

# 2 Foundations & Terminology

This thesis builds on a few key developments in the field of compiler construction that we briefly introduce in this chapter. More detailed explanations are given, for example, by Aho et al. [2006] and Kennedy & Allen [2002].

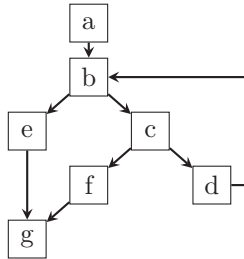
## 2.1 Basic Block

A *basic block* consists of a list of instructions that have to be executed in order. This implies that any operation that can influence which instruction is executed next, i.e., any branch instruction, may only occur at the end of a block.

## 2.2 Control Flow Graph (CFG)

A CFG is a directed, possibly cyclic graph with a dedicated entry node. It represents the structure of a function by expressing instructions or basic blocks as nodes and their successor relations as edges. We consider CFGs with basic blocks as nodes. If the function code at one point allows multiple different code paths to be executed next, the corresponding block in the CFG has multiple outgoing edges. This is the case for explicit control flow constructs like `if` statements or loops, but can also occur in other circumstances, e.g. for operations that can throw an exception. If previously disjoint paths merge again, the corresponding block has multiple incoming edges, e.g. behind a source-level `if` statement. Figure 2.1 shows the CFG that corresponds to the `Mandelbrot` example in the introduction.

**Critical Edges.** An edge of a CFG is said to be *critical* if its source block has multiple outgoing edges and its target block has multiple incoming edges. For example, the edge  $a \rightarrow c$  in Figure 2.2 is critical. Such edges are usually not desired, and can be easily removed by splitting the edge and placing an additional block at the split point. All algorithms presented in this thesis can handle critical edges. The Partial CFG Linearization phase (Section 6.3) in some cases performs implicit breaking of critical edges.



**Figure 2.1:** The CFG that corresponds to the **Mandelbrot** function in Listing 1.1. Special loop blocks according to Section 2.4: Block *a* is the entry block and at the same time the preheader of the loop that consists of blocks *b*, *c*, and *d*. Block *b* is the header of this loop, *d* is a loop latch. The blocks *b* and *c* are loop exiting blocks, *e* and *f* the corresponding exit blocks. Block *g* is the final block that holds a **return** statement.

## 2.3 Dominance and Postdominance

A basic block *a* *dominates* another block *b* if *a* is part of every possible path from the entry block of the function to *b* [Lowry & Medlock 1969]. In particular, the entry block of a function dominates all reachable blocks. Also, according to the definition in Section 2.4, a header of a loop dominates all blocks in the loop. A basic block *a* *strictly dominates* a block *b* if *a* dominates *b* and *a* and *b* are distinct blocks. A basic block *a* is the unique *immediate dominator* of a block *b* if *a* strictly dominates *b* and does not strictly dominate any other strict dominator of *b*.

The immediate dominance relation of basic blocks induces a tree structure, the *dominator tree*. The *dominance frontier* of a block *a* is the set of nodes *B* of which a predecessor of each block in *B* is dominated by *a* but each block in *B* is not strictly dominated by *a*. The *iterated dominance frontier* of blocks  $a_1, \dots, a_n$  is the limit of the increasing sequence of sets of nodes *D*, where *D* is initialized with the dominance frontiers of  $a_1, \dots, a_n$ , and each iteration step increases the set with the dominance frontiers of all nodes of *D* [Cytron et al. 1991].

A basic block *b* *postdominates* another block *a* if *b* is part of every possible path from *a* to the function exit block.<sup>1</sup> As an example, if there is exactly

<sup>1</sup>If the function has multiple exit blocks and/or non-terminating loops, we assume a *virtual*, unique exit block that all these blocks jump to instead of returning.

one edge in a loop that jumps back to the header, the source block of that edge is a postdominator of all blocks in the loop (see Section 2.4). The terms *strict postdomination*, *immediate postdominator*, *postdominator tree*, *postdominance frontier*, and *iterated postdominance frontier* are defined analogously.

Various algorithmic approaches exist to efficiently compute dominance, dominance frontiers, and dominator trees [Cytron et al. 1991, Lengauer & Tarjan 1979, Ramalingam 2002].

Note that these definitions are straightforward to extend from basic blocks to SSA values: If two values are defined in the same basic block, the upper one in the list of instructions strictly dominates the lower one. Otherwise, the dominator relation of their parent blocks is the dominator relation of the values.

## 2.4 Loops

A loop is a strongly connected component of blocks in a CFG. The blocks that are enclosed in such a region are called the *body* of the loop. In addition, we use the following definitions [Ramalingam 2002]:

- A *header* is a block of a loop that is not dominated by any other block of the loop.
- A *preheader* is a block outside the loop that has an unconditional branch to a loop header.
- A *back edge* is an edge that goes from a block of a loop to a loop header.
- A *latch* is a block of a loop that has a back edge as one of its outgoing edges.
- An *exiting* block of a loop is a block inside the loop that has an outgoing edge to a block that is outside the loop.
- An *exit* block of a loop is a block outside the loop that has an incoming edge from a block that is inside the loop.

Note that, by definition, an exiting block has to end with a conditional branch, since otherwise it would not be part of the loop itself. Figure 2.1 exemplifies these terms at the example of the **Mandelbrot** CFG.

If not mentioned otherwise, we consider only *reducible* loops, which are the vast majority of loops found in real-world programs. In a reducible loop, the header dominates (Section 2.3) its source, the latch [Hecht & Ullman 1972]. This implies that a reducible loop may not have more than one header, i.e., it is a loop with a single entry. The single header then also dominates

all blocks of the loop. Note that, if required, irreducible control flow can be transformed into reducible control flow using node splitting [Janssen & Corporaal 1997]. However, this can result in an exponential blowup of the code size [Carter et al. 2003].

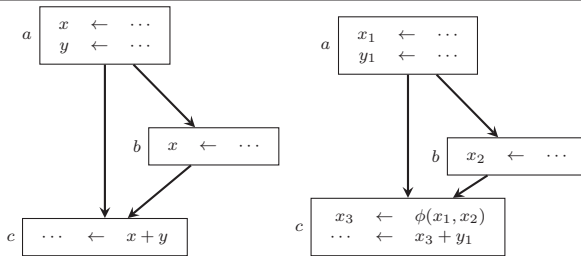
In addition, we consider all loops to be *simplified*, i.e., they have exactly one preheader and one latch, and all exits have a unique successor block that has no other incoming edges. This does not impose any restrictions on the source code. This simplified definition allows to determine a tree-shaped hierarchy of *nested* loops for every CFG:<sup>2</sup> A loop  $L_n$  is *nested* in another loop  $L_p$  if the header of  $L_n$  is part of the loop body of  $L_p$ . In such a case, we call  $L_p$  the *outer loop* or *parent loop*, and  $L_n$  the *inner loop* or *child loop*. We also say that  $L_n$  is at a *nesting level* that is one level *deeper* than  $L_p$ . Multiple loops may be nested in one loop, but a loop may not be nested in multiple loops. Note that this definition allows exit edges to leave multiple loops at once, going up multiple nesting levels. However, an entry edge may not enter a loop of a nesting level deeper than the one below its own nesting level.

## 2.5 Static Single Assignment (SSA) Form

Static Single Assignment form [Alpern et al. 1988], short *SSA* form, is a program representation in which each variable has exactly one static definition. To transform a given program into SSA form, an SSA variable is created for *each* definition of a variable. Figure 2.2 shows an example for the transformation of a normal CFG into a CFG in SSA form. To represent situations where multiple definitions of the same original variable can reach a use, so-called  $\phi$ -functions have to be inserted at dominance frontiers [Cytron et al. 1991]. A  $\phi$ -function has an incoming value for every incoming edge. The operation returns the value that corresponds to the edge over which its basic block was entered. This is required to ensure that the original relations between definitions and uses are preserved while maintaining the SSA property that every use has exactly one definition.

---

<sup>2</sup>Note that this definition requires the detection of nested SCCs, which can be achieved by ignoring back edges.



**Figure 2.2:** A CFG and its SSA counterpart

### 2.5.1 LCSSA Form (LCSSA)

We consider all CFGs to be in LCSSA form, an extension to SSA form that was first introduced by Zdenek Dvorak<sup>3</sup> in the GCC compiler.<sup>4</sup> LCSSA guarantees that for a value that is defined in a loop, every use that is outside the loop is a  $\phi$ -function in the corresponding loop exit block. This simplifies handling of loop result values during the different phases of the WFV algorithm.

## 2.6 Control Dependence

A statement  $y$  is said to be *control dependent* on another statement  $x$  if (1) there exists a nontrivial path from  $x$  to  $y$  such that every statement  $z \neq x$  in the path is postdominated by  $y$ , and (2)  $x$  is not postdominated by  $y$  [Kennedy & Allen 2002].

This definition implies that there can be no control dependencies within a basic block. Thus, it makes sense to talk about control dependence relations of basic blocks. For example, in Figure 2.1, blocks  $e$  and  $c$  are control dependent on  $b$ , and blocks  $b$ ,  $f$ , and  $d$  are control dependent on  $c$ . Note that blocks can have multiple control dependencies, e.g. in nested conditionals.

## 2.7 Live Values

The classic definition of a *live variable* in a non-SSA context is the following [Kennedy & Allen 2002]: A variable  $x$  is said to be *live* at a given point  $s$

<sup>3</sup><http://gcc.gnu.org/ml/gcc-patches/2004-03/msg02212.html>

<sup>4</sup>[gcc.gnu.org](http://gcc.gnu.org)

in a program if there is a control flow path from  $s$  to a use of  $x$  that contains no definition of  $x$  prior to the use. Most importantly, this includes program points where the variable is not yet defined. In our context, we are only interested in the live values that are already defined at program point  $s$ : An SSA value  $v$  is said to be *live* at a given point  $s$  if there is a control flow path from  $v$  to  $s$  and there is a control flow path from  $s$  to a use of  $v$  that contains no definition of  $v$ . For example, in the CFG on the right of Figure 2.2,  $y_1$  is live in block  $b$ .

## 2.8 Register Pressure

Every processor has a fixed set of registers where values are stored. For convenience, programming languages usually abstract from this by providing an unlimited set of registers to the user. The compiler then has to determine how to best map the used registers to those available in hardware. This is called *register allocation*.

*Register pressure* describes the number of registers that are required for a given program. If there are not enough registers, values have to be *spilled*, i.e., stored to memory, and reloaded later when they are used again. This has implications for the performance of the program: register access is much faster than accessing memory. Thus, it is desirable for a compiler to optimize the usage of registers and minimize the number of spill and reload operations and their impact on performance. Especially in a GPU environment, register pressure is a critical variable: Since the available registers are shared between threads, the number of registers used by a function determines how many threads can execute it in parallel.

In consequence, if a code transformation increases register pressure, this may result in decreased performance even if the resulting code itself is more efficient. As will be discussed in Section 3.3, this is an important issue for SIMD vectorization.

## 2.9 LLVM

The compiler infrastructure *LLVM* [Lattner & Adve 2004] forms the basis for our implementation of the Whole-Function Vectorization algorithm. Amongst other tools and libraries, the framework includes:

- a language and target independent, typed, intermediate representation in the form of a control flow graph in SSA form,

- a front end (Clang) for the C language family (C/C++/ObjC),<sup>5</sup>
- various back ends, e.g. for x86, ARM, or PTX, and
- a just-in-time compiler.

Our implementation works entirely on LLVM’s intermediate representation (IR), which allows to use it independently of the source-language and the target-architecture.

### 2.9.1 Intermediate Representation (IR)

Throughout the next chapters, we show examples in the human-readable representation of the LLVM IR, which we briefly describe in the following.

The LLVM IR is a typed, low-level, assembler-like language. There are neither advanced language features like overloading or inheritance nor control flow statements such as **if-then-else** statements or explicit loops. Instead, the control flow of a function is represented by a CFG whose edges are induced by low-level instructions such as conditional or unconditional branches. The nodes of the CFG are basic blocks with lists of instructions. Each instruction corresponds to exactly one operation and usually also represents an SSA value. The only exception to this are values with return type **void**, such as **branch**, **call**, and **store** operations. Every other value has a name that starts with “%” which is referenced everywhere that value is used. A typical instruction looks like this:

```
%r = fadd <4 x float> %a, %b
```

The example shows the LLVM IR equivalent for the SSE2 vector addition intrinsic **ADDPS** (`_mm_add_ps(__m128 a, __m128 b)` in C/C++). The left-hand side of the expression is the name of the value. On the right hand side, the operation identifier is followed by its operands, each with its type preceding the name. If types are equal, they can be omitted for all operands after the first.

The following code defines a function **foo** with return type **float** and argument types **int** and **float**:

```
define float @foo(int %a, float %b) {
entry:
    %x = fsub float %b, 1.000000e+01
    ret float %x
}
```

The label **entry** marks the only basic block which is followed by the instructions of that block.

---

<sup>5</sup>[clang.llvm.org](http://clang.llvm.org)

## 2.9.2 Data Types

The LLVM IR supports a large set of different data types. Table 2.1 shows examples for the most important types and their typical C/C++ counterparts.

**Table 2.1** Examples for the most important LLVM data types and their C/C++ counterparts. Note that this list only shows *typical* C/C++ types, since types like `int` are machine dependent.

LLVM Type	C/C++ Type	Explanation
<code>i1</code>	<code>bool</code>	truth value ( <code>true</code> (1) or <code>false</code> (0))
<code>i8</code>	<code>char</code>	single character
<code>i32</code>	<code>int</code>	32bit integer value
<code>i64</code>	<code>long</code>	64bit integer value
<code>float</code>	<code>float</code>	32bit floating-point value
<code>type*</code>	<code>type *</code>	pointer to type <code>type</code>
<code>i8*</code>	<code>void *</code>	void pointer
<code>&lt;4 x float&gt;</code>	<code>_m128</code>	vector of 4 32bit floating-point values
<code>&lt;4 x i32&gt;</code>	<code>_m128i</code>	vector of 4 32bit integer values
<code>&lt;8 x float&gt;</code>	<code>_m256</code>	vector of 8 32bit floating-point values
<code>&lt;8 x i32&gt;</code>	<code>_m256i</code>	vector of 8 32bit integer values
<code>&lt;4 x i1&gt;</code>	<code>(_m128)</code>	mask vector
<code>{ types }</code>	<code>struct { types }</code>	structure of types <code>types</code>
<code>[ N × type ]</code>	<code>type t[ N ]</code>	array of size <code>N</code> of type <code>type</code>

## 2.9.3 Important Instructions

Most instructions of the IR are standard instructions that can be found in most assembly languages and need not be described in detail. However, there are a few that require additional explanations:

- *Phi*

The `phi` instruction is the equivalent to the  $\phi$ -function described in Section 2.5. It chooses a value depending on which predecessor block was executed:

```
%r = phi float [ %a, %blockA ], [ %b, %blockB ]
```

The value of `r` is set to `a` if the previously executed block was `blockA` or to `b` if the previously executed block was `blockB`.



- *Select*

The `select` instruction returns either its second or third operand depending on the evaluation of its condition:

```
%r = select i1 %c, float %a, float %b
```

The value of `r` is set to `a` if condition `c` is `true` and to `b` otherwise. If the select statement has operands of vector type, a new vector is created by a *blend* operation that merges the two input vectors on the basis of a per-element evaluation of the condition vector (see Section 6.2). The terms “select” and “blend” are thus used interchangeably for the same operation.

- *GetElementPointer (GEP)*

The `GetElementPointer` instruction returns a pointer to a member of a possibly nested data structure. It receives the data structure and a list of indices as inputs. The indices denote the position of the requested member on each nesting level of the structure. In the following example, the `GEP` instruction (`%r`) creates a pointer to the `float` element of the struct of type `struct.B` that is nested in the struct `%A` and stores 3.14 to that location:

```
%struct.A = type { i8*, i32, %struct.B }
%struct.B = type { i64, float, i32 }
...
%r = getelementptr %struct.A* %A, i32 0, i32 2, i32 1
store float 0x40091EB860000000, float* %r, align 4
```

The first index is required to step through the pointer, the second index references the third element of the struct (which is the nested struct) and the third index references the second element of that nested struct. It is important to note that a `GEP` only performs an address calculation and does not access memory itself.

- *InsertElement / ExtractElement*

The `InsertElement` and `ExtractElement` instructions are required if single elements of a vector have to be accessed:

```
%p2 = insertelement <4 x float> %p, float %elem, i32 1
%res = extractelement <4 x float> %p2, i32 1
```

The first instruction inserts the `float` value `elem` at position 1 into vector `p`, yielding a new vector SSA value. The second instruction extracts the same `float` from that vector `p2`, yielding a scalar value again.

- *Comparison Operations*

The `ICmp` and `FCmp` instructions allow to compare values of the same integer or floating point types and return a truth value. If the values are of vector type, the result of the comparison is stored component-wise as a vector of truth values `<W x i1>`.

- *“All-`false`” Vector Comparison*

In vector programs, one often needs to take a branch if *all* elements of a mask vector (e.g. `<4 x i1>`) are `false`. Since that requires a single truth value to base the decision upon, one cannot employ a vector comparison operation, which returns a vector of truth values. As of LLVM 3.3, the most efficient solution to this is the following IR pattern: Given a vector condition of type `<W x i1>`, sign extend it to `<W x i32>`, bitcast to `i(W*32)`, and compare to zero. For example, when targeting SSE with a target SIMD width of 4, the following IR is recognized by the x86 back end:

```
%ext = sext <4 x i1> %cond to <4 x i32>
%bc  = bitcast <4 x i32> %ext to i128
%test = icmp ne, i128 %bc, 0
br i1 %test, %blockA, %blockB
```

The resulting assembly is an efficient `PTEST` followed by a conditional jump.

In addition, we define two special instructions that do not actually exist in LLVM IR, but help us keep code in listings more compact:

- *Merge*

The `merge` instruction creates a new vector from a set of input values:

```
%v = merge float %a, %b, %c, %d
```

This is equivalent to the following IR:

```
%v0 = insertelement <4 x float> undef, float %a, i32 0
%v1 = insertelement <4 x float> %v0, float %b, i32 1
%v2 = insertelement <4 x float> %v1, float %c, i32 2
%v  = insertelement <4 x float> %v2, float %d, i32 3
```

- *Broadcast*

The `broadcast` instruction creates a new vector of the given type where each element is a copy of the single operand:

```
%v = broadcast float %a to <4 x float>
```

Automatic SIMD Vectorization of SSA-based Control Flow  
Graphs

Karrenberg, R.

2015, XVI, 187 p. 41 illus., 5 illus. in color., Softcover

ISBN: 978-3-658-10112-1