QEmu TCG Enhancements for Speeding-up the Emulation of SIMD instructions

Luc Michel, Nicolas Fournel and Frédéric Pétrot

TIMA Laboratory

System Level Synthesis Group

DATE’11 W8
18/03/2011
Outline

1 Introduction
   - About QEmu
   - About SIMD instructions

2 QEmu operation
   - The intermediate representation
   - The helpers

3 Improving Neon instructions translation
   - A solution to improve the translation
   - Intermediate representation extension choices

4 Tests and results
   - Tests protocol
   - Performance measurement
QEmu: a fast and portable dynamic translator

Simulation with QEmu

- Open-source simulation and virtualization software,
- Dynamic binary translation of the code of a target architecture,
- To be executed on an host architecture.

Precise goal of the present work

- Accelerate the cross-execution of the Neon instructions.
What are SIMD instructions?

SIMD Instructions: Single Instruction, Multiple Data

- Same operation on multiple data in parallel,
- Very efficient to optimize some algorithms: parts of media codecs, of radio processes, . . . ,
- 64 bits or 128 bits data vectors,
- 8, 16, 32, 64 bits data depending on the instructions.
Example: vadd.i16

Taken from the ARM Neon instruction set

```
16bits 16bits 16bits 16bits 16bits 16bits 16bits 16bits
```

```
q1 + + + + + + + +
```

```
q2 + + + + + + + +
```

```
q0 ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
```

128bits
Outline

1. Introduction
   - About QEmu
   - About SIMD instructions

2. QEmu operation
   - The intermediate representation
   - The helpers

3. Improving Neon instructions translation
   - A solution to improve the translation
   - Intermediate representation extension choices

4. Tests and results
   - Tests protocol
   - Performance measurement
The intermediate representation of QEmu

Independent intermediate representation consists of micro-operations.

- add_i32
- mov_i32
- or_i32

Two steps translation

1. Target architecture code \rightarrow micro-operations,
2. micro-operations \rightarrow host architecture code.

Intermediate representation benefits

- Independence between targets and hosts architectures.
**Binary translation example**

```
... 0x00000110: mov sp, r0
 0x00000114: sub r0, r0, #32768
 0x00000118: msr CPSR_c, #209
...

Binary Code from the target machine (ARM)
```

```
... mov_i32 tmp9,$0xdf
 mov_i32 tmp10,$cpsr_write
 call tmp10,$0x0,$0,tmp8,tmp9
 movi_i32 pc,$0x11c
 exit_tb $0x0
...

Intermediate representation
```

```
... 0xb4f270b4: mov $0xdf,%ecx
 0xb4f270b9: mov %ecx,0x4(%esp)
 0xb4f270bd: mov %eax,0x34(%ebp)
 0xb4f270c0: mov %edx,0x0(%ebp)
 0xb4f270c3: call 0x821fa25
 0xb4f270c8: mov $0x11c,%eax
 0xb4f270cd: mov %eax,0x3c(%ebp)
...

Binary code generated for the host machine (x86)
```
The helpers

- C functions, simulate an instruction,
- Compiled as a part of QEmu,
- Called when translating the corresponding Neon instruction.
Example with a helper

Binary code from the target machine (ARM)

```
0x00008584: vld1.32 {d0-d1}, [r0]
0x00008588: vld1.32 {d2-d3}, [r1]
0x0000858c: vadd.i16 q0, q1, q2
0x00008590: vst1.32 {d0-d1}, [r0]
```

Intermediate representation

```
ld_i32 tmp8,env,$0x2f8
ld_i32 tmp9,env,$0x308
movi_i32 tmp10,$0x_add_u16
call tmp10,$0x0,tmp8,tmp8,tmp9
st_i32 tmp8,env,$0x2e8
```

Binary code generated for the host machine (x86)

```
0xb56c38bb: je 0xb56c38c9
0xb56c38bd: mov $0x1,%edx
0xb56c38c2: call 0xb21ce38
0xb56c38c7: jmp 0xb56c38ce
0xb56c38c9: add $0x(%edx),%eax
```

Helper C function

```c
uint32_t Helper(neon_add_u16)
(uint32_t a, uint32_t b)
{
    uint32_t mask;
    mask = (a ^ b) & 0x00000000u;
    a &= ~0x00000000u;
    b &= ~0x00000000u;
    return (a + b) ^ mask;
}
```
Helpers overhead

- Function call,
  - Adapting the arguments,
  - Passing the arguments,
  - Getting the result.
- Multiple calls because each 64b/128b vector split into 32b parts
Outline

1. Introduction
   - About QEmu
   - About SIMD instructions

2. QEmu operation
   - The intermediate representation
   - The helpers

3. Improving Neon instructions translation
   - A solution to improve the translation
   - Intermediate representation extension choices

4. Tests and results
   - Tests protocol
   - Performance measurement
A solution to improve the translation

The idea
- Be able to take advantage of the host SIMD capabilities,
- Add some SIMD micro-operations to the QEmu IR,
- Translate these micro-operations to host SIMD instructions.

The practical example of this work
- ARM Neon instruction set $\rightarrow$ Intel x86 MMX/SSE instruction set.
How to extend the IR

Choose how to extend the QEmu IR

- Adding a micro-operation for each target instruction,
- Keep a little IR and add only *elementary* micro-operations.

Our choice

Try to keep the IR as simple as possible.
Direct mapping between two instructions

- The most favorable case,
- micro-operation with the semantic of these two instructions.

Mapping between \texttt{vadd.i16} (Neon) and \texttt{paddw} (MMX/SSE)

- \texttt{vadd.i16} (ARM Neon instruction)
- IR micro-operation translation
- \texttt{simd_128_add_i16}
- IR micro-operation
- \texttt{paddw}
- x86 MMX/SSE instruction
- Host code generation
A Neon instruction emits multiple micro-operations

- The Neon instruction is not *elementary*,
- split into several elementary micro-operations.

Translating the `vsra.u32` (Neon) instruction

- `vsra.u32` (ARM Neon instruction)
- `simd_128_shr_i32` (IR micro-operation)
- `simd_128_add_i32` (IR micro-operation)
- `psrld` (x86 MMX/SSE instruction)
- `padd` (x86 MMX/SSE instruction)
A micro-operation generates multiple host instructions

- No equivalent for this micro-operation on the host,
- micro-operation behavior reproduced with host instructions,
- Harder to perform with QEmu than previous case.

The `simd_128_shl_i8` micro-op emits several host instructions
Outline

1. Introduction
   - About QEmu
   - About SIMD instructions

2. QEmu operation
   - The intermediate representation
   - The helpers

3. Improving Neon instructions translation
   - A solution to improve the translation
   - Intermediate representation extension choices

4. Tests and results
   - Tests protocol
   - Performance measurement
What kind of tests?

**Unitary tests**
- Ensure correctness of the translation,
- detect regression during the development phase.

**Performance measurement**
- Execution time.
Tests environment

Linux in QEmu

- Minimalist Linux system,
- Cross-compilation toolchain to compile some programs for the test system.

Real BeagleBoard system

- Board embedding an ARM Cortex-A8 CPU with Neon extension,
- Used to validate our unitary tests.
Performance tests

The three chosen instructions

- vadd.i16,
- vsra.u16,
- vshl.u8.

For each instruction...

- 101 assembly functions,
- containing 0% to 100% of this Neon instruction,
- filled with classical instructions,
- executed several times in a loop,
- total execution time measured for the *helpers* and *mapping* strategies
Performance tests results

![Graph showing relative execution time (%) compared to helpers for various SIMD instructions: vadd.i16, vsra.u16, vshl.u8. The x-axis represents SIMD instructions (%) ranging from 0 to 100, and the y-axis represents the relative execution time (%) ranging from 0 to 110. The graph indicates improvements in execution time for vadd.i16, vsra.u16, and vshl.u8 over the tested range.]
Take away message

Conclusion
- Results are very encouraging, but Amdahl’s law still rules

What to do next?
- Extend the implementation to more SIMD instruction sets,
- Probably with the help of automation tools

Call to QEmu development community
- Should this approach be promoted into mainstream QEmu?
Thanks for your attention

And now ready to answer your questions!