Automatic Generation of Efficient Dynamic Binary Translators

Frédéric Pétrot, Luc Michel and Nicolas Fournel

Tima Laboratory,
Grenoble, France

MPSoC'14

Simulation Context

Motivations

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

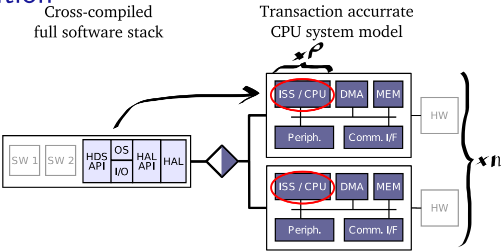
Difficulties

- 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



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

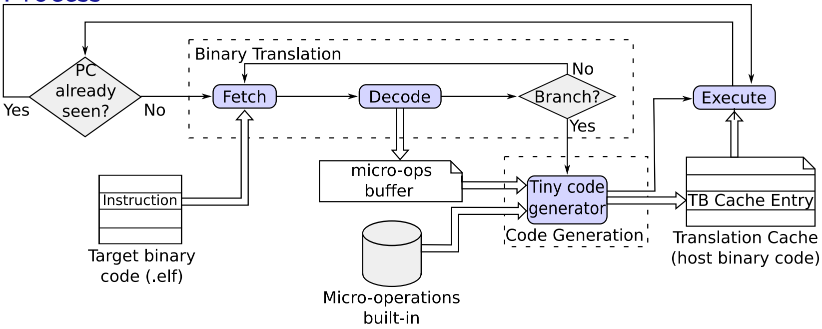
- Automate the production of dynamic binary translators
- Benefit from automation to produce *faster* simulators

Agenda

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

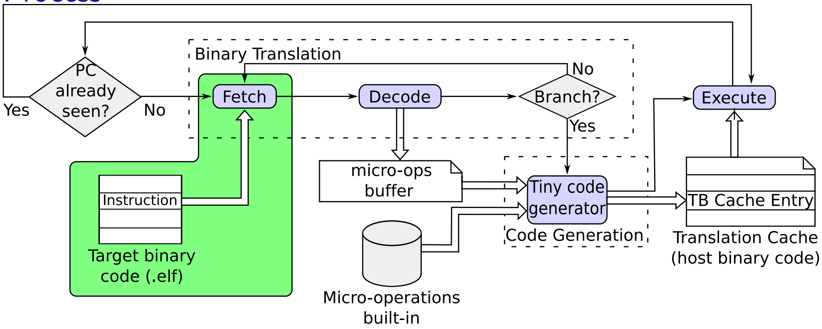
DBT Principle

Process



DBT Principle

Process

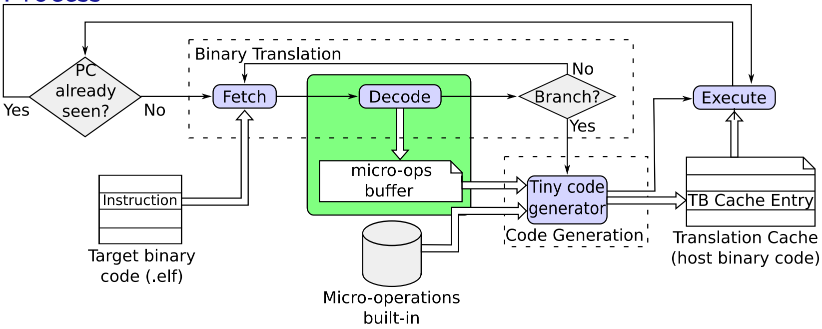


Code generation example

18 instrX_target

DBT Principle

Process



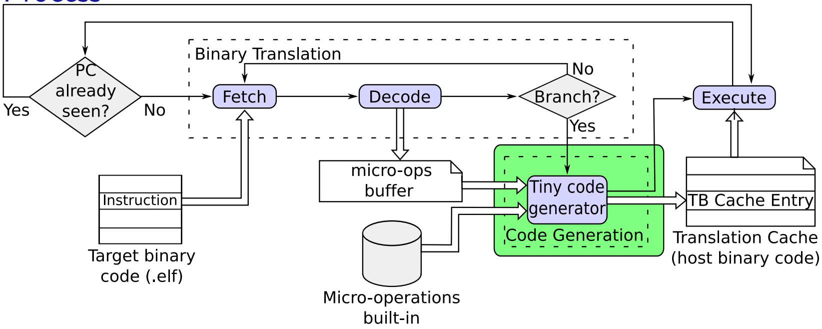
Code generation example

18 instrX_target micro-op1_instrX

micro-op2_instrX

DBT Principle

Process



Code generation example

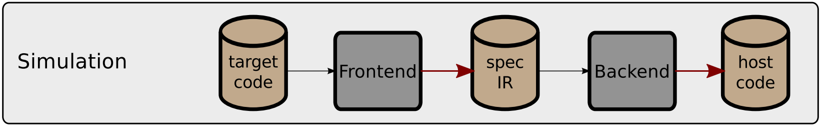
18	instrX_target	micro-op1_instrX	host_instr1_micro-op1_instrX
			host_instr2_micro-op1_instrX
			host_instr3_micro-op1_instrX
		micro-op2_instrX	host_instr1_micro-op2_instrX

Automatic simulator generation

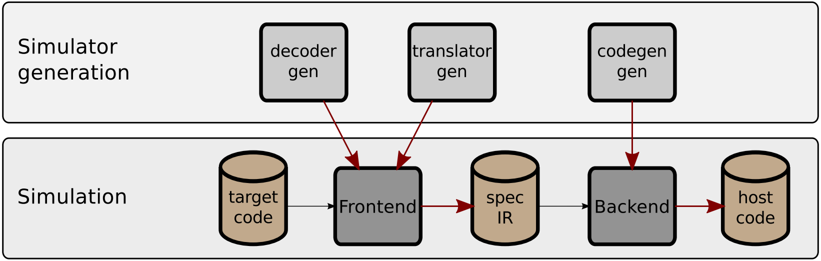
The generator

- 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.

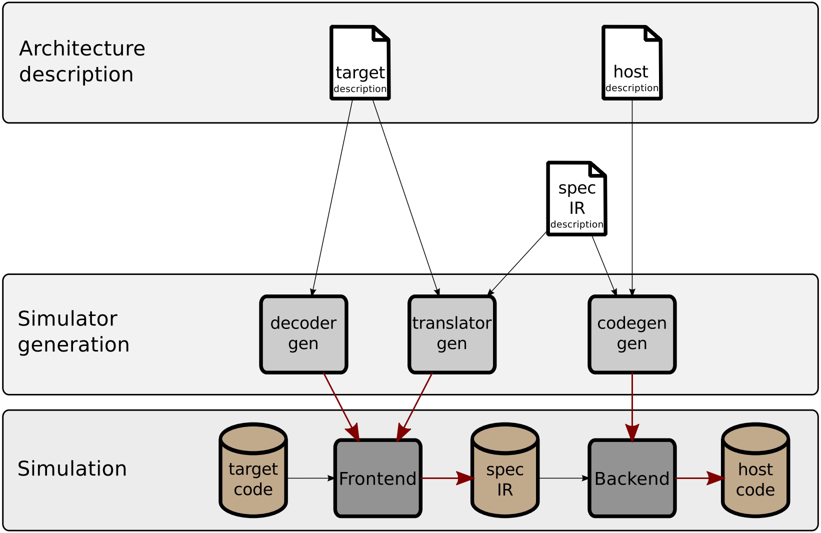
General design flow



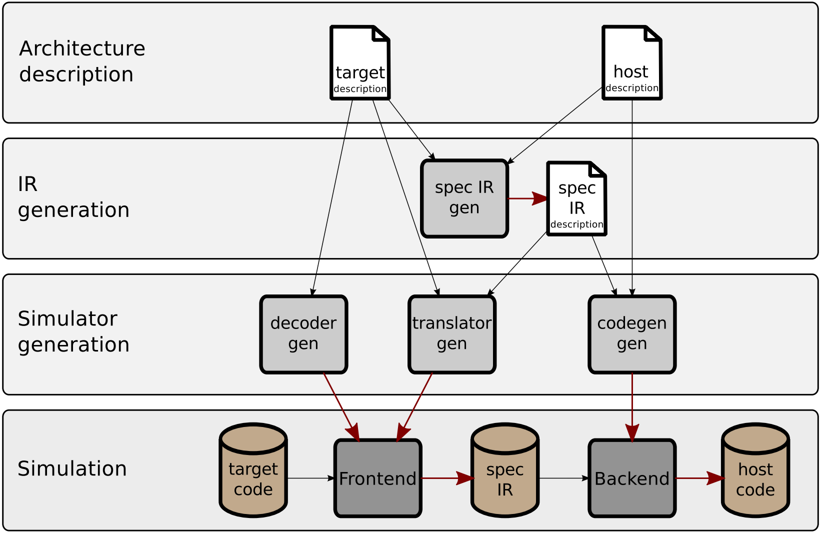
General design flow



General design flow



General design flow

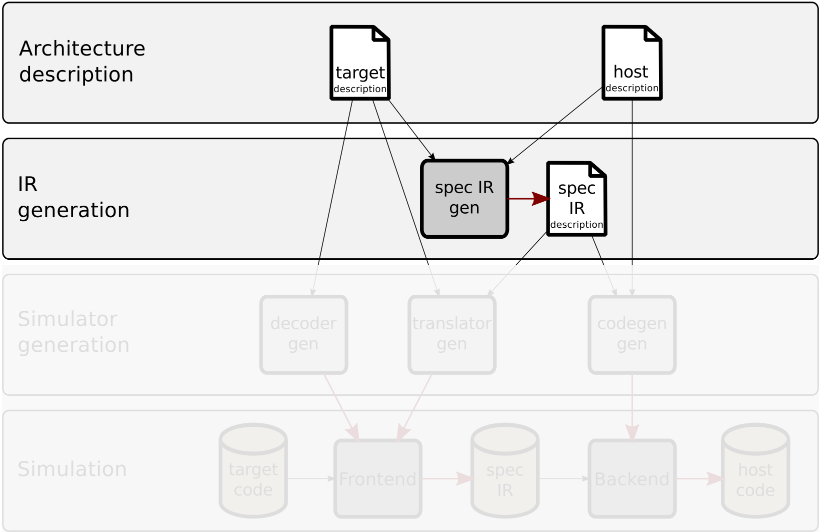


Why keep an IR at runtime?

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

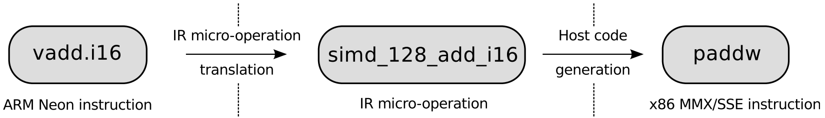


Why generating the IR?

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` $\rightarrow$ x86 `MMX/SSE`) in QEMU.

Direct mapping case

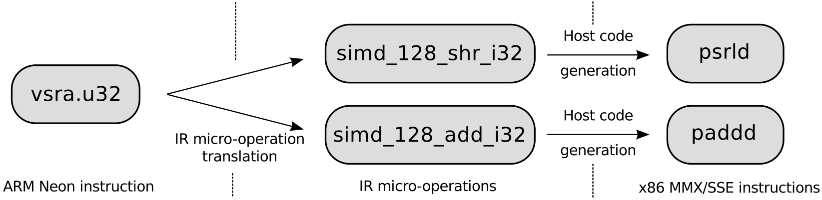


Why generating the IR?

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

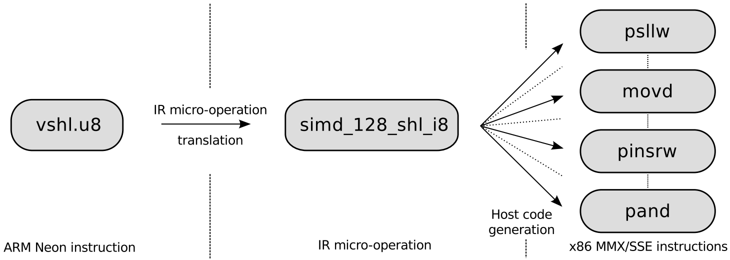


Why generating the IR?

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 $\rightarrow$ x86 MMX/SSE) in QEMU.

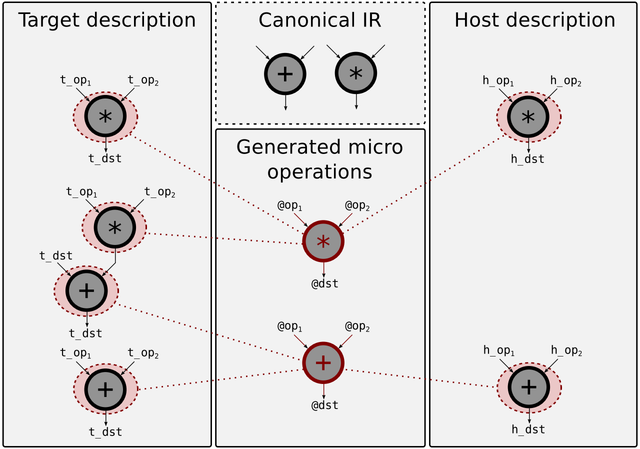
Multiple host instructions



How to auto-generate a specialized IR?

Start from a canonical IR

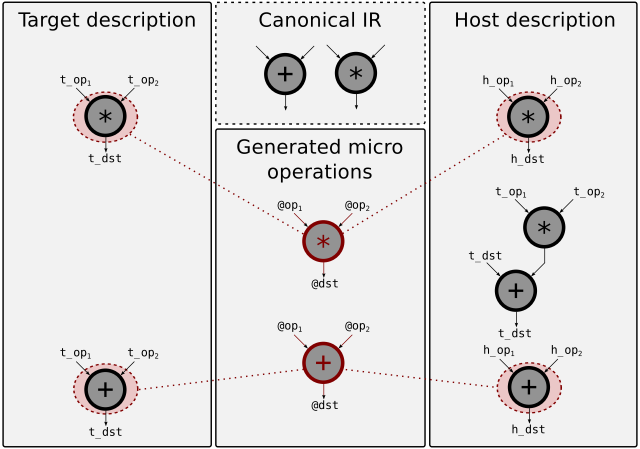
- Used to describe the instructions in the target and host description,
- Each (part of) target instruction is matched against host instruction.



How to auto-generate a specialized IR?

Start from a canonical IR

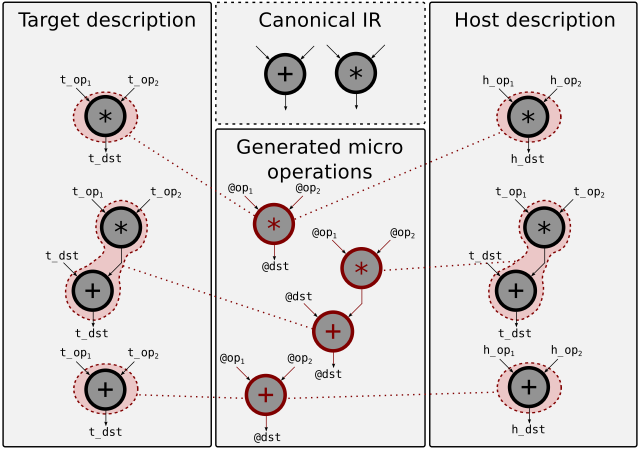
- Used to describe the instructions in the target and host description,
- Each (part of) target instruction is matched against host instruction.



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?

Conclusion

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!

- F. Bellard, *QEMU, a fast and portable dynamic translator*, the USENIX Annual Technical Conference, 2005, pp. 41–46.
- C. Cifuentes and M. Van Emmerik, *UQBT: Adaptable Binary Translation at Low Cost*, Computer **33** (2000), no. 3, 60–66.
- L. Michel, N. Fournel, and F. Pétrot, *Speeding-up SIMD instructions dynamic binary translation in embedded processor simulation*, Proceedings of the Design, Automation & Test in Europe Conference, IEEE, 2011, pp. 277–280.
- A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann, *A universal technique for fast and flexible instruction-set architecture simulation*, 39th Design Automation Conference, 2002, pp. 22–27.
- D. Ung and C. Cifuentes, *Machine-adaptable dynamic binary translation*, DYNAMO '00, 2000, pp. 41–51.