GETTING STACK SIZE JUST RIGHT ON XCORE

JACK MCCREA | 7TH OCTOBER 2020

XMOS

Bringing technology to life

WHAT IS XCORE?

- RISC microprocessor architecture designed by XMOS
- Multiple logical cores with shared address space
- Backend for 1st generation added to LLVM in 2008
- XMOS currently shipping LLVM-based toolchain
  - But this hasn’t been upstreamed in some time – partly due to modifications relating to thread stack allocation

QUICK THREADING EXAMPLE

“Hello World” IN A WORKER THREAD

```
        .text
        .global main
        .align  4
main:
        getr r0, 4                                # Get a free thread
        ldap r11, puts                            # Get the address of "puts"
        init t[r0]:pc, r11                        # Initialise the thread's PC
        ldaw r1, dp[.L.my_stack+2040]             # Get the end of our stack space
        init t[r0]:sp, r1                         # Initialise the thread's SP
        ldaw r11, cp[.L.hello_str]                # Get the address of our argument
        set t[r0]:r0, r11                         # Initialise the 0th argument
        ldap r11, __xcore_unsynchronised_thread_end # Get an address to go to when done
        init t[r0]:lr, r11                        # Initialise our thread's LR
        start t[r0]                               # Start the thread
        bu -1                                     # Loop forever

        .section .cp.rodata.string,"aMSc",@progbits
        .align  4
.L.hello_str:
        .asciiz"Hello World!"

        .section .dp.bss,"awd",@nobits
        .align  8
.L.my_stack:
        .space  8192
```

WHY DO WE CARE ABOUT STACKS?

- 5 threads run at full speed; launch in a few instructions
- Single-cycle memory access (no cache hierarchy for RAM)

=> Not threading = wasting time

- No paged MMU (can’t thin provision)
- No memory protection (can’t detect underprovision)
- Applications often memory hungry (can’t afford overprovision)

=> We have to be careful with stacks

SOLUTION

- Functions annotated with stack requirement at object/ASM level
- Annotations added by the backend where possible
- Linker allocates stacks for ‘top-level’ (permanent) tasks statically
    - Zero runtime overhead
- Transient worker task stacks taken from top of parent’s stack
    - A few cycles launch overhead

STACK SIZE ANNOTATION

C

```
void a(void)
{
  int a[5];
}
```

GENERATED XCORE ASM

```
.set    a.nstackwords,5
```

STACK SIZE ANNOTATION - CALLS

C

```
void a(void)
{
    int a[5];
    b();
}
```

GENERATED XCORE ASM

```
.set a.nstackwords,
     (b.nstackwords + 5)
```

STACK SIZE ANNOTATION - CALLS

C

```
void a(void)
{
  int a[5];
  b();
  c();
}
```

GENERATED XCORE ASM

```
.set a.nstackwords,
     ((b.nstackwords
       $M c.nstackwords) + 5)
```

STACK SIZE ANNOTATION – CHILD THREADS

C

```
void a(void)
{
  int a[5];
  b();
  IN_PARALLEL( c(), d() );
}
```

GENERATED XCORE ASM

```
.set a.nstackwords,
     ((b.nstackwords
       $M (c.nstackwords
           + d.nstackwords + 1))
    + 5)
```

STACK SIZE ANNOTATION – INDIRECT CALLS

C

```
__attribute__((fptrgroup("my_funcs")))
void f1(void) {}

__attribute__((fptrgroup("my_funcs")))
void f2(void) {}

void(*
  __attribute__((fptrgroup("my_funcs")))
  fptr)(void);

void a(void)
{
  fptr();
}
```

GENERATED XCORE ASM

```
.add_to_set _fptrgroup.my_funcs.group,
            f1, f1
.add_to_set _fptrgroup.my_funcs.group,
            f2, f2
.max_reduce
  _fptrgroup.my_funcs.nstackwords,
  _fptrgroup.my_funcs.nstackwords.group,
  0

.set a.nstackwords,
     _fptrgroup.my_funcs.nstackwords
```

CURRENT IMPLEMENTATION

- Currently shipping an implementation to users
- Calculation is performed as a pre-emit pass
  - This makes most calculation trivial
  - But makes calculations for indirect calls difficult

WHAT NEXT?

- Aiming to make some changes with a view to upstreaming:
  - Move most calculation logic before instruction selection
  - Develop a low-friction approach to indirect call annotation in C/C++

THANK YOU

QUESTIONS/COMMENTS/THOUGHTS?

jackmccrea@xmos.com