TCG Plugin in Practice: A Case of Microarchitecture Research

Akihiko Odaki <u>akihiko.odaki@daynix.com</u> Daynix Computing Ltd. <u>odaki@rsg.ci.i.u-tokyo.ac.jp</u> The University of Tokyo



Conventional TCG use cases

TCG: The CPU emulation engine of QEMU

Examples of conventional TCG use cases:

- Cross-development
 (e.g., debugging <u>Windows Arm64 on x86</u>)
- Retro/hobby-computing (e.g., <u>Amiga</u>)

Common goal:

Emulate **fixed hardware design fast** for **software**



TCG for μ arch research

Microarchitecture (µarch) research:

Optimizing **designs of microprocessors (= µarch)**

Our goal:

Simulate various CPU designs and evaluate their performance

Idea:

- 1. Generate software execution traces with a TCG plugin
- 2. Feed them to simulators modeling hardware designs

Why µarch?

Why research **µarch**?

To exploit more **parallelism**

Assumption: Moore's Law

Transistors gets smaller

→ #transistors doubles for every 2 years

Slowing down a bit, but **has not stopped yet**

Moore's law: The number of transistors per microprocessor

Moore's law is the observation that the number of transistors in an integrated circuit doubles about every two years, thanks to improvements in production. It was first described by Gordon E. Moore, the co-founder of Intel, in 1965.



Our World in Data

End of Dennard scaling

Old good days (~2006): Dennard scaling

- 1. Transistors gets smaller (Moore's law)
- 2. The threshold current decreases
- Raise clock to exploit the extra power budget
 → everything gets 40% faster
 for each technology generation

No longer true: transistors too small result in excessive current leakage

- Extra power consumption
- Causes subthreshold condition, preventing lowering the threshold current



Adi Fuchs, 2019.

Challenge of Amdahl's law

Reminder: **Moore's Law is still alive**; the number of transistors is continuously increasing.

Utilize extra transistors with Parallelism

Challenge: Amdahl's law

Small part of execution that cannot be parallelized bottlenecks the overall performance.



Daniels220 at English Wikipedia, 2008. CC-BY-SA 3.0 Unported

Fighting with Amdahl's law

Software Approaches:

- Multi-threading
- Vectorization

Hardware Approaches:

- Multi-core/SMT (a.k.a. hyper-threading)
- SIMD units
- Out-of-order execution
 - Instruction-level parallelization

Fight with Amdahl's law both in software and hardware



Challenge in µarch Research

µarch is **complex**.

Naive: Execute instructions top to bottom

Educated: Superscalar pipeline Reality: Speculate everywhere Parallelize everywhere

Dur	np of assembler code 0x0000aaaaaaab02c0 0x0000aaaaaaab02c4 0x0000aaaaaaab02c8	e for function < +0>: < +4>: < +8>:	main: paciasp sub	sp, sp, #0xd0
=>	0x0000aaaaaaab02cc 0x0000aaaaaaab02cc 0x0000aaaaaaab02d0 0x0000aaaaaaab02d4 0x0000aaaaaaab02d8 0x0000aaaaaaab02dc	<pre>< +12>: < +16>: < +20>: < +24>: < +28>:</pre>	add stp adrp mov ldr	x29, x30, [SP, #144] x29, sp, #0x90 x19, x20, [sp, #160] x19, 0xaaaaaaadf000 x20, x1 x3, [x19, #4064]

ID	EX	MEM	WB				
ID	ΕX	MEM	WB				
IF	ID	EX	MEM	WB			
IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB
			IF	ID	EX	MEM	WB

IF

IF



Simulator in rescue

- A simulator estimates execution time and optionally power/area
- A researcher can **omit details** they don't care about from the simulator
- e.g., floating-point arithmetics
 - Dark magic most people don't understand
 - Simulator: just add some constant time latency

Trace-based simulation

Problems with simulation:

- Omitting details result in a non-functional system
- Simulating whole program execution takes too long (months)

Solution: generate execution traces with a functional emulator

- Traces characterize software
- A hardware simulator follows traces and sums up latency
- Only generate traces of regions of interest (ROIs)
 - <u>SimPoint</u> automatically finds ROIs

Generating traces: conventional approaches

- Static instrumentation by compiler (LLVM)
 - Affects instruction stream
 Not applicable for µarch research
- Dynamic instrumentation
 - Intel Pin
 - **x86 is terrible** for implementers
 - Hard to extend the instruction set
- Reference interpreters
 - Slow
 - Architecture-specific
 - Inflexible
 - Limited userspace support
 - Limited/complicated debugger setup

Generating traces with QEMU/TCG

- TCG is fast
- TCG supports various architectures
 - RISC-V
 - \circ Even vector extension
- TCG has great **userspace** emulation
- QEMU works with **GDB**
- TCG plugins can generate various information



Information simulators need

States included in traces:

- PC (program counter)
- Registers
 - I and Alex Bennée developed plugin APIs to read registers (available since 9.0)
- Memory
 - Only accessed data are available to plugins

Points to generate traces:

- 1. The beginning of the execution
 - To omit program loading
- 2. After system calls
 - To omit system call implementation
- 3. Each instruction
 - \circ To omit every computation

Case study: Sniper

- Settings:
 - Sniper simulator
 - RISC-V Linux userspace on x86
 - Benchmarks
 - SPEC CPU 2017
 - GAP Benchmark Suite
 A graph benchmark suite
- Intel Pin
 - The default tracer
 - Incompatible with RISC-V
- Spike, the reference emulator of RISC-V
 - **Too slow** to run GAP benchmark suite



Case study: Sniper

Used two TCG plugins:

- Basic block vector generator for SimPoint (libbbv.so)
 - Uses <u>conditional callbacks (available since 9.1)</u> for fast execution
 - Upstreaming: [PATCH v3] contrib/plugins: Add a plugin to generate basic block vectors (co-developed with Yotaro Nada)
- Sniper frontend for generating traces (libqemu-frontend.so)
 - \circ $\,$ $\,$ Traces PC and registers for each instruction $\,$
 - Infers memory from registers



Case study: Sniper

Results:

- Succeeded in running GAP benchmark suite until end
- Passed 100% SPEC CPU 2017 validations with a few fixes (upstreamed with 8.1.0)



[PATCH v2 0/6] linux-user: brk/mmap fixes

Open problem: µarch speculation

• Speculative execution

- Triggered by branch prediction
- Allows early execution of instructions following branches
- Sometimes executes wrong instructions

• Prefetcher

- \circ $\,$ Guess the region of memory the processor will access soon
- Fills caches early
- e.g., Indirect memory prefetcher
 - A modern, complex prefetcher
 - De-references pointers in an array (i.e., requires memory content)
 - Present in Apple M1+

µarch speculation matters

Not present in traces

Traces do not contain µarch details

Enables side/covert-channel attacks

- Speculative execution: <u>Spectre</u>
- Indirect memory prefetcher: <u>GoFetch</u>

Affects performance

- Prefetchers significantly improve performance
- Wrong instruction execution after mispredicted branches often fills caches (behaves like prefetcher)
- Affects SMT





Simulating indirect memory prefetcher

Requires controlled memory read

• Needs to read memory not accessed in traces

Currently TCG plugins can only read registers and record accessed memory

• Dumping all memory may result in a huge file

GAP benchmark suite may consume 30 GB of memory

Simulating speculative execution

PC must be controllable

• Set PC to execute the wrong path

Needs checkpoint/restore

- Checkpoint before starting speculative execution
- Restore after the wrong speculative execution
- Also necessary to capture the region of interest (ROI)

Checkpoint/restore for simulation

- Normal checkpoint/restore
 - Requires huge amount of storage
 - Requires full-system emulation/virtualization
 - Slow
- Checkpoint/restore for speculative execution
 - Happens very frequently
 - No need to run system calls during speculative execution
 - The interval between checkpoint and restore is small (includes < 200 memory access insts.)
- Checkpoint/restore for the ROIs
 - Multiple ROIs

Needs to minimize storage usage

Idea: QEMU as a library

- µarch simulation poses unique requirements
- Let the simulator handle its own requirements
- <u>libgemu: Removed in 2011, leaked too much internal details</u>

```
/* install exception handler for CPU emulator */
    struct sigaction act;
    sigfillset(&act.sa mask);
   act.sa flags = SA SIGINFO;
               act.sa flags |= SA ONSTACK;
    act.sa sigaction = host segv handler;
    sigaction (SIGSEGV, &act, NULL);
    sigaction(SIGBUS, &act, NULL);
        cpu set log(CPU LOG TB IN ASM | CPU LOG TB OUT ASM | CPU LOG EXEC);
env = cpu init("qemu32");
cpu x86 set cpl(env, 3);
env->cr[0] = CR0 PG MASK | CR0 WP MASK | CR0 PE MASK;
/* NOTE: hflags duplicates some of the virtual CPU state */
env->hflags |= HF PE MASK | VM MASK;
/* flags setup : we activate the IRQs by default as in user
mode. We also activate the VM86 flag to run DOS code */
env->eflags |= IF MASK | VM MASK;
```

Idea: QEMU as a library

```
• Unicorn: out-of-tree QEMU fork
```

```
err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
if (err) {
    printf("Failed on uc_open() with error returned: %u\n", err);
    return;
}
// map 2MB memory for this emulation
uc_mem_map(uc, ADDRESS, 2 * 1024 * 1024, UC_PROT_ALL);
// write machine code to be emulated to memory
if (uc_mem_write(uc, ADDRESS, X86_CODE32, sizeof(X86_CODE32) - 1)) {
    printf("Failed to write emulation code to memory, quit!\n");
    return;
}
```

// initialize machine registers
uc_reg_write(uc, UC_X86_REG_ECX, &r_ecx);
uc_reg_write(uc, UC_X86_REG_EDX, &r_edx);
uc_reg_write(uc, UC_X86_REG_XMM0, &r_xmm0);
uc_reg_write(uc, UC_X86_REG_XMM1, &r_xmm1);

```
// tracing all basic blocks with customized callback
uc_hook_add(uc, &trace1, UC_HOOK_BLOCK, hook_block, NULL, 1, 0);
```

```
// tracing all instruction by having @begin > @end
uc_hook_add(uc, &trace2, UC_HOOK_CODE, hook_code, NULL, 1, 0);
```

```
// emulate machine code in infinite time
err = uc_emu_start(uc, ADDRESS, ADDRESS + sizeof(X86_CODE32) - 1, 0, 0);
```



Idea: QEMU as a library

- Exposing full features of QEMU as a library is impractical
 - Device emulation, userspace emulation, etc.
 - Too many interfaces
 - Too unstable interfaces
- A µarch simulator only requires to model a processor
 - Specifying processor
 - Mapping memory with RWX
 - **Reading/writing registers**; a requirement shared with gdbstub/TCG plugins
 - Executing instructions
 - **Trapping**; allows implementing application-specific behaviors
- Potentially useful for compiler research, reverse engineering, etc.

Conclusion

- µarch matters
 - \circ ~ Cope with Amdahl's law by exploiting transistors we get with Moore's law
- Trace-based simulation significantly aids µarch research
- QEMU/TCG is ideal for trace-based simulation
 - Fast
 - Rich features
- **µarch speculation** remains as an open problem
- Time to rethink **QEMU** as a library?
 - Hide internal details and only provide features commonly needed
 - Reuse gdbstub/TCG plugin code/interface
 - Not only useful for µarch simulation but also for software research and reverse engineering

Acknowledgement

- Daynix Computing Ltd supported:
 - The development of register read feature of TCG
 - Linux userspace emulation fixes
 - The basic block generator development
 - The travel to KVM Forum
 - And this presentation
- The presented µarch research was supported by JSPS KAKENHI Grant Number JP-20H04153.