informatics

icsa | Institute for Computing Systems Architecture

Custom Alias-analysis in an LLVM-backed region-based Dynamic Binary Translator

Tom Spink

April 2014

Table of contents

Introduction
- Dynamic Binary Translators
- Instruction Set Simulators

Our ISS
- Components
- LLVM Bitcode Generator

The Problem

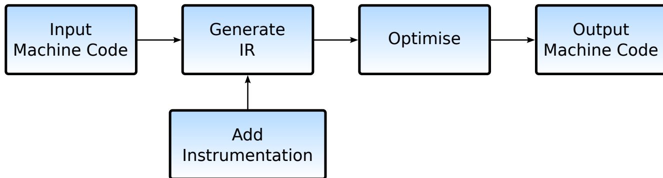
The Solution
- Challenges

Comparison to QEMU

Additional Highlights

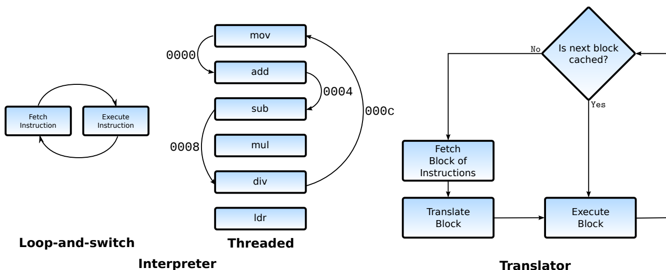
Dynamic Binary Translators

- Takes machine code from one Instruction Set Architecture (ISA) and translates it into machine code for (possibly) a different ISA.
- Usually converts input machine code into some kind of IR, and then generates output machine code from this IR.
- May optimise the IR to produce efficient output code.
- May instrument the input code during translation (e.g. for statistics, or debugging).



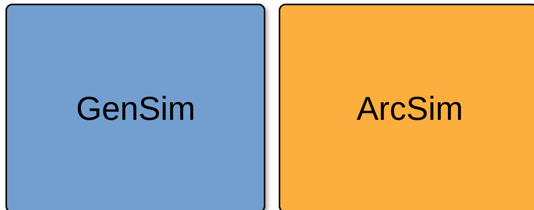
Instruction Set Simulators

- Emulates an ISA for a target platform, on a (possibly) different host platform.
- Can be built as a loop-and-switch or threaded interpreter.
- Can be built as a static binary translator.
- Can be built as a dynamic binary translator.



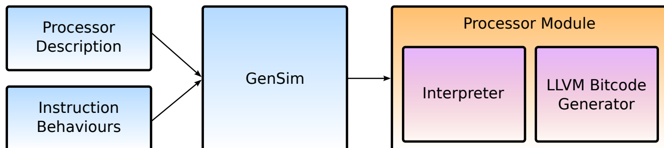
Our Iss

- GenSim: Offline processor module generator
- ArcSim: Instruction set simulation framework



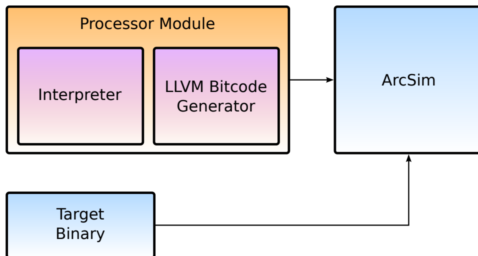
GenSim

- GenSim generates a processor module from a high-level architecture description.
- It accepts a processor and ISA description, written in a variant of Arch-C.
- It parses and optimises instruction execution behaviours from a C-like language.
- It generates an interpreter and an LLVM bitcode generator.



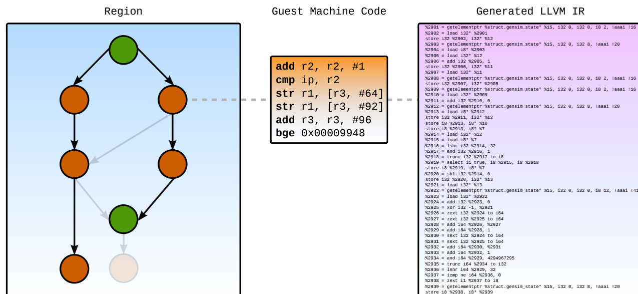
ArcSim

- ArcSim is our high-performance instruction set simulator, based on an asynchronous JIT/Interpreter model.
- Code is initially interpreted and profiled into regions.
- Regions become hot (after exceeding an execution threshold) and are compiled on separate worker threads, whilst main execution and profiling continues in the interpreter.



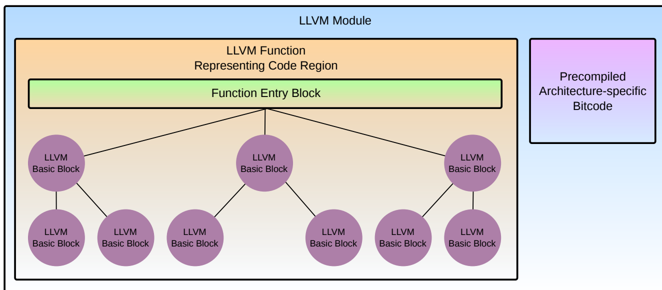
ArcSim Code Generation

- Hot regions, containing discovered target *basic-blocks*, are dispatched to an idle compilation worker thread.
- The worker thread creates a single LLVM function to represent the entire region.
- Initially, the LLVM function contains LLVM basic-blocks that correspond exactly to each *target* basic-block.



ArcSim Code Generation

- Once the LLVM IR function has been generated, it is linked with a precompiled bitcode file, containing target architecture-specific helper routines.
- Standard optimisations (`-O3`) are then applied to the module.
- The function is compiled, with `getPointerToFunction()`



LLVM Bitcode Generator

- Each target basic-block in the work-unit is considered, and a new LLVM basic-block is created for it.
- The decoded instructions in the target basic-block are iterated over, and the corresponding (pre-generated) bitcode generator is invoked for it - building the LLVM IR in to the associated LLVM basic-block.
- Additional LLVM basic-blocks may be generated at this stage, to account for control-flow within an instruction.
- We employ a partial evaluation technique, to only emit IR relevant to the decoded instruction.

The Problem

- Initially, we noticed a severe performance deficit in comparison to QEMU.
- Ultimately, it was tracked down to redundant loads and dead stores existing after optimisation.
- This was due to the existing alias analysis implementations lacking context, and hence suboptimal redundant load and dead store elimination.

```
store i32 36076, i32* %4
%42 = load i64* inttoptr (i64 61931224 to i64*)
%43 = add i64 %42, 6
store i64 %43, i64* inttoptr (i64 61931224 to i64*)
store i32 36076, i32* %4
...
store i32 36092, i32* %4
```

```
movl $37076, 60(%r12)
addq $6, 61931224
movl $37076, 60(%r12)
...
movl $36092, 60(%r12)
```

The Problem

- Fundamentally, we are not compiling a whole program – just snippets.
- The existing alias analysis implementations do not have enough context to work with.
- Making the alias analysis more complex introduces unnecessary delay.

The Solution

- LLVM doesn't distinguish between our simulated memory operations and virtual register operations as they are both essentially accesses to memory, via `getelementptr`s or `inttoptr`s.

Guest Register Write

```
%67 = load i32* %13
%68 = getelementptr %struct.gensim_state* %15, i32 0, i32 0, i8 3
store i32 %67, i32* %68
```

Guest Memory Write

```
%2958 = getelementptr %struct.gensim_state* %15, i32 0, i32 0, i8 1
%2959 = load i32* %2958
%2960 = zext i32 %2957 to i64
%2961 = add i64 %2960, 140522132680704
%2962 = inttoptr i64 %2961 to i32*
store i32 %2959, i32* %2962
```

The Solution

- But, our generator knows all about the guest memory operations, and virtual register operations.

Guest Register Write

```
%67 = load i32* %13
%68 = getelementptr %struct.gensim_state* %15, i32 0, i32 0, i8 3, !aaai !19
store i32 %67, i32* %68
```

Guest Memory Write

```
%2958 = getelementptr %struct.gensim_state* %15, i32 0, i32 0, i8 1, !aaai !18
%2959 = load i32* %2958
%2960 = zext i32 %2957 to i64
%2961 = add i64 %2960, 140522132680704
%2962 = inttoptr i64 %2961 to i32*, !aaai !15
store i32 %2959, i32* %2962
```

- We insert a custom alias analysis pass, which uses the custom metadata to directly reason about pointer aliasing information.

The Solution

- We introduce different aliasing classes for the different memory operations:
  - CPU state/flag accesses
  - Virtual register accesses
  - Guest memory accesses
  - Internal JIT structure accesses
  - ...

```
%jump_location_ptr = getelementptr inbounds void (%struct.cpuState*)** %17, i32 %target_pc_page, !aaai !14
%2962 = inttoptr i64 %2961 to i32*, !aaai !15
%2958 = getelementptr %struct.gensim_state* %15, i32 0, i32 0, i8 1, !aaai !18
%68 = getelementptr %struct.gensim_state* %15, i32 0, i32 0, i8 3, !aaai !19

!14 = metadata !{i32 5}              Region Chaining Table Access
!15 = metadata !{i32 2}              Guest Memory Access
!16 = metadata !{i32 1, i32 0, i8 2}  Virtual Register (r2) access
!17 = metadata !{i32 1, i32 0, i8 7}  Virtual Register (r7) access
!18 = metadata !{i32 1, i32 0, i8 1}  Virtual Register (r1) access
!19 = metadata !{i32 1, i32 0, i8 3}  Virtual Register (r3) access
```

The Solution

```
!15 = metadata !{i32 2}
!18 = metadata !{i32 1, i32 0, i8 1}

%2962 = inttoptr i64 %2961 to i32*, !aaai !15
%2958 = getelementptr %struct.gensim_state* %15, i32 0, i32 0, i8 1, !aaai !18

if (CONSTVAL(md1->getOperand(0)) != CONSTVAL(md2->getOperand(0))) {
  return NoAlias;
}
```

The Solution

```
!18 = metadata !{i32 1, i32 0, i8 1}
!19 = metadata !{i32 1, i32 0, i8 3}

%2958 = getelementptr %struct.gensim_state* %15, i32 0, i32 0, i8 1, !aaai !18
%68 = getelementptr %struct.gensim_state* %15, i32 0, i32 0, i8 3, !aaai !19

// Check the register bank
if (CONSTVAL(md1->getOperand(1)) == CONSTVAL(md2->getOperand(1))) {
  // Check the register index
  if (CONSTVAL(md1->getOperand(2)) == CONSTVAL(md2->getOperand(2))) {
    return MustAlias;
  } else {
    return NoAlias;
  }
} else {
  return NoAlias;
}
```

Challenges

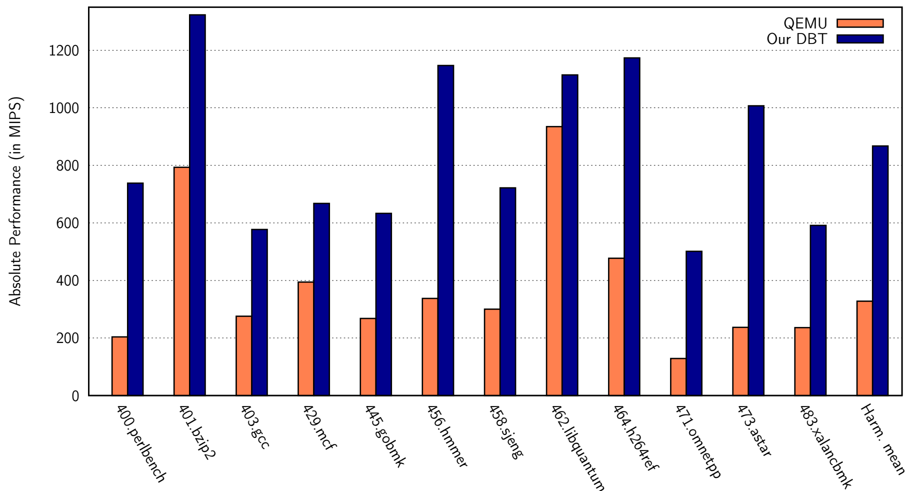
- Inserting a new alias-analysis pass is hard.

```
static inline void add_pass(llvm::PassManager &pm, llvm::Pass *p, bool with_aa)
{
  if (with_aa) {
    pm.add(createArcSimAliasAnalysisPass());
  }

  pm.add(p);
}

pm->add(llvm::createTypeBasedAliasAnalysisPass());

...

add_pass(*pm, llvm::createGlobalOptimizerPass(), with_aa);
add_pass(*pm, llvm::createIPSCCPPass(), with_aa);
add_pass(*pm, llvm::createDeadArgEliminationPass(), with_aa);
```

Comparison to Qemu

- We compare our target instruction throughput to that of Qemu, using the **SPEC2006** and **EEMBC** benchmark suites.
- This is the number of target instructions executed per second, measured in Mips.
- The number of instructions executed is **constant** between Qemu and ArcSim, as we use exactly the same binaries, with exactly the same input.
- Therefore, throughput also directly correlates to total runtime.

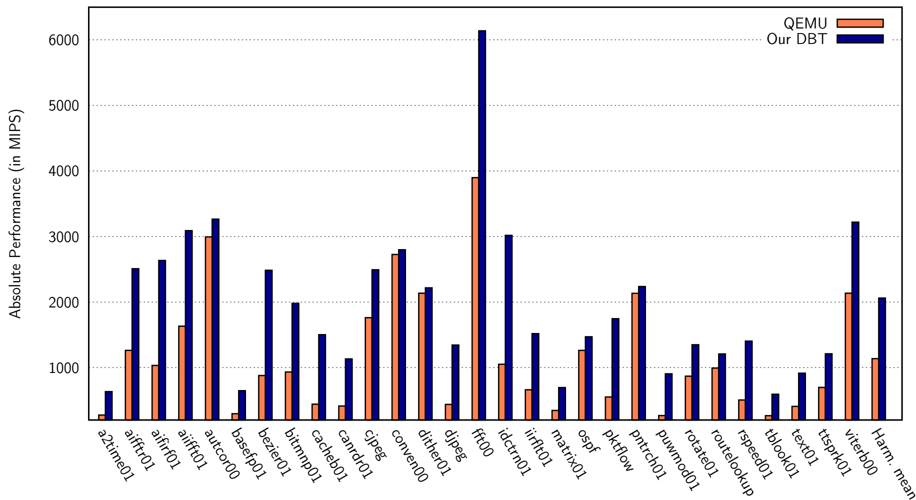
SPEC2006 - Integer Benchmarks



informatics

icsa Institute for Computing Systems Architecture

EEMBC



Additional Highlights

- Partial evaluation.
- Region-entry optimisation.
- Region chaining.
- Parallel compilation task farm.
- Efficient interrupt handling.
- Instruction decode cache.
- Extensive tracing and profiling infrastructure.
- Multi-ISA support.

Demonstration

- Demonstrations may be available across the road in the **Informatics Forum**.