P4.Binary Translation Framework

Nuno Paulino
CTM
INESC TEC

Binary Translation Framework

- Our own previous work:
  - Targeted only MicroBlaze G
  - Generated/supported only one specific type of pipelined loop accelerators
  - Functional (+), but limited (-)

How to explore hardware generation from trace/post-compile information for **more ISAs**, and targeting **more/different accelerator/core designs?**

The purpose of the Binary Translation Stack is to implement this flow.

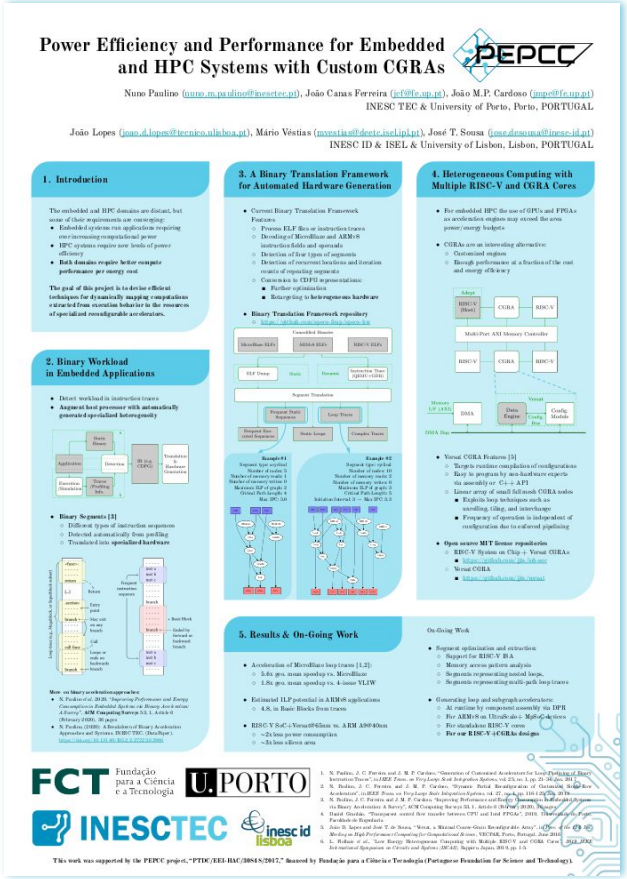
Papers and Demos

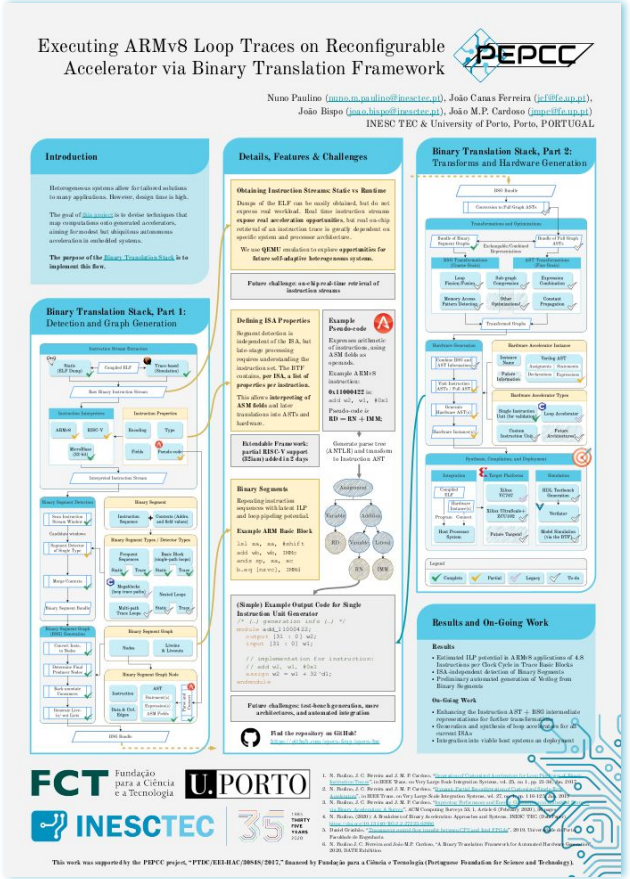
- IEEE Micro Special Issue on FPGAs in Computing
  - N. Paulino, J. Bispo, J. C. Ferreira and J. M. P. Cardoso, "*A Binary Translation Framework for Automated Hardware Generation*," in IEEE Micro, vol. 41, no. 4, pp. 15-23, 1 July-Aug. 2021, doi: 10.1109/MM.2021.3088670.
- FPT'2021
  - T. Santos, N. Paulino, J. Bispo, J. M. P. Cardoso and J. C. Ferreira, "*On the Performance Effect of Loop Trace Window Size on Scheduling for Configurable Coarse Grain Loop Accelerators,*" 2021 Intl. Conf. on FPT, 2021, pp. 1-4, doi: 10.1109/ICFPT52863.2021.9609868.
- FPL'2020
  - N. Paulino, J. C. Ferreira, J. Bispo and J. M. P. Cardoso, "Executing ARMv8 Loop Traces on Reconfigurable Accelerator via Binary Translation Framework," 2020 30th Intl. Conf. on FPL, 2020, pp. 367-367, doi: 10.1109/FPL50879.2020.00072.
- DATE'2020
  - N. Paulino, J. C. Ferreira, J. M. P. Cardoso, J. T. de Sousa, "*Power Efficiency and Performance for Embedded and HPC Systems with Custom CGRAs*", Design, Automation & Test In Europe, Demo Booth, 2020, doi: http://dx.doi.org/10.13140/RG.2.2.14545.97128

Posters

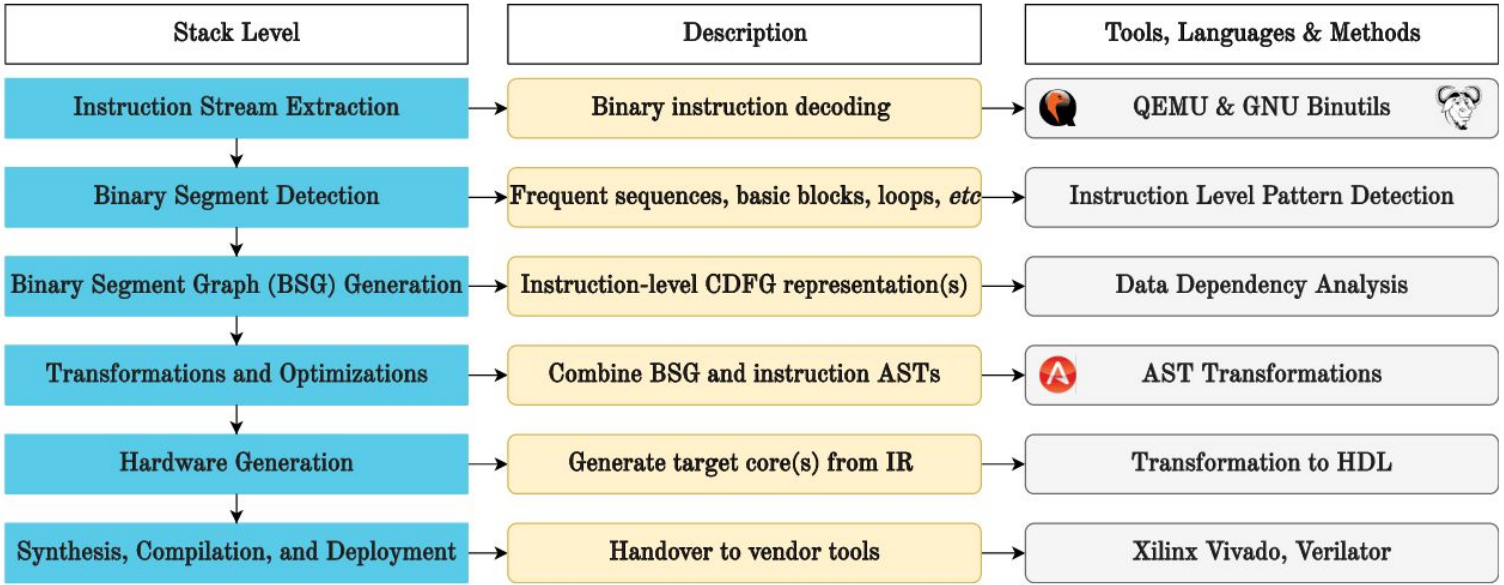
- DATE'2020
  - Holistic view of the project
- FPL'2020
  - Specifically on the Binary Translation aspect

Both include video demos





Framework Stack



- Implemented in Java
- Starts by analysis of ELF file, or trace dump
- Produces CDFGs of repetitive patterns
- Outputs arbitrary Verilog (generated programmatically)

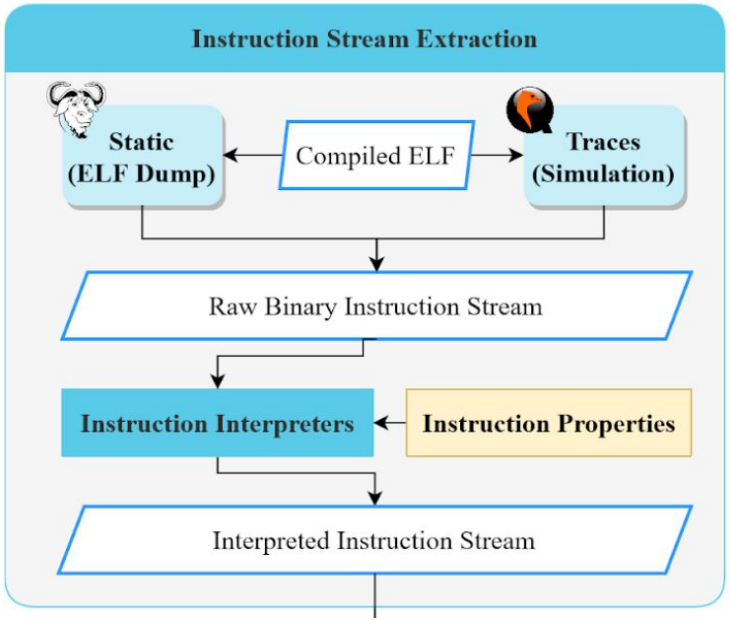
Instruction Stream Parsing

- Types of instruction streams
  - Static analysis (via objdump)
  - Trace analysis (via QEMU + GDB)

- ISA Decoders
  - MicroBlaze (32 bit)
  - ARMv8
  - RISC-V (32IMAF)

- Requires QEMU and GNU utils for all supported ISAs...
  - (Some effort... including bug reports to Sourceware)

Phase 1 - Decoding



Instruction Stream Parsing

- Standardized properties per ISA
  - "*InstructionProperties*" interface
  - Encoding
  - Binary format
  - Generic type classifier for later stages

- Lists of parsers for formats of an ISA
  - Easy to read and specify
  - Named parsed fields, literal fields, and D/C
  - Priority ordered
  - Support predicates; useful for sub-encodings and particular corner cases

```
public enum RiscvInstructionProperties
  implements InstructionProperties {

  // R-type: fn7|rs2|rs1|fn3|rd|0110011
  add(0x0000_0033, R, G_ADD),
  sub(0x4000_0033, R, G_SUB),
  sll(0x0000_0833, R, G_LOGICAL),
  // (...)
}
```

```
public interface ArmInstructionParsers {
  // (...)
  List<AsmParser> PARSERS = Arrays.asList(
    newInstance(DPRTWOSOURCE,
      "sf(1)_0_opa(1)_11010110"
        + "rm(5)_opb(6)_rn(5)_rd(5)"),
    newInstance(CONDITIONALBRANCH,
      "0101010_opa(1)_imm(19)"
      + "opb(1)_cond(4)"),
    // (...)
  );
  // (...)
}
```

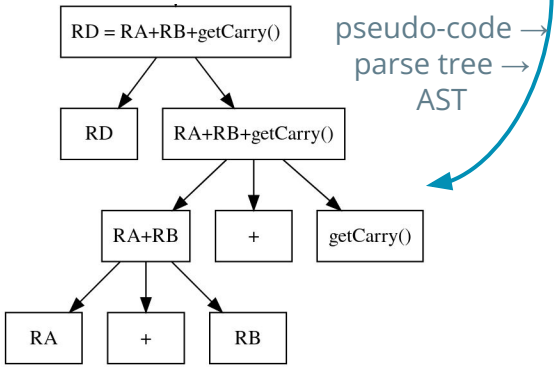
Instruction Stream Parsing

- But what does an instruction do?
  - Parsing the fields, field values, and mnemonic (e.g. *add*) is insufficient

```
public enum MicroBlazePseudocode
  implements InstructionPseudocode {

  add("RD = RA + RB;"),
  addc("RD = RA + RB + getCarry();"
      + "setCarry(msb(RD));"),
  br("$pc = $pc + RB;"),
  idivu("if(RA == 0) {RD = 0; $dzo = 1;}"
      + " else {RD = RB / RA;}"),
  // (...)
}
```

- Solution: ISA-independent behaviour specification
  - In-house g4 syntax processed by ANTLR4
  - “Pseudo-Instruction”
  - Specifies the actual interaction between fields of any instruction of any ISA



Instruction Stream Parsing

- Quick look at a g4 syntax file (partial)
- Java tokenizer and parser are generated by ANTLR4
- Transforming the parse tree into our own AST is done by our code

```
 * Copyright 2020 SPeCS.

/*
 * The purpose of this grammar is to be able to express the operation of
 * an instruction using the ASM fields as operators
 */

grammar PseudoInstruction;

@header {

 * Parsing
pseudoInstruction : statement+;

/* Question mark stands for: zero or one

statementlist
    : statement
    | LBRACE statement* RBRACE;

statement
    : expression STATEMENTEND                                                          # plainStmt
    | 'if' LPAREN condition=expression RPAREN ifsats=statementlist                      # ifStatement
    | 'if' LPAREN condition=expression RPAREN ifsats=statementlist ('else' elsestats=statementlist) # ifElseStatement;

expression
    : operand                                                  # variableExpr
    | LPAREN expression RPAREN                                 # parenExpr
    | functionName LPAREN arguments? RPAREN                    # functionExpr
    | operator right=expression                                # unaryExpr
    | left=expression operator right=expression                # binaryExpr
    | left=expression rlop right=expression                    # assignmentExpr;

rlop: EQ;
```

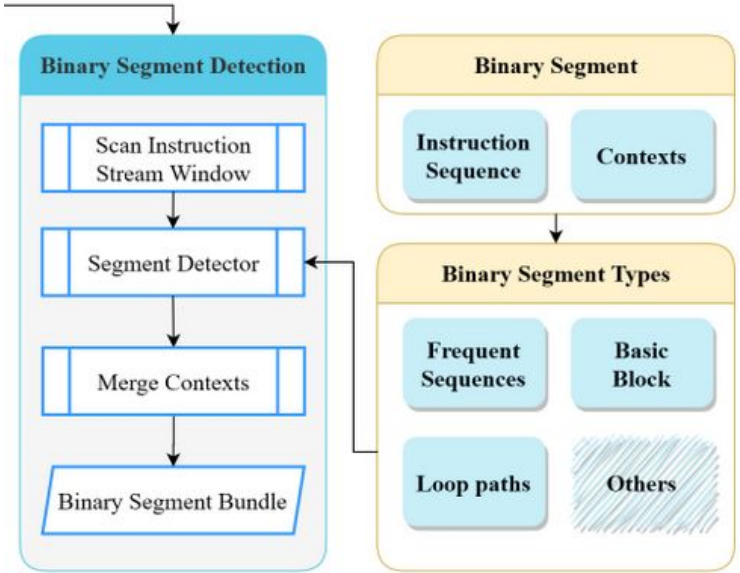
Binary Segment Detection

- Idea: detect repetitive sequences of instructions
  - → generate HDL to exploit parallelism

- Types of sequences
  - Frequent sequences
  - Basic Blocks
  - Megablocks (from previous work)
  - Other types? (e.g., nested loops or merged megablock paths)

- Sequences can be static or trace based
- How do we defined that two sequences are **equal**?

Phase 2 - Detection



Binary Segment Detection

- Detection is performed over a candidate window of max size
  - Using a hash to define equality
  - Changing the hash simplifies equality condition and re-uses detection code

- Example for frequent static sequences
  - Same sequence can be “different” if registers different but CDFG would be the same. Solution?
    - Abstract the registers

```
(...)
50b8:58c6c100 fmul r6, r6, r24
50bc:594a3000 fadd 10, r10, r6
50c0:d9432800 sw   r10, r3, r5
(...)
50cc:58e7c100 fmul r7, r7, r24
50d0:58c63800 fadd r6, r6, r7
50d4:d8c39800 sw   r6, r3, r19
(...)
```

2 sequences at 2 different addresses implement the same dataflow despite different registers

```
fmul r<a>, r<a>, r<b>
fadd r<c>, r<c>, r<d>
sw   r<c>, r<e>, r<f>
```

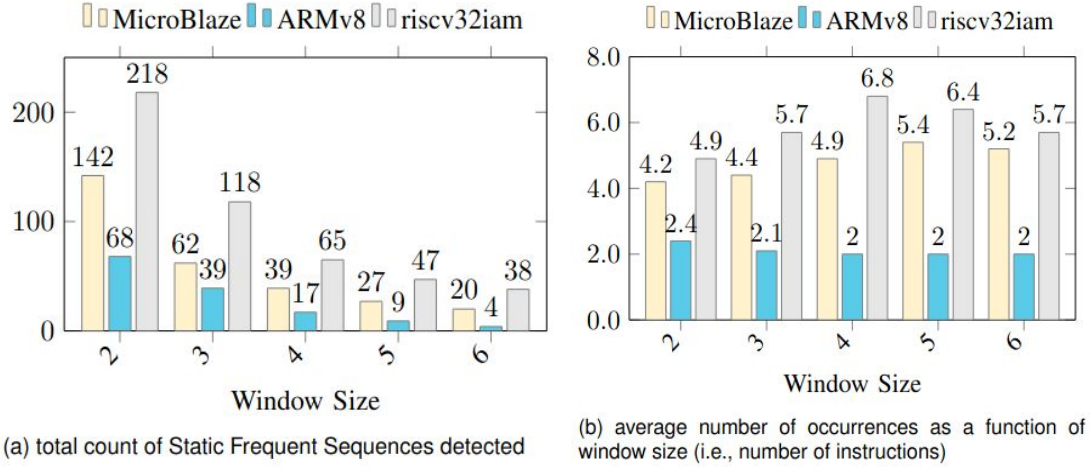
Binary Segment Detection

- Part of the FPL2020 demo (6:40m to end demonstrates detection)

Binary Segment Detection

- From IEEE Micro paper
- 3 ISAs and 13 kernels from Polybench
- 2 cases:
  - static acyclic sequences of sizes 2 to 6 (this slide)
  - trace basic blocks (next slide)

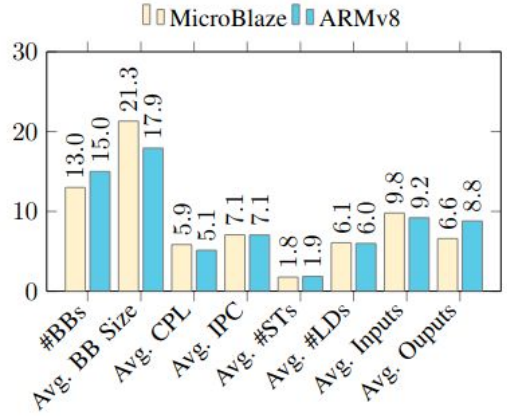
- On avg., per sequence:
  - *loads* → 30-50%
  - *stores* → 10-50%



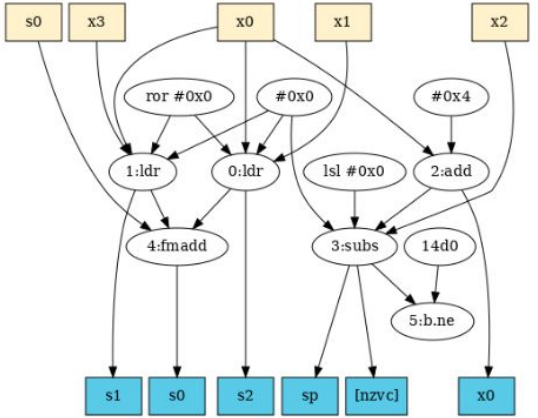
Detected static frequent sequences for the supported ISAs

Graph Generation

- Trace basic blocks
  - Fully simulated the 13 kernels for all 2 ISAs
  - Searched for basic blocks sizes 4 to 50
  - Collected average metrics
- CDFG generation
  - Analysing when registers are last written
  - Identifying live-ins, live-outs
  - II is computed based on longest backwards connection



(a) Summary metrics for the graphs of detected hot basic blocks



(b) CDFG for the basic block detected in the *inner product* kernel, for ARMv8 (CPL=3, IPC=2, II = 3)

Summary metrics for the graphs of detected hot basic blocks

(RISC-V execution in QEMU failed with floating-point kernels... since solved!)

Hardware Generation

- Analyse CDFGs of inst. sequences
  - Process level by level, top to bottom
  - Perform rudimentary SSA
  - Analyse AST of each **node** (i.e. inst.)
  - This example simply emits combinatory blocks per CDFG level
    - Single cycle module

- Recent
  - Generation of Verilog changed significantly from the version used to produce this output (see session P8)

```
// Sequence occurs at = [ 0x2e8c(256) ]
// addk  r6,  r8,  r4
// bslli r5,  r4,  0x2
// bslli r6,  r6,  0x2

module trace_frequent_sequence_103887628;
input [31 : 0] r8, r4;
output [31 : 0] r5_0, r6_1;

// level 0 of graph (2 nodes)
always_comb
  r6_0 = ( r8 + r4 ); // node 0:addk
  r5_0 = r4 << 2;     // node 1:bslli
end

// level 1 of graph (1 nodes)
assign r6_1 = r6_0 << 2; // node 2:bslli
endmodule
```

Hardware Generation

- On-going
  - Graph analysis of memory access patterns and AGU extraxtion
  - Integration with synthesis and simulation backends (via TCL or others)
  - Better programmatic Verilog generation via internal DSL

- Future aspects to consider in automated hardware generation
  - Memory Accesses (i.e., patterns, partitioning, coalescing, custom cache systems etc)
  - Implementation of branches / predication / multiple-paths
  - Multiple Configurations (methods of reconfigurability and granularity)
  - Fully custom HDL (compliant to some interfaces) vs. Template specialization
    - i.e., low control vs requiring some architecture specific “compiler” (e.g., for CGRAs)

Conclusion

- A “sandbox” tool for exploring
  - Different IRs
  - Optimizations of IRs
  - Methods for workload representation
  - Different target hardware
    - by functional simulation
    - by emitting to RTL

- Hoping to focus on RISC-V based architectures in future