Australian Government
Department of Defence
Defence Science and Technology Group

Lift-*offline*: Instruction lifter generators

SAS 2024, October 2024

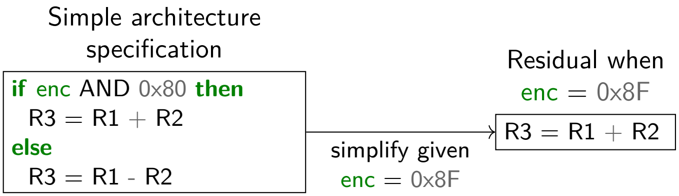
Nicholas Coughlin, Alistair Michael, and **Kait Lam**
DST Group / The University of Queensland

Background

- Reasoning about assembly code requires accurate, trustworthy models of the instruction-set architecture (ISA).
- Existing decompilers use hand-written semantics with limited correctness arguments.
- ARM publishes a machine-readable specification of its ISA (Reid 2016), expressed in ARM Specification Language (ASL).
- However, formal models remain difficult to integrate with analysis tools.

Previous Work

- Previous work: ASLp, an *online* partial evaluator for ASL (Lam & Coughlin 2023)
  - Simplifies given a known instruction encoding (opcode).
  - Validated by differential testing against an ASL interpreter.
  - Found bugs in the instruction semantics of RetDec and Remill.



Simple architecture specification

```
if enc AND 0x80 then
  R3 = R1 + R2
else
  R3 = R1 - R2
```

simplify given enc = 0x8F

Residual when enc = 0x8F

```
R3 = R1 + R2
```

Problems Arising

- Online partial evaluation requires traversing the specification for each opcode.
- Several disadvantages:
  - Tight coupling between downstream projects and ASLp + OCaml runtime + ASL specification.
  - Difficult to reason universally about the produced semantics.
  - Difficult to integrate with analysis tools (different languages and different IRs).

Observations

Aim: generate a standalone lifter in the style of hand-written lifters (a *lifter* is a program which returns instruction semantics, given an opcode).

- ASL specification is structurally similar to a lifter.
- However, a lifter will distinguish two stages:
  - **Lift-time**: execution of the lifter to generate semantics.
  - **Run-time**: execution of the semantics to produce side-effects.
- ASL specification does not differentiate the two, but we can deduce them.

Comparison — ASL vs RetDec

Example: add <Xd>, <Xn>, <Xm>.

```
bits(datasize) result; bits(4) flags;

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

(result, flags) = AddWithCarry(
  operand1, operand2, '0');
X[d] = result;
```

```
void translateAdd(cs_insn* i,
  cs_arm64* ai, llvm::IRBuilder<>& irb)

std::tie(op1, op2) = loadOpBinary(ai);

auto *val = irb.CreateAdd(op1, op2);

storeOp(ai->operands[0], val);
```

An Offline Approach

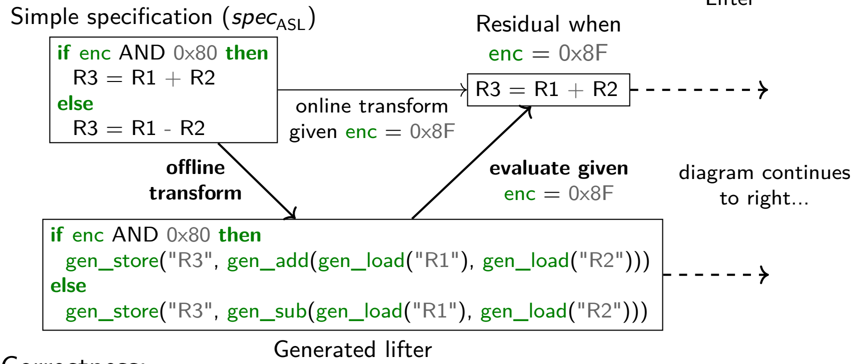
Offline partial evaluation to generate a lifter *ahead-of-time* in two phases.

1. Binding-time analysis:
   - Mark static values (e.g. opcode, constants) as *lift-time*.
   - Propagate and mark computations as *lift-time* if all inputs are lift-time, otherwise *run-time*.
2. Offline transformation:
   - For lift-time operations, no change needed.
   - For run-time operations, instead produce function calls which construct an AST.

Offline vs Online Partial Evaluation

Abstractly, difference between online and offline is akin to currying:

$$\text{online} : \mathit{Spec} \times \mathit{Opcode} \to \mathit{Sem}, \quad \text{offline} : \mathit{Spec} \to \underbrace{(\mathit{Opcode} \to \mathit{Sem})}_{\text{Lifter}}.$$



Simple specification ($spec_{\mathsf{ASL}}$)

```
if enc AND 0x80 then
  R3 = R1 + R2
else
  R3 = R1 - R2
```

Residual when enc = 0x8F

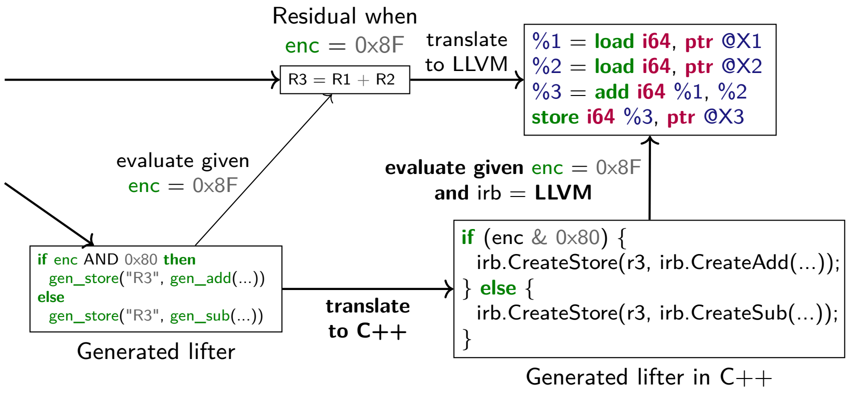
```
R3 = R1 + R2
```

```
if enc AND 0x80 then
  gen_store("R3", gen_add(gen_load("R1"), gen_load("R2")))
else
  gen_store("R3", gen_sub(gen_load("R1"), gen_load("R2")))
```

Correctness:

$$\forall op \bullet spec_{\mathsf{ASL}}(op) \simeq \text{online}(op, spec_{\mathsf{ASL}}) \simeq \text{offline}(spec_{\mathsf{ASL}})(op).$$

Offline vs Online Partial Evaluation (cont.)



```
%1 = load i64, ptr @X1
%2 = load i64, ptr @X2
%3 = add i64 %1, %2
store i64 %3, ptr @X3
```

```
if enc AND 0x80 then
  gen_store("R3", gen_add(...))
else
  gen_store("R3", gen_sub(...))
```

```
if (enc & 0x80) {
  irb.CreateStore(r3, irb.CreateAdd(...));
} else {
  irb.CreateStore(r3, irb.CreateSub(...));
}
```

Translating

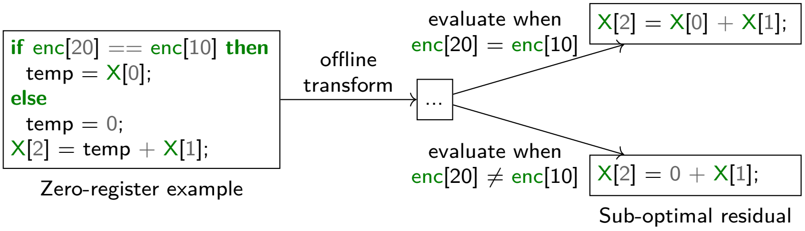
- Generated lifter represents two stages of execution within the same ASL program.

	Lift-time	**Run-time** (deferred)
Literal	0	gen_int("0")
Addition	x + y	gen_add(x, y)
Variable	**bits**(64) x;	x = decl_bv("x", 64);
Branch	**if** c **then** t **else** f	(t,f,j) = gen_branch(c)

- Translating to different lift-time languages is only a syntactic transformation.
- One lifter can target different run-time languages (e.g. by polymorphism or duck typing).

Refining Offline Partial Evaluation

- *Online* partial evaluation is extremely powerful, due to precise knowledge of the opcode.
  - Post-processing can clean up many sub-optimal structures.
- Offline residuals are less amenable to post-processing.
  - Noticeable over-approximation.
  - Simplifications based on algebraic rules are less effective.


```
if enc[20] == enc[10] then
  temp = X[0];
else
  temp = 0;
X[2] = temp + X[1];
```
Zero-register example

Opcode-Sensitive Analyses

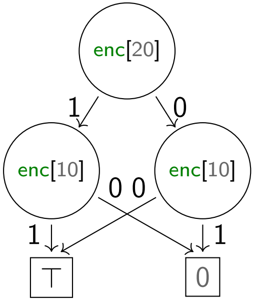
- Difference between online and offline is knowledge of the opcode value.
  - What if we encoded that within the analysis domain?
- Use an *opcode-sensitive* analysis:
  Given an abstract domain $(A, \sqsubset)$ and $\mathrm{tf} : \mathit{Stmt} \to A \to A$, derive $(\mathit{Opcode} \to A, \sqsubset_{op})$ where

$$x \sqsubset_{op} y \equiv \forall i \in \mathit{Opcode} \bullet (x(i) \sqsubset y(i))$$
$$\mathrm{tf}_{\mathsf{op}}(s)(x) \equiv \lambda op \bullet \mathrm{tf}(s[\mathsf{enc} \leftarrow op])(x(op))$$

- Objects of type $\mathit{Opcode} \to A$ are efficiently represented as multi-terminal binary decision diagrams (MTBDD).

Example — Value Analysis

```
if enc[20] == enc[10] then
  temp = X[0];
else
  temp = 0;
X[2] = temp + X[1];
```



- MTBDD represents objects of type *Opcode* $\rightarrow A$.
  Here, $A = \{\bot, 0, 1, \ldots, \top\}$.
- Bits of enc are decision nodes and values of A are terminals.
- Static ordering of decision nodes from most-significant to least-significant bits.

Opcode-Sensitive Analyses — Applications

- Results used to transform lifter to produce more concise semantics.
- For conditionally-applicable simplifications, *split* statements by inserting a branch into applicable and non-applicable cases.
  - Dead-code elimination.
  - Copy propagation.
  - Constant propagation.
- Previous slide's example becomes:

```
X[2] = (if enc[20] == enc[10] then (X[0] + X[1]) else X[1]);
```

Evaluation — Compilation & Lifting Times

- Offline-generated lifters are much faster at lift-time, at the expense of compilation time.

Lifter	Time (s)
Online OCaml 4.14	2.60
Offline OCaml 4.14	63.22
Offline Scala 3.3	95.11
Offline C++	73.35

Table: Compilation times.

		Avg. time (ms)	
Class	Tested	Online	Offline
Branch	189	1.342	0.011
Float	3,846	1.400	0.023
Integer	12,667	1.360	0.023
Memory	29,397	3.762	0.039
Vector	106,863	2.745	0.052

Table: Per-instruction lift time.

Evaluation — Semantics Comparison

- Validated using differential testing against original spec.
- Produced residual programs are semantically equivalent.
- Comparing outputs produced by the offline and online lifters, 32% of the 152,703 tested instructions are textually identical.

	Exprs			
	online		offline	
	mean	max	mean	max
Branch	15.40	19	18.20	27
Float	31.53	53	33.13	80
Integer	245.10	5,183	273.69	5,558
Memory	146.64	1,855	147.06	1,855
Vector	345.43	4,447	391.78	3,879

Table: Online and offline instruction complexity by instruction class.

Evaluation — Verification Performance

- Tested with assertions generated by BASIL, an in-development binary analysis tool.
- Weakest-precondition assertions passed to Boogie then Z3.

	Verification time (s)			Resource count		
	mean	max	std. dev	mean	max	std dev
Offline	0.05	8.12	0.17	29,078	5,380,601	191,142
Online	0.04	1.44	0.09	21,989	4,867,142	181,745

Table: Solving times by SMT solver.

Conclusion

- Developed offline partial evaluation as an extension to ASLp.
- Generates standalone lifters from ARM's ISA semantics.
- Opcode-sensitive analyses and transformations.
- Lifter can be translated to arbitrary languages and IRs.

Future Work

- Implementing further simplifications on the offline lifter.
- More formal verification of the generation pipeline (e.g. translation validation).
- Deriving more useful tools from the architecture specification.
- Exploring additional architecture specifications (e.g. Intel).

Thank you!

https://github.com/UQ-PAC/aslp
“uq pac aslp”

https://doi.org/10.5281/zenodo.13219112
“lift offline artifact”

https://katrinafyi.github.io/aslp-web/
“aslp web arm”