A Binary Translation Framework for Automated Hardware Generation

Nuno Paulino, *Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência, 4200-465, Porto, Portugal*

João Bispo, João C. Ferreira, and João M. P. Cardoso, *Faculdade de Engenharia da Universidade do Porto, 4200-465, Porto, Portugal*

As applications move to the edge, efficiency in computing power and power/energy consumption is required. Heterogeneous computing promises to meet these requirements through application-specific hardware accelerators. Runtime adaptivity might be of paramount importance to realize the potential of hardware specialization, but further study is required on workload retargeting and offloading to reconfigurable hardware. This article presents our framework for the exploration of both offloading and hardware generation techniques. The framework is currently able to process instruction sequences from MicroBlaze, ARMv8, and riscv32imaf binaries, and to represent them as Control and Dataflow Graphs for transformation to implementations of hardware modules. We illustrate the framework's capabilities for identifying binary sequences for hardware translation with a set of 13 benchmarks.

The emergence of heterogeneous devices as platforms for data-intensive algorithms can be attributed to three major factors: 1) the stagnation of compute performance on conventional multicore central processing units (CPUs); 2), emergence of field-programmable gate arrays (FPGAs) with greater logic cell density, operating frequency, integrated specialized cores, and better development tools; and 3) the push toward edge computing, backed by Internet-of-Things and artificial intelligence applications, which can benefit from reconfigurable or heterogeneous computing through FPGAs. However, the use of these devices is limited by the development effort required to exploit hardware specialization. We summarize the technology trends underlying the emergence of heterogeneous platforms and provide an overview of a framework for exploring binary translation-based acceleration techniques applicable to future self-adaptive systems that shift work to available reconfigurable resources at runtime.

0272-1732 © 2021 IEEE
Digital Object Identifier 10.1109/MM.2021.3088670
Date of publication 11 June 2021; date of current version 2 July 2021.

TRENDS IN DESKTOP PROCESSOR TECHNOLOGY

Figure 1(a) illustrates six features of high-end processors over the last five decades. The data points represent the average of the 30 highest end devices for each year. Near 2005, the end of Dennard scaling lead to a stagnation of single-core processor performance. The exponential increase of transistor density and core frequency cause unsustainable thermal dissipation, known as the *Power Wall*[2].

Multicore processors were introduced to enable further performance scaling, by exploiting data and process parallelism. However, this era encountered its own limit in Amdahl's Law. That is, performance does not scale with thread count.

Consider the recently released Apple M1 System-on-a-Chip (SoC). Despite the state-of-the-art 5-nm process, the CPU clock frequency remains stagnated at 3.2 GHz. Instead, the claimed 3.5× performance increase was enabled by reduction of communication overheads, via single-chip integration of heterogeneous components, and tightly coupled hardware/software codesign. In future, such alternative approaches of increased specialized heterogeneity and integration will be paramount to ensure performance scaling.

Finally, the latent instruction level parallelism (ILP) in compiled code is still not fully explored, despite

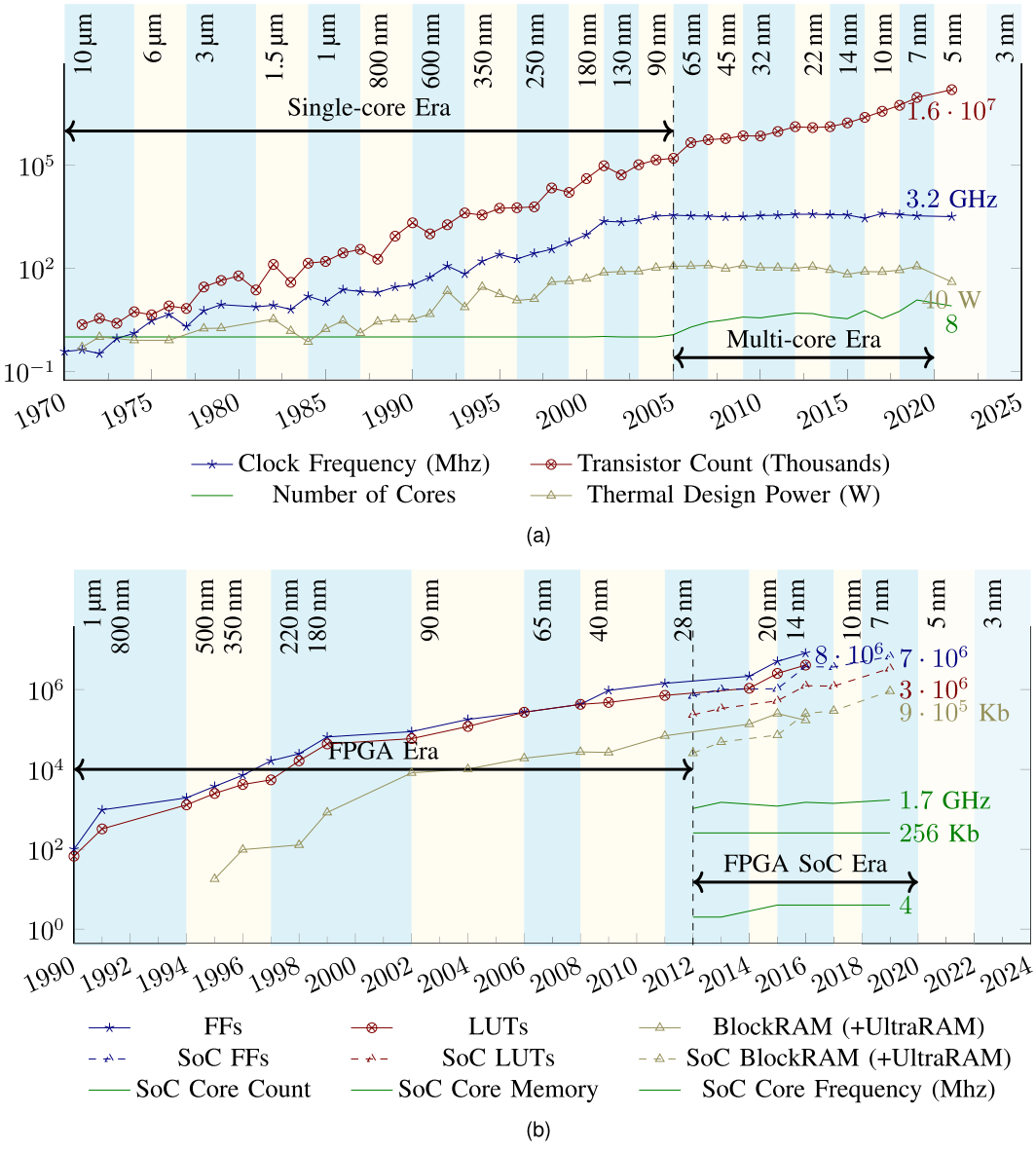

FIGURE 1. Technology and characteristic trends for CPUs and FPGAs.

multiple-issue units, out-of-order execution, vectorization, and branch prediction.[3] Therefore, there are two barriers to further improvement, one technological and one architectural, that justify the exploration of novel approaches to better explore parallelism, heterogeneity, and memory-centric computing. FPGAs are seen as likely platforms for this future performance scaling.

TRENDS IN FPGA TECHNOLOGY

Figure 1(b) summarizes information analogous to Figure 1(a) for FPGA technology for the last 30 years. The characteristic of FPGAs that relate to performance are the amount of flip-flops (FFs), lookup tables (LUTs), and on-chip memory. Since FPGAs are reconfigurable at the logic gate level, the maximum operating frequency is design dependent. However, designs typically operate at less than 1 GHz. The performance gains come from workload-specific designs that exploit parallelism and data streaming.

Although they appeared 20 years after CPUs, FPGAs now use the latest process nodes and achieve comparable transistor densities. AMD currently supplies the CPU with the highest transistor count of 38.5 billion (at 7 or 12 nm), while the densest FPGA contains 43.3 billion transistors (at 14 nm). Like the transistor

density of CPUs, the number of logic elements in FPGAs increases exponentially with the process node. The higher end device of the first family of FPGAs (Xilinx XC2000) contained 100 4-input LUTs, while the latest FPGAs contain up to 4 million 6-input LUTs.

ALTHOUGH THEY APPEARED 20 YEARS AFTER CPUs, FPGAs NOW USE THE LATEST PROCESS NODES AND ACHIEVE COMPARABLE TRANSISTOR DENSITIES.

The most significant change in FPGAs occurs near 2012, with an architectural change analogous to the multicore era of CPUs. Namely, the emergence of the SoC FPGA weakened the distinction between general purpose SoCs and FPGAs. No FPGA device after 2012 lacks an integrated processor system and multiple heterogeneous function-specific cores, such as Ethernet, cryptography, real-time processor, and mobile-grade graphics processing unit.

Consequently, the reconfigurable fabric portion of the chip becomes the platform for application or function specific circuits for the software applications executing on the processor system. Therefore, the study of techniques for the generation of these circuits is important and timely. We specifically focus on techniques based on binary translation, to support runtime offloading of workload to hardware.

COMPILATION FOR FPGAs: From Code to Circuits

As noted by Trimberger,[4] the use of FPGAs is intimidating due to the complexity of the design process and associated toolflow, even though the capabilities of the hardware meet the needs of computationally intensive applications.

High level synthesis (HLS) tools have contributed significantly to FPGA adoption, but still require significant knowledge of hardware design and code rewriting to fully optimize solutions. The design effort is reasonable for well-defined functions whose underlying computations can exploit a data stream model so that efficient deep pipelines can be generated.

However, other work has shown that significant improvements can be achieved if binary instructions are instead redirected to user-defined circuits.[1] We focus on this paradigm to promote future self-adaptive systems capable of offloading computation to reconfigurable resources. For example, runtime configuration of user-defined instruction units[5] or loop scheduling in coarse-grained reconfigurable arrays (CGRAs). Although these designs promise to outperform GPUs, current high level synthesis (HLS) tools cannot infer CGRAs, and dense mapping of operations to these architectures is still an unresolved issue.[6,7] Large instruction traces extracted at runtime can provide a means of speedup that exploits these architectures more efficiently while removing these concerns from software design. The resulting speedup is not expected to compete with compiler-driven or manual circuit design, but it would be ubiquitous and provide a speedup for embedded edge applications. Moreover, studies have shown that CPUs spend most of the energy on memory accesses, instruction fetching and decoding, and out-of-order mechanisms.[7] Significant power savings are thus possible since computationally specific circuits do not require these features.

Our previous experiments based on the proposed approach were focused on contiguous repeating sequences of instructions as segments offloaded from a CPU execution (MicroBlaze) to an accelerator consisting of a row of functional units and with modulo scheduling.[8] CPU and accelerator shared the data memory, and register values were transferred using special CPU ports accessed via get/put instructions. This binary translation scenario achieved speedups from 1.5× to 18.9× and a geometric mean speedup of 6.6× for 13 floating-point kernels.[9] We now present the ongoing work on a framework which iterates on our binary translation techniques, to expand their application to other instruction set architectures (ISAs) and accelerator architectures, and to explore new techniques with focus on memory-centric optimizations.

The remainder of this article explains the processing stack of our framework for study of binary translation techniques, which is geared toward multiple ISAs and backend target architectures. We present examples of profiling exploration, workload extraction, and hardware generation as they are currently performed, and outline aspects of ongoing and future development steps. We hope to provide an insight into techniques for the translation of instruction traces to accelerators.

BINARY TRANSLATION FRAMEWORK: STACK

The binary translation framework (BTF), summarized in Figure 2, is implemented primarily in Java, and available as open source.[a]

[a]Nuno Paulino, João Bispo, "Hardware-Related Libraries and Applications," 2020, GitHub repository, https://github.com/specs-feup/specs-hw

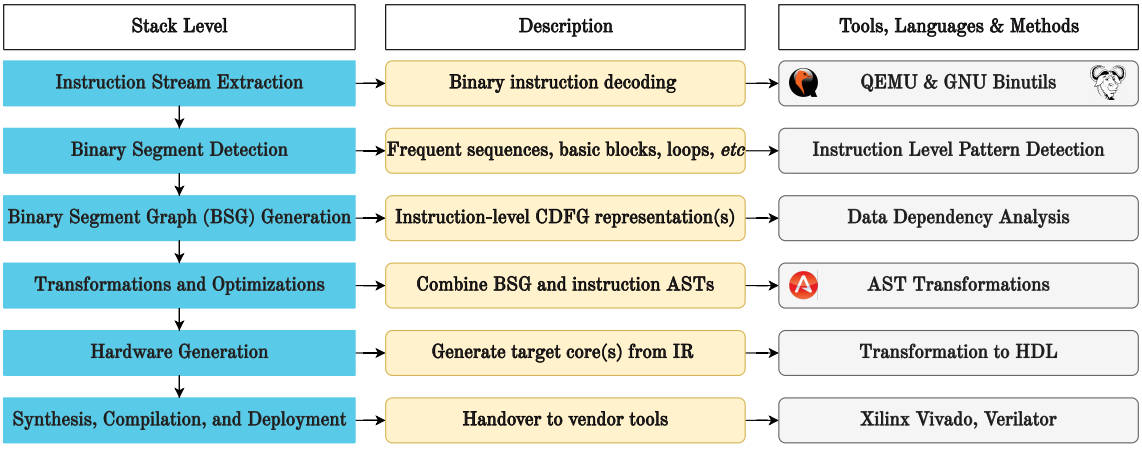

FIGURE 2. Overview of the binary translation stack.

The framework is a iteration on a previous approach for loop accelerator generation for the MicroBlaze processor, based on trace loop detection and runtime offloading.[8] The main purpose of the BTF is to detect *binary segments* (i.e., sequences of instructions terminated by one or more control instructions) from an incoming instruction stream, perform optimizations, and *translate* them to hardware descriptions for the synthesis, validation, and integration.

Most of the processing steps are independent of the ISA of the incoming instruction stream. To support additional ISAs, extensions can be provided to the core of the framework as lists of instruction encodings, operands, and pseudocode implementations. These implementations describe instruction behavior as a function of named instruction fields. As later framework stages operate on intermediate representations (IRs) that are ISA independent, there is the potential to quickly explore hardware generation for new ISAs.

To generate HDL descriptions, some well-known approaches start from source code, rely on existing IRs (e.g., LLVM) and develop back-ends for the hardware generation.[10] Instead, we interpret native instructions from the target ISAs directly for multiple reasons. Since our intent is to explore binary offloading techniques for future self-adaptive capabilities, native machine code is the information we expect to have during runtime on-chip translation. Past approaches have validated this method, within certain constraints.[1] We also wish to target large instruction windows from traces, which requires execution of native code. For complementary exploration, we may also use the framework as a future front-end to emergent IRs geared for the description of hardware and parallelism.[11]

The following sections describe the function and current implementations of the most developed stages. These are the instruction stream extraction, segment detection, control and dataflow graph (CDFG) generation, and hardware generation stages. For preliminary results, we used a set of 13 floating-point kernels from the Livermore Loops set,[9] compiled for three ISAs: MicroBlaze, ARMv8, and *riscv32imaf*. The optimization level used was *-O2* for all ISAs, and the availability of hardware floating-point units is assumed.

INSTRUCTION STREAM PARSING

The framework supports static streams obtained from the analysis of a compiled executable and linkable format (ELF) file or traces obtained from execution. To obtain instruction streams from program execution, the framework runs QEMU under a GNU Debugger (GDB) session. The BTF launches GDB and retrieves every executed address and instruction tuple.

The implemented instruction interpreters, which are required for binary sequence detection, are independent from the source of the instruction stream. However, for consistency, the GNU toolchains and the architecture-specific QEMU emulators are used to obtain static and trace-based streams for all supported ISAs. Currently, these include significant subsets of the 32bit MicroBlaze from Xilinx, the 32-b ARMv8, and the *riscv32ima* subset of the RISC-V specification.

Each instruction is characterized by its encoding, its constituent fields, and a type classification. For example, the following information is required for RISC-V instructions:

```
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),
  // (...)
}
```

The first field is the instruction encoding, and the second refers to the instruction format. Any bits which compose the *opcode* are set according to each instruction, while other fields such as operands are ignored during parsing. The last argument is a list of generic instruction types used during later processing steps.

The various instruction formats used in the ISA must also be specified. For example, for the ARMv8 ISA, the formats for operations on two register values and for conditional branches are specified by the following string-based formats:

```
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)"),
    // (...)
  );
  // (...)
}
```

The first argument indicates the name of the instruction format. The second argument specifies all the fields, along with their bit widths. Bit fields such as operands are named for further decoding in later stages. Literal binary fields allow for distinction between overall instruction format.

Each instruction requires the description of its implementation in an ISA-independent language. The encoding and interpretation of operand fields only allows for determining which fields are being used by the instruction as inputs or outputs. The pseudocode specifies the arithmetic, logical, and control behavior of the instruction. For example, the description of some MicroBlaze instruction is the following:

Instruction behaviors can be specified using bit-fields as operands, along with all arithmetic and logical operators, and also *if-else* clauses. Fields not encoded in the instruction format can be addressed with the $prefix, such as the *carry* bit or the program counter. A limited set of built-ins (e.g., *getCarry*) aids in the

```
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;}"),
  // (...)
}
```

specification. The descriptions allow for compact and extensible representation of instruction behavior, and are used to generate abstract syntax trees (ASTs) in later transformation and hardware description language (HDL) generation steps.

BINARY SEGMENT DETECTION

Segment detection is the process of identifying repetitive sequences of instructions using either static or trace streams. The current detectors differ only in the criteria that determine whether a given candidate window is valid, and share most other processing steps. Some types of segments are only applicable to trace streams, such as recurring loop paths. Together with a list of contexts, each repeating pattern forms a binary segment. A context is the set of values of the instruction fields at each address at which the sequence occurs. The BTF was designed to detect and translate several types of binary segments, which can be distinguished mainly by the type of control flow permitted.[1]

1) **Frequent sequences** are lists of an arbitrary number of instructions, which may not contain any kind of branch instructions.
2) **Basic blocks** are lists of an arbitrary number of instructions ending in a single backward branch to the first instruction in the sequence.
3) **Megablocks** are a type of sequence we have explored in previous work.[8] They are single-path loop traces spanning multiple backwards or forwards branches, ending in a backwards branch to the first instruction in the sequence.
4) **Other types** of segments can be implemented in future work, e.g., extending Megablocks to represent multiple loop paths, or capturing nested loops by rerolling the innermost loops in traces.

Segments are detected by capturing candidate windows of a fixed size, and creating a hash from the instruction information in the window. When a valid candidate window matches an existing hashed

window, its occurrence count is incremented. After the instruction stream has ended, sequences that occur only once are discarded.

The identification of similar sequences does not use strict equality, as this criterion would consider the two following sequences to be different:

```
(...)
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
(...)
```

Instead, we abstract away the operands, so that only the relative position of operands determines the equality condition of two segments at different addresses. With this criterion, the two sequences as considered equivalent, and the respective CDFGs will also be isomorphic.

Currently, the framework can detect two types of sequences, namely frequent sequences and basic blocks, either via static or trace analysis. To detect sequences of different sizes, multiple detectors configured with different parameters can be executed in separate threads over the same incoming instruction stream.

Consider the detection of static frequent sequences for the set of 13 kernels, for all three ISAs. These sequences are those that occur more than once within the statically analyzed assembly code. Since each application executes identical startup code, detection was performed within the address range of the kernel functions. Twenty static frequent sequence detectors ran in parallel, for window sizes ranging from 2 to 20. Figure 3 shows detection results for window sizes from two to six. Each bar represents the total number of such sequences for each window size, per ISAs.

The longest sequences found are of size 11, except for ARMv8, where no sequences of size greater than 6 were found, as the greater heterogeneity of this ISA makes repeated sequences less likely. The number of *load* instructions represents between 30% and 50% of all instructions for all window sizes, and the number of *store* instructions ranges from 10% and 50%. The number of memory instructions relative to other types quantifies how memory bounded the processing is, a characteristic that is important for the development of specialized architectures.

GRAPH GENERATION

To investigate any speedup potential, CDFGs are generated that expose instruction parallelism and

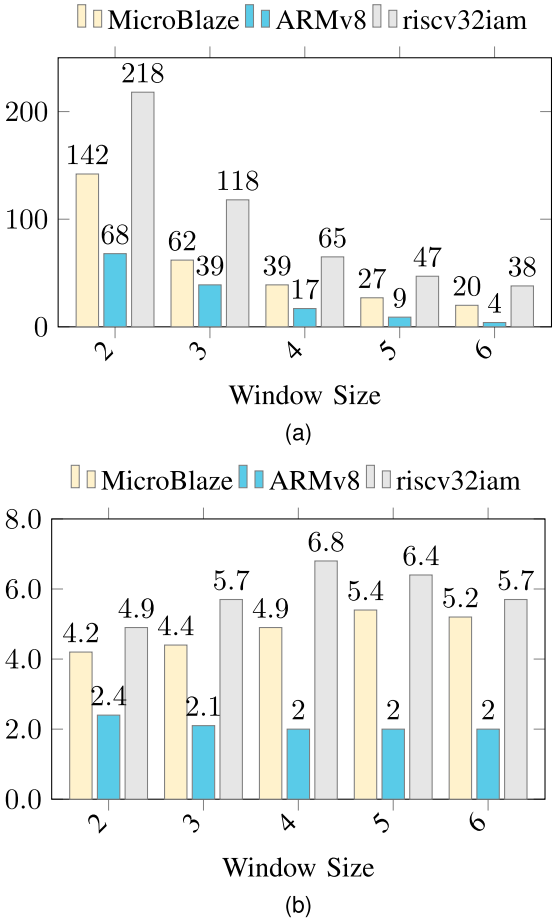

FIGURE 3. Detected static frequent sequences for the supported ISAs.

iteration pipelining. To generate a graph, operands from the processor's register file are replaced with connections to the output of previous instructions. The framework analyzes each instruction sequence in turn and, for each instruction, pulls the last generator of the required operand values and performs this replacement.

For example, consider the detection of hot basic blocks for the same benchmark set, for MicroBlaze and ARMv8. Detectors for window sizes from 4 to 50 were run for each benchmark.

Figure 4(a) shows the total number of basic blocks (#BBs) found, and the averages for metrics related to the respective CDFGs. critical path length (CPL) is the CDFG depth; instructions per clock (IPC) represents acceleration potential, and is computed as the number of instructions over the initiation interval (II). The II of a loop dictates the sequential stages that must execute to begin a new iteration. For nearly all detected basic blocks, the II is three.

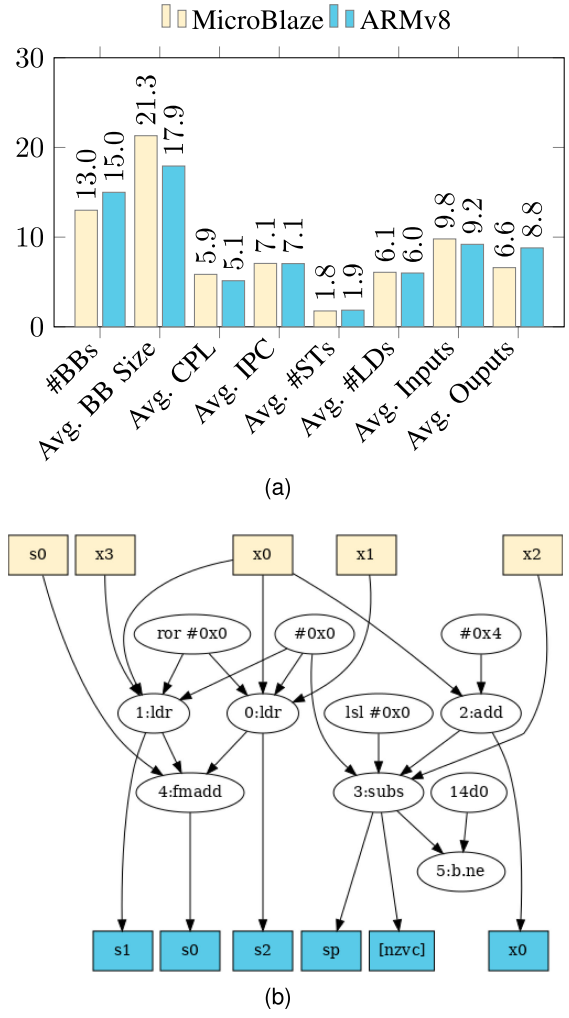

FIGURE 4. Detection of trace basic blocks (for MicroBlaze and ARMv8).

Figure 4(b) shows the CDFG for the segment in the *inner prod* benchmark for the ARMv8. Instructions without data dependencies are placed at the same level, while values exchanged with the processor's register file are shown at the top and bottom. The *b.ne* instruction is the backward branch of the loop. Assuming a model where the branch conditions of an iteration must be evaluated before a new iteration is initiated, this CDFG has a II of 3, and an IPC of 2.

As an additional set of preliminary results, we compiled the seven BLAS kernels of the PolyBench Benchmark Suite[12] for the MicroBlaze and performed trace basic block detection via the BTF, producing 13 CDFGs. From the number of clock cycles required for their software execution, and the respective latent ILP and II, an estimated geometric mean speedup of 9.3× was reported by BTF, which is in line with the results obtained with our accelerator-enhanced architectures.[8]

HARDWARE GENERATION

To generate hardware descriptions, we perform transformations over the target CDFGs, which includes analyzing the AST of each node. These ASTs are built by parsing the description of each instruction and applying to each operand the concrete values extracted from the interpretation of the instruction. Currently, we can translate such trees to Verilog ASTs and output the corresponding code. We currently do not address integration, but we target system architectures where either the accelerators and host processor fully share data memory/caches or the accelerators are integrated into processor pipeline, so as to avoid costly data transfers which might negate any benefits.

The following example illustrates the generated HDL for a sequence implemented as a single-cycle hardware module.

Currently, different accelerator architecture targets remain to be explored. Factors to consider in this exploration include the following.

```
// 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
```

Memory Accesses: The scheduling of memory accesses is greatly dependent on the overall system architecture. Automated design of a specialized memory architecture is one of the exploration objectives of the BTF. Retrieving information regarding the memory accesses performed by the segments will aid the automated generation a specialized memory architectures on a per-case basis.

Implementation of Branches: Sequences with branches require additional hardware to ensure proper execution of the segment, e.g., evaluating execution paths or conditional memory accesses. In

addition, segments with multiple branch instructions (i.e., exit points) can be implemented by discarding partial executions of a segment or by implementing the control required to support arbitrary exit points.

Multiple Configurations: Accelerating multiple segments can be addressed by either generating multiple single-function accelerators, or by generating a single reconfigurable accelerator.[8] The former option introduces architecture design concerns regarding the interfaces between the accelerators, the memory ports, and the host processor. The latter choice introduces additional complexity in generating the accelerator hardware.

Specialized Hardware Generation Versus Templates: The exploration of target accelerator architectures includes studying the benefits of generating a specialized description for one segment or a set of segments, potentially of different types versus scheduling operations onto an existing configurable template. The former approach promotes specialization, while the latter may simplify the translation process. Memory access behavior also influences the choice of target architecture, e.g., full-custom pipelines for predictable data streaming or coarse-grained arrays equipped with distributed memories for workloads with localized and parallelizable data accesses.

CONCLUSION

This article presented a high-level view of our model for a binary translation framework (BTF). We developed the processing stages of the framework with the intention of using it as a compilation and research tool for exploring automated hardware generation approaches starting from object code (static and dynamic).

Currently, the BTF can process instruction streams from the 32-b MicroBlaze, ARMv8, and *riscv32imaf* ISAs. We have demonstrated the detection and extraction of repeating instruction patterns of several types and respective metrics on their characteristics. The latent acceleration potential for the analyzed ISAs is demonstrated. We are currently capable of generating ASTs from the instructions within these patterns, which are used to generate HDL code.

Future development includes detection of more complex segment types, optimizations such as removal of *load/store* operations based on memory accesses analysis, generation of test benches for the output HDL, and progression towards integration, deployment, and testing.

ACKNOWLEDGMENTS

This work was supported by the PEPCC project, "PTDC/EEI-HAC/30848/2017," financed by Fundação para a Ciência e Tecnologia (FCT).

REFERENCES

1. N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Improving performance and energy consumption in embedded systems via binary acceleration: A survey," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–36, Feb. 2020.
2. T. M. Conte, E. P. DeBenedictis, P. A. Gargini, and E. Track, "Rebooting computing: The road ahead," *Computer*, vol. 50, no. 1, pp. 20–29, 2017.
3. E. Fatehi and P. V. Gratz, "ILP and TLP in shared memory applications: A limit study," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2014, pp. 113–125.
4. S. M. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," *Proc. IEEE*, vol. 103, no. 3, pp. 318–331, Mar. 2015.
5. C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," *ACM Trans. Reconfigurable Tech. Syst.*, vol. 4, no. 2, pp. 18:1–18:28, May 2011.
6. A. Podobas, K. Sano, and S. Matsuoka, "A survey on coarse-grained reconfigurable architectures from a performance perspective," *IEEE Access*, vol. 8, pp. 146 719–146 743, 2020.
7. L. Liu *et al.*, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Comput. Surv.*, vol. 52, no. 6, pp. 1–39, Oct. 2019.
8. N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Dynamic partial reconfiguration of customized single-row accelerators," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 1, pp. 116–125, Jan. 2019.
9. T. Peters, "Livermore Loops coded in C," 1992, Accessed: Apr. 1, 2021. [Online]. Available: http://www.netlib.org/benchmark/livermorec
10. A. Canis *et al.*, "Legup: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 2, pp. 1–27, 2013.
11. R. Nigam, S. Thomas, Z. Li, and A. Sampson, "A compiler infrastructure for accelerator generators," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2021, p. 804–817.
12. L. Pouchet, "PolyBench/C, the Polyhedral Benchmark suite," 2021, Accessed: Jun. 3, 2021. [Online]. Available: http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/

NUNO PAULINO is a Researcher with INESC-TEC on run-time reconfigurable systems, co-processor hardware acceleration, and tools for hardware design. Paulino received a Ph.D. degree in electrical engineering from the Faculty of Engineering, University of Porto, in 2015. Contact him at nuno.m.paulino@inesctec.pt.

JOÃO BISPO is an Assistant Professor with the Faculty of Engineering, University of Porto. Currently, his work is focused on compilers and distributed computing. Bispo received a Ph.D. degree from IST, Lisbon, in 2012, with a thesis on automatic runtime migration of loops in assembly traces to customized hardware. Contact him at jbispo@fe.up.pt.

JOÃO C. FERREIRA is an Associate Professor with the Faculty of Engineering, University of Porto and a Senior Researcher at INESC-TEC. His research interests include dynamically reconfigurable systems and heterogeneous computing. Contact him at jcf@fe.up.pt.

JOÃO M. P. CARDOSO is a Full Professor with the Faculty of Engineering, University of Porto, and a Senior Researcher with INESC-TEC. His research interests include compilation techniques, domain-specific languages, reconfigurable computing, and application-specific architectures. He is a senior member of IEEE and ACM. Contact him at jmpc@fe.up.pt.