Chapter 3
A Short Introduction to Compilers

This chapter summarizes briefly some basic terms and definitions of compiler construction as well as the underlying concepts. It focuses mainly on the terminology but not on detailed algorithms. More comprehensive surveys can be found, e.g., in [3, 229, 244].

3.1 General Overview

A compiler is a program that translates a program written in one language (the source language) into a semantically equivalent representation in another language (the target language). Over the years, new programming languages have emerged, the target architectures continue to change, and the input programs become ever more ambitious in their scale and complexity. Thus, despite the long history of compiler design, and its standing as a relatively mature computing technology, it is still an active research field. However, the basic tasks that any compiler must perform remain essentially the same.

Conceptually, the translation process can be subdivided into several phases as shown in Fig. 3.1. The first is the *analysis* phase, often called the *frontend*, which creates an *intermediate representation* (IR) of the source program. On this specification, many compilers apply a sequence of high-level, typically machine-independent optimizations to transform the IR into a form that is better suitable for code generation. This includes tasks such as common subexpression elimination, constant folding, and constant propagation. A very common set of high-level optimizations is described in [1]. This phase is also referred to as the *midend* of the compiler. Finally, the *synthesis* phase, or the *backend*, constructs the desired target program from the IR. The concrete organization within each phase, however, may strongly vary between different compilers, especially that of the optimizations in the midend and backend.

Frontend and backend are presented in more detail in the following sections.

3.2 Compiler Frontend

The first phase in the frontend is the *lexical analysis*. A *scanner* breaks up the program into constituent pieces, called *tokens*. Each token denotes a primitive element

DOI 10.1007/978-1-4419-1176-6_3, © Springer Science+Business Media, LLC 2010

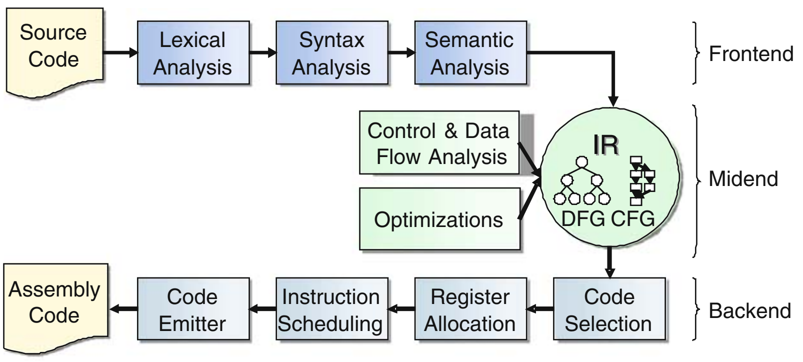


Fig. 3.1 Common compiler phases

of the source language, e.g., a keyword, an identifier, a character, etc. Generally, most of these elements can be represented by *regular expressions*, which can be parsed by *finite state machines* (FSMs). An FSM consists of a finite number of states and a function that determines transitions from one state to another as symbols are read from an input stream (i.e., the source program). The machine transitions from state to state as it reads the source code. A language element (e.g., a keyword or an integer number) is accepted if the machine reaches one of a designated set of *final* states. In this case, a corresponding token is emitted and the machine returns to the initial state to proceed with the next character in the stream. Given a list of regular expressions, scanner generators such as GNU's FLEX [106] can produce C code for the corresponding FSM that can recognize these expressions.

Definition 3.1 (Context-free grammar) *A context-free grammar G is a tuple $G = (T, N, R, S)$, where T denotes a finite set of terminals (i.e., the set of possible tokens), N a finite set of nonterminals, and $S \in N$ the start symbol. R is a relation from X to $(T \cup N)^*$, where X must be a member set of N.*

The tokens are then further processed by the *parser* to perform a *syntax analysis*. Based upon a *context-free grammar*, it identifies the language constructs and maintains a symbol table that records the identifiers used in the program and their properties. The result is a *parse tree* that represents a derivation of the input program from the start symbol S. If the token string contains syntactical errors, the parser may produce the corresponding error messages. Again, parser generators are available (e.g., GNU's BISON [105]), which can generate a C implementation from a context-free grammar specification.

Finally, a *semantic analysis* is performed that checks if the input program satisfies the semantic requirements as defined by the source language; for instance, whether all used identifiers are consistently declared and used. For practical reasons, semantic analysis can be partially integrated into the syntax analysis using an *attribute grammar* [67], an "extended" context-free grammar. Such grammars allow the annotation of a symbol $s \in (T \cup N)$ with an attribute set $A(s)$. An attribute $a \in A(s)$ stores semantical information about a symbol's type or scope. Each grammar rule r, with $r \in R$, using a can be assigned an attribute definition $D(a)$. The attributes are divided into two groups: *synthesized* attributes and *inherited* attributes. The former are used to pass semantic information up the parse tree, while inherited attributes passing them down. Both kinds are needed to implement a reasonable semantic analysis. Such attribute grammar specifications can be further processed by tools such as OX [143] (an extension of FLEX and BISON) to finally create a parser with integrated semantic analysis.

The output IR format of the frontend is typically a list of expression trees or *three-address code*. Generally, the frontend is not dependent on the target processor. Thus, an existing language frontend can be combined with any target-specific backend, provided that all of them use the same IR format (Fig. 3.2).

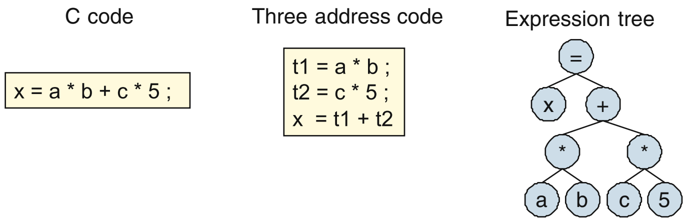


Fig. 3.2 IR format examples

3.3 Compiler Backend

The task of the backend is the code generation that consists of several subtasks. Since many of them are known to be NP-complete [163] problems, i.e., solving such problems most likely requires algorithms with exponential runtime, code generation typically relies on heuristics. Therefore and due to software engineering reasons, all code generation tasks are implemented by separate algorithms. However, these tasks are usually interdependent, i.e., decisions made in one phase impose constraints in subsequent phases. While this works well for regular architectures, it typically results in poor code quality for irregular architectures [270]. This is also known as the *phase coupling* problem.

Before the different subtasks are presented in the following sections, several program representations essential for most code generation subtasks (and for most compiler optimizations) are introduced first.

3.3.1 Data- and Control-Flow Graphs

The data- and control-flow graphs provide more detailed information about the program semantics than the plain IR representation. First, the *control flow* needs to be computed. Each function is split into its *basic blocks*.

Definition 3.2 (Basic block) *A basic block $B = (s_1, ..., s_n)$ is a sequence of IR statements of maximum length, for which the following conditions are true: B can only be entered at statement s_1 and left at s_n. Statement s_1 is called the leader of the basic block. It can either be a function entry point, a jump destination, or a statement that follows immediately after a jump or a return.*

Consequently, if the first statement of a basic block is executed, then all other statements are executed as well. This allows certain assumptions about the statements in the basic block, which enable the rearrangement of computations during scheduling for instance. Basic blocks can be easily computed by searching for IR nodes that modify the control flow of the program (e.g., goto and return statements). Once the basic blocks have been identified, the *control-flow graph* can be constructed. An example is given in Fig. 3.3.

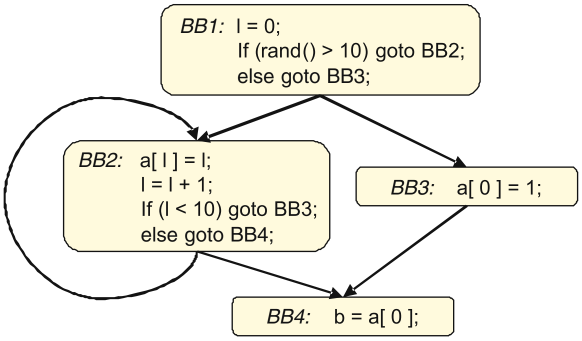


Fig. 3.3 Control-flow graph example

Definition 3.3 (Control-flow graph) *A control-flow graph (CFG) of a function F is a directed graph $G_F = (V_F, E_F)$. Each node $v \in V_F$ represents a basic block, and E_F contains an edge $(v, v') \in V_F \times V_F$ if v' might be directly executed after v. The set of successors succ of a basic block B is given by $succ_B = \{v \in V_F \mid (b, v) \in E_F\}$ and the set of predecessors pred of a basic block B is given by $pred_B = \{v \in V_F \mid (v, b) \in E_F\}$.*

The obvious edges are those resulting from jumps to explicit labels as the last statement s_n of a basic block. Furthermore, if s_n is a conditional jump or a conditional return, then a *fallthrough* edge to the successor basic block is additionally created. In certain cases, s_n is not a jump nor a return. Thus, in case a successor block exists and its first statement follows immediately after s_n in the IR representation, an edge to the successor block is created. Blocks without any outgoing edges have a return statement at the end. In case the resulting CFG contains unconnected basic blocks, there is an unreachable code that can be eliminated by a *dead code elimination* optimization without changing the program semantics.

While the CFG stores the control flow on a basic block level, another important data structure deals with the *data dependencies* between statements.

Definition 3.4 (Data dependency) *A statement s_j of a basic block $B = (s_1, ..., s_n)$ is data dependent on statement s_i, with $i < j$, if s_i defines a value that is used by s_j (i.e., s_i needs to be executed before s_j).*

A *data-flow analysis* (DFA) in its simplest form computes the data dependencies just for single basic blocks, and thus is referred to as local DFA. Basically, for each statement S, a data-flow equation is created, which requires the following information:

- in[S], the directly available information before S
- out[S] the information available after S
- gen[S] new information generated within S
- kill[S] the information killed by S

The equations depend on the kind of data-flow information that has to be computed. For the computation of reaching definitions, the equations have the following form:

$$\mathrm{in}[S] = \bigcup_{p \in \mathrm{pred}(S)} \mathrm{out}[p] \tag{3.1}$$

$$\mathrm{out}[S] = \mathrm{gen}[S] \cup (\mathrm{in}[S] - \mathrm{kill}[S]) \tag{3.2}$$

In order to obtain the information about available expressions, the equations change to

$$\mathrm{in}[S] = \bigcap_{p \in \mathrm{pred}(S)} \mathrm{out}[p] \tag{3.3}$$

$$\mathrm{out}[S] = \mathrm{gen}[S] \cup (\mathrm{in}[S] - \mathrm{kill}[S]) \tag{3.4}$$

Similar data-flow equations exists to compute the variables that are active at a certain program point. This information is required, e.g., for the register allocation. Solving the resulting system of equations gives the concrete data-flow information for the basic block. The result is stored in a *Data Flow Graph* (DFG).

Definition 3.5 (Data-flow graph) *A data-flow graph (DFG) for a basic block B is a directed acyclic graph $G_B = (V_B, E_B)$, where each node $v \in V_B$ represents an input operand (constant, variable), an output (variable) operand, or an IR operation. An edge $e = (v_i, v_j) \in E_B \subset V_B \times V_B$ indicates that the value defined by v_i is used by v_j.*

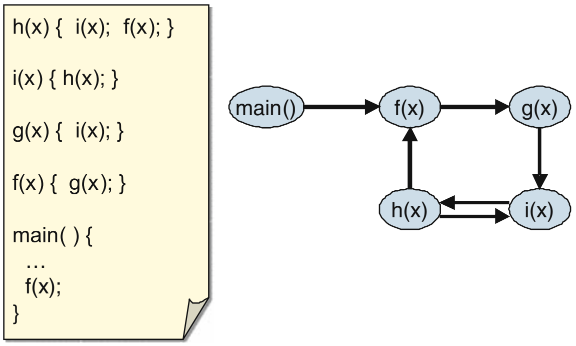
A DFG is called *data-flow tree* (DFT) if no node has more than one outgoing edge, i.e., there are no *common subexpressions*. Typically, DFTs build the input data for many popular code-selection techniques.

In practice, compilers perform a DFA for an entire function, called global DFA, since local DFA hinders many optimization opportunities. Suppose, a basic block has several outgoing control-flow edges, i.e., a definition of a variable (e.g., initialized with a constant) may reach multiple uses, possibly in different basic blocks. Thus, in order to exploit the full potential of, e.g., constant propagation, all uses reached by that definitions are required, which can only be provided by a global DFA. Typically, local DFA is embedded as a sub routine in the global DFA that iteratively solves the data-flow equations for an entire procedure.

The analysis can be extended even beyond function boundaries. The general idea behind a so-called interprocedural analysis is to collect the information flowing into a function and then use it, to update the local information. This requires information about

- which functions f_t any particular function f calls,
- f's return values,
- which functions f_c call any particular function g, and
- which arguments f_c passes to g.

The information about the calling behavior is usually captured in the concept of a *call graph*. Figure 3.4 depicts an example call graph.



```
h(x) {  i(x);  f(x); }

i(x) { h(x); }

g(x) {  i(x); }

f(x) {  g(x); }

main( ) {
  ...
  f(x);
}
```

Fig. 3.4 Call graph example

Definition 3.6 (Call graph) *If a program P is composed of functions $f_1, \ldots, f_n$, then the call graph for P is a directed, edge-annotated graph $G_P = (V, E, s)$ with $V = \{f_1, \ldots, f_n\}$, $E \subset V \times V$, and $s : E \mapsto S$, where S is the set of call sites. If $e = (f_i, f_j) \in E$ and $s(e) = k$, then the function f_i calls the function f_j from the label k inside f_i.*

The interprocedural analysis therefore starts with the creation of a call graph to capture the dependencies. If the whole program is visible to the compiler, the direct and correct creation of a call graph is straightforward. Regardless of that, most modern software consists of separate compilation units, which are linked after their separate compilation to form the final program. The compiler is therefore not able to analyze the whole program at once. This also prohibits the creation of a complete call graph, since several uncertainties arise:

- *Library functions* may be called by the code known to the compiler. In that case, the name and type of the callee are usually known, but the code is not analyzable.
- A function might be called by a code outside the compilation unit. This is usually the case if the compiled module is part of a library. Many languages allow to

specify storage-class attributes such as the `static` keyword in C. Using those effectively rules out this possibility for specific functions.

- Functions may not be called directly by name but via function pointers. In that case, an explicit data flow analysis is necessary to determine the set of actual callees of a function call in a program.

3.3.2 Code Selection

Code selection is typically the first phase in the backend. Its task is to map the IR to a semantically equivalent sequence of machine instructions. A common technique for code selection uses DFTs as input and is based on *tree parsing*. This can be efficiently implemented by tree pattern matching combined with dynamic programming [2]. The basic idea is to describe the instruction-set of the target processor by a *context-free tree grammar* specification.

Definition 3.7 (Context-free tree grammar) *A context-free tree grammar G is a tuple $G = (T, N, P, S, w)$, where T denotes a finite set of terminals, N a finite set of nonterminals, and $P \subseteq N \times (N \cup T)^*$ a set of production rules. $S \in N$ is the start symbol and w is a cost metric $P \rightarrow \mathbb{R}$ for the production rules.*

In the context of tree pattern matching, T can be seen as the set of all IR nodes and N as some sort of temporaries or storage location (e.g., registers or memory) to transfer intermediate results either between or inside instructions. The *cost metric* describes the costs caused by executing the corresponding instruction, e.g., with regard to performance, code size, or power consumption. The target code is generated by reducing the DFT to a single node (or covering the DFT) by repeatedly applying one of the production rules P, i.e., a subtree T can be replaced by a nonterminal $n \in N$ if the rule $n \rightarrow T$ is in P.

As a typical example for a tree grammar rule, consider the rule for a register to register `ADD` instruction:

$$\text{reg} \rightarrow \text{PLUS(reg, reg)}\{\text{costs}\} = \{\text{actions}\} \tag{3.5}$$

with $reg \in N$ and PLUS $\in T$. If the DFT contains a subtree that matches a subtree whose root is labeled by the operator “PLUS” and its left and right sons are labeled with “*reg*,” it can be replaced by *reg*. It should be noted here that both sons might also be the result of further tree grammar rules that have been applied before. Each rule is associated with a *cost* and an *action* section. The latter typically contains the code to emit the corresponding assembly instruction.

It might happen that more than one rule covers a subtree. A cover is optimal if the sum over all costs of involved rules is minimal. This can be implemented by a *dynamic programming* approach, i.e., the optimum solution is based on the optimum solution of (typically smaller) subproblems. More specifically, a tree pattern matcher traverses the DFT twice:

In the first *bottom-up* traversal, each node i of a DFT T is labeled with the set of nonterminals it can be reduced to, the cheapest rule $r \in P$ producing n and the total cost (i.e., the costs covering the subtree rooted at i). This includes also those nonterminals that might be produced by a sequence of rules. When the root node of T has been reached, the rule that produces the start nonterminal S with minimum cost is known.

In a second *top-down* traversal, the pattern matcher exploits the fact that a rule for a node i also implicitly determines the nonterminals the subtrees of i must be reduced to (otherwise the rule could not have been applied to i). Thus, starting at the root node, it can now be determined which nonterminals must be at the next lower level in T. Therewith for each nonterminal, the corresponding rule r can be obtained whose action section emits finally the instructions. This traversal is recursively repeated until the leaves of T have been reached. Figure 3.5 illustrates this process using the tree grammar specification in Table 3.1.

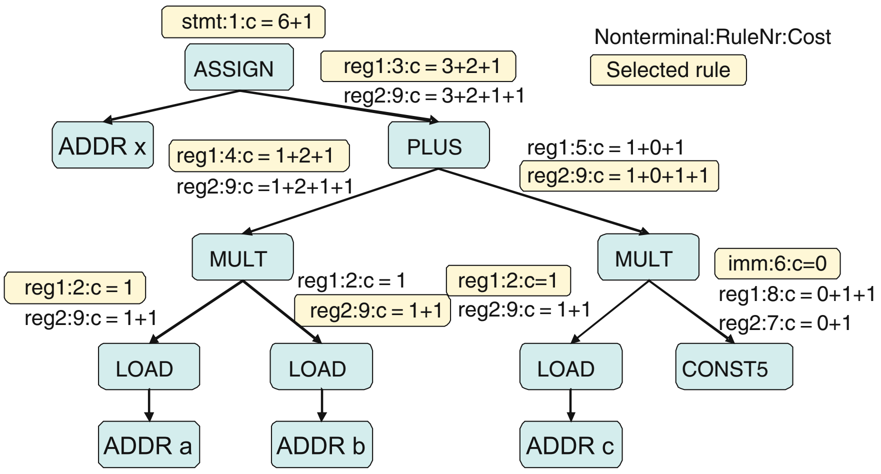


Fig. 3.5 Tree-pattern-matching example for the statement $x = a * b + c * 5$

Table 3.1 Tree grammar specification

Rule No.	Nonterminal		Tree pattern	Instruction	Costs
1	stmt	→	ASSIGN(ADDR,reg1)	STORE dst = src	1
2	reg1	→	LOAD(ADDR)	LOAD dst = src	1
3	reg1	→	PLUS(reg1,reg2)	ADD dest = src1, src2	1
4	reg1	→	MULT(reg1,reg2)	MUL dest = src1, src2	1
5	reg1	→	MULT(reg1,imm)	MULI dest = src1, src2	1
6	imm	→	CONST		0
7	reg2	→	imm	LOADI dst = src	1
8	reg1	→	reg2	MOVE21 dst = src	1
9	reg2	→	reg1	MOVE12 dst = src	1

Tree pattern matching finds an optimal set of instructions for a single DFT at linear time in the number of DFT nodes. Furthermore, a number of tools are available that can generate tree pattern matchers from a target-specific tree grammar specification. Examples of such so-called *code generator generators* are `BEG` [108], `burg` [52], `iburg` [51], `lburg` (code selector of the *lcc* compiler [50]), `OLIVE` (code selector of the SPAM compiler [247]), and `twig` [2].

In case the IR takes the form of a *direct acyclic graph* (DAG) (due to common subexpressions), it is usually split into a forest of DFTs based on heuristics. While this works well for regular architectures, for irregular architectures or architectures with special custom instructions this may result in suboptimal code quality. Typically, such architectures comprise instructions that exceed the scope of a single DFT. Therefore, different approaches to DAG-based code selection have been developed such as in [159, 234]. Unfortunately, optimal code selection on DAGs is known to be NP-complete. Thus, many approaches employ heuristics, impose several restrictions, or are mostly limited to small problem sizes in order to cope with the excessive runtime requirements. The work in [111] presents a code generator generator, called `cburg`, for a DAG-based code selector.

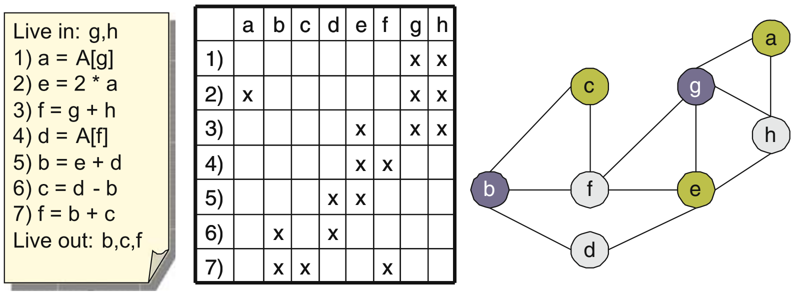
3.3.3 Register Allocation

The task of the register allocator is to assign variables and temporary values to a limited set of physical machine registers. Registers are very expensive with regard to area and power consumption. Therefore, many processor architectures implement only a small register file. Due to the increasing gap between the processor's speed and the memory access time, the register allocation must keep the largest possible number of variables and temporaries in registers to achieve good code quality. In the following, the most important definitions and concepts of register allocation are summarized.

Definition 3.8 (Life range) *A virtual register r is live at a program point p, if there exist a path in the control flow graph starting from p to an use of r on which r is not defined. Otherwise r is dead at p.*

Definition 3.9 (Interference graph) *Let V denote a set of virtual registers. An undirected graph $G = (V, E)$ is called interference graph if for all $v, w \in V$, the following condition holds: v and w have intersecting life ranges.*

State-of-the-art techniques for register allocation are based on a graph-coloring paradigm. The notion of abstracting storage allocation problems to graph coloring dates from the early 1960s [242]. More specifically, the problem of register allocation is translated into the problem of coloring the *interference graph* by K colors, where K denotes the number of available physical registers. The basic idea of the graph-coloring method is based on the following observation: If G contains a node n with degree d (i.e., the number of edges connected to n) with $d < K$, a color k from the set of K colors can be assigned to n that is different from the colors of all its neighbors. The node n is removed from G and a new graph $G' = G - n$ is obtained that, consequently, contains one node and several edges fewer and the algorithm proceeds with the next node. This approach leads to a step-by-step reduction of the interference graph. Since graph coloring is NP-complete, heuristics are employed to search for a K-coloring. If such a coloring cannot be found for the graph, some values are spilled, i.e., values are kept in memory rather than in registers, which results in a new interference graph. This step is repeated until a K-colorable interference graph is found. An example is given in Fig. 3.6.



```
Live in: g,h
1) a = A[g]
2) e = 2 * a
3) f = g + h
4) d = A[f]
5) b = e + d
6) c = d - b
7) f = b + c
Live out: b,c,f
```

	a	b	c	d	e	f	g	h
1)							x	x
2)	x						x	x
3)					x		x	x
4)					x	x		
5)				x	x			
6)		x		x				
7)		x	x			x		

Fig. 3.6 Code example, life ranges, interference graph, and its coloring ($K = 3$)

The first implementation of a graph-coloring register allocator was performed by Chaitin et al. [93, 94]. Later, a priority-based scheme for allocation using graph coloring has been described in [80, 81]. Almost all subsequent work is based on these approaches.

The register allocation algorithms can be further subdivided according to their scope. *Local* register allocation, such as in [80, 94], works only on a single basic block at a time. In contrast, *global* register allocation algorithms exceed the basic block boundaries and take the control-flow structure of the program into account, e.g., an entire procedure or even a collection of procedures. Since the latter is able to take execution frequencies of loop bodies, life ranges over basic block boundaries, and calling conventions into account, a better cost analysis can be performed to improve the spill heuristics. Therefore, many register allocators today are global register allocators. Examples for graph-coloring-based global allocators are in [81, 199].

Of course, not all global allocation methods are based on graph coloring. Examples for different approaches include the bin-packing algorithm [198] and the probablistic register allocation given in [255]. Although graph-coloring allocators can be implemented efficiently, they have a quadratic runtime complexity. This makes them impractical whenever the compile time is a major concern like in *dynamic compilation environments* or *just-in-time (JIT) compilers*. For this domain, an allocator with linear runtime and acceptable code quality, called *linear scan allocator*, has been proposed [174]. The linear scan algorithm consists of the following four steps:

1. Order all instructions linearly.
2. Calculate the set of live intervals.

3. Allocate a register to each interval (or spill the corresponding temporary).
4. Rewrite the code with the calculated allocation.

The linear scan algorithm relies on a linear approximation of the instructions in order to determine simultaneously alive intervals. This order influences the extent and accuracy of live intervals, and hence the quality of the register allocation. As investigated in [151], a *depth-first* ordering is the optimal one.

After instruction ordering is performed, the live intervals are computed. For temporaries outside of a loop, the interval starts at the first definition of the register and ends at its last use. For temporaries alive inside a loop, the interval must be extended to the end of the loop. Given live variable information (e.g., via data-flow analysis [1]), live intervals can be computed easily with one pass through the ordered instruction list. Intervals interfere if they overlap. The number of overlapping intervals changes only at the start and end points of an interval. The computed live intervals are stored in a list that is ordered in increasing start points to make the allocation more efficient.

As defined in [174], given R available registers and a list of live intervals, the linear scan algorithm must allocate registers to as many intervals as possible, but such that no two overlapping live intervals are allocated to the same register. If $n > R$ live intervals overlap at any point, then at least $n - R$ of them must be spilled. For allocation, the linear scan algorithm maintains a number of sets:

1. The set of already allocated intervals called *Allocated*.
2. The mapping of active intervals to registers stored in the set named *Active*.

The algorithm starts with an empty *Active* set. For each newly processed live interval, the algorithm scans *Active* from the beginning to the end and moves those intervals to *Allocated* whose end points precede the processed interval's start point. Removing an interval from *Active* makes the corresponding register again available for allocation. The processed interval's start point becomes the new start position for the algorithm and gets a physical register assigned that is not used by any interval in *Active*. If all registers are already in use, one interval must be spilled. The spill heuristics selects the interval with the highest end position.

Figure 3.7 depicts an example. The live intervals shown in the middle correspond to the instruction ordering on the left. Suppose the set of allocatable physical registers is `R1`, `R2`, and `R3`. In the first step, the interval `V1` is processed and, since the *Active* list is empty, gets the physical register `R1` assigned. Consequently, `V1` is added to the *Active* list. When `V2` is visited in the next step, `V1` is still live and another register `R2` is assigned to `V2` and added to *Active*. Afterward, interval `V3` is processed and gets the last free physical register `R3` assigned. Since no physical register is available for `V4`, one interval must be spilled. The algorithm selects `V1` for spilling because it has the highest end position and removes it from the *Active* list. The example shows the corresponding state of the intervals and the active list. The final allocation after processing all intervals is depicted on the right.

A retargetable linear scan allocator for the CoSy environment [38] was implemented in [11] and compared to the regular graph-based register allocator. The

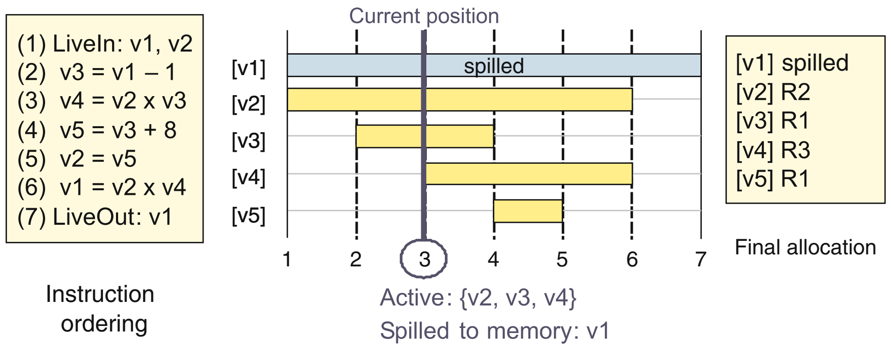


Fig. 3.7 Linear scan allocation example

results show an average speedup of 1.6–7.1 for the register allocation while attaining good code quality (average overhead in cycle count/code size is within 1–3%).

3.3.4 Instruction Scheduling

Most contemporary processors use pipelining to partially overlap the execution of instructions or even *Instruction-Level Parallelism* (ILP) to execute several instructions in parallel such as *Very Long Instruction Word* (VLIW) machines for instance. Generally, scheduling is the process of reordering instructions in such a way that the maximum amount of parallelism among instructions is exploited. Similar to register allocation, *local* schedulers work at the basic block level whereas *global* scheduler deal with complete functions.

The scheduling process is limited by two major constraints [214]: first, *data hazards* or *control hazards* causing dependencies between instructions that force a sequential ordering and second resource limitations, i.e., *structural hazards*, that force serialization of instructions requiring the same resource. A *dependency graph* that captures these constraints constitutes the input for most scheduling techniques.

Definition 3.10 (Dependency graph) *A dependency graph (DG) is an edge-weighted directed acyclic graph $G = (V, E, type, delay)$, where each node v in V represents a schedulable instruction. The resource allocation of each instruction is given by its reservation table $r(v)$. An edge $e = (v_i, v_j) \in E \subseteq V \times V$ indicates a dependency between v_i and v_j and it is weighted with the minimum delay cycles given by $delay(e)$ the instruction v_j can be started after v_i.*

The dependencies between instruction v_i and v_j, $i < j$, can be further categorized into the following kinds [135]:

Data dependence: v_i writes to a resource read by v_j. Consequently, v_i must be scheduled before v_j. This dependency is also referred to as *read after write* (RAW) dependency and is also the most common type.

Anti-dependence: v_j reads a storage location written by v_k with $k \neq i$ that is overwritten by v_i. Thus, in a correct schedule, v_j reads the value defined by v_k before v_i overwrites it. This is also known as *write after read* (WAR) dependence. Since this is often the result of instructions that write results late in the pipeline while others read the result early in the pipeline, the associated delay is usually negative.

Output dependence: v_i and v_j write to the same storage location. A valid schedule must perform the writes in their original order, i.e., the storage location contains the result of v_j after executing both instructions. This dependency is also denoted as *write after write* (WAW) dependency.

Control dependence: Determines the ordering of v_j with respect to a branch instruction v_i so that v_i is executed in correct program order and only if it should be. Thus v_j is not executed until the branch destination is known. Generally, this kind of dependency can also be seen as a data dependency on the *program counter* (PC) resource.

Note that the *read after read* (RAR) dependency is not considered a data hazard.

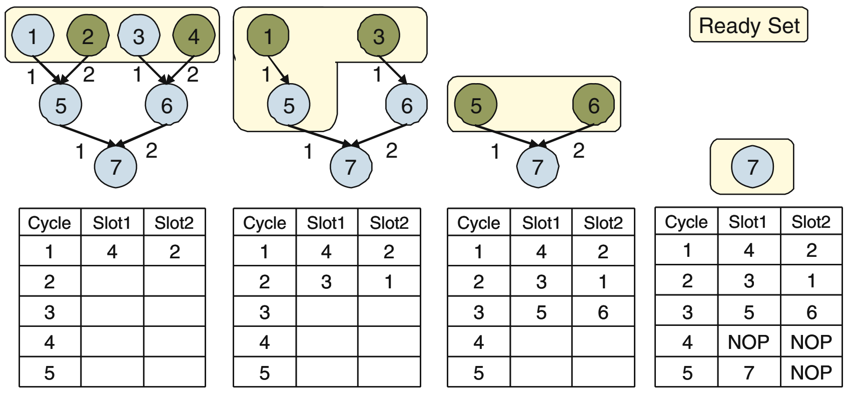
Since an instruction v_i may take several cycles until its result becomes available to v_j, it is the scheduler's task to fill these so-called *delay slots* with useful instructions instead of no-operations (NOPs). Given a dependency graph, a valid schedule is obtained with a mapping function S that assigns each node $v \in V$ a start cycle number c, $c \in \mathbb{N}$, such that

1. $S(v_i) + delay(v_i) < S(v_j)$ to guarantee that no dependencies are violated.
2. $r(v_i) \cap r(v_j) \neq \emptyset$ to avoid structural hazards.

The goal is now to find a schedule S_{opt} that needs the fewest number of cycles to execute. Let I denote the set of available machine instructions, then the length $L(S)$ of a schedule S can be described as follows:

$$L(S) = \max(S(v) + \max(delay(v, w))), \quad \forall v \in V, w \in I \tag{3.6}$$

The worst-case delay makes sure that the results are definitely available before instructions of potential successor basic blocks are executed. Unfortunately, computing the optimal schedule S_{opt} is an NP-complete problem. Several heuristics are in use for scheduling whereas *list scheduling* [68] is the most common approach. This algorithm for local scheduling keeps a *ready set* that contains all instructions v which predecessors in the dependency graph have already been scheduled. The list scheduler selects an instruction from the ready set and inserts it into the schedule S. Afterward, the ready set is updated accordingly and the scheduler proceeds with the next instruction from the ready set. Different heuristics have been proposed to pick a node from the ready set since this strongly influences the length of the schedule. For instance, one heuristic picks the instruction on the current *critical path*. This path represents the theoretical optimal schedule length. Figure 3.8 shows an example using this heuristic.



Cycle	Slot1	Slot2
1	4	2
2		
3		
4		
5		

Cycle	Slot1	Slot2
1	4	2
2	3	1
3		
4		
5		

Cycle	Slot1	Slot2
1	4	2
2	3	1
3	5	6
4		
5		

Cycle	Slot1	Slot2
1	4	2
2	3	1
3	5	6
4	NOP	NOP
5	7	NOP

Fig. 3.8 List-scheduling example; note that two instructions are scheduled in each step

List scheduling has a worst-case complexity that is quadratic in the number of instructions to schedule. However, list scheduling is conceptually not effective in handling negative latencies (in case of anti-dependencies) and filling delay slots. A solution to this problem are *backtracking schedulers* [232]. Such schedulers can revert previous scheduling decisions to schedule the current instruction earlier if this is likely to be more advantageous.

The amount of parallelism that can be exploited within a single basic block is quite limited since it contains only a few instructions on average. This is especially a problem for loop bodies that constitute typically the hot-spots of a program. One way to increase the number of instructions in loop bodies is *loop unrolling*, i.e., duplicating the loop body while reducing the number of required iterations. Another possibility is a scheduling technique especially for loops, called *modulo scheduling* [47]. It is an algorithm for *software pipelining* loops [173], i.e., the overlapping execution of several iterations.

An algorithm for global scheduling is *trace scheduling* [130]. The basic idea is to jointly schedule instructions of frequently executed and consecutive basic blocks. The execution frequency of basic blocks has to be obtained by profiling. Such a sequence of basic blocks is called a *trace* and is considered as a single, large basic block. In this way, the opportunities for ILP exploitation are increased. However, since the basic block boundaries are neglected, undesired side effects may arise. In order to fix this, compensation code has to be inserted. Of course, this results in a significant code-size increase that constitutes the major drawback of this approach.

3.3.5 Code Emitter

The code emitter is the final phase of the compiler backend. It is responsible to write the result of the previous phases into a syntactically correct assembly program, typically in an output file. The data structure of the emitter is an emission table. Each row, sorted in increasing order, represents a clock cycle and each column an instruction. The code emitter first fills the emission table using the clock cycle information determined by the scheduler. Thus, each row represents the instructions that are executed together. Afterward, the table is dumped row by row, where empty cells are replaced by NOP instructions. While this is straightforward for single issue architectures, i.e., the table has only one column, constructing instructions for ILP architectures is sometimes more difficult. Such architectures typically impose constraints on how the instructions can be combined to build a valid instruction word. Therefore, a *packer* is incorporated in the emitter that composes syntactically correct assembly instructions for a given row. The final executable is then build from the assembly file using an assembler and linker. Both are usually separate tools that run after the compiler.

3.4 Retargetable Compilers

The embedded domain is characterized by a large variety of processor designs. Obviously, designing a new compiler for every single one of them is too costly. Additionally, developing a compiler is a time-consuming task, and hence, it may become available too late to be really useful for the architecture exploration phase. In many cases, this results in a compiler architecture mismatch that makes it quite difficult for Compiler Designers to ensure good code quality. This has led to the development of retargetable compilers. Such compilers are capable of generating code for different hardware architectures with few modifications of its source code. Such compilers take a formal description, e.g., specified in an ADL, of the target architecture as input and adapt themselves to generate code for the given target. The retargetability support mostly needs to be provided for code selector, scheduler, and register allocator, i.e., the compiler backend (Fig. 3.9).

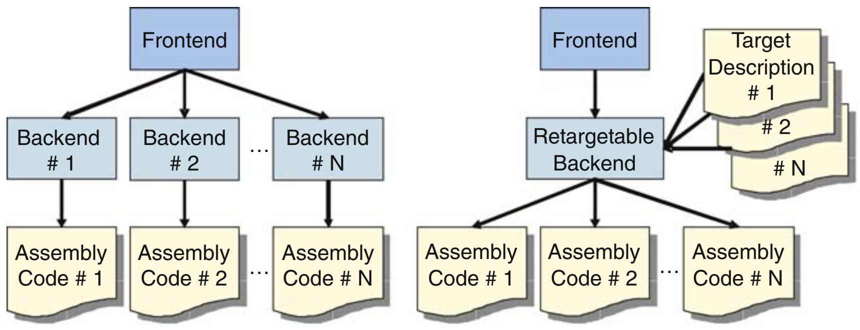


Fig. 3.9 Non-retargetable vs. retargetable compiler flow

Different degrees of retargetability exists to achieve this goal. According to the classification in [219], compilers can be assigned to one of the following classes:

Parameterizable: Such compilers can only be retargeted to a specific class of processors sharing the same basic structure. The compiler source code is largely fixed. The machine description only consists of numerical parameters such as register file sizes, word lengths, the number of functional units, or different instruction latencies.

User retargetable: An external machine description given in a dedicated language contains the retargeting information. All information required for code generation is automatically derived from this description. The specification does not require in-depth compiler knowledge, and hence can be performed by an experienced user.

Developer retargetable: Retargeting is also based on an external target description. However, the specification requires extensive compiler expertise usually possessed only by very experienced users or Compiler Designers.

A retargetable compiler has to be as target-independent as possible in order to be applicable for a wide variety of processor types. As a result, such compilers can only make few assumptions about the target machine, i.e., less target-specific hardware features can be exploited to produce efficient code. Hence, one of the major challenges in the design of retargetable compiler is to provide good optimizations for a wide variety of target architectures. Therefore, many retargetable compilers are limited to a certain processor class in order to ensure good code quality. New retargetable optimization techniques offer a solution to extend the range of target processor. This is further discussed in Chapters 8 and 9. Typically, retargetable compilers are limited to one of the following processor classes:

General purpose processors (GPPs): GPPs are characterized by an universal instruction-set architecture that provides a high degree of flexibility. As a result, they achieve good performance for a wide variety of applications. Unfortunately, this comes usually at the expense of a higher power consumption that makes them pretty much unusable for the embedded domain. Instead, such processors are widespread in desktop or portable PCs. Prominent examples for this class are MIPS [179], ARM [41], and the well-known Intel x86 architectures [122].

Very long instruction word processors (VLIW): This architecture is designed to exploit ILP that comes along with very high performance. Several functional units can be executed in parallel, whereas each unit is related to a specific field in the instruction word. Since such processors do not feature dedicated scheduling hardware such as superscalar architectures, the compiler is responsible for exploiting the ILP that might be present in the given applications. Representative examples of this processor class include the TriMedia and Nexperia architectures [190], the Embedded Vector Processor [152], and the ST200 [84].

Digital signal processors (DSPs): DSPs have been specifically designed for signal-processing applications. Consequently, their instruction-set supports dedicated instructions for the efficient execution of common signal-processing computations, such as fast Fourier transformation (FFT) or digital filtering. Additionally, such processors usually feature hardware multipliers, address generation units (AGUs), and zero overhead loops. Typical DSP examples are the TI C5x and C6x [259], the ADSP 2101 [42], and the MagicDSP [70].

Micro-controllers: Micro-controllers operate at clock speeds of as low as a few MHz and are very area efficient. The processor core implements a complex instruction-set computer (CISC) architecture. The chip typically integrates additional elements such as read-only memory (ROM) and random access memory (RAM), erasable programmable ROM (EPROM) for permanent data storage, peripheral devices, and input/output (I/O) interfaces. They are frequently used in automatically controlled products and devices, such as engine control systems, remote controls, office machines, and appliances. Examples for this kind of architecture are the Motorola 6502 [181] and the Intel 8052 [122].

Application specific instruction-set processors (ASIPs): ASIPs show highly optimized instruction-sets and architectures, tailored for dedicated application domains such as image processing or network traffic management. In this way, they achieve a good compromise between flexibility and efficiency. Examples of this kind are ICORE [251], SODA [281], a channel decoder architecture for third-generation mobile wireless terminals [78], and an ASIP for Internet Protocol Security (IPSec) encryption [109].

Some prominent retargetable compilers primarily for GPPs are *gcc* [87] and *lcc* [50]. Trimaran [263] and IMPACT [57] are examples for retargetable compilers for VLIW architectures. Other examples include CoSy [38], LANCE [222], SPAM [247], and SUIF [249]. Some of them constitute a key component of the ASIP design environments discussed in Chapter 4. A comprehensive survey of retargetable compilers can be found in [224].

3.5 Synopsis

- Compilers can be coarsely separated into a frontend and a target-specific backend (code selector, scheduler, register allocator).
- Retargetable compilers can be quickly adapted to varying processor configurations.
- Such compilers are capable of generating the backend components from a formalized processor description (e.g., an ADL model).