decoupling and ordering of the code generation tasks. Recent phase coupling approaches are outlined. In chapter 8 a short summary of timing constraint aspects is given. Chapter 9 summarizes the major issues of the report.

Chapter 2

Intermediate Representations

In this chapter an outline of existing intermediate program representations is given, with priority on graph based representations. The basic notions will be introduced and special interest is addressed to graph based representations. For the interested reader this chapter serves as a source for further investigation by several references to recent approaches in this area. The chapter is no introduction to intermediate representations. The reader unfamiliar to intermediate representations is refered to [ASU86] or [WM95].

2.1 Introduction

As shown in figure 1.1, the process of compilation can be divided into the following phases:

- syntactical analysis
- semantical analysis
- program optimizations
- code generation

The first two phases check for the syntactical and semantical correctness of a source program, resulting in a certain *intermediate representation (IR)* of the program, being amenable to further program optimizations and code generation. The choice of an IR has profound effect on the design, complexity and implementation of optimizations in a compiler. Some of the well known program optimizations are:

- *Constant Folding:* Operands of an operation are all constants, therefore the result can be replaced by the computed constant result.
- *Copy Propagation:* Removing assignments between variables by using the original variable whenever possible.
- *Code Motion:* Operations are moved to program regions where they are less frequently executed, e.g. moving loop–invariant code out of loops.

- *Dead Code Elimination:* Elimination of instructions that are known to be never executed.

- *Common Subexpression Elimination:* Finding operations that compute the same result, keeping the original result available in a certain destination and substituting these operations by the destination (most prominent technique is value numbering).

- *Redundancy Elimination:* It determines instructions which compute the same result and eliminates superfluous recomputations.

- *Strength Reduction:* Reducing an expensive operation to a less expensive one.

- *Evaluation Order Determination:* Reordering of statements to reduce the amount of registers used to evaluate certain expressions.

- *Branch Chain Elimination:* Changing of branches that transfers control to another branch to branch directly to the destination of the second branch.

Good *intermediate representations (IR)* are of much interest regarding retargetable code generation and there are several desirable characteristics an intermediate representation should have, respectively:

- *Machine independence:* The IR should be suitable to a wide spectrum of architectures, which is an important aspect for using it in a retargetable code generator (and prevents the rewriting of the front–end for retargeting a compiler to a new target machine), while still utilizing a wide spectrum of optimizations.

  There are several levels of abstraction covering the representation of source level operators up to the description of primitive operations on the register transfer level. The usage of source level operators has several advantages, e.g. being easy to construct and easy to understand. It also offers the highest amount of portability. Since retargetability is the issue of interest this level is the most adequate to be used. Also, control flow can be represented at different levels. It can be represented by using the control structures of the source language or making the branching structure explicit by using conditional and unconditional jumps. A description of the advantages and disadvantages of the different abstraction levels and representations can be found in [Bra95].

- *Exploitation of parallelism:* It should be possible to easily extract potentially coarse grain (task level) and fine grain (instruction level) parallelism. A program representation should not only facilitate the detection of parallelism but also should easily enable program transformations that increase opportunities for parallelism.

- *Well–defined:* The IR should be well defined and should have a clear operational semantics making it usable for abstract interpretation (and verification).

- *Suitability for* subsequent integration of aspects of the target machine with the aim of enabling easy detection and exploitation of capabilities of the target architecture.

2.2 Concepts of Representation

Two basic concepts of representing programs are distinguished: the first one is the representation of a program by a sequence of abstract machine instructions. The second concept relies on graph based representations.

Abstract Machine Instructions Abstract machines are designed to simplify the compilation process for specific language classes like imperative languages, functional languages or logic programming languages. An abstract machine instruction performs complex tasks and often fully implements a high level language construct, e.g. procedure calls or memory management.

A customary intermediate language for imperative languages is the **3–address code** [ASU86]. Some of the frequently used instructions are

- *assignment statements* of the form `x := y bop z`, `x := op y` or `x:=y`, where `bop` represents a binary operator and `op` a unary operator;
- *indexed assignments* `x:=y[i]` or `x[i]:=y`;
- *unconditional jumps* `goto L`;
- *conditional jumps* `if x relop y goto L`.

There are also instructions for parameter passing and subroutine calls and adress–, and pointer assignments.

The *SECD–machine* and the *G–machine* [FH88] are abstract machines for the execution of functional languages. The *WAM (Warren Abstract Machine)* is the most frequently used machine model used for implementing logic programming languages like Prolog. For more detailed descriptions on abstract machines and abstract instruction sets see [WM95, Kog91].

Problems arise, when the level of abstraction is too high. Global data flow analysis can become very difficult when details that are necessary for detection of potentially optimizations are not explicitly presented.

Graph–Based Representations In graph based representations certain dependencies of program entities are made explicit, e.g. data flow or control flow. Therefore graph–based representations offer better possibilities for program analysis and program transformation. In these representations entities of the program are associated with nodes in a graph where dependencies represent edges between nodes.

In general a program represented by a sequence of abstract machine instructions can be transformed into a graph–based representation. But determining the dependencies becomes harder when instructions represent complex tasks.
There are features inherent to the represented language that also effect analysis–, and optimization techniques. Two very important and closely related features are:

Multi Assignment Form Imperative languages usually allow multiple assignments to the same variable. An IR allowing multiple assignments has impact on the analysis

methods used, e.g. a redefinition of a certain variable introduce an anti dependence that restricts the reordering of statements in the program.

Static Single Assignment Form Static Single Assignment (SSA) form was recently proposed by [CFR+89, CFR+91]. In an imperative programming language the same variable can be assigned more than one time. In SSA each variable is only defined once, therefore every redefinition of a variable and corresponding uses are renamed uniquely. When control flow is coalescing, more than one definition of a variable can reach a use of the variable. In this case the dummy function Φ is included, which selects one of its parameters, depending on the control flow that was taken during runtime to reach the Φ–function. In the SSA all anti dependencies and output dependencies are eliminated.

Imperative languages with multi assignments can easily be transformed into SSA [CF87]. A detailed overview of SSA is given in [San94, pp.8–9]. There are also many extentions of SSA e.g. [Bra95].

2.3 Graph–Based Representations

There are many works concerned with graph–based intermediate representations which often results in different definitions of the same notions. Due to that fact, the definitions introduced here may differ from one or the other definitions given in other works. I tried to choose a common description frame–work for a compareable introduction of the entities and features of the intermediate representations stated.

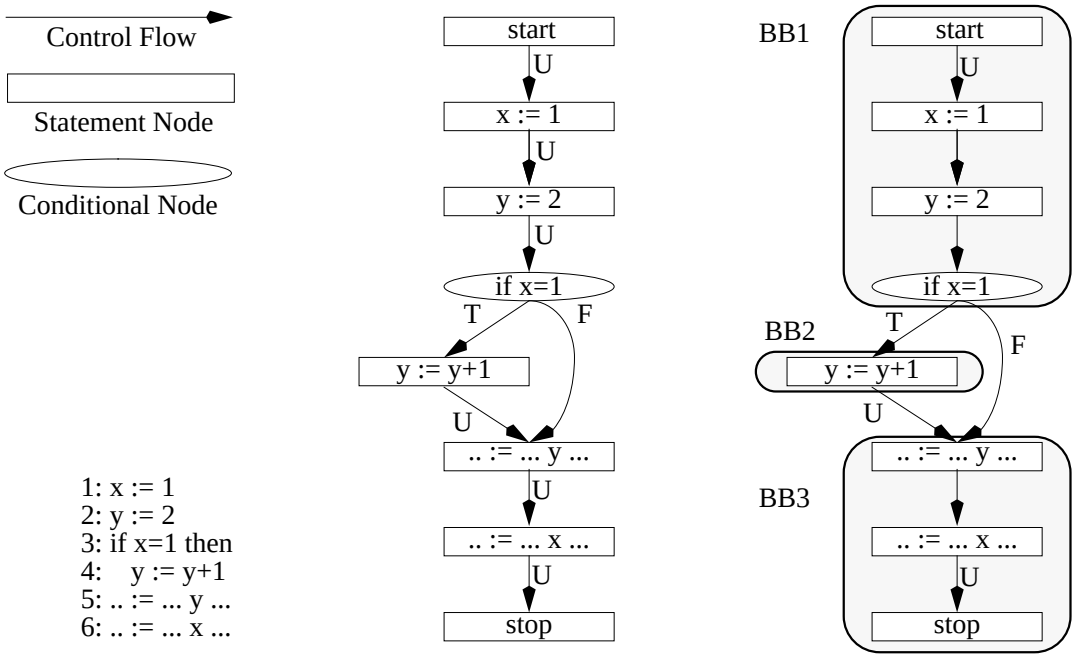
2.3.1 Terminology

In the following we assume that an **assignment statement** is of the form *x:=y op z,* and a **conditional expression** is a logical expression that evaluates to either of the values $true$ or $false$. A **definition of a variable v** is an assignment statement with destination v, i.e. v on its left hand side. The **use of variable** v is an occurance of v in the right hand side of an assignment statement or in a conditional expression.
A use of variable v is **reachable** by a definition of v if the execution of the definition may be followed by execution of the use of v without intervening execution of any other definition of variable v.

2.3.2 Control Flow Graph

The control flow graph directly reflects the branching structure of a program. There are several different definitions of the notion control flow graph. The definition introduced here differs from that given in [ASU86] and is similar to that given in [SS93].

Definition 2.3.1 *The* **contol flow graph** *(CFG) is a directed labeled graph* $CFG = (N, E_{cf}, \tau)$, *where:*



(a) Program (b) Statement Level (c) Basic Block Level

Figure 2.1: Control Flow Graph

- *N is a finite set of labeled nodes, representing either an assignment statement or a conditional expression. There are two special, non–labeled nodes $n_{start} \in N$ and $n_{stop} \in N$.*
- *$E_{cf} \subset N \times N \times \{T, F, U\}$ is a set of labeled edges, representing possible transfer of control between the nodes.*
- *$\tau : N \rightarrow \{START, STOP, COMPUTE, PREDICATE\}$ is a type function identifying the type of a node. $\tau(n_{start}) = START$ and $\tau(n_{stop}) = STOP$.*

For every node $n \in N$ there exists a directed path from n_{start} to n and there exists a directed path from n to n_{stop}. Nodes of type $COMPUTE$ are labeled with an assignment statement and have only one unique successor with the corresponding edge labeled with U (unconditional). Nodes of type $PREDICATE$ are labeled with a conditional expression. They always have two successors and outgoing edges are labeled with T and F, denoting the flow of control in case the conditional expression evaluates to $true$ and $false$, respectively.

In [ASU86] the CFG is constructed with nodes representing basic blocks. In this case an edge from node x to node y exists, if the last instruction of block x is a conditional or unconditional jump to the first instruction of block y. In figure 2.1 an example program with its corresponding CFG representation is shown. Figure 2.1(c) shows an equivalent representation on basic block level. The following definition of basic blocks is based on the definition of CFGs:

Definition 2.3.2 *Given a $CFG = (N, E_{cf}, \tau)$. A* **basic block** *is a path $P = [n_1, \ldots, n_{max}]$ ($n_1, \ldots, n_{max} \in N$) of maximal length, such that at most node n_1 has more than one incoming edge, and at most n_{max} has more than one outgoing edge.*

A graph consisting of nodes corresponding to basic blocks and edges that denote control flow between basic block will be called **basic block graph**.
An advantage of CFGs is their compact representation and their easy operational semantics. Although being easy to implement, this approach has several drawbacks. Disadvantages arise in the context of optimizations where the usage of the CFG leads to unefficient implementations, i.e. information is passed throughout the complete control flow graph, even in program regions where it is not needed [PBJS90, JP93].

2.3.3 Control Dependence Graph

In contrast to the control flow graph, the control dependence graph explicitly shows the essential dependencies, i.e. the conditional expressions responsible for the execution of a statement, depending on the value of conditional expressions during program execution.
The notions introduced here are of great importance, e.g. in the transformation of a program to SSA and in the definitions for other IRs like program dependence graphs (see 2.3.5).

Definition 2.3.3 *A node x is a* **dominator** *of a node y, denoted by $x \triangle_d y$ (x dominates y) iff every path from n_{start} to y contains x [ASU86]. A node always dominates itself.*

Definition 2.3.4 *The set of dominators of a node x form a chain. x is an* **immediate dominator** *of y iff $x \triangle_d y$ and*
$\neg\exists z : x \triangle_d z \wedge z \triangle_d y \wedge z \neq x$.

Definition 2.3.5 *The* **dominator tree** *of a CFG is a tree including the nodes of the CFG, n_{start} is the root. A dominator tree has edges between nodes x and y iff x is an immediate dominator of y.*

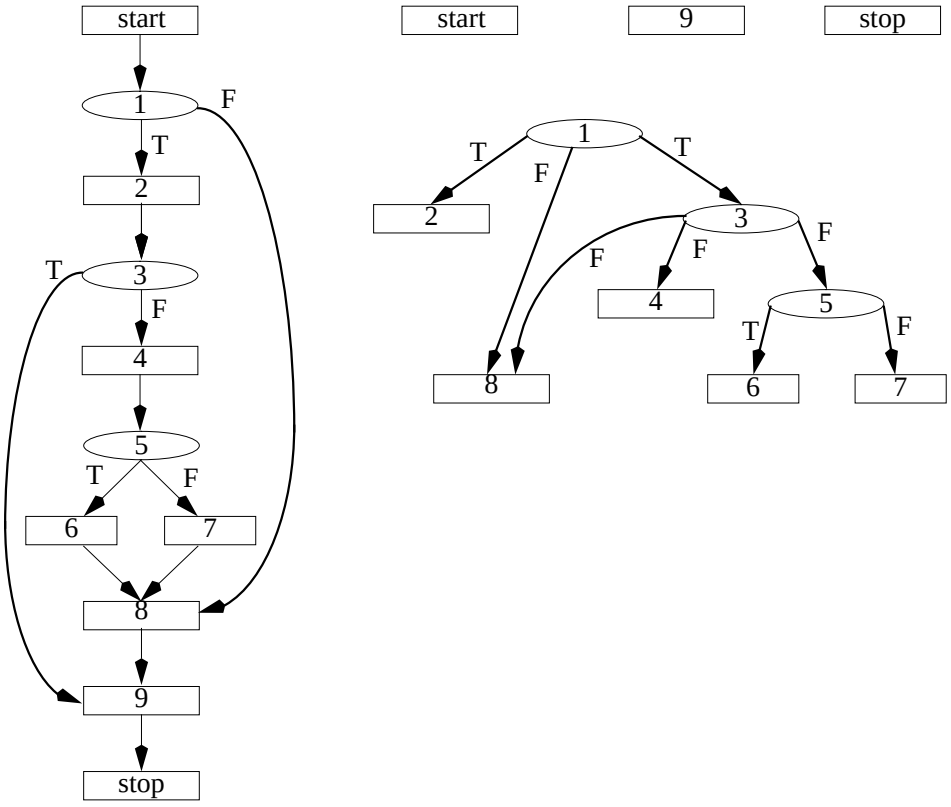
A node x **post–dominates** a node y, $x \triangle_p y$, iff every path from y to $STOP$ contains x. A node never post–dominates itself. The reflexive closure of the post–dominance is denoted by $\overline{\triangle}_p$. The least node in the chain of **post–dominators** of a certain node x is called the **immediate post–dominator** of x. The set of post–dominators of $x \neq n_{stop}$ is non–empty, hence all nodes except n_{stop} have an unique immediate post–dominator. The *post–dominator tree* is a directed graph rooted by n_{stop}, and an edge from node x to y denotes, that x is an immediate post–dominator of y. The post–dominator tree can be computed as the dominator tree over the reversed CFG.

Example 2.1:

> The node of type $START$ is a dominator of every node. In figure 2.1 the node of type $PREDICATE$ is an immediate dominator of the nodes associated with statement 4 and 5. Statement 4 is no dominator of statement 5. The node of type $STOP$ is a post–dominator of every node.

Definition 2.3.6 *Given a $CFG = (N, E_{cf}, \tau)$. Node x has* **control dependence** *on node y denoted $x\delta_c^a y$ iff*

1. $(x, a, tfu) \in E_{cf}$,



(a) Control Flow Graph (b) Control Dependence Graph

Figure 2.2: Control Dependence Graph

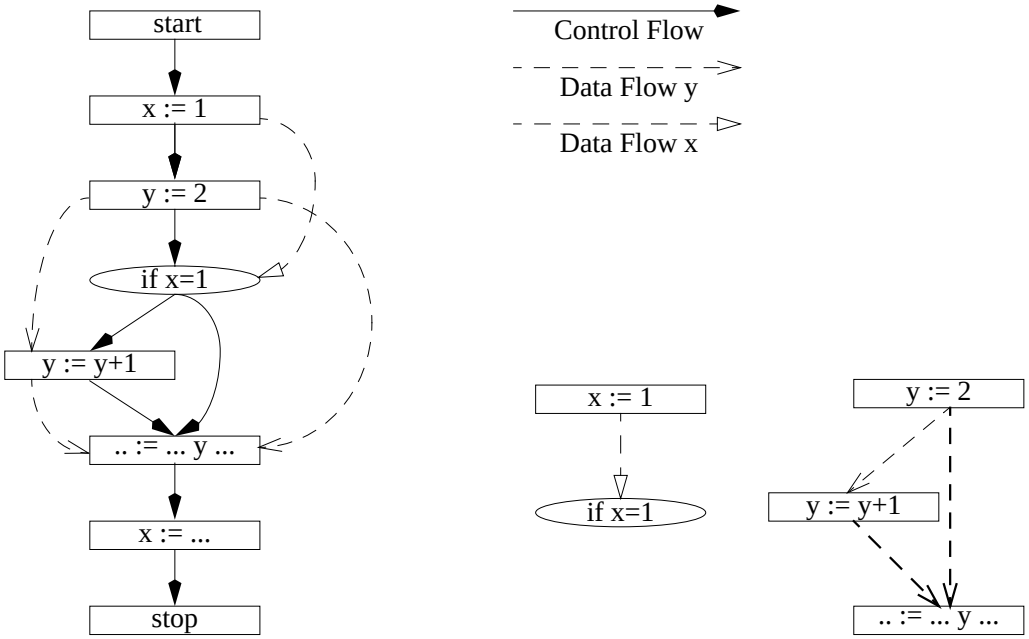
2. $\neg(y \triangle_p x)$, *i.e.* y *does not post–dominate* x, *and*

3. *there exists a non–empty path* $p = x, a, \ldots, y$, *such that for any* $z \in p$ *with* $z \neq x$, $z \neq y$: $y \triangle_p z$.

The index c *denotes that* δ_c *is a control dependence relation, to distinguish it from the data dependence relation* δ_d. *If* $x\delta_c^a y$, *then* $y\overline{\triangle_p}a$. $x\delta_c^a y$ *can also be stated as* y *is* control dependent *of* x.

Definition 2.3.7 *The* **control dependence graph** *(CDG) of a* $CFG = (N, E_{cf}, \tau)$ *is defined as a directed graph* $CDG = (N, E_{cd}, \tau)$ *with labeled edges, such that* $(x, y, tf) \in E_{cd}$ *iff* $x\delta_c^a y$ *and* $(x, a, tf) \in E_{cf}$ *and* $tf \in \{T, F\}$.

The source of a control dependence edge is a predicate node. Like in the CFG an edge from a predicate node is labeled with T or F, indicating the value of the predicate under which the statement at the sink of the edge will be executed. The construction of the CDG is described in [GP92].



(a) Control Flow Graph (b) Def-Use Chains

Figure 2.3: Def–Use Chains

Example 2.2:

In figure 2.2(b) an example of a CDG of the control flow graph in figure 2.2(a) is shown (the explicit contents of the nodes and the U–labels are omitted). Node 1 has control dependence on nodes 2 and 3, as these two nodes are only executed if the conditional expression of node 1 evaluates to $true$. Node 3 is not control dependent of node 2 because it post–dominates node 2. Node 9 is a post–dominator of every node except node $stop$ and itself. Therefore it is not control dependent of any node.

2.3.4 Def–Use Chains, Data Dependence Graphs and Data Flow Graphs

A very customary representation of **data flow** are *def–use chains. Def–use graphs* are graphs that have the same set of nodes as the CFG, where edges connect each definition of a variable to all uses of the variable [ASU86].

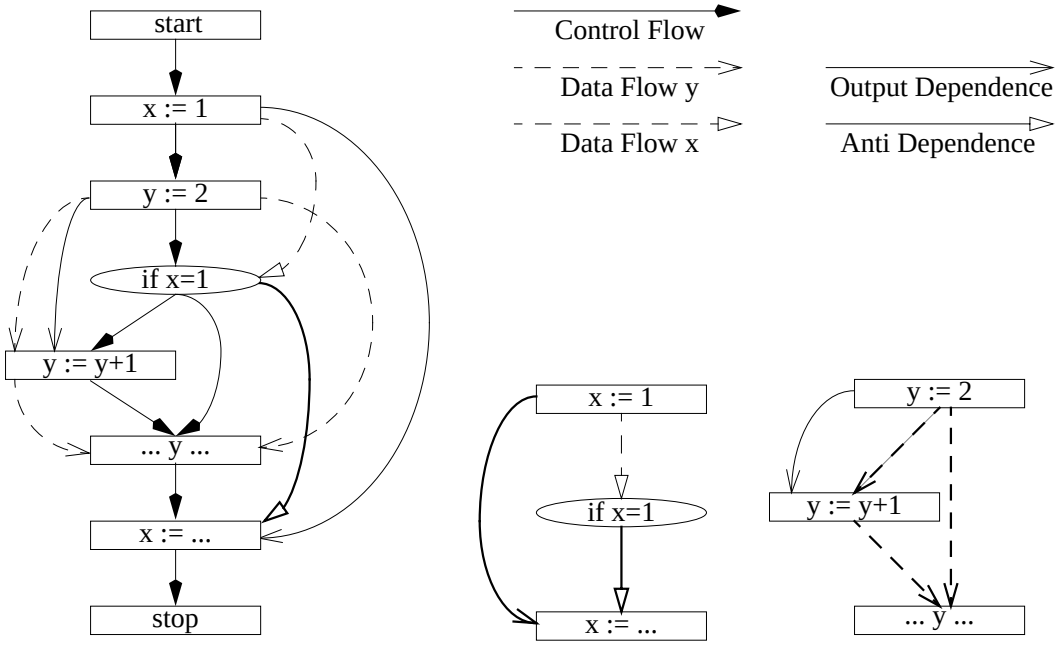
Definition 2.3.8 *Given a CFG. A* **def–use chain** *for a variable v is a node pair $(n_1, n_2) \in N \times N$, such that n_1 defines v, n_2 uses v and n_1 reaches n_2.*

Definition 2.3.9 *Given a $CFG = (N, E_{cf}, \tau)$. $DUG = (N, E_{du}, \tau)$ is a* **def–use graph**, *such that $(n_1, n_2) \in E_{du}$ iff (n_1, n_2) is a def–use chain with respect to the CFG.*

Example 2.3:

In figure 2.3(a) the control flow graph is augmented with def–use edges for the variables `x` and `y`. Omitting the control flow edges results in the corresponding def–use chains shown in 2.3(b).

Def–use chains provide partial solutions of the drawbacks of CFGs. Direct information flow between definitions and uses of a variable is permitted. By this, unnecessary propagation



Code Selection (a) Control Flow Graph (b) Data Dependence Graph

Figure 2.4: Data Dependence Graph

of information is prevented. Def–use graphs eliminate unnecessary statement orderings, exposing parallelism.
But also def–use chains suffer from several drawbacks. E.g. they cannot be used for backward data flow problems (e.g. elimination of redundant computations), because not enough information about the control flow structure is incorporated. Also, def–use chains can effect the precision of analysis in forward data flow problems, i.e. they can prevent that certain sources of optimizations are found that can be determined when using a CFG (e.g. in constant propagation).

Data Dependence Graphs

Data dependence graphs (DDG) are a generalization of def–use chains, that take into account the execution reordering constraints between nodes that arise by redefinition of certain variables. The edges of the DDG represent conflicts between two nodes x and y in the CFG, i.e. exchanging the execution order of the statements associated with the nodes changes the semantics of the program.

- *Flow Dependence:* y is on a path from x to n_{stop} in the CFG, such that the definition x reaches the use y.
- *Anti Dependence:* y is on a path from x to n_{stop} in the CFG, such that y subsequently redefines a variable used in x.
- *Output Dependence:* subsequent redefinition of the same variable.

These dependencies force strict ordering among the corresponding statements to ensure program correctness. Figure 2.4(b) shows the DDG of the control flow graph in figure 2.4(a).

Programs that contain loops must be handled with care. In this case static edges in the DDG must be distinguished from dynamic edges that order two nodes from successive iterations:

- *Loop independent dependence:* Dependencies that denote an order of nodes of the same dynamic iteration.
- *Loop carried dependence:* Two nodes representing instances of statements in successive iterations.

Data Flow Graphs

Data flow graphs (DFG) represent global data dependence at the operator level. Nodes in a DFG with no incoming edges represent values and internal nodes represent operators. DFGs completely abstract from statements and statement ordering. Def–use graphs can be transformed to a similar representation, where operator nodes are labeled with a (possible empty) set of variables, representing assignments to the corresponding variables [ASU86]. In [ASU86] this representation is used to represent the data flow of a basic block and is called the DAG of a basic block (figure 2.5).
There are extended approaches of data flow graphs incorporating control flow, which are often used in the context of functional languages with semantics based on data flow machines.

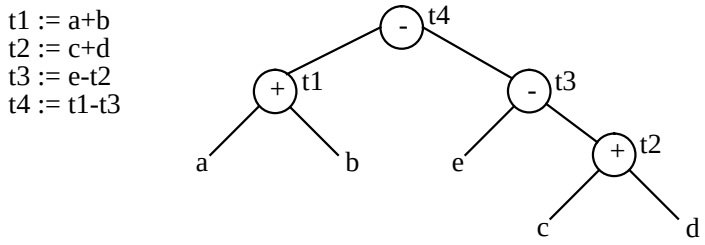


Figure 2.5: DAG of a Basic Block

2.3.5 Program Dependence Graph

Solutions to overcome the drawbacks of data flow representations use the CFG together with the DDG (or DFG). However, it is difficult to keep both representations consistent in the context of program transformations. The problem of maintaining two data structures to represent the program execution semantics and its dependencies is addressed by the **program dependence graph**. A PDG contains the DDG augemented with control dependence edges [FOW87]. Therefore it can be stated as the union of the relevant control and data dependencies. The PDG incorporates the CDG which represents only the essential control relationships of a program.

Definition 2.3.10 *The* **program dependence graph** *(PDG) is derived from a given* $CFG = (N_{cf}, E_{cf}, \tau_{cf})$*, such that*
$PDG = (N_{pd}, E_{cd}, E_{dd}, \tau_{pd})$ *with*

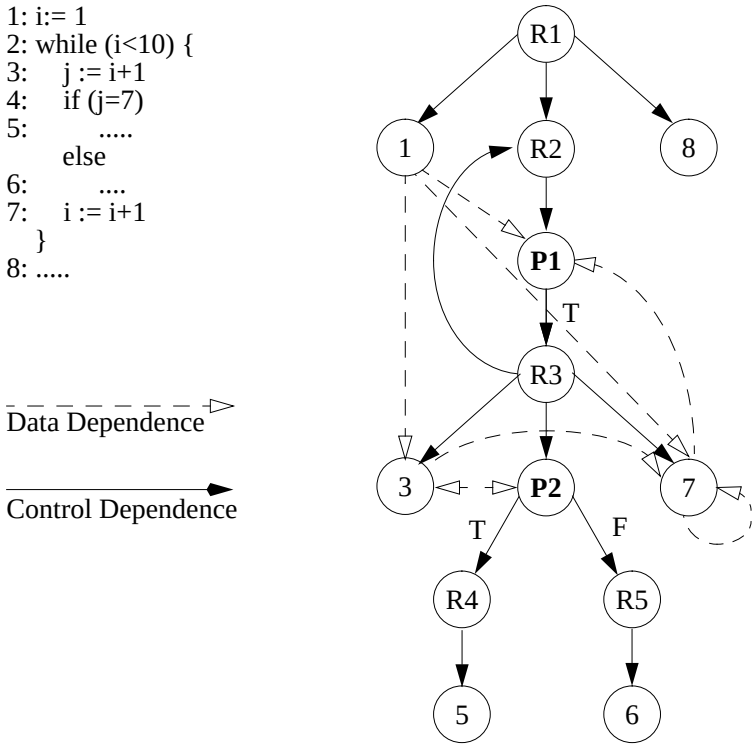


Figure 2.6: Program Dependence Graph

- $N_{cf} \setminus \{n_{stop}\} \subset N_{pd}$.
- $\tau_{pd} : N \rightarrow \{START, REGION, PREDICATE, COMPUTE\}$ *is a node type mapping. In contrast to CFGs, nodes with type COMPUTE have no outgoing edges and there is no STOP node.*
- $E_{cd} \subset N_{pd} \times N_{pd} \times \{T, F, U\}$ *is a set of labeled edges, such that* $(n_1, n_2, L) \in E_{cd}$ *identifies a control dependence from* n_1 *to* n_2 *with label* L.
- $E_{dd} \subset N_{pd} \times N_{pd} \times D$ *is a set of labeled edges, such that* $(n_1, n_2, D) \in E_{dd}$ *identifies a data dependence from* n_1 *to* n_2 *from a set of data dependencies* D*, e.g. flow–, anti–, or output dependencies.*

The PDG contains the same nodes as the control flow graph and is aughmented with additional nodes called region nodes. These are inserted into the graph to summarize the set of control conditions for a node and to group all of the nodes that are executed under the same control conditions together as the successors of the same region node. Each predicate node has at most one successor node of type $REGION$ *labeled with* T *or* F*. Edges from* E_{cd} *with the source node is of type* $REGION$ *are always labeled with* U.

Figure 2.6 shows a small program together with its program dependence graph representation. Many of the optimizations operate more efficiently on the PDG [FOW87] and also incremental program transformations on control flow and data flow are permitted. Detailed descriptions of PDGs are given in [SS93, GP92, FOW87]. There have been efforts to give PDGs a formal semantics, with the objective for using it in the correctness proofs of program transformations,

but this has proved to be very difficult [JP93]. Also linearization has been found to be very difficult using the PDG.

2.3.6 Global Unified Resource Requirement Representation

The global unified resource requirement representation (**GURRR**) augments the program dependence graph with information about the resource requirements and resource availability. It was developed to enable a better integration of register allocation and instruction scheduling, while taking into account the real requirements of the target machine [BGS95]. Therefore this approach also considers important aspects for retargetability. The allocation of specific resources of the target machine is performed while considering the overall execution time of the program. Another aim of this approach is to define a common base for a high amount of optimizations and to overcome the drawback of using several different representation, thus restricting the degree of phase integration.
A major goal is to support instruction level parallelism that is appropriate for a certain target architecture by detecting regions of the program that are over–utilized, and regions that are under–utilized with resources. The representation permits to determine the impact of each decision made with major regards to the execution time of a program.

2.3.7 Other Works

There is a wide spectrum of other IRs and extended approaches of the introduced IRs. An introduction to this IRs is out of the scope of this paragraph, therefore only some short remarks on further approaches are given in the following.

Dependence Flow Graph The **dependence flow graph** is a generalization of def–use chains and SSA, solving some of the drawbacks shown in the context of CFGs, def–use chains and PDGs. The dependence flow graph utilizes the propagation of control flow information, while bypassing informations not relevant to certain regions [JP93, Joh94]. Also, the dependence flow graph has a well–defined semantics [PBJS90].

Parallel Program Graph The **parallel program graph** (PPG) is a variant of the PDG. It contains control edges that represent parallel flow of control and synchronization edges [SS93].

Program Dependence Web The **program dependence web** (PDW) is an executable program representation derived from the program dependence graph and was designed to support control–driven, data–driven and demand–driven execution. The intention is to provide a single IR to support multiple styles of programming languages (e.g. functional and imperative) and multiple architectures (e.g. von Neumann, data flow, reduction machines) [BMO90].

Hierachical Task Graph The **hierarchical task graph** (HTG) consists of five types of nodes:

- $START$ and $STOP$: indicating entries and exits to HTGs. In contrast to CFGs a single HTG may have more than one node of either type $START$ or $STOP$ according to the fact that a HTG may consist of nodes containing sub–HTGs;
- $SIMPLE$: contains an instruction;
- $COMPLEX$: representing a sub–HTG;
- $LOOP$: represent loops whose body is a sub–HTG;

The HTG is more coarse grained than the PDG. It therefore allows program transformations on a more abstract level. (see [GP92] or [NN93] where the HTG is used for instruction scheduling).

Restricted Permutation Trees (see [San94]).

2.4 Summary

Intermediate representations have great impact on the effectiveness of optimization with regards to either implementation and precision of the performed optimizations. The major goal is to find a single intermediate representation to be used, that enables a high amount of optimizations to be effectively performed, with special regards to phase integration. There is a large number of intermediate representations developed in recent years. Control flow graphs explicitly reflect the original structure of the source program. Several representations were developed, for only denoting the relevant dependencies of a program, like def–use chains, data dependence graphs, and control dependence graphs. Program dependence graphs and data dependence graphs try to overcome the drawback of using several different representations for a program. Extentions of the program dependence graphs try to integrate aspects of fine–graine (instruction level) and coarse grain (task or functional level) parallelism and to integrate aspects of the resource requirements of the target machine.

Chapter 3

Retargetable Code Generation

The chapter gives an introduction to traditional tasks of *retargetable code generation*: *code selection*, *register allocation*, and *instruction scheduling*. A small formal model of machine operations is defined, getting a common frame-work for the description of the subjects of code generation tasks. The essential entities necessary for retargeting the tasks are exposed. The chapter is structured as follows:

- Section 3.1 gives a short introduction of the tasks of a retargetable code generator.
- In section 3.2 elementary machine operations are identified. It will be shown how the entities (i.e. machine resources) of the corresponding target architecture are composed to yield elementary operations. It is further described how elementary operations are combined for constituting machine instructions. In section 3.2.1 a representation for the encodings of machine operations and machine instruction is presented.
- In section 3.3 the elementary entities for retargeting the tasks of code generation are exposed within operation specifications. The relationship of the intermediate representation and machine operation is specified.
- The last section summarizes the introduced notions important for further reading of the report.

3.1 Introduction

The task of code generation is the mapping of an intermediate representation IR of a source program to a target machine program. The aim of code generation is the selection of a nearly optimal machine instruction sequence making effective usage of features of the target machine with respect to the semantics of the initial program. *Nearly optimal* in this context means that an optimal solution can not generally be computed because code generation consists of NP-hard subtasks:

- **code selection** is the task of mapping an intermediate representation IR to a semantically equivalent sequence of machine executable operations. There are usually several semantically equivalent sequences for representing one program. A problem is the selection of a favourable instruction sequence.

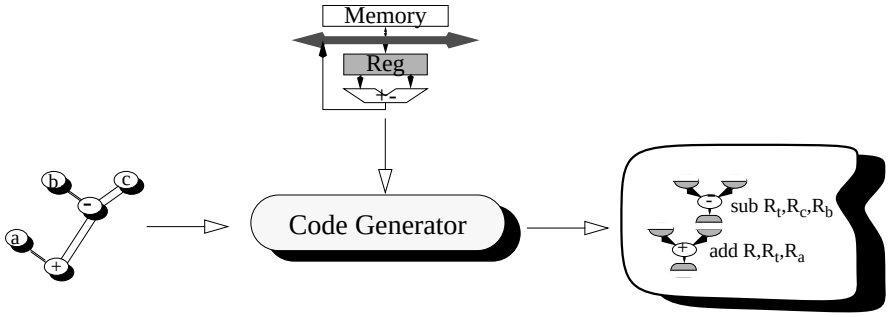


Figure 3.1: Retargetable Code Generator

- The goal of **register allocation** is to map values in the intermediate representation to physical registers in order to minimize the number of accesses to memory during program execution. It consist of two subtasks:
  - Allocate values to registers over a certain life times of the values. In general the number of concurrently alive values exceeds the number of registers. Therefore the register allocator must make decisions upon which values to keep in registers, with the goal of reducing data transfers. In the context of distrubuted register sets, a task of increasing importance is to allocate values to certain register sets.
  - After allocation, the physical registers where values should reside must be determined. This is the task of register assignment. During register assignment it has to be known which registers are occupied by which values and which registers are free for being occupied by new values.
- The traditional task of **insruction scheduling** is the reordering of machine instructions with the aim of minimizing spill code. In the context of fine grain parallelism, instruction scheduling is the task of reordering an instruction sequence for gaining effective usage of the machines parallelism. This incorporates either the **compaction** of parallel executable machine operations into one machine instruction for VLIW like architectures, or the reordering of machine instructions for avoiding pipeline stalls in RISC like architectures.

A seldom stated task of code generation is **resource allocation**. It is concerned with binding operations and values to machine resources (e.g., functional units and storage resources); this task is also called **binding**. Resource allocation can hardly be viewed as a separate task as each of the code generation tasks is concerned with some forms of binding. Register allocation can be seen as a subtask of resource allocation. Also, resource allocation can be performed before or after each task of code generation. A retargetable code generator (fig. 3.1) gets an intermediate representation of the source program and a description of the target machine. The basic task of a code generator is to identify certain patterns in the intermediate representation as elementary operations, that can be executed on the target machine. Thus, an important task of the retargetable code generator is to derive a mapping from operations occuring in the intermediate representations to target machine operations. Functional units together with the required allocations of the operands (i.e., the storage resources like registers or memories where the operands must reside) must be determined, and the corresponding

encodings of machine instructions must be extracted. In some specifications this relationship between the intermediate representation and machine instructions are explicitly exposed. A code generator generator (fig. 3.2) uses a specification that explicitly reflects mappings from patterns occuring in the intermediate representation to the corresponding target machine instructions. The output of a code generator generator is a code generator that maps the intermediate representation to the specified target machine code. There are three basic classes for modelling target machines, depending on the details of information that are available for a designer:

- **behavioral models** provide a high abstraction of the hardware. These models are generally used in code generator generators. E.g., **instruction set models** reflect the relations between the intermediate representation and machine instructions, which are explicitly specified.
- **structural models** contain more details and are usually specified by *hardware description languages (HDL)* (e.g., VHDL [BFMR92], MIMOLA [Mar93, BBH+94]); generally, more aspects of the target machine can be specified, that cannot be defined a purely behavioral model (e.g., complex timing behavior). The machine operations are implicitly inherent and must be extracted from the description. Generally, this is restricted to single cycle machine instructions.
- **mixed models** consist of information from both previously mentioned models. In common, instruction set models are enhanced by additional informations, e.g., used machine resources, size of storage resourcesor encodings of machine instructions. These are basically informations, necessary for effectively retargeting register allocation and instruction scheduling.

The question of which model should be applied depends either on the informations available to the designer, but also on the class of architecture being modeled. As mentioned, multicycle machine instructions are hard to be extracted from purely structural models. A behavioral descriptions lacks of informations for retargeting register allocation and instruction scheduling. It is still a topic of further research finding adequate description models for both of

- utilizing the design process of architectures, and
- support effective retargeting of all the subtasks of code generation.

In the following section elementary operations of a machine are specialized. It is shown how they are combined to machine instructions.

3.2 Machine Operations and Machine Instructions

To enable an understanding about the interrelations of an intermediate representation and the target machine code, the elementary operations a target machine is able to execute are identified. It is shown how this elementary operations are combined to yield machine instructions. In the following we will define machine operations and machine instructions based on the terminology developed in the area of microprogramming [DLSM81].

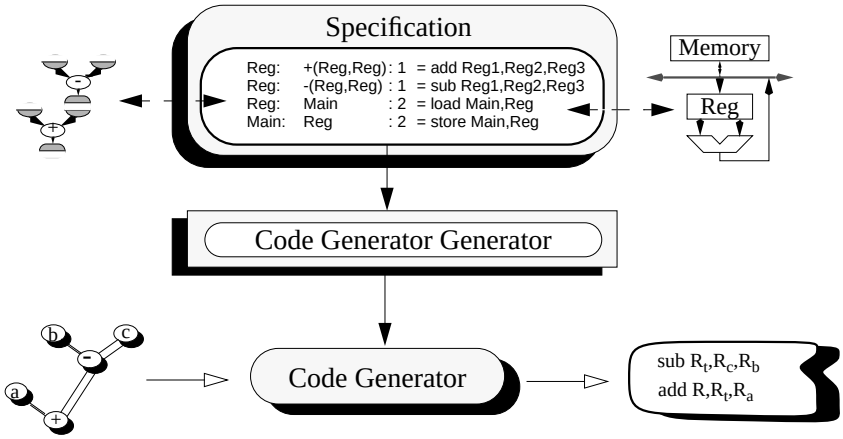


Figure 3.2: Code Generator Generator

3.2.1 Microoperations and Microinstructions

A target architecture is well-defined by its storage resources (e.g. register sets, memory) and operations that can be performed. A **microoperation** (MO) is an elementary operation executable by the target machine and performed on data stored in one or more storage resources (possibly distributed on more than one register set) and stores a result into certain storage resources. Microperations that can be performed in parallel are combined to a **microinstruction** (MI). Microinstructions are controlled by signals from the **control word** in the **control unit**. The control word can be represented by a string of a certain length (called its *bit width* or *width*) over the alphabet $\{0, 1, X\}$. The X is representative for a signal that can be either 0 or 1 without changing the semantics of any of the MOs within the MI.

A microinstruction can be partitioned into several logical fields, where a certain set of fields being responsible for initiating certain microoperations. The control unit can be either hardwired or microprogrammable. A *programmable control unit* contains a memory called the control memory. A sequence of microinstructions constitutes a **microprogram** that is stored in the control memory. Thereby, microinstructions are stored in certain cells of control memory. A microinstruction stored in cell n of the control memory is called to be mapped to **instruction cycle** n. The control memory can be read-only (ROM) or reloadable to load new microprograms. The actual control word is determined by the **microprogram counter**.

Example 3.1:

In figure 3.3 a configuration of a control unit is shown. It has a bit width of twenty bits. The notation $c(m : n)$ specifies certain bits of the control word, e.g. $c(10 : 5)$ denotes bits 5 to 10 of the control word c and addresses the register file B. $c(11)$ controls the ALU $F2$, i.e. the encoding of $c(11)$ determines the operation performed by $F2$, i.e. * or + ($c(21)$ controls the ALU $F1$, i.e., - or +).

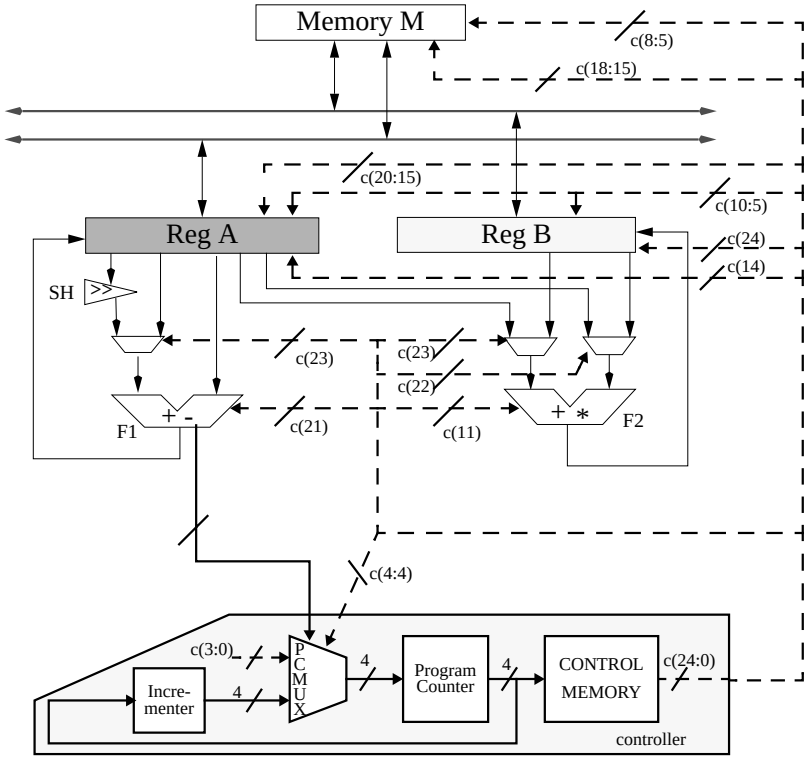


Figure 3.3: Example Architecture

Example 3.2:

We consider the previous example for illustrating the notions microoperations and microinstructions according to functional units and storage resources of a simple architecture. In figure 3.4 the control unit and control signals are omitted. The figure shows an architecture that contains functional units $F1$, $F2$ and SH. There are three storage resources: register set A, register set B and the memory M. $F1$ can perform an addition or a substraction of two operands stored in register cells of register set A. The result is stored into register A. Shifting of the first operand is optional. $F2$ can perform an addition and a multiplication of operands stored in A or B with the result stored into register B.

MOs are defined as operations performed on data that resides in storage resources with results written to storage resources; thus the shifter SH does not constitute a complete microoperation in this sense. Only in combination with $F1$ a complete MO is given. Operands accessable to $F2$ can reside either in A or B. The result is stored to B . Data can be loaded from the memory M to A or B and it can also be stored to memory M from A or B. An illustration of the performable microoperations is shown in figure 3.5.

3.2.2 From Microoperations to Machine Operations

If the source language specifies a microprogram, the task of the code generator is to determine a corresponding microinstruction sequence. In some architectures this is the only level of programmability like e.g. some *application specific instruction set processors* (ASIPS). But generally there can be higher levels of programmability. In this case a program is a sequence of

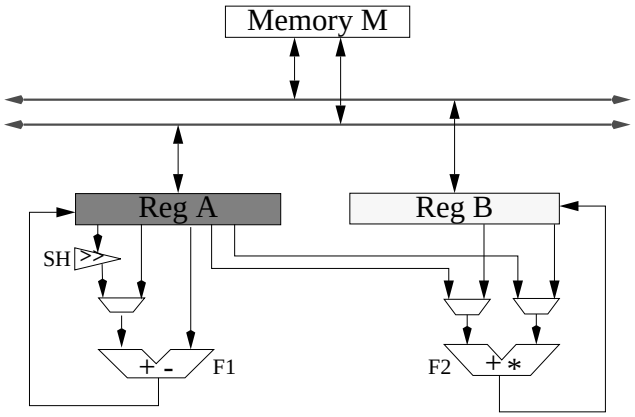


Figure 3.4: Simplified Example Architecture

machine instructions stored in an additional memory. When executed, a machine instruction initiates a microinstruction or a sequence of microinstructions. During program execution the microprogram usually does not change. Changing the microprogram enables the integration of late design decisions, correction of design errors and reuseability. Also, the modification of the target architectures instruction sets is utilized, e.g. instruction sets that support specific features of programming languages (e.g. WAM or SECD–Machine [Kog91]).
If more abstract levels of visibility of hardware details and of programability are taken into account, we use the terms **machine operation** and **machine instruction** instead of micro-operation and microinstruction. Generally, a machine operation will denote an elementary operation on the **register transfer level** of the target machine, that is visible from the current point of view (or level of abstraction). A machine instruction can consist of one or more concurrently executable machine operations if this parallelism is explicitly visible. For example, in a VLIW architecture we have explicit instruction level parallelism. A CISC like architecture does not offer this kind of parallelism explicitly. However, implicitly a CISC instruction may be implemented by a sequence of microinstructions.

3.2.3 Machine Operation Pattern

Figure 3.6 shows the **machine operation patterns**. The patterns will be the basic subject for relating operators in the intermediate representation with target machine operations. Storage resources are shaded as in the corresponding architecture in figure 3.4. The representation is based on the symbols of the operators and storage resources and reflects the data flow of machine operations. Notations of a corresponding **register transfer language** (see [Man93]) is associated with each the machine operation pattern. The notation describes the register transfer level behavior of the machine operation patterns , but abstracts from a certain location (i.e., address of the denoted storage resource).

3.2.4 Multicycle Machine Operations

The view of machine operations we have considered so far indicates that each machine operation is performed in one machine instruction cycle. Generally, a machine operation can require several instruction cycles. Thereby it occupies a certain set of machine resources in

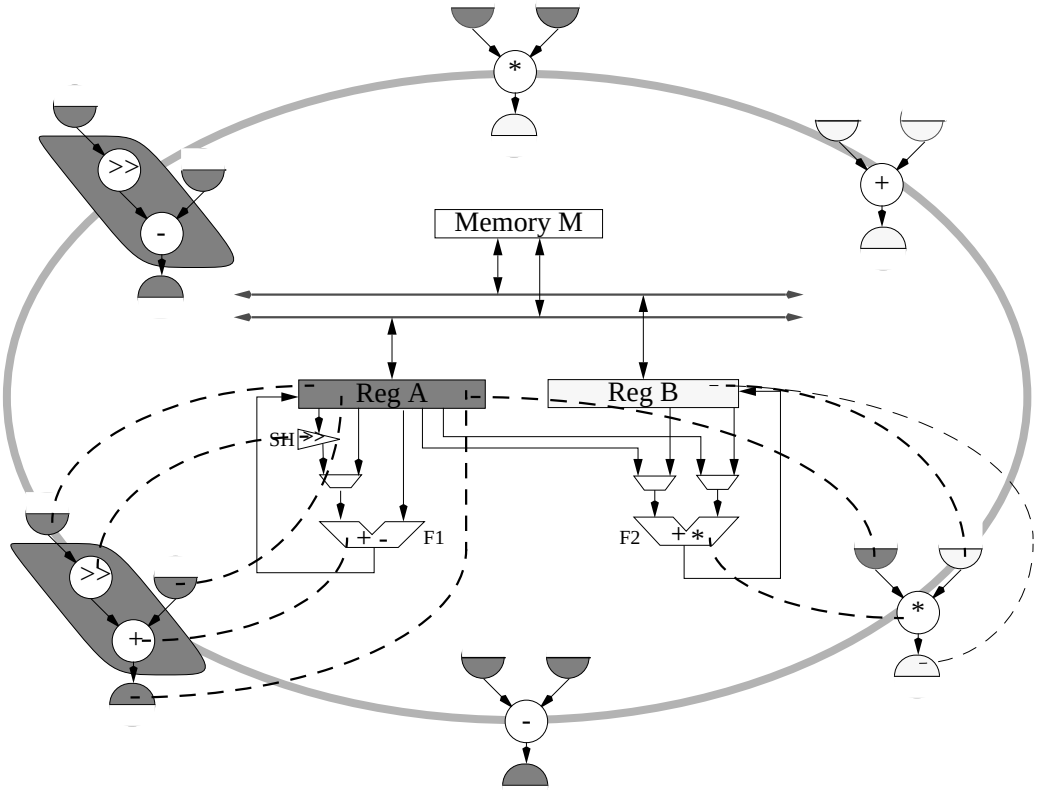


Figure 3.5: Elementary Operations

each cycle. If various machine operations take different numbers of cycles to execute, the meaning of a machine instruction becomes vague. If we now consider machine instructions and assume that certain fields within the machine instruction initiate certain machine operations, the execution time for machine operations have to be taken into account when machine operations are mapped to certain instruction cycles. The point of view is changed from considering machine instructions consisting of machine operations to machine operations that are mapped to certain instruction cycles.

Example 3.3:

> In figure 3.7 (a) and (b) machine operations with multiple instruction cycles are shown. In figure 3.7 (b) it is illustrated that certain machine operations can be initiated before other machine operations terminated, e.g. $MO5, MO6$ and MO7.

In the following we will basically consider single cycle machine operations. The problems that are considered in this report are inherent in this restricted model.

3.2.5 Conflicting Machine Operations

We will now discuss demands of concurrent execution of machine operations and introduce the notion of conflicting machine operations. Thus the point of interest are the reasons that prevent machine operation from parallel execution. A set of machine opererations can be

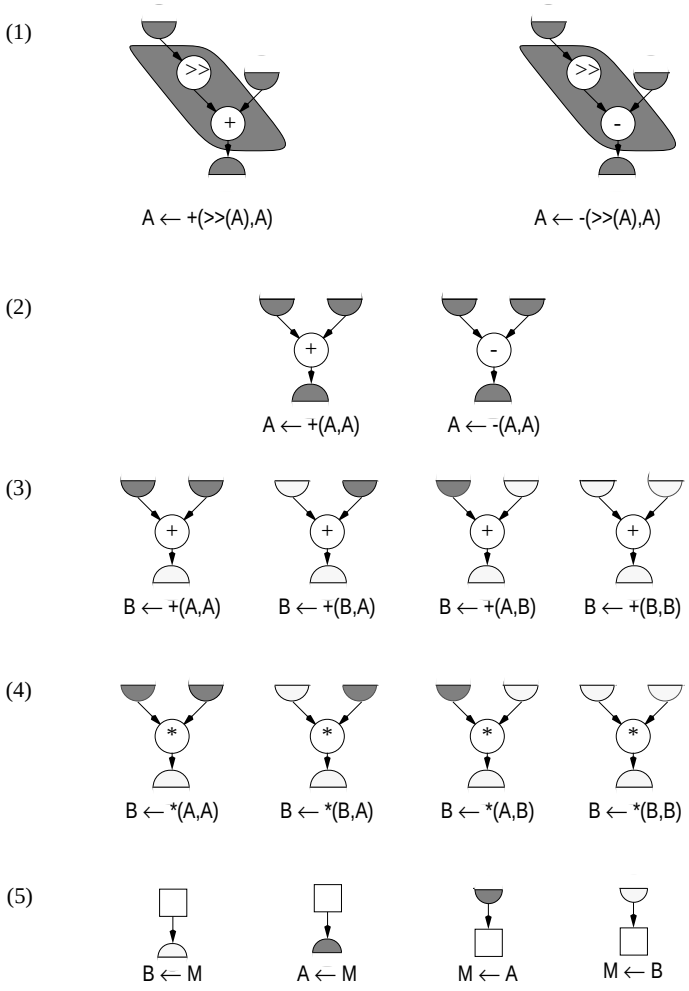


Figure 3.6: Machine Operation Patterns

executed in parallel if no **resource conflicts** occur. A resource conflict occurs, if the number of available resources accessed is exceeded. For example, each functional unit can only be used by one machine operation in each machine instruction cycle. Storage resources only allow write access according to their number of write ports.

Example 3.4:

> In the example architecture an addition with shifting of the first operand can never be executed in parallel with a substraction, because both operations need the functionol unit $F1$. If we assume, that the register sets A is equipped with a single write port, parallel execution of machine operations involving $F1$ and a transfer machine operation from memory to A would also cause a resource conflict.

Another class of resource conflicts can occur if certain machine operations are controled by the same fields in a machine instruction word. An **encoding conflicts** occurs if two machine operations have different encodings for a certain fields. For more details about conflicts and how to model the detection of conflicts consult [DLSM81, Gas89, Hei93]. If we suppose single cycle operations, resource conflicts can be mapped to encoding conflicts.

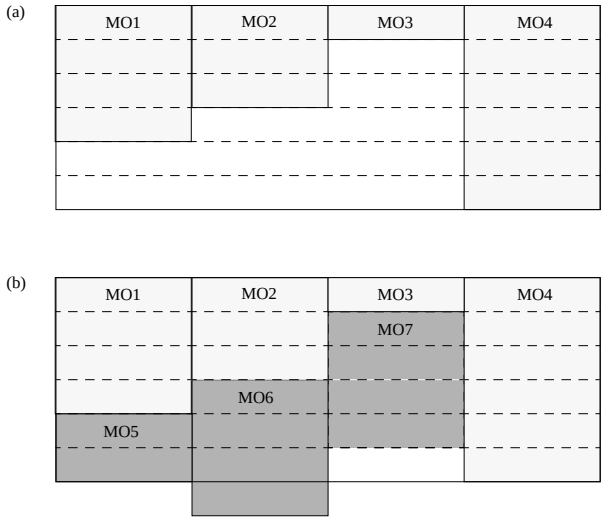


Figure 3.7: Machine Operations with Multiple Instruction Cyles

3.2.6 Encoding of Machine Instructions

This subsection is concerned with a precise notion of encoding machine operations and machine instructions. How to determine encodings for machine operations from hardware descriptions is not considered in this report (see [LM94]).

Definition 3.2.1 *A* **machine instruction string** *of width w is a string over the alphabet $\{0, 1, X\}$ denoted by $\{0, 1, X\}^w$. It is of the form $a_{w-1}a_{w-2}\ldots a_0$, $a_i \in \{0, 1, X\}$ for $(w > i \geq 0)$. Each i is a* **bit position** *with associated value a_i. A* **bit range** *$(w_1 : w_2)$, $w > w_1 \geq w_2 \geq 0$, specifies a sequence of bit positions $w_1 \ldots w_2$ with the associated machine instruction string $a_{w_1} \ldots a_{w_2}$ of width $w_1 - w_2 + 1$.*

We assume that machine operations can be mapped to a machine instruction string. X denotes a signal at the corresponding bit position not effecting the behavior of the machine operations initiated by the machine instruction, therefore the value can be either 0 or 1. A machine operation can be encoded by a machine instruction string or by a set of alternative machine instruction strings. All bit positions not relevant for the execution of the machine operation should be represented by X's.

Definition 3.2.2 *Two machine operations* **conflict***, iff there exists a bit position i in the corresponding machine instruction strings $mis_1 = a_{w-1}\ldots a_0$ and $mis_2 = b_{w-1}\ldots b_0$, such that $b_i \neq a_i$ and $b_i \neq X$ and also $a_i \neq X$.*

Thus, two machine operations can be performed if their machine instruction strings do not conflict. This criterium is used in the compaction phase of some retargetable code generators, e.g. MSSV [Mar93].

Machine Instruction Format

In the following, a formalism is introduced for defining the machine instruction strings and reflecting the logical partitioning of machine instructions into a certain set of control fields.

Definition 3.2.3 *A* **machine instruction format** *of width* w *is a sequence* $MIF = [f_1, \ldots, f_k]$ *of* fields $f_i = (id_i, m_i, SF_i)$*, such that (for* $k \geq i, j \geq 1$*)*

- id_i *is a field identifier that denotes the field* f_i *and* $id_i \neq id_j$ *iff* $i \neq j$*;*
- m_i *defines the highest bit position of field* f_i *with* $w > m_i \geq 0$*. The bit range* $(m_i : m_{i+1} + 1)$ *defines the bit positions of field* f_i *with* $m_i > m_{i+1}$ *and dummy position* $m_{k+1} = -1$*. It is required that* $m_1 = w - 1$*.*

  With each id_i *a set* $S_{id_i} \subseteq \{0, 1, X\}^{m_i - m_{i+1}}$ *of machine instructions of width* $m_i - m_{i+1}$ *is associated.*
- *A field can be further partitioned into subfields;* SF_i *specifies a (possible empty) set of machine instruction formats for field* f_i*, denoted* $id_i = mif_{i,1}| \ldots |mif_{i,n_i}$*. Each* $mif \in \{mif_{i,1}, \ldots, mif_{i,n_i}\}$ *specifies a machine instruction format of width* $m_i - m_{i+1}$*.*

MIF specifies a set of machine instructions denoted MI_{MIF} *and* $mi \in MI_{MIF}$ *iff* $mi = s_1 \bullet \ldots \bullet s_n \wedge s_i \in S_{id_i}$*. For* $id_i = mif_{i,1}| \ldots |mif_{i,n_i}$ *the union* $MI_{mip_{i,1}} \cup \ldots \cup MI_{mip_{i,n_i}}$ *is exactly* S_{id_i}*. Therefore a non-empty set* SF_i *specifies* $S_i = MI_{mip_{i,1}} \cup \ldots \cup MI_{mip_{i,n_i}}$*.*

A MIF enables to define the formats for a complete machine instruction set in a compact representation. A encoding for a certain machine operation is defined by a specific configuration of fields. All fields not relevant for the execution of the machine operation should be represented by X's.

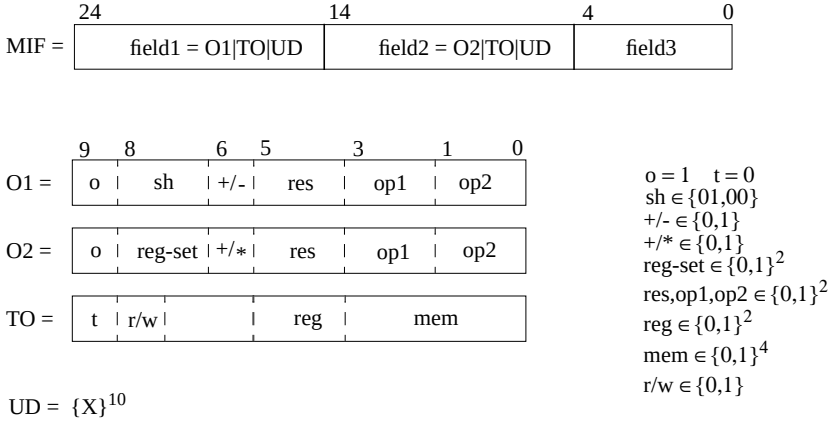


Figure 3.8: Machine Instruction Format

Example 3.5:

A machine instruction format for the example architecture (figure 3.4 is shown in figure 3.8. The first field $field1$ controls the machine operations that involve the functional unit $F1$. It can also initiate the transfer machine operations between the memory M and the register set A. Which of either an arithmetic machine operation or a transfer machine operation is performed is determined with the value of bit position 24.

The machine instruction format is composed from the submachine instruction formats $O1$, $O2$, TO and UD; $field1$ can be composed from either of the machine instruction formats $O1$ (specifying the arithmetic machine operations), TO (transfer machine operations) or UD (no machine operation specified). The subfield sh of $field1$ (specified in $O1$) controls the shifting of the first operand of $F1$. $+/-$ encodes the operation performed by the functional unit $F1$. The other fields of $O1$ specify the addresses of the operands of the operation, i.e. the locations in register set A. Field $reg - set$ in $O2$ determines the sources for the operands. The subfield r/w specified in $T0$ determines loads or stores of a transfer machine operation. The field $field3$ is used to control the program counter for selecting the next machine instruction to execute. It is not further specified here and is omitted in the following. The machine instruction format allows to encode two arithmetic machine operations or two data transfers in one machine instruction. It allows the encoding of a data transfer between memory and registers and an arithmetic machine operation within one machine instruction.

Versions and Partial Versions of Machine Instruction Strings

A certain machine operation can be encoded by several distinct machine instruction strings. Each machine instruction string will be called a **version** of the corresponding machine operation. A machine instruction string denoting the encoding of a subfield of a version is called a **partial version**. I.e., partial versions represent the signals of the control word necessary for controling a certain machine resource, involved in a certain machine operation.

Restricting Machine Instruction Formats

A restricted machine instruction formats reduces a given machine instruction format MIF to a subset of its formats. Hereby, the encoding for a certain machine operation can be indicated by an existing MIF.

Definition 3.2.4 *Given a fixed* $MIF = [(id_1, m_1, SF_1), \ldots, (id_k, m_k, SF_k)]$. *The* **MIF restriction** *of a machine instruction format is defined as*

$$MIF' = [(id'_1, m_1, SF'_1), \ldots, (id'_k, m_k, SF'_k)]$$

with associated sets $S'_{id_1}, \ldots, S'_{id_k}$*, such that*

- $S'_{id_1} \subseteq S_{id_1}, \ldots, S'_{id_k} \subseteq S_{id_k}$ *and*
- *for each non-empty set of subfields* SF'_i *each* $mif' \in SF'_i$ *is a restriction of a* $mif \in SF_i$.

We denote a restriction of a machine instruction format MIF by $MIF' = [id'_1 : S_{id_1}, \ldots, id'_k : S_{id'_k}]$ and we will omit all fields $id'_i : S_{id_i}$ with $f'_i = f_i$.

Example 3.6:

In figure 3.9 the restricted machine instruction formats for each machine operation are shown. The first machine operation defines the machine instruction format of the *shift and add* operation. *field2* is restricted to contain the undefined operation, i.e. the subfield only contains X's. $field1$ is restricted to the format $O1'$ a restriction of the format $O1$ which contains the encodings of the addition (`+/- = 1`) and shifting (`sh = 01`). The subfields res, $op1$, and $op2$ of $O1$ are kept unchanged.

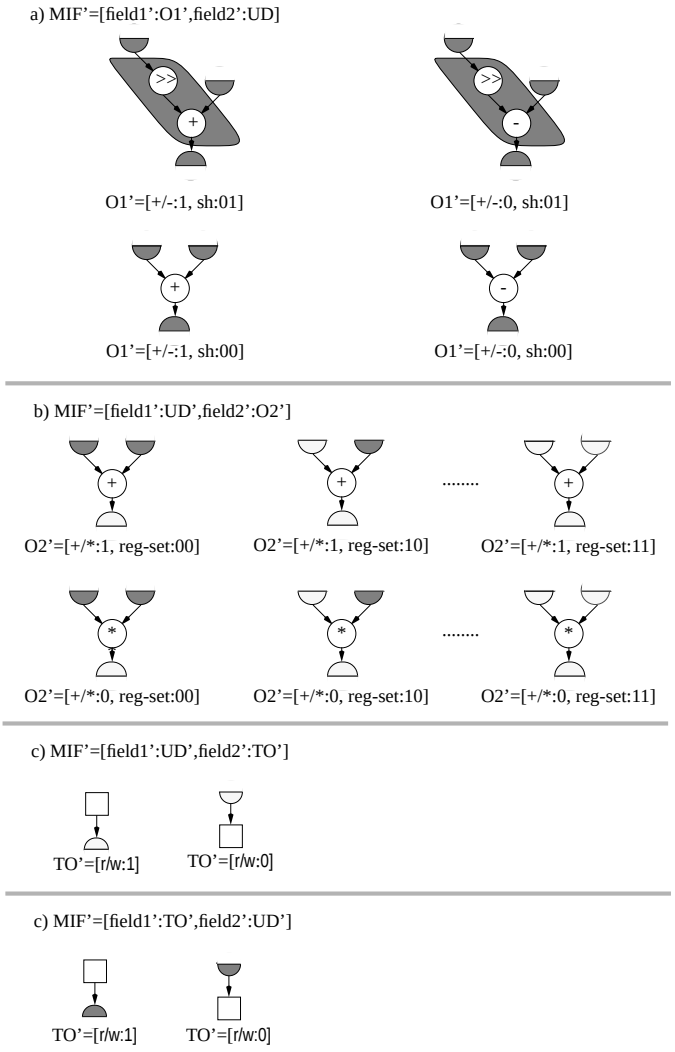


Figure 3.9: Encodings of Machine Operation Pattern

3.3 Operation Specification

An operation specifications specifies the alternative machine operations for implementing a certain operation on the target machine. The model shown here is an extended instruction set model and exposes the necessary entities for retargeting code selection, register allocation and instruction scheduling for single cycle machine operations.

In the following a fixed set of functional units of a target machine is assumed, denoted by $\mathcal{FU}$. Further a fixed set of storage resources is given, denoted by $\mathcal{SR}$. With each $sr \in \mathcal{SR}$ a certain set Loc_{sr} of permitted locations (i.e., addresses) is associated. Additionaly, for each $sr \in \mathcal{SR}$ there is a unique symbolic representation to denote sr. Further, we assume a fixed width m of the machine instruction words and a fixed set of machine instruction strings, denoted M.

Definition 3.3.1 *An* **operation specification** *for an operation op is a quadruple $OS_{op} = (id, arity, mos, V_{id})$ and consists of*

- id: *a unique symbol denoting the operation op;*
- arity: *denoting the number of operands including the destination involved;*
- mos: *denotes a* **machine operation scheme** *and reflects the behavior of op; for every operand it contains a template from \$1,. . . ,\$arity, representative for certain combinations of storage resources;*

  example: \$1 ← +(\$2,\$3);
- *V_{id} represents the set of all machine operations that implement the behavior of the machine operation scheme; each $v \in V_{id}$ is a* **resource machine operation** *that implements the machine operation scheme occupying the same configuration of machine resources, denoted by $\mathcal{R}$–MO; $(F, [sr_1, \ldots, sr_{arity}], \phi) \in V_{id}$ consists of:*
  - *$F \subseteq \mathcal{FU}$, the set of involved functional units.*
  - *$sr_1, \ldots, sr_{arity} \in SR$ are the storage resources where operands of the $\mathcal{R}$–version reside; each sr_i corresponds to the template \$i in the machine operation scheme; the substitution of storage resource symbols for the corresponding templates $\$1, \ldots, \$arity$ in the machine operation scheme constitutes the* **machine operation pattern***; the right hand side of a machine operation pattern is called a* **machine expression pattern***;*
  - *$\phi : L_1 \times \ldots \times L_{arity} \rightarrow \mathcal{P}(M)$ is an* **encoding function***; it is a mapping from locations to a set of machine instruction strings; each L_i denotes the set of legal locations for addressing sr_i the $\mathcal{R}$–version has access to; it is required that $L_i \subseteq Loc_{sr_i}$; the encoding function maps a sequence of locations $(l_1, \ldots, l_{arity}) \in L_1 \times \ldots \times L_{arity}$ to the corresponding machine instruction string.*

There are some special classes of operations called **transfer operations** and **noload operation**. *Transfer operations* describe data movements between storage resources, denoted by the machine operation scheme $\$1 \leftarrow \$2$ and the corresponding machine operation patterns. *Noload operations* are necessary for architectures providing fine–grain parallelism. They prevent undesirable side–effects, i.e., the modification of certain storage resources . Noload operations are denoted by the machine operation scheme $\$1 \leftarrow$. This model so far assumes, complete control by the machines control unit. Some architectures allow that certain configurations of the machine state are necessary to initiate certain machine operations, termed **residual control**. Code generators that produce code for such architectures are concerned

with generating code for creating the corresponding machine states. This subject is not further addressed in this report. In the following we assume non–residual control. Addressing modes are also not considered.
Other instruction based models for operations can be found in [DLSM81, BHE91, Hei93, Coh94]. The notions defined in [DLSM81] were basically introduced for comparing different compaction methods of machine programs. Informations, utilizing code selection are not involved. [BHE91, Hei93, Coh94] all take into account multi cycle machine operationsand extend the model for specifying features of RISC like architectures. [BHE91] only considers a single machine operation pattern for each operation. In contrast to code selection based specifications (see section 4) these models are concerned with utilizing register allocation and instruction scheduling. The models basically differ in the degree of details of machine resources incorporated, and relations between machine resources described. While they still can be classifed as behavioral (instruction based models), more and more aspects of structural models are integrated.

3.3.1 Abstract Machine Operations

An operation specification specifies the set of machine operations that implement the behavior given by the machine operation scheme. It defines a hierarchy on this set of machine operations, each constituting a certain level of abstraction, also exposing a certain degree of binding machine resources. This degree of binding has much impact for the tasks of code generation. It can be disadvantegeous if operations are fixed to certain machine resources by one of the tasks, then restricting subsequent tasks. E.g., if code selection selects machine operations, all resources are fixed for the operation. There are the following degrees of binding machine resources:

- An $\mathcal{L}$–MO specifies all storage resources and the locations (addresses) of its operands and the functional units that perform the operation. It consists of a set of **versions**, specified by the encoding function.
- An $\mathcal{R}$–MO defines the set of machine operations, such that each machine operation occupies the same set of functional units and assumes each operand op_i in the same storage resource sr_i; locations are not bound.
- An $\mathcal{SR}$–MO consists of the union of $\mathcal{R}$–MOs with the same machine operation pattern. Each machine operation of an $\mathcal{SR}$–MO assumes each operand op_i in the same storage resource sr_i. Whether the functional units nor locations are bound.
- An $\mathcal{U}$–MO is the complete set of mos specified by an operation specification. Thus it does not bind any machine resource or location.

Other levels of abstraction are possible when the complete set of operation specifications is analyzed. We assume, that there is a symbolic representation of operations, such that each set of associated machine operations is uniquely denoted. This incorporates symbolic representations for storage resources. The symbolic representations are called **abstract machine operations** ($\mathcal{A}$–MOs). $\mathcal{R}$–MOs, $\mathcal{SR}$–MOs, and $\mathcal{U}$–MOs define specific classes of $\mathcal{A}$–MOs. The abstract representations for storage resources are called **virtual registers**.

We extend the notion machine instruction, such that a machine instruction consists of a set $\{amo_1, \ldots, amo_n\}$ of $\mathcal{A}$–MOs , whereby there is at least one set $\{mo_1, \ldots, mo_n\}$ of machine operations that can be executed in parallel, with $mo_1 \in amo_1, \ldots, mo_n \in amo_n$.

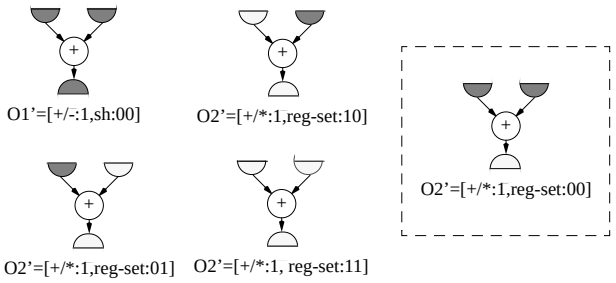


Figure 3.10: $\mathcal{SR}$–MOs

Example 3.7:

If we consider the example architecture, every $\mathcal{R}$–MO is also a $\mathcal{SR}$–MO, as no two $\mathcal{R}$–MOs have a common machine operation pattern . We have a look at the operation specification for the addition (machine operation patterns shown in figure 3.10). We assume that the domain of locations for A and B is $\{0,1,2\}$ for every version, i.e. each register set has three accessable register cells.

```
(add, 3,$1 := +($2,$3),
  {(F1,[A,A,A],(a1,a2,a3) ->
               [field2:UD,
                field1:O1'=[sh:00,+/-:1,res:a1,op1:a2,op1:a3]])
   (F2,[B,A,A],(a1,a2,a3) ->
               [field1:UD,
                field2:O2'=[+/*:1,reg_set:00,res:a1,op1:a2,op1:a3]])
   (F2,[B,B,A],(a1,a2,a3) ->
               [field1:UD,
                field2:O2'=[+/*:1,reg_set:10,res:a1,op1:a2,op1:a3]])
   (F2,[B,A,B],(a1,a2,a3) ->
               [field1:UD,
                field2:O2'=[+/*:1,reg_set:01,res:a1,op1:a2,op1:a3]])
   (F2,[B,B,B],(a1,a2,a3) ->
               [field1:UD,
                field2:O2'=[+/*:1,reg_set:11,res:a1,op1:a2,op1:a3]])})
```

Each encoding function is specified with a restricted machine instruction format. There are five $\mathcal{R}$–MOs, each $\mathcal{R}$–MO also representing a $\mathcal{SR}$–MO, because each $\mathcal{R}$–MO belongs to a different machine operation pattern of the operation specification. We will have a closer look at the $\mathcal{SR}$–MO

```
(F2, [B,A,A], (a1,a2,a3) ->
               [field1:UD,
                field2:O2'=[+/*:1,reg_set:00,res:a1,op1:a2,op1:a3]])
```

It involves the functional unit $F2$. According to the machine instruction format defined for the example architecture, $field2$ is used to encode the operation +. There fore $+/*$ is restricted to encode the addition. $field1$ is completely set to don't care (X's). A cell i of a storage resource sr is denoted by $sr[i]$. If we further specify certain locations for the operands assuming that we want to add register $A[0]$ and $A[1]$ and store the result into register $B[2]$ we will yield the following $\mathcal{L}$–MO version:

```
1)  Locations = (2,0,1) -> B[2] := A[0]+A[1]
2)  RIF = [ field1:UD,
            field2:O2'=[sh:00,+/-:1,res:10,op1:00,op1:01] ]
3)  Encoding  = XXXXXXXXXX1001100001
```

In the following the transfer operation specifications for the example architecture are shown:

```
(load, 2,$1 := $2,
     {({},[A,M],(a,m) ->
            [field2:UD,
             field1:TO'=[r/w:1,reg:a,mem:m]])
      ({},[B,M], ... )})
(store, 2,$1 := $2,
     {({},[M,A],(a,m) ->
              [field2:UD,
               field1:TO'=[r/w:0,reg:a,mem:m]])
      ({},[M,B], ... )})
```

3.3.2 Intermediate Representation and Machine Operation Patterns

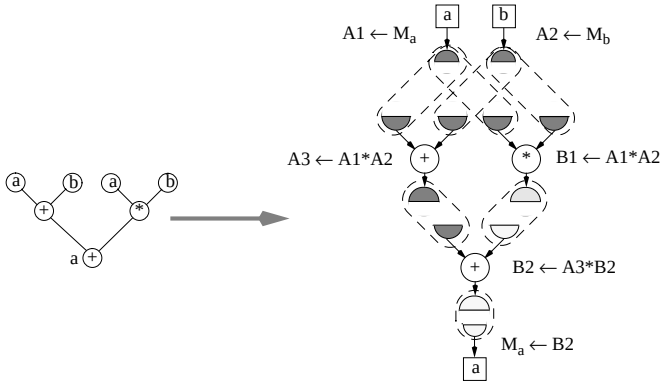


Figure 3.11: Decomposing Expressions

One basic task of code generation is to decompose the expressions in the intermediate representation into machine operation patterns , illustrated in figure 3.11. All variables occuring in the program are mapped to different virtual registers, according to the storage resources they are required and accessable to data deopendend operations that use them. Additionally, tempory virtual registers to store intermediate results are intodruced. In intermediate representations like *3–address code* [ASU86] statements are already decomposed to the form $x := binop(y, z)$ or $x := unop(y)$ (see figure 3.12). If complex expressions are supported by the architecture, the decompositions may be to fine–grained. Therefore the introduction of temporary results should not be incorporated in the intermediate representation. The decomposition should be based on the machine operations of the target machine (see figure 3.13).

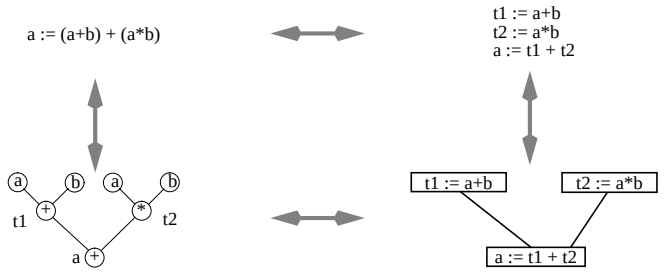


Figure 3.12: Decomposition of a DAG

3.4 Summary of Notions

Finally the major notions used throughout the subsequent text are summarized: *machine operations* are the elementary operations a target machine is able to perform on the visible register transfer level. A machine operation can be implemented by a set of *versions*, each version given by an alternative encoding (if residual control is incorporated, versions

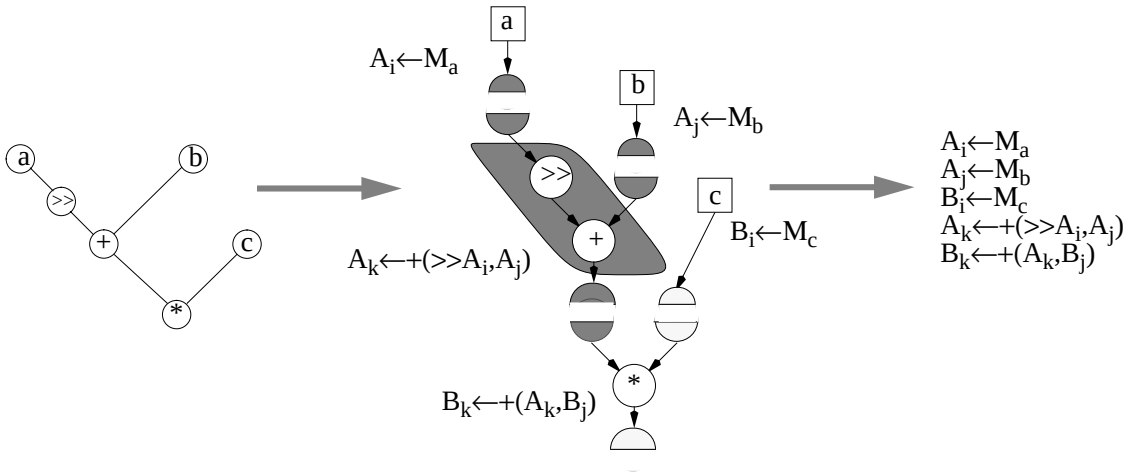


Figure 3.13: Complex Patterns

may be based also one alternative machine states). An *operation specification* defines the complete set of machine operations for implementing the specified operation. It also defines the following abstraction levels of machine operations: $\mathcal{R}$–MOs, $\mathcal{SR}$–MOs, and $\mathcal{U}$–MOs. Further abstraction levels are possible. An $\mathcal{A}$–MO is the symbolic representation denoting a set of machine operations implementing a certain operation. Each $\mathcal{R}$–MO, $\mathcal{SR}$–MO and $\mathcal{U}$–MO is an $\mathcal{A}$–MO. A *machine instruction* can be either a single $\mathcal{A}$–MO or a set of concurrently executable $\mathcal{A}$–MOs.

Chapter 4

Code Selection

This chapter is basically concerned with methods developed in the context of *code selector generators*. *Tree pattern matching* is the preferable technique for code selection. Specification techniques based on behavioral models are introduced, that can automatically be transformed to a *tree pattern matcher*. This is based on formal foundations of mapping regular tree grammars to finite tree auyomata. A short introduction of these foundations is given. Detailed introduction can be found in [FSW94, WM95]. The last two sections of this chapter are related to supported and unsupported features (of the architectures of interest), using tree pattern matching and with aspects of retargetability, respectively.
Aspects, leading to problems in the context of the architectures of interest are exposed. Finaly, relations relations between structural and behavioral models are shown. Thereby, the issue of interest is yielding regular tree grammars from structural models. The chapter is organized as follows:

- A general introduction to code selection is given, followed by by the introduction to the formal foundations of code selector specifications. This includes an illustration how these specifications can be transformed automatically to tree pattern matchers.
- Section 4.3 is concerned with existing code selector generators. The major improvements of specification techniques are outlined introducing *term rewriting rules*.
- Section 4.4 is related to the supported and not supported features of non–regular architectures, when using the introduced specification techniques and tree pattern matching. The basic problems to solve in this context lead to the integration of code selection with either register allocation or instruction scheduling considered in chapter 7.
- In the final section 4.5, the relations between structural and behavioral models are outlined. The point of interest is, yielding specifications for generation of tree pattern matchers from structural models.

4.1 Introduction

Code selection is the task of mapping the intermediate representation of a program to a sequence ob $\mathcal{A}$–MOs (a common level of abstraction are $\mathcal{SR}$–MOs). Also graph based

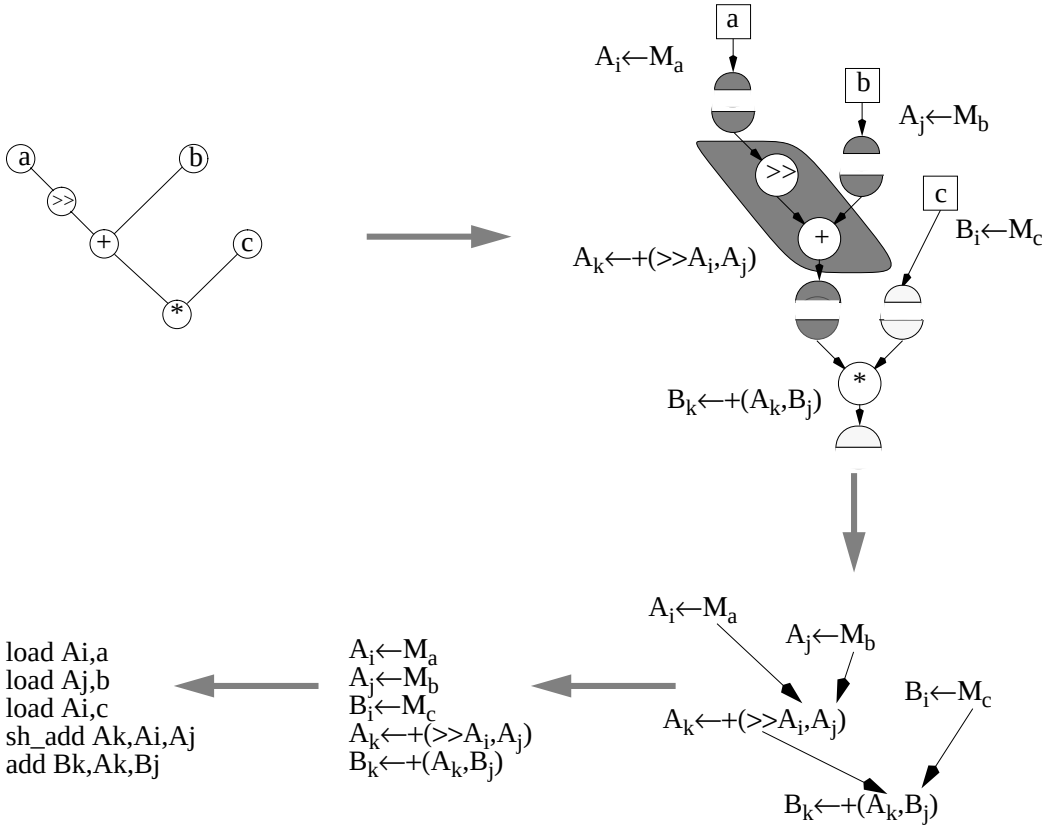


Figure 4.1: A Covering of an Expression Tree

or term based representations are possible outputs of a code selector. Most of the recent code selection techniques are performed on tree based intermediate representations of a program (e.g., DAGs; see section 2). It is assumed that common subexpressions (CSEs) were extracted and assigned to fresh variables. Each occurence of a certain CSE is substituted by the corresponding variable. Code selection can be performed on expression level, and statement level, but also on basic block level (e.g., datadependence graph or def–use chains [ASU86]). In the following we will be simply talking of input trees. Informally, a code selector tries to cover an input tree with machine operation patterns, such that there is no overlapping of patterns. This can be aslo seen as a decomposition into machine operation patterns. The storage resources of results of selected machine operation patterns must always correspond to the *use* in the input tree. I.e., that they are either identical or there exists a sequence of data transfers, that move the operand to the required storage resource. A covering that guaranties this is called a *legal covering*. Generally there exists more than one legal covering for a given tree. Conventional code selectors select the cheapest solution with respect to a given cost model. The task of finding a covering for an input tree for a fixed set of machine operation patterns is stated as **tree pattern matching**. The technique used for selecting the cheapest covering is incorporated into the tree pattern matching process and is based on the **dynamic programming** approach introduced in [AJ76].

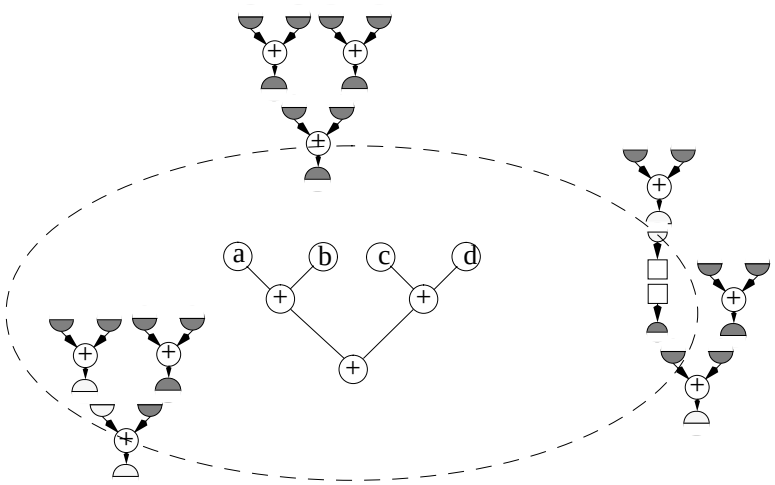


Figure 4.2: Some Legal Coverings of an Expression Tree

Example 4.1:

> In 4.1 a covering for the expression `(>>(a)+b)*c` from machine operation patterns of the architecture in figure 3.4 (page 23) is shown. The patterns are associated with $\mathcal{SR}$–MOs (here notated in register transfer language notation). Values are mapped to virtual registers. The $\mathcal{SR}$–MOs are sequentialized with respect to their data dependencies, also shown in figure 3.4. This sequentialization is not generally necessary, therefore either of the tree based or sequentialized form of versions can be passed to subsequent phases. An assembler like notation is also given in the figure. In figure 4.2 a set of possible coverings of machine operation patterns is shown. It is assumed, that the variables were allready loaded to storage resource A.

If the target machine is specified by a structural model, the machine operation patterns have to be extracted from the specification. In behavioral models they are explicitly represented by rules. These rules are based on **regular tree grammars**. Such a tree grammar can be automatically transformed into a *tree pattern matcher* by a **code selector generator**. The tree grammar specifies the set of all input trees that can be covered. Only legal coverings are selected by the tree pattern matcher. The principles of tree pattern matchers can be constituted on finite tree automata [FSW94, WM95]. Tree pattern matchers are able to find the complete set of coverings for an input tree, with respect to a given tree grammar. To enable the selection of minimum cost coverings, tree grammars are augmented with costs. These grammars are then called **weighted tree grammars**.

4.2 Formal Foundations of Tree Pattern Matchers

So far the notions tree pattern matching, covering, and tree pattern matcher were introduced informally. In this section the formal foundations of tree grammars are described. The relationship to machine operation patterns are exposed to utilize an imagination of what entities are described by regular tree grammars, and also how they are described. Finally, the construction of finite tree automata from regular tree grammars is shown. A detailed introduction can be found in [FSW94, WM95].

4.2.1 Tree Pattern Matching

The notions pattern and tree pattern matching will be constituted on a term based representations now. However, in the following the notions expression term and expression tree will be used as synonyms.

Definition 4.2.1 *A* **ranked alphabet** *is a finite set Σ of operator symbols together with a ranking function $\rho : \Sigma \rightarrow N_0$. Σ_k denotes the set $\{a \in \Sigma | \rho(a) = k\}$. The* **homogeneous tree language** *over Σ, T_Σ is the smallest set T such that*

- $\Sigma_0 \subseteq T$
- *If $t_1, \ldots, t_k$ are in T then $a(t_1, \ldots, t_k)$ for $a \in \Sigma_k$ is in T.*

Example 4.2:

> $\Sigma = \{+, >>, c\}$ with $\rho(+) = 2, \rho(>>) = 1, \rho(c) = 0$. c represents a constant value.
> Legal terms that can be constructed are e.g. $+(c, >> (+(c, c)))$ or $>> (+(c, >> (c)))$.

There are no restrictions to the operands of operators except that the number of operands of an application of an operator o must be equal to $\rho(o)$. An operand can be any legal term of T_Σ. The following definition enables to distinguish operands of different sorts and restricts the structure of terms.

Definition 4.2.2 *A* **signature** Sig_Σ *over a ranked alphabet Σ is defined as $Sig_\Sigma = (\mathcal{S}, \phi)$, where $\mathcal{S}$ is a finite set of sort symbols and ϕ is a type function defining the type of a certain $o \in \Sigma$, such that $\phi(o) = (s_1, \ldots, s_{\rho(o)}, s)$ with $s_1, \ldots, s_{\rho(o)}, s \in \mathcal{S}$. $\phi(o)$ will be denoted as $o : s_1, \ldots, s_{\rho(o)} \rightarrow s$; if $\phi(o) = s$ this will be denoted as $o :\rightarrow s$. A term of sort s is inductively defined as:*

1. *o is called a constant of sort s iff $o \in \Sigma_0$ and $o :\rightarrow s$;*
2. *if $t : s_1, \ldots, s_{\rho(t)} \rightarrow s$ and $t_1, \ldots, t_{\rho(t)}$ are terms of type $s_1, \ldots, s_{\rho(t)}$ respectively, then $t(t_1, \ldots, t_{\rho(t)})$ is a term of type s.*

T_Σ^s denotes the set of all terms of sort s and $T_\Sigma = \cup_{s \in S} T_\Sigma^s$ is the set of all terms over $\mathcal{S}$.

Signatures restrict the structure of terms, i.e., the way that terms can be constructed. The notions *operator symbol* and *sort symbol* reflect that signatures denote pure syntax. Generally the notions sorts and operators are used in the context of a certain interpretation and reflect the semantical level of description. In the following we will use sorts and sort symbols, and operators and operator symbols as synonym notions.

Example 4.3:

```
Signature EX
Sorts:
   sh,n
Operators:
   c  : -> n
   +  : sh,n -> n
   >> : n -> sh
```

The type function restricts the left operand l of a term $+(l, r)$ to be of the form $>>(t)$ for any term t of sort n. $+(>>(c), c)$ is a legal term of sort n but $+(c, c)$ and $+(+(c, c), c)$ are not.

Definition 4.2.3 *Given a signature Sig_Σ. V_s is a set of variables of rank 0 and of sort s and $V = \cup_{s \in S} V_s$. A member of $T_\Sigma(V) := T_{\Sigma \cup V}$ is called a* **pattern**. *A pattern is called* **linear** *if no variable occurs more than once. Two patterns are said to be equivalent, if they are identical up to variable renaming.*

Example 4.4:

If we consider the example signature and the sets $V_n = \{X, Y\}$ and $V_{sh} = Z$, then $>>(X)$ and $+(Z, +(>>(X), Y))$ are legal pattern. $+(X, Y)$ is no legal pattern because the variable X is not of sort sh. A pattern defines a set of terms such that any term of this set can be constructed by replacing the variables within with the pattern by a term of the corresponding sort. In the case of non–linear patterns, equal variables must be substituted by the same term.

We now define the meaning of pattern matching. Therefore, substitution for variables must be further specified.

Definition 4.2.4 *A* **substitution** *is a mapping $\Theta : V \to T_\Sigma(V)$. Θ is extended to a mapping $\Theta : T_\Sigma(V) \to T_\Sigma(V)$ by $t\Theta := x\Theta$ if $t = x$ and $t\Theta := a(t_1\Theta, \ldots, t_k\Theta)$ for $t = a(t_1, \ldots, t_k)$. A substitution Θ is also written $[t_1 \backslash x_1, \ldots, t_j \backslash x_j]$ for a set of variables $x_1, \ldots, x_j \subseteq V$, such that $x_i\Theta = t_i$ for $1 \leq i \leq j$.*

Definition 4.2.5 *A pattern $\tau \in T_\Sigma(V)$* **matches** *a tree t if there is a substitution Θ such that $\tau\Theta = t$.*

Example 4.5:

Assume that $V_n = \{X, Y\}$ and $V_{sh} = \{Z\}$. Then $+(Z, X), >>(Y)$ and $+(>>(X), Y)$ are legal patterns. $+(>>(X), Y)$ matches $+(>>(c), +(Z, c))$ with substitution $[c \backslash X, +(Z, c) \backslash Y]$.

Definition 4.2.6 *An* **instance** *of the* tree pattern matching problem *consists of a finite set of patterns $T = \tau_1, \ldots, \tau_n \subset T_\Sigma(V)$ together with an input tree $t \in T_\Sigma$. The* solution *to the tree pattern matching problem for this instance is the set of all pairs (n, i) such that pattern τ_i matches t/n.*

The tree pattern matching problem consists in finding all positions of subterms in an expression tree represented by a term t, that can be matched by a certain pattern.

Definition 4.2.7 *An algorithm that returns a solution for every input tree $t \in T_\Sigma$ for the tree pattern matching problem (T, t), $T = \tau_1, \ldots, \tau_n \subset T_\Sigma(V)$, is called a* **tree pattern matcher** *for T.*

Definition 4.2.8 *An algorithm that on every $T = \tau_1, \ldots, \tau_n \subset T_\Sigma(V)$ returns a tree pattern matcher for T is called a* **tree pattern matcher generator**.

Most tree pattern matchers work on linear patterns. Therefore, they do not have to check for common subexpressions. But there are approaches that also work with non-linear patterns. First, one can introduce tests for equality for subtrees, but this may not be efficient because the pattern matcher has to visit some subtrees several times. Second, a pattern matcher can execute all equality tests in advance [DST80] (for further discussion on common subexpressions consult [ASU86]).
If each variable that occurs in a certain pattern is replaced by its corresponding sort (assuming a symbolic representation of that sort), a pattern can be regarded as the formal counterpart of a machine operation pattern and we observe the following correspondences:

1. storage resources correspond to the sorts in signatures;

2. the sort of a pattern corresponds to the destination storage resources of a machine operation pattern (e.g. the sort of $+(>>(X), X)$ is n and the destination storage resource of $B \leftarrow +(>>(A), A)$ is B);

3. operators in the signature correspond to operators in the machine operation patterns , but they also represent constants and variables.

The basic differences are that:

1. transfer machine operations do not have a corresponding entity; they can only be specified by introducing extra operators with the purpose of sort casting, i.e., mapping a sort to another sort;

2. machine operation patterns allow that the same machine expression pattern can be assigned to different storage resources; in the terminology of sorts this assigns different sorts to the same pattern (or term); a solution to overcome this again would involve sort casting operators.

As signatures themselve do not incorporate constructs for the specification of complex patterns, for specifying machine operation patterns, additional constructs are necessary. A common frame–work for introducing storage resources and complex machine operation patterns together with transfer machine operations is given by regular tree grammars, also overcoming the introduction of extra sort casting operators.

4.2.2 Regular Tree Grammars and Tree Parsing

Definition 4.2.9 *A* **regular tree grammar** G *is a triple* (N, Σ, P) *where*

- N *is a finite set of* nonterminals,
- Σ *is a ranked alphabet of* terminals,
- P *is a finite set of* rules *of the form* $X \longleftarrow t$ *with* $X \in N$ *and* $t \in T_\Sigma(N)$.

If we consider regular tree grammars with regards to machine descriptions, the terminals of a regular tree grammar represent operators, constants and variables of the intermediate representation. The nonterminals correspond to the storage resources.

Example 4.6:

In figure 4.3 the regular tree grammar for the example hardware in 3.4 is represented. A, B, and M denote the nonterminals. All variables occuring in the intermediate representation are mapped to the operator var. Such abstractions from certain sets (like variables or constants) can be made if the values have no impact on the matching process. If certain values must occur in a pattern (e.g. the constant 0 in a pattern $+(0, X)$) a specific operator can be specified.

```
A ← +(>>(A),A)          B ← +(A,A)
A ← -(>>(A),A)          B ← +(B,A)
A ← +(A,A)              B ← +(A,B)
A ← -(A,A)              B ← +(B,B)
                        B ← *(A,A)
B ← M                   B ← *(B,A)
A ← M                   B ← *(A,B)
M ← A                   B ← *(B,B)
M ← B

M ← var
```

Figure 4.3: Regular Tree Grammar

A signature can be easily transformed to a tree grammar by observing the following correspondence:

Definition 4.2.10 *Let* $p : X \longleftarrow t$ *be a rule of* P *of a grammar* (N, Σ, P). P *is of* **type** $(X_1, \ldots, X_n) \rightarrow X$*, if the j-th occurence of a nonterminal in* t *is* X_j.

The nonterminals correspond to the sorts of a signature. The drawback of signatures, only being able to express transfer machine operations by introducing extra sort casting operators is not inherent to regular tree grammars. In regular tree grammars the specification of data movements is embedded in the specification technique given by the following class of rules:

Definition 4.2.11 *Let* $p : X \longleftarrow t$ *be a rule of* P *of a grammar* (N, Σ, P). P *is called a* **chain rule** *is* $t \in N$*, otherwise a* **non-chain rule.**

In figure 4.3 $M \longleftarrow A$, $M \longleftarrow B$, $A \longleftarrow M$, and $B \longleftarrow M$ are chain rules. They represent the data movements between certain storage resources. Nonterminals are also used for factoring with the purpose of reducing the amount of tree grammar rules necessary for specifying a certain target machine.

Example 4.7:

In figure 4.4 an example is shown for the grammar in figure 4.3, where factoring is used to reduce the number of combinations of operands accessable to the functional unit $F2$. The nonterminal O summarizes all possible sources of operands to $F2$.

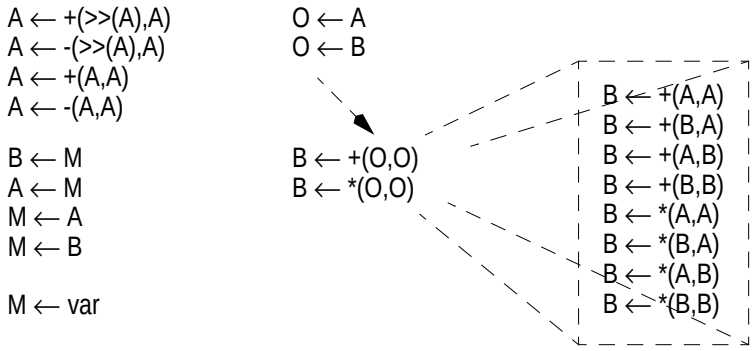


Figure 4.4: Factoring

We will no specify the notion covering. It describes, how a covering of a certain expression tree is constructed with respect to given regular tree grammar. We assume that each rule of the tree grammar can be identified by a unique symbol p, denoted by $p : X \longleftarrow t$.

Definition 4.2.12 *Given a regular tree grammar $G = (N, \Sigma, P)$ and a $X \in N$. A X–* **derivation tree** *for a tree $t \in T_\Sigma(N)$ is a derivation tree $\Psi \in T_P(N)$ satisfying the following conditions:*

- *If $\Psi \in N$ then $t = \Psi$.*
- *If $\Psi \notin N$ then $\Psi = p(\Psi_1, \ldots, \Psi_n)$ for some rule $p : X \longleftarrow t' \in P$ of type $(X_1, \ldots, X_n) \rightarrow X$, such that $t = t'[t_1 \backslash X_1, \ldots, t_n \backslash X_n]$ and Ψ_j is the X-derivation tree for t_j.*

An X-derivation for a certain term represents one possible covering for that term. Conversly, it also reflects the necessary application of grammar rules for constructing a certain input tree. An example of a B-derivation is shown in 4.5.

Definition 4.2.13 (Language of Grammar G) *For $X \in N$ the* language of G relative to X *is defined as:*

$L(G, X) := \{t \in T_\Sigma | \exists \Psi \in T_P(N) : \Psi$ *is a X-derivation tree for* $t\}$.

a:	A ← +(>>(A),A)	j:	O ← A
b:	A ← -(>>(A),A)	k:	O ← B
c:	A ← +(A,A)		
d:	A ← -(A,A)		
e:	B ← M	l:	B ← +(O,O)
f:	A ← M	m:	B ← *(O,O)
g:	M ← A		
h:	M ← B		
i:	M ← var		

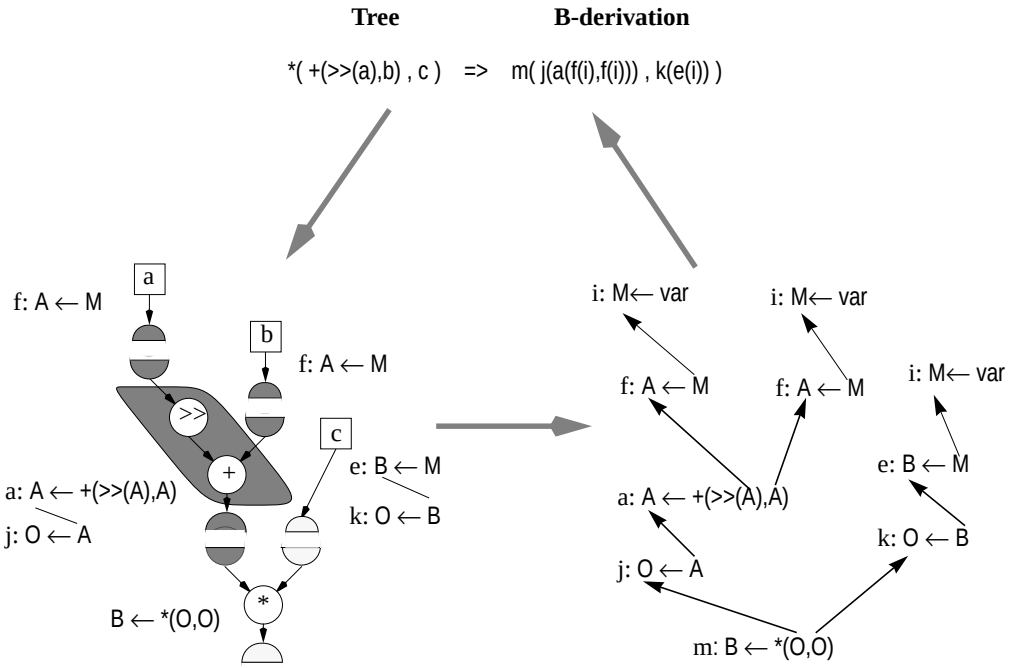


Figure 4.5: B–Derivation

The language of a grammar specifies the set of terms that can be covered. This set should be the set of all expected terms of the intermediate representation. Therefore it is important that the regular tree grammar completly specifies this set. Otherwise a code selector could fail in trying to cover certain terms of the intermediate representation.

Definition 4.2.14 (The Tree Parsing Problem) *An instance of the* tree parsing problem *consists of a regular tree grammar G together with a nonterminal $X \in N$ and an input tree $t \in T_\Sigma$. The solution for this instance is the set of all X-derivation trees of G for t.*

The tree parsing problem consists in finding all possible coverings of a given input term, with respect to a certain storage resource.

Definition 4.2.15 (Tree Parser) *A* tree parser *for a regular tree grammar G is an algorithm that, for every input tree t, returns the solution of the parsing problem for given nonterminal X.*

Definition 4.2.16 (Tree Parser Generator) *A* tree parser generator *is an algorithm that, for every regular tree grammar G, returns a tree parser for G.*

The difference between the tree matching problem and the tree parsing problem is, that a solution of the tree parsing problem results in a set of complete coverings for a given tree. In contrast to this, a solution to the tree matching problem returns a set of positions where certain pattern match a given tree. Note that a tree parser not necessarily must be implemented by a parser, e.g. LR-parser or LALR-parser. The definition only says *what* the algorithm does and not *how* it does it. In the following we will use the notion tree pattern matcher as a synonym for tree parser, as the aim of a tree pattern matcher is to find a complete covering.

4.2.3 Finite Tree Automata

The principles of tree pattern matchers can be constituted on finite tree automata. In this subsection only the basic notions and principles of tree automata can be described. It is illustrated how regular tree grammars are transformed into a corresponding tree automaton accepting the language specified by the grammar.

Definition 4.2.17 *A* **finite tree automaton** *is a 4-tuple $TA = (Q, \Sigma, \delta, Q_F)$ where*

- *Q is a finite set of states;*
- *$Q_F \subseteq Q$ is a set of* final accepting *states;*
- *Σ is a finite ranked input alphabet;*
- *$\delta \subseteq \cup_{j \geq 0} Q \times \Sigma_j \times Q^j$ is the set of transitions.*

A tree automaton A is called deterministic *if for every $a \in \Sigma_k$ and every sequence $q_1, \ldots, q_k$ of states there is at most one transition $(q, a, q_1, \ldots, q_k) \in \delta$. In this case δ can be written as a partial function.*

The following definition specifies the computation of a finite tree automaton TA for determining if a given input tree is in the accepted language. The language $L(TA)$ accepted by TA consist of all trees for which an accepting computation exists.

Definition 4.2.18 *Let Σ be a ranked alphabet and Q a finite set of states. The* **extended ranked alphabet** *is defined as $\Sigma \times Q$ and it's operators consist of* pairs *of operators from Σ and states.*
*Let $q \in Q$: A q-***computation** *Φ of the finite tree automaton TA on the input tree $t = a(t_1, \ldots, t_k)$ is inductively defined as a computation tree $\Phi = \langle a, q \rangle(\Phi_1, \ldots, \Phi_k) \in T_{\Sigma \times Q}$ where Φ_j is the q_j-computation for the subtrees t_j, such that $(q, a, q_1, \ldots, q_k)$ is the transition from δ. Φ is called accepting if $q \in Q_F$.*

The transitions of the automaton correspond to the types of the operator symbols, given by the input alphabet of a tree automaton. An Automaton A_τ for the pattern matching problem that matches a single linear pattern $\tau \in T_\Sigma(V)$ is constructed as follows: First we assume that there is an unspecified state $\perp_s$ for every sort $s \in S$. Every variable of sort s in τ is replaced by $\perp_s$. $A_\tau = (Q_\tau, \Sigma, \delta_\tau, Q_{\tau,F})$ is defined as

- $Q_\tau := \{s | s \text{ is a subpattern of } \tau\} \cup \{\perp_s | s \in S\}$;
- $Q_{\tau,F} := \tau$;
- $(\perp_s, a, \perp_{s_1} \ldots \perp s_{\rho(a)}) \in \delta_\tau$ for all $a \in \Sigma$, and if $s \in Q_f$ and $s = a(s_1, \ldots, s_k)$ then $(s, a, s_1, \ldots, s_k) \in \delta_\tau$.

For every tree of sort s there is a $\perp_s$–computation and for a tree t there is a pattern τ there is a $\tau - -computaion$ if and only if τ matches t.
The automaton can be extended to accept a set $T = \tau_1, \ldots, \tau_n$ of patterns. Determining which patterns match a tree t consists of computing all accepting states of the tree automaton. A method to determine this set by one computation can be obtained by means of **subset construction**, yielding a deterministic tree automaton performing all possible computations concurrently.
The approach can further be extended to q–computations that represent X-derivations of a given tree grammar $G = (N, \Sigma, P)$ and $X \in N$ for a certain tree t [FSW94]. The tree parsing problem for a regular tree grammar is therefore reduced to the problem of determining all accepting computations of a finite tree automaton. A tree automaton can be constructed that performs all possible X–derivation simultaniously and finds all possible coverings during one computation.
Tree automata can be further extended to select the a minimum cost derivation, when given a weighted regular tree grammar, by integrating the costs into the transitions of the tree automaton. These automata are the called **weighted tree automata**. For further details of constructing tree automata from regular tree grammars consult [FSW94, WM95].
The basic principle of determining the coverings is now illustrated in a small example, using the factored tree grammar introduced in fig 4.4. The example schematically shows a bottom up computation, i.e. deriving the compuation tree starting at the leafs of the input tree and applying the rules of the tree grammar.

Example 4.8:

As already stated, variables are mapped to the terminal var. Therefore the rule $i : M \leftarrow var$ is the only rule applyable for the leafs, shown in figure 4.6 (a). The leafs of the input tree are associated with the applied rules. In the next step shown in figure 4.6 (a), all chaining rules $X \leftarrow M$ are applied. Chaining rule represent ϵ–transitions of the automaton, therefore consuming no operators. We assume that each node of the input tree is associated with the set of storage resources SR that are reachable sofar. In figure 4.6 (b) this set consists of M for each leaf. We also assume that each node is associated with the rules that match the node and a set CR of chaining rules (shown in the dark gray shaded boxes). The set CR is succesivly constructed from the set SR. Initially this set is empty and the set SR contains the destination nonterminals of the non-chaining rules that matched the considered node. Successively all chaining rules $cr : X \leftarrow Y$ are added to CR, such that $cr \notin CR$ and $Y \in SR$; when the chaining rule is added SR is updated to $SR \cup \{X\}$. The set SR denotes the possible $X - derivations$ for $X \in SR$. It also denotes the set of all data movements with corresponding pathes for the value associated with the node (i.e. the value of the subtree whose root is the node considered. In figure 4.6 the possible data movements are shown in the grey shaded areas.

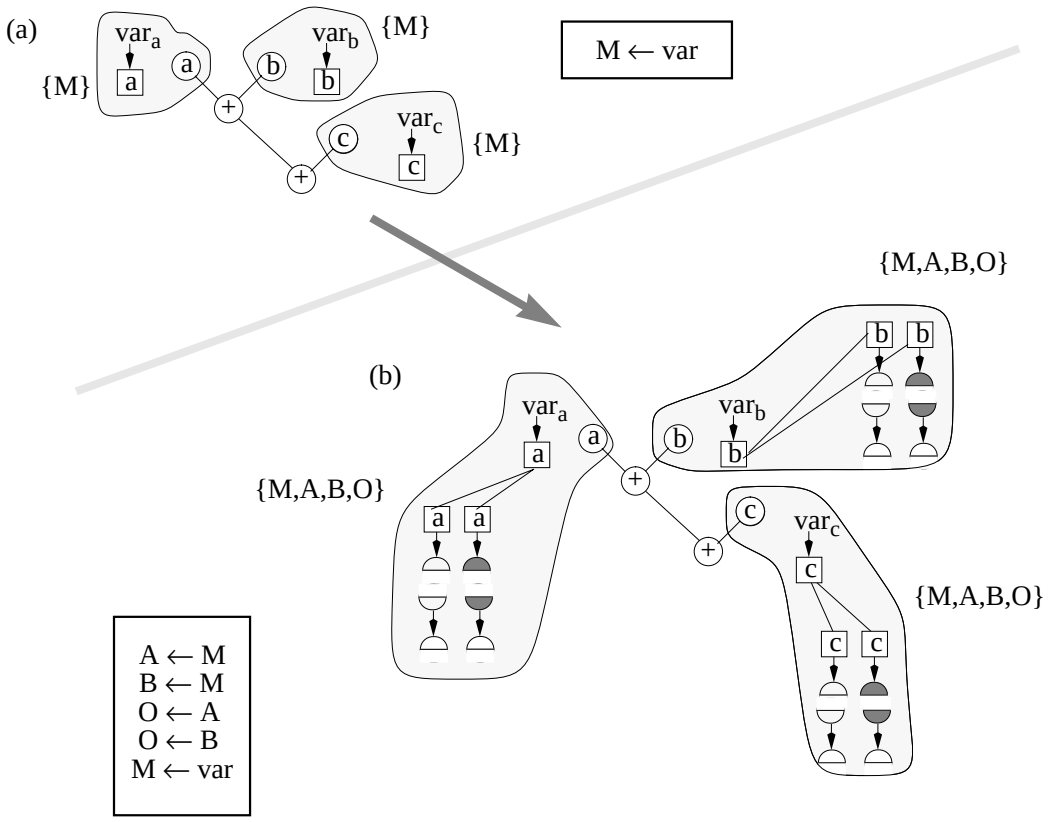


Figure 4.6: Determining the Coverings

In the next step we travers the internal nodes in a bottom–up manner. A rule $r : X \leftarrow p$ with type $X_1, \ldots, X_n \rightarrow X$ matches at a node n if the pattern p matches the subtree at the node n and each $X_i \in SR_i$ of the corresponding operand nodes (figure 4.7 step (c,d)). In figure 4.7 (c) there are two possible rules matching the node $+(a, b)$. Again, all possible chaining rules are added in the mentioned way.

In the example we see, that an input tree is covered with non-chaining rules and chaining rules. The non–chaining rules cover certain patterns for the input tree. The chaining rules don't cover any pattern. They represent the possible data movements from the storage recources where the machine operations (corresponding to the patterns of the non–chaining rules) will store their results. Therefore the set SR_n of a certain node n in the input tree denotes the set of all possible storage resources to which a value can be computed. Additionally, the set CR_n implicitly contains the set of all possibe pathes from storage resources to storage resources of SR_n. This also denotes the set of possible spill pathes.
If all statements of a basic block are covered separately the storage resources of used variables should be taken into account. After all input trees are covered, the SR sets should be reduced to the necessary amount. It should only contain those storage resources that are incorporated in the machine operations that will use the value, all storage resources from machine operations that produce this value and all storage resources involved in data movements between definition and usage. The final tree contains all legal coverings for the input tree. In figure 4.8 the main aspects of a code selector generators and its corresponding formal foundations

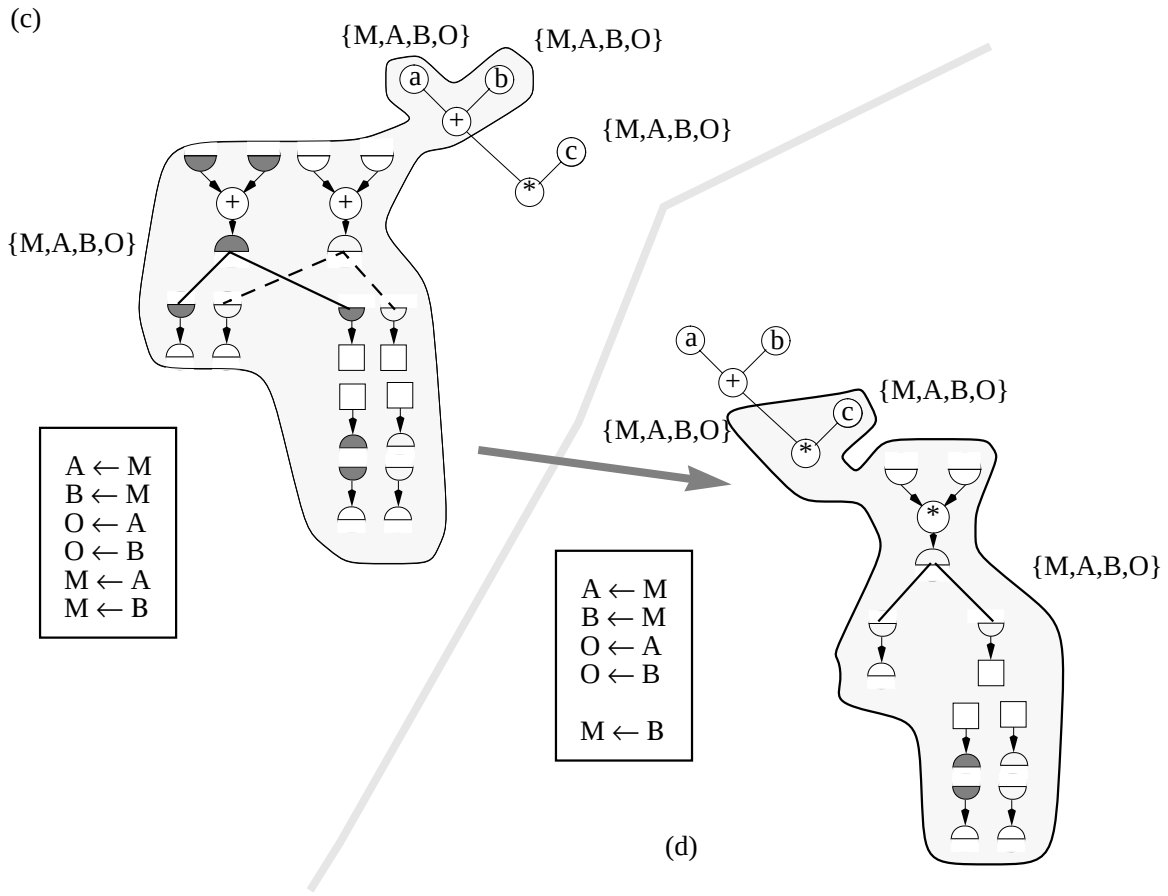


Figure 4.7: Determining the Coverings

are summarized.

4.3 Generation of Code Selectors

Many efforts have been made for gaining efficient tree pattern matchers and efficient code selector generators. There are various approaches in recent years. Initiated by the work of Graham–Glanville [GR77] LR-parsing techniques where used for pattern matching, whereby the target machine specification was defined by a context free grammar. A parser generator was used for generating the code selector. A very good summary of first approaches using grammars and attributed grammars for specifying code selectors can be found in [GFH82]. Limitations of the approaches are also shown. The basic problem is that input trees of ambiguous grammars were not covered propriately. Best cost coverings were only approximated. Tree pattern matching with dynamic programming [AJ76] constituted a solution to this problem (figure 4.10). The pattern matching process is based on tree parsing using weighted bottom–up tree automata.

The pattern matching process is generally based on tree parsing using weighted bottom–up tree automata. There are several techniques for implementing tree automata, e.g. state transition tables or decision trees. The informations computed by dynamic programming can

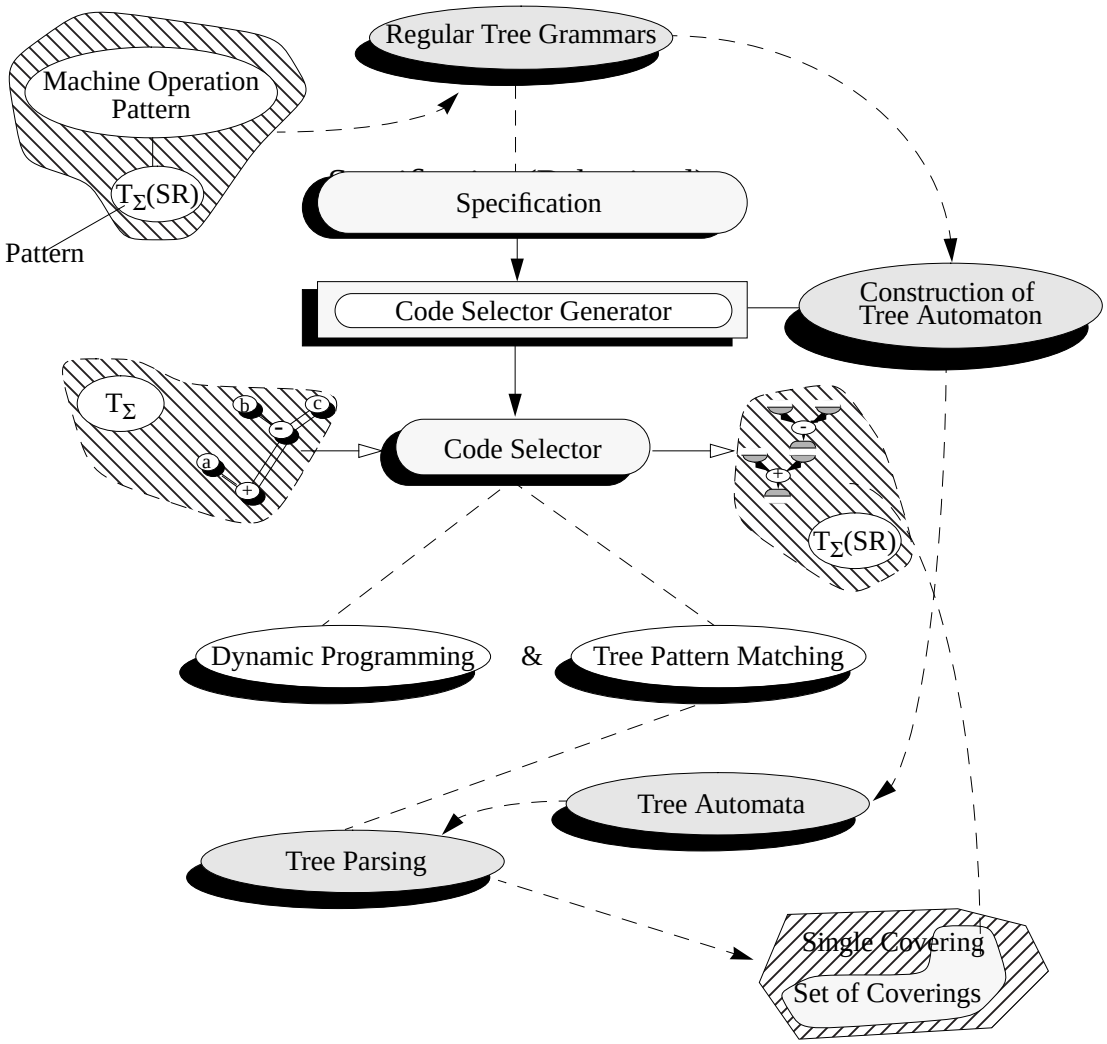


Figure 4.8: Foundations of Code Selector Generator

be generated at compile-compile time and can be integrated in state transition tables (figure 4.11) [PLG88, Hen89c, Hen89a, Hen89b, FSW94]. One disadvantage of this approach is the loss of expressiveness, i.e. the costs are restricted to constants (e.g. burg).
Many code-selector generators use tree pattern matching and dynamic programming. They produce tree pattern matchers that make two passes over expression trees. The first pass is bottom up and finds a covering with minimum costs. The second pass is top down and produces the final output of the code selector, i.e. a representation of the target machine program. Examples for code generator generators based on this model are: *BEG*[ESL89], *Twig*[AGT89], *burg* [FHP92b], *iburg* [FHP92a], and *CBC* [FHKM94]. The informations computed by dynamic programming can be generated at compile-compile time and can be integrated in state transition tables [PLG88, Hen89c, Hen89a, Hen89b, FSW94]. One disadvantage of this approach is the loss of expressiveness, i.e., the cost model of the tree grammar is restricted to constants (e.g., burg).

- **BEG** tree pattern matchers are hard coded and mirror the tree patterns like recursive decent parsers mirror their input grammars. Dynamic programming is used at compile

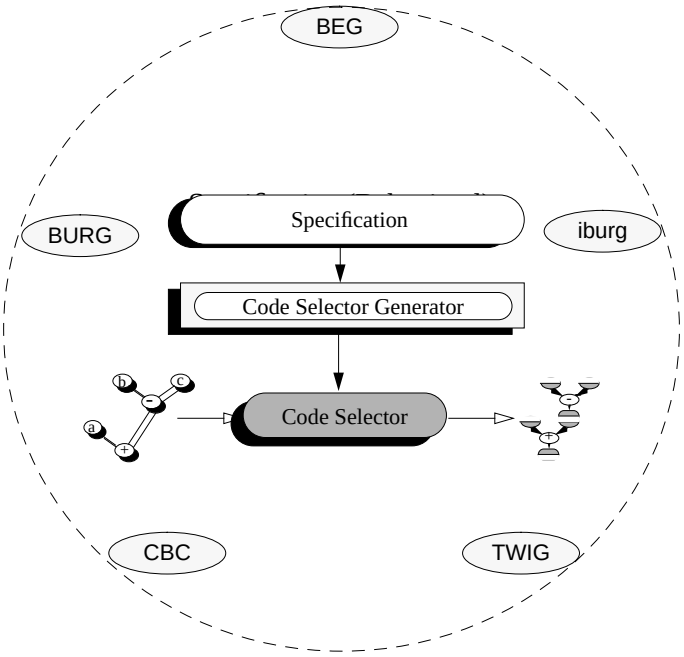


Figure 4.9: Code Selector Generators

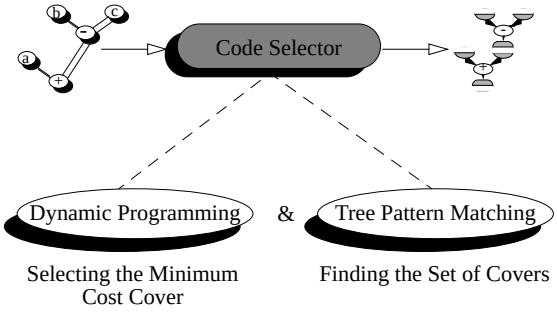


Figure 4.10: Dynamic Programming

time to identify the minimum cost cover.

- **Twig** matchers use a table-driven variant of string matching. This representation identifies all possible matches at the same time, resulting in a higher overhead. Dynamic programming is used at compile time.

- **burg** uses *BURS* (bottom-up rewrite system) theory [PLG88] to move dynamic programming to compile-compile time. A main disadvantage of BURS is, that costs must be constants; systems that delay dynamic programming to compile time allow arbitrary cost models permitting dynamic computations of costs associated with rules. This also allows to propagate context-sensitive informations from subtrees.

- **iburg** reads burg specifications. In contrast to burg it perfoms dynamic programming at compile time and like BEG it is hard-coded. It is simpler than burg and is amenable for user defined modifications.

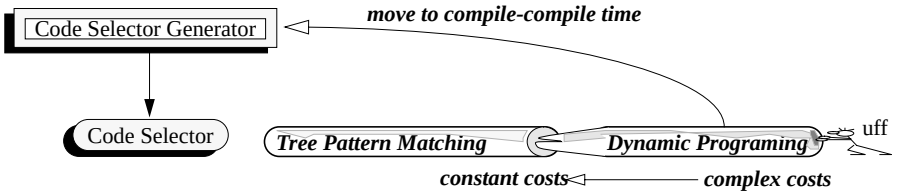


Figure 4.11: Moving Dynamic Programming to Compile Time

- **CBC** [FHKM94] is based on a machine description based on the hardware description language *nML*. The machine specification is tranformed to an iburg specification and the code selector is generated by an extended version of iburg. CBC was developed in the context of irregular architectures. The major differences to classic code selectors are:

  – handling of complex data paths;

  – taking into account the different word length of storage resources (termed type handling);

  – considers instruction level parallelism by delayed binding of machine resources;

  – handling of DAGs, thus also considering common subexpressions during code selection;

  – machine based description.

  CBC also tries to take into account common subexpressions that additionally transcends basic blocks by a technique called *heuristic node duplication*. Hereby a control data flow graph is modified in order to create more complex machine operation patterns across basic block boundaries by node dupplication, but only at places where this will lead to improved code.

4.3.1 Code Selector Specifications

The major goal in the research of specification techniques is gaining more expressiveness. There are several approaches to extend the specification techniques for gaining more expressive power, making specifications more readable, and easier to develop. One aim is a comfortable incorporation of algebraic rules. They cannot be expressed appropriately using regular tree grammars, while maintaining a readable structure of the grammar rules. The specification of algebraic rules can blow up the size of the patterns of the rules. Thus, the rules gain unreadability and are hardly to understand. Term rewriting rules allow the specification of algebraic rules in an intuitive notation, that is easy to understand and prevents the designer from errors. Restricted sets of term rewriting systems, can be transformed automatically to regular tree grammars. Therefore, conventional tree pattern matchers can be used for driving the task of code selection.

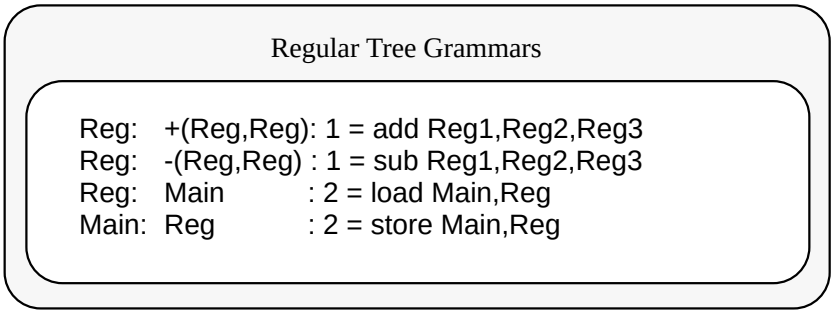


Figure 4.12: Tree Reduction Rules

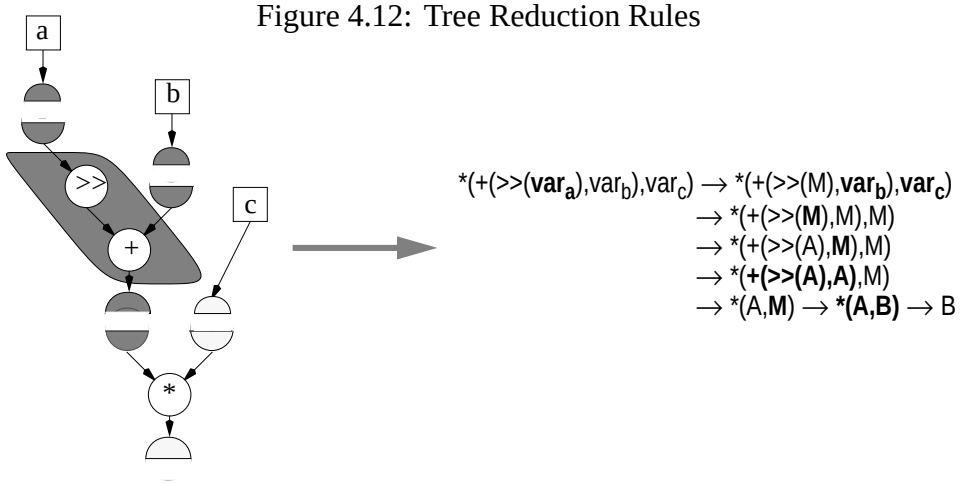


Figure 4.13: Reduction Sequence

Tree Grammar Based Approaches

The common specification techniques used are based on weighted regular tree grammars. Tree grammars are represented by a set of tree reduction rules of the form $X \leftarrow pattern[cost] : action$. The action part is used to generate the final output of the code selector, e.g., a sequence of $\mathcal{A}$–MOs. Generally, it initiates some user–defined procedures that emit target machine code. The notion *tree reduction rules* reflects a view of the tree parsing process as the reduction of a certain input tree to a certain nonterminal, by using the rules of the tree grammar as rewriting rules. A pattern detected in the input tree is replaced (or better substituded) by the nonterminal on the left hand sight of the corresponding rule. A reduction sequence is shown in figure 4.13.
Tree grammars are ambiguous, i.e. there exists more then one covering for a certain expression tree. Each covering represents correct code, but with differently code quality given by the costs of the rules. A very good summary of first approaches using grammars and attributed grammars for specifying code selectors can be found in [GFH82]. Recent approaches based on regular tree grammars are e.g. BEG, Twig,burg and iburg. The following extended BNF grammar defines the format of *burg* and *iburg* specifications:

```
grammar ::= {dcl} %% {rule}
dcl     ::= %start nonterm
        |   %term {id=integer}
rule    ::= nonterm:tree = action [cost];
```

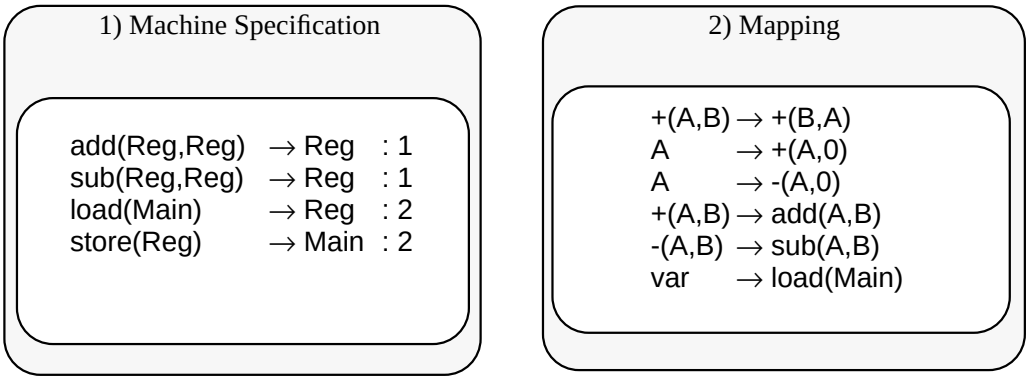


Figure 4.14: Term Rewriting System

```
pattern ::= term(pattern,pattern)
        |   term(pattern)
        |   term
        |   nonterm
action  ::= integer
cost    ::= integer
```

The declaration part `dcl` defines the nonterminals and the ranked alphabet defining the terminals of the regular tree grammar. The tree reduction rules differ a little from the introduced format. The costs are optional and are given after the action. The actions are specified by integers that are associated with user-defined actions for emitting corresponding machine instructions. A burg specification only permitts costs given by constants. The cost of a covering are then defined as the sum of all costs associated with the machine operation patterns of the covering. In iburg arbitrary cost computations are allowed.

Term Rewriting Rules

Recent approaches try to extend the expressive power of the specification techniques. A common aspect of most approaches is the integration of term rewriting rules. Regular tree grammars are augmented with a set of term rewriting rules, permitting concise definition of algebraic rules.

Term rewriting rules allow to transform the structure of the input tree by rewriting a certain complex pattern to a new (generally semantically equivalent) complex pattern. Thus new patterns can be matched, not occuring in the input tree without rewriting, which are effectively supported by certain machine operations available on the target machine. Additionally it is possible to substitute nonterminal or terminal symbols with complex patterns. Such rules enable to insert patterns that do not necessarily have to be determined when generating the input tree. E.g., if there is a set of alternative addressing modes for specifying an operand, a proper mode can be selected during code selection. These rules enable to incorporate decisions into code selection, that usually are due to other tasks of code generation or the front–end.

BURS theory introduced by Pelegri–Llopart [PLG88] allows to specify term rewriting rules fulfilling the restriction of being *k–burs* also called *finite–burs*. This restriction requires that

every reduction sequence must be reduced to *j–normalform* in finite steps (see [PLG88]). A rewriting rule of the form $a \rightarrow b(a)$ does not fullfill this restriction. Thus, rules like $A \rightarrow plus(A, 0)$ are not supported.
The approach of H. Emmelmann also incorporates term rewriting rules [Emm92]. In his approach, the machine instructions are represented as terms also. I.e., that not only the intermediate representation is term based. The intermediate representation is mapped to a legal target machine term by applying rewriting rules, describing the transformations from the intermediate representation to target machine terms. The regular tree grammar is used to define the set of legal target machine terms. Only terms with an existing covering with respect to the machine grammar are legal target machine terms. I.e., that the target machine term must be reducable to a nonterminal of the grammar. The specification technique allowes to specify terms rewriting rules like $A \rightarrow plus(A, 0)$. The code selector keeps track of the termination of applying such rules. The two specification parts are transformed into a conventional pattern matching based code selector specification. Therefore all techniques known to build generators can be used. This also reflects, that it is possible to express the algebraic rules by regular tree grammars. But the specification by term rewriting rules is much more concise. The specification seems to be more naturally and easier to develope with regards to specify correct rules. In figure 4.15 the basic issues and their interdependemncies are illustrated.

4.4 Support of Architectural Features

This section is concerned with examining what features of irregular architectures are coped/not coped by tree pattern matching. We will first consider what features are expressable using regular tree grammars. Thereafter, aspects that are coped by the tree pattern matcher are examined. Regular tree grammars enable to denote the following aspects:

- *distributed register sets:* each storage resource is denoted by a certain nonterminal symbol; therefore the set of distributed storage resources is specified by the according nonterminals;

- *special purpose registers:* e.g., address register can be specified by assuming that they represent a certain storage resource. This is achieved by introducing a corresponding nonterminal for denoting the register or registers. Uses and definitions in machine operation patterns that have access only to such special registers are specified with the corresponding nonterminals.

- *register classes:* a *chaining rule* of the form $X \leftarrow\!\!- Y$ can describe a transfer operation. It may also denote a subset relation between certain sets of registers. Hereby a hierarchy of register classes can be defined. This allows to restrict machine operations to certain register access.

- *complex data routes:* complex data routes are detected in the tree pattern matching process. Only coverings with legal connectivities are selected. Thus the tree pattern matchers checks the reachability of definitions and uses.

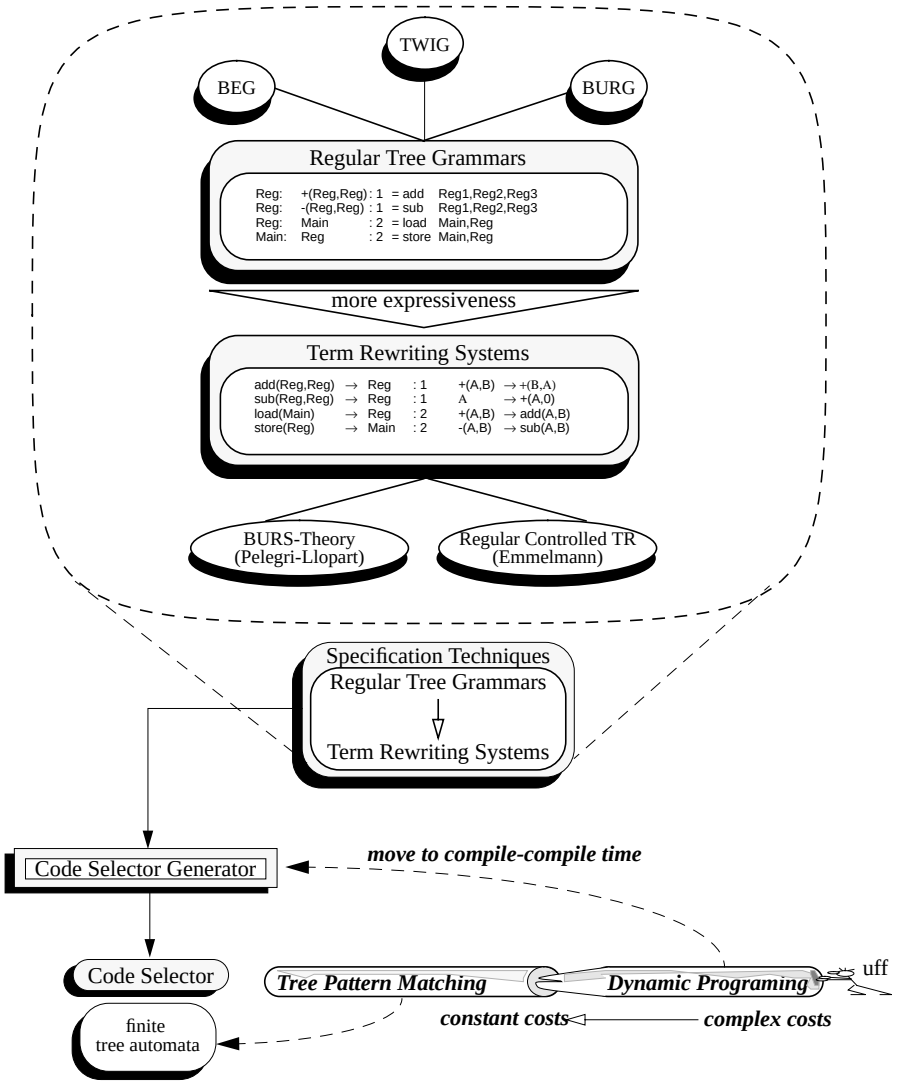


Figure 4.15: Code Generator Generators

Example 4.9:

Consider the registers in figure 4.16. The register R set is subdivided into the register classes A and B each denoting a certain subset of the main register set. The left operand of the `shift_add` operation must reside in register set A, the right operand in register set B or in the $Akku$. The addition has access to the complete register set R. To enable the pattern matcher to match operands of the addition that were produced by the `shift_add` operation two chaining rules are introduced: $R \leftarrow A$ and $R \leftarrow B$; they denote that either A and B belong to the complete register set. They can be regarded as virtual transfer operations. Data movements to memory can only be performed from $Akku$. Therefore a transfer operation from the register set to $Akku$ exists.

General Problems

With respect to a sequential view, data routes leading to minimal costs are selected. *Spilling costs* are not considered during *tree pattern matching*. In irregular architectures, complex data

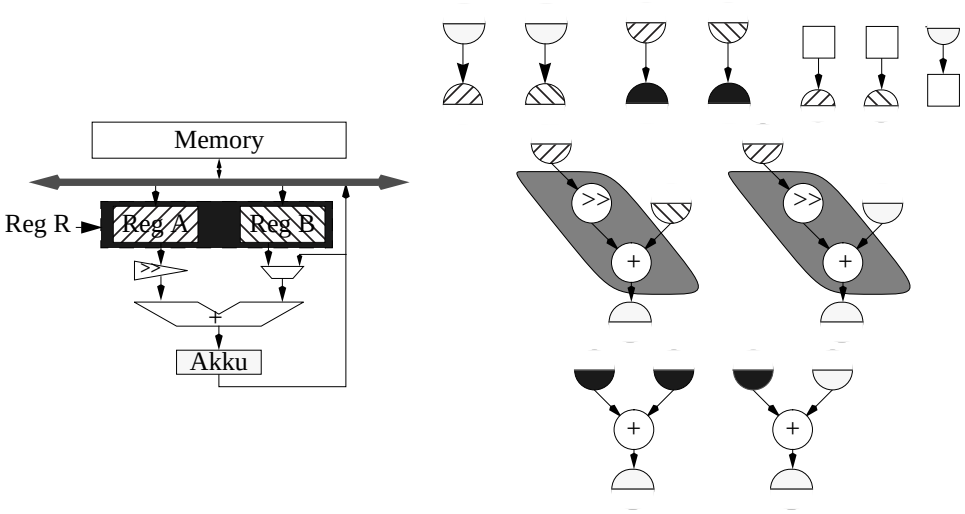


Figure 4.16: Irregular Register Sets

paths may induce a spilling accross a route of register sets. Spilling can have much impact on the quality of the selected code. In order to compute spill costs effectively, knowledge of the locations of other values and the order for executing subtrees is necessary. This implies, that the costs of subtrees of a certain node are not independent anymore. But: this violates the condition of the input tree to be executable contiguously, a necessary supposition for perfoming dynamic programming. An analoguous problem arises in the context of *instruction level parallelism*, which cannot be supported appropriatly by tree pattern matching. The minimal cost covering must not be optimal with respect to parallelism. Again, the problem is, that dynamic programming is based on the supposition, that an optimal solution of an input tree can be constructed from optimal solutions for the subtrees [ASU86, AGT89]. If the target machine allows certain subtrees of an operator to be executed in parallel, the necessary supposition is violated.
The basic problem is the consideration of global, context sensitive **and** mutual dependent information, which has to be accessable when selecting a certain covering (see figure 4.17). Statements for determining the quality of spilling and exploiting parallelism are based on informations of the complete covering. I.e., the informations that are necessary for selecting an appropriate covering are only known when a covering is selected (a quite unsatisfying situation). The consequence is the selection of a certain covering, which can restrict the subsequent tasks of code selection.

Overcoming the Problems

A solution for partially overcoming the problems is to delay the binding of machine resources as long as possible, while preserving the traditional scheme of tree pattern matching. For each operation an $\mathcal{A}$–MO is selected, offering a set of possible machine operations. This enables the subsequent tasks in choosing among a set of machine resources. Another solution is the integration of code selection into subsequent tasks. Tree pattern matchers are able to effectively compute the complete set of coverings for a given input tree, with respect to a

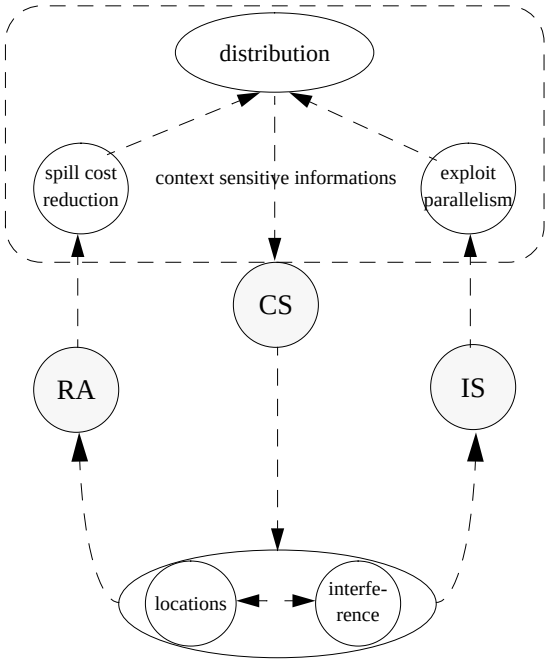


Figure 4.17: Mutual Dependencies of Code Selection

given tree grammar. In the following we give a very coarse–grain, hierarchical classification for subsets of $\mathcal{SR}$–MOcoverings :

1. *level–0*: a single covering of $\mathcal{SR}$–MOs is considered.
2. *level–1*: the set of coverings differs in storage resources of definitions and uses.
3. *level–2*: the set of covering differs in data routes between definitions and uses. I.e., additional transfer operations occur between definitions and uses.
4. *level–3*: if the target machines offers machine operations that implement complex patterns (like $(>> (a), b)$), different granularities of coverings are considered. Algebraic transformations are considered, such that the set of operations incorporated in an input tree changes (e.g., $a + a \rightarrow 2 * a$).

The levels can be further partitioned, by considering certain functional units allowed for certain operations, or restricting the length of data routes between definitions and uses. It should always be taken care of what is mend by the term code selection. Selection of machine resources of levels 0 and 1 is generally stated as resource allocation and not code selection. Often, the selection of coverings of level–2 is also not considered as code selection. The notion code selection is generally seen as selecting between coverings of level–3, as this is concerned with selection of different operations.

Approaches

The approaches based on integrating code selection are concerned with phase coupling, therefore described in section 7. In [AM95] irregular architectures with **no parallelism** (i.e.,

pipelining or instruction level parallelism) and single register sets and one main memory are considered (called $[1, \infty]$ model). For this machine model, register allocation is automatically performed by tree pattern matching. If instruction scheduling and spilling are disguarded, the code selection for expression trees is optimal with respect to a sequential cost model and the machine model. The storage resources of operands and the data routes are determined by the matching process. The costs of data movements are taken into account by the costs associated with the chaining rules. However, an unfavourable selection may still result in too many spillings that might compensate the saved costs of the optimal covering. To prevent spilling an instruction scheduling approach is proposed that produces spill free schedules for expression trees if the target architecture fulfills the **RTG Criteria** (see [AM95] for details). There is still some research necessary, for which classes of target machines, extending the $[1, \infty]$ model, still effective code is produced. One continuation of the approach is to extend the algorithm to $[N(M)]$ models, i.e., N classes of registers with M registers available.

4.5 Retargeting: Extracting Code Selector Specifications from HDLs

The automatic generation of code selector specifications from structural models is an important issue. Generally this means to map the structural model to a behavioral model, and consists of the following basic tasks:

- the machine operation patterns must be extracted to generate the tree grammar rules;
- residual control and addressing modes of operands must be determined and have to be embedded into tree grammar rules;
- the alternatives for implementing control flow have to be extracted (conditional and unconditional jumps); this is basically due to analysing the interconnections of the program counter and the controller;
- The encodings (machine instruction strings) that are necessary to initiate the machine operations must be determined (see [LM94]). This also incorporates NOLOAD (NOP) operations, which are necessary for preventing side effects unwanted.

It may happen, that operations of the intermediate representation are not available on the target architecture. Such operations must be converted to machine executable operations. Additionally, types of the operands have to be taken into account. This is very important in the context of register sets with different bit width.
The CBC [FHKM94] transforms a machine description specified in nML into an *iburg* specification. The intermediate representation (CDFG) is transformed in advance, such that operations in the CDFG are expanded into machine executable operations (MEOs) available on the target machine. This set is determined by analysing the machine specification during the retargeting process. In the terminology used in [FHKM94] the machine operations are called **chains** or data path operations (DOs). All machine operation patterns are generated by the nML front–end and represented by so called *match–replace–pairs* that include a term

representation to that a matched machine operation pattern is rewritten. From this an iburg specification is generated.
If we consider extended techniques like term rewriting rules a question is, if it is possible to extract them from structural models. A more general question is, what kind of information can be extracted from a structural model at all? What informations have to be added by the user? These also incorporates questions about how informations have to be represented (formal representation). Remaining questions are concerned with the support of auto–increment/decrement registers.

4.6 Summary

The main fields of interest in the area of code selection are:

- Fast retargeting to new target machines;
- the developement of efficient code selectors and code selector generators;
- gaining more expressiveness of specification techniques.

The preferable technique for code selection is tree pattern matching assuming a tree based intermediate representation. Tree pattern matchers are able to find all possible coverings for a certain expression tree and they also find a minimum cost covering by dynamic programming. The specification for code selectors is based on regular tree grammars. The principle of constructing tree pattern matchers is based on constructing finite tree automata from regular tree grammars.
Supported features of irregular architectures are:

- *heterogeneous register sets*
- *register classes*
- *data routes*

Instruction level parallelism and spill code minimization cannot be exploited appropriately by traditional tree pattern matching techniques. The major problem is an appropriate integration of code selection with the other tasks of code generation.
The preverable specification techniques for retargeting can be classified as a behavioral specification model. Structural models generally allow a more precise and detailed specification and can be transformed to behavioral specifications. Therefore techniques developed for tree pattern matchers are available for code generators based on structural specifications. Remaining questions are concerned with the relations between extended specification techniques and structural models. Integration of specific architectural features into tree pattern matching is another issue for further investigations.

Chapter 5

Register Allocation

The prefered technique for register allocation is *graph coloring*. Thus, this section is mainly concerned with approaches based on graph coloring. The section discusses the supported features of irregular architectures and basic drawbacks. The basic notions like *live range*, *interference*, and *live range splitting* are shortly intodruced (for details see [Bri92]). Problems in the context of irregular architectures with fine–grain parallelism are shown. Some remarks on retargeting register allocators are given. The subsequent sections are structured as follows:

- An introduction to the basic notions and principles of graph coloring is given in the following two section.
- In section 5.3 an overview of the basic classes of graph coloring techniques are described, disgarding approaches concerned with phase coupling.
- Section 5.4 discusses the problems arising in the presence of non–regular architectures. Like in code selection solutions of the arising problems are basically concerned with phase integration, which are described in chapter 7.
- Finaly, section 5.5 is concerned with common aspects of the retargetability of register allocators.

5.1 Introduction

The task of the register allocator is to make effective usage of registers provided by each architecture, which is essential for producing high quality code. Register allocation consists of two components: the **register allocator** determines which values are stored in registers at each program point; the **register assigner** determines the physical register (location) where a value resides, that was allocated to a register.
In conventional code generators, the code selector maps values to virtual registers (level–0 coverings of $\mathcal{R}$–MOs or $\mathcal{SR}$–MOs). Using this strategy, the register allocator must assign physical registers of the target machine to virtual registers. There are the following levels register allocation can take place:

- **local register allocation**: The local register allocation techniques are restricted to expressions or statements of basic blocks. Values are loaded to registers at the beginning

of a basic block and stored to memory at the end of a basic block. *Minimal path costs* is a technique based on finding minimal paths in a DAG. Graph nodes represent the configuration of values residing in registers and edges are labeled with costs corresponding to the number of loads and stores required to change the configuration, acording to the pair. Belady's optimal *page replacement algorithm*, developed for operating systems can also be used to perform optimal local register allocation [Bel66]. *Use counts* developed by Freiburghouse is based on keeping track of the amount that a virtual register will be referenced in the future [Fre74].

- **global register allocation**: Global register allocation methods transcendends basic block level taking into account the control flow structure of a procedure. Techniques are e.g., *Packing algorithms* (see [Ben94]), *probabilistic register allocation* [PF92], a combination of local and global allocation, and *graph coloring* [Bri92], a combined allocation and assignment technique.

5.2 Foundations of Graph Coloring

A **value** is a typed quantity that can reside either in a register, in memory, or, when it can be computed by a sequence of instructions, in program code. The definitions introduced are based on the CFG representation of a program. A variable x is **defined** at a point in a program if a value is assigned to it. A variable is **used** when its value is referenced in an expression (also see section 2.3.1).

Definition 5.2.1 *A variable x is* **live** *at a node n if there exists a path from n_{start} to n containing a definition of x and there is a path from n to a use u of x ($u \neq n$) containing no redefinition of x.*

Definition 5.2.2 *The* **live range** *LR_x of a variable x is the set of all nodes $LR_x = \{n_1, \ldots, n_k\}$ such that x is alive at n_i $(1 \leq i \leq k)$.*

Definition 5.2.3 *Two variables x and y* **interfere** *iff they are simultaneously alive, i.e. the intersection of their live ranges is not empty: $LR_x \cap LR_y = \emptyset$.*

Definition 5.2.4 *The* **interference graph** *is an undirceted graph $IG = (N, E)$, such that N is a set of nodes corresponding to live ranges of variables and $(LR_x, LR_y) \in E$ if x and y interfere.*

Two variables that interfere cannot occupy the same physical register. Two nodes n_i and n_j with $(n_i, n_j) \in E$ are called *neighbors*. The number of neighbors of a certain node n is called its *degree* denoted by n^o. A *propper assignment* is a mapping from live ranges into the available registers of the target architecture, such that no neighbors of the interference graph are mapped to the same register.

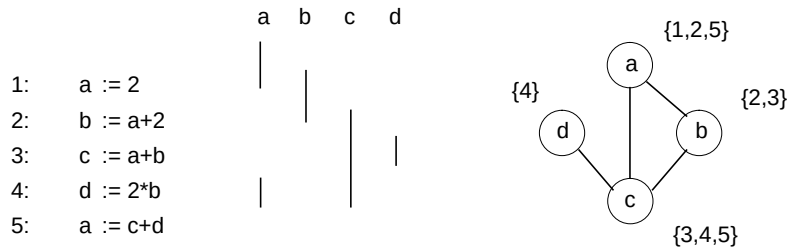


Figure 5.1: Live Ranges and Interference Graph

Example 5.1:

> Figure 5.1 shows an example program and an interference graph with its corresponding live ranges associated. We see that the nodes of variable a and d do not interfere because the intersection of their live ranges is empty. **Remark:** it should become clear at this point, that the definition of interference is based on the ordering of the program statements.

The problem of finding a propper assignment for an interference graph IG is reduced to the problem of finding a k–coloring for IG. I.e. a mapping of nodes to k colors, such that neighbors will always have different colors. If k is choosen to be the number of available machine registers than a k–coloring can be mapped easily to a propper assignment.

Definition 5.2.5 (Graph k–Colorability) *Let $G = (N, E)$ be a graph with a set N of nodes and a set $E \subseteq N \times N$ of edges. Furthermore, let there be a set of k colors. G is k–colorable iff there is a function $f : N \to \{1, \ldots, k\}$ such that $f(u) \neq f(v)$ for $(u, v) \in E$.*

It has been shown, that the problem of finding a k–coloring for fixed $k \geq 3$ is NP–complete [GJ79], therefore a heuristic method is used to search for a coloring. This method can't guarantee to find a k–coloring for a k–colorable graph. The heuristic is based on the following observasion: if a node p exists with n neighbors ($p^o = n$) and $n < k$, then there exists a color different from p's neighbors. We eliminate p from the graph which results in a smaller graph IG'. The problem is therefore reduced to a smaller problem. If all nodes can be eliminated in this way, there exists a k–coloring for the graph.

Example 5.2:

> In figure 5.2 a 3–coloring of the interference graph from 5.1 is illustrated. The set of colors is $\{r, g, y\}$ (red, green, yellow). A 2–coloring does not exist for this graph.

If a k–coloring cannot be found some live ranges are spilled. Let there be a live range LR_x, such that the set of colors assigned to its neighbors is $\{1, \ldots, k\}$. There is no color left for LR_x. There are three approaches to solve this problem:

- x is mapped to memory and all references to x access memory. This method cannot be taken for load–store architectures and is also not the best solution if x is frequently used.

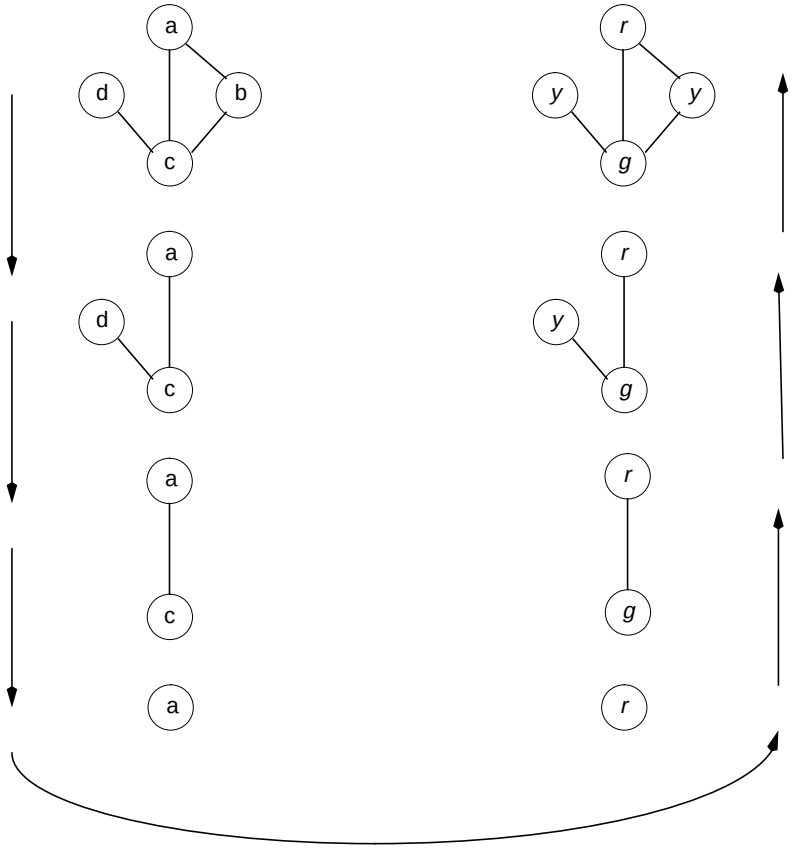


Figure 5.2: A 3–Coloring of a Graph

- The introduction of spill code stores a live range after each definition of a variable and reloads it before every use of the variable.

- The last solution introduces spill code only at certain points of its live range and loads it back at certain points. Hereby some of the nodes in the live range can be eliminated. Generally the live range is splitted into a set of new, disjuntive live ranges. Each new live range constitutes a new node in the interference graph generally with less interferences then the original live range. This thechnique is called **live range splitting**. Chow and Hennessy used this idea [CH84].

Spilling a node (or splitting a live range) results in a new interference graph. Therefore the register allocator must iteratively spill some nodes and color the new resulting graph. In contrast to include spill code for every occurence of a variable, live range splitting may avoid spilling when not necessary. Introducing spill code for every occurence can be seen as the extreme case of live range splitting. Unfortunately live range splitting is very difficult, i.e. determining live ranges to split and picking places to split them. Finding optimal solutions is NP–hard for both problems.

Example 5.3:

In figure 5.3 the splitting of the live ranges of variables a and c from the graph in figure 5.2 is demonstrated. The splitting results in a spill–free 2–coloring of the resulting graph.

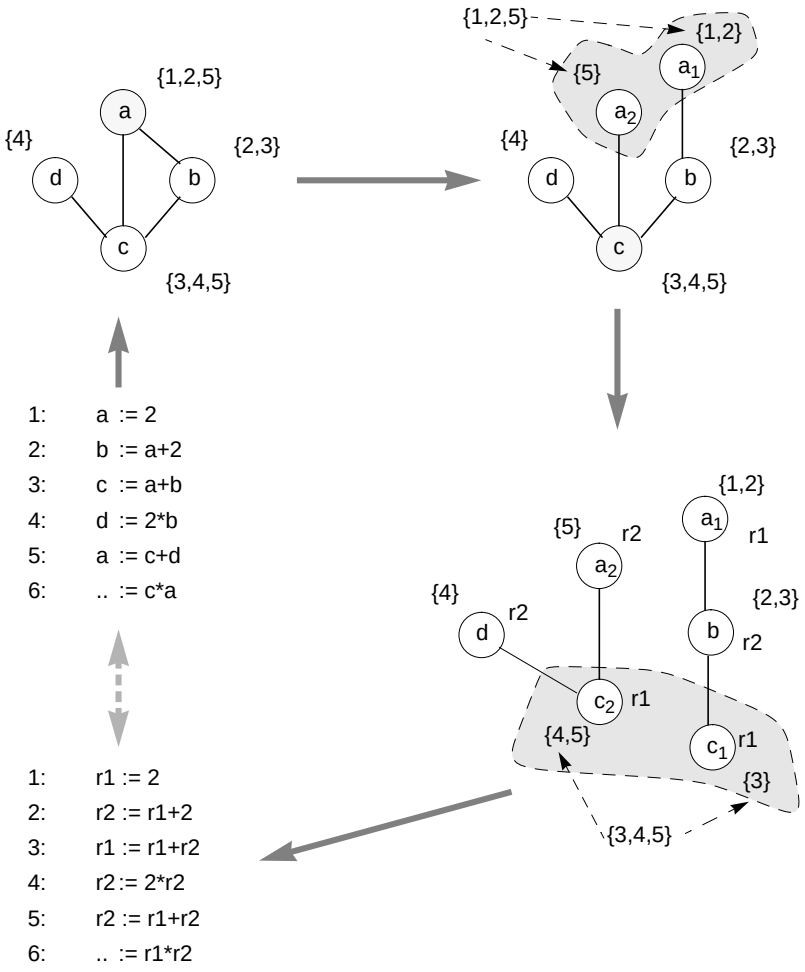


Figure 5.3: 2–Coloring of a Graph by Live Range Splitting

In the former example there are two classes of live range splittings. The first class splitts the live ranges where a use is followed by a definition. Hereby no insertion of spill code is needed, as the value is redefined anyway. The second class splitts between to uses of a variable and may need the insertion of spill code when the assigned register is overwritten. In our example no spill code was necessary, because the value of the variable c stored in register $r1$ was not overwritten because of the availability of register $r2$. There are some approaches using a more coarse grained representation of live ranges based on basic blocks, i.e. the live ranges are not constructed from the nodes of the CFG, but from the nodes of the corresponding basic block graph (e.g. [CH84]). An overview is given in figure 5.4.
An antagonosic task to live range splitting is **coalescing** of live ranges. This is performed for variable copyings like `x := y`. In this case the live ranges of x and y can be coalesceded to

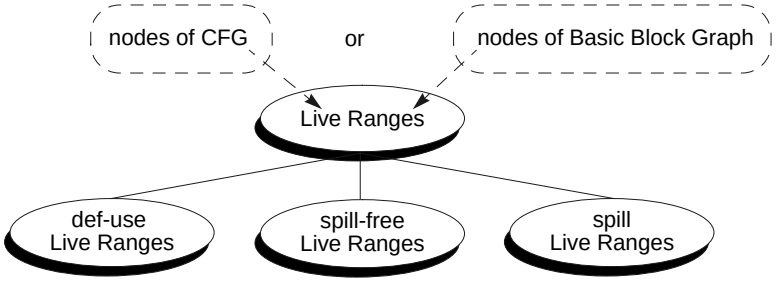


Figure 5.4: 2–Coloring of a Graph by Live Range Splitting

one live range by the union of the two live ranges of x and y. This results in assigning x and y to the same physical registers. A very good overview of the advantages and problems of live range splitting and coalescing is found in [Bri92]. Due to the fact that optimal solutions cannot effectively be computed (while it not can be shown that $P = NP$), researchers are concentrated with finding efficient heuristics to solve the spill problem, i.e. minimizing the amount of spilling.
The subsequent section outlines basic classes of graph coloring techniques, disgarding approaches concerned with phase coupling. In section 5.4 the problems arising in the presence of irregular architectures are discussed, followed by a discussion of retargeting register allocators.

5.3 Graph Coloring Register Allocators

One of the first approaches on memory allocation and graph coloring was published in [Lav62]. A summary of early approaches can be found in [Bri92]. The first implementation of a global register allocator via graph coloring was done by Chaitin et alias (**Yorktown register allocator**) [CAC+81]. Chow and Hennessy (C&H) described a technique based on a combination of local register allocation and graph coloring (**priority–based register allocation**). There are the following criterea for distinguishing the two approaches.

1. *Abstraction levels:*

   (a) Chaitins allocator performes allocation on $\mathcal{A}$–MOs, and C&Hs approach is performed on source code.

   (b) Chaitins defines the granularity of live ranges on $\mathcal{A}$–MOs, in contrast to C&Hs live ranges based on basic blocks.

2. *Allocation model*: Chaitin assumes values in registers. C&H assume, that values reside in memory in advance.

3. *Priority estimations and Coloring:* In Chaitins allocator, priority estimations are used to select a candidate for spilling among the set of *constraint* live ranges. A live range is called constraint if its degree exceeds the number of available registers for the live range. In C&Hs approach, the priority estimation denote the benefits for allocating a

value to a register. Due to their allocation model, they do not spill live ranges in the sence of Chaitins allocator. They consider constraint live ranges and color the one with the highest priority first, thereby reducing the set of colors constituting candidates for the interfering live ranges.

4. *Live range splitting versus spilling:* If there are no register candidates left for a live range, C&H s allocator performs live range splitting. All nodes which are uncolorable are not allocated to registers. Chaitin includes spill code for every definition and use of a variable.

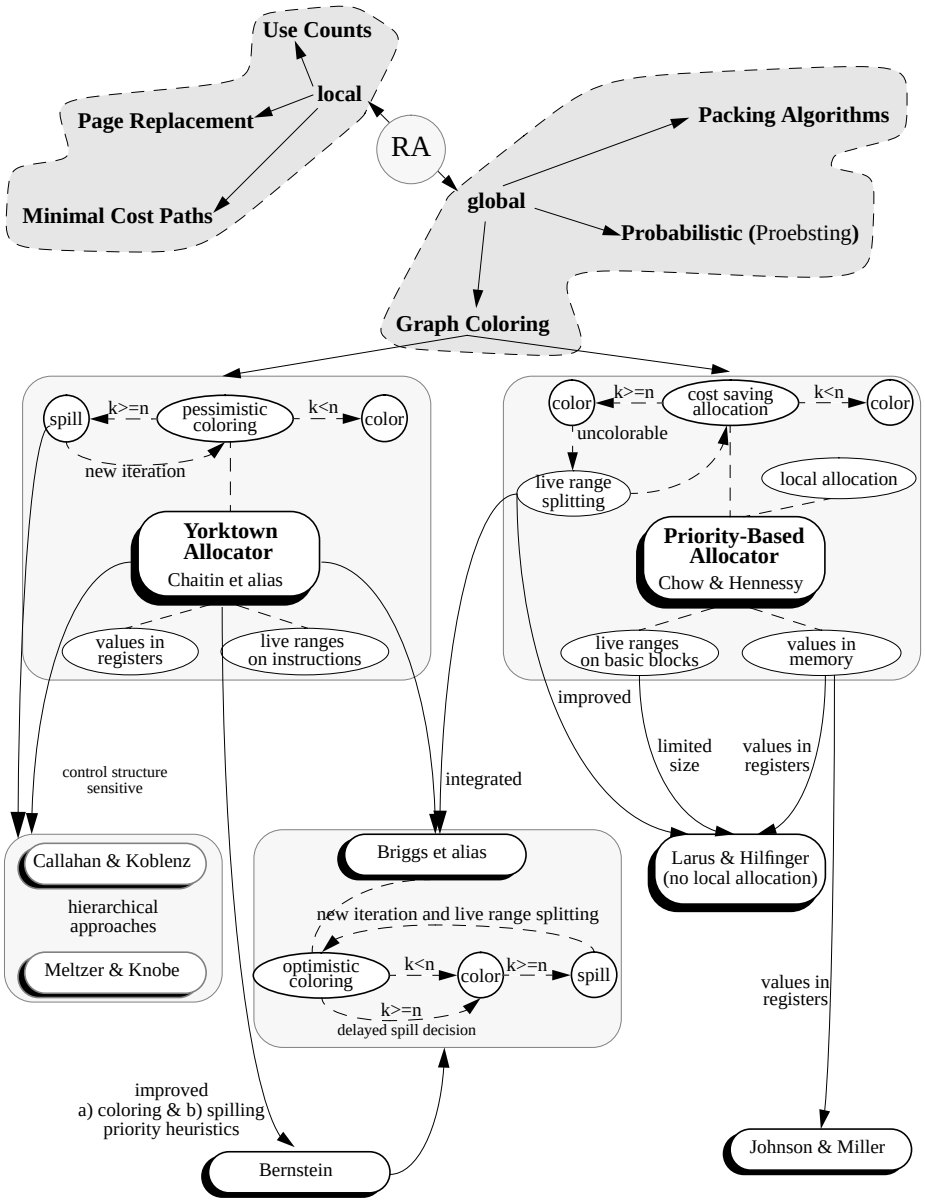


Figure 5.5: Hierarchy of Register Allocation Techniques

There are several approaches improving Chaitins method [GSS89, BGG+89, Bri92], and Chows and Hennessys method [LH86, JM86]. An overview of register allocation techniques

and their dependencies is shown in figure 5.5. The improvements can be characterized as follows:

1. *Reducing spill costs by*

   - *increasing probability of k–colorability.* These approaches have in common to use a modified improved heuristic to color the graph, such that the set of graphs that are colorable but not colored by Chaitin's approach is decreased.
     - improved priority estimations, also with regards to selecting nodes to color [BGG+89];
     - delaying spilling decissions (*optimistic coloring [Bri92, BCT94]).*
   - *improved live range splitting*. The developement of good splitting heuristics is one field of interest in this research area (e.g., [BCT91, CK91]). It is also concerned with integrating live range splitting into Chaitin's approach, instead of inserting spill code for every definition and use.

2. *Consideration of control structure:* this is achieved by an appropriate mapping of the loop structure to priority estimations and/or live range splitting heuristics. There are approaches of hierachical coloring algorithms that take into account the nestings of the control structure by coloring nested regions bottom–up [KM90a, CK91].

3. *Efficiency of register allocators* [GSS89].

Recent approaches take into acount instruction level parallelism. These approaches are concerned with phase coupling and outlined in section 7. Subsequently the basic approaches and improvements to this approaches are outlined.

5.3.1 The Yorktown Register Allocator

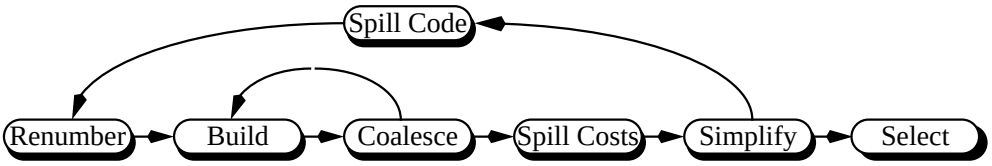


Figure 5.6: Chaitin's Register Allocator

Figure 5.6 shows the tasks of Chaitin register allocator which consists of seven components (also see [BCT94]):

Renumber Chaitins register allocator defines a live range for each new definition of a variable. At a use it unions all the live ranges that reach the use, thus coalescing live ranges where control flow coalesces. The live ranges are statement based and represent def-use chains. All live ranges are uniquely named.

Build constructs the interference graph.

Coalesce attempts to shrink the live ranges. Two live ranges can be combined if the definition of a variable corresponding to the live range is a copy of the other (e.g. `x := y`) and they do not otherwise interfere. By combining two live ranges the corresponding copy instruction can be eliminated.

Spill Costs computes an estimation of the costs that are added if the value is spilled for each live range. This is estimated by computing the additional loads and stores to spill the value, with each machine operation weighted by $c * 10^d$, where c is the transfer machine operation cost on the target machine and d is the statements loop–nesting depth.

Simplify chooses a node with degree $n < k$, removes it from the graph together with its corresponding edges, and places the node on a stack. If there is no node with degree $n < k$ a node to spill is choosen according to the computed spill costs, with regards to minimize spill costs. The node is removed from the graph together with its corresponding edges and is marked to spill. If all nodes are removed from the graph and there are nodes that are marked for spilling, *Spill Code* is initiated, otherwise **Select** (see figure 5.6).

Spill Code is invoked if nodes were marked to spill. Spill code is inserted at the corresponding places in the CFG (or program). Each marked live range is decomposed into it elements resulting in many live ranges were a load is inserted before each use of a variable and a store after every definition. Therefore the structure of live ranges changes and register allocation restarts. The task of simplify together with the computation of the spill costs can be denoted as the allocation task.

Select is performed when no live ranges are marked for spilling. It colors the nodes on the stack in the reversed order Simplify removed them from the graph. A node is popped from the stack, inserted into its position in the graph and gets a color different to that of its neighbors that were sofar inserted by Select.

The task of simplify is reflected by the left reduction sequence of figure 5.2 and select is the reconstruction shown on the right side of 5.2. Simplify only pushes a node to the stack when it can prove that a node can be colored. If simplify does not find a node to color, it selects a node to spill. The metric for picking candidates for spilling is a very important task. As the interference graph abstracts completely from the loop structure of the CFG the nestings of values are mapped to the spill costs. The node with the smallest ratio of spill costs is choosen. Thus, the loop-structure is taken into account by the selection of the next spill candidate. The aim of selecting a node for spilling is, to spill a value that is used very infrequently, such that spill costs can be minimized. Several heuristics for making good choices were developed and are summarized at the end of the section.

5.3.2 Piority–Based Coloring

In the approach of [CH84, CH90] graph coloring is performed after a local allocation phase. The task of local allocation is to estimate the saving costs for a value that is allocated to a register. The analysis is performed on basic blocks. Chow and Hennessy assume that all variables are located to memory a priory. If a value is assigned to a register, first uses

of values not preceeded by a definition of the value must load the value from memory to registers. If the value is modified in the basic block it must be stored back to memory if it is live at the end of the basic block. The saving costs are computed by regarding the amount of execution time that is saved by accessing a value in memory in contrast to accessing it in a register, for each basic block:

1. $maxsave = (loadsave * u) + (storesave * d)$;
2. $minsave = (loadsave * u) + (storesave * d) - (movecost * n)$;

such that

- $loadsave$ is the execution time saved when a value is assigned to a register compared with the corresponding memory reference. u is the number of uses in the basic block considered.
- $storesave$ is the execution time saved when a value is assigned to a register compared with the corresponding storage to memory for a definition of a variable. d is the number of definitions in the basic block considered.
- $movecost$ is the execution time for loading and storing a value at the beginning and at the end of a basic block. Thereby n is either $0, 1$ or 2, depending on if the initial load or the storage at the end of the basic block is necessary.

The global saving costs for a value are computed as the sum of the saving costs of all basic blocks contained in the values live range. In the coloring algorithm live ranges with negative saving costs are marked for spilling, even if they could be allocated to a register indicating that allocation to a register does not result in any saving for this live range and should be avoided for the live range. Chow and Hennessy distinguish between *constrained* and *unconstrained* live ranges. Unconstrained live ranges denote nodes in the interference graph with degree $n < k$.
The first tasks of the register allocator are to compute all live ranges, to construct the interference graph and to seperate all unconstrained nodes from the graph. The unconstrained live ranges are not colored until the very end because it is certain, that unused colors can be found for them. In contrast to Chaitins register allocator, the priority–based register allocator tries to color constrained live ranges, i.e live ranges that are marked for spilling in Chaitins register allocator. The allocator performes the following steps:

- For each constrained live range the complete saving costs are determined.
- All live ranges with negative saving costs are deleted from the interference graph and are marked as noncandidates.
- All live ranges that are uncolorable are removed and marked. An uncolorable live range consists only of program points where all available registers are occupied by other live ranges.

- The live range with the highest amount of saving costs to which a color can be assigned is determined. A color is selected, not being in the *forbidden set* of the live range, i.e. that there is no neighbor that received this color. The forbidden sets of all neighbors of the selected live range are updated. It is checked if any neighbor must be splittet, what is the fact if the set of available registers of the neighbors live range is equal to its forbidden set. For all new live ranges (resulting from splitting) compute there saving costs.

- Continue with this procedure until all constrained live ranges are colored or there are only uncolorable live ranges left.

Splitting is performed by seperating out a component of the original live range that is as large as possible. This has the effect of avoiding the creation of too small live ranges.

5.3.3 Optimistic Coloring

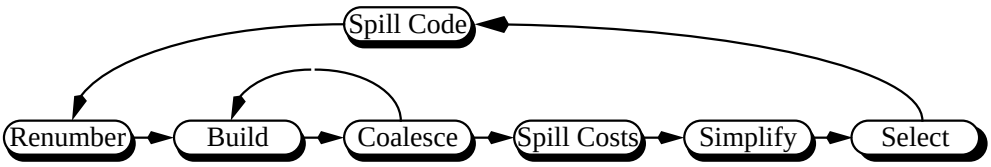


Figure 5.7: Optimistic Coloring (Briggs Allocator)

A basic drawback of Chaitin's allocator is that it pessimistically assumes that all of the neighbors of a certain live range will get different colors. If this situation occurs the live range cannot be colored, but this must not necessarily happen. This can be illustrated, by observing a simple example.

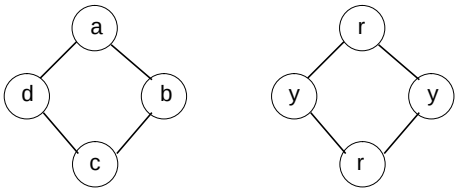


Figure 5.8: Diamond Graph

Example 5.4:

> Figure 5.8 shows the diamond graph which is 2–colorable as seen on the right side. If Chaitin's approach is used, it will select one of the four nodes for spilling, because there is no node with degree $n < 2$. Therefore the graph will not be 2–colored. It is said, that the register allocator **overspills**.

The supposition that a live range LR_x will get a color is approximated by *has LR_x degree less than k* in Chaitin's register allocator. This is a sufficient but not necessary condition. In the example we have seen that a live range may have k neighbors with less than k colors occupied.

The problem is that spilling decissions are made to early in the coloring algorithm. A solution to overcome this drawback is the approach from Briggs, that combines a coloring heuristic from [MB83] with Chaitins mechanism for cost–guided spill selection. Two modifications in Chaitins register allocator are necessary [Bri92]:

1. *Simplify:* All nodes with degree less than k are removed in arbitrary order and pushed to a stack. If there are remaining nodes with degree greater (or equal) then k a spill candidate is selected. The node is removed from the graph but is not marked for spilling. The node is pushed to the stack in spite of its degree and it is optimistically assumed that that a color will be available for the node. Thus nodes are removed in the same order as in Chaitin's register allocator, but spill nodes are incorporated in the stack.

2. *Select:* The detection of necessary spilling is delayed until the Select task. Select may discover, that no color is available for the actual live range popped from the stack. The live range is left uncolored and Select continues. Any uncolored live range must be a node, that Chaitins register allocator would also spill.

   If any live ranges remain uncolored spill code is inserted and register allocation is restarted with the resulting interference graph.

In figure 5.7 the structer of Brigg's register allocator is shown. The decision of spilling occurs in the select task. Defering the spill decisions eliminates unproductive spillings and provides a stronger coloring heuristic, thus increasing the set of k–colored graphs. In figure 5.9 the coloring of the diamond graph using Brigg's approach is demonstrated.

5.3.4 Hierachical Coloring

Callahan and Koblenz [CK91] approach is also an extension of Chaitin's work. It decomposes the CFG into a tree of *tiles* reflecting the control structure of a program. Tiles are visited bottom–up and a local interference graph is created and colored with pseudo registers for each tile. The interference graph of a tile is passed to it's parent tile and is incorporated into the parent tile's interference graph. In a top down pass all pseudo registers are assigned to physical registers and spill code is inserted where it is required. This must not necessarily be where decision to spill were made by graph coloring. The allocation of registers is sensitive to local register usage while maintaining the global aspects of register allocation. In each tile registers are allocated by using standard graph coloring.

5.3.5 Other Approaches

There are several approaches for improving Chaitins method (e.g. [GSS89, BGG$^+$89] and Chow and Hennessys priority–based graph coloring method (e.g. [LH86, JM86]). Proebsting and Fisher propose an approach not based on graph coloring [PF92] called **probabilistic register allocation**. Probabilistics are used as a heuristic measure to drive the allocator. No register assignment is performed. Probabilistic register allocation may be used for utilizing good spilling decisions during graph coloring.

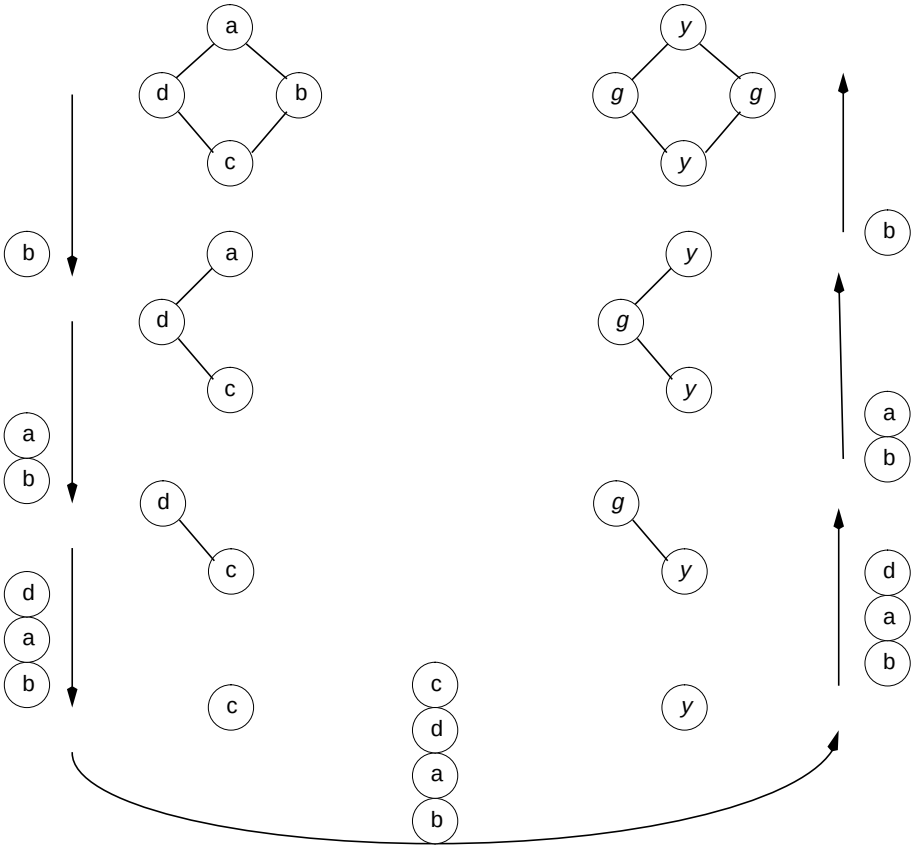


Figure 5.9: Coloring the Diamond Graph

5.4 Support of Architectural Features

Distributed register sets constitute no problem for graph coloring if the storage resources of values are known. Live ranges of values that reside in different register sets will not interfere. *Register classes* can also be incorporated in graph coloring, by extending the notion of interference. The set of storage resources where a value may reside in, is assigned to every node in the interference graph. The notion of interference is extended in the sense, that live ranges only interfere if the intersection of the associated sets is not empty [Bri92].

Common Suppositions

To outline the problems occuring in the context of irregular architectures, we first consider the original goals and suppositions made for performing conventional register allocation:

1. The *original goal* was the minimization of transfer costs of data. Reduction of spill costs should seen as a special form of this task (e.g., Chow and Hennessy do not consider spilling, due to their allocation model).

2. There are the following common *suppositions*:

- Allocating a value to a register is based on certain priority estimations; they rely on static factors (with regards to a certain program) like the number of uses of a value, corresponding nesting depth, and the costs for loading/storing a value from/to memory.
- Candidates for spilling are determined by priority estimations. Generally, spilling to memory is considered and direct memory access is assumed.
- Live ranges are based on a fixed execution ordering of the statements of a program. Conflicts occur if too many live ranges interfere at certain points of the program.

New Goals

One motivation of providing distributed storage resources is to increase multiple storage resource access. This can only be supported effectively if accessed values are properly distributed to storage resources. Therefore, a new goal of register allocation is an adequate distribution of values to register sets. The original goal is also concerned with new tasks, e.g., *finding adequate data routes for minimizing tranfer costs*. A distribution to storage resources should also utilize minimizing spill costs. The support for auto–increment/decrement register is an additionally new task for register allocation. Values used for addressing of subsequent memory cells are candidates to be allocated to such special registers.

Mutual Dependencies

In figure 5.10 the dependencies of goals and tasks of code generation are shown. The new, problematic situations that occur can be outlined as follows:

1. Allocation to register sets cannot be disconnected from code selection. The storage resources, that values are allocated to, depend on the machine operation patterns related to the operations. Mutual dependence between the storage resources and operations is given. If code selection results in a covering of machine operation patterns, storage resources are fixed in advance. Thus, there is no choice for distributing values. A preallocation is quite difficult, because the mutual dependencies have to be taken into account → **mutual dependence of register allocation and code selection**.

2. Spilling must not necessarily be performed to memory. It can be impossible to spill a value directly to memory → **complex spill routes and restricted spilling**.

3. Spilling across data routes can involve the spilling of other values. Therefore, the spill costs of a value depend on the values concurrently alive and their locations → **dynamic spill cost estimations**.

4. Concepts based on fixed execution ordering are not adequate for *utilizing parallelism* provided by the machine:

   (a) Informations for minimizing spill costs are based on strict ordering of statements and often allow the generation of **unnecessary false dependencies**. False dependencies restrict the instruction scheduler in exploiting available parallelism.

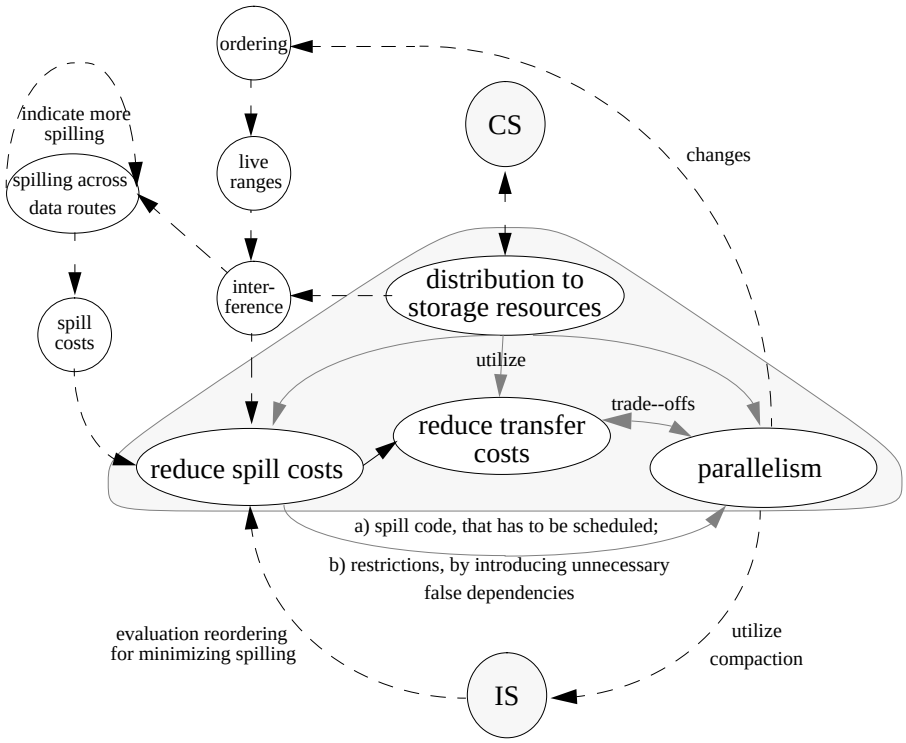


Figure 5.10: Mutual Dependencies of Register Allocation

(b) If register allocation is concerned with taking parallelism into account it has to make some estimations about the available parallelism. An over–estimation of available parallelism leads to a high amount of interference, that can lead to a high amount of spill code. This may result in worser code, than the uncompacted program. For making exact estimations about the interference, instruction scheduling has to be performed before register allocation. By this more exact estimation about available parallelism are made, but spill code is not considered. It is determined in the (so called) *late register allocation* phase, thus leading to a rescheduling. Again, the additional spill code can lead to worser code → **cyclic dependencies: estimations are based on factors that change, when estimation based transformations are applied**.

5. In some cases it is unecessary to avoid spilling or more expensive data transfers, because they can be performed concurrently with other machine operations. Therefore they can be integrated into the generated schedule without producing extra costs → **trade–off between exploiting parallelism and transfer cost reduction**.

6. A very important issue is the distribution of values to storage resources for exploiting parallel execution. This can only be performed appropriately, when register allocation is moved into instruction scheduling. As this will indicate a selection of certain machine operation patterns, code selection is also embedded. Moving code selection into scheduling makes it difficult to perform global register allocation, because certain storage resources are not determined, when a certain machine operation is scheduled

→ **uncertainty of storage resources for certain values**. Therefore, incremental techniques are required.

The new task of distributing values to storage resources basically constitutes a phase coupling problem. Section 7 is concerned with this subject. Aditionally, an adequate technique taking parallelism into account has to make good trade–offs between exploiting parallelism and preventing spill code. In the following, recent approaches trying to solve some of the problems and not concerned with phase coupling are described.

Approaches

In the following, some approaches are outlined trying to solve some of the mentioned problems. More approaches concerned with phase coupling are given in section 7. It is mentioned in advance, that the basic remaining problems are the distribution to storage resources and adequate coupling with code selection. The support of auto–increment/decrement registers seems to be also a very scarce investigated area, with regards to retargeting.

Storage Resource Abstraction

In [Hei93] storage resources are combined to new abstract storage resources by introducing corresponding new nonterminals and machine operation patterns. It must be ensured, that each storage resource belonging to an abstract storage resource is accessible to the set of functional units, with respect to the machine operation patterns where the corresponding nonterminals occur in. A storage resource of the set of storage resourcesdenoted by the abstract storage resource, is either

- directly accessible, or
- reachable from all directly accessable storage resources contained in the set.

By this, traditional tree pattern matching can be performed producing a single covering of $\mathcal{A}$–MOs. This covering leaves some freedom to the register allocator in distributing values to those storage resources, denoted by corresponding abstract storage resources.

Exploiting Distributed Storage Resources

An approach trying to distribute values for exploiting parallelism proposed in [ADK+95] is based on extended versions of graph coloring but is restricted to distributed memories. Therefore it is not confronted with restricted data paths and register connectivities. The goal is to maximize the number of parallel move operations between registers and memory. Late register allocation (i.e., register allocation after instruction scheduling (see chapter 7)) is performed which assigns symbolic registers to physical registers and variables to distributed memory banks.
In [CDN94] an approach is proposed called hypergraph coloring and also performs late register allocation. The task of register allocation is delayed until after scheduling, because informations about the concurrently alive values are necessary. They assume idealized VLIW like architectures, such that every functional unit is connected to every register set. Therefore no restrictions of connectivities and non of the problems of interest do occur.

Register Classifications

Paulin [PLMS95] adopts a method whereby a number of overlapping register classes for a given target architecture can be specified. The code selection task is concerned with handling machine operation patterns (Paulin uses the term micro–instruction patterns) with register classes. In contrast to tree pattern matching described in section 4 the pattern do not contain nonterminals for denoting different storage resources or register classes. Instead there are specific write and read terminals annotated with register classes specifying a certain set of registers. The register candidates for a value are determined by the intersection of the register classes annotated to connected definitions and uses. Empty intersection indicates that a register move is necesssary. These register moves are determined after matching. It is quite clear how reachability is checked to yield a legal covering of the input tree. The approach is based on an improved left–edge algorithm. Even if Paulin states to overcome the drawbacks of graph coloring, it is not quite clear which drawbacks are really solved. The intersection sets are determined by the matching process. Therefore the possible locations for values are determined. Live ranges are also known before register allocation. Register classes can be handled by graph coloring aswell. Therefore, the aspects considered by Paulin can be coped by graph coloring.

5.5 Retargeting

If conventional register allocation techniques are considered, retargeting seems to be of no problem at a first glance. However, this is generally related to consider retargetability as using the same algorithm for a wide range of architectures, without the need of modifying it. This aspect of retargetability does not take into account, that certain architectures are utilized by a specific technique, while other architectures are not. A good retargetable code generator should also select a register allocator suitable for the target architecture, regarding and exploiting the specific features of a machine. There are three factors that should be considered:

1. *Heuristics*: Register allocation techniques are based on heuristics for selecting candidates for coloring or spilling. There are various heuristics developed in recent years. An adequate heuristic has to be determined during retargeting.

2. *Amount of registers*: An architecture may contain only very few, few, or many registers. If only very few register are available, global register allocation seems to be unnecessery or even undesirable. Depending on the amount of registers, either local techniques, trade–offs between local and global techniques, or global techniques have to be selected.

3. *Phase coupling:* With regards to other code generation tasks, specific features indicate a certain ordering and/or integration of these tasks. This must also be seen in the context of other well known optimization techniques (e.g., constant folding, code motion [ASU86]), which are not considered in this report. The selection of a good register allocation technique has to take into account the mutual dependencies of selected technique with other tasks, i.e., the order of phases and the degree of integration.

E.g., a register allocator that tries to reduce register usage will drastically constrain instruction scheduling for instruction level parallelism.

A detailed analysis of retargetable register allocation in the context of other optimizations and phase ordering is given in [Ben94]. Investigations in finding criterea and analysis techniques for classifying architectures are necessary, that enable the selection of adequate register allocation techniques and corresponding interaction with other optimization tasks.

5.6 Summary

Graph coloring is the most popular subject of research. Graph coloring is able to handle distributed register sets, if the locations of values are known. A more important task is an appropriate distribution of values to registers. In the context of restricted interconnections of storage resources and functional units , this becomes a very difficult task, due to the strong mutual dependence to code selection and instruction scheduling. Therefore, a complete integration of code selection, register allocation and instruction scheduling is an important subject for further investigations (so far, there are few approaches, shown in chapter 7). Retargeting of register allocation should be concerned with the selection of adequate techniques and corresponding interaction with other optimization tasks.

Chapter 6

Instruction Scheduling

In this chapter the basic classes of instruction scheduling developed in recent years are described. The foundations of compaction techniques for microcoded architectures are described. These techniques constitute a basis for many subsequent instruction scheduling techniques for instruction level parallelism. The basic drawbacks are in the context of irregular architectures are discussed. Some new aspects of retargeting schedulers are outlined. We will discuss the following topics:

- Section 6.2 will introduce the formal foundations of local compaction and the basic local compaction methods.
- In section 6.3 the classes of global scheduling techniques are described and compared. All these techniques are concerned with fine–grain parallelism. However, no specialized techniques concerned with code generation for RISC like architectures are included.
- Section 6.4 emphasizes the basic problems arising when irregular architectures have to be taken into account.
- Remarks on retargeting schedulers are given in section 6.5.

6.1 Introduction

Instruction scheduling is the task of reordering the machine instructions generated from a source program, with the aim of getting a machine program with less execution time and less code space. In the absence of parallelism the reordering of machine instructions utilized the task of register allocation for minimizing register pressure. In recent years there is increasing interest in architectures with instruction level parallelism. The main task of instruction scheduling is to find a mapping of machine operations to instruction cycles that effectively exploits the available parallelism of the target machine, while maintaining the semantics of the original program. Compaction is a subtask of instruction scheduling. The aim of compaction is to combine machine operations into machine instructions , such that parallelism is effectively supporting program execution. If machine resources were not strictly bound by previous tasks, compaction is concerned with the mapping of operations

to machine resources. Early works were developed in the context of microcoded machines, termed compaction techniques. A summary of this work's can be found in [DLSM81].

A few classes of instruction scheduling techniques have evolved in the last years. There are four basic classes of scheduling techniques: **Local list scheduling** techniques are performed on basic block level, i.e., the machine instructions are restricted to contain only machine operations from the same basic block. **Trace scheduling** [Fis81] is an extended list scheduling technique not restricted to basic block boundaries and employs branching probabilities to select the most likely execution path of a program. The selected path (trace) is the regarded as if it was a basic block. To preserve program semantics when moving machine operations across conditional jumps, *compensation code* has to be inserted. **Percolation scheduling** also is a global scheduling technique based on four semantic preserving program tranformation rules performed on the program flow graph (similar to the control flow graph) [Nic85]. Percolation scheduling reduces the generation of superfluous compensation code; code explotion is a major draw back of trace scheduling. **Region scheduling** is a technique based on the program dependency graph and allows movements of machine instructions over wider program regions than percolation scheduling [GS90]. Region scheduling redistributes available parallelism, such that machine resources are fully utilized. This is also achieved by applying semantic preserving transformation rules on the program dependence graph. Region scheduling can also be used for coarse grain scheduling on source code level.

6.2 Local Compaction

As introduced in section 3, a machine instruction is composed of machine operations . If only one or very few machine operation can be performed in parallel, the target machine is said to be **vertical** (e.g., RISC). If many machine operations can be combined in one machine instruction , the machine is said to be **horizontal**. In the context of vertical and horizontal machines instruction scheduling has to solve two different tasks:

1. If the target machine is of vertical nature, it still can offer some fine grain parallelism, as given by the principle of pipelining in RISC machines. The major tasks of the scheduler is to minimize register pressure for avoiding spill code, and keeping each stage of the pipeline busy.

2. In horizontal machines an aim is to parallelize as many machine operations as possible, with regards to effective program execution. Avoiding spill code is not a basic aim of compaction. In some situations the generation of spill code can lead to more efficient programs if spilling can be done in parallel with other operations.

There are no uniquely definitions that distinguish compaction from instruction scheduling. In [Gas89] formal definitions of the local compaction problem is given as follows:

Definition 6.2.1 (Local Compaction Problem) *Given:*

1. *a machine $\mathcal{M}$ and a set of m machine resources $\mathcal{R}$;*
2. *a resource configuration vector $\vec{\mathcal{R}}_{\mathcal{M}} \in N^m$, where the kth entry ($1 \leq k \leq m$) of $\vec{\mathcal{R}}_{\mathcal{M}}$ gives the number of units of resource $r_k \in \mathcal{R}$ available in the machine configuration;*
3. *a set of operations $\mathcal{O} = \{op_1, \ldots, op_l\}$*
4. *a duration function $d : \mathcal{O} \rightarrow N$ where $d(op_j)$ denotes the number of machine cycles op_j takes to execute;*
5. *a resource usage function $\vec{\mathcal{R}}_{\mathcal{O}} : \mathcal{O} \times N_0 \rightarrow N_0^m$. $\vec{\mathcal{R}}_{\mathcal{O}}(op_j, x)(k)$ gives the number of units of resource r_k needed in the xth time step of operation op_j;*
6. *a data dependence graph DDG imposing a partial ordering on $\mathcal{O}$;*
7. *a delay function $\delta : E \rightarrow N_0$ defined on the edges of the DDG. $e = (op_i, op_j)$ and $e \in E$; $\delta(e)$ is the delay that has to be respected before scheduling op_j as soon as op_i has been scheduled.*

The **local compaction problem** *consists in finding a schedule $\sigma : \mathcal{O} \rightarrow N$ such that:*

1. minimality: *σ is of minimal length, and*
2. dependence constraints: $\forall e = (op_i, op_j) \in E : \sigma(op_j) - \sigma(op_i) \geq \delta(e)$, *and*
3. resource constraints:
   $\forall t(0 \leq t \leq length(\sigma)) : (\sum_{j=1}^{l} \vec{\mathcal{R}}_{\mathcal{O}}(op_j, t - \sigma(op_j))) < \vec{\mathcal{R}}_{\mathcal{M}}$

Informally, the problem that compaction tries to solve, is to place machine operations into as few machine instructions as possible, constituting the final schedule. Legal schedules are determined using the data dependencies reflected by the data dependency graph (see section 2.3.4 page 2.3.4). The definition considers multicycle machine operations . Conflict analysis determines wether or not a machine operation can be placed within a machine instruction, without violating data dependencies and resource constraints of the target architecture. Encoding conflicts are not determined in this defintion of the compaction problem. Finding an optimal solution for the compaction problem has been shown to be NP–complete by [DeW76].
The next candidate which is selected for being scheduled is taken from the **data ready set**. These set includes machine operations whose with no predecessors in the data dependence graph, or whose predecessors already have been scheduled. In the following we outline some of the decisions a scheduler has to make. These decisions are made in both local and global compaction techniques.

- *choosing an appropriate cycle:* using list scheduling techniques the scheduler has not much freedom in selecting other cycles than the actual cycle. Linear analysis always chooses the earliest cycle for placing a microoperation. Instead of placing a new microoperation at the rise limit, one could think of placing it at the end of the list.

- *choosing a machine operation from the data ready set:* there are several criterea for selecting the next machine operation from the data ready set, e.g.:

  – prefer machine operation with the maximum path in the DDG;

  – select machine operations from critical path first;

  – timing constraints;

  A summary of heuristics for selecting the next microoperation to schedule is given in [Bea91].

It is advantageous to delay the binding of certain machine resources until compaction. In this case, the candidates in the data ready set are $\mathcal{A}$–MOs. The following additional decisions are necessary:

- *choosing a functional unit:* this requires that code selection has not selected a specific machine operation for implementing a certain operation in the intermediate representation. If architectures are used offering more than one functional unit (e.g. VLIW architectures or some DSPs) it must be determined which functional unit is occupied. If the set of functional units is identical and each functional unit has access to all storage resources this is no big problem. Section 6.4 is concerned with the subject if these idealized assumptions are not present. If there is more than one possible machine operation that can be placed within a microinstruction it can be advantageous to delay the decision of which machine operation to choose as long a possible.

- *choosing result destinations:* i.e. selecting a storage resource and a location where the result of a machine operation is stored in. This again requires that code selection has not selected a specific machine operation. If register allocation was performed before instruction scheduling or a functional unit has already been selected, there is generally less freedom for the instruction scheduler (see section 6.4).

Early compaction techniques where developed for microcoded machines. Many principles developed for compaction techniques can be found in recent scheduling techniques. In [LDSM80] the basic compaction techniques are described, e.g., *linear analysis* [DT76], *critical path* [RT74], *branch and bound* [YST74], and *list scheduling*. These techniques are described do not consider timing constraints and assume single cycle microoperations:

Linear Analysis [DT76]: This approach starts with an empty list of microinstructions. It attempts to place microoperations into existing microinstructions in the list in the order they appear in the basic block. If there is no microinstruction the microoperation considered can be placed in without violating the resource constraints of the target machine; a new micronstruction is inserted into the list (completely don't care) and the microoperation is placed into it. The scope of microinstructions where a microoperation

can be placed reaches from the last microinstruction in the list to that microinstruction the microoperation is data dependent on. This location is denoted as the **rise limit**. Thus the rise limit denotes the earliest microinstruction the microoperation can be placed in, in spite of any resource constraints. Now a microinstruction is searched where the microoperation can be placed in, searching from the rice limit back to the end of the list. If no microinstruction can be found that accepts the microoperation a new and empty microinstruction is placed at the rice limit. By this microoperations are placed as early as possible. If no rice limit is found and no microinstruction accepts the microoperation, the new microinstruction is inserted in front of the actual first microinstruction in the list.

Critical Path [RT74] : The minimal length of the list of microinstructions is the depth of the DDG, i.e. its critical path. An *early partition* of microoperations is constructed, placing all microoperations that occur at the same level in the DDG into the same microinstruction, in spite of any resource conflicts. An analogous *late partition* is constructed. All microoperations having the same position in the list of the early and in the list of the late partitioning constitute the critical path. From these two partitions a *critical path partition* is constructed (for details consult [LDSM80]).

Branch and Bound [YST74]: A tree is constructed which nodes represent microinstructions. A path from the root to a certain leaf corresponds to one possible ordering of microinstructions. The tree branches in that cases when there is more than one possible microinstruction that can be placed at the point in the tree. If the algorithm finds a path with the length of the critical path it has found an optimal answer. Generally heuristics are applied for not constructing a complete tree (that would take exponential time).

List Scheduling This is a special case of branch and bound, where only one branch is constructed (followed) at every node. List scheduling starts with an empty list of microinstructions. Microoperations are placed into the last microinstruction of the list if they are

1. *data ready*, and
2. they have the *highest priority* (computed by some weighting heuristics) among the set of data ready machine operations, and
3. they can be placed into the last microinstruction.

If no microoperation from the data ready set can be placed a new microinstruction is placed at the end of the list. The weighting heuristic has great impact upon the final schedule.

The compaction methods rely on heuristics to reduce the search space of possible schedules with the aim of pruning solutions representing no good results. List scheduling is the most popular technique used in local instruction scheduling techniques, because there exists a known bound on the time it takes to execute. It has complexity $O(n^2)$ [LDSM80, Gas89]. List scheduling produces good results in the presence of adequate heuristics and is easy to implement. A detailed introduction to compaction methods, list scheduling, and a summary

of heuristics can be found in [Bea91]. Instruction scheduling techniques considering timing constraints between machine operations are summarized in chapter 8.

6.3 Global Instruction Scheduling

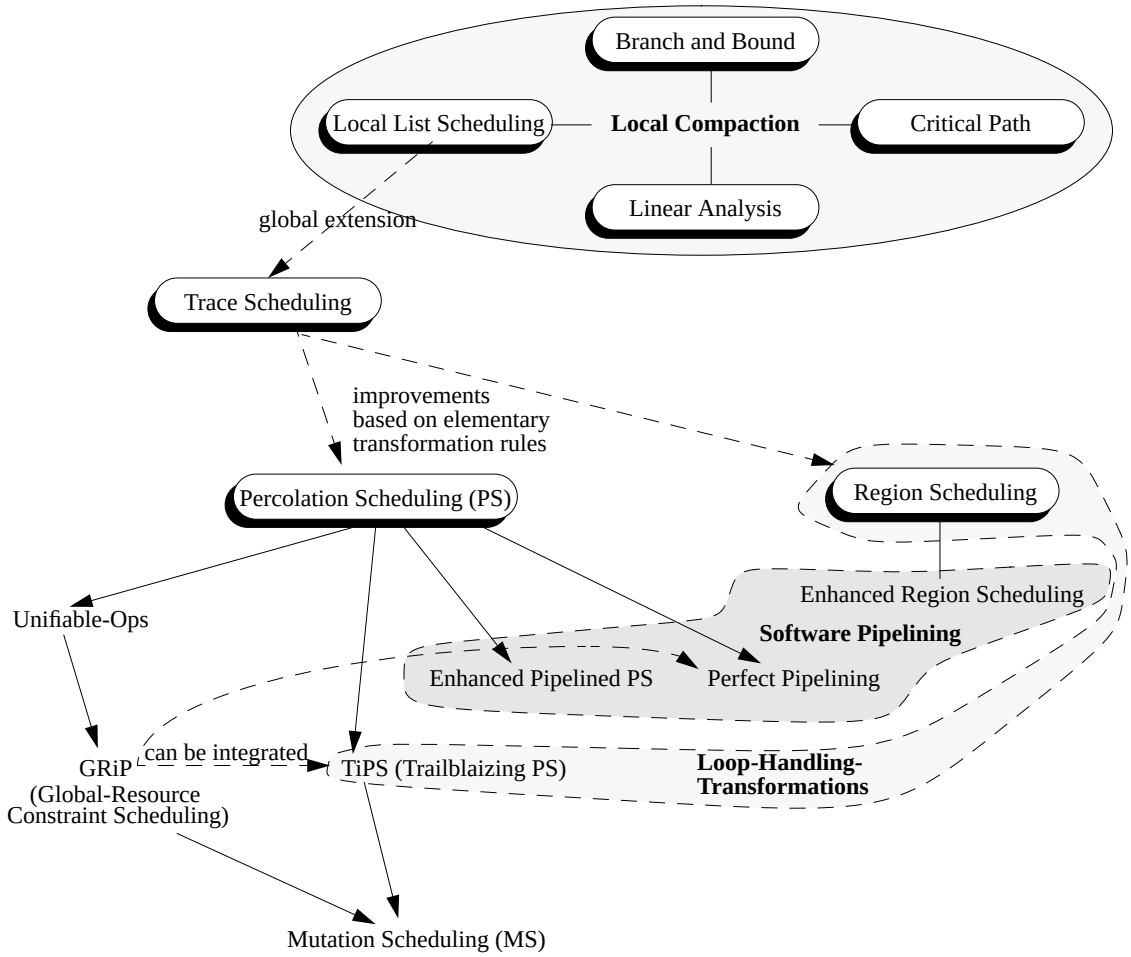


Figure 6.1: Hierarchy of Scheduling Techniques

The techniques of the last section are local scheduling techniques operating on basic block level. A disadvantage is that basic blocks generally offer a more limited degree of instruction level parallelism. Therefore architectures that offer a high amount of parallelism are not supported appropriatly. Recent research has succeeded in overcoming the basic block boundaries. In figure 6.1 an overview of the basic scheduling techniques and their improvements is shown, including the local compaction techniques. These techniques are discussed in the subsequent sbsections. The basic issues of investigations in scheduling techniques and their superimposed and enhanced approaches are:

- improvements in effectively support of instruction level parallelism is a major goal of all approaches;

- taking resource constraints into account is becoming of much interest; e.g., percolation scheduling assumes unbounded resources; many efforts are made to integrate the consideration of a restricted number of machine resources [EN89b, ME92, NN92];
- considerations of over–, and under–utilized regions of a program with respect to the underlying target architectures are made [GS90, BGS95];
- integration of scheduling accross loops.
- Adapting global scheduling techniques for advanced RISC architectures (*remark:* this topic was out of the scope of this report and is therefore not involved, but should not be ignored in further research).

Other issues of improvements are avoiding the generation of superfluous compensation code and integrating transformations rules that allow the movement of operations accross large program regions. Upto region scheduling, the basic scheduling techniques can only handle acyclic code. In region scheduling, loops are compacted by unrolling its body. This approach can be very inefficient with regards to code space. Therefore another class of scheduling techniques based on **software pipelining** has been developed. Hereby a new iteration of the loop is started before the preceeding iteration has completed. Extended versions based on percolation scheduling and region scheduling where developed to incorporate software pipelining (e.g., **perfect pipelining** [AN88b], enhanced pipelining percolation scheduling [EN89a], and enhanced region scheduling [AJLS92]).

6.3.1 Trace Scheduling

Trace scheduling was originally developed by Fisher [Fis81] as a technique for global microcode compaction. Trace scheduling uses a programs basic block graph (see section 2.3.2). Nodes of the graph are the basic blocks. Edges represent the possible control flow. The scheduler partitions the basic block graph into an ordered set of non–overlapping loop–free pathes called traces. The first trace is the most frequently executed trace, the second one is the trace with the next hiqhest frequency, and so on. The set of traces is exhaustive, i.e. the basic block graph is covered completely by the set of traces. Execution frequencies are either extracted from profiling information or estimated from loop–nesting and branch prediction.
The scheduler repeatedly picks traces consisting of uncompacted basic blocks. It now treats the trace as a single basic block and performs list scheduling. It is based on the data dependence graph to determine the legal reorderings of machine instructions. The scheduler determines the data ready set, which contains all machine operations whose predecessors in the DDG have already been scheduled. A machine operation of the data ready set is selected for scheduling and is assigned to the earliest cycle it can be placed in without causing any resource conflicts.
If compaction is restricted to basic blocks, reordering is uncritical, but for traces consisting of many basic blocks the branches have to be taken into account. Multiple conditional or unconditional jumps may leave the trace and there may be multiple nodes of the trace that represent entry points from outside into the trace. To preserve data dependencies, the scheduler has to take care of moving machine instructions accross conditional and unconditional

jumps. If machine instructions are moved from above a conditional jump to below it, it has to be ensured that this operation is also executed on the path that leaves the trace by the conditional jump. If the scheduler moves a machine instruction from below a conditional jump to above it, it must be ensured, that a copy of the machine instruction has to be added to incomming traces of the corresponding basic block. Also, it must be ensured that moving up definitions of variables will not change the programs semantics. Additional instructions that have to be added to other traces are called **compensation code**. The compaction of a certain trace therefore results in adding machine instructions to other traces for preserving program semantic correctness and may lead to slowing down the execution performance of the other traces. Thereby it does not take into account the execution frequencies of the trace compensation code is added to.
Nicolau [Nic84] has shown that trace scheduling may produce exponential number of machine instruction copies. Some heuristics were developed, that restrict the schedulers freedom of reordering machine instructions for preventing code explosion [LA83, Lin83, SDJ84, Ell86]. A good introduction to trace scheduling can be found in [WM95]. The movements of machine operations are described in form of transformations rules.

6.3.2 Percolation Scheduling

Percolation scheduling overcomes the problem of code explosion, but can still produce superfluous code in some cases. The technique was introduced by Nicolau [Nic85]. It does not consider a program as a set of basic blocks anymore. There is a set of transformation rules, that locally transform a program graph. The program graph is similar to the control flow graph. In contrast to the control flow graph there is only one type of nodes, where each node is associated with an instruction. In the terminology of Nicolau an instruction is a set of operations involving conditional jumps. Comparing to our terminology conditional jumps are corresponding to conditional expressions in the CFG. Operations can be compared with machine operations. In contrast to machine instructions, an instruction in the program graph must not necessarily consist of operations that can be performed in parallel. Percolation scheduling does not consider any resource constraints during the application of its transformations. An assumption in percolation scheduling is that operations always execute in one instruction cycle. An instruction may contain multiple conditional expressions that form a directed acyclic graph with respect to control flow, whose leaves are again nodes of the program graph, representing multi–way branches corresponding to the branching structure in the CFG.
The initial program graph has the same structure as the control flow graph (CFG with non–typed nodes). Percolation scheduling performs parallelizing transformations on the program graph moving up operations (or conditional jumps) in the program graph, while preserving the semantics of control flow and data flow. Repeatedly applying the transformation rules allows machine operations to *percolate* toward the top of the program graph. The process of scheduling is termed **migration**. Applying a core transformation always results in a semantically equivalent, but more parallel program graph. [Nic85] proposed four core transformation rules which were simplified to three transformations by [Aik88].
The basic idea of percolation scheduling is to start with a program graph that represents the original sequential program. Each node contains exactly one assignment or one test. The

core transformations are then repeatedly applied to the program graph. The order in which the transformations are applied is fundamental for two reasons:

1. the transformations are not confluent, i.e. when two different transformation can be applied to a graph G, resulting G' or G'' respectively, it can not be guaranteed that both G' and G'' can be transformed into a final unique result graph.

2. the transformations are not complete, i.e. a program graph cannot always be transformed into a semantically equivalent program graph with maximum parallelism.

Finding an optimal sequence of transformations is NP–complete. Therefore heuristics are applied for choosing the operation to move next. Approaches were proposed by Aiken and Nicolau [Aik88, AN88a] called *compact–block, compact–path, and compact–global*. Like in trace scheduling, the scheduler chooses frequently executed paths first. This method has a major problem: compact global relies on a target architecture with infinite resources that is able to concurrently execute an arbitrary set of machine operations without any restriction in any cycle. The created schedule has to be repartitioned with respect to the resource constraints of the target machine. Another disadvantage of percolation scheduling (or the superimposed techniques) is that the transformation rules enforce continious traversals of the program graph. This is due to the locality of the transformation rules, resulting in a slow compilation process.
Two overcome this problems two approaches have been developed integrating the consideration of resource constraints [EN89b] (e.g. percolation scheduling with resources constraints). Thereby no transformation rule move an machine operation into a node if this causes the requirement of more machine resources then available for the instruction.

GRiP

Global resource–constrained percolation scheduling **GRiP** [NN92] was motivated by the belief that resource constraints should be integrated into scheduling. It is not restricted to for using it in percolation scheduling but to be also used in parallelization techniques, such as trace scheduling or Enhanced Pipelined Percolation Scheduling [EN89a]. GRiP also introduced some improvements for software pipelining (keyword *gap prediction*).
Aggressive speculative movements (i.e. across conditional jumps) of operations in the absence of resource constraints can yield in moving possibly useless operations competing with useful operations for scarce resources. The knowledge about the resources available during scheduling allows more sophisticated decisions, e.g.

- when a large number of resources is available it might be worthwile to allow speculative movements;

- if only few resources are available, it might be better to prohibit the speculative movements until all non–speculative movements have been performed;

- data dependencies often permit operation to move farer than resource constraints would allow.

GRiP was inspired by a technique based on *unifiable operations* [EN89b]. The set of unify operations of a node n in the program graph is the set of all operations that can be moved to n without raising resource conflicts. Unifiable–Op scheduling consists of traversing a program graph in a top down fashion and filling the resources of each node (thus nodes are viewed as machine instructions) with the best nodes from the unifiable operation set. The disadvantage of this technique is that the computation of the sets is very expensive and must be incrementally updated, thus not really practicable for application. GRiP is based on the same principle, filling resources of nodes by using migration to move operations. When a node is scheduled, all operations that are in nodes that are dominated by the node considered can move, with respect to data dependencies and resource constraints. An operation becomes unmovable if

- it has moved into or above the actual node being scheduled, or
- a resource constraint prevents any application of the transformation rule of percolation scheduling, or
- it is prevented from moving by a data dependence on an operation itself not movable.

GRiP performs the following tasks:

1. A heuristic is applied to rank the importance of all operations of the program graph.
2. A set of moveable operations is determined for each node in the program graph. Initially these sets contain the operations of the subgraph that dominates the corresponding node.
3. Scheduling nodes is performed in top down traversal of the program graph. Migrating is performed to operations in the associated sets of the scheduled node in ranked order, until no operation is moveable anymore.

A disadvantage in GRiP (also in contrast to unifiable-op scheduling) is the occurence of **resource barriers**. Consider a path $[n_A, n_B, n_C]$; if there is an operation in n_C that is prevented from moving into n_B, because n_B is full and the operation would be movable from n_B to n_A, then n_B is called a resource barrier. These barriers prevent the moving of nodes through regions of programs with filled resources. Resource barriers can cause that operations with higher priority have to wait until nodes with lower priority have moved out of resource barriers. Thus the order indicated by the priority can be violated. This may cause, that an operation with lower priority may occupy a resource predestinated for an operation with higher priority stuck in a resource barrier. In [NN92] it is stated, that using adequate heuristics resource barriers are not likely a problem. The ordering of operations rely on heuristics. Any heuristic can be used in GRiP, e.g. heuristics used in list scheduling.

Trailblaizing Percolation Scheduling (TiPS)

Trailblaizing Percolation Scheduling [NN93] tries to overcome the incremental feature of percolation scheduling; only very local transformations are performed, therefore leading to very large sequencies of transformation for moving operations across larger program regions. This also has the disadvantage of producing more compensation code then necessary, enforced

by the transformation rules (code explosion). One reason is that transformations are based on the structure of the control flow graph. Therefore often transformations are applied to nodes the operation that is moved is not dependend on. The operation could be moved across regions without visiting any of such nodes.
Trailblaizing is based on **hierarchical task graphs** (HTGs; see chapter 2) [GP92] representing the essential dependencies and structure of a program. Trailblaizing extends percolation schedulings core transformations to exploit the structure of HTGs. By this technique operations can be moved across large program regions applying only a few transformation rules and avoiding a high amount of unnecessary compensation code. This also includes moving operations accross loops, a feature not inherent to normal percolation scheduling.
Trailblaizing improves compilation time by reducing the amount of transformation steps and also results in performence improvements of the parallelized code by allowing transformations not enabled in percolation scheduling. Trailblaizing is not dependent on any heuristics or considerations of resource constraints. These aspects are completly isolated from the basic algorithm.

Mutation Scheduling

Mutation Scheduling is a complete trade–off between code selection, register allocation and instruction scheduling. Mutation scheduling is based on trailblaizing percolation scheduling and also integrates GRiP for taking into account resource constraints.
Using GRiP, operations are scheduled using transformations from percolation scheduling and TiPS until either

- a true dependence, or
- a false dependency is encountered and dynamic renaming is not possible, i.e. there is no free register available, or
- a resource dependence blocks GRiP.

Mutation scheduling is then attempt to remove the occuring dependence, by trying an alternative operation for implementing the value considered from the mutation set. More aspects are discussed in capter 7.

6.3.3 Region Scheduling

Region scheduling is a technique for detecting coarse–grain and fine–grain parallelism. It was first proposed in [GS90]. The technique is based on a extended form of the program dependence graph. The scheduler is guided by estimations of present parallelism of the regions represented in the PDG. The region scheduler repeatedly transforms the extended PDG, uncovering potential parallelism until an estimate of the parallel capabilities in each region matches the parallel capabilities of the target architecture, or no transformations are applicable. The transformations defined for region scheduling can redistribute fine–grain parallelism among regions by the transfer of machine operations. Thus, overestimated parallelism in certain regions can be transfered to another region with unsufficient parallelism.

Trace scheduling and region scheduling both use reordering of the program to generate a schedule that enables parallel execution. In trace scheduling, transformations are applied based on execution frequency. In region scheduling, transformations are driven by parallel opportunities, i.e., the available parallelism of the underlying architecture is taken into account for selecting the next transformation step. The region scheduler can exploit at least as much parallelism as the trace scheduler [GS90]. In contrast to trace scheduling, the region scheduler is able to move complete regions to other regions. The drawback of trace scheduling generating fast schedules for highly prioritized traces at the expense of the others is avoided in region scheduling. The transformations are directed towards to increase parallelism. An improved approach of region scheduling was introduced by [AJLS92]. This approach termed enhanced region scheduling also incorporates software pipelining.
Gupta states, that the technique is architecture indipendent [GS90]. However, in region scheduling target machine code generation is done after all transformations were applied. This requires an architecture of idealized structure. The question how irregular register sets and data paths can be integrated are so far not considered.

6.4 Support of Architectural Features

The original goals of instruction scheduling are exploiting fine–grain parallelism (*compaction*) or -if no vertical architectures are considered - reordering of $\mathcal{A}$–MOs (*evaluation reordering*) with regards to reducing register pressure (with the aim of spill cost reduction). The approaches introduced in the previous section are basically concerned with the first issue (compaction).

Common Suppositions

We will have a look at the common suppositions of instruction scheduling:

1. Generally, idealized machine models are considered. I.e., identical functional units with immediate access to all storage resources are assumed; restricted connectivities are seldom addressed.

2. Resource constraints are considered with regards to the number of resources that are available. If all operations can be performed on all functional units (with access to all storage resources), resource allocation during instruction scheduling does not constitute any problem. I.e., binding operations to a functional units and determining storage resources for the operands has no impact on data dependent operations.

3. Conventional scheduling algorithms are confronted with the following decisions:

   - *choosing an appropriate machine instruction for inserting the operation;*
   - *choosing a machine operation from the data ready set.*

New Goals and Problems

The techniques described so far do not take into account the problems arising in the context of irregular architectures. Like in the section concerned with register allocation, we will now summarize the new situations, instruction scheduling is concerned with:

1. Neither identical functional units nor general access to storage resources is given.
2. Instruction scheduling is **mutual dependent** on code selection. A certain covering can result in a (possibly restrictive) binding of machine resources. Allocating machine resources by the scheduler indicates that data dependent operations are restricted to certain machine resources:
   - A storage resource, selected for the result of a machine operation may not be immidiatly accessable to the machine operations that use this value. In this case additional data movements are necessary, supposed, they are possible.
   - It is relevant to consider which operations are available if certain operations can only be performed on specific functional units.
3. **Spilling accross data routes** has to be considered. It also has to be taken into account, that spilling may not be possible from every storage resource→ **restricted spilling**.
4. The **strong mutual dependece** between functional units and storage resources enforces the scheduler to make trade–offs between the decisions of
   - *choosing a functional unit;*
   - *choosing result destinations.*

The effects of restricted connectivities are not examined for percolation scheduling and region scheduling so far. Complex data routes together with their effects on spilling are not considered in the approaches introduced so far. In [Hei93] some aspects of irregularity are integrated into a trace scheduler. Approaches considering the stated aspects are concerned with phase coupling, therefore described in section 7.

6.5 Retargeting Instruction Schedulers

If we consider the amount of instruction scheduling approaches, the subject of retargeting is seldom addressed (see section 3.3 page 31). So far, list scheduling and trace scheduling are the preferable candidate considered in this context. The basic principles of these techniques do not rely on a specific target machine architecture. However, the selection of a member of the data ready set relies on heuristics and on a certain ordering for making decisions (choosing a functional unit, or storage resource). No heuristic effectively supports the complete range of architecture classes [Hei93]. The order in which decisions are made may have impact on the code quality. Both, the order of decisions and the heuristic, depend on the features of the target architecture. The question is, how schedulers can be retargeted to effectively support this features. The problem of retargeting gets more complicated if the scheduler is

confronted with partial code selection, i.e., regarding different coverings. In this case, the selection of equivalent operations and different data routes must also be considered.
Percolation scheduling and region scheduling based schedulers have a modular and hierarchical concept. Transformation rules constitute the lowest level. Based on this rules, the selection of rules for moving an operation into a certain machine instructions is adapted. On top of this, heuristics for choosing operations for moving are implemented. This modular concept enables a fast modification of the schedular (by hand) with regards to new architectures that have to be supported. However, retargeting based on automatically adapting new heuristics is not considered. In [Bea91] an approach based on **genetic algorithms** is proposed. Thereby a local list scheduler learns which heuristics are adequate for a certain target architecture. In [Hei93] a trace scheduler is generated based on analysing the target architecture and **composed from certain subtasks** with regards to the following criteria:

- if each functional unit has access to all storage resources, data placement is of no concern and the selection of a functional unit has priority;
- if functional units have restricted data access, three classes of functional units are considered:
    1. *data placement is of no concern*: therefore, selection of a functional unit has priority;
    2. *data placement is of minor concern*: selection of functional unit still has priority, but direct access to storage resources is examined and incorporated in the decision;
    3. *data placement is of major concern*: promising destinations for the results of operations are choosen first.

A detailed analyses for further criterea is an important subject of further research. Determining fine–grained subtasks of scheduling, relevant for supporting certain hardware features becomes necessary, for utilizing an architecture based composition of schedulers. It is also necessary to find out how the outlined instruction scheduling techniques can cope with such decisions. As already stated in section 5.5, retargeting should consist of

- determining adequate techniques and heuristics (e.g., if no parallelism is provided by the machine, instruction scheduling is merely concerned with evaluation reordering, for minimizing register usage);
- a good composition of subtasks, constituting the fine-grain adjusted selected technique;
- taking into account the interaction and the coupling with other tasks of code generation.

The field of retargeting instruction schedulers is very scarcely investigated, therefore more efforts of research are very important in this area.

6.6 Summary

The research area of most interest is the the support for fine–grain parallelism, especially instruction level parallelism. There are many efforts made in developing global techniques and integrating

- consideration of resouce–constraints,
- finding adequate solutions for scheduling loops (software pipelining).

The basic drawback is that the impacts of irregular architectures are so far of no much major concern. There are efforts necessary to examine how these techniques can be augmented with the requirements arising from irregular architectures and extend the considerations of resource constraints to mutual dependencies of functional units, their destinations of of the results, and data dependend operations.
The second major drawback arises in the context of retargeting a scheduler. Heuristics used, may have fundamental impact on the generated machine code when different target machines are considered. How the scheduler can be automatically tailored to the requirements of the considered target machine is very a seldom subject of interest. Therefore, much investigations for retargeting schedulers seems to be necessary. This involves examinations of the degree of retargetability of the scheduling techniques and if certain scheduling techniques are restricted to support only special classes of architectures.

Chapter 7

Phase Coupling

In this section the problems resulting from performing the tasks of code generation strictly decoupled and in a certain order ae discussed. It is shown that a coupling of these tasks is an important issue, all the more the features of irregular architectures together with fine–grained parallelism are incorporated. Problems that have to be solved and approaches concerned with phase coupling are described. Phase coupling with other optimization tasks is not addressed. An overview of early works in the context of phase coupling is given in [AM87].

7.1 Phase Ordering Problems

We will first outline the basic issues that lead to phase ordering problems and indicate an integration for certain tasks of code generation. Code selection and register allocation can hardly be considered as separated tasks. The selection of a certain machine operation pattern immediately fixes a certain set of storage resources for the operands. With regards to instruction scheduling, a certain set of machine resource is fixed in advance. Binding certain machine resources before code selection will have great impact on the legal coverings (and may be even impossible). Therefore, each decision determind in one of the tasks effects the other. The generation of code selectors using tree pattern matching is a very sophisticated technique. Using dynamic programming, an optimal solution is selected with regards to a sequential view of program execution. The minimum cost covering selected by the dynamic programming approach is generally no optimal solution with respect to spill cost reduction and exploitation of parallelism. The selection of unfavourable machine operations can result in machine instructions that hardly contain parallelism and results in a high amount of spill code. An integration of spilling and parallelism aspects into the cost model would enforce tree pattern matching to consider context sensitive informations. But generally most of these informations necessary for selecting good code are only available when the code is selected. E.g., for determining effects on spilling, locations of all the other live values must be known; but for knowing the locations, the machine operation patterns for implementing operations have to be known; therefore, a covering has to be selected first. The basic drawback is, that certain operations and machine resources are fixed (bound) after code selection. Some of these decisions should be delayed until scheduling. For selecting between different operations and data routes, the integration of code selection into instruction scheduling becomes of growing

importance. Performing instruction scheduling before code selection can be performed for architectures with identical functional units that have access to all storage resources (e.g., region scheduling). Perfoming instruction scheduling before code selection is impracticable for irregular architectures.
The phase ordering problems between register allocation and instruction scheduling already occur in absens of distributed register sets: (1) Register allocation tries to reuse as many registers as possible, therefore adding many additional false dependencies that inhibit the instruction schedulers ability to reorder machine instructions. (2) Instruction scheduling tries to parallelize as many machine operations as possible therefore resulting in high register pressure which increases the amount of interferences drastically. If we perform register allocation before instruction scheduling (also stated as **early register allocation**) the instruction scheduler is inhibited in reordering the instructions by additional false dependencies, i.e., anti–, and output–dependencies. This dependencies are resulting from the re–definition of registers when multiple live ranges are mapped to the same register. This is an immediate result of the sequential view of statements as represented by the CFG, which is reflected in the live ranges of values and lead to certain interferences with other values. Non–overlapping live ranges may be mapped to the same physical register. If register allocation is performed after instruction scheduling (stated as **late register allocation**) new interferences may be introduced between values, thus resulting in possibly many spillings. Additional spillings mean new spill code, that also has to be scheduled. Therefore, a rescheduling becomes necessary [BS95].
The interdependence between machine resources and selected code leads to the strong mutal dependence of code selection, register allocation, and instruction scheduling. Informations necessary for performing certain code improvements are based on the available machine resources, related to operations and values:

- *spilling* is based on interference informations of values, whereby interference itself is dependend on the locations (i.e., storage resources) of values;

- *parallelism* is dependend on the machine resources occupied by data independend operations.

As machine resources are determined by the selected code and only legal coverings should be considered, tree pattern matching should be performed in advance. Otherwise there is no basis for constituting code improvements. A utilization of code improvements can be enabled on certain levels of delayed binding of machine resources:

- *Recomputation* of values, if this can be performed in parallel to already scheduled code.

- Construct $\mathcal{A}$–MOs , that bind as less machine resources as possible. A single covering is selected with conventional techniques. Thereby, a single covering should ensure, that each combination of machine resources specified by the covering is legal covering of coresponding machine operations. Hereby code selection is decoupled from the other tasks. If parallelism is proided by the machine, good trade–offs between transfer cost reduction (especially spill cost reduction) and exploiting parallelism have to be found

- Take different coverings of $\mathcal{A}$–MOs into account. This can be also partitioned in different levels of coverings (see section 4.4 page 56):
  - fixed machine operation patterns but different data routes;
  - machine operation patterns only differ in the set of storage resources, thus also different dataroutes are considered;
  - different granularity of covering the operators;
  - coverings, where certain operations are exchanged by applying algebraic transformations.

In the subsequent sections an overview of approaches is given that incorporate phase coupling, classified in the mentioned levels of coverings which are considered. Additional criterea are the considered architectural features and the degree of retargetability. As stated in the previous sections, this is concerned with the selection of techniques and their used heuristics, their combination and degree of integration.

7.2 Single Covering (Level–0)

7.2.1 Recomputation (Rematerialization)

One approach, called **rematerialization**, embeds code selection partially into graph coloring with the goal of reducing spill costs [CAC+81, BCT94]. The idea of rematerialization is to choose the least expensive mechanism to accomplish spilling. This is basically concerned with detecting situations where a recomputation of a value is more profitable than spilling the value to memory. In [CAC+81] it is pointed out that certain values can be recomputed by single machine operations and that certain required operands will always be available for the computation (e.g., immediate values in the machine instruction or hard coded constants). Chaitins allocator cannot handle rematerialization of live ranges comprising several values. An improvement is given in the approach of [BCT94].

7.2.2 Delayed Binding

$\mathcal{A}$–MOs leave a certain degree of freedom to the instruction scheduler, while maintaining the sequential view of traditional code selectors. An approach by Rainer Leupers, currently in progress at our institute, extracts $\mathcal{SR}$–MOs from a hardware description specified in MIMOLA [BBH+94]. A corresponding iburg specification is constructed. The generated tree pattern matcher selects a minimal cost covering. Local register allocation is performed, whereby each variable is assumed to be located in memory. Local instruction scheduling is then performed using an IP–solver that determines an optimal schedule on basic block level, taking into account given timing constraints. The basic drawback of using an IP–solver is that large basic blocks cannot be effectively scheduled anymore.

7.2.3 Taking into Account Potential Parallelism and Limited Registers

This subsection is concerned with register allocation techniques that take into account issues of parallelism and instruction scheduling taking into account the limited amount of registers.

Goodman and Hsu [GH88] compared two methods against both early and late register allocation. They developed a data dependence graph driven method on basic block level. They manipulate the schedulers data dependence graph, such that it's width is no greater than the number of registers available. Their second method is based on late register allocation and is called integrated prepass scheduling (IPS). Hereby, a local scheduler is restricted to use a fixed number of registers for local values (local pseudo registers) of each basic block. If this *register limit* is reached, the scheduler tries to free some of the registers, and may increase the register limit if freeing is not possible. The subsequent local register allocation can generate spill code which enforces a rescheduling. Bradlee compares two strategies with graph coloring followed by scheduling. All three strategies are embedded in the retargetable code generator MARION [BEH91, Bra91, BHE91]. The first strategie is an improvement of IPS and performes global postpass register allocation. The second one called RASE first perfomes initial passes of the instruction scheduler for estimatimating local schedule costs, given a very limited number of registers and then with the maximum number of available registers. The computed estimations are used in the priority scheme of graph coloring, followed by a local list scheduler. The MARION system is intended for constructing code generators for RISC like architectures, based on an instruction set model (including resource requirements of the instructions). It was developed for analyzing different code generation strategies, but an automatic selection of strategies is not described. Distributed register sets are considered, and *explicitly advanced pipelines* are supported, which requires the support of complex data routes.

Freudenberger [FR91] describes a method that integrates register allocation into trace–scheduling. The scheduler takes as many registers from a pool of available registers as it needs (*greedy*). It also saves information about which registers contain which values for the entry and exit points of a trace, i.e., the corresponding nodes the control flow graph, where control flow branches or coalesces. These informations are used to minimize data movements in the traces corresponding to the entry and exit points. As trace scheduling is performed on the crucial paths first, the global aspects of register allocation are incorporated, by allocating values to registers that are frequently used.

Norris and Pollok [NP93] perform early register allocation and add edges to the to the interfernce graph to estimate the re–ordering effect of instruction scheduling. They build the interference graph from the data dependence graph rather than from a linear representation like given by the control flow graph. Generally the data dependence graph contains more parallelism than the target machine offers to be executed in parallel. Large interference graphs are constructed that are hardly to color. Norris and Pollok developed several heuristics to reduce the amount of parallelism given by the DDG, while maintaining enough parallelism for utilizing the scheduler. Pinter [Pin93] also constructs an interference by adding additional edges. Therefore she first constructs a graph from the data dependence graph, where the transitive closure of all dependence edges are placed into a graph as undirected edges. Target machine resource conflicts are added that restrict the parallel execution of machine operations. From this resulting graph, the graph's complement is constructed and the union with the

register alocators interference graph is constructed. This resulting graph is called the *parallel interference graph*. Brasier [BS95] proposes a method based on late register allocation and limits the additional interferences to false dependencies that will limit the instruction scheduler. Only if spilling becomes necessary during late register allocation it is switched back to early register allocation. The interference graph of early register is aughmented with edges from the interference graph of late register allocation. Those edges are added between nodes (live ranges) in the early interference graph that are exclusively found in the late interference graph and which are colored with the same color in the early interference graph. The resulting schedule will be accepted. Further works based on utilizing register allocation with aspects of parallelism based on graph colloring are [AEBK94, NP94, NP95].
In Bersons approach [BGS94] the data dependence graph is incrementally sequentialized with regards to global aspects of over and under–utilized regions of resource requirements (excessive sets and resource holes, respectively). Register allocation is performed on–the–fly, together with appropriate spilling. Approaches like [ME92, NPW91, NN93] start with an initial register allocation. During instruction scheduling false dependencies are eliminated using dynamic renaming [CFR$^+$91]. But allocated registers are never released (e.g., by spilling).
The aspects of other phases considered in a certain phase are generally based on potential possibilities of parallelism or resource requirements that are often extremly over- or underestimated. Therefore research is merely directed to improve the precisness of estimations. The problematic issues of irregular register sets are not addressed in the described approaches. The effects of mutual dependencies of storage resources and functional units are avoided by either not involving restricted connectivities, or binding resources in advance. However, initial register allocation with the possibility of incrementally rejecting some of the decisions during instruction scheduling seems to be a good approach and should be further considered.

7.3 Data Routing (Level–1,2)

Data routing incorporates register allocation into scheduling, due to distributed register sets. Coverings are considered, containing fixed functional units for operations, but differ in data routes between definitions and uses. The aim of data routing is the selection of good routing paths for values, with regards to exploiting instruction level parallelism. The BULLDOG Compiler of Ellis [Ell86] and the CBC [Har92] perform local scheduling together with greedy register allocation on the fly. Hereby, good spill decisions are not considered. The approach proposed in [LCGM94] tries to overcome this lack by examining various data routes with regards to global spilling and recomputation. Functional units are bound in advance. Pattern matching is performed during scheduling by combining partial versions (see section 3.2.6 page 28) of machine operations to complete versions (**bundling**). Hereby, complex patterns accross basic block boundaries are taken into account. The approach takes into account the problematics of irregular register sets and is integrated in the synthesis and retargetable code generation system **CHESS**.
An approach combining delayed binding of functional units with consideration of different data routes is proposed in [Hei93]. Irregular register sets together with fine–grain parallelism are taken into account. Storage resources are composed to more abstract storage resources.

It is ensured, that definitions and the corresponding uses of values are always reachable. The code selector performs traditional tree pattern matching with dynamic programming. A trace scheduler is generated from a machine specification to guide the order of the choices the trace scheduler has to make with respect to the requirements of the target machine. The trace scheduler perfomes register allocation on–the–fly.

7.4 Integrated Code Selection (Level–3)

The approaches described here perform code selection during instruction scheduling. Incremental **tree hight reduction** (ITHR) partially integrates code selection into instruction scheduling. ITHR changes the structure of expressions according to associative or distributive properties of operators. Incremental tree hight reduction was used to change the structure of expressions during instruction scheduling [NPW91].
The retargetable code generator **MSSQ** embedded in the **MIMOLA** software system performs code selection within local instruction scheduling. A set of coverings is generated for each assignment statement. Each covering constitutes of $\mathcal{L}$–MOs , represented by the coresponding versions. The versions are constructed from partial versions during pattern matching (bundling). The partial versions are extracted from the structural description of the hardware (specified in MIMOLA). Variables are pre–allocated to certain storage resources, defined by the user. Temporary values are located to register cells on–the–fly during pattern matching, i.e., during partial version determination. Each temporary register cell can be only assigned in one version. The selection of versions is performed during local compaction. Transformation rules enable to consider algebraic transformation during compaction. Spilling and global optimizations are not considered in this approach.
Mutation scheduling is based on trailblaizing percolation scheduling. It integrates code selection and register allocation into instruction scheduling. Each value in the program is assosiated with a set of functional equivalent expressions, each using a different set of resources of the target architecture. The sets are called *mutation sets*. During instruction scheduling one of this alternatives is selected. If the resources for a selected expression are occupied, another expression (*mutation*) is selected. The mutation sets can change dynamically during scheduling to contain expressions that may become available for a value. When a value is evaluated into a register, a reference to that register is added to the mutation set. If a value is spilled, a *load* entry with the corresponding location is added. Initial register allocation is performed like in [ME92, NPW91, NN93] incorporating dynamic renaming for eliminating false dependencies. But in contrast to these approaches spilling is also integrated. If recomputation of a value has more advantages the recomputation of a value is selected [NN94]. In contrast to rematerialization, every equivalent expression can be selected. Incremental tree hight reduction is incorporated. The applied heuristics can be easily adjusted (by hand), due to the modular concept of percolation scheduling based approaches. However, the problematic issues of irregular register sets again are not of interest.

Chapter 8

Timing Constraints

This chapter is concerned with code generation for given timing constraints. I.e., the generated code must either fulfill a certain timing behavior specified by the designer (*explicit timing constraints*), or the code generator has to take care for timing constraints predicted by hardware components (*implicit timing constraints*). Explicit timing constraints are becomming of increasing importance in the context of real time systems (RTS). Implicit timing constraints are due to e.g. certain delay times of machine operations or maximum duration times a machine resource will hold a certain value (e.g. transient resources).
The runtime of certain program regions depends on the length of the final machine instruction sequence, and this sequence is only known after the final scheduling phase. Therefore, the analysis of timing constraints is commonly integrated into scheduling. There are the following basic research areas in the context of timing constraints:

- Modelling of explicit timing constraints and formal analysis if these constraints are feasible (cp. [Hon94, KM90b]).
- Scheduling with regards to explicit constraints. I.e., reordering the program by moving instructions from overloaded program regions (not fulfilling certain constraints) to non–critical program regions, with respect to given timing constraints (cp. [Hon94]).
- List scheduling with regards to given implicit timing constraints. So far, there are some extensions of list scheduling, incorporating the management of implicit timing constraints. This is achieved by labeling the edges of the data dependence graph with timing informations, i.e., each edge (n, n') is associated with a tuple (min, max). This indicates, that if n is scheduled in instruction i, n' has to be scheduled in instruction i', such that $i + min \leq i' \leq i + max$. List scheduling will not always result in valid schedules, if not all max values are set to $infty$, even if they exists a solution. There are several techniques proposed for increasing the likelihood of generating a valid schedule (consult [Bea91] for details):
  - *absolute timing*
  - *foresight scheduling*
  - *incremental foresight scheduling*
  - *lookahead scheduling*

Generally, code generation is involved for yielding the timing estimations for certain program fragments. Thus, the results obtained extremely depend on the code quality of the incorporated code generators. So far, we have not found existing approaches, integrating explicit timing constraints into code generation techniques, except the one exposed in [Hon94]. But here, timing constraints are analysed on a more coarse–grain and abstract level. There are no efforts made to generate high quality code. The feasibility of constraints is gained by moving insructions from critical regions to uncritical regions. This is performed by using an adapted version of trace scheduling.
There is some further research necessary to find out which techniques, developed in the *real–time–system* community, are adaptable to utilize code generation, especially instruction scheduling techniques.

Chapter 9

Summary

There is a high amount of techniques and superimposed improvements concerned with code selection, register allocation and instruction scheduling. These techniques were generally developed with regular classes of architectures in mind. With such suppositions, very sophisticated results are gained.
The **major research areas** and **tendencies of investigation** can be outlined as follows:

- *Tree pattern matching* is the preferable technique concerned with code selection. Tree pattern matchers can construct the complete set of coverings with respect to a regular tree grammar in effective time. *Dynamic programming* is incorporated for finding an optimal solution with regards to a given cost model.

- The common technique used for register allocation is *graph coloring*. It can cope with distributed register sets and register classes.

- The basic research issues for instruction scheduling are global techniques: *trace scheduling, percolation scheduling* and *region scheduling* were developed mainly for utilizing instruction level parallelism. The integration of resource constraints and software pipelining are the major research topics for improvements.

- A major research area is the support of architectures that provide **instruction level parallelism**. Global techniques are required for effectively exploiting the available parallelism. As traditional register allocation techniques rely on a strict ordering of statements, the pre–allocation of registers may restrict the scheduler. A post–allocation can lead to spill code that also has to be scheduled. For overcomming this mutual dependence, recent approaches are concerned with **phase coupling** of *register allocation and instruction scheduling*.

Irregular architectures indicate a strong mutual dependence of code selection, register allocation and instruction scheduling and indicate hard problems with regards to global optimizations, due to the strong interdependence of code selection and resource allocations: A certain covering may restrict the subsequent tasks, because of an unfavourable binding of resources, due to the selected $\mathcal{A}$–MOs. Therefore, binding should be delayed as long as possible. Effective delayed binding enforces the consideration of certain coverings. Thus, **phase coupling** of *code selection with register allocation and instruction scheduling* is an

important issue for supporting the generation of high quality code. Delayed binding of machine recources indicates, that several informations required for global optimizations are uncertain (e.g., locations of values, which are nececassry to determine interference). Thus, global optimizations are very hard to perform with traditional models, based on certain, static factors. Transformations performed during optimizations are usually based on global static factors. The major problem occuring is, that application of a transformation can completely destroy the suppositions responsible for performing the transformation. So far, approaches that address the features of irregular architectures avoid to incorporate all the problems caused. Either a certain problematic feature is not considered, certain global optimizations are not performed, or certain machine resources are bound in advance, thereby avoiding the problematic mutual dependencies. However, the described approaches were not developed for solving all the occuring problems.
For providing the complete integration of tasks with regards to all features of irregular architectures the following efforts seem to be required:

- Formal classification of coverings and specification of their features. Good representations must be developed including informations necessary for global optimizations, together with techniques for traversing coverings while incrementally updating the associated informations.

- Developement of incremental techniques, that take into acount global aspects like over and under–utilization of machine resources in program regions (approaches like mutation scheduling [NN94] and those proposed by Berson et alias [BGS94, BGS95] seem to be a good basis for further investigation). Thereby, the mutual dependencies of functional units and storage resources have to be regarded. Incremental approaches seem to be necessary, as each transformation performed during optimization can have much impact on factors that are necessary for further decisions and estimations. Questions have to be answered, how global optimizations should deal with uncertain informations.

- Integration of intelligent backtracking for rejecting unfavourable decisions should be considered.

Approaches described in chapter 7 should constitute the basis for further investigations. Many partial problems were solved and it should be examined how these solutions can be extended and/or combined for achieving a full integration. Furthermore, research is necessary for supporting autoincrement and autodecrement registers and ring buffers. This seems to be a very scarcely investigated area.
If we consider **retargeting**, tree pattern matching techniques were developed with the aim of fast retargeting. The target machine is specified by a behavioral model (instruction set model) based on regular tree grammars. Structural models can be converted to regular tree grammars. Additionally, they include informations for utilizing effective resource allocation, necessary for retargeting of register allocation and instruction scheduling. Graph coloring and instruction scheduling techniques can be regarded as potential retargetable. But this retargetability is rather viewed with regards to fast adaption than with regards to code quality. Retargetability considered is basically concerned with the application of a certain technique that cope with a high amount of common aspects of various target architectures. Questions

about retargeting register allocation and scheduling techniques with regards to an automatic selection of appropriate techniques (and incorporated heuristics) are of minor interest and have still to be examined. For supporting an adequate retargeting of code generation techniques the following investigations are necessary:

- *Classification of techniques* with respect to effective support for architecture classes. As stated, a certain and effective technique will not support the range of architecture classes with the same degree of code quality. Therefore, it is necessary to find out which techniques utilize which architectures, and which do not. Thus, the various approaches developed with regards to certain architectures are very valuable sources for such investigations.

- A *decomposition of the tasks of code generation into fine–grained subtasks*, whereby each subtask is responsable for certain decisions. The decomposition should enable the observation of effects due to a reordering or exchanging of some of the subtasks, with regards to the code quality.

- Find *criterea for determining a certain architecture class*. Techniques (or rules) for *composition of the corresponding code generation tasks* must be developed. It has to be determined if global or local techniques should be applied, or trade–offs between local and global techniques are desirable. With regards to phase coupling, determination of the degree of integration and interaction of tasks is necessary.

Furthermore, questions have to be answered, if effective retargeting of techniques can be performed automatically. Semi–automatcally support up to fully integrated user interactions should also be taken into account. **Specification models** are required for supporting both effective retargeting of all code generation tasks (with regards to high code quality) and design process together with synthesis. At least, such models should be convertable to common models. Behavioral models lack of effectively utilizing retargeting of all tasks, while multi cycle instructions are difficult to be extracted from structural models. Thus, adequate trade–offs are necessary. Investigations should be concerned with the following questions:

- What informations are necessary to support effective retargeting?

- How should they be represented (specified), with regards to design and synthesis support?

- Adaption (extraction, conversion) and relations to other existing models?

With regards to **timing constraints**, generally, code generation is involved for yielding the timing estimations for certain program fragments. Integration of explicit timing constraints (timing behavior of the system) into code generation is no issue of interest. Timing constraints are analysed on a more coarse–grain and abstract level. However, due to the amount of approaches, there is some further research necessary to find out which techniques, developed in the *real–time–system* community, are adaptable to utilize code generation.

Bibliography

[ADK+95] Guido Araujo, Srinivas Devadas, Kurt Keutzer, Sharad Malik, Ashok Sudarsanam, Steve Tjiang, and Albert Wang. Challenges in code generation. In Peter Marwedel and Gert Goossens, editors, *Code Generation for Embedded Processors*, chapter 3, pages 48–64. Peter Marwedel and Gert Goossens, 1995.

[AEBK94] Wolfgang Ambrosch, Anton Ertl, Felix Beer, and Andreas Krall. Dependence conscious register allocation. In Juergen Gutknecht, editor, *Programming Languages and System Architectures*, volume 782, pages 125–136. LNCS Series, Springer–Verlag, Zurich, Switzerland, March 1994.

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[Aik88] A. Aiken. *Compaction–Based Parallelization*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1988. TR 88–09–22.

[AJ76] Alfred V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, 1976.

[AJLS92] Vicky H. Allan, J. Janardhan, R.M. Lee, and M. Srinivas. Enhanced region scheduling on a program dependence graph. In *MICRO–25*, pages 72–80, 1992.

[AM87] Vicky H. Allan and Robert Mueller. Phase coupling for horizontal microcode generation. In *MICRO–20*, pages 115–125, 1987.

[AM95] Guido Araujo and Sharad Malik. Optimal code generation for embedded memory non–homogeneous register architectures. In *ISSS'95*, Princeton University, 1995. Submitted to Intl. Symp. on System Synthesis.

[AN88a] Alexander Aiken and Alexandru Nicolau. A developement environment for horizontal microcode. *IEEE Transactions on Software Engineering*, 14(5):584–594, May 1988.

[AN88b] Alexander Aiken and Alexandru Nicolau. Perfect pipelining: A new loop parallelization technique. In *European Symposium on Programming*, volume 300. LNCS Series, Springer–Verlag, 1988.

[ASU86] Alfred V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, New York, 1986.

[BBH+94] Steven Bashford, Ulrich Bieker, Berthold Harking, Rainer Leupers, Peter Marwedel, Andreas Neumann, and Dietmar Voggenauer. The mimola language version 4.1. Internal Report, University of Dortmund, September 1994.

[BCT91] Preston Briggs, K. Cooper, and L. Torczon. Aggressive live range splitting. Technical report, Rice University, 1991.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[Bea91] Steven John Beaty. *Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Department of Mechanical Engineering, Colorado State University, Fort Collins, Colorado, Fall 1991.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 122–131, Santa Clara, California, 1991.

[Bel66] L.A. Belady. A study for replacement algorythms for a virtual–storage computer. *IBM System Journals*, 5(2):78–101, April 1966.

[Ben94] Manuel Enrique Benitez. *Register Allocation and Phase Interactions in Retargetable Optimizing Compilers*. PhD thesis, University of Virginia, May 1994.

[BFMR92] Jean-Michel Berge, Alain Fonkoua, Serge Maginot, and Jaques Rouillard. *VHDL Designers Reference*. Kluwer Academic Publishers, 1992.

[BGG+89] D. Bernstein, D. Goldin, M. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Pinter. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[BGS94] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Resource spacling: A framework for integrating register allocation in local and global schedulers. *Working Conf. on Parallel Architectures and Compilation Techniques*, August 1994.

[BGS95] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Gurrr: A global unified resource requirements representation. *SIGPLAN Notices*, 30(4):23–34, April 1995. Proceedings of the ACM SIGPLAN on Intermediate Representations IR'95.

[BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion system for retargetable instruction scheduling. *SIGPLAN Notices*, 26(6):229–240, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.

[BMO90] R.A. Ballence, A.B. Maccabe, and K.J. Ottenstein. The program dependence web: A representation supporting control–, data–, and demand–driven interpretation of imperative languages. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 257–271, June 1990.

[Bra91] David G. Bradlee. Retargetable instruction scheduling for pipelined processors. PhD Thesis 91-08-07, Dept. of Computer Science, Univ. of Washington, 1991.

[Bra95] Marc Michael Brandis. *Optimizing Compilers for Structured Programming Languages.* PhD thesis, ETH Zurich, 1995.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, Texas, April 1992.

[BS95] Thomas S. Brasier and Phillip H. Sweany. Craig: A practical framework for combining instruction scheduling and register assignment. In *PACT'95*, Limassol, Cypros, 1995.

[CAC+81] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markenstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[CDN94] Andreas Capitanio, Nikil Dutt, and Alexandru Nicolau. Partitioning of variables for multiple register-file vliw architectures. In *Proceedings of the International Conference on Parallel Processing*, pages I 298–301, 1994.

[CF87] Ron Cytron and Jeanne Ferrante. What is a name? - the value of renaming for parallelism detection and storage allocation. In *Proceedings of the Sixteenth International Conference on Parallel Processing*, pages 19–27, University Park, Pennsylvania, 1987. The Pennsylvania University Press.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth F. Zadeck. An efficient method of computing static single assignment. *SIGPLAN Notices*, pages 25–35, January 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth F. Zadeck. Efficiently computing the static single assignment and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CH84] F.C. Chow and J.L. Hennessy. Register allocation by priority–based coloring. *SIGPLAN Notices*, 19(6):222–232, June 1984. Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. *SIGPLAN Notices*, 26(6):192–203, 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.

[Coh94] William Eden Cohen. *Automatic Construction of Optimizing, Parallelizing Compilers from Specification*. PhD thesis, Purdue University, December 1994.

[DeW76] D.J. DeWitt. *A Machine–Independent Approach to the Problem of Optimal Horizontal Microcode*. PhD thesis, Department of Computer and Communication Science University of Michigan, Ann Arbor, MI, 1976.

[DLSM81] Scott Davidson, David Landskov, Bruce D. Shriver, and Patrick W. Mallet. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, C-30(7):460–477, July 1981.

[DST80] P. J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.

[DT76] S. Dasgupta and J. Tartar. The identification of maximal parallelism in straight–line microcode. *IEEE Transactions on Computers*, C–25(10):986–992, October 1976.

[Ell86] J.R. Ellis. Bulldog: A compiler for vliw architectures. The MIT Press, Cambridge, Mass., 1986.

[Emm92] H. Emmelmann. Code selection by regular controlled term rewriting. In R. Giegerich and S.L. Graham, editors, *Code Generation: Concepts, Tools, Techniques ,Workshop in Computing Series*, pages 3–29. Springer–Verlag, Berlin, Heidelberg, 1992.

[EN89a] K. Ebcioglu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches. In *2nd Workshop on Programming Languages and Compilers for Parallel Computing*, 1989.

[EN89b] K. Ebcioglu and Alexandru Nicolau. A global resource–constrained parallelization technique. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 154–163, 1989.

[ESL89] Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. BEG – A generator for efficient back ends. *SIGPLAN Notices*, 24(7):227–237, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[FH88] Field and Harrison. *Functional Programming*. Addison–Wesley, 1988.

[FHKM94] Andreas Fauth, G. Hommel, A. Knoll, and C Mueller. Global code selection for directed acyclic graphs. In Peter A. Fritzson, editor, *Compiler Construction*, volume 786 of *LNCS*, pages 128–141. Springer–Verlag, Eddinburgh, U.K., April 1994. 5'th International Conference, CC'94.

[FHP92a] C. Fraser, R. Henry, and Todd A. Proebsting. Engeneering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[FHP92b] C. Fraser, R. Henry, and Todd A. Proebsting. BURG – fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[Fis81] J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C–30(7):478–490, July 1981.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependency graph and its use in optimizations. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FR91] Stefan M. Freudenberger and John C. Ruttenberg. *"Phase Ordering of Register Allocation and Instruction Scheduling"*. In Robert Giegerich and Susan L. Graham, editors, *"Code Generation — Concepts, Tools, Techniques", Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, 20-24 May 1991*, Workshops in Computing, pages 146–172. Springer-Verlag, 1991. ISBN 3-540-19757-5 and 3-387-19757-5.

[Fre74] R.A. Freiburhouse. Register allocation via usage counts. *Communications of the Association of Computer Machinery*, 17(11):638–642, November 1974.

[FSW94] Christian Ferdinand, Helmut Seidl, and Reinhard Wilhelm. Tree automata for code selection. *Acta Informatica,Springer-Verlag*, pages 741–760, 1994.

[Gas89] F. Gasperoni. Compilation techniques for vliw architectures. Technical report, Courant Institute of Mathemathical Science, New York University, March 1989.

[GFH82] Mahadevan Ganapathi, C.N. Fisher, and J.L. Hennessy. Retargetable compiler code generation. *Computing Surveys*, 14(4), October 1982.

[GH88] J. Goodman and W. Hsu. Code scheduling and register allocation. In Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, 1988.

[GJ79] Michael R. Garey and David S. Johnson. Computers and intractability: A guide to the theory of np–completness. W.H. Freemann & Co, 1979.

[GP92] Milind Girkar and Constantine D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, March 1992.

[GR77] S.L. Graham and R.S.Glanville. A new method for compiler code generation. *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 231–240, 1977.

[GS90] Rajiv Gupta and Mary Lou Soffa. Region scheduling: An aproach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.

[GSS89] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, pages 264–274, July 1989.

[Har92] R. Hartmann. Combined scheduling and data routing for programmable asic systems. In *Proceedings of EDAC'92*, pages 486–490, March 1992.

[Hei93] Werner Heinrich. *Formal Desciption of Parallel Computer Architectures as a Basis of Optimizing Code Generation*. PhD thesis, TU Munich, 1993.

[Hen89a] R. Henry. Algorithms for table–driven code generators using tree pattern matching. Technical Report 89–02–03, Computer Science Department, University of Washington, Seattle, WA 98195 USA, 1989.

[Hen89b] R. Henry. Encoding optimal pattern selection in atable–driven bottom–up tree pattern matcher. Technical Report 89–02–04, Computer Science Department, University of Washington, Seattle, WA 98195 USA, 1989.

[Hen89c] R. Henry. Performance of table–driven code generators using tree pattern matching. Technical Report 89–02–02, Computer Science Department, University of Washington, Seattle, WA 98195 USA, 1989.

[Hon94] Seongsoo Hong. *Compiler–Assisted Scheduling for Real–Time Applications: A Static Alternative to Low–Level Tuning*. PhD thesis, University of Maryland, 1994.

[JM86] M.S. Johnson and T.C. Miller. Effectiveness of a machine–level global optimizer. *SIGPLAN Notices*, 21(7):99–108, July 1986. Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction.

[Joh94] Richard Craig Johnson. *Efficient Program Analysis Using Dependence Flow Graphs*. PhD thesis, Graduate School of Cornell University, 1994.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. *SIGPLAN Notices*, 26(6):78–89, 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[KM90a] K. Knobe and A. Meltzer. Control tree based register allocation. Technical report, COMPASS, 1990.

[KM90b] Davis Ku and Giovanni De Micheli. Relative scheduling under timing constraints. *27th ACM/IEEE Design Automation Conference*, pages 59–64, 1990.

[Kog91] Peter M. Kogge. *The Architecture of Symbolic Computers*. McGraw–Hill, 1991.

[LA83] J. Lah and D.E. Atkins. Tree compaction in microprograms. In *Proceedings of the 16th Annual Workshop on Microprogramming*, pages 23–33, 1983.

[Lav62] S.S. Lavrov. Store economy in closed operator schemes. Journal of Computational Mathematics and Mathematical Physics 3, 1962. 1(4):687–701.

[LCGM94] Dirk Lanner, Marco Cornero, Gert Goossens, and Hugo De Man. Data routing: a paradigm for efficient data–path synthesis and code generation. In *Proc. 7th IEEE/ACM Int. Symp. on High–Level Synthesis*, May 1994.

[LDSM80] D. Landskov, S. Davidson, B.D. Shriver, and P.W. Mallet. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, 1980.

[LH86] J.R. Larus and P.N. Hilfinger. Register allocation inthe spur lisp compiler. *SIGPLAN Notices*, 21(7):255–263, July 1986. Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction.

[Lin83] J.L. Linn. Srdag compaction – a generalization of trace scheduling to increase the use of context information. In *Proceedings of the 16th Annual Workshop on Microprogramming*, pages 11–22, 1983.

[LM94] Rainer Leupers and Peter Marwedel. Instruction set extraction from programmable structures. In *Proc. EURO-DAC 1994*. 1994. http://ls12-www.informatik.uni-dortmund.de/publications/brief.html.

[Man93] M. Morris Mano. *Computer System Architecture*. Prentice Hall Internal Editions, 1993.

[Mar93] Peter Marwedel. Mssv: Tree–based mapping of algorithms to predefined structures. Technical Report Report No. 431, Department of Computer Science, University of Dortmund, January 1993.

[MB83] D.W. Matula and L.L. Becks. Smallest–last ordering and clustering and graph coloring algorithms. *Journal of ACM*, 30(3):417–427, July 1983.

[ME92] S. Moon and K. Ebcioglu. An efficient resource constraint global scheduling technique for superscalar and vliw processors. In $MICRO$, December 1992.

[Nic84] Alexandru Nicolau. *Parallelism, Memory Anti–aliasing, and Correctness Issues for a Trace Scheduling Compiler*. PhD thesis, Department of Computer Science, Yale University, New Haven, Conn, December 1984.

[Nic85] Alexandru Nicolau. Percolation scheduling: A parallel compilation technique. Technical report, Department of Computer Science, Cornell University, Ithaca, New York, May 1985.

[NN92] Steven Novack and Alexandru Nicolau. An efficient global resource constrained technique for exploiting instruction level parallelism. In Kang G. Shin, editor, *Proceedings of the International Conference on Parallel Processing*, pages II 297–301, August 1992.

[NN93] Steven Novack and Alexandru Nicolau. Trailblaizing: A hierarchical approach to percolation scheduling. Technical Report TR–92–56, Irvine University, August 1993.

[NN94] Steven Novack and Alexandru Nicolau. Mutation scheduling: A unified approach to compiling for fine–grain parallelism. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 892 of *LNCS*, pages 16–30. Springer–Verlag, Ithaca,NY,USA, August 1994.

[NP93] Cindy Norris and L. Pollok. A scheduler–sensitive global register allocator. In *Proceedings of Supercomputing'93*, 1993.

[NP94] Cindy Norris and L. Pollok. Register allocation over the program dependence graph. *SIGPLAN Notices*, 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[NP95] Cindy Norris and L. Pollok. Register allocation sensitive region scheduling. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'95)*, 1995.

[NPW91] Alexandru Nicolau, R. Potasman, and H. Wang. Register allocation, renaming and their impact on parallelization. In *Languages and Compilers for Parallel Computing*, volume 589. LNCS Series, Springer–Verlag, 1991.

[PBJS90] Keshav Pingali, Micah Beck, Richard Johnson, and Paul Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. http://cs-tr.cs.cornell.edu/TR/CORNELLCS:TR90-1152/Print, September 1990.

[PF92] Todd A. Proebsting and Charles N. Fisher. Probabilistic register allocation. In Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, pages 300–310, June 1992.

[Pin93] S.S Pinter. Register allocation with instruction scheduling. In Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, pages 248–257, 1993.

[PLG88] Eduardo Pelegrí-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application of BURS theory. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 294–308, San Diego, California, January 1988.

[PLMS95] Pierre G. Paulin, Clifford Liem, Trevor C. May, and Shailesh Sutarwala. Flexware: A flexible firmware developement environment for embedded systems. chapter 4, pages 67–84. Kluver Academic Publishers, 1995.

[RT74] C.V. Ramamoorthy and M. Tsuchiya. A high–level language for horizontal microprogramming. *IEEE Transactions on Computers*, C–23(8):791–801, August 1974.

[San94] Nandakumur Sankaran. Program optimizations via locality. Master's thesis, Graduate School of Clemson University, December 1994.

[SDJ84] B. Su, S. Ding, and L. Jin. An improvement of trace scheduling for global microcode compaction. In *Proceedings of the 17th Annual Workshop on Microprogramming*, pages 78–85, 1984.

[SS93] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768, pages 633–655, Portland, Oregon, USA, August 1993. Springer LNCS.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison Wesley, 1995.

[YST74] S.S. Yau, A.C. Schowe, and M. Tsuchhiya. On storage optimization of horizontal microprograms. In $MICRO-7$, pages 98–106, Pao Alto, CA, October 1974. Proceedings of the 7th Micro Programming Workshop.

Index