QEMU internals

Chad D. Kersey

January 28, 2009
Where to get the source

svn co svn://svn.savannah.nongnu.org/qemu

Make sure you have the latest sources if you’re reading along. A lot has changed since the previous release.
Functional simulation

- Simulate what a processor does, not how it does it.
- Needs separate model for timing analysis (if needed).
- Faster than “cycle-accurate” simulators.
- Good enough to use applications written for another CPU.
QEMU system simulation

- QEMU simulates VGA, serial, and ethernet.
- hw/* contain all of the supported boards.
- Includes rather complete PC, Nokia N-series, PCI ultrasparc.
- Various development boards in varying levels of completion.
What dynamic translation isn’t

- Interpreters execute instructions one at a time.
- Significant slowdown from constant overhead.
- Easier to write and debug than dynamic translators.

Guest Code → Static Code

→ Control Flow

...→ Data Flow
What dynamic translation is

- Dynamic translators convert code as needed.
- Try to spend most time executing in translation cache.
- Translate basic blocks as needed.
- Store translated blocks in code cache.
Getting into and out of the code cache

- `cpu_exec()` called each time around main loop.
- Program executes until an unchained block is encountered.
- Returns to `cpu_exec()` through epilogue.
Portable dynamic translation

- QEMU uses an intermediate form.
- Frontends are in `target-*/`
- Backends are in `tcg/*/`
- Selected with preprocessor evil.

```
gen_intermediate_code()
TCG Operations
tcg_gen_code()
```

Guest Code

Host Code
Portable dynamic translation: stage 1

```
Guest Code

gen_intermediate_code()

TCG Operations

tcg_gen_code()

Host Code

push  %ebp
mov   %esp,%ebp
not   %eax
add   %eax,%edx
mov   %edx,%eax
xor   $0x55555555,%eax
pop   %ebp
ret
```
Portable dynamic translation: stage 2

Guest Code

\[ \text{gen\_intermediate\_code()} \]

TCG Operations

\[ \text{tcg\_gen\_code()} \]

Host Code

\[ \ldots \]

\[ \text{ld\_i32 tmp2,env,0x10} \]
\[ \text{qemu\_ld32u tmp0,tmp2,0xffffffff} \]
\[ \text{ld\_i32 tmp4,env,0x10} \]
\[ \text{movi\_i32 tmp14,0x4} \]
\[ \text{add\_i32 tmp4,tmp4,tmp14} \]
\[ \text{st\_i32 tmp4,env,0x10} \]
\[ \text{st\_i32 tmp0,env,0x20} \]
\[ \text{movi\_i32 cc\_op,0x18} \]
\[ \text{exit\_tb 0x0} \]
Portable dynamic translation: stage 3

Guest Code

`gen_intermediate_code()`

TCG Operations

`tcg_gen_code()`

Host Code

```
... 
mov 0x10(%ebp),%eax
mov %eax,%ecx
mov (%ecx),%eax
mov %eax,%ecx
add $0x4,%edx
mov %edx,0x10(%ebp)
mov %eax,0x20(%ebp)
mov $0x18,%eax
mov %eax,0x30(%ebp)
xor %eax,%eax
jmp 0xba0db428
/*This represents just the ret instruction!*/
```
Basic block chaining

- Returning from code cache is slow.
- Solution: jump directly between basic blocks!
- Make space for a jump, follow by a return to the epilogue.
- Every time a block returns, try to chain it.

Pre-generated code

Translation Cache

- Prologue
- Epilogue
- TB
- TB
- TB

cpu_exec()
Basic block chaining: step 1

Pre-generated code

Prologue

Epilogue

Translation Cache

TB

TB

TB

TB

cpu_exec()
Basic block chaining: step 2

Pre-generated code

Prologue

Epilogue

Translation Cache

TB

TB

TB

TB

cpu_exec()
Basic block chaining: step 3

Pre-generated code
- Prologue
- Epilogue

Translation Cache
- TB
- TB
- TB

cpu_exec()
Basic block chaining: step 4

Pre-generated code

Prologue

Epilogue

Translation Cache

TB

TB

TB

cpu_exec()
Basic block chaining: step 5

Pre-generated code
- Prologue
- Epilogue

Translation Cache
- TB
- TB
- TB

cpu_exec()
Now how do we interrupt the processor?
Have another thread unchain the blocks.
Code organization

- TranslationBlock structure in translate-all.h
- Translation cache is code_gen_buffer in exec.c
- cpu-exec() in cpu-exec.c orchestrates translation and block chaining.
- target-*/translate.c: guest ISA specific code.
- tcg-*/*: host ISA specific code.
- linux-user/*: Linux usermode specific code.
- vl.c: Main loop for system emulation.
- hw/*: Hardware, including video, audio, and boards.
Ways to have fun

- Add extra instructions to an ISA.
- Generate execution traces to drive timing models.
- Try to integrate timing models.
- Retarget frontend or backend.
- Improve optimization, say, by retaining chaining across interrupts.
Acknowledgments

- QEMU by Fabrice Bellard: www.bellard.org/
- Current qemu-internals: http://bellard.org/qemu/qemu-tech.html
- Some graphics in these slides part of work funded by DOE grant.
Questions?