Hardware Verification using Software Analyzers

Rajdeep Mukherjee
University of Oxford

Daniel Kroening
University of Oxford

Tom Melham
University of Oxford

***Abstract*—Program analysis is a highly active area of research, and the capacity and precision of software analyzers is improving rapidly. We investigate the use of modern *software* verification tools for formal property checking of *hardware* given in Verilog at register-transfer level. To this end, we translate RTL Verilog into an equivalent word-level ANSI-C program, according to synthesis semantics. The property of interest is instrumented into the C program as an assertion. We subsequently apply three different software verification techniques—bounded model checking, path-based symbolic simulation and abstract interpretation—and compare their performance to conventional methods for property verification of hardware designs at netlist and register-transfer level. Our experimental results indicate that speedups of more than an order of magnitude are possible. To the best of our knowledge, this is the first attempt to perform property verification of hardware IPs given at register-transfer level using software verifiers.**

I. Introduction

Early tools for formal property checking of hardware converted the design into a netlist, typically represented using And-Inverter graphs (AIGs). This approach misses the opportunity to exploit the word-level structure of designs given at the register transfer level (RTL). Formal hardware verification tools have therefore now switched to a word-level representation of the transition relation. This change in the representation has enabled the use of modern solvers for Satisfiability Modulo Theories (SMT) [1] in the back-end of the tools [2]–[5].

Consider Bounded Model Checking (BMC) as an exemplar of the way contemporary formal verifiers for hardware work (Figure 1). The input Verilog design is first translated into a transition relation at register-transfer level, in which one transition corresponds to one clock cycle. The transition relation for a system and its specification are jointly unwound up to a user-defined depth to form a word-level formula. This formula is then given to a suitable SMT solver. If the formula is determined to be satisfiable, then there is a bug and the verifier extracts a trace of the circuit leading to the bug from the satisfying assignment.

The performance of word-level symbolic execution engines is determined by the level of abstraction of the symbolic expressions and the power of the rewrite engine used by the SMT solvers. Tools implementing this approach scale up to block level or small IP level [3], [4], for example, a FIFO controller or transceiver/receiver of a USB IP. They generally do not scale to large IPs, subsystems, or full SoCs.

In this paper, we argue that it is time for another, orthogonal, change in how designs are represented—this time to enable and leverage leading-edge research in modern *software* analysis techniques in the *front-end* of our tools, as a complement better formal reasoning in the back-end. Specifically, we propose to translate circuit designs given initially in RTL into word-level ANSI-C programs that are equivalent according to synthesis semantics—and then undertake our formal analysis on this alternative software representation. The aim is a step-change in performance and scalability in the verification, since the new representation enables the direct application of a range of modern analysis techniques software, leveraging continued major advances in this technology and the sustained efforts of a large research community. In particular, effective analysis techniques such as abstract interpretation, software Bounded Model Checking, and symbolic execution [6]–[8] can be exploited.

Figure 2 gives an overview of the tool flow we propose. Note that the hardware design may be augmented with software, such as firmware or high-level models of surrounding IPs given in C. We first translate the Verilog design to ANSI-C and thus obtain a common representation in the early phase of the verification process. There is then a broad choice of software verification technologies that can be applied. In this paper, we evaluate three of these: software BMC, path-based forward symbolic simulation, and abstract interpretation. We observe substantial speed-ups compared to traditional approaches based on transition relations extracted from RTL.

The natural software language for our representation is C, so our approach bears some resemblance to methods for verifying system-level or transaction-level descriptions of hardware written in C/C++ or SystemC. Of course the idea of raising

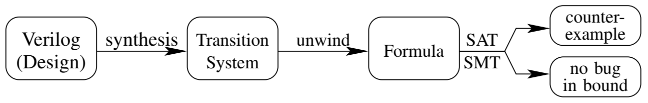

Fig. 1. Conventional flow for hardware property verification using BMC

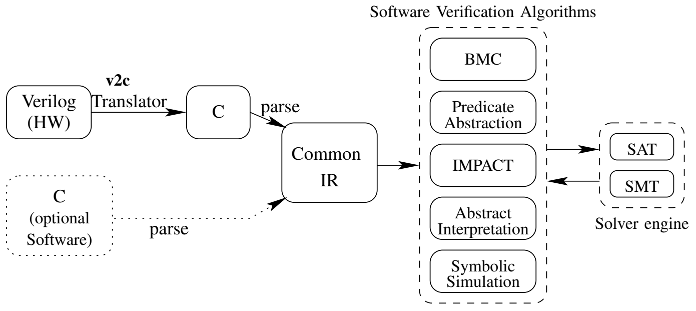

Fig. 2. Proposed new tool flow

Supported by EPSRC EP/J012564/1, ERC project 280053 and the Semiconductor Research Corporation (SRC) under task 2269.001.

the level of abstraction above RTL by expressing designs in software has been widely advocated already—we will mention only [3], which highlights verification-related benefits in the context of SoC design. But we emphasise that these abstract models are usually written *manually* and are expensive to maintain—and often disconnected from a 'golden' RTL design model from which the chip is ultimately realised. Our method, by contrast, aims at a fully automatic verification technique for existing Verilog RTL, expressed at the register-transfer level of abstraction.

Contribution: This paper makes two main contributions:

1) We present a method for constructing a tool for checking properties of Verilog RTL. The proposed tool flow is significantly different from existing verification techniques, in that we translate RTL designs into software and build an analysis engine capable of reasoning with this representation. As we start with the same input files as conventional analysers, tools built this way can be integrated into current industrial formal verification flows. We do not require manually-written high-level models.
2) We evaluate three major exemplars of software analysis techniques within the proposed tool flow and offer a performance comparison to conventional RTL-based tools. To enable a full comparison, we implement the bounded model checking algorithm at different levels of design abstraction and in different representations—including netlists/AIGs, word-level RTL, and software given as C program. Our tool also permits a performance comparison between various SAT/SMT back-ends.

II. Background

This section provides background on BMC and symbolic simulation for both hardware and software. We also briefly summarise options for back-end decision procedures.

A. Symbolic Hardware Simulation and Software Execution

Symbolic simulation of hardware designs aims to replace multiple simulation runs, each with different inputs, by a single symbolic simulation run that uses symbolic values as its inputs. The outcome is a set of symbolic expressions for the outputs of the design. And each assignment of values to the variables in these expressions produces a set of values observable on the outputs. The symbolic expressions are then passed to an appropriate symbolic reasoning engine together with the desired properties. Symbolic simulation can be performed on hardware at different abstraction levels—but also on software, where it is known as *symbolic execution*. We will discuss specific instances of symbolic simulation next.

B. Bounded Model Checking for Hardware

Bounded Model Checking was first introduced as a hardware verification technique [9]. The hardware model is typically at the register transfer level. The idea of BMC can be sketched as follows. Given a depth k and a set of error states F, BMC operates by unwinding the transition relation T up to depth k starting from initial state x_0, represented by an initial state predicate I. This results in the following formula:

$$I(x_0) \wedge T(x_0,x_1) \wedge \ldots \wedge T(x_{k-1},x_k) \wedge (F(x_1) \vee \ldots \vee F(x_k))$$

Verilog	Word-level Simulation
module top(reset,a,b,x,y);	
output reg[3:0] x,y;	
input [3:0] a,b;	
input reset;	
always @(a **or** b) **begin**	
if(reset) **begin**	
x = 3'b0;	x'=$ite(reset,0,$
y = 3'b0;	$ite((a>b),a+b,x))$
end	
else begin	y'=$ite(reset,0,$
if(a > b)	$ite((a>b),y,(a\&3) \ll b))$
x = a+b;	
else	
y = (a & 3) << b;	
end	
end	
endmodule	

Fig. 3. Word-level symbolic simulation of Verilog

The formula is then checked for satisfiability using an efficient SAT or SMT procedure. If the formula is satisfiable, then it is possible for one or more of the error states to be reached. An error trace can be then extracted from the output of the SAT procedure. If the formula is not satisfiable, the system and its specification are further unwound. This process terminates when the length of the potential error trace exceeds a certain completeness threshold (it is sufficiently long to ensure that no trace exists) or when the SAT/SMT procedure exceeds its capacity limit.

The implementation details of BMC tools strongly depend on how the transition relation T, which constitutes a formal *model* of the design, is represented. The challenges related to the generation of formal models from designs given in hardware description languages are mostly shared among all analysis techniques for hardware. Most HDLs have, for example, both *simulation semantics* and *synthesis semantics*. Simulation semantics are typically based on an event queue. On the other hand, the synthesis semantics is closer to the actual hardware produced, and may uncover design flaws that go unnoticed during simulation. In addition, the semantics of the symbolic expressions generated by the tool differs, depending on the level of abstraction at which the analysis is carried out.

BMC on Netlists: One way to represent the transition relation is to use a netlist, which can be obtained from Verilog using standard behavioral synthesis. The netlist captures the effect of one clock period on the state-holding elements. The netlist consists of a network of and-gates, inverters, and memory elements referred to as registers. A typical way to represent a netlist is to use an *And-Inverter Graph* (AIG).

Word-level BMC: Word-level reasoning engines have motivated the use of word-level representations for the transition relation [3], [4]. The use of the term "word" refers to a *bit-vector* encoding of the registers wires, rather than representing them as individual bits. Consider the Verilog example in the left-hand column of Figure 3. The circuit has two state-holding registers, each of four bits. Thus, two next-state functions are generated, denoted by x' and y'. The branching in the input Verilog program yields expressions with the *ite* operator. As in the case of the netlist-based transition relation, the word-level transition relation encodes the effect of one clock period on the state-holding elements.

Program	Path Constraint 1	Path Constraint 2	Path Constraint 3

`void top(){`
`  if(reset) {`
`    x=0;`
`    y=0;`
`  }`
`  else {`
`    if(a > b)`
`      x=a+b;`
`    else`
`      y=(a & 3)<<b;`
`  }`
`}`
	$C_1 \equiv$
	$reset_1 \neq 0 \wedge$
	$x_2 = 0 \wedge$
	$y_2 = 0$
		$C_2 \equiv$
		$reset_1 = 0 \wedge$
		$b_1 \ngeq a_1 \wedge$
		$x_3 = a_1 + b_1$
			$C_3 \equiv$
			$reset_1 = 0 \wedge$
			$b_1 \geq a_1 \wedge$
			$y_3 = (a_1\&3) << b_1$

Fig. 4. Single-path forward symbolic simulation

Program	Program Constraint

`void top(){`
`  if(reset){`
`    x=0;`
`    y=0;`
`  }`
`  else {`
`    if(a > b)`
`      x=a+b;`
`    else`
`      y=(a & 3)<<b;`
`  }`
`}`
	$C \iff ((guard_1 = \neg(reset_1 = 0))\wedge$
	$(x_2 = 0) \wedge (y_2 = 0)\wedge$
	$(x_3 = x_1) \wedge (y_3 = y_1)\wedge$
	$(guard_2 = \neg(b_1 >= a_1))\wedge$
	$(x_4 = a_1 + b_1) \wedge (x_5 = x_3)\wedge$
	$(y_4 = (a_1\&3) << b_1)\wedge$
	$(x_6 = ite(guard_2, x_4, x_5))\wedge$
	$(y_5 = ite(guard_2, y_3, y_4))\wedge$
	$(x_7 = ite(guard_1, 0, x_6))\wedge$
	$(y_6 = ite(guard_1, 0, y_5)))$

Fig. 5. Translation of C program into bit-vector constraint

C. Path-based Symbolic Execution for Software

In the case of software, symbolic execution is a combination of traditional program testing together with symbolic methods [4], [10], [11]. Recent advances in SMT solvers and constraint solving have greatly promoted the use of symbolic execution for formal property verification and test generation in large software systems. Instead of unwinding the entire transition relation, path-based software analyzers perform forward symbolic execution along individual program paths up to a given depth. The resulting formula is then passed to the SAT/SMT solver. This basic approach has a range of applications. For example, it can be used to check arbitrary safety properties or to generate test vectors to achieve particular coverage goals.

We will use the software equivalent to the Verilog code in Figure 3 to illustrate symbolic execution on software. In their most basic form, symbolic simulators such as KLEE [11] use path-based encodings to explore precisely one path at a time. Figure 4 gives the three different path constraints corresponding to three paths in the program on the left. Note that all paths in this program are feasible. An advantage of this approach is that the generated formulas are often simple and can be solved effectively. But path-based exploration suffers from the *path explosion problem*, as the number of paths is in general exponential. A principal method to address this is *path merging*. The idea is to selectively merge the formulas that correspond to two (or more) paths at points of reconverging control-flow. As a result, the number of formulas is reduced.

D. BMC for Software

On the extreme end, the CBMC Bounded Model Checker *always* merges—generating only a *single* formula for a given unwinding bound k [12]. This formula is linear in the size of the program and linear in k even if there is an exponential number of paths in the program. The right-hand side of Figure 5 gives the outcome of the translation of our running example. The symbolic values of variables are computed as expressions over the initial values of variables x and y. The branching in the program yields expressions with the *ite* operator. The input program is translated into a bit-vector equation C that forms the set of constraints as given in Figure 5.

E. Bit-level Versus Word-level Proof Engines

The result of symbolic simulation or execution is a formula, which is given to a decision procedure. We briefly summarise the options for the decision procedure.

Bit-level Solvers: Current state-of-the-art propositional SAT solvers use a host of advanced techniques, such as Boolean constraint propagation, conflict-driven clause learning, non-chronological backtracking, pre-processing steps, and rapid restarts. These mechanisms have enable SAT solvers to reason very effectively about bit-level encodings of circuit designs. Our BMC engine uses MiniSat 2.2.0 as the default SAT solver. It is worth emphasising that bit-level reasoning engines can benefit from word-level input, as custom clause-level encodings into CNF can be used.

Word-level Solver Engines: Solver engines that perform word-level reasoning are now commonly referred to as solvers for *Satisfiability Modulo Theories* (SMT). Well-known solvers include Boolector, CVC4, MathSAT and Z3. SMT solvers offer a variety of *theories*, which determine the syntax and semantics of the input formulas. The theory typically used in verification of hardware or low-level software is the theory of bit-vectors combined with the theory of arrays. Solvers for this combination rely heavily on rewrite engines that exploit word-level equivalences between subformulas. When the word-level engine is unable to solve the problem using rewriting, the formula is incrementally translated to bit level and is then passed to the propositional SAT solver.

III. RTL Analysis via Translation into ANSI-C

A. Translating Verilog RTL to ANSI-C

The semiconductor industry is rapidly moving to building hardware designs above RTL to further increase design productivity [3], [13]. Although SoC designs are increasingly written at this higher level of abstraction, there is still a significant body of design IP blocks that are written in VHDL or Verilog. In this section, we briefly discuss the working of our tool V2C for translating Verilog to ANSI-C.

V2C translates synthesizable Verilog code to structurally equivalent word-level ANSI-C code based on a synthesis semantics interpretation of the Verilog code. We refrain from synthesis or simulation-based optimizations during the translation process in order to obtain a trustworthy translator. Figure 6 gives an example of translation performed by V2C. A very similar translation was proposed by Greaves [14], but we were unable to obtain a copy of his VTOC tool. Verilator is an open-source translator from Verilog to C++. But Verilator produces very large output code, which is acceptable for simulation but not suitable for formal analysis.

Verilog

```
module top(Din,En,CLK,Dout);
  wire cs; reg ns;
  input CLK, Din, En;
  output Dout;

  // Combinational Block
  assign Dout = cs;
  always @(Din or cs or En)
  begin
  if (En)
    ns = Din;
  else
    ns = cs;
  end
  ff ff1(ns,CLK,cs);
endmodule

// Sequential Block
module ff(Din, CLK, Dout);
  input Din, CLK;
  output Dout;
  reg q;
  assign Dout = q;
  always @(posedge CLK)
    q <= Din;
endmodule
```

C Program

```
struct s_ff {
 _Bool q;
} sff;

struct s_en {
 _Bool ns;
 struct s_ff sff;
} sen;

// Combinational Block
void top(_Bool CLK,_Bool Din,
_Bool En, _Bool *Dout) {
  _Bool cs;
  if(En) sen.ns = Din;
  else sen.ns = cs;
  ff(CLK, sen.ns, &cs);
  *Dout = cs;
}

// Sequential Block
_Bool ff(_Bool CLK,
_Bool Din, _Bool *Dout) {
  _Bool tmp0;
  tmp0 = Din;
  sff.q = tmp0;
  *Dout = sff.q;
  return;
}
```

Fig. 6. Example for translation of Verilog RTL to ANSI-C with V2C

B. Path-explosion Problem in Hardware Simulation

The path explosion problem in RTL design is much more severe than in typical application software code. This is caused by structural properties of RTL, which models concurrent computations. Sequential behavior is obtained using clocked logic, which corresponds to one large outer loop. This leads to very complex paths, which are uncommon in software programs. Further, the number of paths usually grows exponentially with the number of clock cycles. All these factors make RTL simulation challenging and result in scalability issues for modern hardware verification tools [13].

C. Application of Software Verifiers

Once the software model for the hardware design is obtained, we can apply a range of formal software analysis tools. We now briefly discuss some of the back-end analysis engines given in Figure 2.

Application of BMC: SAT-based BMC techniques can be easily applied to both RTL and C-style software languages. We used a multi-path symbolic execution engine, CBMC, to verify the system-level models of hardware IPs. Here, the transition relation for a system and its specification are jointly unwound to obtain a formula, which is then checked for satisfiability using an efficient SAT or SMT procedure. This corresponds to replicating the basic blocks along the path k times, followed by a transformation of the concatenation of these blocks into static single assignment (SSA) form [12]. CBMC implements loop unrolling and uses bit-flattening to decide the resulting bit-vector formula. The tool also supports SMT solvers as proof engines in the backend.

Application of Path-Symex: We implemented a single-path forward symbolic execution engine, PATH-SYMEX, that prunes out infeasible paths on the fly and generates incremental path constraints along the feasible branch in a program. The tool uses MiniSat 2.2.0 to perform incremental SAT solving for the generated path constraints. PATH-SYMEX can be combined with property-based slicing techniques to further scale up property verification of system-level models.

Application of Abstract Interpretation: Static program analysis based on abstract interpretation [15] has been widely used to verify certain classes of properties for safety-critical systems. In abstract interpretation, a given program is analysed with respect to a set of given abstract domains. We are not aware of earlier attempts to apply analyzers that are based on abstract interpretation to the verification of properties of register-transfer level circuits.

IV. Experimental Results

A. Experimental Setup

In this section, we report experimental results on the application of software verifiers to complex hardware IPs. All our experiments were performed on an Intel Xeon machine with 8 cores at 3.07 GHz with 48 GB RAM.

We use MiniSat 2.2.0 for purely propositional reasoning and the SMT solvers Z3 4.3.1, MathSAT 5.2.11, CVC4 1.4 and Yices 2.2.1 for word-level reasoning. We have observed only minor differences in the performance of the SMT solvers, and therefore omit the times for the runs with MathSAT, CVC4 and Yices.

We target two different classes of circuits: data-path intensive circuits, exemplified by DSP filters and arithmetic circuits, and control-intensive designs, represented by a USB PHY IP core, an Ethernet MAC IP, and an implementation of a cache coherence protocol from opencores.org. All benchmarks are available at www.cprover.org/hardware/v2c-isvlsi/ to enable other researchers to reproduce our results. We now provide a brief summary of the circuits and give examples of the properties we check.

B. Data-path Intensive Benchmarks

We verified properties of the implementation of a BCD-to-binary converter circuit, serial adder, parallel adder and an integer divider. Typical properties ranged from the absence of arithmetic overflow and underflow to the computation of the correct sum. The divider circuit implements a 16-bit restoring division algorithm. We also verified designs from the DSP domain, namely an FIR filter. The properties that were checked include the round-off error and several symmetry and periodicity properties.

C. Control-Intensive Benchmarks

USB PHY IP: The *USB PHY IP* core provides functions for interfacing to a USB 1.1 bus. This includes serial/parallel conversion, bit stuffing and unstuffing, NRZI encoding and decoding and a DPLL, and offers a UTMI interface. We verify the following properties for the USB PHY IP core:

- *Prop_1:* When rst=1, the UTMI interface of Transmitter and Receiver must be set *HIGH*. Thus the corresponding signals *TxReady_o* and *RxActive_o* of the Transmitter and Receiver must be both *HIGH*.
- *Prop_2:* In case of errors like like sync errors, bit-stuff error and byte-stuff error in transmitter and receiver module, the signals *TxError* and *RxError* must be flagged *HIGH*.

- *Prop_3:* If the transmission is in progress, i.e. $tx\_ip = 1$, then the *data_done* signal must be asserted *FALSE*. If $tx\_ip = 0$, then *data_done* must be asserted *HIGH* in the next cycle.

Cache Coherence Protocol: The MESI protocol is a widely used cache coherence and memory coherence protocol in shared-memory MIMD multiprocessor systems. The protocol supports write-back caches and is used to maintain the consistency between the L1 and L2 caches in the microprocessors from the Intel Pentium family. The snoopy cache controller continuously monitors the activity of the bus and takes actions whenever a bus transaction involves data stored in their local caches. Each cache line in MESI protocol can be in one of the four states, *Modified*, *Exclusive*, *Shared* and *Invalid*. Each cache controller may take different action when there is a request for data read or data write from the CPU. These include *Read Hit*, *Read Miss*, *Write Hit* and *Write Miss*. We verified properties relating to the read miss and write miss actions.

- *Prop_1:* If a cache has a read miss and another cache has that line in the *exclusive* state, then the cache line is changed to be in a shared state in both caches after the read is complete.
- *Prop_2:* In case of write miss, the cache controller sends signal to other caches to see if they have a copy of the line. If another cache has that line in *exclusive* state, or if several other caches have the line in *shared* state, then the state of the line is changed to *invalid*.

Ethernet MAC IP: The tri-mode Ethernet MAC implements a MAC controller conforming to the IEEE 802.3 specification and supports serial PHY and parallel PHY interfaces. It also supports automated pause frame generation and termination as well as half-duplex for the 10 and 100 Mbps modes. We verified the encoding and the decoding features in the transceiver module and various operations relating to the block sync and word sync functionality.

D. Discussion

The verification times reported in Table I are in seconds. Best times for a particular benchmark are in bold font. We use two variants of each design; one is safe and one is unsafe.

Columns 1–4 in Table I give the name of benchmark and whether the *safe (y)* or *unsafe (n)* version of the design is used, the *bound* up to which the design is unwound and the number of properties verified for each design, respectively. For smaller benchmarks such as BCD_BINARY, the adders, the integer divider and the FSM design, the sequential depth can be computed easily and is used as the bound for BMC. However, for large designs, computing the sequential depth is hard and thus we chose a high value (50) as the bound. The pair x/y in column 4 gives the number of passing (x) and failing properties (y) for the unsafe designs. The timeout (TO) is set to one hour.

The columns with the heading "Conventional Analysis" give the runtimes for applying Bounded Model Checking as implemented in EBMC 4.2 and HW-CBMC 5.0 to the original Verilog code in three different modes: at the netlist level using MiniSat and at RTL using either MiniSat or Z3.

The columns with the heading "Analysis using Software Verifiers" give the runtimes for analysing the C code generated by V2C using three different software verification tools. The first two columns give the runtime of CBMC 5.0 on the C program using the given bound. We report times using MiniSat and Z3. The second verifier is PATH-SYMEX 5.0, which is a path-based forward symbolic simulator. This technique heavily relies on incremental solving, which is not well supported by the SMT solvers that are currently available. We therefore only report the runtime when using the MiniSat backend, which has good support for incremental solving. Finally, the last two columns give the runtime taken by Astrée for unbounded and bounded analysis. Astrée implements abstract interpretation to verify safety properties of C source code [6].

We first discuss the results obtained with the conventional approaches. Our results reconfirm observations made in earlier work that the symbolic simulation of word-level RTL models outperforms netlist-based methods [3], [4]. However, our results also show that propositional SAT solvers still perform better than the SMT solvers for hardware property verification. SMT solvers strongly rely on built-in rewrite engines, which is known to be effective for certain equivalence checking tasks but ineffective for hardware property verification [4].

We now discuss the results obtained using our V2C tool and the software verifiers. Our experimental results highlight the potential of changing the design representation used in the verifiers. We observe that, for data-path intensive designs, BMC on the software models is on average 10.5 times faster than BMC on netlists. For control-intensive circuits, BMC on software models is on average 6.2 times faster than BMC on netlists. The symbolic simulators for software are consistently and significantly faster than their counterparts on word-level RTL or netlists. This holds true irrespective of the backend used. This is quantified by the average speed-up of 10 times for the software model over the word-level RTL using BMC implementation with MiniSAT as back-end SAT solver. With Z3 as the backend SMT solver, we obtain an average speed-up of 6.2 times for the software model over the word-level RTL model. We furthermore observe that PATH-SYMEX performs better than CBMC for some of the data-path intensive benchmarks and a few control-intensive benchmarks like the FSM design and the USB PHY IP core. CBMC with MiniSat is able to prove all benchmarks in reasonable time.

Astrée is the fastest verifier for a broad variety of circuits. Unlike the other methods, however, Astrée may report "Verification Unknown" (denoted by $*$), as the abstract domains that are used by the tool have been explicitly designed for proving the absence of runtime errors and may not be sufficiently precise to prove our functional properties. We also perform bounded analysis using Atrée. This is done by annotating the loops in the program with an unwinding bound, which helps Astrée to compute more precise results. Comparing the netlist-based analysis with the best software verifier, we observe an average speed-up of 50.8. This is mostly because Atrée outperforms the netlist-based approaches very significantly on the control-intensive benchmarks. Atrée was never optimised for hardware analysis, and we thus believe that there is scope here for new tools that implement abstract interpretation using abstract domains developed specifically for this task, e.g., by applying abstract conflict driven learning [16].

V. Related Work

There have been attempts to adapt software verification techniques to the analysis of hardware. For instance, Jain

Circuit	Safe	Bound	# Properties passing/failing	Conventional Analysis			Analysis using Software Verifiers				
				Netlist	RTL		CBMC		Path-Symex	Astrée	
				MiniSat	MiniSat	Z3	MiniSat	Z3	MiniSat	Unbounded	Bounded
Data-path Intensive Designs											
BCD_BINARY	y	8	4	0.47	0.46	0.51	0.14	0.12	0.13	**0.06**	0.08
	n		3/1	0.66	0.74	0.54	0.25	0.22	0.19	**0.03**	0.05
SERIAL_ADDER	y	10	20	66.12	68.61	56.19	6.47	5.86	**0.82**	$*$	3.93
	n		18/2	79.31	71.37	66.30	5.78	5.31	**0.97**	$*$	3.84
PARALLEL_ADDER	y	10	35	4.22	2.83	3.51	0.45	0.49	0.36	**0.22**	0.28
	n		30/5	3.89	2.91	2.78	0.54	0.45	0.37	**0.03**	0.05
FIR	y	50	30	7.91	4.67	6.89	0.37	2.97	0.27	**0.26**	0.28
	n		25/5	7.84	7.85	4.26	0.36	2.92	0.26	**0.13**	0.17
INTEGER_DIVISION	y	16	2	90.50	74.78	84.38	14.51	10.86	**1.66**	$*$	5.98
	n		1/1	15.86	10.56	68.74	15.23	7.15	0.27	$*$	**0.21**
Control-intensive Designs											
FSM	y	50	10	12.53	57.12	10.78	1.56	1.41	0.29	**0.18**	0.20
	n		7/3	13.75	58.17	10.23	1.57	1.65	0.41	**0.05**	0.07
CACHE COHERENCE	y	50	23	245.11	242.55	396.76	19.12	84.64	TO	$*$	10.49
	n		17/6	358.61	346.72	456.78	21.08	112.14		$*$	0.85
USB PHY IP CORE	y	50	25	342.78	337.17	TO	253.21	TO	33.68	**7.72**	17.58
	n		22/3	297.35	270.62		218.14		26.12	**0.13**	0.18
ETHERNET MAC IP	y	50	21	678.26	658.17	TO	571.84	TO	TO	$*$	**128.84**
	n		17/4	277.02	271.84		236.89			$*$	**18.24**

TABLE I. HARDWARE PROPERTY VERIFICATION FOR DESIGNS DESCRIBED AT DIFFERENT LEVELS OF GRANULARITY (TIMEOUT SET TO 1 HOUR, A $*$ DENOTES "VERIFICATION UNKNOWN")

et al. [17] have applied predicate abstraction, a technique made popular by Microsoft's Driver Verifier, to Verilog RTL. But these techniques still operate on the conventional coarse-grained representation as a transition system; they are therefore limited in their scalability.

There is a large body of work in which software verification methods are applied to *high-level* models of circuits, e.g., given in SystemC or SpecC [18]–[20]. In [21], a high-level description is partially synthesised into RTL, followed by co-verification together with the remaining part of the SystemC model. The main problem of this approach is obtaining a suitable high-level model. The closest work to ours is [13], in which a hardware design at register-transfer level is analyzed using path-based symbolic simulation, similar to our experiments with PATH-SYMEX.

VI. CONCLUSION

Demand for more scalable verification tools is ever growing. In this paper, we propose an alternative solution for verifying hardware, at the heart of which is a verifier for software. We suggest that verifiers should employ a software representation of circuits, which makes the advanced techniques in modern software analysis tools available. We evaluate this approach using three different program analysis techniques, which are BMC, path-based symbolic execution and abstract interpretation and observe performance gains between one and two orders of magnitude on our diverse set of benchmarks compared to conventional approaches based on netlists or RTL.

REFERENCES

[1] D. Kroening and O. Strichman, *Decision Procedures*. Springer, 2008.

[2] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah, "Reveal: A formal verification tool for Verilog designs," in *LPAR*, ser. LNCS, vol. 5330. Springer, 2008, pp. 343–352.

[3] M. Keating, *The Simple Art of SoC Design*. Springer, 2011.

[4] S. Sunkari, S. Chakraborty, V. M. Vedula, and K. Maneparambil, "A scalable symbolic simulator for Verilog RTL," in *MTV*, 2007, pp. 51–59.

[5] P. Bjesse, "A practical approach to word level model checking of industrial netlists," in *Computer Aided Verification (CAV)*, ser. LNCS, vol. 5123. Springer, 2008, pp. 446–458.

[6] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *Programming Language Design and Implementation (PLDI)*. ACM, 2003, pp. 196–207.

[7] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," in *POPL*, 2002, pp. 1–3.

[8] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007.

[9] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[10] J. Yang and C. H. Seger, "Introduction to generalized symbolic trajectory evaluation," in *International Conference on Computer Design (ICCD)*. IEEE, 2001, pp. 360–367.

[11] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*. USENIX, 2008, pp. 209–224.

[12] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, ser. LNCS. Springer, 2004, pp. 168–176.

[13] L. Liu and S. Vasudevan, "Scaling input stimulus generation through hybrid static and dynamic analysis of RTL," *ACM TODAES*, vol. 20, no. 1, pp. 4:1–4:33, 2014.

[14] D. J. Greaves, "A Verilog to C compiler," in *RSP*. IEEE Computer Society, 2000, pp. 122–127.

[15] P. Cousot, "Proving the absence of run-time errors in safety-critical avionics code," in *EMSOFT*, 2007, pp. 7–9.

[16] V. D'Silva, L. Haller, and D. Kroening, "Abstract conflict driven learning," in *Principles of Programming Languages (POPL)*. ACM, 2013, pp. 143–154.

[17] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "Word level predicate abstraction and refinement for verifying RTL Verilog," in *DAC*. ACM, 2005, pp. 445–450.

[18] E. Clarke, H. Jain, and D. Kroening, "Verification of SpecC using predicate abstraction," *Formal Methods in System Design (FMSD)*, vol. 30, no. 1, pp. 5–28, February 2007.

[19] N. Blanc and D. Kroening, "Race analysis for SystemC using Model Checking," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 15, no. 3, May 2010.

[20] A. Cimatti, I. Narasamdya, and M. Roveri, "Software model checking SystemC," *IEEE TCAD*, vol. 32, no. 5, pp. 774–787, 2013.

[21] D. Kroening and N. Sharygina, "Formal verification of SystemC by automatic hardware/software partitioning," in *MEMOCODE 2005*. IEEE, 2005, pp. 101–110.