A closer look at ARM code quality

Tilmann Scheller
LLVM Compiler Engineer
t.scheller@samsung.com

Samsung Open Source Group
Samsung Research UK

2014 LLVM Developers' Meeting
San Jose, USA, October 28 – 29, 2014

Overview

- Introduction
- ARM architecture
- Performance
- Case study
- Summary

Introduction

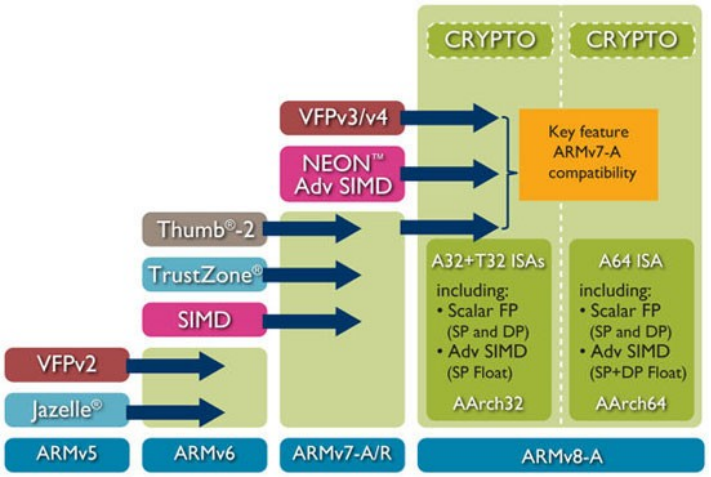
Introduction

- Find out how we are doing on ARM
- Comparison against GCC
- Pick a benchmark and compare the generated assembly code
- Try to find out what we need to change in LLVM to get better performance

ARM architecture

ARM architecture

- 32-bit/64-bit RISC architecture
- Load-store architecture
- Barrel shifter: **add r4, r3, r6, lsl #4**
- Powerful indexed addressing modes: **ldr r0, [r1, #4]!**
- Predication: **ldreq r3, [r4]**
- Family of 32-bit instruction sets evolved over time: ARM, Thumb, Thumb-2
- Focus on the Thumb-2 instruction set in this talk
- Instruction set extensions:
  - VFP
  - Advanced SIMD (NEON)



Thumb-2 ISA

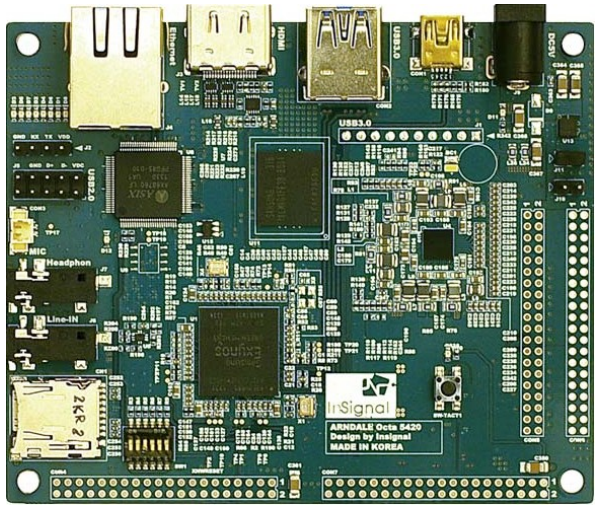
- Goal: Code density similar to Thumb, performance like original ARM instruction set
- Variable-length instructions (16-bit/32-bit)
- 16 32-bit GPRs (including PC and SP)
- 16 or 32 64-bit floating-point registers for VFP/NEON
- Conditional execution with IT (if-then) instruction

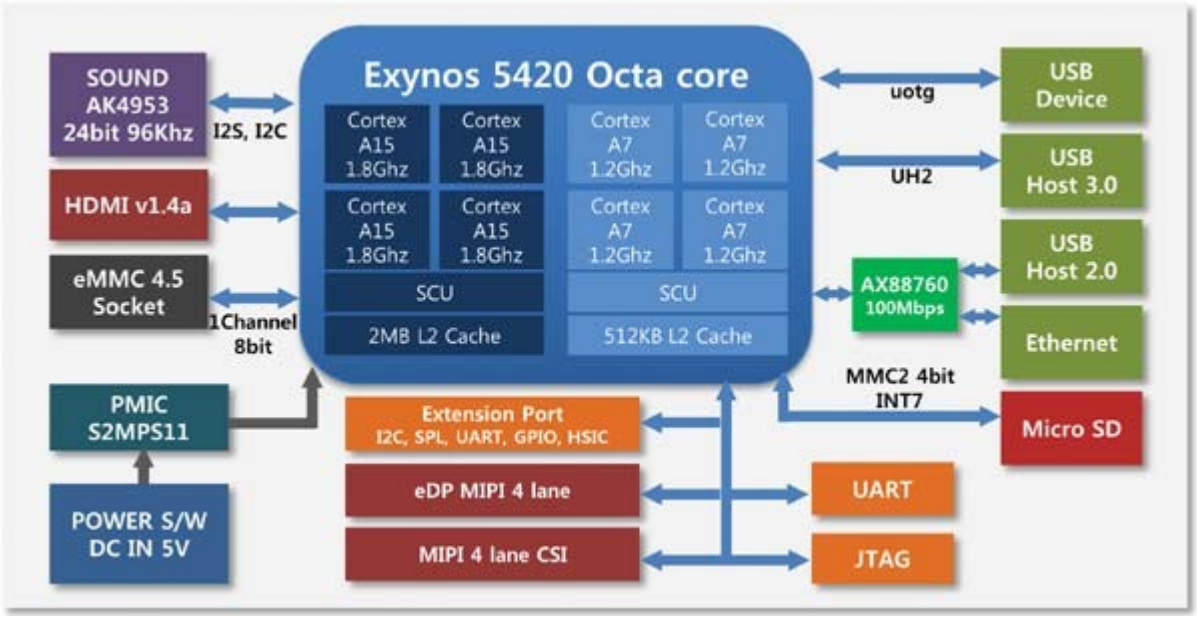
```
; if (r0 == r1)
cmp r0, r1
ite eq          ; ARM: no code ... Thumb: IT instruction
; then r0 = r2;
moveq r0, r2   ; ARM: conditional; Thumb: condition via ITE 'T' (then)
; else r0 = r3;
movne r0, r3   ; ARM: conditional; Thumb: condition via ITE 'E' (else)
; recall that the Thumb MOV instruction has no bits to encode "EQ" or "NE"
```

Performance

Hardware

- Arndale Octa board
- Cortex-A15 clocked at 1.8GHz
- 2GB of RAM
- Ubuntu 14.04 provided by Linaro



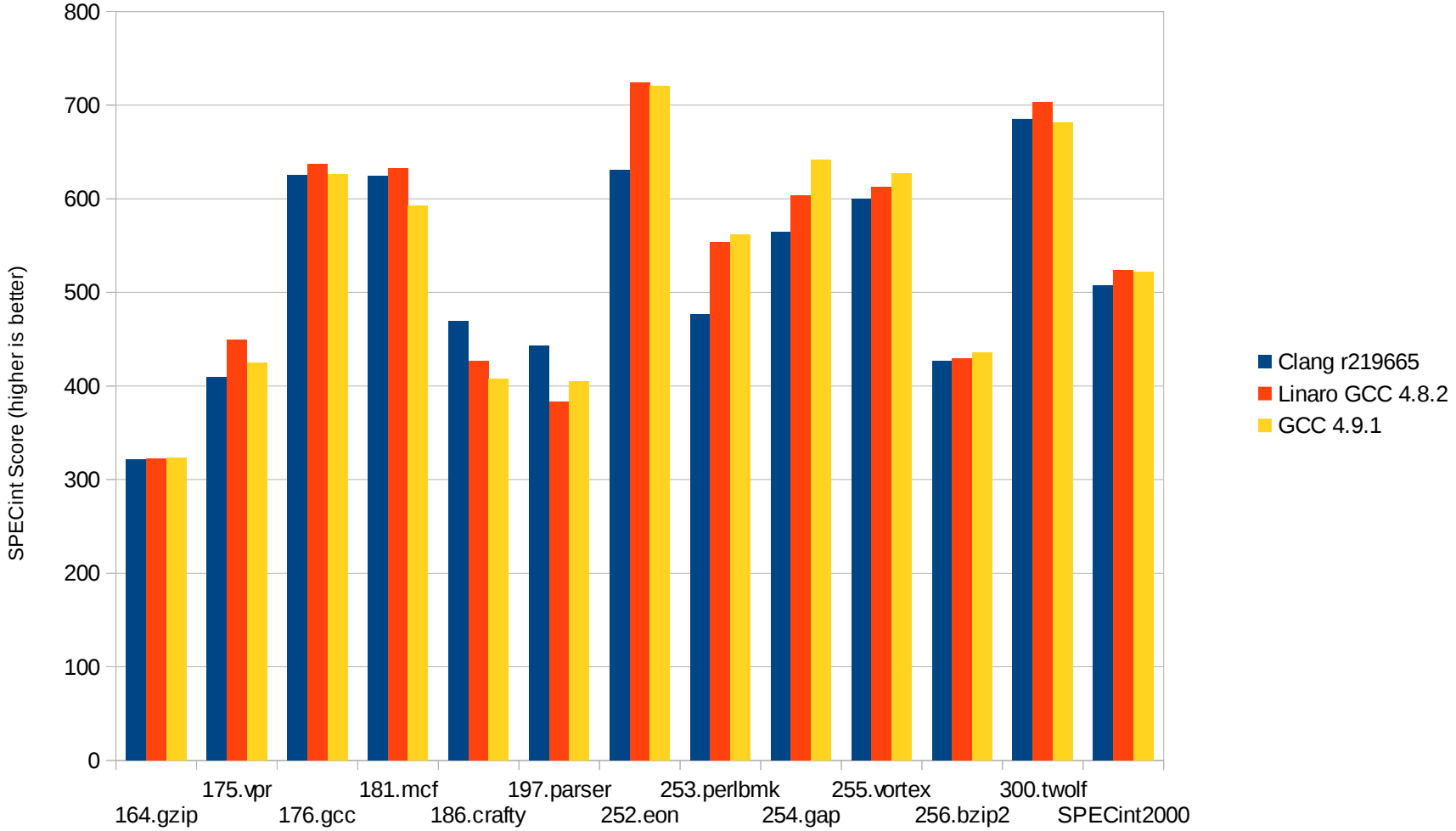


Preparations

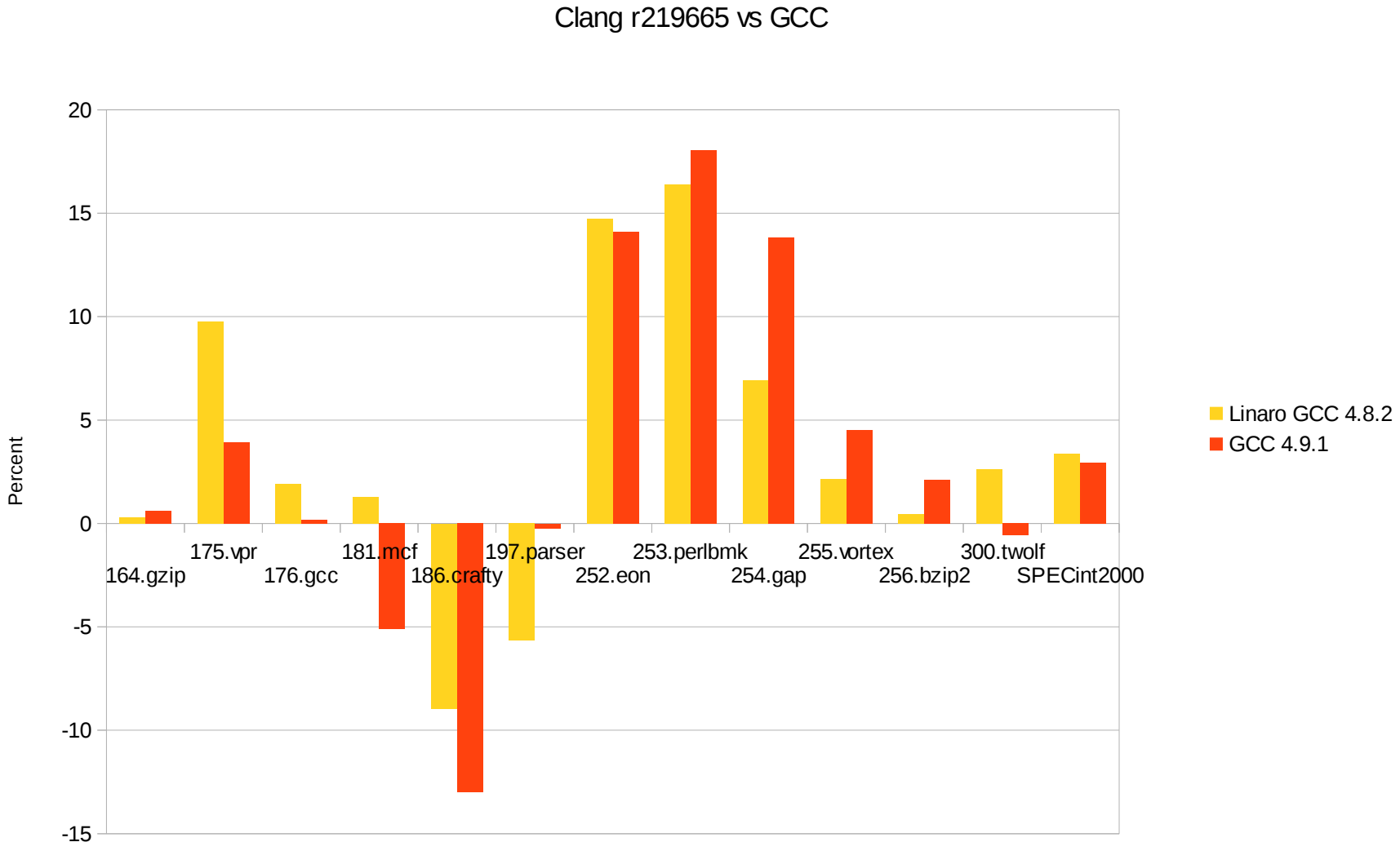
- Getting stable results:
  - Kill all unneeded services
  - Disable cron jobs
  - Turn off frequency scaling
  - Disable ASLR
  - Turn off all cores except one
  - Put benchmark into RAM disk
  - Static builds

SPEC CPU2000

-mcpu=cortex-a15 -mfpu=neon-vfpv4 -O3



SPEC CPU2000



SPEC CPU2000

- On average GCC is just ~3% faster
- Four benchmarks where GCC is doing significantly better: 175.vpr, 252.eon, 253.perlbmk, 254.gap
- 254.gap relies on signed overflow, needs to be compiled with -fwrapv
- Let's have a closer look at 175.vpr

Case study

175.vpr

- VPR = **V**ersatile **P**lace and **R**oute
- FPGA circuit placement and routing
- Simulated annealing, graph algorithms
- Two invocations one for place, one for route
  - Place: 6.49% slowdown
  - Route: 10.46% slowdown
- Open source

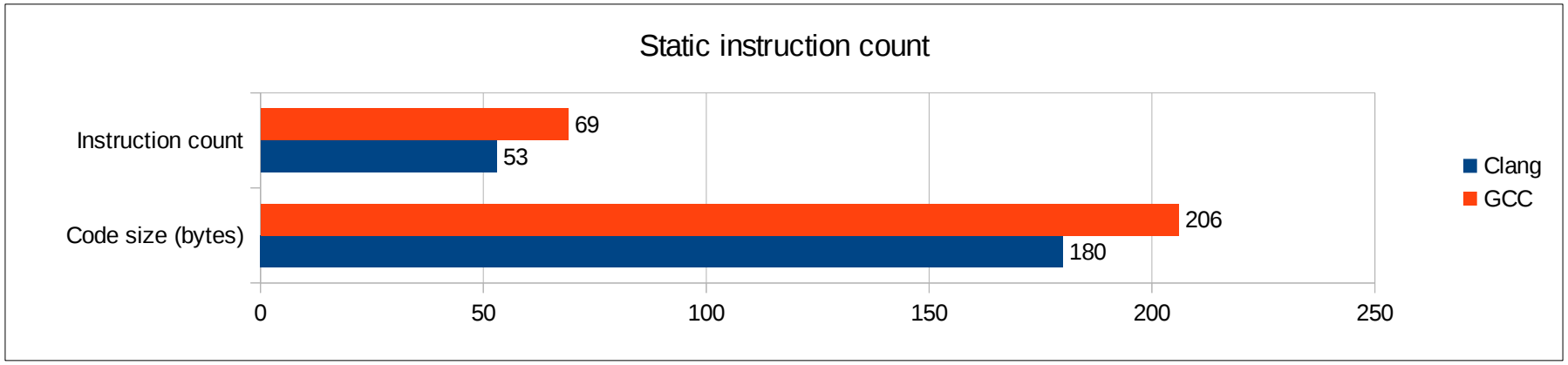
More information about 175.vpr at http://www.spec.org/cpu2000/CINT2000/175.vpr/docs/175.vpr.html

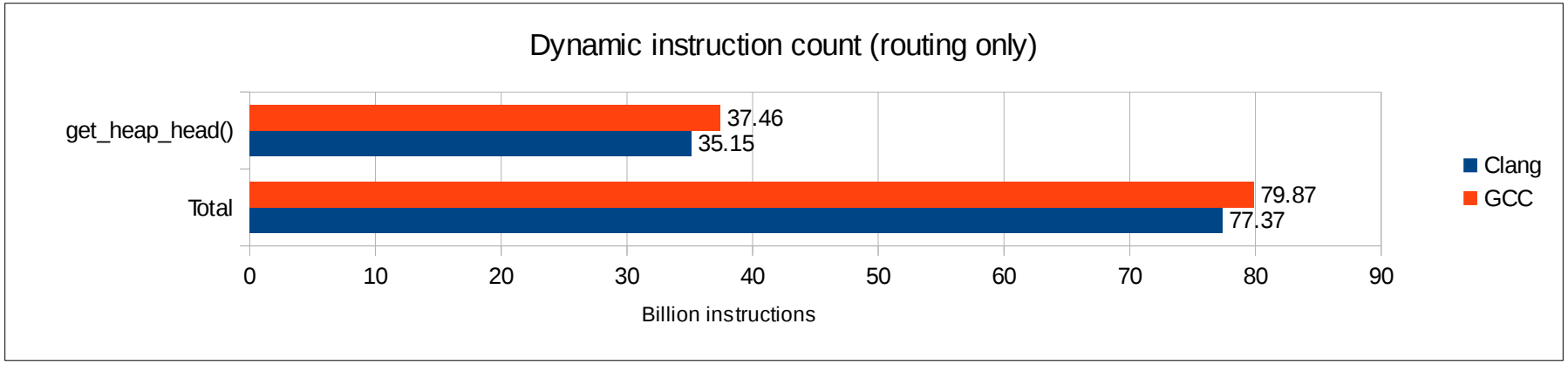
175.vpr

- Measuring against GCC 4.8.2 as it generates better code for 175.vpr than GCC 4.9.1
- Built with: -mcpu=cortex-a15 -O3 -fno-inline -fno-vectorize
- ~83% of the time spent in the top three functions

```
 46.70% get_heap_head
 23.94% expand_neighbours
 11.89% add_to_heap
  4.58% route_net
  3.69% node_to_heap
  3.19% alloc_heap_data
  1.68% free_heap_data
  0.94% reset_path_costs
  0.91% alloc_linked_f_pointer
  0.66% empty_heap
...
```

Some metrics for get_heap_head()





GCC is executing ~2 billion more instructions but they take less time to execute

175.vpr - get_heap_head()

```
struct s_heap *get_heap_head (void) {
/* Returns the smallest element on the heap. */
 int ito, ifrom;
 struct s_heap *heap_head, *temp_ptr;

 do {
  if (heap_tail == 1) {    /* Empty heap. */
   printf("Error:  Empty heap...
   exit(1);
  }
  heap_head = heap[1]; /* Smallest element. */

  /* Now fix up the heap */
  heap_tail--;
  heap[1] = heap[heap_tail];
  ifrom = 1;
  ito = 2*ifrom;

  while (ito < heap_tail) {
    if (heap[ito+1]->cost < heap[ito]->cost)
      ito++;
    if (heap[ito]->cost > heap[ifrom]->cost)
      break;
    temp_ptr = heap[ito];
    heap[ito] = heap[ifrom];
    heap[ifrom] = temp_ptr;
    ifrom = ito;
    ito = 2*ifrom;
  }
 /* Get another one if invalid entry. */
 } while (heap_head->index == OPEN);
 return(heap_head);
}
```

```
Globals:

/* Used by the heap as its fundamental
   data structure. */
struct s_heap {...; float cost; ...};

/* Indexed from [1..heap_size] */
static struct s_heap **heap;

/* Index of first unused slot in the
   heap array */
static int heap_tail;
```

All sources from VPR 4.22 at
http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html

175.vpr – get_heap_head() - Clang

```
get_heap_head:
 push.w {r4, r5, r6, r7,
         r11, lr}
 add    r7, sp, #12
 movw   r12, :lower16:MG
 movt   r12, :upper16:MG
 ldr.w  lr, [r12, #8]
L1:
 cmp.w  lr, #1
 beq    L4

 ldr.w  r1, [r12, #4]
 sub.w  lr, lr, #1
 cmp.w  lr, #3
 ldr    r0, [r1, #4]
 str.w  lr, [r12, #8]
 ldr.w  r2, [r1, lr, lsl #2]
 str    r2, [r1, #4]
 blt    L3

 movs   r2, #1
 movs   r3, #2
```

```
L2:
 ldr.w  r4, [r12, #4]
 orr    r1, r3, #1
 ldr.w  r5, [r4, r3, lsl #2]
 ldr.w  r6, [r4, r1, lsl #2]
 vldr   s0, [r5, #4]
 vldr   s2, [r6, #4]
 vcmpe  s2, s0
 vmrs   APSR_nzcv, fpscr
 it     pl
 movpl  r1, r3
 ldr.w  r5, [r4, r2, lsl #2]
 ldr.w  r3, [r4, r1, lsl #2]
 vldr   s0, [r5, #4]
 vldr   s2, [r3, #4]
 vcmpe  s2, s0
 vmrs   APSR_nzcv, fpscr
 bgt    L3

 str.w  r5, [r4, r1, lsl #2]
 ldr.w  r4, [r12, #4]
 str.w  r3, [r4, r2, lsl #2]
 lsl.w  r3, r1, #1
 mov    r2, r1
 cmp    r3, lr
 blt    L2
```

```
L3:
 ldr    r1, [r0]
 cmp.w  r1, #-1
 beq    L1

 pop.w  {r4, r5, r6, r7, r11,
         pc}
L4:
 movw   r0, :lower16:.Lstr35
 movt   r0, :upper16:.Lstr35
 bl     puts
 movw   r0, :lower16:.Lstr36
 movt   r0, :upper16:.Lstr36
 bl     puts
 movs   r0, #0
 pop.w  {r4, r5, r6, r7, r11,
         pc}
```

175.vpr – get_heap_head() - GCC

```
get_heap_head:
 movw    r12, #:lower16:MG
 strd    r3, r4, [sp, #-32]!
 movt    r12, #:upper16:MG
 strd    r9, lr, [sp, #24]
 ldrd    r2, r3, [r12, #4]
 strd    r5, r6, [sp, #8]
 strd    r7, r8, [sp, #16]
 cmp     r2, #1
 add     lr, r3, r2, lsl #2
 beq     L6
L1:
 ldr     r1, [lr, #-4]!
 subs    r0, r2, #1
 cmp     r0, #2
 ldr     r8, [r3, #4]
 itt     gt
 movgt   r6, #1
 movgt   r2, #2
 str     r1, [r3, #4]
 bgt     L3
 b       L5
L2:
 cmp     r0, r7
 str     r4, [r5]
 str     r1, [r3, r6, lsl #2]
 mov     r6, r2
 mov     r2, r7
 ble     L5
```

```
L3:
 adds    r4, r2, #1
 lsls    r7, r4, #2
 ldr     r9, [r3, r4, lsl #2]
 subs    r5, r7, #4
 ldr     r1, [r3, r5]
 add     r5, r5, r3
 vldr    s14, [r9, #4]
 vldr    s15, [r1, #4]
 vcmpe   s14, s15
 vmrs    APSR_nzcv, fpscr
 bpl     L4
 vmov    s15, s14
 mov     r2, r4
 adds    r5, r3, r7
 mov     r1, r9
L4:
 ldr     r4, [r3, r6, lsl #2]
 lsls    r7, r2, #1
 vldr    s14, [r4, #4]
 vcmpe   s14, s15
 vmrs    APSR_nzcv, fpscr
 bpl     L2
```

```
L5:
 ldr     r2, [r8]
 adds    r2, r2, #1
 bne     L8
 mov     r2, r0
 cmp     r2, #1
 bne     L1
L6:
 movw    r0, #:lower16:.LC7
 str     r2, [r12, #4]
 movt    r0, #:upper16:.LC7
 bl      puts
 movw    r0, #:lower16:.LC8
 movt    r0, #:upper16:.LC8
 bl      puts
 movs    r0, #0
L7:
 ldrd    r3, r4, [sp]
 ldrd    r5, r6, [sp, #8]
 ldrd    r7, r8, [sp, #16]
 add     sp, sp, #24
 pop     {r9, pc}
L8:
 str     r0, [r12, #4]
 mov     r0, r8
 b       L7
```

175.vpr – get_heap_head()

Clang:

```
 push.w {r4, r5, r6, r7, r11, lr}
 add    r7, sp, #12
 movw   r12, :lower16:MG
 movt   r12, :upper16:MG
 // lr = heap_tail
 ldr.w  lr, [r12, #8]
L1:
 // if (heap_tail == 1)
 cmp.w  lr, #1
 beq    L4

 // r1 = heap
 ldr.w  r1, [r12, #4]
 // heap_tail--
 sub.w  lr, lr, #1
 cmp.w  lr, #3
 // r0 = heap[1]
 ldr    r0, [r1, #4]
 // Update heap_tail in memory.
 str.w  lr, [r12, #8]
 // r2 = heap[heap_tail]
 ldr.w  r2, [r1, lr, lsl #2]
 // heap[1] = heap[heap_tail]
 str    r2, [r1, #4]
 blt    L3

 movs   r2, #1 // ifrom = 1
 movs   r3, #2 // ito = 2*ifrom
```

GCC:

```
 movw   r12, #:lower16:MG
 strd   r3, r4, [sp, #-32]!
 movt   r12, #:upper16:MG
 strd   r9, lr, [sp, #24]
 // r2 = heap_tail, r3 = heap
 ldrd   r2, r3, [r12, #4]
 strd   r5, r6, [sp, #8]
 strd   r7, r8, [sp, #16]
 // if (heap_tail == 1)
 cmp    r2, #1
 add    lr, r3, r2, lsl #2
 // lr = heap[heap_tail]
 beq    L6
L1:
 // r1 = heap[heap_tail--]
 ldr    r1, [lr, #-4]!
 // r0 = heap_tail--
 subs   r0, r2, #1
 cmp    r0, #2
 // r8 = heap[1]
 ldr    r8, [r3, #4]
 itt    gt
 movgt  r6, #1 // ifrom = 1
 movgt  r2, #2 // ito = 2*ifrom
 // heap[1] = heap[heap_tail]
 str    r1, [r3, #4]
 bgt    L3
 b      L5
```

175.vpr – get_heap_head()

Clang:

```
 push.w {r4, r5, r6, r7, r11, lr}
 add    r7, sp, #12
 movw   r12, :lower16:MG
 movt   r12, :upper16:MG
 // lr = heap_tail
 ldr.w  lr, [r12, #8]
L1:
 // if (heap_tail == 1)
 cmp.w  lr, #1
 beq    L4

 // r1 = heap
 ldr.w  r1, [r12, #4]
 // heap_tail--
 sub.w  lr, lr, #1
 cmp.w  lr, #3
 // r0 = heap[1]
 ldr    r0, [r1, #4]
 // Update heap_tail in memory.
 str.w  lr, [r12, #8]
 // r2 = heap[heap_tail]
 ldr.w  r2, [r1, lr, lsl #2]
 // heap[1] = heap[heap_tail]
 str    r2, [r1, #4]
 blt    L3

 movs   r2, #1 // ifrom = 1
 movs   r3, #2 // ito = 2*ifrom
```

GCC:

```
 movw   r12, #:lower16:MG
 strd   r3, r4, [sp, #-32]!
 movt   r12, #:upper16:MG
 strd   r9, lr, [sp, #24]
 // r2 = heap_tail, r3 = heap
 ldrd   r2, r3, [r12, #4]
 strd   r5, r6, [sp, #
 strd   r7, r8, [sp, #
 // if (heap_tail == 1
 cmp    r2, #1
 add    lr, r3, r2, ls
 // lr = heap[heap_tai
 beq    L6
L1:
 // r1 = heap[heap_tai
 ldr    r1, [lr, #-4]
 // r0 = heap_tail--
 subs   r0, r2, #1
 cmp    r0, #2
 // r8 = heap[1]
 ldr    r8, [r3, #4]
 itt    gt
 movgt  r6, #1 // ifrom = 1
 movgt  r2, #2 // ito = 2*ifrom
 // heap[1] = heap[heap_tail]
 str    r1, [r3, #4]
 bgt    L3
 b      L5
```

```
/* Empty heap. */
 if (heap_tail == 1) {
    printf("...
    exit(1);
 }

/* Smallest element. */
 heap_head = heap[1];

/* Now fix up the heap */
 heap_tail--;
 heap[1] = heap[heap_tail];
 ifrom = 1;
 ito = 2*ifrom;
```

175.vpr – get_heap_head()

Clang:

```
 push.w {r4, r5, r6, r7, r11, lr}
 add    r7, sp, #12
 movw   r12, :lower16:MG
 movt   r12, :upper16:MG
 // lr = heap_tail
 ldr.w  lr, [r12, #8]
L1:
 // if (heap_tail == 1)
 cmp.w  lr, #1
 beq    L4

 // r1 = heap
 ldr.w  r1, [r12, #4]
 // heap_tail--
 sub.w  lr, lr, #1
 cmp.w  lr, #3
 // r0 = heap[1]
 ldr    r0, [r1, #4]
 // Update heap_tail in memory.
 str.w  lr, [r12, #8]
 // r2 = heap[heap_tail]
 ldr.w  r2, [r1, lr, lsl #2]
 // heap[1] = heap[heap_tail]
 str    r2, [r1, #4]
 blt    L3

 movs   r2, #1 // ifrom = 1
 movs   r3, #2 // ito = 2*ifrom
```

GCC:

```
 movw   r12, #:lower16:MG
 strd   r3, r4, [sp, #-32]!
 movt   r12, #:upper16:MG
 strd   r9, lr, [sp, #24]
 // r2 = heap_tail, r3 = heap
 ldrd   r2, r3, [r12, #4]
 strd   r5, r6, [sp, #8]
 strd   r7, r8, [sp, #16]
 // if (heap_tail == 1)
 cmp    r2, #1
 add    lr, r3, r2, lsl #2
 // lr = heap[heap_tail]
 beq    L6
L1:
 // r1 = heap[heap_tail--]
 ldr    r1, [lr, #-4]!
 // r0 = heap_tail--
 subs   r0, r2, #1
 cmp    r0, #2
 // r8 = heap[1]
 ldr    r8, [r3, #4]
 itt    gt
 movgt  r6, #1 // ifrom = 1
 movgt  r2, #2 // ito = 2*ifrom
 // heap[1] = heap[heap_tail]
 str    r1, [r3, #4]
 bgt    L3
 b      L5
```

175.vpr – get_heap_head()

Clang:

```
 push.w {r4, r5, r6, r7, r11, lr}
 add    r7, sp, #12
 movw   r12, :lower16:MG
 movt   r12, :upper16:MG
 // lr = heap_tail
 ldr.w  lr, [r12, #8]
L1:
 // if (heap_tail == 1)
 cmp.w  lr, #1
 beq    L4

 // r1 = heap
 ldr.w  r1, [r12, #4]
 // heap_tail--
 sub.w  lr, lr, #1
 cmp.w  lr, #3
 // r0 = heap[1]
 ldr    r0, [r1, #4]
 // Update heap_tail in memory.
 str.w  lr, [r12, #8]
 // r2 = heap[heap_tail]
 ldr.w  r2, [r1, lr, lsl #2]
 // heap[1] = heap[heap_tail]
 str    r2, [r1, #4]
 blt    L3

 movs   r2, #1 // ifrom = 1
 movs   r3, #2 // ito = 2*ifrom
```

GCC:

```
 movw   r12, #:lower16:MG
 strd   r3, r4, [sp, #-32]!
 movt   r12, #:upper16:MG
 strd   r9, lr, [sp, #24]
 // r2 = heap_tail, r3 = heap
 ldrd   r2, r3, [r12, #4]
 strd   r5, r6, [sp, #8]
 strd   r7, r8, [sp, #16]
 // if (heap_tail == 1)
 cmp    r2, #1
 add    lr, r3, r2, lsl #2
 // lr = heap[heap_tail]
 beq    L6
L1:
 // r1 = heap[heap_tail--]
 ldr    r1, [lr, #-4]!
 // r0 = heap_tail--
 subs   r0, r2, #1
 cmp    r0, #2
 // r8 = heap[1]
 ldr    r8, [r3, #4]
 itt    gt
 movgt  r6, #1 // ifrom = 1
 movgt  r2, #2 // ito = 2*ifrom
 // heap[1] = heap[heap_tail]
 str    r1, [r3, #4]
 bgt    L3
 b      L5
```

175.vpr – get_heap_head()

```
Clang

L2:
 // r4 = heap
 ldr.w  r4, [r12, #4]
 // r1 = ito+1
 orr    r1, r3, #1
 // r5 = heap[ito]
 ldr.w  r5, [r4, r3, lsl #2]
 // r6 = heap[ito+1]
 ldr.w  r6, [r4, r1, lsl #2]
 // s0 = heap[ito]->cost
 vldr   s0, [r5, #4]
 // s2 = heap[ito+1]->cost
 vldr   s2, [r6, #4]
 vcmpe  s2, s0
 vmrs   APSR_nzcv, fpscr
 it     pl
 movpl  r1, r3 // r1 = ito
 // r5 = heap[ifrom]
 ldr.w  r5, [r4, r2, lsl #2]
 // r3 = heap[ito]
 ldr.w  r3, [r4, r1, lsl #2]
 // s0 = heap[ifrom]->cost
 vldr   s0, [r5, #4]
 // s2 = heap[ito]->cost
 vldr   s2, [r3, #4]
 vcmpe  s2, s0
 vmrs   APSR_nzcv, fpscr
 bgt    L3
```

```
GCC

L3:
 adds   r4, r2, #1    // r4 = ito+1
 lsls   r7, r4, #2     // r7 = (ito+1)*4
 // r9 = heap[ito+1]
 ldr    r9, [r3, r4, lsl #2]
 subs   r5, r7, #4     // r5 = (ito)*4
 ldr    r1, [r3, r5]   // r1 = heap[ito]
 add    r5, r5, r3     // r5 = &(heap[ito])
 vldr   s14, [r9, #4] // s14 = heap[ito+1]->cost
 vldr   s15, [r1, #4] // s15 = heap[ito]->cost
 vcmpe  s14, s15
 vmrs   APSR_nzcv, fpscr
 bpl    L4
 // ito++
 vmov   s15, s14    // s15 = heap[ito+1]->cost
 mov    r2, r4      // r2 = ito+1
 adds   r5, r3, r7 // r5 = &(heap[ito+1])
 mov    r1, r9      // r1 = heap[ito+1]
L4:
 // r4 = heap[ifrom]
 ldr    r4, [r3, r6, lsl #2]
 lsls   r7, r2, #1     // r7 = 2*ifrom
 vldr   s14, [r4, #4] // s15 = heap[ifrom]->cost
 vcmpe  s14, s15
 vmrs   APSR_nzcv, fpscr
 bpl    L2 // Swap heap[ito] and heap[ifrom].
```

175.vpr – get_h

```
while (ito < heap_tail) {
 if (heap[ito+1]->cost < heap[ito]->cost)
   ito++;
 if (heap[ito]->cost > heap[ifrom]->cost)
   break;
 ...
}
```

Clang

```
L2:
 // r4 = heap
 ldr.w  r4, [r12, #4]
 // r1 = ito+1
 orr    r1, r3, #1
 // r5 = heap[ito]
 ldr.w  r5, [r4, r3, lsl #2]
 // r6 = heap[ito+1]
 ldr.w  r6, [r4, r1, lsl #2]
 // s0 = heap[ito]->cost
 vldr   s0, [r5, #4]
 // s2 = heap[ito+1]->cost
 vldr   s2, [r6, #4]
 vcmpe  s2, s0
 vmrs   APSR_nzcv, fpscr
 it     pl
 movpl  r1, r3 // r1 = ito
 // r5 = heap[ifrom]
 ldr.w  r5, [r4, r2, lsl #2]
 // r3 = heap[ito]
 ldr.w  r3, [r4, r1, lsl #2]
 // s0 = heap[ifrom]->cost
 vldr   s0, [r5, #4]
 // s2 = heap[ito]->cost
 vldr   s2, [r3, #4]
 vcmpe  s2, s0
 vmrs   APSR_nzcv, fpscr
 bgt    L3
```

```
L3:
 adds   r4, r2, #1    // r4 = ito+1
 lsls   r7, r4, #2    // r7 = (ito+1)*4
 // r9 = heap[ito+1]
 ldr    r9, [r3, r4, lsl #2]
 subs   r5, r7, #4    // r5 = (ito)*4
 ldr    r1, [r3, r5]  // r1 = heap[ito]
 add    r5, r5, r3    // r5 = &(heap[ito])
 vldr   s14, [r9, #4] // s14 = heap[ito+1]->cost
 vldr   s15, [r1, #4] // s15 = heap[ito]->cost
 vcmpe  s14, s15
 vmrs   APSR_nzcv, fpscr
 bpl    L4
 // ito++
 vmov   s15, s14   // s15 = heap[ito+1]->cost
 mov    r2, r4     // r2 = ito+1
 adds   r5, r3, r7 // r5 = &(heap[ito+1])
 mov    r1, r9     // r1 = heap[ito+1]
L4:
 // r4 = heap[ifrom]
 ldr    r4, [r3, r6, lsl #2]
 lsls   r7, r2, #1    // r7 = 2*ifrom
 vldr   s14, [r4, #4] // s15 = heap[ifrom]->cost
 vcmpe  s14, s15
 vmrs   APSR_nzcv, fpscr
 bpl    L2 // Swap heap[ito] and heap[ifrom].
```

175.vpr – get_heap_head()

```
...
if (heap[ito+1]->cost < heap[ito]->cost)
        ito++;
if (heap[ito]->cost > heap[ifrom]->cost)
        break;

temp_ptr = heap[ito];
heap[ito] = heap[ifrom];
heap[ifrom] = temp_ptr;
...
```

Clang

```
L2:
 // r4 = heap
 ldr.w  r4, [r12, #
 // r1 = ito+1
 orr    r1, r3, #1
 // r5 = heap[ito]
 ldr.w  r5, [r4, r3
 // r6 = heap[ito+1
 ldr.w  r6, [r4, r1, lsl #2]
 // s0 = heap[ito]->cost
 vldr   s0, [r5, #4]
 // s2 = heap[ito+1]->cost
 vldr   s2, [r6, #4]
 vcmpe  s2, s0
 vmrs   APSR_nzcv, fpscr
 it     pl
 movpl  r1, r3 // r1 = ito
 // r5 = heap[ifrom]
 ldr.w  r5, [r4, r2, lsl #2]
 // r3 = heap[ito]
 ldr.w  r3, [r4, r1, lsl #2]
 // s0 = heap[ifrom]->cost
 vldr   s0, [r5, #4]
 // s2 = heap[ito]->cost
 vldr   s2, [r3, #4]
 vcmpe  s2, s0
 vmrs   APSR_nzcv, fpscr
 bgt    L3
```

```
                                    r4 = ito+1
                                    r7 = (ito+1)*4

                                    #2]
                                    r5 = (ito)*4
                                    r1 = heap[ito]
 add    r5, r5, r3     // r5 = &(heap[ito])
 vldr   s14, [r9, #4]  // s14 = heap[ito+1]->cost
 vldr   s15, [r1, #4]  // s15 = heap[ito]->cost
 vcmpe  s14, s15
 vmrs   APSR_nzcv, fpscr
 bpl    L4
 // ito++
 vmov   s15, s14    // s15 = heap[ito+1]->cost
 mov    r2, r4      // r2 = ito+1
 adds   r5, r3, r7  // r5 = &(heap[ito+1])
 mov    r1, r9      // r1 = heap[ito+1]
L4:
 // r4 = heap[ifrom]
 ldr    r4, [r3, r6, lsl #2]
 lsls   r7, r2, #1     // r7 = 2*ifrom
 vldr   s14, [r4, #4]  // s15 = heap[ifrom]->cost
 vcmpe  s14, s15
 vmrs   APSR_nzcv, fpscr
 bpl    L2 // Swap heap[ito] and heap[ifrom].
```

175.vpr – get_heap_head()

Clang

```
L2:
 // r4 = heap
 ldr.w  r4, [r12, #4]
 // r1 = ito+1
 orr    r1, r3, #1
 // r5 = heap[ito]
 ldr.w  r5, [r4, r3, lsl #2]
 // r6 = heap[ito+1]
 ldr.w  r6, [r4, r1, lsl #2]
 // s0 = heap[ito]->cost
 vldr   s0, [r5, #4]
 // s2 = heap[ito+1]->cost
 vldr   s2, [r6, #4]
 vcmpe  s2, s0
 vmrs   APSR_nzcv, fpscr
 it     pl
 movpl  r1, r3 // r1 = ito
 // r5 = heap[ifrom]
 ldr.w  r5, [r4, r2, lsl #2]
 // r3 = heap[ito]
 ldr.w  r3, [r4, r1, lsl #2]
 // s0 = heap[ifrom]->cost
 vldr   s0, [r5, #4]
 // s2 = heap[ito]->cost
 vldr   s2, [r3, #4]
 vcmpe  s2, s0
 vmrs   APSR_nzcv, fpscr
 bgt    L3
```

GCC

```
L3:
 adds   r4, r2, #1    // r4 = ito+1
 lsls   r7, r4, #2     // r7 = (ito+1)*4
 // r9 = heap[ito+1]
 ldr    r9, [r3, r4, lsl #2]
 subs   r5, r7, #4     // r5 = (ito)*4
 ldr    r1, [r3, r5]   // r1 = heap[ito]
 add    r5, r5, r3     // r5 = &(heap[ito])
 vldr   s14, [r9, #4] // s14 = heap[ito+1]->cost
 vldr   s15, [r1, #4] // s15 = heap[ito]->cost
 vcmpe  s14, s15
 vmrs   APSR_nzcv, fpscr
 bpl    L4
 // ito++
 vmov   s15, s14    // s15 = heap[ito+1]->cost
 mov    r2, r4      // r2 = ito+1
 adds   r5, r3, r7 // r5 = &(heap[ito+1])
 mov    r1, r9      // r1 = heap[ito+1]
L4:
 // r4 = heap[ifrom]
 ldr    r4, [r3, r6, lsl #2]
 lsls   r7, r2, #1     // r7 = 2*ifrom
 vldr   s14, [r4, #4] // s15 = heap[ifrom]->cost
 vcmpe  s14, s15
 vmrs   APSR_nzcv, fpscr
 bpl    L2 // Swap heap[ito] and heap[ifrom].
```

175.vpr – get_heap_head()

Clang

```
 // heap[ito] = heap[ifrom]
 str.w  r5, [r4, r1, lsl #2]
 // r4 = heap
 ldr.w  r4, [r12, #4]
 // heap[ifrom] = heap[ito]
 str.w  r3, [r4, r2, lsl #2]
 lsl.w  r3, r1, #1 // r3: ito = 2*ifrom
 mov    r2, r1     // r2: ifrom = ito
 // while (ito < heap_tail)
 cmp    r3, lr
 blt    L2
L3:
 // r1 = heap_head->index
 ldr    r1, [r0]
 // while (heap_head->index == OPEN)
 cmp.w  r1, #-1
 beq    L1

 pop.w  {r4, r5, r6, r7, r11, pc}
L4:
 movw   r0, :lower16:.Lstr35
 movt   r0, :upper16:.Lstr35
 bl     puts
 movw   r0, :lower16:.Lstr36
 movt   r0, :upper16:.Lstr36
 bl     puts
 movs   r0, #0
 pop.w  {r4, r5, r6, r7, r11, pc}
```

GCC

```
L2:
 cmp    r0, r7
 str    r4, [r5] // heap[ito] = heap[ifrom]
 // heap[ifrom] = heap[ito]
 str    r1, [r3, r6, lsl #2]
 mov    r6, r2   // r6: ifrom = ito
 mov    r2, r7   // r2: ito = 2*ifrom
 ble    L5       // ito >= heap_tail
L5:
 ldr    r2, [r8] // r2 = heap_head->index
 adds   r2, r2, #1
 bne    L8       // heap_head->index != OPEN

 mov    r2, r0
 cmp    r2, #1
 bne    L1       // if (heap_tail != 1)
L6:
 printf("Empty heap...
 ...
L7:
 ldrd   r3, r4, [sp]
 ldrd   r5, r6, [sp, #8]
 ldrd   r7, r8, [sp, #16]
 add    sp, sp, #24
 pop    {r9, pc}
L8:
 ...
```

175.vpr – get_heap_head()

Clang

```
 // heap[ito] = heap[ifrom]
 str.w  r5, [r4, r1, lsl #2]
 // r4 = heap
 ldr.w  r4, [r12, #4]
 // heap[ifrom] = heap[ito]
 str.w  r3, [r4, r2, lsl #2]
 lsl.w  r3, r1, #1 // r3: ito = 2*ifrom
 mov    r2, r1     // r2: ifrom = ito
 // while (ito < heap_tail)
 cmp    r3, lr
 blt    L2
L3:
 // r1 = heap_head->index
 ldr    r1, [r0]
 // while (heap_head->index == OPEN)
 cmp.w  r1, #-1
 beq    L1

 pop.w  {r4, r5, r6, r7, r11, pc}
L4:
 movw   r0, :lower16:.Lstr35
 movt   r0, :upper16:.Lstr35
 bl     puts
 movw   r0, :lower16:.Lstr36
 movt   r0, :upper16:.Lstr36
 bl     puts
 movs   r0, #0
 pop.w  {r4, r5, r6, r7, r11, pc}
```

GCC

```
L2:
 cmp    r0, r7
 str    r4, [r5] // heap[ito] = heap[ifrom]
 // heap[ifrom] = heap[ito]
 str    r1, [r3, r6, lsl #2]
 mov    r6, r2   // r6: ifrom = ito
 mov    r2, r7   // r2: ito = 2*ifrom
 ble    L5       // ito >= heap_tail
L5:
 ldr    r2, [r8] // r2 = heap_head->index
 adds   r2, r2, #1
 bne    L8       // heap_head->index != OPEN
```

```
do {
 ...
  while (ito < heap_tail) {
  ...
    temp_ptr = heap[ito];
    heap[ito] = heap[ifrom];
    heap[ifrom] = temp_ptr;
    ifrom = ito;
    ito = 2*ifrom;
  }
/* Get another one if invalid entry. */
} while (heap_head->index == OPEN)
```

```
 pop    {r5, pc}
L8:
 ...
```

175.vpr – get_heap_head()

Clang

```
 // heap[ito] = heap[ifrom]
 str.w  r5, [r4, r1, lsl #2]
 // r4 = heap
 ldr.w  r4, [r12, #4]
 // heap[ifrom] = heap[ito]
 str.w  r3, [r4, r2, lsl #2]
 lsl.w  r3, r1, #1 // r3: ito = 2*ifrom
 mov    r2, r1     // r2: ifrom = ito
 // while (ito < heap_tail)
 cmp    r3, lr
 blt    L2
L3:
 // r1 = heap_head->index
 ldr    r1, [r0]
 // while (heap_head->index == OPEN)
 cmp.w  r1, #-1
 beq    L1

 pop.w  {r4, r5, r6, r7, r11, pc}
L4:
 movw   r0, :lower16:.Lstr35
 movt   r0, :upper16:.Lstr35
 bl     puts
 movw   r0, :lower16:.Lstr36
 movt   r0, :upper16:.Lstr36
 bl     puts
 movs   r0, #0
 pop.w  {r4, r5, r6, r7, r11, pc}
```

GCC

```
L2:
 cmp    r0, r7
 str    r4, [r5] // heap[ito] = heap[ifrom]
 // heap[ifrom] = heap[ito]
 str    r1, [r3, r6, lsl #2]
 mov    r6, r2   // r6: ifrom = ito
 mov    r2, r7   // r2: ito = 2*ifrom
 ble    L5       // ito >= heap_tail
L5:
 ldr    r2, [r8] // r2 = heap_head->index
 adds   r2, r2, #1
 bne    L8       // heap_head->index != OPEN

 mov    r2, r0
 cmp    r2, #1
 bne    L1       // if (heap_tail != 1)
L6:
 printf("Empty heap...
 ...
L7:
 ldrd   r3, r4, [sp]
 ldrd   r5, r6, [sp, #8]
 ldrd   r7, r8, [sp, #16]
 add    sp, sp, #24
 pop    {r9, pc}
L8:
 ...
```

175.vpr – get_heap_head()

Clang

```
 // heap[ito] = heap[ifrom]
 str.w  r5, [r4, r1, lsl #2]
 // r4 = heap
 ldr.w  r4, [r12, #4]
 // heap[ifrom] = heap[ito]
 str.w  r3, [r4, r2, lsl #2]
 lsl.w  r3, r1, #1 // r3: ito = 2*ifrom
 mov    r2, r1     // r2: ifrom = ito
 // while (ito < heap_tail)
 cmp    r3, lr
 blt    L2
L3:
 // r1 = heap_head->index
 ldr    r1, [r0]
 // while (heap_head->index == OPEN)
 cmp.w  r1, #-1
 beq    L1

 pop.w  {r4, r5, r6, r7, r11, pc}
L4:
 movw   r0, :lower16:.Lstr35
 movt   r0, :upper16:.Lstr35
 bl     puts
 movw   r0, :lower16:.Lstr36
 movt   r0, :upper16:.Lstr36
 bl     puts
 movs   r0, #0
 pop.w  {r4, r5, r6, r7, r11, pc}
```

GCC

```
L2:
 cmp    r0, r7
 str    r4, [r5] // heap[ito] = heap[ifrom]
 // heap[ifrom] = heap[ito]
 str    r1, [r3, r6, lsl #2]
 mov    r6, r2   // r6: ifrom = ito
 mov    r2, r7   // r2: ito = 2*ifrom
 ble    L5       // ito >= heap_tail
L5:
 ldr    r2, [r8] // r2 = heap_head->index
 adds   r2, r2, #1
 bne    L8       // heap_head->index != OPEN

 mov    r2, r0
 cmp    r2, #1
 bne    L1       // if (heap_tail != 1)
L6:
 printf("Empty heap...
 ...
L7:
 ldrd   r3, r4, [sp]
 ldrd   r5, r6, [sp, #8]
 ldrd   r7, r8, [sp, #16]
 add    sp, sp, #24
 pop    {r9, pc}
L8:
 ...
```

175.vpr - get_heap_head()

- Recap: List of potential optimizations
  - Replace redundant load into floating-point register with a move instead
  - Eliminate the two redundant loads of “heap”
  - Combining loads across basic blocks into paired loads
  - Emit LDRD/STRD instead of PUSH/POP on the Cortex-A15?
  - Code size (comparison against -1)

Summary

Summary

- Ongoing work to implement optimizations
- Optimizations in the middle-end will benefit other targets as well
- LLVM getting close to be on par with GCC on 32-bit ARM
- Just four benchmarks where GCC is doing significantly better
- Overall code quality is very high

Thank you.

We are hiring!

Contact Information:

Tilmann Scheller
t.scheller@samsung.com

Samsung Open Source Group
Samsung Research UK