Automatic Generation of Efficient Dynamic Binary Translators

Frédéric Pétrot, Luc Michel and Nicolas Fournel

Tima Laboratory,
Grenoble, France
Motivations

- Design space exploration and Early Software Development
- Goal: (co-)optimize chips and applications for performances

Difficulties

- Higher number of processor reduces simulation performances
- Sequential simulation speed is still a great concern

Current state of the art solution

Dynamic Binary Translation based ISS.

- Pros: fast and relatively precise
- Cons: complex development

Cross-compiled full software stack

Transaction accurate CPU system model
Solutions for fast development of simulators

Automatic generation, a need

- To avoid complex development,
- To allow quick availability of simulation platforms.

Automatic fast simulators generation

- Solutions has been proposed before [UC00, CVE00, NBS+02],
- Proprietary, not available, no details, no "full software execution" support, . . .

Our goal

- Automate the production of dynamic binary translators
- Benefit from automation to produce faster simulators
Agenda

- Principle of Dynamic Binary Translation
- Design flow
- Intermediate Representation Generation
- Conclusion
DBT Principle

Process

PC already seen? Yes → Instruction
No → Target binary code (.elf)

Fetch

Decode

Binary Translation

Branch?

No

Execute

TB Cache Entry

Translation Cache (host binary code)

Yes → Tiny code generator

micro-ops buffer

Code Generation

Micro-operations built-in
**DBT Principle**

**Process**

- **PC already seen?**
  - Yes → **Fetch**
  - No → **Binary Translation**

**Fetch**

**Decode**

**Branch?**

- No → **Execute**
- Yes → **Branch?**

**Execute**

**TB Cache Entry**

**Translation Cache (host binary code)**

**Micro-operations built-in**

**Tiny code generator**

**Code Generation**

**Instruction**

**Target binary code (.elf)**

**Code generation example**

18 instrx_target
DBT Principle

Process

Yes
PC already seen?

No

Instruction
Target binary code (.elf)

Micro-operations built-in

Binary Translation

Fetch

Decode

Branch?

No

Yes

Execute

TB Cache Entry

Translation Cache (host binary code)

Code generation example

18  instrX_target

micro-op1_instrX

micro-op2_instrX
DBT Principle

Process

Yes
PC already seen?

No

Binary Translation

Fetch
Decoder
Branch?

Yes
Execute

No

Instruction

Target binary code (.elf)

micro-ops buffer

Tiny code generator

Code Generation

Micro-operations built-in

TB Cache Entry

Translation Cache (host binary code)

Code generation example

18 instrX_target

micro-op1_instrX

host_instr1_micro-op1_instrX

host_instr2_micro-op1_instrX

host_instr3_micro-op1_instrX

micro-op2_instrX

host_instr1_micro-op2_instrX
The generator

- Takes a description of the *target* (simulated) architecture, and a description of the *host* (machine simulation is run on) architecture,
- Generated ISS relies on *Dynamic Binary Translation* approach,
- DBT process uses an intermediate representation.
General design flow
General design flow
General design flow
General design flow

Architecture description
- Target description
- Host description

IR generation
- Spec IR generator
- Spec IR description

Simulator generation
- Decoder generator
- Translator generator
- Codegen generator

Simulation
- Target code
- Frontend
- Spec IR
- Backend
- Host code
Why keep an IR at runtime?

Direct target to host translation possible

- But previous works shown interests in having one [UC00, CVE00, Bel05],
  - Allows for runtime optimizations,
  - Easier debugging.
Intermediate Representation Generation

Architecture description

IR generation

Simulator generation

Simulation
Why generating the IR?

Generating an IR specialized to the target/host pair

- Previous works show dramatic performance gains
- Speeding-up SIMD instructions dynamic binary translation [MFP11]
- Better SIMD translation, adapted IR (ARM Neon → x86 MMX/SSE) in QEMU.

Direct mapping case
Why generating the IR?

Generating an IR specialized to the target/host pair

- Previous works show dramatic performance gains
- Speeding-up SIMD instructions dynamic binary translation [MFP11]
- Better SIMD translation, adapted IR (ARM Neon → x86 MMX/SSE) in QEMU.

Multiple micro-operations
Why generating the IR?

Generating an IR specialized to the target/host pair

▸ Previous works show dramatic performance gains
▸ Speeding-up SIMD instructions dynamic binary translation [MFP11]
▸ Better SIMD translation, adapted IR (ARM Neon → x86 MMX/SSE) in QEMU.

Multiple host instructions
How to auto-generate a specialized IR?

Start from a canonical IR

- Used to describe the instructions in the target and host description,
- Each (part of) target instruction is matched against host instruction.
How to auto-generate a specialized IR?

Start from a canonical IR

- Used to describe the instructions in the target and host description,
- Each (part of) target instruction is matched against host instruction.
How to auto-generate a specialized IR?

Start from a canonical IR

- Used to describe the instructions in the target and host description,
- Each (part of) target instruction is matched against host instruction.
Matching constraints

Operations

- Host implements the canonical IR atoms
  - Back-end simple and efficient
  - Not an issue since IR automatically generated for (target, host) couple

Operands

- Operand size, type and location induce loose matching
- Specific code generation to handle conversions

Control

- Flags, ...
- Related to run-time on BB boundaries
Still at an early stage!

First working prototype

▶ MIPS to simple virtual machine
▶ Translator generation fitting into QEMU

Many open questions, among which

▶ Is this more efficient than using a fixed IR?
▶ Will the generated IR runtime allow optimization?
▶ How to Efficient handle non-functional properties?
Convenient design flow for DBT based simulator generation

- Fast development,
- DBT based,
- Specialized intermediate representation.

- Some parts have been addressed by previous works,
- but still a work in progress.
Thank you!


