Bachelor Thesis
Computing Science

Radboud University Nijmegen

Code generation for the Thumb-2 instruction set

Author:
Camil Staps

First supervisor:
prof. dr. dr.h.c. ir. M.J. Plasmeijer

Second supervisor:
drs. J.H.G. van Groningen

Thursday 19th January, 2017

This page intentionally left blank.

Abstract

The Thumb-2 instruction set combines the best features of the ARM and Thumb instruction sets (speed and small code size, respectively). We discuss the differences between the ARM and Thumb-2 instruction sets, and their influences on code generation. Specifically, we look at code generation for the purely functional programming language Clean. The code generator proposed here can be used in situations where small code size is important, and on devices where the ARM instruction set is not available, like the Cortex-M series. It produces on average 20% smaller code than the ARM code generator, which is only around 4% slower.

This page intentionally left blank.

Contents

1 Introduction

1.1 ARM, Thumb and Thumb-2

ARM is a RISC architecture[1] with several enhancements to a basic RISC architecture allowing ARM processors to 'achieve a good balance of high performance, small program size, low power consumption, and small silicon area' [1, A1.1].

Several instruction sets were designed for the ARM architecture. First of all, the 32-bit ARM ISA allows the programmer to easily make full use of all the architecture's features. The Thumb instruction set provides a 16-bit alternative to the ARM ISA, giving in on performance to achieve improved code density. Starting from ARMv6T2, an extension to the Thumb instruction set, known as Thumb-2, adds 32-bit instructions to Thumb to 'achieve performance similar to ARM code, with code density better than that of earlier Thumb code' [1, A1.2]. This gives the ARM and Thumb instruction sets 'almost identical functionality' [1, A1.2], whereas Thumb gives a smaller code size.

In this thesis, we will usually use 'Thumb' where the Thumb-2 extension is meant. Only when the distinction with pre-ARMv6T2 Thumb is important will we distinguish between (early) Thumb and Thumb-2. As for 'ARM', it should be clear from the context whether the architecture or the instruction set is meant.

Using the Unified Assembler Language (UAL), one can write assembly code for both the ARM and the Thumb instruction set and fix the target ISA only at assemble-time [2, A4.2].

The main differences between ARM and Thumb-2 are the following:

1.1.1 Conditional execution

In ARM, every instruction has a 4-bit conditional field that allows for conditional execution. In the Thumb instruction set, all conditional instructions except branches have to be in an 'IT block'. A first `it` instruction gives the base condition and a then-else pattern. The statements after the `it` instruction are executed conditionally. For example:

```
itte   gt     @ tte: then, then, else
movgt  r2,r3  @ mov if gt
movgt  r0,r1  @ mov if gt
movle  r0,#0  @ mov if le (= not gt)
```

This is UAL syntax. When assembling for Thumb, an `it` instruction with three `mov` instructions (without conditional field) is generated. For ARM, the `it` instruction is ignored and three conditional `mov` instructions are generated.

ARMv8-A deprecates some uses of the `it` instruction for performance reasons [3, J5.2].

1.1.2 Register usage

ARM processors have sixteen registers. ARM instructions have 4-bit register fields to address them. Some 16-bit Thumb instructions have 3-bit register fields that can only address the lowest eight registers. For these instructions there exist 32-bit variants that can address all sixteen registers.

[1]RISC stands for *Reduced Instruction Set Computing.* RISC architectures provide relatively few, basic instructions that can be executed fast to improve performance compared to large, slow architectures.

1.1.3 Interworking

The ARM and Thumb instruction sets are designed to interwork: different parts of a program can be assembled for different instruction sets and it is possible to switch instruction set when an instruction writes to the program counter [1, A4.1].

The Thumb-2 code generator proposed in this thesis does not produce ARM code, though the existence of the interworking facility has effects on the techniques that can be used in it. This will be covered in section 3.

1.2 Clean

Clean is ‘a general purpose, state-of-the-art, pure and lazy functional programming language designed for making real-world applications’ [4]. It evolved from LEAN, an intermediate language based on graph rewriting [5]. A Clean program consists of a list of rewrite rules, for example:

```
fac 0 = 1
fac n = n * fac (n - 1)
Start = fac 4
```

Executing a Clean program means rewriting the `Start` rule until no rewriting is possible any more. The first rewriting steps of the above program are shown in figure 1.

1.2.1 The ABC-machine

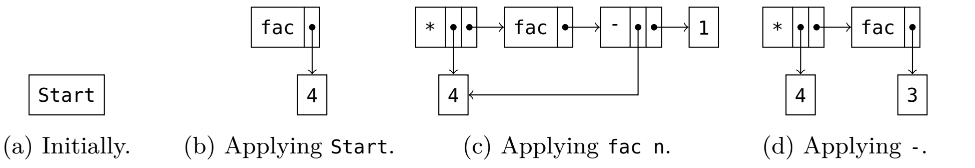
A Clean program is compiled to ABC-code, a platform-independent language for the ABC-machine, an abstract graph rewriting machine [6]. A code generator is used to generate machine code equivalent to the ABC-code for concrete machines. This, again, is done in several steps, so that only a relatively small part of the compilation process is target-dependent.

The ABC-machine is ‘an imperative abstract machine architecture for graph rewriting’ [6]. It consists of three stacks: for arguments (A), basic values (B) and control information like return addresses (C). It also has a program store containing the graph rewriting rules that make up the program, a graph store that contains the graph to be rewritten, and several other elements that we need not mention here.

We do not discuss the internals of the ABC-machine in much depth here — they have been described in [6] and [7]. However, to get a better idea of the working of the ABC-machine, we consider the ABC-code for the factorial example above briefly.

The two rules for `fac 0` and `fac n` are translated to the following ABC-code[2]:

[2]The actual code generated by the compiler is slightly larger, since it uses some unnecessary instructions. They are removed automatically by the code generator and are not considered here for brevity.



(a) Initially. (b) Applying `Start`. (c) Applying `fac n`. (d) Applying `-`.

Figure 1: Rewriting a Clean node.

```
sfac.1
    eqI_b 0 0       | Is argument on B-stack 0?
    jmp_true case.1 | Jump to first alternative
    jmp case.2      | Jump to second alternative
case.1              | First alternative (fac 0)
    pop_b 1         | Pop argument from B-stack
    pushI 1         | Push result to B-stack
    rtn             | Return
case.2              | Second alternative (fac 1)
    pushI 1         | Push 1 to B-stack
    push_b 1        | Copy argument to B-stack
    subI            | Subtract on B-stack
    jsr sfac.1      | Call factorial
    mulI            | Multiply two stack elements
    rtn             | Return
```

The code under `sfac.1` is for the pattern match on the first argument. In `case.1`, we handle the case that it is zero. We only need to remove it from the stack and push the return value 1. In `case.2`, we copy the argument, decrement it and recursively call `sfac.1`. After the `jsr` instruction, we will have two elements on the stack: the result of the recursive call and the original argument. We can the multiply them and return.

This function uses the B-stack, for basic values, for its arguments and return values, because it uses only integers. More complex types would be passed on the A-stack.

The `Start` rule compiles to[3]:

```
__fac_Start
    build _ 0 nStart.2 | Build the Start node
    jmp _driver        | Jump to the RTS
nStart.2               | The Start node entry
    jsr eaStart.2      | Jump to actual code
    fillI_b 0 0        | Copy result to A-stack
    pop_b 1            | Pop B-stack
    rtn                | Return
eaStart.2              | Actual code
    pushI 4            | Push 4 to B-stack
    jmp sfac.1         | Call factorial
```

When the program starts and `__fac_Start` is executed, a node for `Start` is created in the graph store. We then jump to the run-time system (discussed in section 1.2.3), where the `_driver` subroutine will make sure that this node is reduced and printed. In `nStart.2`, we call `eaStart.2`. This subroutine will execute `fac 4` and return the result on the B-stack. Since `nStart.2` is expected to return its result on the A-stack, we need to copy it and clean up the B-stack. In `eaStart.2`, we only need to set the argument for the factorial, 4, on the stack, and we jump to `sfac.1` to let it do the rest.

1.2.2 The code generator

The code generator translates the abstract ABC-code into concrete machine code. While the ABC-machine can be implemented in a straightforward manner using macro expansion, Clean's code generator does more than that: it introduces several optimisations, some of which are target-dependent.

The ARM code for the factorial example is as follows[4]:

[3]The code has been shorted insignificantly for brevity.
[4]Some irrelevant peculiarities have been removed for brevity.

```
sfac_P1:
  cmp  r4,#0       @ Is argument 0?
  bne  case_P2
case_P1:
  mov  r4,#1       @ Return 1
  ldr  pc,[sp],#4
case_P2:
  str  r4,[sp,#-4]! @ Copy argument
  add  r4,r4,#-1    @ Decrement argument
  str  pc,[sp,#-4]!
  bl   sfac_P1      @ Call factorial
  ldr  r12,[sp],#4  @ Multiply with own argument
  mul  r4,r12,r4
  ldr  pc,[sp],#4
```

We see that the argument is passed in `r4`, and that register is also used for the result. In ARM, `r4` through `r0` are used for the top of the B-stack. The machine stack pointer `sp` is used as the B-stack pointer. For intermediate results, `r12` is used as a scratch register. When a reduction has finished, control is passed back to the callee by loading the program counter from the stack (`ldr pc,[sp],#4`).

The `Start` rule translates roughly to:

```
____fac__Start:
  ldr  r12,=nStart_P2 @ Store Start node on the heap
  str  r12,[r10]
  mov  r6,r10         @ Save node pointer on A-stack
  add  r10,r10,#12    @ Reserve heap space
  b    __driver       @ Jump to RTS
nStart_P2:
  str  r6,[r9],#4     @ Save node entry
  str  pc,[sp,#-4]!   @ Save return address
  bl   eaStart_P2     @ Jump to actual code
  ldr  r6,[r9,#-4]!   @ Restore node entry
  ldr  r12,=INT+2     @ Make an Int node on A-stack
  str  r12,[r6]
  str  r4,[r6,#4]     @ Give that node value r4
  ldr  pc,[sp],#4     @ Return to callee
eaStart_P2:
  mov  r4,#4          @ Push 4 to B-stack
sfac_P1               @ A jump to sfac_P1 is not
  @ ...               @ needed as it is right below
```

For the A-stack, `r6` through `r8` and `r11` are used, and `r9` is the A-stack pointer. Nodes (the graph store of the ABC-machine) are stored on a heap which uses `r10` as a pointer. To store the `Start` node on the heap, we need to write its address and then increase `r10` to reserve the space. The `__driver` subroutine will reduce the top of the A-stack, so saving the pointer to the `Start` node in `r6`, the top of the A-stack, makes sure that `__driver` jumps to `nStart_P2`. There, we do some administration and jump to `eaStart_P2`. In that routine, the literal `4` is pushed to the B-stack and we 'jump' to `sfac_P1` — the code generator optimises this by placing `sfac_P1` right below and removing the branch instruction. When `sfac_P1` returns, we continue in `nStart_P2`. There we need to copy the result from the B-stack to the A-stack. Since the B-stack is untyped (the compiler makes sure that it is safe), while the A-stack is typed, we need to create a node on the heap with the `INT+2` entry address.

Note that ABC instructions like `pushI 4` (push 4 to the B-stack) are translated to ARM code like `mov r4,#4`: the content of the B-stack is not moved down, as one might expect. This is possible because the code generator knows that the B-stack will be empty at this point. Had it not been empty, but had one element, for example, a `mov r3,r4` instruction would have preceded to move it down. The actual ABC-code contains annotations that tell the code generator how many elements the stacks hold, so that these optimisations can be made.

1.2.3 The run-time system

After compilation, a Clean program is linked together with the Clean run-time system (RTS). The RTS ensures that that the `Start` node is reduced and printed and takes care of garbage collecting. This system is written in Clean, C and assembly, so making Clean available on Thumb-2 inherently means adapting the platform-dependent parts of the RTS as well.

1.3 A Thumb backend for Clean

In this thesis, we propose a Thumb backend for Clean. The ARM code generator and RTS were taken as a starting point. Thanks to the Unified Assembler Language, only little time was needed to make these systems assemble for the Thumb instruction set.

The proposed backend does not use ARM and Thumb interworking. The reason for this is threefold. First, there are several processors, like the ARMv7-M series [2, A4.1], that do support Thumb instructions but cannot run ARM code. By only using Thumb, we target the widest possible range of devices. Second, we doubt that using interworking can be done efficiently. In the run-time system, only minimal time overhead is introduced by using Thumb instructions. For generated code it would be complicated to detect if the ARM or Thumb instruction set would give better results, and this would give significantly better results only in specific cases. Third, the problem discussed in section 3 could be solved efficiently only without interworking. Using interworking would introduce overhead at every branch instruction, since the solution to this problem would have to be adapted.

1.4 Organisation

In much of the rest of this thesis we discuss differences between ARM and Thumb, their influences on code generation, and the way they were dealt with in the Thumb backend for Clean proposed in this thesis. In section 2, we consider an issue arising from halfword-aligned instructions and the way a read of PC is interpreted under Thumb. Section 3 discusses problems related to the fact that Thumb instruction addresses use bit 1 and should have bit 0 set for interworking, while under ARM a branch will automatically clear both these bits. For some instructions, the constants in the ARM variant can be larger than those in the Thumb variant. Section 4 deals with related issues in the `ldr` instruction.

Moving to Thumb introduces a number of interesting optimisation vectors. One of them, register allocation, is discussed in section 5.

We benchmark the Thumb backend for code size and running time in section 6. The results are discussed in section 7, where we also look at ideas for further research.

For an overview of the current status of the Thumb backend, see appendix A. Our testing environment is described in appendix B.

2 Storing the program counter

2.1 Introduction

Storing the program counter on the stack is something done commonly in many languages during a function call. Usually, an offset to the actual program counter at the moment of the store instruction is stored, to allow for a branch after that instruction, and have the callee return to the address after that branch. The ARM architecture accommodates for this: an ARM instruction that reads the program counter, actually reads 'the address of the current instruction plus 8' [1, A2.3]. The following ARM assembly example illustrates this (`armstartup.s:540-2`, [8]):

```
str     pc,[sp,#-4]!  @ 0x20
bl      init_clean    @ 0x24
tst     r4,r4         @ 0x28
```

Dummy addresses have been indicated in comments. When execution arrives at `0x20`, the program counter is set to `0x20`. Per the above documentation, `str` stores `0x20+8` on the stack (to be precise: to the address indicated by the stack pointer with pre-indexed offset `-4`). The processor branches to `init_clean`, which loads the value from the stack back into the program counter to return to the caller. The program counter is then `0x28`. For the next instruction cycle, the `tst` command is executed.

There are two reasons why the above cannot be used in Thumb-2 code. First, `pc` is not allowed as the first operand of a `str` instruction.

The second problem we meet is that the instruction to store the program counter may be halfword-aligned rather than word-aligned. We saw above that a read of the program counter in ARM mode reads as PC+8. In Thumb mode this is more complicated. In this case, we '[r]ead the word-aligned PC value, that is, the address of the current instruction + 4, with bits [1:0] forced to zero' [2, A5.1.2]. This means that when the `add` instruction above is at `0x22`, we will still store `0x2d` on the stack, since the word-aligned program counter is `0x20` as before. However, in this case `bl` is located at `0x2a`, and since this is a 32 bits instruction we point to the middle of that instruction.

2.2 Usage in Clean

Storing the PC is required for jumping to subroutines. We see a clear example of this when a Clean function definition uses pattern matching. Consider the following example:

```
isEmpty :: [a] -> Bool
isEmpty [] = True
isEmpty _  = False
```

The generated ABC-code looks as below[5]. Since `isEmpty` pattern matches on its first argument, it needs to be evaluated to head normal form. This is done with `jsr_eval 0`. Only after that can we check if its constructor is `_Nil` (the empty list), which is done with `eq_desc _Nil 0 0`.

[5]The current Clean compiler misses the `jsr_eval 0` line: the strictness analyser recognises that `isEmpty` is strict in its first argument, so evaluating the argument to head normal form is not needed any more. In cases where the strictness analyser cannot derive strictness, code similar to this example is generated. The code given here can be reproduced with `clm -nsa`.

```
sisEmpty.1
    jsr_eval 0        | Evaluate argument
    eq_desc _Nil 0 0  | If it equals []
    jmp_true case.1   | .. jump to case.1
    jmp case.2        | [else] to case.2
case.1
    pop_a 1           | Pop argument
    pushB TRUE        | Return True
    rtn
case.2
    pop_a 1           | Pop argument
    pushB FALSE       | Return False
    rtn
```

Evaluating the argument is done by jumping to the subroutine indicated by the node entry of that argument. Hence, we store the PC, jump to that address, and continue with `eq_desc _Nil 0 0` when the node has been evaluated to head normal form. The ARM code for the pattern match is:

```
sisEmpty_P1:
  ldr  r12,[r6]      @ Load node entry point
  tst  r12,#2        @ If in HNF already
  bne  e_0           @ Skip evaluation
  str  pc,[sp,#-4]!  @ Store PC
  blx  r12           @ Evaluate argument
e_0:
  ldr  r12,[r6]      @ If it does not equal []
  ldr  r14,=__Nil+2
  cmp  r12,r14
  bne  case_P2
```

We can see here that evaluating a node requires a `jsr_eval` ABC instruction, and that `jsr` ABC instructions require storing the PC in ARM assembly.

Of course, evaluating nodes is something that happens throughout the source code and has to be done all the time during the execution of a Clean program. We therefore need a fast, small Thumb alternative for the ARM code.

2.3 Solution

Recall from section 2.1 that we meet two issues when generating Thumb code. First, that `pc` cannot be the first operand of a `str` instruction; second, that the value read for the program counter is word-aligned while the read instruction may be halfword-aligned.

To solve the first problem, we need to first move `pc` to an auxiliary register, and then push that on the stack. We then get, for the example from `armstartup.s:540-2`:

```
add     lr,pc,#9      @ 0x20
str     lr,[sp,#-4]!  @ 0x24
bl      init_clean    @ 0x28
tst     r4,r4         @ 0x2c
```

We store the value `0x2d`. This address points to the `tst` instruction as before, with the LSB set to 1 to indicate Thumb mode.

The offset, 9, is calculated as the number of bytes to the instruction after the branch plus one for Thumb mode. For `b` and `bl` instructions, this means an offset of 9, since these instructions are 32-bit. The `bx` and `blx` instructions are 16-bit, and require an offset of 7.

The second problem is that when the `add` instruction is located at the start of a halfword (e.g. `0x22`), the value that is read for PC will still be `0x20`, as it is word-aligned [2, A5.1.2] When generating object code, we need to keep track of the current alignment and add either 7, 9 or 11 to the read program counter, depending on both the alignment and the size of the branch instruction. When generating assembly code, we are less concerned with efficiency. In this code generator, we simply force-align the `add` instruction (by adding an **`.align`** directive, which will insert a `nop` if necessary).

2.4 Implementation details

A jump always jumps to either a label (with `b` or `bl`) or a register (with `bx` or `blx`). The latter occurs for example in the case of a `jsr_eval` ABC instruction. This instruction is used to evaluate a node on the A-stack. First, the node entry address has to be fetched; then we jump to that address.

In the first case, we need one scratch register to store the PC temporarily. In the second case, we need two scratch registers: also one for the address we are jumping to. For the ARM instruction set we needed zero and one scratch register(s), respectively. The ARM backend uses two scratch registers, S0 and S1 (for more details, see section 5.2). The latter is used only when two scratch registers are needed, so S0 is used in this case.

The Thumb-2 code generator uses S0 to store the PC temporarily in the first case, and S1 in the second case (where S0 is still used for the address we are jumping to). This makes sure that S0 is used as much as possible. The slightly easier implementation would use S1 in both cases. However, in Thumb-2 it is convenient to have great variation in register usage: this allows for a massive code size optimisation (see section 5). For this reason it is better to use S0 whenever possible.

2.5 Comparison

Assuming the worst case, that all instructions in the jump block are wide, we need four more bytes in Thumb than in ARM. As a benchmark, the Clean compiler has 41,006 jumps of this kind in 1,253,978 instructions, or approximately 3.27%. The four extra bytes in Thumb mean a size increase of $41006 \cdot 4 \approx 160$KiB on the 5.3MiB file, an increase of 3.00%.

As for the time complexity: every subroutine call requires an extra instruction cycle. In particular, every reduction needs an extra cycle. It is hard to tell what effect this has on Clean programs in general, and it may well be very dependent on the kind of program. A general comparison of running time under ARM and Thumb is made in section 6.3.

2.6 Other solutions

Another solution than the one we have considered here makes use of the link register. Some branch instructions, like `bl`, store the address of the next instruction in the link register. We could therefore imagine a setup where the callee gets the return address from that register rather than from the stack. This is the approach taken by GCC. The code of a typical C subroutine starts with `push {...,lr}` and ends with `pop {...,pc}`.

When generating code for a functional language, it is not straightforward to do this, due to tail recursion. An example of this can be found in the following basic function:

```
length :: !Int ![a] -> Int
length n []     = n
length n [_:xs] = length (n+1) xs
```

The current Thumb backend generates the following code:

```
slength_P2:
      ldr   r0,[r6]         @ Load top of A-stack (the list)
      ldr   r14,=__Nil+2    @ Check if it is Nil
      cmp   r0,r14
      bne   case_P2
case_P1:                    @ If the list is Nil
      ldr   pc,[sp],#4      @ Simply return
      .ltorg
case_P2:                    @ If the list is not Nil
      ldr   r6,[r6,#8]      @ Load the Cons part of the list
      ldr   r0,[r6]         @ Check if it has been evaluated already
      tst   r0,#2
      bne   e_0
      str   r4,[sp,#-4]!    @ If the Cons part has not been evaluated
      .align
      add.w r14,pc,#7       @ Evaluate the cons part
      str.w r14,[sp,#-4]!
      blx   r0
      ldr   r4,[sp],#4
e_0:                        @ If / after the Cons part has been evaluated
      add   r4,r4,#1        @ Increase counter
      b     slength_P2      @ Tail recursively jump back to start of function
      .ltorg
```

In this setup, it is not easily possible to let the callee store the return address, since we enter the function as many times as there are elements in the list, while we only return from it once, in `case_P1`. A more straightforward solution to have the caller responsible for storing the return address, which is why this approach is taken in Clean's ARM code generator [9] and why we continue along these lines for the Thumb backend.

3 Code addresses

3.1 Introduction

The ARM ISA only has 32-bit instructions, meaning that the two least significant bits of a code address are always zero. When a branch is attempted to an address which has one or both of these bits set, the processor will automatically align the address by clearing these bits.

Thumb mixes 32 and 16-bit instructions. Instructions are halfword-aligned, so bit 1 is part of the address. Additionally, bit 0 is used to facilitate ARM and Thumb interworking. When the PC is written to with a value where the LSB is cleared, the processor jumps and switches to ARM mode. When the LSB is set, it switches to Thumb mode [1, A2.3.2].

3.2 Usage in Clean

The fact that in ARM mode the two lowest bits of a value written to the PC are cleared automatically has been exploited in the ARM code generator: these two bits are used to store information. Bit 1 is used to mark a node as having reached head normal form[6] (HNF). Bit 0 is used in Clean's default garbage collector, the copying collector, to record whether a node has been copied to the other semispace already: when cleared (the normal state of an address as stored on the heap), it has not been copied yet.

Porting to Thumb, we need to find a way to store this information without jumping to the middle of an instruction or accidentally switching to ARM mode.

3.3 Solution

The solution for bit 1 is straightforward. By aligning all code addresses that we will ever want to jump to (the node entry addresses) on words rather than half-words, we ensure that bit 1 is always cleared. When a node has reached head normal form, setting bit 1 will give a corrupt address. Jumping to it would either jump after the first instruction (if it is 16-bit) or to the middle of the first instruction (if it is 32-bit). However, when a node is in HNF, there is no pointing in jumping to its entry address, so the corrupted address is not a problem.

The Thumb backend for Clean proposed in this thesis does not use interworking. However, we do need to make sure that whenever the PC is written to, the LSB is set. To that end, we store node entry addresses with the LSB on heap and stack. This only introduces problems in the copying collector, which will then not copy any node (since having bit 0 set means having been copied already). Flipping the meaning of the LSB in the copying collector fixes this issue.

3.4 Comparison

Flipping the meaning of the LSB in the garbage collector amounts to swapping `bne` and `beq` and similar changes that do not effect the program's efficiency or size.

By word-aligning all node entry addresses we lose one alignment byte per node entry address on average (assuming that half of the node entry points are word-aligned already). This increases code size slightly, but since many instructions that were 32-bit in ARM are now 16-bit, the overall

[6]A node is in *head* or *root normal form* when it cannot be rewritten itself (though its children may).

code size is still smaller. Aligning node entries has no effect on the program's efficiency since the `nop` instruction that is inserted above the entry is never executed.

3.5 Other solutions

The solution described above is Clean-specific, since it exploits the fact that bit 0 of a code address is only used inside the garbage collector. The solution for bit 1, however, is not specific to the Clean RTS. Therefore, a general solution to the problem that the two LSBs of a code address cannot be used to store information in Thumb mode would be to align all addresses that we need to store info of on double-words, that is, ensuring the three LSBs are always zero. That way, the LSB can be used for ARM and Thumb interworking, and bit 1 and 2 can be used to store information.

Of course, whether this is a viable solution depends on the density of code addresses that should then be aligned. If every second instruction needs to be aligned, it would introduce so many padding instructions that code size will increase dramatically (even compared to ARM) and that performance degrades significantly.

Then again, in many programs the issue we have explored in this section will not be a problem at all, because the two LSBs of code addresses are not commonly used.

4 Reduced offset space for memory load instructions

4.1 Introduction

The `LDR` (immediate) instruction loads a word from memory into a register. The address is specified by a base register and an offset (which can possibly be added to the base register before or after the load). On ARM, this offset is encoded in twelve bits, and one bit is used for its sign. That allows for any offset between -4095 and 4095 bytes. For details, see [1, A8.8.63].

The Thumb instruction set defines four different variants of the immediate `LDR` instruction [2, p. 6.7.42]:

- 16-bits, any base register, offsets between 0 and 124; 0 modulo 4.
- 16-bits, SP as base, offsets between 0 and 1020; 0 modulo 4.
- 32-bits, any base register, offsets between 0 and 4095.
- 32-bits, any base register, offsets between -255 and 255.

Only the last variant can add the offset to the base register, either before or after the load instruction.

While in some cases a narrow `LDR` instructions can be used in Thumb, some uses in ARM cannot be used directly in Thumb: the offset can now only be between -255 and 4095 bytes.

4.2 Usage in Clean

For most Clean programs, the generated code will not attempt to load memory with large negative offsets. However, there are some modules with exceptionally large right hand sides, for which large negative offsets are generated. The largest in the Clean compiler is -600, in the `frontend/predef` module.

It is worth noting that the ARM code generator attempts to use as few clock cycles as possible to modify the A-stack. Instead of updating the A-stack pointer every time a load is done, the code generator keeps track of the difference between the stored A-stack pointer and the actual top of the stack. So, when three arguments are popped off the A-stack, we do not generate this code[7]:

```
ldr   r0,[r9,#4]!
ldr   r1,[r9,#4]!
ldr   r2,[r9,#4]!
```

Instead, the code is optimised to:

```
ldr   r0,[r9,#4]
ldr   r1,[r9,#8]
ldr   r2,[r9,#12]!
```

Instead of updating `r9`, the A-stack pointer, after every pop, the code generator modifies the offset and updates `r9` only after the last load.

[7]This is a hypothetical example, so we do not show ABC-code here.

4.3 Solution

We extend the optimisation scheme described in the previous paragraph to ensure no large negative offsets are generated. When generating code for a basic block, we now compute the minimum offset to the A-stack. If it is lower than -255, a `sub` instruction is added to modify the A-stack pointer before the load is executed.

In this case, we optimise the offset for code size. Let O be the set of offsets used in the basic block and $o_{min} \in O$ the lowest offset in that set. If we call the offset used in the `sub` instruction o_{start}, this expression gives the total size in bytes of all load instructions:

$$h\left(o_{start}\right) = \sum_{o \in O} s(o - o_{start}),$$

where

$$s(x) = \begin{cases} 16 & \text{if } 4 \mid x \text{ and } 0 \leq x \leq 124; \\ 32 & \text{otherwise.} \end{cases}$$

We therefore want to calculate

$$\min_{o_{min}-3 \leq o_{start} \leq o_{min}+255} h\left(o_{start}\right).$$

There is no reason to go lower than $o_{min} - 3$. In this case, all load offsets are positive, and decreasing o_{start} further will make less instruction fall into the first 16-bit variant listed above. We cannot go higher than $o_{min} + 255$, because we would then need a load offset lower than -255.

The `sub` instruction is inserted right before the first `ldr` instruction with a too large negative offset. This way, as many instructions as possible before the `sub` instruction fall into the 16-bit variant.

For example, if we have to load data with the offsets $0, 4, 8, 12, -512, 16, 20, 24, 28$, the following code is generated:

```
ldr   ..,[r9]
ldr   ..,[r9,#4]
ldr   ..,[r9,#8]
ldr   ..,[r9,#12]
sub   r9,r9,#-512
ldr   ..,[r9]
ldr   ..,[r9,#528]
ldr   ..,[r9,#532]
ldr   ..,[r9,#536]
ldr   ..,[r9,#540]
```

The first four loads are 16-bit, while putting the `sub` instruction at the start of the block would make them 32-bit.

4.4 Comparison

We have assumed that the largest difference between A-stack offsets used in one basic block is at most 4095. If it would be more, we would have to modify the A-stack pointer not only to avoid large negative offsets, but also to avoid large positive offsets. We have not found any module that uses large positive offsets, so the assumption is reasonable for the moment.

This assumption allows us to add only one `sub` instruction per basic block. With this, we lose one clock cycle. In the worst case, all `ldr` instructions are 32-bit. In that case we also lose 4 bytes of code size. However, in practice, at least some of the `ldr` instructions will be 16-bit, and we will in fact save some space.

Negative offsets occur so infrequently that they do not influence the overall results significantly.

4.5 Other solutions

Note that the solution proposed here makes a rigorous choice to minimise clock cycles. In the above example, we could optimise for code size by restoring `r9` when the offset is not needed any more:

```
ldr   ..,[r9]
ldr   ..,[r9,#4]
ldr   ..,[r9,#8]
ldr   ..,[r9,#12]
sub   r9,r9,#-512
ldr   ..,[r9]
add   r9,r9,#-512
ldr   ..,[r9,#16]
ldr   ..,[r9,#20]
ldr   ..,[r9,#24]
ldr   ..,[r9,#28]
```

Here, all `ldr` instructions are 16-bit, at the cost of one extra `add` instruction. We choose to optimise for speed, because it is easier to implement and this situation is so rare that a different scheme would not give significantly smaller code.

5 Optimising register allocation

5.1 Introduction

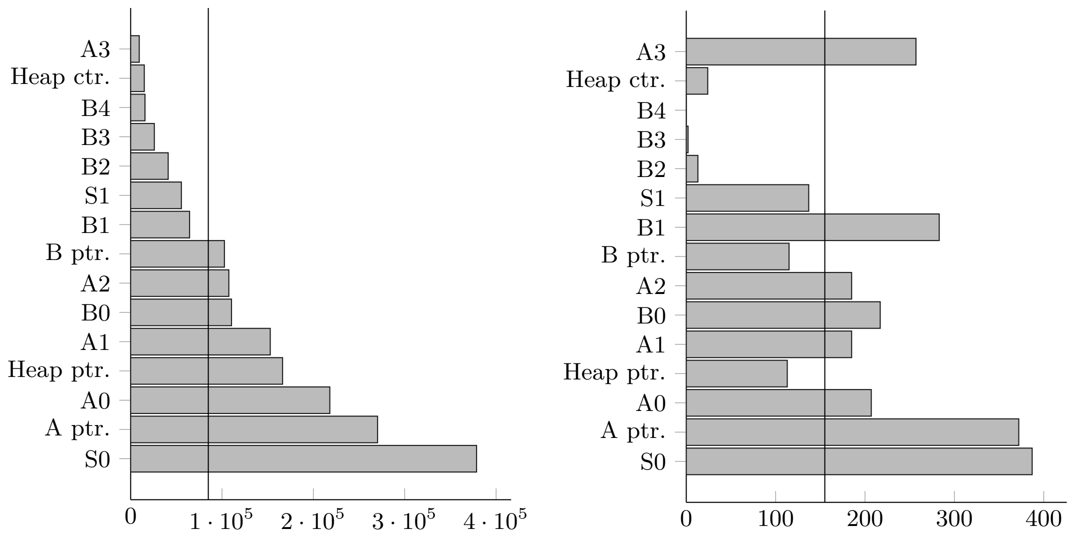
In Thumb-1, i.e., the 16-bit subset of Thumb-2, register fields are 3 bits wide, allowing the programmer to access `r0` through `r7`. Special encoding variants are defined to access `sp` (`r13`) and `pc` (`r15`), though these instructions have limited functionality compared to those working on the eight low registers. The 32-bit instructions defined by Thumb-2 have access to all sixteen registers.

This introduces an interesting code size optimisation vector. If we put the most-used registers in the eight low registers, we may save code size.

5.2 Usage in Clean

In Clean programs, the ARM registers are used as follows [9, `armstartup.s`]:

- Five registers for the top of the B-stack, which we call B0 through B4.
- Four registers for the top of the A-stack, A0 through A3.
- Three registers for the A-stack, B-stack and heap pointers: A ptr., B ptr., Heap ptr., respectively. The B-stack is interleaved with the C-stack and uses SP as its pointer.
- One register for the number of free words on the heap, Heap ctr.
- Two scratch registers S0 and S1.
- The program counter PC.



(a) In the Clean compiler.

(b) In the RTS.

Figure 2: Register usage with the Thumb-2 backend.

In figure 2, we count for each register the number of instructions it occurs in, in the code generated for the Clean compiler [10]. When a register is used multiple times in the same instruction, this counts as one occurrence. The vertical line at indicates the number of occurrences that a variable should have to justify putting it in the lower registers, if we were to put the most-used variables in the lower registers (e.g., there are eight registers with over 85,000 occurrences in the Clean compiler).

The RTS shows an entirely different pattern (figure 2b), because many registers have another meaning in the RTS than in generated code. For large programs, register usage in the RTS is negligible. For small programs it may not be negligible, but these programs are not likely to need rigorous optimisation for code size, considering that they will be small anyway.

The counting method used here is rather simplistic: basing a register allocation on these counts alone means assuming that every instruction has a 16-bit variant, which is not the case. A more accurate method would be to only count those instructions where using a high or low register actually makes a difference. This is much more complicated, and for a rough estimate the simplistic method used here will already allow us to shrink down the code size.

5.3 Optimisation

The register allocation used in the ARM backend is inefficient for Thumb-2 on several points (table 1). A number of often-used registers (the main scratch register, the A-stack pointer and the heap pointer) are in the upper half, while registers that are used less often are in the lower half (the heap counter and B2 through B4).

Figure 2 suggests that we put all variables that occur more than 85,000 times in the low registers. However, the B-stack pointer has to be SP: the B- and C-stacks are combined into one and the C-stack pointer should be the system pointer to allow for an efficient foreign function interface.

The third column in table 1 shows the proposed allocation for Thumb. The heap counter and the A-stack pointer are swapped, as are A2 and B2. The registers for S0, B4, the heap pointer, B3 and A3 have been rotated. This way, all eight most often used registers are in the lower half except the B-stack pointer.

Register	ARM	Thumb-2
A3	`r11`	**`r12`**
Heap counter	`r5`	**`r9`**
B4	`r0`	**`r10`**
B3	`r1`	**`r11`**
B2	`r2`	**`r8`**
S1	`r14`	`r14`
B1	`r3`	`r3`

Register	ARM	Thumb-2
B pointer	`sp`	`sp`
A2	`r8`	**`r2`**
B0	`r4`	`r4`
A1	`r7`	`r7`
Heap pointer	`r10`	**`r1`**
A0	`r6`	`r6`
A pointer	`r9`	**`r5`**
S0	`r12`	**`r0`**

Table 1: Register allocation in the ARM and Thumb backends.

5.4 The foreign function interface

Changing the register allocation has its impact on the foreign function interface. Clean provides mechanisms to export Clean functions, so that they can be called from other software, and to call other functions from Clean, as long as the other software respects a certain high-level infrastructure [11, chp. 11].

The low-level interface (which registers are used, for example) is platform-dependent. For ARM, it is defined in [12].

Some registers have a special function: `r15` is the program counter; `r13` the stack pointer. The Clean backend cannot use these registers in another way. There are four argument / result / scratch registers, `r0` through `r3`. These are not guaranteed to be preserved upon a function call. The link register, `r14`, and `r12`, cannot be used freely either: a subroutine jumps to the address in the link register when it is done (and can use `r14` as a local variable if it stores its value upon entering on the stack); and `r12` can be used by the linker when extra instructions are needed when a branch instruction attempts to jump to a label so far away that it does not fit in the instruction any more [12, p. 5.3.1.1]. For local variables, `r4` through `r8`, `r10` and `r11` can be used: these have to be preserved by subroutines. The last register, `r9`, is platform-dependent. Some quick tests indicate that it is callee-saved on our test setup (see appendix B).

The register allocation in the ARM backend is optimised for the foreign function interface: the B-stack registers are in the argument registers, because the B-stack is usually empty during function calls. The link register `r14` and `r12` are used as scratch registers, because they are only used for short term storage. All other variables have to be kept over subroutine calls and are kept in the other registers, which are callee-saved.

Changing the register allocation in the way proposed above means that this foreign function interface will be less efficient. Whenever a foreign function needs to be called, the caller-saved registers that need to be preserved have to be saved. With the allocation as proposed in table 1, these are the heap pointer (`r1`), A2 (`r2`) and A3 (`r12`). Before every call, a wide `push` instruction needs to be inserted; after every return, a wide `pop` instruction. This introduces an 8-byte overhead per foreign function call and also means that every call will be slightly slower. We deem the foreign function interface to be less important than the actual Clean code, so this is acceptable.

5.5 Results

To measure the code size improvement introduced by this optimisation, we again take the Clean compiler and compile it in three different ways:

- With the existing ARM backend [9].
- With the new Thumb backend, leaving the register allocation as in the ARM backend.
- With the new Thumb backend, with the optimised register allocation as in the third column of table 1.
- With the new Thumb backend, with the optimised register allocation and leaving out any **`.align`** directives to approximate object-code level efficiency.

The object code generator has not been finished yet at the moment these measurements were done, so we can only estimate its efficiency. A modification described in section 2.3 required us to add `.align` to jump instructions, adding a `nop` instruction in approximately half the instances. In the last measurement, we approximate object-code level efficiency by leaving out these `.align` directives, as they will not be needed in the object code generator.

We only measure the size of generated code, that is, all `.text` segments except those that belong to the RTS or external libraries. These measurements were done on the system described in appendix B. One may argue that the Clean compiler is not representative code because of some peculiarities like large lookup tables and arrays that are rarely used in practice. For this reason, we compare both the total code size and the size of a part of the lexer [10, `frontend/scanner.icl`], which can be considered representative. The results are in figure 3.

We see that even without the optimisation discussed in this section, some code size is saved. After all, at least some instructions can be made 16-bit, and in the sections above we have hardly ever had to add extra instructions. However, optimising register allocation allows us to save much more, up to almost 20%.

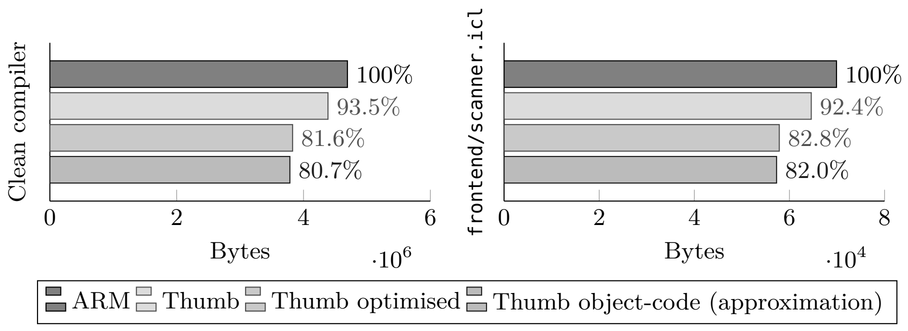


Figure 3: Code size for different backends.

6 Results

6.1 Test suite

We use a standard set of Clean programs to compare ARM and Thumb performance. These programs are included in the `examples` directory of every Clean release [4]. They are:

Ack $A(3, 10)$, where A is the Ackermann function.

E The approximation of e to 2000 digits.

Fib Computing the 35th Fibonacci number using naive recursion.

FSve The sieve of Eratosthenes (optimised), computing the first 50,000 primes.

Ham The Hamming function, computing the first 300 numbers with only 2, 3 and 5 as prime factors.

LQns Computing the possibilities to place 11 queens on an 11×11 'chess' board (not optimised).

Mat Multiplying two 6×6 matrices.

Perm Inverting a permutation of `[1..16]`.

Psc Pretty-printing the first 18 rows of the triangle of Pascal.

Rev Reversing a 10,000-elements list, 10,000 times.

RevTw Reversing a list of 500 elements 65,536 times using the higher-order function `twice` (with a heap size of 500M[8]).

RFib Computing the 35th Fibonacci number using naive recursion and `Reals` instead of `Ints`.

SQns Computing the possibilities to place 11 queens on a 11×11 'chess' board (optimised).

Sve The sieve of Eratosthenes (not optimised), computing the first 3,000 primes.

STw Incrementing an integer 65,536 times using the higher-order function `twice` (optimised).

Tak $\tau(32, 16, 8)$, where τ is the Takeuchi function.

Tw Incrementing an integer 65,536 times using the higher-order function `twice` (not optimised).

WSeq A sequential version of Warshall's shortest path algorithm on a 6×6 matrix.

[8]The large heap size is needed to prevent switching to another garbage collector, which had not yet been finished at the moment of writing.

		Ack	E	Fib	FSve	Ham	LQns	Mat	Perm	Psc	Rev	RevTw	RFib	SQns	Sve	STw	Tak	Tw	WSeq
	ARM (b)	188	2388	164	1036	1576	2708	4452	1756	2940	796	820	248	1660	1200	340	232	360	2628
Size	Thumb (b)	164	1944	148	840	1200	2196	3392	1332	2452	636	632	248	1328	936	304	204	312	2020
	Diff. (−%)	12.8	18.6	9.8	18.9	23.9	18.9	23.8	24.1	16.6	20.1	22.9	0.0	20.0	22.0	10.6	12.1	13.3	23.1
	ARM (s)	0.59	1.39	0.59	1.18	—	1.64	—	—	—	4.09	1.07	1.15	1.16	0.99	—	0.87	—	—
Time	Thumb (s)	0.66	1.44	0.62	1.11	—	1.70	—	—	—	4.31	1.08	1.18	1.22	0.99	—	0.94	—	—
	Diff. (%)	11.9	3.6	5.1	5.9	—	3.7	—	—	—	5.4	0.9	2.6	5.2	0.0	—	8.0	—	—

Table 2: Code size and running time comparison for ARM and Thumb-2.

6.2 Code size

We compared the code size for the code generated by the ARM and Thumb backends for all programs in the test suite. This does not include the run-time system, but does include parts of Clean's standard library, StdEnv, that are necessary for the program. The results are in table 2.

The Thumb backend produces on average 17.3% smaller code, with a standard deviation of 6.3pp. If we ignore those programs for which the ARM code size is less than 1,000 bytes, this is 21.0%, with a standard deviation of 2.6pp., comparable to the 19.3% found for the Clean compiler in section 5.5. Small programs with short function blocks skew the data, because they contain relatively many jumps which are longer in Thumb than in ARM (see section 2).

6.3 Running time

For the same programs, we compare the running time. Some of the examples as provided in the Clean releases have a too short running time to be able to compare them, so the parameters had to be tweaked. In this case, the new parameters are mentioned in section 6.1. For some examples it was not possible to increase the running time enough. Table 2 lists '—' for these programs.

The increase in running time is relatively low compared to the decrease in code size: 4.8% on average with a standard deviation of 3.1pp. For the larger programs, this is 3.7% with a standard deviation of 2.04pp. (though this is based on only five programs).

7 Discussion

In this section, we discuss three optimisation vectors that are not yet considered. First, we look at subroutine calls, which are expensive in Thumb, compared to ARM. In section 7.2, we consider branch optimisation, which concerns moving code blocks around to make as many branch instructions narrow by minimising their offsets. The last optimisation vector is about instructions that exist in Thumb but not in ARM. These are discussed in section 7.3.

We also look at the results from section 6 in more detail, in section 7.4.

7.1 Optimising subroutine calls

In section 2, we saw that subroutine calls are slower and larger in Thumb than in ARM, because an extra `add` instruction has to be inserted. We discussed that a more efficient solution would exploit the link register: upon a `bl` and `blx` instruction, the link register is loaded with the address of the instruction after the branch. If the callee would be responsible for saving this address, rather than the caller, we could delay storing the return address and exploit these instructions to eliminate the `add` instruction. However, we also mentioned that tail recursion makes this difficult (see section 2.6).

7.1.1 Tail recursive entry points

To improve on this, two things can be done. First, we could add a *tail recursive entry point* to function blocks. The code for the `length` function from section 2.6 could then look like this:

```
slength_P2:
      str   lr,[sp,#-4]!    @ Store the return address
slength_P2_tr:              @ The tail recursive entry point
      ldr   r0,[r6]         @ Load top of A-stack (the list)

      @ ... etc

e_0:                        @ If / after the Cons part has been evaluated
      add   r4,r4,#1        @ Increase counter
      b     slength_P2_tr   @ Tail recursively jump back to start of function
      .ltorg
```

This way, the code to store the return address is only executed once. The return address is saved on the stack, as before. To call the subroutine, the following block can be used:

```
bl    slength_P2
```

This is an improvement of between two and six bytes per subroutine call: in any case, we win two bytes since the `add` instruction is eliminated. In the minimal case that a function is called only once, we save only these two bytes. If the function is called often, many `str` instructions can be removed at the cost of one extra `str` instruction in the function itself, and the space saved per subroutine call asymptotically approaches six bytes.

As for running time, we win one instruction cycle per subroutine call. The `add` instruction is removed, but the `str` instruction is still executed (whether it is placed in the caller's or the callee's block does not matter).

In the current code generator, branches are often removed by putting the callee direct behind the caller. Adding a tail recursive entry point before the callee complicates this: we would need to jump behind that entry point or place the entry point somewhere else and jump from the tail recursive entry point to the start of the function. It should be tested if any of these options is an improvement over the current slow subroutine calls.

To implement tail recursive entry points, one would need to be able to recognise tail recursion such that the tail recursive entry point is used only for these subroutine calls, but not for the 'plain' recursive subroutine calls. Also, more research is needed to see if a tail recursive entry point is sufficient in all cases.

7.1.2 Mixed calling convention

A second possibility would be to have a *mixed calling convention.* Currently, in every subroutine call, the caller is responsible for storing the return address on the stack. In a mixed calling convention, the callee would be responsible for this, except for functions that exploit tail recursion or that are not always called but also jumped to (as described above). Recognising tail recursion can be done on a relatively high level of abstraction, so annotations can be added to the intermediate ABC-code to indicate what calling convention should be used for a certain function.

An advantage of this approach might be that there are less cases to consider. The obvious disadvantage is that it makes the calling convention overly complex.

7.2 Branch optimisation

A second optimisation vector that is yet to be considered is branch optimisation. The Thumb instruction set has four variants of the simple branch instruction `b` [2, A6.7.12]:

- 16-bits, conditional, with an offset between -256 and 254.
- 16-bits, not conditional, with an offset between $-2,048$ and $2,046$.
- 32-bits, conditional, with an offset between $-1,048,576$ and $1,048,574$.
- 32-bits, not conditional, with an offset between $-16,777,216$ and $16,777,214$.

By reorganising the code, the code generator could make as many branch instructions as possible 16-bit.

This optimisation is restricted to the `b` instruction: the branch with link (`bl`) is always 32-bit; the branch and exchange (`bx`) and branch with link and exchange (`blx`) instructions always 16-bit. Since the `b` instruction is used relatively little by the code generator, we cannot hope for much improvement. The Clean compiler, with a code size of 3,827,868 bytes (Thumb optimised, figure 3), counts 56,009 simple branches, of which 13752 (25%) are already narrow. In the best case we can make all the wide branches narrow and win $(56,009 - 13,752) \cdot 2 = 84,514$ bytes, that is 2.2%. The `bl`, `blx` and `bx` instructions are used $42,436$, $23,070$ and 374 times in the Clean compiler, respectively.

7.3 Thumb-only instructions

The Thumb instruction set introduces some instructions that are not available on ARM. These include:

- `cbz` and `cbnz`, 'Compare and Branch on (Non-)Zero' can conditionally branch forward an even number of bytes between 0 and 126 when a register is (un)equal to zero [2, A6.7.21]. This is an improvement over combining a `tst` and a `beq` or `bne` instruction.

- `tbb` and `tbh`, 'Table Branch Byte' and 'Table Branch Halfword' can branch forward using a table of single byte or halfword offsets, when given a pointer to and an index in the table [2, A6.7.139].

Where they are applicable, using these functions will be more efficient than the equivalent ARM code, which is being used now. There do not seem to be direct use cases for `tbb` and `tbh`, but `cbz` and `cbnz` can improve performance. It has not been checked yet how much can be won.

The `it` instruction allows for conditional execution in the Thumb instruction set. It does not exist in the ARM instruction set, which has a four-bit conditional field in every instruction (see section 1.1.1). As such, the `it` instruction is new in Thumb. However, it does not introduce a performance optimisation, as there is no slower alternative to `it` blocks.

7.4 Matching programs and instruction sets

If we compare the code size decrease with the running time increase, we get the plot in figure 4. It seems that code size decrease correlates negatively with increase in running time, suggesting that Thumb is more suitable for some programs than others. However, we do not have enough data to see if this is significant. More research is needed to see if there actually is such a relationship, and if so, what programs are more suitable for Thumb.

The outliers in figure 4 are RFib $(0.0, 2.6)$, Ack $(12.8, 11.9)$, Fib $(9.8, 5.1)$ and Tak $(12.1, 8.0)$, all small programs that rely heavily on recursion (for a description of the programs, see section 6.1 above). This is explained by the relatively high number of jumps: in Thumb we had to add several instructions, introducing extra code size and running time (see section 2).

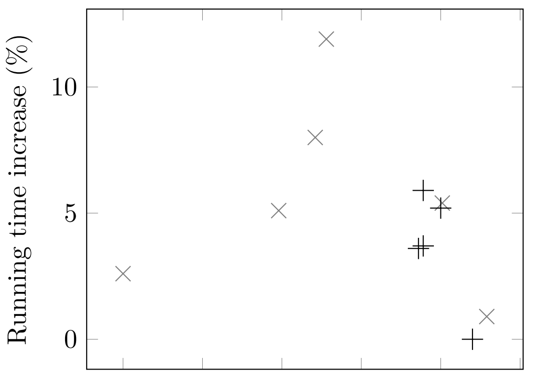


Figure 4: Comparison of code size decrease and running time increase. Grey crosses indicate programs that are smaller than 1000b in ARM; black pluses correspond to larger programs.

This page intentionally left blank.

A Current status

In this section, we briefly discuss the current status of the Thumb backend for Clean.

A.1 Run-time system

The latest version of the RTS can be found at `https://git.camilstaps.nl/clean-run-time-system.git/`. The Thumb backend is located in the `thumb2*` files and can be built with `Makefile.linux_thumb2`.

Register aliases have been used to ease changing the register allocation. The current allocation is set in `thumb2regs.s`.

The Clean RTS has three garbage collectors: a copying, a compacting and a marking collector. At this point only the first works. The expectation is that the others can be fixed rather easily and that only a few bits need to be flipped. We don't expect any problems other than the ones encountered in the copying collector (which has been discussed in section 3).

A.2 Code generator

The latest version of the code generator can be found at `https://git.camilstaps.nl/clean-code-generator.git/`. The Thumb-specific part is in the `cgthumb2*` files and throughout other files in a few `#ifdef` blocks. It can be built with `Makefile.linux_thumb2`.

There are two code generators: one that generates readable assembly code (`thumb2was.c`) and one that generates object code (`thumb2as.c`). At the moment of writing, only the first has been adapted to work for Thumb — the second is the same as the ARM code generator.

A.3 Building programs

To build a file `mymodule.icl` for a Thumb target, the following workflow can be used:

```
# Build _system, needed only once
cg _system -s _system.s
as -o _system.o _system.s -march=armv7-a

# Build cgopts, needed only once
as -o cgopts.o cgopts.s -march=armv7-a

# Build actual program
clm -ABC mymodule
cg Clean\ System\ Files/mymodule -s mymodule.s
as -o mymodule.o mymodule.s -march=armv7-a
cc -o mymodule \
      /path/to/rts/_startup.o \
      /path/to/_system.o \
      cgopts.o \
      mymodule.o \
      -lc -lm \
      -march=armv7-a
```

The `_system.abc` that is needed can be taken from any Clean distribution [4].

In `cgopts.s`, some variables are set that are normally added by the Clean make tool `clm`. The file may look like this:

```
        .data
        .global ab_stack_size
        .global flags
        .global heap_size
        .global heap_size_multiple
        .global initial_heap_size

heap_size:          .word 0x00200000
ab_stack_size:      .word 0x00080000
flags:              .word 0x00000008
heap_size_multiple: .word 0x00001400
initial_heap_size:  .word 0x00019000
```

Another option to build Clean programs using the Thumb backend is to get the Clean make tool `clm` from `https://svn.cs.ru.nl/repos/clean-tools/trunk/clm/` and build it so that `NO_ASSEMBLE` is *not* defined (which will cause it to execute the readable code generator instead of the object code generator).

B System setup

Tests were run on a Raspberry Pi 3, Model B with 1GB RAM. Its processor is listed in `/proc/cpuinfo` as an 'ARMv7 Processor rev 4 (v7l)', although the Hardware field is 'BCM2709', which is an ARMv8 chip.

To build programs, we used the following suite:

- gcc (Raspbian 4.9.2-10) 4.9.2
- GNU assembler (GNU Binutils for Raspbian) 2.25
- GNU ld (GNU Binutils for Raspbian) 2.25
- A Clean compiler built from intermediate ABC files retrieved on December 7, 2016 [10], built with the ARM code generator revision 295 [9] and the ARM RTS revision 387 [8].

These programs are given `-march=armv7-a` when applicable to optimise for the ARMv7 architecture. The Thumb backend proposed in this thesis uses features that are deprecated by ARMv8-A (see section 1.1.1).

References

[1] ARM Ltd., *Arm architecture reference manual. armv7-a and armv7-r edition*, 1996.

[2] ——, *Armv7-m architecture reference manual*, 2006.

[3] ——, *Arm architecture reference manual. armv8, for armv8-a architecture profile (beta)*, 2013.

[4] Software Technology Research Group. (2004). The clean home page, Radboud University Nijmegen, [Online]. Available: `http://clean.cs.ru.nl/` (Retrieved 15/11/2016).

[5] H. Barendregt, M. van Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer and M. Sleep, 'Lean: An intermediate language based on graph rewriting', *Parallel Computing 9*, 1988.

[6] P. Koopman, 'Functional programs as executable specifications', PhD thesis, Radboud University Nijmegen, 1990.

[7] J. van Groningen, 'Implementing the abc-machine on m680x0 based architectures', Master's thesis, Radboud University Nijmegen, 1990.

[8] Software Technology Research Group. (2016). Clean's arm run-time system, Revision 387, Radboud University Nijmegen, [Online]. Available: `https://svn.cs.ru.nl/repos/clean-run-time-system` (Retrieved 15/11/2016).

[9] ——, (2016). Clean's arm code generator, Revision 295, Radboud University Nijmegen, [Online]. Available: `https://svn.cs.ru.nl/repos/clean-code-generator` (Retrieved 15/11/2016).

[10] ——, (2016). The clean compiler, Bootstrap from intermediate abc files, 32-bit., Radboud University Nijmegen, [Online]. Available: `http://clean.cs.ru.nl/Download_Clean` (Retrieved 07/12/2016).

[11] R. Plasmeijer, M. van Eekelen and J. van Groningen, *Clean language report, Version 2.2*, Department of Software Technology, University of Nijmegen, 2011.

[12] ARM Ltd., *Procedure call standard for the arm architecture*, 2015.