Property-driven Automatic Generation of Reduced-ISA Hardware

Nathan Bleier       John Sartori       Rakesh Kumar

Abstract—
As the diversity of computing workloads and customers continues to increase, so does the need to customize hardware at low cost for different computing needs. This work focuses on automatic customization of a given hardware, available as a soft or firm IP, through eliminating unneeded or undesired instruction set architecture (ISA) instructions. We present a property-based framework for automatically generating reduced-ISA hardware. Our framework directly operates on a given arbitrary RTL or gate-level netlist, uses property checking to identify gates that are guaranteed to not toggle if only a reduced ISA needs to be supported, and automatically eliminates these untoggable gates to generate a new design. We show a 14% gate count reduction when the Ibex [19] core is optimized using our framework for the instructions required by a set of embedded (MiBench) workloads. Reduced-ISA versions generated by our framework that support a limited set of ISA extensions and which cannot be generated using Ibex’s parameterization options provide 10%-47% gate count reduction. For an obfuscated Cortex M0 netlist optimized to support the instructions in the MiBench benchmarks, we observe a 20% area reduction and 18% gate count reduction compared to the baseline core, demonstrating applicability of our framework to obfuscated designs. We demonstrate the scalability of our approach by applying our framework to a 100,000-gate RIDECORE [21] design, showing a 14%-17% gate count reduction.

I. INTRODUCTION

The ever-increasing diversity in the needs of different customers and applications that use the same microprocessor or accelerator design (as a soft or firm IP, for example) has researchers and vendors looking for low-cost approaches to customize hardware designs for different needs [10], [17]. Ability to customize computing hardware at low cost improves computing efficiency for end customers by reducing unnecessary delay, area, and power costs. Low-cost customizability also makes it easier to react to any performance or correctness bugs or security vulnerabilities discovered after a design is finalized.

This work focuses on automatic customization of a given hardware design (RTL or gate-level netlist, obfuscated or open) available as a soft or firm IP, through instruction set architecture (ISA) trimming. We observe (Section VII) that eliminating support for unneeded or undesired instructions from a microprocessor design can generate significant efficiency benefits. An instruction may be unneeded due to the characteristics of target workloads (especially in an embedded setting) or ISA trimming. Similarly, an instruction may be undesired due to high implementation cost, a security vulnerability it may cause, or a bug/error in its implementation. The ability to automatically customize a processor core for a specified ISA subset can also aid generation of multi-ISA heterogeneous multi-core designs [15], where ISAs of the different cores correspond to different subsets of the same composite or base ISA.

Some existing designs support ISA customization in a limited fashion. If RTL is available, support for some instruction set extensions can be removed easily in some modularly-implemented designs, especially for modular ISAs such as RISC-V. For example, the Ibex core RTL [19] uses elaboration time parameters to disable some of the RISC-V ISA extensions it implements. However, a modular ISA does not imply a modular implementation of that ISA. Unless the implementation is truly modular at the extension-level, support for arbitrary individual extensions cannot be easily removed. For example, Ibex does not support core configurations without the c, Zicsr, or Zifencei ISA extensions. Ibex implementation of these extensions contains logic that is tightly coupled with that of other instructions.

Removing support for only a subset of an ISA extension or the base ISA is even harder (even for a modular ISA such as RISC-V) and requires instruction-level modularity in implementation. Consider the case where we want to remove the division instructions, but not the multiply instructions of the RV32m extension [24]. Since RISC-V does not provide this level of modularization, we have no option but to directly modify the RTL to remove those instructions. This also requires global awareness of the design to ensure that requisite changes are made to all impacted components, including the decoder, the execution unit, and the distributed logic of the stall controller. This process is error prone and potentially time-consuming. We are unaware of any RISC-V core design that implements instruction-level modularity.

Furthermore, many popular ISAs are not modular! For example, the openMSP430 open-source implementation of the MSP430 microcontroller provides no option for removing support for instructions, grouped in extension or otherwise, largely because the MSP430 ISA itself is not modular. Similarly, the ARMv6-M architecture of the Cortex M0 and M1 series microprocessors is not modular. It is unclear how to remove support for unneeded or undesired instructions from these IPs without a manual, intrusive, globally-aware and error-prone change to the RTL. Finally, above methods largely do not work if RTL is not available.

In this work, we develop a property-driven framework (Figure 1) for automatically generating hardware for a specified reduced ISA from the base RTL or gate-level netlist (which is how soft and firm IPs are usually available). At high level, the framework allows for specification of a rich set of constraints to the base design, expressed as temporal logic formulas [14]. We attach to every gate (or an RTL module) a property that is checked under the constraint(s). An action – e.g., removal of gates from the base design – is taken if a property is proved to hold. In the specific context explored here, the ISA subset that does not need to be supported (identified through
profiling, testing, or in field) is expressed as a constraint to the base
design’s execution environment. For every gate, property checking [1]
is used to check for gate invariants, such as whether or not a gate’s
output is constant when the specified ISA subset is not supported.
If it is proved that a gate’s output is constant, the net attached
to the gate’s output is detached from the gate and assigned to its
constant value. The design is then re-synthesized, eliminating the
unneeded gate (and potentially eliminating or optimizing many more).
Our approach is largely black box (i.e., requires limited knowledge
or understanding of the microarchitecture implemented by the RTL
or the gate-level netlist), is compatible with any synthesis flow, is
applicable to arbitrary processor and accelerator designs (indeed to an
arbitrary synchronous circuit), and can eliminate arbitrary instructions
in the ISA, including base-ISA instructions. Indeed, we show that our
approach applies even to obfuscated cores (Section VII-B), although
obfuscation may impact the area and gate-count reduction achieved.
To the best of our knowledge, this is first such framework for
automated generation of reduced-ISA hardware.

II. RELATED WORK

A large body of work exists on application-specific instruction
processors (ASIs) and extensible processors. Tensilica’s Xtensa
processors [5], for example, allowed user-specified extensions (using
TiE) to the Xtensa base instruction set using automated customization
tools. ARC [4] allowed designers to add custom instructions using
ARCHitect Processor Configurator. Several MIPS processors allow
application-specific extensions [25]. ARM recently announced Arm
Custom Instructions and associated software development tools [17].
Codasip [10] allows optional or custom hardware extensions to a
RISC-V core supporting the standard ISA.

Our work differs in three important aspects. First, previous works
are focused on allowing new instructions to be added to a design that
implements at least a base or standard ISA. We are focused instead on
automatically removing hardware support for instructions, including
instructions in the base ISA. Second, prior automatic customization
frameworks are tied to a given design. For example, Codasip supplies
its own RISC-V cores as modifiable CodAL models (Codasip’s
processor-modeling language), which can then be customized using
Codasip’s tools (e.g., Codasip Studio). Tools from Tensilica, MIPS,
and ARC were similarly specific to their own processors. Our
framework takes as an input an arbitrary design, including even
gate-level netlists and obfuscated designs, and generates its reduced-
ISA version automatically. Third, prior frameworks are primarily
based on parameterization and metaprogramming. Our approach is
fundamentally different; we identify gates in the original design
that are not needed for the specified reduced-ISA subset and then
eliminate them automatically.

There is some work demonstrating that reducing supported ISA
lead to efficiency benefits [15]. However, these works do not show
how to generate reduced-ISA hardware. We present an approach
that can automatically generate reduced-ISA hardware starting with
a given arbitrary base design.

Recent work automatically generates a bespoke design customized
for a given application program binary [2]. The resulting design is
not guaranteed to execute correctly any other program binary. Our
focus is automatically generating a reduced ISA design. The resulting
design can support arbitrary applications that use the reduced ISA.

Finally, property checking has had a rich history in hardware
verification [3]. There has also been some work on synthesizing
property checks directly into hardware [8]. We use property checking
directly in the hardware synthesis flow to perform automated hard-
ware transformations, specifically focused on automatic generation
of reduced-ISA hardware. To the best of our knowledge, this is the
first use of property checking in automating hardware optimization.

III. MOTIVATION

Consider an embedded setting in which a core targets a fixed
set of workloads. Table I shows the number of instructions that
are supported by the Ibex RISC-V core, as well as the number of
RISC-V instructions used across several embedded (MiBench)
group (i.e., networking, security, automotive) uses only a fraction
of the instructions supported by the Ibex core. In fact, only 68%
of the base ISA is used to support all the groups. This suggests
that there may be significant opportunity to customize the Ibex core
for a reduced ISA if the goal is to target only a small number
of applications in an embedded setting. Table I shows that the
opportunity may be even greater for the Cortex-M0 core, since only
60% of the ARMv6-M base ISA is used to support all the groups;
higher opportunity stems from a richer base ISA (ARMv6-M), with
83 instructions (vs 78 instructions supported by Ibex).

Similar opportunity exists when an IP is used in a (likely embed-
ded) setting where a subset of supported extensions is not needed.
Table I shows that the number of instructions supported by Ibex
implementing different RISC-V ISA extensions can vary by almost
2×. The variation can reach 4× for IPs that implement more
extensions than Ibex (e.g., Ibex does not implement floating point
or atomic extensions). The ability to easily transform an IP for a
reduced-ISA variant could lead to significant benefits.

Ability to automatically generate reduced-ISA hardware may be
useful also to eliminate support for deprecated or rarely-used in-
structions. A study of x86 applications showed that more than 500
instructions were never used [18], and thus contribute unnecessary
overhead. A reduced-ISA hardware can eliminate this overhead by
removing support for rarely-used instructions.

Motivation also exists in terms of trustworthy execution. Instruc-
tions are often diagnosed (post-design or in-field) as having buggy
implementation – one need only look at errata sheets for processors
– or as causing security vulnerabilities. Notorious examples include
correctness or security vulnerabilities due to FDIV [22], TSX in-
structions [12], RDRAND and RDSEED [11], SWAPS [20], etc.
Eliminating support for these instructions from an existing IP may
fix the bug or vulnerability and increase efficiency at the same
time, without requiring intrusive hardware changes. In some instances,
this may be a feasible, interim solution before a significant microarchi-
tecture re-design can be done. The approach is particularly attractive
when a microcode ROM – which can be used to eliminate support for
instructions by changing the microcode – is not available (embedded
microcontrollers are often not microcoded) or when a change in the
microcode cannot fix the problem [9]. In some embedded settings,
a reduced-ISA IP may also be desirable to preventively eliminate
instructions that may cause security vulnerabilities (e.g., indirect
jumps – exploits due to indirect jumps are well known [23]) or whose
implementation may not have been fully verified [22].

Finally, some instructions may be diagnosed in the field as being
expensive (e.g., several AVX instructions were discovered to routinely
cause voltage emergencies leading to large performance degrada-
tion [7]). Such instructions can be automatically eliminated from the
IP through automatic generation of reduced-ISA hardware.

IV. PROPOSED FRAMEWORK

Inputs to the proposed Property-Driven Automatic Transformation
(PDAT) framework, described in Figure 2, are (a) a gate-level netlist
for the sequential digital circuit design, synthesized to either
A physical standard cell library or a logical standard cell library (e.g., GTech), which is annotated with elements of (b) a Property Library which contains properties that capture invariants about gates, and (c) a collection of restrictions to the execution environment which ensure that only desired instructions are by the property checker.

1) Property Library: The Property Library we use in our analysis is written in SystemVerilog, and its properties are expressed as SystemVerilog Assertions (SVA). An example property module from this library is depicted in Listing 1. Property modules are bound to each instance of the associated cell-type in the netlist (e.g., the module and2_properties is bound to each two-input AND gate in the netlist). The properties check for semantically-meaningful invariants on the gate inputs and outputs in what we term gate-level property checking. For example, the property and_in_A2_A1 checks that if the cell’s A1 input is T, then so is the cell’s A2 input (at all times and in every possible execution). If this property is satisfied, then it means that the associated cell can be rewired by assigning its output to the net driving A1, without impacting the functional behavior of the design. An advantage of property checking at the gate level, as opposed to a higher level of abstraction, is that the Property Libraries can be used to enable optimizations for any design synthesized to a standard cell library, including designs for which the microarchitecture is not known/understood (e.g., obfuscated designs).

2) Annotated Netlist: We annotate the IP’s netlist with properties from the Property Library. Each gate in the netlist has bound to it an instance of the appropriate property module (e.g., a copy of and2_properties is bound to each AND2 gate in the netlist). Thus for each gate in the annotated netlist, there are one or more asserted properties.

3) Environment Restrictions: We use ‘environment restrictions’ to constrain the property checking effort to consider only programs from the targeted ISA subset and only programs from the targeted ISA subset. Environment restrictions are expressed as SVA properties. Environment restrictions also manage memory reads and writes and constrain the netlist’s primary inputs.

Listing 1

```
module and2_properties(input A1, A2, ZN);
default clocking @0; //global_clock; //endclocking
default disable clk (1'b0);
alinu2z2n2: assert property (ZN == 1'b2);
aalu2z2n1: assert property (ZN == 1'b1);
aand2in2a1: assert property (A1 -> A2);
aand2in1a1: assert property (A2 -> A1);
endmodule
```

Figure 3 depicts the versatility of this approach. We can encode ISA restrictions (e.g., removal of instructions, removal of ISA extensions), restrictions on I/O protocols (e.g., bounded or deterministic memory latencies), explicit mapping of specific code sequences to address regions (e.g., reset handlers, trap vectors, entire programs, or operating system code), etc.

A. Property Checking Stage

This is the first and generally most time-consuming stage of the PDAT pipeline. The property checker takes the annotated netlist, property library, and environment restrictions as inputs, and checks to see if the properties hold or are violated by allowed executions. Property checking produces a list of properties that are proved to hold on all allowed executions. In our work, we use Mentor’s Questa Formal software as the property checker.

B. Netlist Rewiring Stage

In this stage of the PDAT pipeline, the original netlist is rewired based on the list of proved properties created in the Property Checking Stage. Note that by limiting this stage to rewiring, we do not remove, transform, or add any cells in the netlist. This stage simply modifies cell port listings and adds assignment statements to the netlist. The rewired netlist is passed to the next stage of the PDAT pipeline for further optimization. If no invariants about a cell were proved during the property checker pipeline stage, then that cell is not changed during this stage.

C. Logic Resynthesis Stage

The rewired netlist is resynthesized using a standard synthesis flow. We rely on logic synthesis to remove and simplify constrained cells, since logic synthesis tools are ostensibly very good at this. This stage produces a transformed netlist, which is optimized with respect to the execution environment.

V. GENERATING A REDUCED-ISA DESIGN USING PDAT

We present an illustrative example of PDAT’s capabilities by using it to generate a reduced-ISA design from a core (such as RIDECORE) implementing the RISC-V RV32I ISA [24], which consists of four-byte instructions. First, we encode ISA instructions as properties, as shown in Listing 2 – lines 2 to 11. For example, beginning on line 2, we define a property which ensures that a 32-bit instruction is formatted as a load-upper immediate (LUI) instruction. The LUI instruction has three fields: a 7-bit opcode in the least significant bits, a 5-bit destination register, and a 20-bit immediate value. Since these last two fields may take any arbitrary value, we leave them unspecified. We then use these properties to restrict the execution environment of the core. We place these restrictions directly onto the instruction port.

Listing 2

```
1 package rv32i_sva;
2 property \lns (logic [31:0] instr);
3 instr[31] == op_lui;
4 endproperty
5 property \unmp (logic [31:0] instr);
6 instr[31] == op_unmp;
7 endproperty
8 property \call (logic [31:0] instr);
9 instr[31] == 2 d beq
10 endproperty
11 property \break (logic [31:0] instr);
12 instr[31] == 32 d bne
13 endproperty
14 property rv32i_all (logic [31:0] instr);
15 \lns (instr) or 
16 \unmp (instr) or 
17 \call (instr); 
18 endproperty
19 property no_exit (logic [31:0] instr);
20 \lns (instr) or 
21 \unmp (instr) or 
22 \call (instr) or 
23 \break (instr);
24 endproperty
25 endpackage
```

Listing 3 shows the PDAT input for the RV32I core instantiated as a top-level SystemVerilog module. The module’s ports, omitted for brevity, mimic those of the RISC-V core. In addition to instantiating...
the core, in lines 14 and 15, we bind gate-level properties to every gate in the netlist, and in lines 7 to 12 we assume a property that forces instr to always be an instruction in the desired ISA subset. Once the top-level module is built, it is fed to the PDAT pipeline (as shown in Figure 2).

The approach described above – using port-based constraints – is relatively straightforward for ISAs with fixed-width instructions, such as RV32i. However, many systems do not have fixed-width instructions. As such, in addition to port-based constraints, where constraints are placed on a core’s instruction memory port, we support cutpoint-based constraints, where constraints are placed on a core’s internal nets.

We define a cutpoint as a net whose value is determined by the property checking tool rather than its netlist drivers (so called because we are, in effect, cutting the net from its true driver). Cutpoints allow us to resolve a class of issues which arise as a result of variable-length instruction encoding and instruction caches. For example, a branch instruction may jump to an address which results in an instruction cache miss. Because this is relatively straightforward for ISAs with fixed-width instructions, such as RV32i, however, many systems do not have fixed-width instructions.

The approach described above – using port-based constraints – is relatively straightforward for ISAs with fixed-width instructions, such as RV32i. However, many systems do not have fixed-width instructions. As such, in addition to port-based constraints, where constraints are placed on a core’s instruction memory port, we support cutpoint-based constraints, where constraints are placed on a core’s internal nets.

We define a cutpoint as a net whose value is determined by the property checking tool rather than its netlist drivers (so called because we are, in effect, cutting the net from its true driver). Cutpoints allow us to resolve a class of issues which arise as a result of variable-length instruction encoding and instruction caches. For example, a branch instruction may jump to an address which results in an instruction cache miss, however, with variable length instructions, there is no guarantee that the branch target address is an instruction boundary and not the middle of an instruction, which in turn, means the core may fetch an ‘instruction’ which is not part of the targeted ISA subset. If the netlist is not obfuscated (i.e., we have visibility into the netlist), we can insert a cutpoint somewhere in the design and then constrain its value in order to ensure that only instructions from the targeted ISA subset are decoded. Figure 4 shows how a cutpoint is used to place a valid, aligned instruction into the fetch-decode pipeline register, rather than the potentially unaligned instruction coming out of the instruction cache.

The approach described above – using port-based constraints – is relatively straightforward for ISAs with fixed-width instructions, such as RV32i. However, many systems do not have fixed-width instructions. As such, in addition to port-based constraints, where constraints are placed on a core’s instruction memory port, we support cutpoint-based constraints, where constraints are placed on a core’s internal nets.

We define a cutpoint as a net whose value is determined by the property checking tool rather than its netlist drivers (so called because we are, in effect, cutting the net from its true driver). Cutpoints allow us to resolve a class of issues which arise as a result of variable-length instruction encoding and instruction caches. For example, a branch instruction may jump to an address which results in an instruction cache miss, however, with variable length instructions, there is no guarantee that the branch target address is an instruction boundary and not the middle of an instruction, which in turn, means the core may fetch an ‘instruction’ which is not part of the targeted ISA subset. If the netlist is not obfuscated (i.e., we have visibility into the netlist), we can insert a cutpoint somewhere in the design and then constrain its value in order to ensure that only instructions from the targeted ISA subset are decoded. Figure 4 shows how a cutpoint is used to place a valid, aligned instruction into the fetch-decode pipeline register, rather than the potentially unaligned instruction coming out of the instruction cache.

The approach described above – using port-based constraints – is relatively straightforward for ISAs with fixed-width instructions, such as RV32i. However, many systems do not have fixed-width instructions. As such, in addition to port-based constraints, where constraints are placed on a core’s instruction memory port, we support cutpoint-based constraints, where constraints are placed on a core’s internal nets.

We define a cutpoint as a net whose value is determined by the property checking tool rather than its netlist drivers (so called because we are, in effect, cutting the net from its true driver). Cutpoints allow us to resolve a class of issues which arise as a result of variable-length instruction encoding and instruction caches. For example, a branch instruction may jump to an address which results in an instruction cache miss, however, with variable length instructions, there is no guarantee that the branch target address is an instruction boundary and not the middle of an instruction, which in turn, means the core may fetch an ‘instruction’ which is not part of the targeted ISA subset. If the netlist is not obfuscated (i.e., we have visibility into the netlist), we can insert a cutpoint somewhere in the design and then constrain its value in order to ensure that only instructions from the targeted ISA subset are decoded. Figure 4 shows how a cutpoint is used to place a valid, aligned instruction into the fetch-decode pipeline register, rather than the potentially unaligned instruction coming out of the instruction cache.

The approach described above – using port-based constraints – is relatively straightforward for ISAs with fixed-width instructions, such as RV32i. However, many systems do not have fixed-width instructions. As such, in addition to port-based constraints, where constraints are placed on a core’s instruction memory port, we support cutpoint-based constraints, where constraints are placed on a core’s internal nets.

We define a cutpoint as a net whose value is determined by the property checking tool rather than its netlist drivers (so called because we are, in effect, cutting the net from its true driver). Cutpoints allow us to resolve a class of issues which arise as a result of variable-length instruction encoding and instruction caches. For example, a branch instruction may jump to an address which results in an instruction cache miss, however, with variable length instructions, there is no guarantee that the branch target address is an instruction boundary and not the middle of an instruction, which in turn, means the core may fetch an ‘instruction’ which is not part of the targeted ISA subset. If the netlist is not obfuscated (i.e., we have visibility into the netlist), we can insert a cutpoint somewhere in the design and then constrain its value in order to ensure that only instructions from the targeted ISA subset aredecoded. Figure 4 shows how a cutpoint is used to place a valid, aligned instruction into the fetch-decode pipeline register, rather than the potentially unaligned instruction coming out of the instruction cache.

The approach described above – using port-based constraints – is relatively straightforward for ISAs with fixed-width instructions, such as RV32i. However, many systems do not have fixed-width instructions. As such, in addition to port-based constraints, where constraints are placed on a core’s instruction memory port, we support cutpoint-based constraints, where constraints are placed on a core’s internal nets.

We define a cutpoint as a net whose value is determined by the property checking tool rather than its netlist drivers (so called because we are, in effect, cutting the net from its true driver). Cutpoints allow us to resolve a class of issues which arise as a result of variable-length instruction encoding and instruction caches. For example, a branch instruction may jump to an address which results in an instruction cache miss, however, with variable length instructions, there is no guarantee that the branch target address is an instruction boundary and not the middle of an instruction, which in turn, means the core may fetch an ‘instruction’ which is not part of the targeted ISA subset. If the netlist is not obfuscated (i.e., we have visibility into the netlist), we can insert a cutpoint somewhere in the design and then constrain its value in order to ensure that only instructions from the targeted ISA subset are decoded. Figure 4 shows how a cutpoint is used to place a valid, aligned instruction into the fetch-decode pipeline register, rather than the potentially unaligned instruction coming out of the instruction cache.

TABLE II
ARCHITECTURE AND MICROARCHITECTURE FEATURES OF IBEX, RIDECORE, AND CORTEX M0.

<table>
<thead>
<tr>
<th>Core</th>
<th>ISA</th>
<th>Stages</th>
<th>IW</th>
<th>ROB Size</th>
<th>BP</th>
<th>RT Entries</th>
<th>Physical Registers</th>
<th>Gate Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ibex</td>
<td>RV32imcz</td>
<td>2 (1)</td>
<td>1</td>
<td>N/A</td>
<td>N/S</td>
<td>N/A</td>
<td>32</td>
<td>10k</td>
</tr>
<tr>
<td>Cortex M0</td>
<td>ARMv6-m</td>
<td>3</td>
<td>1</td>
<td>N/A</td>
<td>N/S</td>
<td>N/A</td>
<td>16</td>
<td>10k</td>
</tr>
</tbody>
</table>

VI. EXPERIMENTAL METHODOLOGY

While the PDAT framework is general and can be applied even to CISC ISAs in non-embedded settings, the primary use case we explore in this work is embedded computing. We used three embedded-class cores for our evaluations (Table II). The first core, Ibex [19] (formerly zero-riscy), is a scalar, in-order, 32-bit RISC-V core that implements the c, m, Zicsr, and Zifencei extensions; we refer to the last two extensions collectively as the ‘z-extension’. We used the two-stage pipeline version of Ibex, IRQ and NMI interrupt lines were disabled for our analysis so that our results are conservative (since we do not count gates removed in the debug, watchdog, and interrupt logic from the baseline design). To avoid issues with misalignment and indirect branches, cutpoint-based constraints were used to generate reduced-ISA Ibex variants (see Section V). The second core, RIDECORE [21], is a two-way, out-of-order 32-bit RISC-V core that implements (most of) the RV32i base ISA, as well as the multiply instructions from the m-extension (though it does not implement hardware division or remainder instructions). Since RIDECORE has word-aligned instructions and does not allow branching to non-word-aligned addresses, we use port-based constraints to generate reduced-ISA designs. The third core, ARM’s Cortex M0 [16], is a three-stage core implementing the ARMv6-M ISA. The core has full support for ISR and exception handling. We analyze an obfuscated version of this core and place constraints directly on the ports to generate reduced-ISA designs (since, due to obfuscation, we cannot place constraints on pipeline registers as was done for Ibex).

We synthesized RTL and netlists into gate-level netlists using Synopsys Design Compiler. Compilation was done with the –ungroup_all option to minimize area at a fixed frequency for each core type using the 45 nm NANGATE standard cell library. Property checking was performed using Mentor’s Questa Formal software, version 2019.4_1.
interesting due to safety, reliability, or security reasons. Nevertheless, such ISA variants may still be
allowed over 6% area and 7% gate count savings over the baseline
example, that restricting Ibex to only word-aligned memory accesses
counts advantage over the RV32i PDAT variant baseline. We see, for
most graph in Figure 5), we do not see a significant area or gate
reductions, with the exception of c-extension removal.

We also see that PDAT-based removal of ISA extensions (again,
consider interesting variants that cannot be generated using
Ibex’s elaboration time parameters) results in substantial area and
gate count reductions, with the exception of c-extension removal.
The RISC-V c-extension includes 16-bit versions of RV32i’s 32-bit
instructions. As these instructions are largely different encodings of
instructions removed, based on their relative lack of importance for
opportunities from ISA subsetting.

VII. RESULTS
A. Automatic Generation of Reduced-ISA Microprocessors
Figure 5 presents results for some reduced-ISA variants of Ibex not supported by elaboration time parameters. An immediately interesting result is the area difference between the ‘Ibex Full’ (design before application of PDAT) and ‘Ibex ISA’ (PDAT run without ISA subsetting) core variants. By restricting the execution environment to the full set of instructions officially supported by the core, we see nearly 10% area savings. This seemingly counterintuitive result (since we have not even reduced the ISA yet!) is due to the inability of standard logic synthesis tools to understand which states are unreachable when only valid ISA instructions are provided as input. PDAT identifies such states, since it explores the state space of the design for a given environmental constraint. The logic corresponding to such states is marked as unneeded by PDAT and subsequently eliminated when the environment is constrained to only valid ISA instructions.

We also see that PDAT-based removal of ISA extensions (again, consider interesting variants that cannot be generated using Ibex’s elaboration time parameters) results in substantial area and gate count reductions, with the exception of c-extension removal. The RISC-V c-extension includes 16-bit versions of RV32i’s 32-bit instructions. As these instructions are largely different encodings of instructions removed, based on their relative lack of importance for opportunities from ISA subsetting.

When considering ISA subsets customized for the MiBench benchmark groups discussed in Table I (assuming an embedded setting), we see that the MiBench Networking and MiBench Security subset cores are over 3% and 11% smaller, with 5% and 12% fewer gates, respectively, than the PDAT baseline RV32imc ISA. These results are even more significant when compared against the PDAT Ibex ISA (RV32imcz) variant. In this case, the MiBench All ISA variant generated by PDAT is 15% smaller and has 18% fewer gates than the PDAT-generated Ibex ISA core variant (and 23% smaller with 14% fewer gates than Ibex without PDAT).

For core variants that support ISAs with special properties (rightmost graph in Figure 5), we do not see a significant area or gate count advantage over the RV32i PDAT variant baseline. We see, for example, that restricting Ibex to only word-aligned memory accesses allows over 6% area and 7% gate count savings over the baseline RV32i PDAT variant. Nevertheless, such ISA variants may still be interesting due to safety, reliability, or security reasons.

B. Reducing Obfuscated Designs
As discussed in Section IV, PDAT can be used to analyze obfuscated cores. Figure 6 shows PDAT results for an obfuscated version of ARM’s Cortex M0 microcontroller. Recall that ARMv6-M, as well as its Cortex M0 implementation, are not modular. So, the studied microcontroller variants cannot be generated automatically without PDAT.

We once again see substantial area and gate count reduction (20%, and 18%, respectively) simply by performing PDAT analysis with the core’s full ISA. Some of the unneeded core area may be attributable to ARM’s obfuscation techniques. Somewhat surprisingly, the ‘MiBench All’ ISA, consisting of all instructions needed to implement the MiBench benchmarks (Table I), has the same area and gate count as the ‘ARMv6-M’ variant. We hypothesize (but are unable to verify due to obfuscation) that this is due to the fact that the MiBench subset includes two and four-byte instructions, as well as indirect branches. As a result, the best way to constrain Cortex M0 for such a subset is with cutpoints (Section IV). However, as the Cortex M0 netlist is obfuscated, we are forced to use port-based constraints, which limits the opportunities from ISA subsetting.

The ‘interesting subset’ is the base ARMv6-M ISA with select instructions removed, based on their relative lack of importance for a scalar, in-order uniprocessor (e.g., memory ordering instructions, inter-core signaling instructions), as well as the multiply instruction, and all seven of the four-byte instructions. As all instructions in this ISA subset are two-byte aligned (the minimum instruction length in the ARMv6-M ISA), this ensures that all branches (direct or indirect)
point to valid instructions from the subset. This ‘interesting subset’ is a practical instruction subset for many embedded applications. The Cortex M0 variant that supports this ISA subset has 23% and 20% lower area and gate count, respectively.

C. Scalability

Unlike in hardware verification, state space explosion is not a crippling issue for PDAT since any inconclusive analysis in PDAT’s Property Checking Stage stage simply means that the resulting transformed netlist may be less optimized than if the property’s invariant was proved to hold.

Figure 7 shows the results of employing PDAT for RIDECORE, which is an order of magnitude larger than Ibex and Cortex M0. None of the studied variants can be generated using elaboration time parameters. Results for RIDECORE are muted compared to Ibex. This is not surprising since, unlike an inorder core such as Ibex, RIDECORE has several large OO-supporting structures such a physical register file that are largely unaffected when support for an ISA subset is removed. We still see an area improvement of 6% by simply running PDAT with the environment restricted to the full RIDECORE ISA. Other RIDECORE variants show small improvements over the RIDECORE ISA variant in terms of percent area or gate reduction. However, in absolute terms, some of these improvements are in the same range as the improvements for Ibex. For example, Ibex RV32i and RV32e variants have a difference of 934 gates, while the RIDECORE RV32i and RV32e variants have a difference of 1920 gates, over $2 \times$ the difference in Ibex.

VI. SUMMARY AND CONCLUSION

As diversity of customers and workloads increases, the need to customize hardware at low cost for different computing needs continues to increase. This work focuses on automatic customization of a given hardware, available as a soft or firm IP, through eliminating unneeded or undesired ISA instructions and instruction sequences. We presented a property-based framework for automatically generating reduced-ISA hardware. Our framework directly operates on a given arbitrary RTL or gate-level netlist, uses property checking to identify gates that are guaranteed to not toggle if only a reduced ISA needs to be supported, and automatically eliminates these unexercisable gates to generate a new design. We showed a 14% gate count reduction when the Ibex core is optimized using our framework for the instructions required by a set of embedded (MiBench) workloads. Reduced-ISA versions generated by our framework that support a limited set of ISA extensions and which cannot be generated using Ibex’s parameterization options provided 10%-47% gate count reduction. We also demonstrate that our framework is applicable to obfuscated designs. For an obfuscated Cortex M0 netlist, we observe a 20% area reduction and 18% gate count reduction for the MiBench benchmarks over the baseline core. When applying our framework to a 100,000-gate RIDECORE design, we saw 14%-17% gate count reduction, demonstrating scalability.

REFERENCES