Compiling for dual issue

Jamie Hanlon

XMOS

Dual issue overview

Two lanes in the execution pipeline.

- Memory: all memory instructions, branching & ALU ops.
- Resource: all resource & ALU ops.

Instructions are statically scheduled by the compiler.

- Invalid schedules cause runtime exceptions.

A thread is either in single- or dual-issue mode.

Dual-issue mode causes the thread to execute:

- two 16-bit instructions (a packet) in parallel;
- one 32-bit instruction.

Dual issue overview

Packets and 32-bit instructions must be 32-bit aligned.

The PC is always a multiple of 32 bits.

Branch targets are scaled by two.

Pad packets with NOPs.

Dual issue activation

Intended to be activated on a per-function basis.

On entry to a function, the issue mode is set (SR bit):

- single issue enabled with ENTSP.
- dual issue enabled with DUALENTSP;

The caller's issue mode is saved in the lowest bit of the LR.

The issue mode is restored by RETSP.

XMOS®

Function entry

All functions should be callable from single and dual issue.

- 32-bit alignment.
- Force issue mode.

```
.issue_mode single
.align 4
f:
  ENTSP_lu6 0
  ...
  retsp 0
```

```
.issue_mode dual
.align 4
g:
  dualentsp 0
  ...
  retsp 0
```

Simple example

```c
int sum(int a[100]) {
  int res = 0;
  for (int i=0; i<100; i++)
    res += a[i];
  return res;
}
```

```
.issue_mode dual
.align 4
sum:
    dualentsp 0
  { ldc r1, 0
  ; ldc r2, 100 }
loop:
  { ldw r3, r0[0]
  ; sub r2, r2, 1 }
  { add r1, r1, r3
  ; add r0, r0, 4 }
    bt r2, loop
  { mov r0, r1
  ; retsp 0 }
```

Channel array out inner loop

```
...
{ out res[r0], r4 ; ldw r4, r3[3] }
{ out res[r0], r4 ; ldw r4, r3[2] }
{ out res[r0], r4 ; ldw r4, r3[1] }
{ out res[r0], r4 ; ldw r4, r3[0] }
...
```

Memcpy inner loop

```
...
memcpy_align4_loop:
  ldd r5, r4, r1[r3]
  std r11, r5, r0[r3]
  sub r3, r3, 1
memcpy_align4_loop_mid:
  ldd r5, r11, r1[r3]
  std r4, r5, r0[r3]
  { sub r3, r3, 1 ; bt r3, memcpy_align4_loop }
...
```

Bytewise memcpy fragment

```
{ ld8u r11, r1[r2] ; sub r2, r2, 1 }
  st8 r11, r0[r2]
{ ld8u r11, r1[r2] ; sub r2, r2, 1 }
  st8 r11, r0[r2]
{ ld8u r11, r1[r2] ; sub r2, r2, 1 }
  st8 r11, r0[r2]
{ ld8u r11, r1[r2] ; sub r2, r2, 1 }
  st8 r11, r0[r2]
```

Dual issue exceptions

Destination operands must be disjoint.

An exception in one lane also aborts the other lane.

If both lanes raise an exception, then only one reported.

Resource-lane stalls causes the memory lane also to stall.

Any memory store in progress completes even if aborted.

When handling an exception:

- the saved PC set to the address of the packet;
- the lane responsible is encoded in the exception type.

LLVM code generator

Instruction selection.

- Produce initial code in target instruction set in a DAG.

Scheduling and formation.

- Determine an ordering of target instructions.

Register allocation.

- Virtual registers to concrete registers (introduce spills).

Function prologue & epilogue code insertion.

- Eliminate abstract stack references.

Code emission.

- Including VLIW packetisation.

LLVM's VLIW packetiser

General code to transform sequences of machine instructions into packets, mapping them to functional units available on the target.

Specify in the target backend what functional units are available, and which instructions can be mapped to them.

Target-dependent hook `isLegalToPacketizeTogether` to handle architectural constraints:

- barrier instructions that can stop control flow reaching next instruction (e.g. branches, WAITEU);
- register dependencies;
- load/store dependencies.

Difficult cases

A lock guarding a store:

```
{ in r0, res[r1] ; store r2, r3[r4] }
```

An ecall guarding a store:

```
{ ecallf r1 ; store r2, r3[r4] }
```

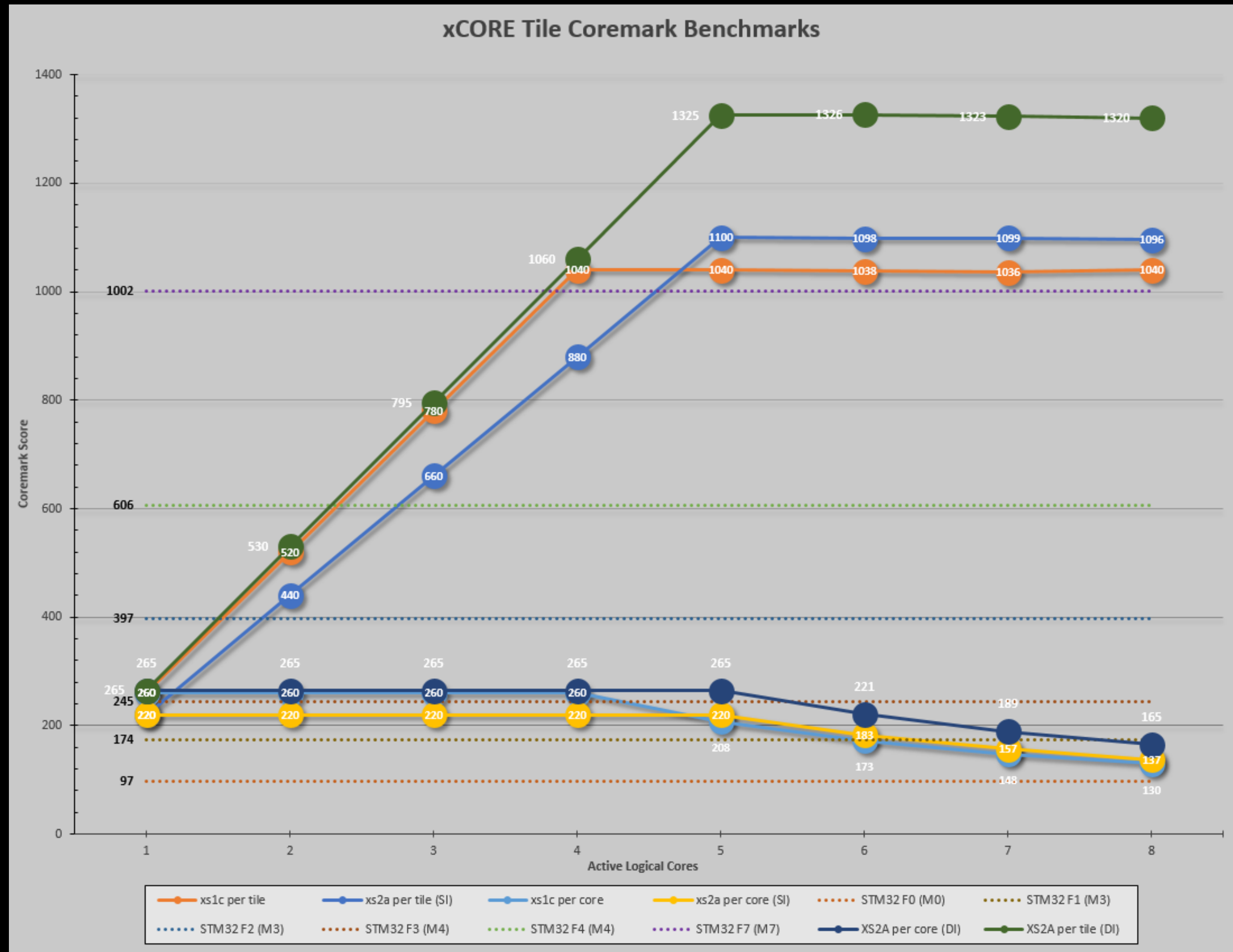
CoreMark performance

∼25% faster dual issue vs. single issue.

Always faster than XS1C in dual issue.

∼25% faster than XS1C with dual issue and 5+ threads.

CoreMark performance



XMOS®

Improvements/optimisations

Packetise over a larger range of instructions.

- Currently, only consider neighbouring instructions within BB.
- Use spare lane as a delay slot and pull instructions from next BB.

Modify scheduling and register allocation to increase number of live registers to reduce false dependencies.

Implement a dual-issue scheduling phase that deals specifically with I/O and memory.

Look for specific cases that yield benefits.

- E.g. incrementing a counter with the closing branch of a loop.

Code generation for the simple example

```
.align 4
.issue_mode dual
foo:
  { mov r1, r0 ; dualentsp 0 }
  { ldc r0, 0 ; nop }
    ldc r2, 100
loop:
  { nop ; ldw r3, r1[0] }
  { add r0, r3, r0 ; sub r2, r2, 1 }
  { add r1, r1, 4 ; nop }
    bt r2, loop
  { nop ; retsp 0 }
```

Questions?