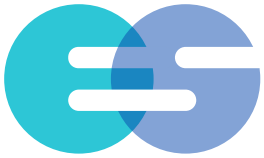


Hamburg University of Technology
Institute of Embedded Systems
Prof. H. Falk

Development of a Code Selector Rule Set for Composed Types for the ARM Architecture inside a WCET-aware Compiler

Bachelor Thesis

Paavo Becker

February 15, 2018

Project Description

Hamburg University of Technology
Institute of Embedded Systems
Prof. H. Falk

Development of a Code Selector Rule Set for Composed Types for the ARM Architecture inside a WCET-aware Compiler

This work's task is the development of a code selector's rule set for composed types within a WCET-aware Compiler for the ARM Architecture with Standard C being the source language. It consists of a complete rule set to handle all sorts of composed types i.e. Arrays, Structs and Union Types as well as combinations of them. Those rules have to handle all cases for the usage of those types, being initialization, component access and passing to and returning from functions. The rules themselves have to serve two purposes, they have to provide an appropriate translation from a high level intermediate to the low level intermediate representation while keeping the semantics the same and they have to specify the cost which will come up with the selection of a rule. The overall second part then consists of an analysis of the produced assembly comparing it to the already established ARM-GCC in terms of code size and runtime.

First Examiner	: Prof. H. Falk
Second Examiner	: M. Sc. Dominic Oehlert
Advisor	: M. Sc. Dominic Oehlert
Due Date	: 19.02.2018

Declaration

This project is the result of my own work, except where explicit reference is made to the work of others, and has not been submitted for another qualification to this or any other university.

(Paavo Becker)
Hamburg, February 15, 2018

Abbreviations

ACET	Average-case execution time
ALU	Arithmetic Logic Unit
BB	Basic block
CPSR	Current programme status register
FP	Frame pointer
HLIR	High-level intermediate representation
IP	Instruction pointer
IR	Intermediate representation
LLIR	Low-level intermediate representation
LR	Link register
PC	Programme counter
RISC	Reduced Instruction Set Computer
SP	Stack pointer
STMT	Statement
TPM	Tree-pattern matcher
WCC	WCET-aware C Compiler
WCET	Worst-case execution time

Contents

Abstract

In the focus of Worst-Case-Execution-Time aware compilation for safety critical systems it is the goal to bring the WCC's power to the ARM platform. This enforces the creation of ARM specific parts within the WCC, since some parts are dedicated to particular TriCore properties and can not be used to model the ARM's behaviour. The aim of this thesis focuses on the extension of the ARM-WCC's code selector to be able to process composed types such as arrays, structures and unions in ANSI-C code as well as a comparison to the ARM-GCC generated assembly code. The first step consists of extending the code selector's rule set by additional rules with appropriate cost determination and code generation for each rule. The rule set covers the general cases of initialization, read and write access and passing of those types to and from functions. The second step then consists of a comparison of the generated low level intermediate representation assembly code to the ARM-GCC output resulting from in advance created ANSI-C code test files in terms of code size and runtime.

Chapter 1

Introduction

With the twenty-first century our everyday life began to heavily rely on electronic devices like notebooks, smartphones or oral assistants like Siri or Alexa. But there are a lot more devices out there which are invisible to their users. They are small computers embedded into automotive, aircraft, medical devices or into household tools like a washing machine or a dishwasher, which yields their name: Embedded Systems. Those computers usually collect an ongoing stream of data from their real world surroundings via different sensors. This data is then processed to make different decisions. An airbag inflation system for example has timing constrains, which have to be met during the decision whether a car crash occurred or not. Those systems are divided into two classes hard real-time and soft real-time systems. [1] Both systems have to meet timing constraints but soft real-time systems are those ones where nobody gets physically harmed if the timing contraints can not be met. Whereas hard real-time systems must not exceed their timing deadline during execution to prevent individuals from being harmed, like an airbag inflation system, which has to decide and inflate within a hard deadline.

The software on those embedded systems is usually developed using a high-level programming languages due to the convenience of having different abstraction layers and due to portability issues. The compiler which then translates the high-level source code to the binary executable is then responsible to generate the correct code for different embedded system platforms. Those compilers generally have multiple optimizing features, where optimisations for code size and runtime are potential candidates for embedded systems due to their limited memory size and processing power. But those optimisations don't come up with the worst-case execution time (WCET), which is the execution time needed to execute the longest path through the programme. Therefore, the most important characterization of a hard real-time system, which has to meet a system's safety critical requirement (timing) in

order to work within the system's specification, is missing.

To determine the WCET during compilation a WCET-aware Compiler is necessary. The WCET-aware C Compiler (WCC) [2] is such a compiler framework, which includes a timing analyser to determine and to optimise the WCET. The current WCC version supports the Infineon TC1796 and TC1797 platform. To bring the benefits of the WCC to a larger amount of devices, the support for the ARM7 platform is currently under development. For this goal, the WCC can be divided into two parts, the target-platform independent front-end and the target-platform depended back-end. Therefore only the back-end has to be customized in order to support a new target-platform. The back-end is responsible for the code-selection process done by the code-selection framework developed by Bouraffa [3], which is the actual process where, generally spoken, the source language is translated into the target processor's language. WCC's code selector uses an OLIVE compatible tree-pattern matcher to generate the target's assembly instructions for each C statement. OLIVE's decisions are based on a predefined set of rules, each one consisting of a cost and an action part, where the action part gets executed if its rule is the cheapest matching one. This work's focus is the development of a ruleset which can generate code for all C statements, which contain array, struct or union code. Control-flow, function call and simple arithmetic rules have already been developed in [4].

Chapter 2

The WCET-aware C Compiler Framework

The WCET-aware C Compiler (WCC) uses a common structure where the compiler is divided into a platform independent front-end and a platform dependent back-end. The front-end is responsible for generating a high-level intermediate representation (HLIR) out of the source code where multiple high level optimisations can then be applied. Afterwards a platform specific code selector is responsible for the translation of the HLIR into a low-level intermediate representation (LLIR), which can undergo further low-level optimisations. Those steps are finally followed by the invocation of the code generator which emits valid assembly code from the LLIR. The WCC is a cross-platform C Compiler. It currently supports the Infineon TriCore TC1796 and TC1797 whereas the support for the ARM7 processor is under active development. The following sections briefly introduce the workflow of the WCC Framework. Its structure is depicted in figure 2.1.

2.1 Frond-End ICD-C

The main task of the front-end is to parse C source code and to produce a data structure called high-level intermediate representation (HLIR), which can be used to analyse, optimise and to further process the source code. WCC uses the ICD-C compiler front-end [6], which provides the parser, syntactic and semantic analysers and an HLIR class structure compatible to internally model the source code. It additionally provides various high level optimisations such as constant folding or local common subexpression elimination. Since the ICD-C HLIR is designed to model the C language, its structure is quite similar to the C source language. A simplified class struc-

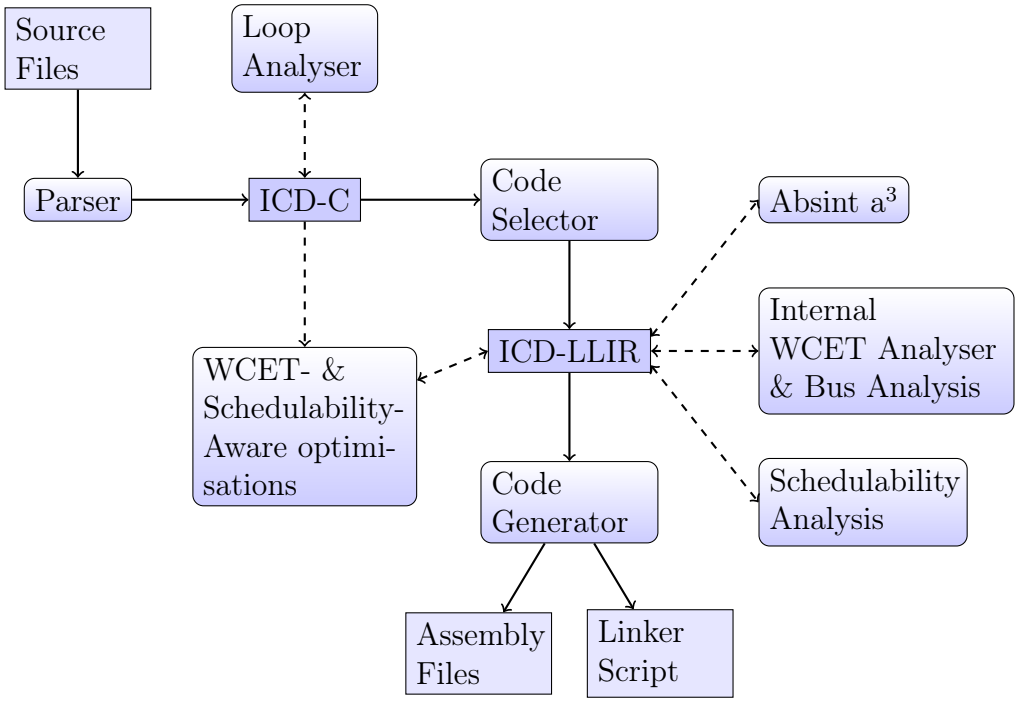


Figure 2.1: Workflow of WCET-aware C compiler WCC [5]

ture of the HLIR is shown in figure 2.2. A source programme is modelled by the IR class Object and each source file of the programme by a unique compilation unit. All units themselves can contain multiple functions which are the top level components of the C language alongside type definitions and global variables. A function comprises basic blocks (BB), where a basic block is defined as following:

Definition 2.1. Basic Block [7] A *basic block* $B = (I_1, ..., I_n)$ is an instruction sequence of maximal length such that

- B is entered only via its very first instruction I_1.
- B is left only via its very last instruction I_n.

C language instructions are represented by the statement class where special statements for conditional branching appear at the end of a BB. Statements again are built from the expression class which allows each statement to be represented in a tree shaped expression form, see section 2.2.1. Each hierarchy of the HLIR down to the BBs has a symbol table associated to it to store symbols corresponding to it like function symbols, new type symbols or variable symbols. The symbol tables are filled throughout the chain of syntactical and semantic analysis of ICD-C after the parsing process.

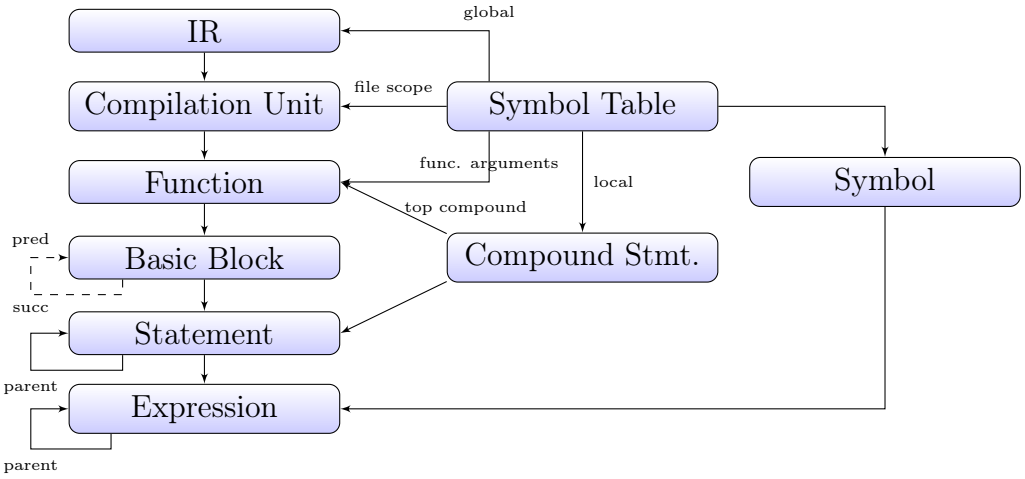


Figure 2.2: Simplified class model of ICD-C IR [5]

2.2 Code Selection

On the road towards valid assembly code construction for a target processor platform the HLIR has to be translated into a low-level intermediate representation (LLIR) which can model the platform specific assembly instructions. The main task is to find and generate a semantically equivalent representation of the code using the instructions available on the target platform. WCC's code generator ICD-CG uses an OLIVE [8] compatible code selector which invokes a tree-pattern matcher for each Statement of the HLIR. With the cost-augmented OLIVE tree grammar, the tree-pattern matcher is able to generate an optimal parse of the HLIR trees. The cost of each rule models the cost of each target instruction used within the rule. Here the cost directly corresponds to an instruction's size.

2.2.1 Tree-Pattern Matching

In this context tree-pattern matching is coupled with the concept of dynamic programming.

> *"Dynamic programming is an approach to decision making in a computational process that depends on the optimality principle that applies to the domain under consideration. The optimality principle asserts that if all subproblems have been solved optimally, then the overall problem can be solved optimally by a particular method of combining the solutions of the subproblems."* [9]

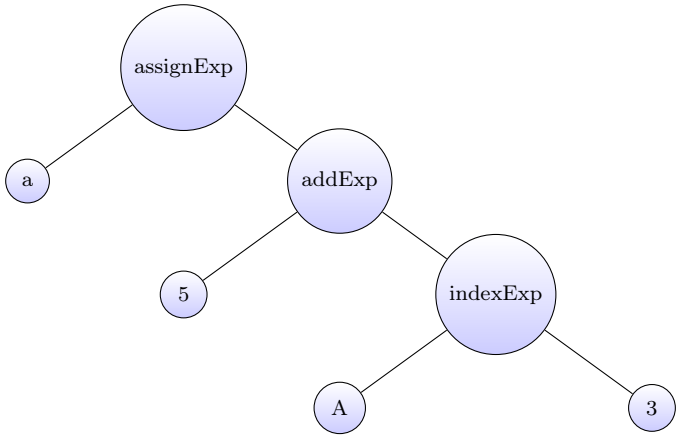


Figure 2.3: Tree-like representation for stmt. in listing 2.1

As noted in section 2.1 each source code statement consists of a tree of expressions. The TPM matches a subtree using a defined rule set, where each rule can match a specific tree. An encountered match is called a cover and reduces the corresponding subtree to a single node within the entire tree. The search for a cover starts at the leaves of the tree. If there are multiple covers available for a subtree, the cheapest one is selected according to the dynamic programming principle. Besides the cost function, each rule holds an action part which contains the LLIR instructions which are needed to model the functionality of the subtree which is covered by this rule. The action part will only be executed if it's rule has finally been selected. Take the following C code statement:

Listing 2.1: compound C Statement

```
a = 5 + A[3];
```

This statement's tree-like representation is depicted in figure 2.3.

To make each rule as reusable as possible during code selection they are usually matching a smallest possible subtree of just three nodes. Remember: If there is a rule which covers a bigger subtree with lower costs, that rule will be preferred. An example would be a multiplication followed by an addition to the result of the multiplication, which can be executed by a single multiply-accumulate instruction if it is supported by the processor platform.

2.2.2 OLIVE

OLIVE [8] is a Code Selector Generator (CGG). It is used to produce a tree-pattern matching based code selector. This code selector is generated by

OLIVE provided with a cost-augmented tree grammar. During the compilation process the code selection framework invokes the generated code selector for all BBs of the HLIR, each of them representing a sequence of expression trees. The grammar consists of tree rewriting rules built from terminal and non-terminal symbols, where the terminals are elementary symbols of the source language. Non-terminals are introduced alongside the grammar and are the symbols which are replaceable to enable a tree rewriting.

The syntax for tree matching grammar of the tree rewriting rules is as follows:

label: pattern {cost} = {action}

- The *label* is a non-terminal and replaceable by *pattern*. The *label* is the root of the expression and the *pattern* corresponds to the children. This is due to the nature of a tree rewriting grammar, where the rewriting process starts at the root of an expression and suitable rules can be applied to rewrite it into an expression tree. The rules however are filled with code so that the *label* represents the result of the rule, since the tree-pattern matcher starts with the leaves of an expression and a matching rule reduces some of them into a single node.

- The *pattern* is the rewriting for the non-terminal *label* and can either consist of a terminal, a non-terminal or a composition of them. Within WCC the terminal symbols are provided by ICD-C to allow an abstraction from the actual symbols in the source language. An example for a rule for the binary plus in C which takes two variables looks like: `reg: tpm_BinaryExpPLUS(reg, reg) {cost} = {action}`. Here the `tpm_BinaryExpPLUS` is the terminal symbol and the `reg`s are non-terminal symbols.

- The cost part can contain any C++ code which is necessary to determine the rule's cost, which is the cost arising from the rule itself and from its children. The cost part will always be executed if a rule matches to determine its cost. The cost part can be used to model each desired form of cost. WCC uses the instruction's size of the platform for its cost calculation.

- Like the cost part, the action part can contain any C++ code to generate the LLIR which is necessary to get the semantically identical code to the HLIR expression the rule is matching. The action part will only be executed when the rule has finally been selected, which means that the selected rule is a member of the tree cover with the least cost.

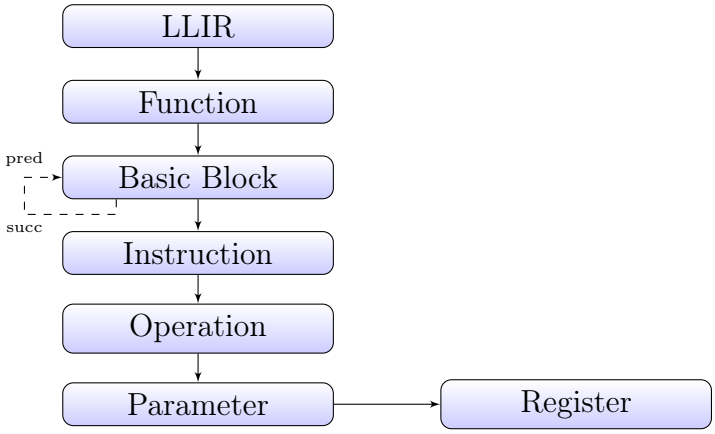


Figure 2.4: Simplified class model of LLIR [5]

2.3 Back-End LLIR

The low-level intermediate representation (LLIR) is another intermediate representation alongside the HLIR to offer a generic data structure for the abstraction of the target processors assembly. The LLIR however can be provided with details of a processor to turn it into a platform specific LLIR. This enables the use of standard assembly-level analysis and optimisations avoiding the need to retarget the needed algorithms for each platform. The LLIR is the result of the code selection process in section 2.2. The LLIR's class structure is a hierarchical design and is depicted in figure 2.4. A LLIR object represents an entire assembly file and results from the code selection of an HLIR compilation unit. Since larger projects usually contain multiple source files a list of LLIR objects is managed. Like the HLIR, the LLIR models functions and basic blocks where multiple LLIR BBs can correspond to a single HLIR BB or vice versa. But the LLIR's basic blocks contain abstractions of the target platform's instructions instead of high level statements. For the ARM7 architecture each instruction holds exactly one operation since the processor is not capable of executing multiple instructions at once.

> *"Each operation is then parametrized with constants, labels, architecture specific operators or registers. The latter are modelled by a separate class since they provide versatile information relevant for the compiler."* [5]

2.3.1 Code-Generation

The LLIR holds a platform-specific description of the programme which is not yet ready to be linked into a binary file. The LLIR provides virtual registers which enables the code selector not having to care about occupied registers or the number of available registers on the target platform. If an empty register is needed during code selection a new virtual one is created. To transform this virtual code into working code using real registers for the target platform the register allocator is invoked. This process does a lifetime analysis on the virtual registers to figure out how long a value has to stay in a register. It then maps the virtual registers to physical registers of the architecture. If a free register is needed but no physical register is available, spill code is inserted which saves the content of a register to the stack to free its register. Before the moved data is needed by another instruction it is loaded back into its former register. Afterwards this code can be assembled and linked to generate the binary executable.

2.3.2 aiT

The worst-case execution time of a programme is the longest execution time a programme can need with respect to all possible inputs.

> *"The path within a program's control flow graph (CFG) which has the maximal WCET is called worst-case execution path (WCEP)"* [2].

Definition 2.2. Control Flow Graph [7] A *Control Flow Graph* $CFG = (C, E, s)$ is a directed graph with

- $V = b_1, ..., b_n$ (set of all basic blocks, see def. 2.1)
- $E = \{(b_i \rightarrow b_j) |$ basic block b_j can be executed immediately after $b_i\}$.
- $s \in V =$ the unique start node of the CFG

Since WCC's aim is to analyse and optimise the WCET of a programme, it includes a static WCET analyser namely aiT. aiT performs a loop bound analysis, cache analysis and a pipeline analysis, which results in the sound WCET for each BB. This process is followed by a path analysis of the programme's CFG to finally find the longest path which results in the WCET for the programme. [10] The analysis requires information about the hardware itself, therefore, aiT is provided with a finished binary and a configuration file which provides details about the target architecture. The results are then passed back into the WCC framework.

Chapter 3

ARM7 Architecture

3.1 General

This chapter provides a brief description of the ARM7 target platform and is not meant to introduce the entire platform, for a detailed explanation see [11]. Originally invented by Acorn, today developed by ARM Ltd. and fully named *Advanced RISC Machines* the ARM architecture is extremely wide-spread across a large variety of devices. ARM Ltd. does not produce their own hardware but sells licenses to many chip manufacturers instead. Therefore, the same basic architecture with some customizations appears in a lot of different devices like the Qualcomm Snapdragon processors [12] in a huge variety of Android devices or the Broadcom chip on a Raspberry Pi [13]. The ARM architecture has emerged from the single processor ARMv1 32-bit architecture with a 4MHz clock to the current ARMv8 64-bit architecture with multiple cores and speeds up to 3GHz.

The 32-bit ARMv4 architecture is the family, where the ARM7 belongs to with its full name being ARM7TDMI. The ARM Architecture Reference Manual [11] states:

> "*The architectural simplicity of ARM processors has traditionally led to very small implementations, and small implementations allow devices with very low power consumption. Implementation size, performance, and very low power consumption remain key attributes in the development of the ARM architecture.*"

The ARM7 is a typical Reduced Instruction Set Computer (RISC) with a large uniform register file, a load/store architecture, where data-processing operation only operate on registers and with fixed-length instruction fields. On top of that, the ARM7 architecture provides control over the Arith-

metic Logic Unit (ALU) in most data-processing instructions as well as auto-increment and auto-decrement addressing modes. It offers Load and Store Multiple instructions to load/store multiple registers at once and finally it allows the conditional execution of almost any instruction. Additionally, the ARM7 architecture provides a reduced 16bit Thumb instruction set, which allows a higher code density, however, not every 32bit instruction has a 16bit version.

3.2 Registers

The ARM7 architecture possesses a total of 37 registers with 31 being general-purpose registers and the other 6 used as status registers. The registers are arranged in partially overlapping banks, with the processor mode controlling which bank is available. Independently from the processor mode 16 general-purpose registers (R0 to R15) are available as well as one status register. The other registers are denoted to fast exception handling and not visible to the user. Some of the visible general-purpose registers are denoted to special roles:

R11 **Frame pointer (FP)** The frame pointer is usually used to point to the first data of the current function on the stack. The FP can be omitted if the stack progress can be modelled entirely inside the compiler. In this case R11 can be used as a usual general-purpose register.

R12 **Instruction pointer (IP)** This register is usually used as a temporary register for data of the other registers of this list.

R13 **Stack pointer (SP)** The stack pointer always points to the last data which has been pushed to the stack.

R14 **Link register (LR)** The link register always contains the return address of a call instruction, to enable the called function to return to its caller.

R15 **Programme counter (PC)** The programme counter always holds the address of the next instruction which has to be executed.

3.3 Status Registers

Besides the general-purpose registers, the *Current Programme Status Register* (CPSR) holds the current state of the processor. Examples are the

Negative, Zero, Carry and Overflow flags which are set during arithmetic operations and can be used for branching instructions or for conditional execution. Other flags indicate if the processor is either in 16bit thumb instruction mode or in normal mode or display the current interrupt mode.

3.4 Exceptions

The ARM7 architecture supports seven different types of exceptions as well as a privileged processing mode for each of them [11]:

1. Reset
2. Attempted execution of an undefined instruction
3. Software interrupt instructions, can be used to make a call to an operating system
4. Prefetch Abort, an instruction fetch memory abort
5. Data Abort, a data access memory abort
6. IRQ, normal interrupt
7. FIQ, fast interrupt

At the occurrence of an exception some of the standard registers are replaced by registers specific to the exception mode. For all exceptions R13 and R14 are replaced by their exception specific *banked* registers. The banked R13 provides a private stack pointer to the exception handler and R14 the return address to the programme where the execution left off before the exception occurred. The CPSR has a banked version for each exception as well, which will be used with the occurrence of their corresponding exceptions.

3.5 ARM Instruction Set

The ARM instruction set provides a lot of different instruction types:

- Branch instructions
- Data-processing instructions
- Status register transfer instructions
- Load and store instructions
- Coprocessor instructions
- Exception-generating instructions

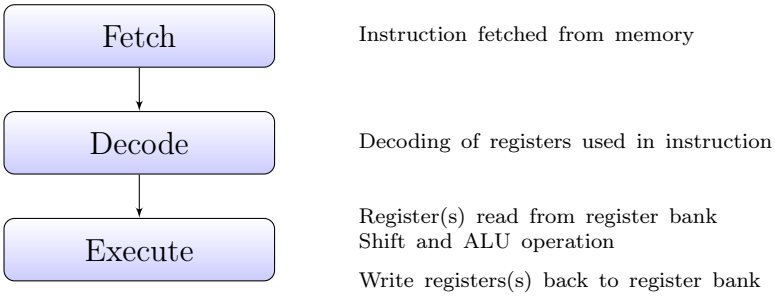


Figure 3.1: Instruction pipeline [14]

The ARM7 processor uses a three-stage pipeline of Instruction Fetch, Decode and Execute shown in figure 3.1. This enables the concurrent decoding of the next instruction and the fetching of the third instruction while an instruction is executed [14]. Since the ARM7 implements the Von Neumann architecture a single data bus is used to transfer both instructions and data. Memory data can only be accessed through load, store and swap instructions, which is characteristic to RISC architectures. [15]

A distinctive character to the ARM architecture is provided by the feature to conditionally execute almost any instruction. The branching decision is based on the CPSR which holds information about a previously executed data processing instruction. A subtraction, for example, can be used to figure out whether two values are equal (the result must be 0). The information is stored into the CPSR, see section 3.3, and can be used to conditionally execute the next instruction. This enables extremely compact code branching, since no extra branching instruction is needed. The conditional execution is accomplished by reserving the highest 4 bit of an instruction for a conditional code. If no code is specified, the *always* condition is used to allow an instruction to be executed, ignoring the result of the previous instruction.

Chapter 4

An ARM Code Selector for Composed Types

Note. In the following *Composed Types* is used to refer to all composed types, namely arrays, structs and unions as a whole, whereas those types are directly named array, struct and union. This is to avoid naming confusion about structs since they are often referred to as composed types.

4.1 Related Work

This thesis' work is related to the development of a code selector rule set for arithmetic and logic operations as well as branching and function call statements presented in [4]. It is placed inside the ARM code selection framework of WCC which has been developed and documented in [3].

4.2 Arrays, Structs and Unions

Arrays, structs and unions are the basic structures to compose complex new types out of the basic types provided by the C language.

Array An array, also referred to as a field, is a series of identical data. During the declaration of an array the type is specified which applies to all the array's cells. Besides the type, the number of elements has to be specified to provide the entire array's size as `number of elements * sizeof(Type)`.

Struct Compared to arrays, structs can contain any configuration of different types. Each struct member is stored one after another, according to

the chronology inside the struct's definition. The struct's size is defined by the sum of the size of all of its members.

Union The union is equally handled like a struct in C but the underlying memory is handled differently. Where a struct stores its members one after another, a union stores its members all into the same place. Therefore, the union's size is equal to the size of its largest member. This enables the access to the same memory location through different types.

4.3 Framework Environment

As already noted in section 2.2, WCC uses a tree-pattern matching approach to transform the HLIR into a LLIR. The code selection process consists of sequentially traversing the HLIRs of all compilation units with the following subtasks: Each function of a compilation unit is traversed and the tree-pattern matcher is invoked for every contained BB separately. Here a HLIR BB is transformed into one or multiple LLIR BB's. Additionally the local stack size of a function is determined during the traversal of the function's BBs to insert appropriate stack adjustment instructions at the beginning of a function. For a detailed explanation refer to [3].

Since this work is about the code selection itself, the necessary information used to generate an ARM specific LLIR is presented in the following.

Processor Definition WCC allows to specify target specific properties, which can be used by the back-end LLIR. For the ARM processor architecture, registers, the conditional codes and the machine instructions itself are specified as well as the cost for every machine instruction. Furthermore one function for each machine instruction is defined, which generates a `LLIR_Operation` to represent a specific machine instruction. Each `LLIR_Operation` is placed into its corresponding `LLIR_Instruction` within another set of functions. `LLIR_Instruction`s are the abstract class to model a machine instruction within the LLIR of WCC, see section 2.3.

Instructionfactory The instructionfactory class is utilized to generate `LLIR_Instruction`s which are used as abstract representations for the ARM's machine instructions inside the LLIR format and to place them into the `LLIR_BB` which is currently generated. Those functions are used throughout the rule set of the code selector to quickly insert a new instruction.

Tree-Pattern Matching Grammar The tree-pattern matching grammar is the core part of the definition of the code selector. The grammar is used to generate a tree-pattern matcher class, which is used within the code selection process.

The following rule definitions show examples for a grammar which covers a simple assignment expression, an index expression and a binary plus expression. The rules are then applied to the expression tree in figure 2.3.

Listing 4.1: Definition of non-terminal `reg`

```
%declare<LLIR_Register*><0> reg<void>;
```

The rules make use of the non-terminal declared in listing 4.1 it can be of any valid C++ type and is introduced with the `%declare` keyword. Such a non-terminal can be parameterised as well. Here the nonterminal reg is of type `LLIR_Register*` and takes no arguments.

Listing 4.2: Example rules to cover tree in figure 2.3

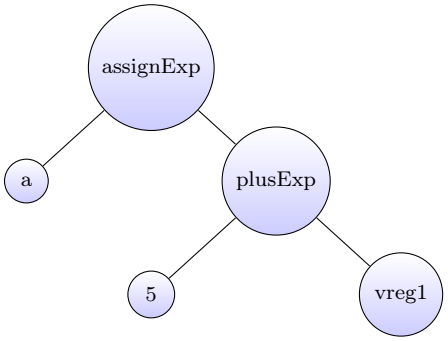
```
1  reg: indexExp(reg, reg)
2  {
3    $cost[0] = $cost[2] + $cost[3];
4    $cost[0] += COST(INS_ADD_32) + COST(INS_LDW_32);
5  }
6  =
7  {
8    LLIR_Register *r = $action[3]();                  // 3
9    LLIR_Register *l = $action[2]();                  // A (vreg0)
10   INSTRUCTIONS->insertADD(l, r->getIntValue() * sizeofType(lhs));
11   LLIR_Register *v  = INSTRUCTIONS->CreateRegister("");   // vreg1
12   INSTRUCTIONS->insertLDW(v, l, 0);
13   return v;
14 }
15
16 reg: plusExp(reg, reg)
17 {
18   $cost[0] = $cost[2] + $cost[3] + COST(INS_ADD_32);
19 }
20 =
21 {
22   LLIR_Register *rhs = $action[3]();                // vreg1
23   LLIR_Register *lhs = $action[2]();                // 5
24   INSTRUCTIONS->insertADD(rhs, lhs->getIntValue());
25   return rhs;                                       // vreg1
```

```
26 }
27
28 reg: tpm_assignExp(reg, reg)
29 {
30   $cost[0] = $cost[2] + $cost[3] + COST(INS_MOV_32);
31 }
32 =
33 {
34   LLIR_Register *rhs = $action[3]();                // vreg1
35   LLIR_Register *lhs = $action[2]();                // vreg2
36   INSTRUCTIONS->insertMOV(lhs, rhs);
37   return lhs;                                       // vreg2
38 }
```

The listing 4.2 shows the three example rules which are able to cover the expression tree in figure 2.3 which corresponds to the C statement in listing 2.1. The cost parts come up with the keyword `cost[...]` where `0` denotes the cost of the current rule, `2` and `3` denote the costs of the two children. `action[...]()` is the second keyword and is used within the action part of a rule to continue code selection on a child node. In other words the statement in line 22 stores the register created by `tpm_indexExp` in `rhs`. Figure 4.1 depicts how the tree-rewriting looks like when applying the rules to the expression tree in figure 2.3. In this example the base address of the array A is stored in the virtual register vreg0. Assuming the array has elements of type `int`, 8 is added to the base address of the array to point to the 3rd element which is read afterwards and stored into vreg1.

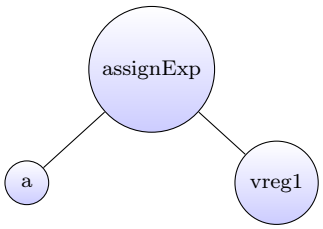
As can be seen in listing 4.3 the offset of the load instruction is 0 since the offset has already been added to the base address of A with the add instruction. The offset inside the load instruction is an immediate operand with a maximal size of 12 bit. This is an example where the same rule can be implemented twice to save instructions which yields a cheaper rule. Here, one rule could be used to insert the offset directly into the load instruction as an immediate operand if it is smaller than 12 bit. Another rule can be implemented to determine an absolute address and use it inside the load instruction with an offset of 0 if the offset is larger than 12 bit. The determination of an absolute address would require additionl instructions, but the cheaper rule can be selected if possible.

After reading from the array, 5 is added to that value in listing 4.4 and the entire result is then assigned to a register represented by vreg2.



Listing (4.3) indexExp applied

```
add vreg0, vreg0, 8
ldw vreg1, [vreg0, #0]
```



Listing (4.4) plusExp applied

```
add vreg1, vreg1, 5
```

Listing (4.5) assignExp applied

```
mov vreg2, vreg1
```

Figure 4.1: Expression reduction using rules from listing 4.2

4.4 The Stack

Before heading into the discussion of the actual rule design for array, struct and union types the stack has to be mentioned. Each of the listed types will get their own place on the stack if they are defined locally within a function in contrast to single variables which are only hold within registers. Those variables will only be put onto the stack by the register allocator if free registers have to be made available to other instructions, see section 2.3.1.

Figure 4.2 shows an example on how local composed type stack variables are organized on the stack. It shows an excerpt of the stack situation right before `foo` returns to `main` based on the listing 4.6. As it can be seen, the stack holds `main`'s local variables `A` and `E` as well as a buffer with the size of the largest struct which can be returned by any function which is called by `main`. If a function with overflow arguments gets called, additional space on the local stack of the calling function is reserved as denoted. A function

Listing 4.6: Local stack partitioning

```
1  struct SomeStruct {
2      int a;
3      int b;
4  };
5
6  struct SomeStruct foo(struct SomeStruct B) {
7      B.a = 5;
8      int C[3] = {1,3,5};
9      struct SomeStruct D = {7,9};
10     return D;
11 }
12
13 int main() {
14   struct SomeStruct A = {3,5};
15   struct SomeStruct E;
16   E = foo(A);
17   return 0;
18 }
```

has overflow arguments if it has more than four 32 bit arguments. Those arguments are than passed via the stack according to the ARM procedure call standard [16]. The first four 32bit arguments are always passed through the registers R0-R3. The stack frame of `foo` starts with the backup of main's registers `fp`, `sp`, `lr` and `pc`. The local data `C` and `D` is filled in from the bottom of `foo`'s stack frame followed by the section where passed structs get copied to. The ARM procedure call standard also states to 8-byte-align a function frame on the stack. If a function's frame is not already 8-byte-aligned, an empty memory section is inserted between the local data and the saved registers of the caller. The entire order of the local stack starting at the bottom is as follows: local variables, passed structs, return buffer, backup of caller's registers. The stack frame Adjustment and the determination of the needed stack frame is described in detail in [3]. The Frame pointer always points to the function frames highest memory address. The WCC Framework provides a `Stack` class to determine the stack layout for each function during compilation.

Remember. The stack grows down in terms of memory addresses with the call of a function. This is why data is stored from bottom to top to enable the access to local data like `C[2]` with `base pointer + sizeof(int) * 2`.

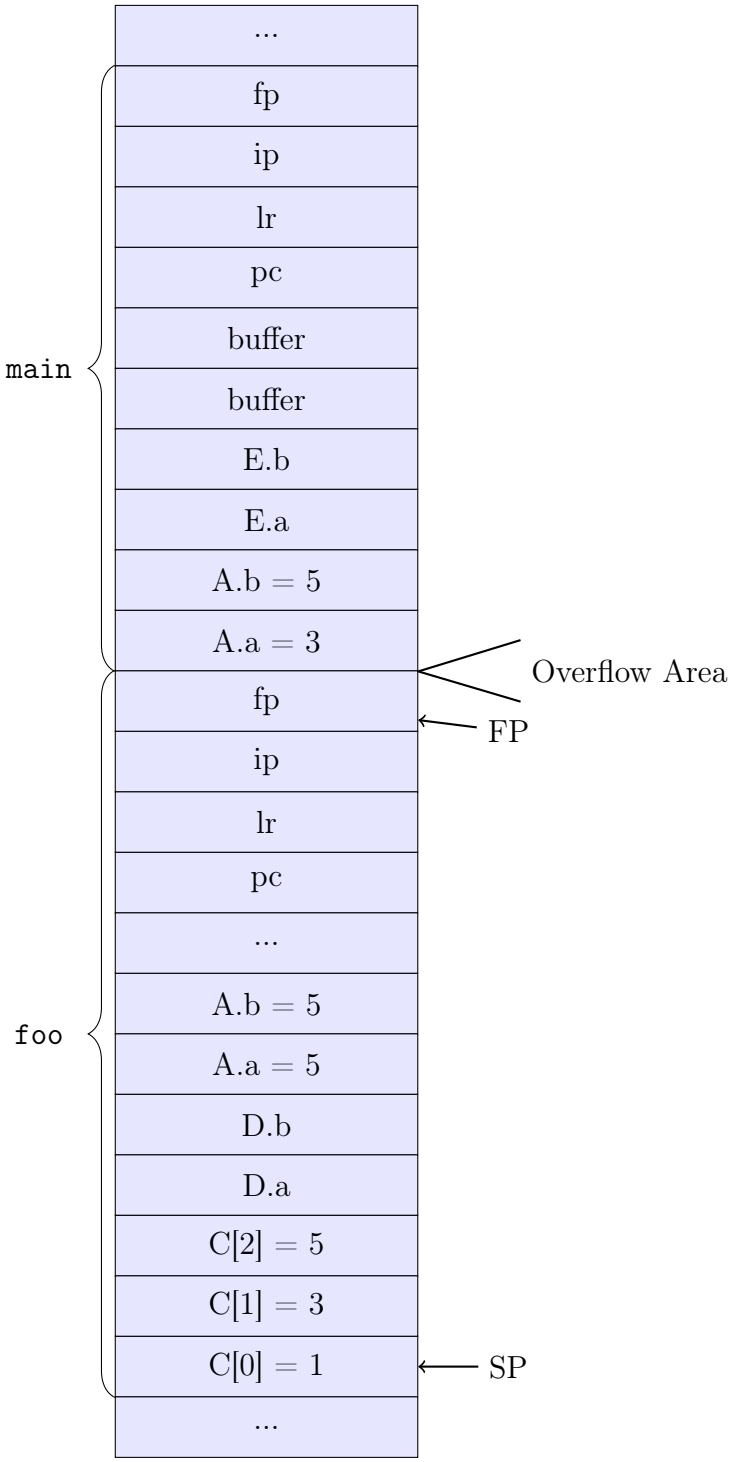


Figure 4.2: Local stack partitioning

Listing 4.7: Struct Person

```
1  struct Person {
2      int age;
3      char name[20];
4  };
5
6  int main() {
7    struct Person people[3] = {{24, "Alice"}};
8    struct Person sam;
9    sam.age = 42;                        // writing access
10   int age = sam.age;                   // reading access
11   people[0] = sam;                     // writing sam to people[0]
12   struct Person alice = people[1];
13 }
```

4.5 Composed Type Handling

Each array, struct or union locally defined inside a function, has its own reserved continuous space on the stack to store its data in. Global arrays are handled differently, they are bundled into the `.data` section of the binary executable. The executable can not only contain machine instructions for the programme itself, held inside the `.text` section, but also data which ships with the programme. The listing 4.7 provides a C example for the usage of composed types throughout the next sections.

4.5.1 Accessing Arrays

The access to arrays can be divided into two types: reading and writing access. The subtree for an expression like `people[0]` is depicted in figure 4.3, where `tpm_SymbolExp` refers to the array `people` and `tpm_ConstExp` to the index `0`. The rule `reg: deref_reg` is optional here. However, from the C perspective, the index expression itself does not tell anything about whether it is a read or write access, visualized in line 11 and 12 in listing 4.7. Therefore it's unknown inside the index expression whether to place a load instruction and return the corresponding register where the value is loaded to or to return a register containing a target location (address) if the index expression is used in a writing context. The C language provides a useful property to solve this problem. Assignment expressions are the only ones where an array can be used as a left hand operand, i.e. writing access can only occur in such an expression. To avoid multiple definitions of `tpm_IndexExp` for both

Listing 4.9: Determine register relative address

```
1 mov vreg2, #elementSize
2 mul vreg3, vreg1, vreg2       // vreg1 holds the variable index
3 add vreg0, vreg0, vreg3       // vreg0 holds the arrays base address
```

access types, a new non-terminal is introduced, called `deref_reg`. The declaration of `deref_reg` is shown in listing 4.8. The type `struct DerefInfo` can hold multiple information like the resulting register of a read access or the base address register and the offset for a writing access. The boolean `loadResult` is used to instruct the `tpm_IndexExp` to either perform a load instruction or to return the memory location for a write access. With the corresponding rule `reg: deref_reg` a reduction from `deref_reg`s to `reg`s can be performed where `tpm_IndexExp` is instructed to load a value (read access). This enables the usage of index expressions within all other rules which expect their operands as a value inside a register. Furthermore, this rule is restricted to `deref_reg`s which are not used in a writing context.

The write access to arrays is handled by a specialized assignment rule with signature `reg: tpm_AssignExpASSIGN(deref_reg, reg)`. As can be seen, the expression requires its left hand operand to be of non-terminal shape `deref_reg`. Here, the rule `tpm_AssignExpASSIGN` receives its operand directly from the `tpm_IndexExp` rule and does not instruct it to perform a load but to place the memory address and offset into the `deref_reg`.

There are two version of `tpm_IndexExp`, one of them to handle constant offsets as depicted in figure 4.3 and another one to cover index expressions with variable offsets. Since the `deref_reg` can only model addresses with constant offsets but not variable ones, some instructions are placed into the LLIR to overcome this problem. The inserted code determines the absolute address of the element from an index expression. This allows the return of a `deref_reg` with an absolute address and offset 0. The inserted assembly instructions are shown in listing 4.9.

Listing 4.8: non-terminal `deref_reg` declaration

```
1 %declare<struct DerefInfo> deref_reg<bool loadResult>;
```

4.5.2 Global vs Local Arrays

As already mentioned in section 4.5 arrays can be either global to the entire programme or local to a function which means that they are stored inside the `.data` section of the executable or on the stack. No matter of which type

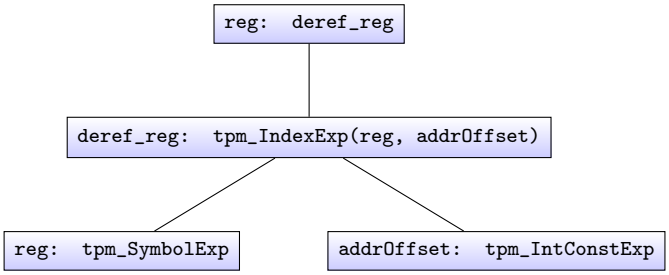


Figure 4.3: Array access rules

they are, they will reside inside the memory during execution. Each time an array is used, its surrounding expression tree will come up with the terminal node `tpm_SymbolExp` as depicted for the index expression in figure 4.3. A specialized rule for array symbols with signature `reg: tpm_SymbolExp` is introduced to figure out the memory address of its symbol. It performs a different action for each scenario.

Local Symbol If the array is local to a function, the symbol's address on the stack is calculated via the insertion of an add instruction, where the symbol's offset to the `sp` is added to the `sp` to yield the final address.

Global Symbol Since the address of the global array is not known during code selection, additional code is inserted to postpone the determination of the address. If a global array gets accessed, a LLIR basic block with a unique label is inserted after the last BB of the function which is currently in the code selection process. This is done once for all accesses to that global array per function. Into that BB a `.word` *`Symbol`* directive is placed. This `.word` will later be filled by the linker with the real address of the global array. This construct allows the usage of a load instruction to the label of the inserted BB, since the first entry will contain the address of the global array in the linked binary. If multiple global variables are accessed, multiple BB's get inserted at the end of a function.

Since this construct inserts data (the global array's address) into the `.text` section, it is important that it sits inside a LLIR basic block behind the last basic block of a function to guarantee that it is not reachable during execution. The interpretation of data as an instruction can lead to undefined behaviour of the programme.

4.5.3 Multi-dimensional Arrays

As C not only allows single dimensional arrays but also multi-dimensional ones, a rule mechanism is needed to handle any arbitrary depth. The idea is to consecutively assemble the address to the cell which is accessed. Imagine the array `{{{1,2},{3,4},{5,6}},{{7,8},{9,10},{11,12}}}` and the expression `A[1][2][0]` pointing to the memory which holds the 11. The rule cover for such an expression is shown in figure 4.4. Assuming that each number is a 4 byte integer, the elements of the innermost arrays are 4 byte in size, the mid level arrays' elements sizes are $2 \cdot 4 = 8$ byte and the outermost array's elements are $3 \cdot 8 = 24$ byte in size. The address calculation is carried out by the compiled programme itself and not by the compiler in advance. This can lead to inefficiencies for deeper nested arrays but can be overcome with later optimisations. As can be seen, the rule cover for a three-dimensional array contains three index expressions. The address calculation starts with the base address of the outermost array, provided by the symbol expression. The first index expression then inserts an add instruction to calculate the offset from the outermost array's address to the next nested array. In this case $1 \cdot 24 = 24$ byte. The result is the absolute address to an inner array. The next index expression uses the calculated address as the base address for its own offset calculation, here the addition of $2 \cdot 8 = 16$ byte. The innermost offset calculation, here $0 \cdot 4 = 0$ byte, is always handled by the index expressions discussed in section 4.5.1. The final offset is 40 byte which is exactly the needed offset from the arrays base address to the address of `A[1][2][0]`.

It can be seen that those index expressions can be chained for an arbitrary depth of multi-dimensional arrays. Like the index expressions discussed in section 4.5.1 the chain rule has a version for variable indices as well. They perform the same insertion of assembly code to generate an absolute address of a nested array.

4.5.4 Array Passing

Passing arrays to functions or returning an array from a function is uncomplicated, since C arrays are always accessed via one base address which points to the first element of the array. Therefore, just the base address has to be passed for a function argument of type array. Hence, the called function works on the original data instead of a copy like for any other function argument type. The same properties apply to return an array. According to the ARM procedure call standard [16], arguments are passed through registers R0-R4 or in the overflow area on the stack to a function and the return value

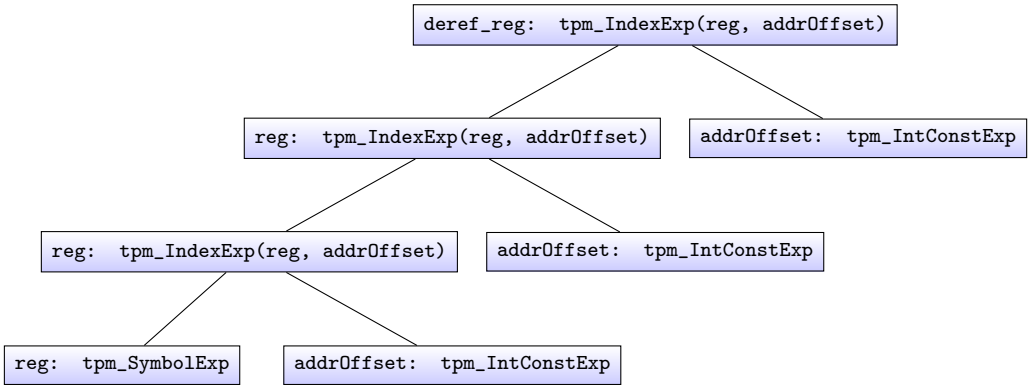


Figure 4.4: Multi-dimensional array access rules

is always returned through register R0. Because only addresses are needed to pass arrays, which can be passed through registers, no further rules have to be implemented to enable arrays as arguments and return values.

Remember. No array locally defined inside a function can be used as a return value, since the function's frame on the stack, where the array is stored, will be destroyed with the return of the function, see section 4.4. Thus, only arrays which came as an argument of the function can be returned. However, the C language prohibits the direct use of arrays as return values. It is only possible to return an array's address indirectly using pointer arithmetic.

4.5.5 Accessing Structs

Just like the access to arrays, the access to structs can be divided into read and write access. A subtree for the expression `sam.age` from listing 4.7 can either be used in a writing context (line 9) or a reading context (line 10) and is visualized in figure 4.5. Here, the left `tpm_SymbolExp` refers to the address of the struct `sam` and the right `tpm_SymbolExp` provides the offset to the element `age` inside the struct. The approach for a rule cover is identical to the one described for arrays in section 4.5.1. In short, the node coverd by the `tpm_ComponentAccessExp` is reduced to a non-terminal node `deref_reg`, which can be used to either get the address of the accessed element inside the struct or can be further reduced by the rule `reg: deref_reg` to get the value of the accessed element.

The assignment to struct members is matched by the same rule which has been introduced for assignments to array elements. But the C language not only allows the assignment to struct members but also the assignment of an entire struct to another one, given that they are equally typed. In contrast,

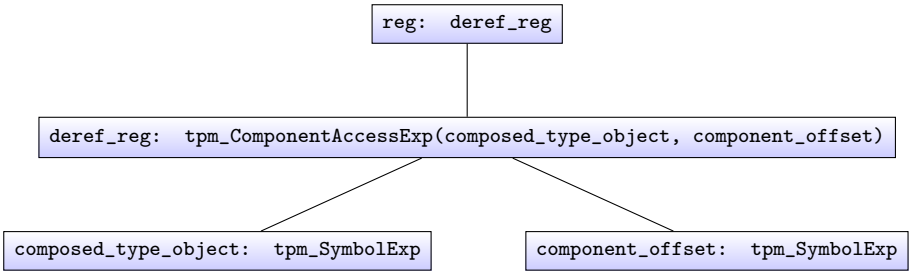


Figure 4.5: Struct access rules

such an assignment is forbidden for equally typed and sized arrays. The assignment of a struct `A` to another one `B` requires a copy mechanism to copy the entire data from `A`'s memory location to `B`'s memory location. This is handled by an assignment rule, designed for this special case. Its signature is `composed_type_object: tpm_AssignExpASSIGN(composed_type_object, composed_type_object )`. Within such an assignment the struct `B` is then copied in 4 byte blocks into the memory location of `A` using a 4 byte load and a 4 byte store instruction to load a 4 byte block from `A`'s to `B`'s location. If the struct's size is not a multiple of 4, the 4 byte load and store instructions are followed by a 2 or 1 byte load and store instruction or both of them.

4.5.6 Global vs Local Structs

Global and local structs are handled exactly the same way arrays are treated. For more details refer to section 4.5.2. However, struct symbols are processed by their own rules dedicated to overcome some struct specific issues. Two rules are introduced for global and local symbols. Both rules have the following signature `composed_type_object: tpm_SymbolExp` to cover a struct symbol node and to reduce it to a composed type object, which can be used throughout the other struct processing rules. One of them is dedicated to global symbols and does exactly the same as its counterpart for global arrays. Local structs instead need special attention, they can either be a struct locally defined to a function or they can be an argument, which came in with the call of the function. This belongs to struct passing, for more see section 4.5.8

4.5.7 Structs inside Structs

Just like multi-dimensional arrays, structs can be used as members of other structs. This requires a similar mechanism to handle the access to an arbitrary depth of nested structs. Listing 4.10 shows an example of a nested

Listing 4.10: Nested struct

```
1  struct Foo {
2    int a;
3    int b;
4  };
5
6  struct Bar {
7    struct Foo c;
8    int d;
9  };
10
11 int main() {
12   struct Bar bar;
13   bar.c.b = 3;
14 }
```

struct, where `Foo` is a member of `Bar`. The rule cover of the nested component access expression `bar.c.b` in line 13 is depicted in figure 4.6. As can be seen, the non-terminal `composed_type_object` is used to model a struct internally. It can hold the base address register and an offset from that base address to the struct. Other than array elements, struct members are addressed by their identifiers defined during the declaration of the struct type, instead of using a constant or variable index. Therefore, the offset to each member inside the struct is known during compilation. This allows to add up the offsets during a nested component access expression. In this example, the offset to the struct `bar` and the offset to its member `c`. This results in a `composed_type_object` being passed to the root `tpm_ComponentAccessExp` with a base address and the offset pointing to last nested struct, `c` in this case.

4.5.8 Passing Structs

Structs as function arguments and return values are passed call-by-value in the C language other than arrays which are just referred to via their address. This requires a copy mechanism which copies arguments of struct type from the stack frame of the calling function into the stack frame of the called function and vice versa to return a struct from a function. This mechanism is required since a copy of a struct cannot be simply passed via a register if it exceeds the size of 4 byte which is usually the case. Although structs of

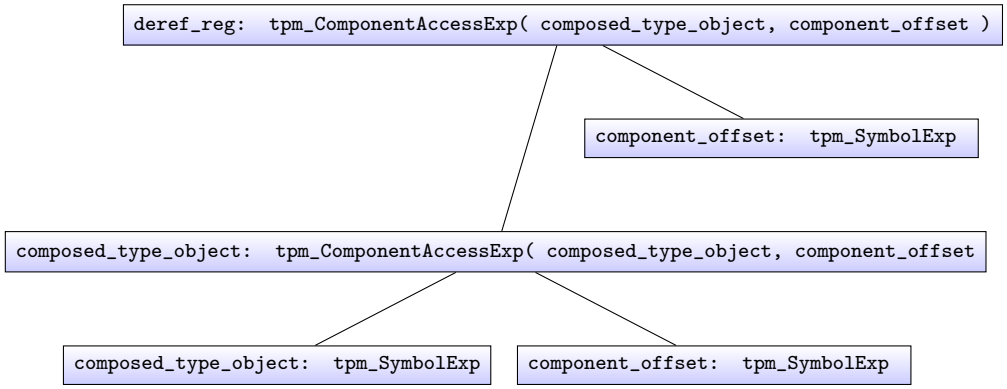


Figure 4.6: Nested Struct rule cover

equal or smaller size could be passed through a register, a special handling for this case is not implemented.

4.5.8.1 Structs as Arguments

During the call, the stack frame of the called function does not exist yet, which requires the copy operation to be performed by the called function after it has been called. Therefore, only the struct's address is passed to the called function. Load and store instructions are then placed into the first BB of the called function to copy the passed struct into its local stack frame. This copy operation however is only performed if the struct is used throughout the function. The called function reserves a buffer for all struct argument it takes, see section 4.4. The copy procedure is the same as the one already introduced for structs during the assignment of an entire struct to another one in section 4.5.5. This functionality is integrated by the specialized rule: `arg: tpm_CallExpARG(composed_type_object, arg)`.

4.5.8.2 Structs as Return Values

To return a struct from a function, a call result buffer is managed by the calling function, see section 4.4. This buffer is then used by the called function to place the copy of the struct, which should be returned, into. The copy to the call result buffer has to be performed by the called function before it returns, otherwise the original and locally stored struct is destroyed. This ultimately results in the need for the address of the call result buffer by the called function. The solution, as suggested by the ARM procedure call standard [16], is to pass the address as an additional parameter. This is accomplished through a special rule where the return type is a composed type

object: `composed_type_object: tpm_CallExp(called_function, arg)`. The additional parameter is prepended to the other function's parameters and is therefore always passed through register $R0$, whereas the other arguments are passed via the registers $R1 - R3$ and the parameter overflow area. This enables the called function to copy the returning struct into the stack frame of the calling function before its own stack frame is destroyed. Since register $R0$ is used to return a value from a function, it holds the address to the call result buffer, where the copied struct resides. The return statement is handled by a specialized rule with signature: `stmt: tpm_ReturnStmt(composed_type_object)`.

If a function contains multiple function call statements where differently sized structs are returned, the call result buffer is sized to fit the largest result into it.

Remember. Other than arrays, structs locally defined inside a function can be used as a return value, since they are copied to the stack frame of the called function before the frame of the function, where it has been defined, is destroyed.

4.5.9 Structs inside Arrays and vice versa

Since arrays and structs are used to compose new complex types out of already existing basic types, they of course can be embedded into each other. This requires extra rules to handle structs inside arrays and to do the opposite, arrays inside structs. As for structs and arrays, those rules are chain rules, where the address to a nested member is determined by the addition of the address of the surrounding object and the offset to the nested member. The rule `composed_type_object: tpm_IndexExp(reg, addrOffset)` is a specialized index expression, which does exactly the same as in section 4.5.3, but returns a composed type object to model a struct instead of a reg which is used to model an array. There is a second version as well, which processes variable indices instead of static ones.

Similarly, arrays inside structs have their own component access expression, namely: `reg: tpm_ComponentAccessExp(composed_type_object, component_offset)`. Like the last introduced rules, the rule performs the offset calculation as well by inserting an add instruction where the offset to the array member is added to the structs base address. The component then returns a reg with the absolute address to the accessed array.

4.5.10 Initializer lists

To rapidly fill a composed type, the C language allows the usage of initializer lists as can be seen in listing 4.7 in line 7. The initializer list allows the complete initialization of a composed data structure during its definition. As can be seen, it is allowed to truncate the initializer list to only partly initialize the data structure, in the example only the first struct of the array gets initialized. Since an initializer list can be of arbitrary length a rule for each possible length would be needed to be able to cover all initializer lists. This is not realizable and therefore the initializer list is expressed as a tree structure consisting of just 3 different terminals, namely `tpm_InitListExp`, `tpm_InitListExpELEM` and `tpm_InitListExpNOELEM`. The entire list is represented by `tpm_InitListExp`. The list itself is built from the nestable expression `tpm_InitListExpELEM` where the corresponding rule, which covers this node, represents one element and the rest of the init list. Figure 4.7 shows the rule cover for the initializer list from listing 4.7 in line 7. The figure additionally shows the handling of initializer lists for a composition of composed types. As visualized, an `init_list` can be reduced to an `init_list_element` to be used as an element of another init list.

The C language further requires that initializer lists can only be applied during the declaration of a composed type variables and not to earlier declared variables. The rule `initialized_object: tpm_AssignExpASSIGN( initializable_object, init_list)` is a specialized rule for assignment expressions which guarantees that the left-hand side of the operation has not been declared earlier.

4.5.11 Unions

As already explained in section 4.2, each union member is referred to with offset 0 within a union type. Due to the fact that the access to unions in C has the same syntax as the access to structs, expression trees containing unions are covered by structs rules as well. The only difference is the offset of each component of a union which is always 0. The offsets for struct or union members are always received through the stack class which enables the usage of the entire rule set, which is already used for structs, to be used for unions as well. Hence, no additional rule is needed to cover the usage of unions.

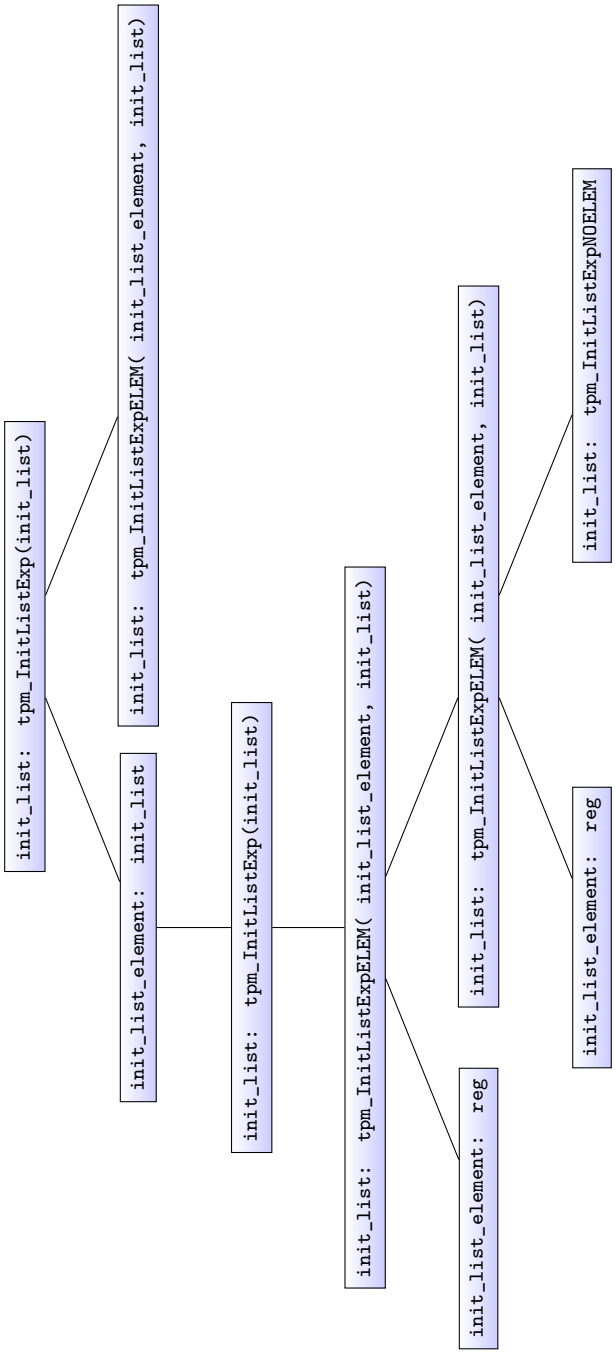


Figure 4.7: Initializer list cover

Rule	Description
`deref_reg: tpm_IndexExp(reg, IntConstExp)`	array access with constant index
`deref_reg: tpm_IndexExp(reg, reg)`	array access with variable index
`reg: tpm_IndexExp(reg, IntConstExp)`	multi-dimensional array access with constant index
`reg: tpm_IndexExp(reg, reg)`	multi-dimensional array access with variable index
`reg: tpm_SymbolExp`	determines address of array symbol
`stmt: tpm_ExpStmt(composed_type_object)`	reduces struct to statement
`deref_reg: tpm_ComponentAccessExp(composed_type_object, component_offset)`	struct access
`composed_type_object: tpm_AssignExpASSIGN(composed_type_object, composed_type_object)`	assignment of entire struct to another one
`composed_type_object: tpm_ComponentAccessExp(composed_type_object, component_offset)`	chain rule for nested struct
`composed_type_object: tpm_CallExp(called_function, arg)`	passing of CallResultBuffer address
`arg: tpm_CallExpARG(composed_type_object, arg)`	passing of struct arguments
`stmt: tpm_ReturnStmt(composed_type_object)`	copys return value to CallResultBuffer
`composed_type_object: tpm_SymbolExp`	determines address and offset of struct
`component_offset: tpm_SymbolExp`	determines offset of struct member
`reg: tpm_AssignExpASSIGN(deref_reg, reg)`	assignment to arrays or structs
`reg: deref_reg`	reduces deref_reg to value stored inside it
`composed_type_object: tpm_IndexExp(reg, addrOffset)`	indexing struct inside array with constant offset
`composed_type_object: tpm_IndexExp(reg, reg)`	indexing struct inside array with variable offset
`reg: tpm_ComponentAccessExp(composed_type_object, component_offset)`	accessing array inside struct
`init_list: tpm_InitListExp(init_list)`	represents whole init list
`init_list: tpm_InitListExpELEM(init_list_element, init_list)`	represents init list as one element and the rest of the list
`init_list: tpm_InitListExpNOELEM`	represents end of init list
`init_list_element: reg`	enables to place constants and variables into init list
`init_list_element: init_list`	allows nested init lists
`initialized_object: tpm_AssignExpASSIGN(initializable_object, init_list)`	initialize variable during definition with initializer list
`initializable_object: tpm_SymbolExp`	garuantees a not earlier declared array variable
`initializable_object: composed_type_object`	guarantees a not earlier declared struct variable
`reg: initialized_object`	reduces array definition to its address
`composed_type_object: initialized_object`	reduces struct definition to its address and offset

Table 4.1: Implemented rules

Chapter 5

Evaluation

This Chapter presents the results of the developed grammar as well as a comparison of the assembly code generated by the WCC and the ARM-GCC. The analysis starts with a visual comparison of the assembly outputs. Five C code examples are compiled with the WCC and the ARM-GCC in the following: the first two handle array types, the next two structs and the last one discusses unions. The analysis only focuses on the code sections which are relevant to array, struct and union processing. For more information on arithmetic, loop and branching rules refer to the work of Plog [4] and for information on the stack adjustment code which comes with function calling, refer to the work of Bouraffa [3].

One difference, which will be visible throughout all examples, is the stack pointer relative addressing mode used by the WCC and the frame pointer relative addressing mode used by the ARM-GCC. Moreover the ARM-GCC generated assembly output is based on real registers where the register allocator has already been invoked. The assembly of the WCC instead is based on virtual registers.

This visual analysis is followed by the evaluation of the WCET and ACET resulting from the binaries generated by the WCC and the ARM-GCC. This analysis makes use of a very basic register allocator to map the virtual registers to physical hardware registers.

5.1 Arrays

There are two examples to demonstrate how arrays are handled by the code selector, the first one highlights indexing array elements and passing of arrays. The second one shows global and multi-dimensional arrays.

5.1.1 Array Indexing and Parsing

The listing 5.1 is the source code of the assembly output of the WCC depicted in listing 5.2, whereas the assembly, generated by the ARM-GCC is shown in listing 5.3.

Listing 5.1: Array indexing and parsing

```
1  int foo(int A[], int a) {
2    A[2] = 11;
3    return 0;
4  }
5
6  int main() {
7    int A[3] = {1,2,3};
8    int a = 1;
9    foo(A, A[a]);
10   return 0;
11 }
```

The listing starts with the definition of the function `foo` which takes an array and an integer. `foo` is followed by the `main` function, which defines a local array `A` and an integer `a`. Upon the function call of foo, `A` is passed as a whole and `a` is used within the index expression, to pass a copy of the second element of `A` (index 1). The overall structure of the assembly outputs of WCC and ARM-GCC are the same. The array handling however differs in some kinds which is depicted in the following. Starting with the function `foo` in the assembly code generated by WCC in figure 5.2, the content of `p_3` is moved to `p_4` right after the stack adjustment instructions at the beginning. `p_3` is precoloured with 0 to force the register allocator to use register `r0` when mapping the virtual registers to real hardware registers. However, the precolouring is not visible in the assembly code. Since the array `A` is moved through `r0` as the first operand, its address resides in `p_4` after the move instruction in line 5. The virtual register `p_4` is then used as the base address with an offset of 8 byte inside the `str` instruction in line 7 to store value 11 into `A`'s third field, after it has been moved to `p_0`. The offset is correctly calculated as 8 byte, since the elements of `A` are of integer type which is 4 byte in size and an array's address always points to the first element of the array. The ARM-GCC instead generates code where the `str` instruction receives the absolute address to the third field of `A` with an offset of 0, see listing 5.3, line 12. The absolute address is calculated in line 10 where the offset of 8 is added to the base address stored in `r3`. The instructions from line 7 to line 9 after the stack adjustment code are spill

instructions inserted by the register allocator to store the arguments passed to `foo` onto its local stack. For a brief explanation to spill code, refer to section 2.3.1. Ignoring the spill instructions, the assembly generated for `foo` by WCC needs one less instruction compared to ARM-GCC.

Continuing with the `main` function, two more differences are visible. Taking the lines 17-22 in listing 5.2, it can be seen how the WCC populates `A`'s stack space with the initializer list. Each value is first moved into a virtual register and then stored into its appropriate stack location. The values have to be moved into a register, since the `str` instruction is not capable of storing an immediate value. The ARM-GCC performs a different strategy, the values are not inserted as immediate values of move instructions, the initializer list instead ships inside the `.rodata` section of the binary. This can be seen from lines 17-21 in listing5.3. The initializer list has the name `C.0.1471` which is used in line 50. In a later stage of the compiler process, this `.word` will be replaced with the address to the memory location of the initializer list inside the `.rodata` section. This enables the usage of the initializer list for the load instruction in line 28. It loads the content from memory, where `.L5` points to, which is exactly the address to the initializer list. The address is loaded to register `r3` which is then used by the load multiple instruction in line 30 to load all three values of the initializer list into the register `r0`, `r1` and `r2` simultaneously. Afterwards, they are stored onto the stack with the store multiple instruction to initialize the array `A`. This approach might need less instructions than the one of the WCC, but it needs all required registers (here: `r0`, `r1` and `r2`) to be unused. This can again lead to the insertion of spill code, if not enough registers are available. The second difference comes with the index expression `A[a]`. As it can be seen in line 23 and line 26 in listing 5.2 the values 1 and 4 are moved to the registers `p_11` and `p_18`. They are the variable index and the size of the array's elements (4 byte). In line 27, they are multiplied to calculate the offset which is stored into `p_17`. Afterwards, the offset is added to the base address of the array, which resides in `p_15` to produce an absolute address pointing to the desired element of `A`. This address is finally used in the load instruction in line 29 with an offset of 0 to retrieve the requested data. This address calculation construct is necessary to enable the usage of variable indices. The ARM-GCC in contrast uses a more complicated address calculation approach, this can be seen in lines 35-40. In short: -16 is stored in `r2` with `mvn` and added to `r3`, where `r3` has been populated with a frame pointer relative address so that the addition of -16 yields the desired memory location of the second element (index 1). Overall, both variants need 6 instructions for the address calculation, and don't differ in their needed code size.

Listing 5.2: Assembly code generated by WCC for listing 5.1

```
1  foo:
2          mov     ip, sp
3          stmfd   sp!, {fp, ip, lr, pc}
4          sub     fp, ip, #4
5          mov     p_4, p_3
6          mov     p_0, #11
7          str     p_0, [p_4, #8]
8          mov     p_6, #0
9          mov     p_5, p_6
10         sub     sp, fp, #12
11         ldmfd   sp, {fp, sp, pc}
12 main:
13         mov     ip, sp
14         stmfd   sp!, {fp, ip, lr, pc}
15         sub     fp, ip, #4
16         sub     sp, sp, #16
17         mov     p_7, #1
18         str     p_7, [sp, #0]
19         mov     p_8, #2
20         str     p_8, [sp, #4]
21         mov     p_9, #3
22         str     p_9, [sp, #8]
23         mov     p_11, #1
24         add     p_12, sp, #0
25         add     p_15, sp, #0
26         mov     p_18, #4
27         mul     p_17, p_18, p_11
28         add     p_15, p_15, p_17
29         ldr     p_14, [p_15, #0]
30         mov     p_3, p_12
31         mov     p_19, p_14
32         bl      foo
33 _L2:
34         mov     p_20, r0
35         mov     p_22, #0
36         mov     p_21, p_22
37         sub     sp, fp, #12
38         ldmfd   sp, {fp, sp, pc}
```

Listing 5.3: Assembly code generated by ARM-GCC for listing 5.1

```
1         .text
2 foo:
3         mov     ip, sp
4         stmfd   sp!, {fp, ip, lr, pc}
5         sub     fp, ip, #4
6         sub     sp, sp, #8
7         str     r0, [fp, #-16]
8         str     r1, [fp, #-20]
9         ldr     r3, [fp, #-16]
10        add     r2, r3, #8
11        mov     r3, #11
12        str     r3, [r2, #0]
13        mov     r3, #0
14        mov     r0, r3
15        sub     sp, fp, #12
16        ldmfd   sp, {fp, sp, pc}
17        .section        .rodata
18 C.0.1471:
19        .word   1
20        .word   2
21        .word   3
22        .text
23 main:
24        mov     ip, sp
25        stmfd   sp!, {fp, ip, lr, pc}
26        sub     fp, ip, #4
27        sub     sp, sp, #16
28        ldr     r3, .L5
29        sub     ip, fp, #28
30        ldmia   r3, {r0, r1, r2}
31        stmia   ip, {r0, r1, r2}
32        mov     r3, #1
33        str     r3, [fp, #-16]
34        ldr     r3, [fp, #-16]
35        mvn     r2, #15
36        mov     r3, r3, asl #2
37        sub     r1, fp, #12
38        add     r3, r3, r1
39        add     r3, r3, r2
40        ldr     r2, [r3, #0]
41        sub     r3, fp, #28
42        mov     r0, r3
```

```
43          mov     r1, r2
44          bl      foo
45          mov     r3, #0
46          mov     r0, r3
47          sub     sp, fp, #12
48          ldmfd   sp, {fp, sp, pc}
49  .L5:
50          .word   C.0.1471
```

5.1.2 Multi-dimensional and Global Array

The C code in listing 5.4 starts with the definition of a global two-dimensional array. Global data is stored into the `.data` section of the executable which will later reside in a memory section which is readable and writeable. This is equally done by the WCC (listing 5.5, line 2) and the ARM-GCC (listing 5.6, line 2). Additionally, it can be seen how the two-dimensional data structure is stored into the one-dimensional memory. The `main` function only contains one statement, where the value 11 is written into the array. The index expression addresses a single element of the two-dimensional array. In this case the value 1 indexes the second array inside the array `A` and the value 2 points to the last element of the nested array, i.e. the last element. The assembly code generated by WCC is depicted in listing 5.5, where the relevant instructions are in lines 14-17. First the base address to the global data is loaded to `p_4` from label `_L1`. Afterwards, the offset for the first index expression is added to the base address, which is 12 byte in this case. This new address is now the base address to the nested array `{4,5,6}`. The offset inside this array is determined by the second index with value 2 and therefore yields an offset of 8 byte which is directly inserted into the store instruction in line 17. The ARM-GCC produces more compact assembly code. This can be seen in lines 14-16 where the entire offset of 20 byte is determined by the compiler and directly inserted into the store instruction. This has to be faced by future optimisations of the WCC, since each level of nesting produces an additional instruction during the access of an element.

Listing 5.4: Global two-dimensional array

```
1  int A[2][3] = {{1,2,3},{4,5,6}};
2
3  int main() {
4    A[1][2] = 11;
5    return 0;
6  }
```

Listing 5.5: Assembly code generated by WCC for listing 5.4

```
1         .section .data
2   A:
3         .word 1
4         .word 2
5         .word 3
6         .word 4
7         .word 5
8         .word 6
9         .section .text
10  main:
11        mov     ip, sp
12        stmfd   sp!, {fp, ip, lr, pc}
13        sub     fp, ip, #4
14        mov     p_0, #11
15        ldr     p_4, _L1
16        add     p_4, p_4, #12
17        str     p_0, [p_4, #8]
18        mov     p_6, #0
19        mov     p_5, p_6
20        sub     sp, fp, #12
21        ldmfd   sp, {fp, sp, pc}
22  _L1:
23        .word   A
```

Listing 5.6: Assembly code generated by ARM-GCC for listing 5.4

```
1         .data
2   A:
3         .word   1
4         .word   2
5         .word   3
6         .word   4
7         .word   5
8         .word   6
9         .text
10  main:
11        mov     ip, sp
12        stmfd   sp!, {fp, ip, lr, pc}
13        sub     fp, ip, #4
14        ldr     r2, .L3
15        mov     r3, #11
16        str     r3, [r2, #20]
17        mov     r3, #0
```

```
18          mov     r0, r3
19          ldmfd   sp, {fp, sp, pc}
20  .L3:
21          .word   A
```

5.2 Structs

Structs are special in terms of passing, since they are passed call-by-value and they are copied upon returning them from functions. Both scenarios are analysed in the following two examples.

5.2.1 Struct Passing

Memory-wise, structs are handled like arrays, local structs are placed into the local function stack and global structs are placed into the `.data` section of the binary. Also, the initialization of local structs using initializer lists is performed in the same way as it is done for arrays, as can be seen in listing 5.8 for the WCC and in listing 5.9 for the ARM-GCC. WCC consecutively moves each value of the initializer lists into a register and stores it into its appropriate stack location. The ARM-WCC again holds the initializer list inside the `.rodata` section and populates `A` using load and store multiple instructions.

This example focuses on the difference between the code which is generated for function calls with structs as arguments. As visible in listing 5.8 in line 38, WCC passes the address of the struct `A` through register `p_1`. `p_1` is precoloured with 0, to make sure that the register allocator uses register `r0` here. Since `main` has no other local data than `A`, `sp` is `A`'s base address. Focusing on the lines 7-16 in `bar`, it can be seen that the function starts with load and store instructions, which copy the struct `A` to the local stack of `bar`. Here, `p_1` in line 6 is again precoloured with 0, where the address of A has been passed. The ARM-GCC uses a different approach: instead of passing the address of the struct, the struct itself is passed. This can be seen in lines 35-38 in listing 5.9. For this task the first 16 byte are passed through the registers `r0-r3`, each 4 byte in size. The rest is passed through the stack. `A` contains the component `a` with size of 4 byte and the second component `b` with a size of 16 byte. That means that `a` and the first 12 byte of `b` are passed through registers `r0-r3` and the last 4 byte of `b` are passed via the stack. In lines 35-36, the last element of `b` is stored into the argument overflow section on the stack. In lines 37-38, `a` and the first 12 byte of `b` are loaded into registers `r0-r3`. In lines 7-8 in `bar` the data which has

been passed via registers is stored onto the stack. The ARM-GCC performs distinctive actions depending on the size of the passed struct: if the struct was divided to pass it via registers and the stack, the data passed through registers is stored into the argument overflow area where the rest already resides. If the entire struct can be passed through registers, the struct will be stored into the local stack of the called function. The WCC is not in need of such a distinction, since the address is passed and the passed struct will always be copied into the local function stack by the called function. However, due to the intensive use of load and store multiple instructions by the ARM-GCC its output is usually smaller in terms of code size.

Listing 5.7: Passing struct to function

```
1  struct Foo {
2    int a;
3    int b[4];
4  };
5
6  int bar (struct Foo B) {
7    B.a = 3;
8    return 0;
9  }
10
11 int main() {
12   struct Foo A = {5,{1,2,3,4}};
13   bar(A);
14   return 0;
15 }
```

Listing 5.8: Assembly code generated by WCC for listing 5.7

```
1  bar:
2          mov     ip, sp
3          stmfd   sp!, {fp, ip, lr, pc}
4          sub     fp, ip, #4
5          sub     sp, sp, #24
6          mov     p_2, p_1
7          ldr     p_3, [p_2, #0]
8          str     p_3, [sp, #0]
9          ldr     p_3, [p_2, #4]
10         str     p_3, [sp, #4]
11         ldr     p_3, [p_2, #8]
12         str     p_3, [sp, #8]
13         ldr     p_3, [p_2, #12]
```

```
14          str     p_3, [sp, #12]
15          ldr     p_3, [p_2, #16]
16          str     p_3, [sp, #16]
17          mov     p_0, #3
18          str     p_0, [sp, #0]
19          mov     p_6, #0
20          mov     p_5, p_6
21          sub     sp, fp, #12
22          ldmfd   sp, {fp, sp, pc}
23  main:
24          mov     ip, sp
25          stmfd   sp!, {fp, ip, lr, pc}
26          sub     fp, ip, #4
27          sub     sp, sp, #24
28          mov     p_7, #5
29          str     p_7, [sp, #0]
30          mov     p_8, #1
31          str     p_8, [sp, #4]
32          mov     p_9, #2
33          str     p_9, [sp, #8]
34          mov     p_10, #3
35          str     p_10, [sp, #12]
36          mov     p_11, #4
37          str     p_11, [sp, #16]
38          mov     p_1, sp
39          bl      bar
40  _L2:
41          mov     p_13, r0
42          mov     p_15, #0
43          mov     p_14, p_15
44          sub     sp, fp, #12
45          ldmfd   sp, {fp, sp, pc}
```

Listing 5.9: Assembly code generated by ARM-GCC for listing 5.7

```
1           .text
2   bar:
3           mov     ip, sp
4           sub     sp, sp, #16
5           stmfd   sp!, {fp, ip, lr, pc}
6           sub     fp, ip, #20
7           add     ip, fp, #4
8           stmia   ip, {r0, r1, r2, r3}
9           mov     r3, #3
```

```
10          str     r3, [fp, #4]
11          mov     r3, #0
12          mov     r0, r3
13          sub     sp, fp, #12
14          ldmfd   sp, {fp, sp, pc}
15          .section        .rodata
16  C.0.1471:
17          .word   5
18          .word   1
19          .word   2
20          .word   3
21          .word   4
22          .text
23  main:
24          mov     ip, sp
25          stmfd   sp!, {fp, ip, lr, pc}
26          sub     fp, ip, #4
27          sub     sp, sp, #24
28          ldr     r3, .L5
29          sub     lr, fp, #32
30          mov     ip, r3
31          ldmia   ip!, {r0, r1, r2, r3}
32          stmia   lr!, {r0, r1, r2, r3}
33          ldr     r3, [ip, #0]
34          str     r3, [lr, #0]
35          ldr     r3, [fp, #-16]
36          str     r3, [sp, #0]
37          sub     r3, fp, #32
38          ldmia   r3, {r0, r1, r2, r3}
39          bl      bar
40          mov     r3, #0
41          mov     r0, r3
42          sub     sp, fp, #12
43          ldmfd   sp, {fp, sp, pc}
44  .L5:
45          .word   C.0.1471
```

5.2.2 Struct Returning

Returning structs from functions is equally managed by the WCC and the ARM-GCC, depicted in listing 5.11 for WCC and listing 5.12 for ARM-GCC, based on the source code in listing 5.10. Since the `main` function does not

contain any local data, the return buffer for structs is the only section on the local stack of `main`. It can be seen in listing 5.11 how the stack pointer with offset 0 is moved to `p_4` in line 25. `p_4` is precoloured with 0 to guarantee the usage of register `r0` for `p_4` during register allocation. This is the desired behaviour, since the address to the return buffer shall be passed in register `r0` if a function returns a struct. In lines 7-16 of the function `bar` the local struct can then be copied to the call result buffer. The address to the call result buffer is then returned in `p_3`, precoloured with 0, to guarantee the return in register `r0`. The ARM-GCC performs the same procedure, but other than the WCC it uses load and store multiple instructions for the copy process. This enables the production of a smaller assembly output than the assembly generated by WCC in terms of code size.

Listing 5.10: Returning struct from function

```
1  struct Foo {
2    int a;
3    int b[4];
4  };
5
6  struct Foo bar() {
7    struct Foo A;
8    return A;
9  }
10
11 int main() {
12   bar();
13   return 0;
14 }
```

Listing 5.11: Assembly code generated by WCC for listing 5.10

```
1  bar:
2          mov     ip, sp
3          stmfd   sp!, {fp, ip, lr, pc}
4          sub     fp, ip, #4
5          sub     sp, sp, #24
6          mov     p_0, p_1
7          ldr     p_2, [sp, #0]
8          str     p_2, [p_0, #0]
9          ldr     p_2, [sp, #4]
10         str     p_2, [p_0, #4]
11         ldr     p_2, [sp, #8]
12         str     p_2, [p_0, #8]
```

```
13          ldr     p_2, [sp, #12]
14          str     p_2, [p_0, #12]
15          ldr     p_2, [sp, #16]
16          str     p_2, [p_0, #16]
17          mov     p_3, p_0
18          sub     sp, fp, #12
19          ldmfd   sp, {fp, sp, pc}
20  main:
21          mov     ip, sp
22          stmfd   sp!, {fp, ip, lr, pc}
23          sub     fp, ip, #4
24          sub     sp, sp, #24
25          add     p_4, sp, #0
26          bl      bar
27  _L2:
28          mov     p_5, r0
29          mov     p_7, #0
30          mov     p_6, p_7
31          sub     sp, fp, #12
32          ldmfd   sp, {fp, sp, pc}
```

Listing 5.12: Assembly code generated by ARM-GCC for listing 5.10

```
1   bar:
2           mov     ip, sp
3           stmfd   sp!, {r4, fp, ip, lr, pc}
4           sub     fp, ip, #4
5           sub     sp, sp, #20
6           mov     r4, r0
7           mov     lr, r4
8           sub     ip, fp, #36
9           ldmia   ip!, {r0, r1, r2, r3}
10          stmia   lr!, {r0, r1, r2, r3}
11          ldr     r3, [ip, #0]
12          str     r3, [lr, #0]
13          mov     r0, r4
14          sub     sp, fp, #16
15          ldmfd   sp, {r4, fp, sp, pc}
16  main:
17          mov     ip, sp
18          stmfd   sp!, {fp, ip, lr, pc}
19          sub     fp, ip, #4
20          sub     sp, sp, #20
21          sub     r3, fp, #32
```

```
22          mov     r0, r3
23          bl      bar
24          mov     r3, #0
25          mov     r0, r3
26          sub     sp, fp, #12
27          ldmfd   sp, {fp, sp, pc}
```

5.3 Unions

The following example in listing 5.13 shows the access to union members. The `main` function contains a union `Foo` named `A`. The value 5 is assigned to the member `a` and afterwards the value 7 to the first field of the member `b`. Since union members share the same memory space, those two assignments should go into the same memory location. This is observable in the assembly output generated by WCC (listing 5.14) in lines 7 and 10. Since the union `A` is the only local data of `main`, `A`'s base address is equal to the stack pointer. As it can be seen, both values are stored into the memory using the stack pointer with an offset of 0. The `add` instruction in line 9 results from the rule which handles component access to struct/union members. The added offset might be non-zero for struct members but is always zero for union members. This can be targeted in future optimisations. The ARM-GCC output in listing 5.15 is looking very similar. But as mentioned, frame pointer relative addressing is used. Nonetheless, both values are stored into the same memory location in lines 7 and 9.

Listing 5.13: Accessing Unions

```
1  union Foo {
2    int a;
3    int b[4];
4  };
5
6  int main() {
7    union Foo A;
8    A.a = 5;
9    A.b[0] = 7;
10   return 0;
11 }
```

Listing 5.14: Assembly code generated by WCC for listing 5.13

```
1  main:
2          mov     ip, sp
3          stmfd   sp!, {fp, ip, lr, pc}
4          sub     fp, ip, #4
5          sub     sp, sp, #16
6          mov     p_0, #5
7          str     p_0, [sp, #0]
8          mov     p_2, #7
9          add     p_4, sp, #0
10         str     p_2, [p_4, #0]
11         mov     p_6, #0
12         mov     p_5, p_6
13         sub     sp, fp, #12
14         ldmfd   sp, {fp, sp, pc}
```

Listing 5.15: Assembly code generated by ARM-GCC for listing 5.13

```
1  main:
2          mov     ip, sp
3          stmfd   sp!, {fp, ip, lr, pc}
4          sub     fp, ip, #4
5          sub     sp, sp, #16
6          mov     r3, #5
7          str     r3, [fp, #-28]
8          mov     r3, #7
9          str     r3, [fp, #-28]
10         mov     r3, #0
11         mov     r0, r3
12         sub     sp, fp, #12
13         ldmfd   sp, {fp, sp, pc}
```

5.4 Automated Tests

The WCC framework allows to perform the compilation with either the built-in code selector, register allocator and linker or to use the ARM-GCC to produce the compiled binary. It then allows to analyse both binaries in terms of worst and avarage case execution times. The resulting timings are given as processor cycles in table 5.1. The table lists the timings for six programmes produced by the WCC and the ARM-GCC. The source code for those can be found on the attached CD as well as the generated assembly code of the

WCC and the ARM-GCC. Since the WCET analysis does not include the timings for included library functions, the WCET for some binaries produced by the ARM-GCC are not listed. ARM-GCC invokes `memcpy` whenever large sections of data have to be copied.

initlist This programme just contains the definition of a local array using an initializer list. Therefore it is compiled with the `-O0` flag to avoid any optimisation. Otherwise, the definition would be discarded. As it can be seen, the ACET of the binary generated by WCC is smaller than the one of its counterpart from the ARM-GCC. The ARM-GCC makes use of `memcpy` to copy the initial values into the array, whereas the WCC populates the array using immediate operands as discussed in section 5.1.1.

arrayCopy Within this example, an array is copied to another array element-wise. It can be seen, that the programme produced by the WCC needs twice as many cycles as the one generated by the ARM-GCC. This is mainly due to the very basic register allocator which performs a spill for each used register. It therefore stores each copied value onto the stack before it is actually copied to the other array.

structCopy structCopy does the same as arrayCopy, but there is no need for an element-wise copy, since structs are always entirely copied if they are assigned, passed or returned. Hence, an assignment of a struct to another one is enough to copy its entire content. The WCC performs a byte-wise copy, whereas the ARM-WCC instructs `memcpy` to perform the copy.

structReturn WCC and ARM-GCC perform the same action for the return of a struct as they do for the assignment of a struct to another. The returning struct has to be copied into the call result buffer of the calling function. WCC performs a byte-wise copy, whereas ARM-GCC invokes `memcpy`.

structCompAccess and arrayIndex structCompAccess defines a structure of nested structs of depth 2, whereas arrayIndex defines a three-dimensional array. Therefore an index expression to an innermost element of the array and a component access expression to an innermost element of the array have the same depth considering the offset calculation. Each example then performs a large amount of accesses to an innermost element. The timings show that the access to arrays needs a significant larger amount of cycles than the access to structs in the

case of the WCC. This is due to WCC being able to determine the entire offset for structs during compilation, whereas the offsets for index expressions are calculated on the target platform. Unexpectedly both programmes come up with the same timings when compiled with the ARM-GCC. As it can be seen structCompAcces generated by WCC needs about twice as many cycles as the ARM-GCC version. The difference for arrayIndex is even larger, the WCC product needs about four times as many cycles as its ARM-GCC counterpart.

	WCC		ARM-GCC		
programmes	WCET	ACET	WCET	ACET	note
initlist	5773	5570	-	6433	-O0, memcpy
arrayCopy	62175	58773	31337	30325	
structCopy	60165	56155	-	26237	memcpy
structReturn	9216	8514	-	5522	memcpy
structCompAccess	36369	34567	16531	15719	-O0
arrayIndex	64369	61367	16531	15719	-O0

Table 5.1: ACETs and WCETs comparison

Chapter 6

Summary and Future Work

During this work, the code selector of the WCC for ARM processors has been extended. A rule set has been developed to face the code generation for the composed types: arrays, structs and unions. The developed rules seamlessly integrate into the existing rules and enable the usage of the composed types in various contexts. Namely, passing to and returning from functions, component access inside other expression as well as the usage of initializer lists. However, the generated assembly code should be targeted in later optimisation stages to discard unnecessary instructions.

Future work to the code selector are the processing of types other than integers as well as implicit and explicit casting to change a variables type. Another topic is the management of pointers which are of special interest for composed types.

In a wider perspective, the development of a fully fledged register allocator is of special interest to generate an optimal mapping from virtual to physical registers. This leads to a significant decrease in required instructions than currently needed as depicted in section 5.4.

List of Figures

List of Tables

Bibliography

[1] Sam Siewert. *Real-time embedded components and systems*. Da Vinci engineering series. Charles River Media, Boston, Mass., 1. ed edition, 2007.

[2] Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46(2):251–300, Oct 2010.

[3] Abir Bouraffa. WCC Code Selection Framework for the ARM Processor Architecture, May 2017.

[4] Janina Plog. Developing a Code Selector's Ruleset for the ARM Processor, April 2017.

[5] Paul Lokuciejewski. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Embedded Systems. 2011.

[6] Informatik Centrum Dortmund. Icd-c compiler framework, March 2010.

[7] Heiko Falk. Compilers for embedded systems. Lecture, 2017.

[8] S. W. K. Tjiang. An olive twig. Technical report, Synopsys, Inc., 1993.

[9] Steven S. Muchnick. *Advanced compiler design and implementation*. 1998.

[10] AbsInt. The industry standard for static timing analysis. Online.

[11] ARM Limited, 110 Fulbourn Road Cambridge, England CB1 9NJ. *ARM Architecture Reference Manual*, i edition, July 2005.

[12] EE Times. Snapdragon seeds qualcomm's future, 2007.

[13] Warren W. Gay. *Raspberry Pi Hardware Reference*. Apress, Berkeley, CA s.l., 2014.

[14] ARM Limited, 110 Fulbourn Road Cambridge, England CB1 9NJ. *ARM7TDMI Technical Reference Manual*, c edition, November 2004.

[15] Daniel Tabak. *RISC systems*. Industrial control, computers and communications series. Research Studies Press u.a., Taunton u.a., 1990.

[16] Procedure call standard for the arm architecture, October 2009.

[17] Informatik Centrum Dortmund. Icd-llir low-level intermediate representation, March 2010.

[18] ISO/IEC. International Standard ISO/IEC 9899:1999(E) Programming languages - C, 1999.

Appendix A

CD Content

The content of the CD is listed in table A.1 below.

Directory	Description
`/thesis.pdf`	This document
`/thesis/`	LaTeX code for this document
`/CODESEL/`	Source files of the entire codeselector
`/evaluation/functionality`	Source and assembly code files for functional evaluation
`/evaluation/performence`	Source and assembly code files for WCET and ACET comparison of WCC and ARM-GCC

Table A.1: CD content