Learning Conditional Abstractions

Bryan Brady
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-24
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-24.html

April 5, 2011

Copyright © 2011, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Learning Conditional Abstractions

Bryan A. Brady
UC Berkeley
bbrady@eecs.berkeley.edu

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

April 5, 2011

Abstract

Abstraction is central to formal verification. In term-level abstraction, the design is abstracted using a fragment of first-order logic with background theories, such as the theory of uninterpreted functions with equality. The main challenge in using term-level abstraction is determining what components to abstract and under what conditions. In this paper, we present an automatic technique to conditionally abstract register transfer level (RTL) hardware designs to the term level. Our approach is a layered approach that combines random simulation and machine learning inside a counter-example guided abstraction refinement (CEGAR) loop. First, random simulation is used to determine modules that are candidates for abstraction. Next, machine learning is used on the resulting simulation traces to generate candidate conditions under which those modules can be abstracted. Finally, a verifier is invoked. If spurious counter-examples arise, we refine the abstraction by performing a further iteration of random simulation and machine learning. We present an experimental evaluation on processor and low-power designs.

1 Introduction

Designs are usually specified at the register-transfer-level (RTL). For formal verification, however, RTL can be a rather low level of abstraction where data are represented as bits and bit vectors, and operations on data are accomplished by bit-level manipulation. In verification tasks that involve proving strongly data-dependent properties, such as equivalence or refinement checking, bit-level RTL quickly leads to state-space explosion, necessitating additional abstraction.

Term-level modeling can make formal verification of data-intensive properties tractable by abstracting away details of data representations and operations, viewing data as symbolic *terms* and operations as *uninterpreted* functions. Term-level abstraction has been found to be especially useful in microprocessor design verification [13, 17, 19, 20]. The precise functionality of units such as instruction decoders and the ALU are abstracted away using *uninterpreted functions*, and decidable fragments of first-order logic are employed in modeling memories, queues, counters, and other common constructs. Efficient satisfiability modulo theories (SMT) solvers [5] are then used as the computational engines for term-level verifiers.

A major obstacle for term-level verification is the need to generate term-level models from word-level RTL. Two recent efforts have sought to automate the generation of term-level models. Andraus and Sakallah [4] were the first to address the problem, proposing a counterexample-guided abstraction refinement (CEGAR) approach. While the CEGAR technique works in some cases, it can require very many iterations of abstraction-refinement in other situations. Brady et al. [8] proposed ATLAS, an approach that combines random simulation with static analysis to compute *interpretation conditions* — conditions under which a functional block is replaced with an uninterpreted function. ATLAS avoids the need for several abstraction-refinement iterations by computing conservative interpretation conditions using static analysis. However, in

some cases, these conditions are so large as to negate the advantages of term-level verification over word-level methods.

In this paper, we present CAL, a new technique for *automatically generating a term-level verification model* from a word-level description. The main focus of this work is function abstraction. Similar to ATLAS, CAL conditionally abstracts functional blocks in the original design with uninterpreted functions. In contrast with previous work, CAL uses a novel layered approach based on a combination of *random simulation*, *machine learning*, and *counterexample-guided abstraction-refinement*. In the first stage, we exploit the module structure specified by the designer using random simulation to identify functional blocks corresponding to module instantiations that are suitable for abstraction with uninterpreted functions. Replacing functional blocks with uninterpreted functions is always sound, that is, the correctness of the resulting abstract design implies the correctness of the original design. However, this abstraction loses information and can lead to spurious counterexamples. In the second stage, we use machine learning inside a CEGAR loop to rule out such spurious counterexamples. First, a verifier is invoked on the unconditionally abstracted verification model. If spurious counterexamples arise, machine learning is used to compute conditions under which abstraction can be performed without loss of precision; i.e., if the resulting term-level design is in correct, then so is the original word-level design. This process is repeated until we arrive with a term-level model that is valid or a legitimate counterexample is found. Fig. 1 illustrates the CAL approach.

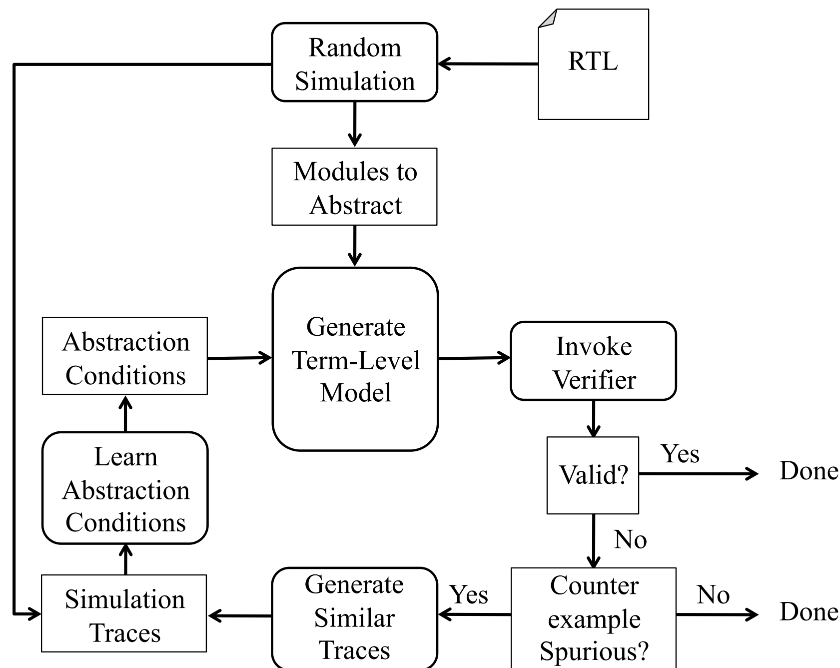

Figure 1: **The CAL approach** A CEGAR-based approach, CAL identifies candidate abstractions with random simulation and uses machine learning to refine the abstraction if necessary.

We present experimental evidence that our approach is efficient and that the resulting term-level models are easier to verify. Moreover, we show that the abstraction conditions that we learn are as good or better than the previous best-known conditions.

The rest of this paper is organized as follows. We discuss some background material and related work in Section 2. In Section 3, we present the formal model for our work as well as some relevant ideas borrowed from our previous work on ATLAS [8]. Our new approach, CAL, is described in Section 4. Case studies

are discussed in detail in Section 5. We conclude in Section 6.

2 Background and Related Work

Background material on term-level abstraction is presented in Sec. 2.1, function abstraction in Sec. 2.2, and related work in Sec. 2.3.

2.1 Term-Level Abstraction

Informally, a (word-level) design is said to be abstracted to the *term-level* if one or more of the following three abstraction techniques is employed [8]:

1. *Function Abstraction:* In function abstraction, bit-vector operators and modules computing bit-vector values are treated as "black-box," *uninterpreted* functions constrained only by functional consistency. That is, they must evaluate to the same values when applied to the same arguments. It is possible for the inputs and outputs of uninterpreted functions to be bit vectors or to be abstract terms (say, interpreted over $\mathbb{Z}$). Function abstraction is the focus of this paper, and we limit ourselves to uninterpreted functions that map bit vectors to bit vectors.
2. *Data Abstraction:* Bit-vector expressions are modeled as abstract terms that are interpreted over a suitable domain (typically a subset of $\mathbb{Z}$). Data abstraction is effective when it is possible to reason over the domain of abstract terms far more efficiently than it is to do so over the original bit-vector domain, through use of small-domain or bit-width reduction techniques. Data abstraction is not the focus of this paper.
3. *Memory Abstraction:* In memory abstraction, memories and data structures are modeled in a suitable theory of arrays or memories, such as by the use of special **read** and **write** functions [13] or lambda expressions [11]. We do not address automatic memory abstraction in this paper.

2.2 Function Abstraction

The concept of function abstraction is illustrated using a toy ALU design [8]. Consider the simplified ALU shown in Figure 2(a). Here a 20-bit instruction is split into a 4-bit opcode and a 16-bit data field. If the opcode indicates that the instruction is a jump, the data field indicates a target address for the jump and is simply passed through the ALU unchanged. Otherwise, the ALU computes the square of its 16-bit input and generates as output the resulting 16-bit result.

Using very coarse-grained term-level abstraction, one could abstract the entire ALU module with a single uninterpreted function (UF), as shown in Figure 2(b). However, we lose the precise mapping from `instr` to `out`.

Such a coarse abstraction is quite easy to perform automatically. However, this abstraction loses information about the behavior of the ALU on jump instructions and can easily result in spurious counterexamples. In Section 3.2, we will describe a larger equivalence checking problem within which such an abstraction is too coarse to be useful.

Suppose that reasoning about the correctness of the larger circuit containing this ALU design only requires one to precisely model the difference in how the jump and squaring instructions are handled. In this case, it would be preferable to use a partially-interpreted ALU model as depicted in Figure 2(c). In this model, the control logic distinguishing the handling of jump and non-jump instructions is precisely modeled, but the datapath is abstracted using the uninterpreted function SQ. However, creating this fine-grained abstraction

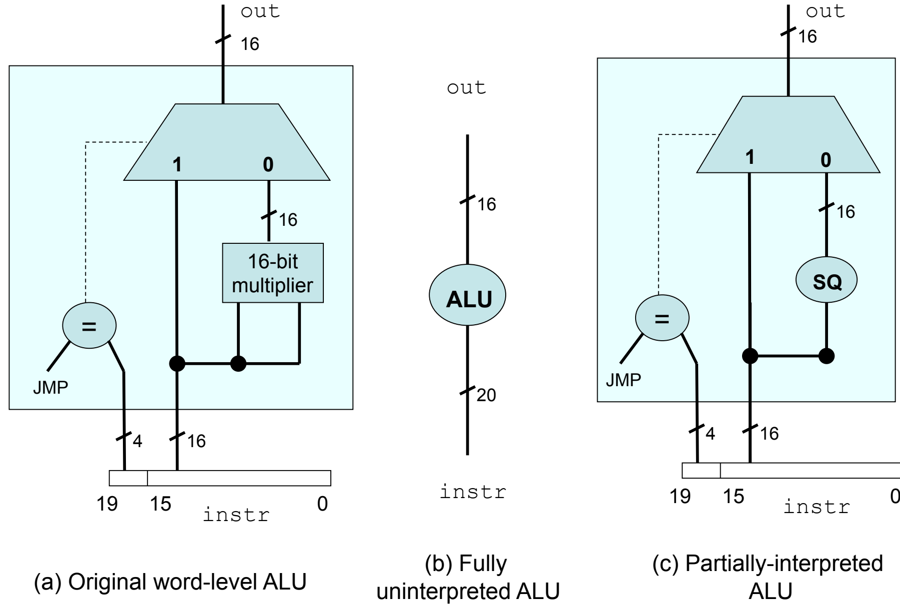

Figure 2: **Three versions of an ALU design.** Boolean signals are shown as dashed lines and bit-vector signals as solid black lines [8].

by hand is difficult in general and places a large burden on the designer. It is this burden that we seek to mitigate using the approach presented in this paper.

2.3 Related Work

The first automatic term-level abstraction tool was Vapor [4], which aimed at generating term-level models from Verilog. The underlying logic for term-level modeling in Vapor is CLU, which originally formed the basis for the UCLID system [11]. Vapor uses a counterexample-guided abstraction-refinement (CEGAR) approach [4]. Vapor has been since subsumed by the Reveal system [2, 3] which differs mainly in the refinement strategies in the CEGAR loop.

Both Vapor and Reveal start by completely abstracting a Verilog description to the UCLID language by modeling all bit-vector signals as abstract terms and all operators as uninterpreted functions. Next, verification is attempted on the abstracted design. If the verification succeeds, the tool terminates. However, if the verification fails, it checks whether the counterexample is spurious using a bit-vector decision procedure. If the counterexample is spurious, a set of bit-vector facts are derived, heuristically reduced, and used on the next iteration of term-level verification. If the counterexample is real, the system terminates, having found a real bug.

The CEGAR approach has shown promise [3]. In many cases, however, several abstraction-refinement iterations are needed to infer fairly straightforward properties of data, thus imposing a significant overhead. For instance, in one example, a chip multiprocessor router [22], the header field of a packet must be extracted and compared several times to determine whether the packet is correctly forwarded. If any one of these extractions is not modeled precisely at the word-level, a spurious counterexample results. The translation is complicated by the need to instantiate relations between individually accessed bit fields of a word modeled as a term using special uninterpreted functions to represent concatenation and extraction operations.

A more recent approach to automatic abstraction is ATLAS [8]. ATLAS is a hybrid approach based on a combination of random simulation and static analysis. Random simulation is used to determine suitable candidates for abstraction with uninterpreted functions. Static analysis is used to heuristically compute conditions under which it is precise to abstract (i.e., performing abstraction will not generate spurious counterexamples.) In many cases, ATLAS is able to compute conditions that lead to easier-to-verify models. However, there are some limitations. Some designs require many iterations of static analysis to compute abstraction conditions which can lead to exponentially large abstraction conditions. Just computing the expressions becomes a bottleneck in such designs. Furthermore, the conservative nature of the heuristics used to compute the abstraction conditions can sometimes lead to the abstraction conditions saturating to **false**, meaning that it is never the case that we can abstract functionality with uninterpreted functions. In this case, ATLAS can create harder to verify models due to the additional constraints required to model functional consistency.

The main difference between the approach described in this paper and ATLAS is the way in which abstraction conditions are computed. Instead of using static analysis to compute these conditions, we use a dynamic approach based on machine learning. The benefit of using a dynamic approach is that the conditions we learn are based on actual spurious counterexamples. Aside from this main difference, our approach shares many features with ATLAS. To the best of our knowledge, our paper is the first to use machine learning to dynamically compute conditions under which it is precise to abstract. As is the case with ATLAS, our new approach can also be combined with bitwidth reduction techniques (e.g. [6, 18]) to perform combined function and data abstraction.

To our knowledge, Clarke, Gupta et al. [14,16] were the first to use machine learning to compute abstractions for model checking. Our work is similar in spirit to theirs. One difference is that we generate term-level abstract models for SMT-based verification, whereas their work focuses on bit-level model checking and localization abstraction. Consequently, the learned concept is different: CAL learns Boolean interpretation conditions whereas their technique learns sets of variables to make visible. Additionally, our use of machine learning is more direct — e.g., while Clarke et al. [14] also use decision tree learning, they only indirectly use the learned decision tree (all variables branched upon in the tree are made visible), whereas we use the Boolean function corresponding to the entire tree as the learned interpretation condition.

3 Preliminaries

We adopt the formal model used in [8] and present it in Sec. 3.1 and give an illustrative example in Sec. 3.2.

3.1 Basic Definitions

We model a design at the word level as a *word-level netlist* $\mathcal{N} = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{C}, Init, \mathcal{A})$ where

- $\mathcal{I}$ is a finite set of input signals;
- $\mathcal{O}$ is a finite set of output signals;
- $\mathcal{S}$ is a finite set of intermediate sequential (state-holding) signals;
- $\mathcal{C}$ is a finite set of intermediate combinational (stateless) signals;
- $Init$ is a set of initial states, i.e., initial valuations to elements of $\mathcal{S}$, and
- $\mathcal{A}$ is a finite set of assignments to outputs and to sequential and combinational intermediate signals. An assignment is an expression that defines how a signal is computed and updated. We elaborate below on the form of assignments.

First, note that input and output signals are assumed combinational, without loss of generality. In word-level designs, a memory is modeled as a flat array of bit-vector signals.

A *combinational assignment* is a rule of the form $v \leftarrow e$, where v is a signal in the disjoint union $\mathcal{C} \uplus \mathcal{O}$ and e is an expression that is a function of $\mathcal{C} \uplus \mathcal{S} \uplus \mathcal{I}$. Combinational loops are disallowed. We differentiate between combinational assignments based on the type of the right-hand side expression and write them as follows:

$$v \leftarrow \mathit{bv} \;\mid\; b \leftarrow \mathit{bool}$$

Here *bv* and *bool* represent bit-vector and Boolean expressions in a word-level design, as listed in the grammar in Fig. 3.

$$\begin{aligned} \mathit{bv} &::= c \mid v \mid \mathit{ITE}(b,\, v_1,\, v_2) \mid \mathsf{bvop}(v_1, \ldots, v_k) \quad (k \geq 1) \\ \mathit{bool} &::= \mathbf{true} \mid \mathbf{false} \mid b \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \\ &\quad \mid v_1 = v_2 \mid \mathsf{bvrel}(v_1, \ldots, v_k) \quad (k \geq 1) \end{aligned}$$

Figure 3: **Syntax for Bit-Vector and Boolean Expressions.** c and v denote a bit-vector constant and variable respectively, and b is a Boolean variable. bvop denotes any arithmetic operator mapping bit vectors to bit vectors, while bvrel is a relational operator other than equality mapping bit vectors to a Boolean value.

Again, we differentiate between sequential assignments based on type, and write them as follows (where v, u are any bit-vector signals and b, b_a are any Boolean signals):

$$v := u \;\mid\; b := b_a$$

Note that we assume that the right-hand side of a sequential assignment is a signal; this loses no expressiveness since we can always introduce a new name to represent any expression.

A *word-level design* $\mathcal{D}$ is a tuple $\langle \mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\} \rangle$, where $\mathcal{I}$ and $\mathcal{O}$ denotes the set of input and output signals of the design, and the design is partitioned into a collection of N word-level netlists. A *well-formed* design is one where (i) every output of a netlist is either an output of the design or an input to some netlist (including itself) – i.e., there are no dangling outputs; and (ii) every input of a netlist is either an input to the design or exactly one output of some netlist. We refer to the netlists $\mathcal{N}_i$ as *functional blocks*, or *fblocks*.

We revise the expression syntax in order to model designs at the term level, with the revisions shown in Fig. 4. Since data abstraction is not addressed in this paper, we exclude abstract term-level expressions from the syntax. We include memories in the syntax since we employ memory abstraction in this paper for some verification problems. The interpreted memory functions **read** and **write** are used for brevity; we can use any specific memory modeling technique including the use of lambda expressions [11].

$$\begin{aligned} \mathit{bv} &::= \mathbf{read}(M, v) \mid UF(v_1, \ldots, v_k) \quad (k \geq 0) \\ \mathit{bool} &::= UP(v_1, \ldots, v_k) \quad (k \geq 0) \\ \mathit{mem} &::= A \mid M \mid \mathbf{write}(M, v_1, v_2) \end{aligned}$$

Figure 4: **Syntax for Term-Level Expressions.** We only show additions to the expression syntax of Fig. 3. UF and UP denote an uninterpreted function and predicate symbol respectively. A and M denote constant and variable memories. The second argument to $\mathbf{read}$ and $\mathbf{write}$ denote addresses and the third argument to $\mathbf{write}$ denotes the data value to be written.

A *term-level netlist* is a generalization of a word-level netlist where expressions can be both from the syntax shown in Figure 3 and that in Fig. 4. Additionally, a term-level netlist can have sequential and combinational assignments to memory variables, of the form below (where M, M_1 are any memory signals and *mem* is any memory expression):

$$M := \mathit{mem}$$

A term-level netlist that has at least one expression of the form $UF(v_1, \ldots, v_k)$ or $UP(v_1, \ldots, v_k)$ is referred to as a *strict term-level netlist*.

A *term-level design* $\mathcal{T}$ is a tuple $(\mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\})$, where each fblock $\mathcal{N}_i$ is a term-level netlist.

Given a word-level design $\mathcal{D} = (\mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\})$, we say that $\mathcal{T}$ is a *term-level abstraction* of $\mathcal{D}$ if $\mathcal{T}$ is obtained from $\mathcal{D}$ by replacing some word-level fblocks $\mathcal{N}_i$ by strict term-level fblocks $\mathcal{N}'_i$.

The verification problems of interest in this paper are *equivalence checking* and *refinement checking*.

Given two word-level designs $\mathcal{D}_1$ and $\mathcal{D}_2$, the *word-level equivalence* (*word-level refinement*) checking problem is to verify that $\mathcal{D}_1$ is *sequentially equivalent* to (*refines*) $\mathcal{D}_2$.

The definition is similarly extended to a pair of term-level designs $\mathcal{T}_1$ and $\mathcal{T}_2$. We also consider bounded equivalence checking problems, where the designs are to be proved equivalent for a bounded number of cycles from the initial state.

The *term-level abstraction problem* we consider in this paper is as follows.

> Given a pair of word-level designs $\mathcal{D}_1$ and $\mathcal{D}_2$, abstract them to term-level designs $\mathcal{T}_1$ and $\mathcal{T}_2$, such that $\mathcal{D}_1$ is equivalent to (refines) $\mathcal{D}_2$ if and only if $\mathcal{T}_1$ is equivalent to (refines) $\mathcal{T}_2$.

We follow the approach taken by ATLAS and generate the term-level abstraction by computing an *interpretation condition* — a condition under which we will retain the precise fblock in the term-level model (i.e, we replace the fblock by an uninterpreted function under the negation of the interpretation condition). The idea of conditional function abstraction is illustrated in Figure 6.

In Section 4, we present our CAL approach to automatically generate term-level abstract models. In Section 5, we show that using CAL can scale up verification by orders of magnitude.

3.2 Illustrative Example

Figure 5 depicts the equivalence checking problem that we will use as a running example. Two variants of the same circuit, denoted Design A and Design B, are to be checked for output equivalence.

Consider Design A. This design models a fragment of a processor datapath. PC models the program counter register, which is an index into the instruction memory denoted as IMem. The instruction is a 20-bit word denoted `instr`, and is an input to the ALU design shown earlier in Figure 2(a). The top four bits of `instr` are the operation code. If the instruction is a jump instruction (`instr`[`19 : 16`] equals JMP), then the PC is set equal to the ALU output `out`; otherwise, it is incremented by 4.

Design B is virtually identical to Design A, except in how the PC is updated. For this version, if `instr`[`19 : 16`] equals JMP, the PC is directly set to be the jump address `instr`[`15 : 0`].

Note that we model the instruction memory as a read-only memory using an uninterpreted function IMem. The same uninterpreted function is used for both Design A and Design B. We also assume that Designs A and B start out with identical values in their PC registers.

The two designs are equivalent iff their outputs are equal at every cycle, meaning that the Boolean assertion `out_ok` $\wedge$ `pc_ok` is always **true**.

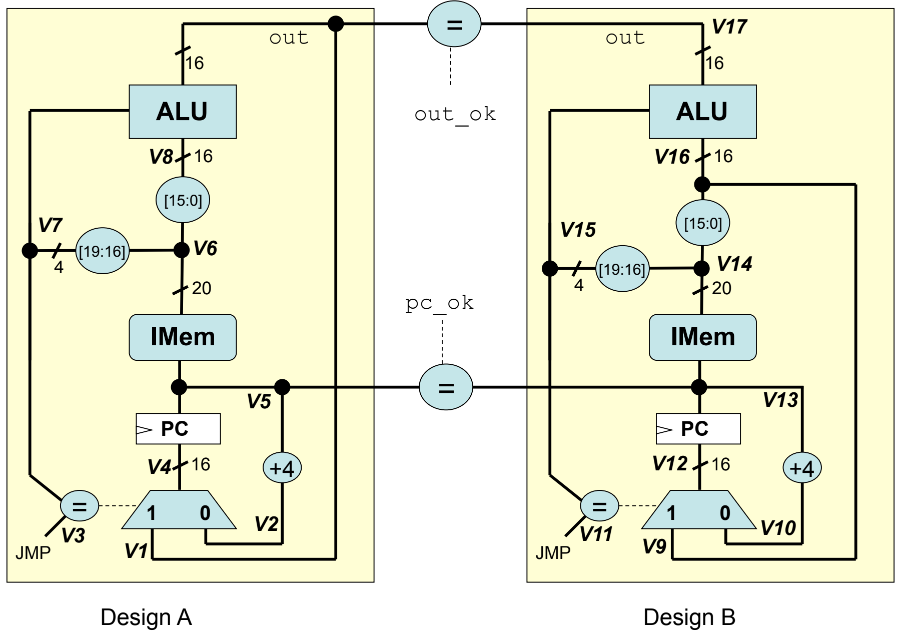

Figure 5: **Equivalence checking of two versions of a portion of a processor design.** Boolean signals are shown as dashed lines and bit-vector signals as solid lines.

It is easy to see that this is the case, because from Figure 2(a) we know that `A.out` always equals `A.instr`[15 : 0] when `A.instr`[19 : 16] equals JMP. The question is whether we can infer this without the full word-level representation of the ALU.

Consider what happens if we use the abstraction of Figure 2(b). In this case, we lose the relationship between `A.out` and `A.instr`[19 : 16]. Thus, the verifier comes back to us with a spurious counterexample, where in cycle 1 a jump instruction is read, with the jump target in Design A different from that in Design B, and hence `A.PC` differs from `B.PC` in cycle 2.

However, if we instead used the partial term-level abstraction of Figure 2(c) then we can see that the proof goes through, because the ALU is precisely modeled under the condition that `A.instr`[19 : 16] equals JMP, which is all that is necessary.

The challenge is to be able to generate this partial term-level abstraction automatically. We describe our approach to solving this problem below.

4 The CAL Approach

The main contribution of this paper is presented in this section. The goal of this step is to compute conditions under which it is precise to abstract using a machine-learning-based CEGAR loop.

4.1 Identifying Candidate fblocks

The first step in CAL is the same as in ATLAS: to use syntactic matching and random simulation to identify a set of fblocks that are candidates for replacement with uninterpreted functions. We review this procedure

in this section since it is crucial to understand the rest of the CAL procedure.

The first step in identifying candidates for abstraction is to identify *replicated fblocks*. A replicated fblock is an fblock in $\mathcal{D}_1$ that has an isomorphic counterpart in $\mathcal{D}_2$. A formal definition can be found in [8]. In equivalence and refinement checking problems, identifying replicated fblocks is a matter of finding instances of the same RTL module present in both designs.

The fblock identification process generates a collection of sets of fblocks $\mathcal{FS} = \{\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k, \}$. Each set $\mathcal{F}_j$ contains replicated fblocks that are isomorphic to each other. Each set $\mathcal{F}_j$ is considered to be an equivalence class of the fblocks it contains. In later steps when function abstractions are computed, it is important to note that the same function abstraction is used for each fblock in $\mathcal{F}_j$.

The next step in the abstraction candidate identification process is to determine which equivalence classes $\mathcal{F} \in \mathcal{FS}$ will be considered for abstraction. This is accomplished using random simulation.

Let the cardinality of $\mathcal{F}$ be l. Let each fblock $f_i \in \mathcal{F}$ have m bit-vector output signals $\langle v_{i1}, \ldots, v_{im} \rangle$, and n input signals $\langle u_{i1}, \ldots, u_{in} \rangle$. Then, we term the tuple of corresponding output signals $\chi = (v_{1j}, v_{2j}, \ldots, v_{lj})$, for each $j = 1, 2, \ldots, m$, as a tuple of *isomorphic output signals*.

Given a tuple of isomorphic output signals $\chi = (v_{1j}, v_{2j}, \ldots, v_{lj})$, we create a *random function* RF_χ unique to χ that has n inputs (corresponding to input signals $\langle v_{i1}, \ldots, v_{in} \rangle$, for fblock f_i).

For each fblock f_i, $i = 1, 2, \ldots, l$, we replace the assignment to the output signal v_{ij} with the random assignment $v_{ij} \leftarrow RF_\chi(u_{i1}, \ldots, u_{in})$. This substitution is performed for all output signals $j = 1, 2, \ldots, m$. The resulting designs $\mathcal{D}_1$ and $\mathcal{D}_2$ are then verified through simulation. This process is repeated for T different random functions RF_χ.

If the fraction of failing verification runs is greater than a threshold τ, then we drop the equivalence class $\mathcal{F}$ from further consideration. Otherwise, we retain $\mathcal{F}$ for further analysis, as described in the following section. We denote the set of equivalence classes that are to be considered for abstraction as $\mathcal{FS}_\mathcal{A}$.

4.2 Top-Level CAL Procedure

The top-level CAL procedure, VERIFYABS, is shown in Algorithm 1. VERIFYABS takes two arguments, the design $\mathcal{D}$ being verified and the set of equivalence classes being abstracted $\mathcal{FS}_\mathcal{A}$. Initially, the interpretation conditions $\mathsf{c}_i \in \mathcal{IC}$ are set to **false** meaning that we start by unconditionally abstracting the fblocks in $\mathcal{D}$. The procedure CONDABS creates the abstracted term-level design $\mathcal{T}$ from the word-level design $\mathcal{D}$, the set of equivalence classes to be abstracted $\mathcal{FS}_\mathcal{A}$, and the set of interpretation conditions $\mathcal{IC}$. Next, we invoke a term-level verifier on $\mathcal{T}$. If VERIFY $(\mathcal{T})$ returns "Valid", we report that result and terminate. If a counterexample arises, we evaluate the counterexample on the word-level design. If the counterexample is non-spurious, we report the counterexample and terminate, otherwise we store the counterexample in $\mathcal{CE}$ and invoke the abstraction condition learning procedure, LEARNABSCONDS $(\mathcal{D}, \mathcal{FS}_\mathcal{A}, \mathcal{CE})$.

We say that VERIFYABS is *sound* if it reports "Valid" iff $\mathcal{D}$ is correct. It is *complete* if any counterexample reported by it when it terminates is a true counterexample (i.e., not spurious). We have the following guarantee for the procedure VERIFYABS:

Theorem 1 *If* VERIFYABS *terminates, it is sound and complete.*

Proof: Any term-level abstraction is a sound abstraction of the original design, since any partially-interpreted function (for any interpretation condition) is a sound abstraction of the fblock it replaces. Thus VERIFYABS is sound. Moreover, VERIFYABS terminates with a counterexample only if it deems the counterexample to be non-spurious, by simulating it on the concrete design $\mathcal{D}$. Therefore VERIFYABS is complete (when it terminates). □

In order to guarantee termination of VERIFYABS, we must impose certain constraints on the learning algorithm LEARNABSCONDS. This is formalized in the theorem below.

Theorem 2 *Suppose that the learning algorithm* LEARNABSCONDS *satisfies the following properties:*

(i) If c_i denotes the interpretation condition for an fblock learned in iteration i of the VERIFYABS *loop, then $\mathsf{c}_i \implies \mathsf{c}_{i+1}$ and $\mathsf{c}_i \neq \mathsf{c}_{i+1}$;*

(ii) The trivial interpretation condition **true** *belongs to the hypothesis space of* LEARNABSCONDS*, and*

(iii) The hypothesis space of LEARNABSCONDS *is finite.*

Then, VERIFYABS *will terminate and return either* Valid *or a non-spurious counterexample.*

Proof: Consider an arbitrary fblock that is a candidate for function abstraction. Let the sequence of interpretation conditions generated in successive iterations of the VERIFYABS loop be $\mathsf{c}_0 = \mathbf{false}, \mathsf{c}_1, \mathsf{c}_2, \ldots$. By condition (i), $\mathsf{c}_0 \implies \mathsf{c}_1 \implies \mathsf{c}_2 \implies \ldots$ where $\mathsf{c}_i \neq \mathsf{c}_{i+1}$. Since no two elements of the sequence are equal, and the hypothesis space is finite, no element of the sequence can repeat. Thus, the sequence (for any fblock) forms a finite chain of implications. Moreover, since **true** belongs to the hypothesis space, in the extreme case, VERIFYABS can generate in its final iteration the term-level design $\mathcal{T}$ identical to the original design $\mathcal{D}$, which will yield termination with either Valid or a non-spurious counterexample. □

In practice, the conditions (i)-(iii) stated above can be implemented on top of any learning procedure. The most straightforward way is to set an upper bound on the number of iterations that LEARNABSCONDS can be invoked, after which the interpretation condition is set to **true**. Another option is to set c_{i+1} to $\mathsf{c}_i \vee d_{i+1}$ where d_{i+1} is the condition learned in the $i+1$th iteration. Yet another option is to keep a log of the interpretation conditions generated, and if an interpretation condition is generated for a second time, the abstraction procedure is terminated by setting the interpretation condition to **true**. Many other heuristics are possible; we leave an exploration of these to future work.

Algorithm 1 Procedure VERIFYABS $(\mathcal{D}, \mathcal{FS}_{\mathcal{A}})$: Top-level CAL verification procedure.

```
1:  // Input: Combined word-level design (miter)
    D := ⟨I, O, {N_i | i = 1, ..., N}⟩
2:  // Input: Equivalence classes of fblocks
    FS_A := {F_j | j = 1, ..., k}
3:  // Output: Verification result
    Result ∈ {Valid, CounterExample}
4:  Set c_i = false for all c_i ∈ IC.
5:  while true do
6:     T = CONDABS (D, FS_A, IC)
7:     Result = VERIFY (T)
8:     if Result = Valid then
9:        Return Valid.
10:    else
11:       Store counterexample in CE.
12:       if CE is spurious then
13:          IC ←LEARNABSCONDS (D, FS_A, CE)
14:       else
15:          Return CounterExample.
16:       end if
17:    end if
18: end while
```

4.3 Conditional Function Abstraction

Procedure CONDABS $(\mathcal{D}, \mathcal{FS}_\mathcal{A}, \mathcal{IC})$ is responsible for creating a term-level design $\mathcal{T}$ from the original word-level design $\mathcal{D}$, the set of equivalence classes to be abstracted $\mathcal{FS}_\mathcal{A}$, and the set of interpretation conditions $\mathcal{IC}$. Algorithm 2 outlines the conditional abstraction procedure.

CONDABS operates by iterating through the equivalence classes in $\mathcal{FS}_\mathcal{A}$. A fresh uninterpreted function symbol UF_j is created for each tuple of isomorphic output signals χ_j associated with equivalence class $\mathcal{F}_i \in \mathcal{FS}_\mathcal{A}$. Each output signal $v_{ij} \in \chi_j$ is conditionally abstracted with UF_j. Fig. 6 illustrates how the output signals of an fblock are conditionally abstracted. The original word-level circuit is shown in Fig. 6(a) and the conditionally abstracted version with interpretation condition c is shown in Fig. 6(b).

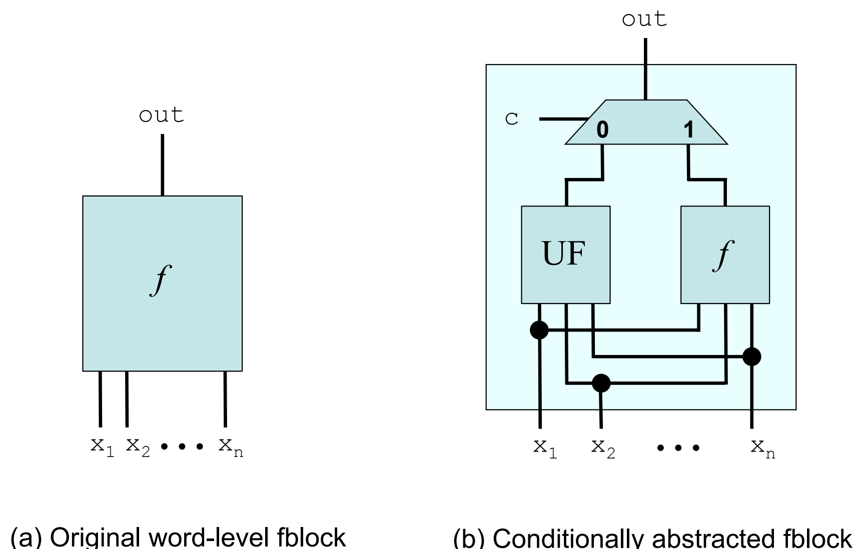

Figure 6: **Conditional abstraction** (a) Original word-level fblock f. (b) Conditionally abstracted version of f with interpretation condition c

4.4 Learning Conditional Abstractions

Spurious counterexamples arise due to imprecision introduced during abstraction. More specifically, when a spurious counterexample arises, it means that at least one fblock $f_i \in \mathcal{F}$ (where $\mathcal{F} \in \mathcal{FS}_\mathcal{A}$) is being abstracted when it needs to be modeled precisely. In the context of our abstraction procedure VERIFYABS, if VERIFY $(\mathcal{T})$ returns a spurious counterexample $\mathcal{CE}$, then we must invoke the procedure LEARNABSCONDS $(\mathcal{D}, \mathcal{FS}_\mathcal{A}, \mathcal{CE})$.

The LEARNABSCONDS procedure invokes a *decision tree learning* algorithm on traces generated by randomly mutating fblocks $f_i \in \mathcal{F}$ to generate "good" and "bad" traces. Good traces are those where the mutation does not lead to a property violation; the other traces are bad. The learning algorithm generates a classifier in the form of a decision tree to separate the good traces from the bad ones. The classifier is essentially a Boolean function over signals in the original word-level design. More information about decision tree learning can be found in Mitchell's textbook [21].

There are three main steps in the LEARNABSCONDS procedure:

1. Generate good and bad traces for the learning procedure;

Algorithm 2 Procedure CONDABS $(\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{IC})$: Create term-level design $\mathcal{T}$ from word-level design $\mathcal{D}$, the set of fblocks being abstracted $\mathcal{FS}_{\mathcal{A}}$, and the set of interpretation conditions $\mathcal{IC}$.

1: // Input: Combined word-level design (miter)
   $\mathcal{D} := \langle \mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\} \rangle$
2: // Input: Equivalence classes of fblocks
   $\mathcal{FS}_{\mathcal{A}} := \{\mathcal{F}_j \mid j = 1, \ldots, k\}$
3: // Input: Set of interpretation conditions: $\mathcal{IC}$
4: // Output: Term-level design: $\mathcal{T}$
5: **for** each equivalence class $\mathcal{F}_i \in \mathcal{FS}_{\mathcal{A}}$ **do**
6:     Let χ_j denote the isomorphic output tuple associated with the j-th output signal in fblock $f_i \in \mathcal{F}_i$
7:     **for all** $j \in 1, ..., k$ **do**
8:         Create a fresh uninterpreted function symbol UF_j.
9:         Let $(i_1, ..., i_k)$ denote the input symbols to fblock f_i.
10:        **for** each output signal $v_{ij} \in \chi_j$ **do**
11:            Let $\mathsf{c}_{v_{ij}} \in \mathcal{IC}$ denote the interpretation condition associated with v_{ij}
12:            Replace the assignment $v_{ij} \leftarrow e$ in f_i with the assignment $v_{ij} \leftarrow \mathit{ITE}(\mathsf{c}_{v_{ij}}, e, UF_j(i_1, ..., i_k))$
13:        **end for**
14:    **end for**
15: **end for**
16: Return the resulting term-level design $\mathcal{T}$

2. Determine meaningful features that will help decision tree learning procedure compute high quality decision trees, and
3. Invoke a decision tree learning algorithm with the above features and traces.

The data input to the decision tree software is a set of tuples where one of the tuple elements is the target attribute and the remaining elements are features. In our context, a target attribute α is either Good or Bad. Our goal is to select features such that we can classify the set of all tuples where $\alpha = \mathsf{Bad}$ based on the rules provided by the decision tree learner. Since we use an off-the-shelf decision tree learning tool, we omit a description of how this works. It is very important to provide the decision tree learning with quality input data and features, otherwise, the rules generated will not be of use. The data generation procedure is described in Sec. 4.5 and feature selection is described in Sec. 4.6.

4.5 Generating Data

In order to obtain high quality decision trees, we need to generate good and bad traces for the design being verified. Of course, whenever the procedure LEARNABSCONDS is called, we have a spurious counterexample stored in $\mathcal{CE}$. However, a single counterexample trace is far from adequate for learning, since it will result in a trivial decision tree stating that the target attribute α is always Bad.

To produce a meaningful decision tree, we must provide the decision tree learner with both good and bad traces. To generate the traces for the decision tree learner, we use a modified version of the random simulation procedure described in Sec. 4.1. Instead of simulating the abstract design when only a single fblock has been replaced with a random function, we replace all fblocks with their respective random functions at the same time and perform verification via simulation.

Let RF_{ij} be the random function associated with the j-th isomorphic output signal of $\mathcal{F}_i \in \mathcal{FS}_{\mathcal{A}}$. Then we replace the assignment to the output signal v_{ij} with the random assignment $v_{ij} \leftarrow RF_{ij}(u_{i1}, ..., u_{in})$ for all output signals $j = 1, 2, ..., m$. Now, instead of only logging the result of the simulation, we log the value of

every signal in the design for every cycle of each simulation. It is up to the feature selection step to decide what signals are important.

To produce good traces, or witnesses, we randomly set the initial state of the design $\mathcal{D}$ before each run of simulation. This usually results in simulation runs that pass. Recall that at this stage we only consider fblocks that produce failing runs in a small fraction of simulation runs. Let $Good$ be the set of all witnesses produced in this step.

In order to generate bad traces, we have several options:

1. Set the initial state of each simulation run so that it is consistent with the initial state of the counterexample stored in $\mathcal{CE}$. This option usually results in many bad traces due to the fact that we're starting from a state which we know can lead to an error state. One drawback to this technique is that while we know the trace is a counterexample, it may not be obvious which cycle is causing the counterexample to arise. We give a concrete example of this in Sec. 5.2.
2. Generate additional counterexamples using the VERIFY procedure where the previously seen counterexamples are ruled out. The drawback to this approach is that using a formal verifier is likely to be more expensive than running a small number of simulations.
3. Modify the property that is being checked by the formal verifier. When performing bounded-model checking it is possible to construct a property such that any counterexample violates the property in every stage of bounded model checking, as opposed to only failing in a single stage. Not all equivalence or refinement checking problems lend themselves to this approach. However, when it is possible to use this option, it can reduce the number of traces required to create a quality decision tree. We give an example of this in Sec. 5.3

We denote the set of all bad traces by Bad. We annotate each trace in Bad with the Bad attribute and each trace in $Good$ with the Good attribute.

4.6 Choosing Features

The quality of the decision tree generated is highly dependent on the features used to generate the decision tree. We use two heuristics to identify features:

1. Include input signals to the fblock being abstracted.
2. Include signals encoding the "unit-of-work" being processed by the design, such as the instruction being executed.

Input signals. Suppose we wish to determine when fblock f must be interpreted. It is very likely that whether or not f must be interpreted is dependent on the inputs to f. So, if f has input signals $(i_1, i_2, ..., i_n)$ it is almost always the case that we would include the input arguments as features to the decision tree learner.

Unit-of-work signals. There are cases when the input arguments alone are not enough to generate a quality decision tree. In these cases, human insight can be provided by defining the unit-of-work being performed by the design. For example, in a microprocessor design, a unit-of-work is an instruction. Similarly, in a network-on-a-chip (NoC), the unit-of-work is a packet, where the relevant signals could include the source address, destination address, or possibly the type of packet being sent across the network. Once a unit-of-work is identified at one part of the design, one can propagate this information to identify all signals directly derived from the original unit-of-work signals. For instance, in the case of a pipelined processor, the registers storing instructions in each stage of the pipeline are relevant signals to treat as features.

In rare cases, the above heuristics are not enough to generate quality decision trees; we discuss these scenarios and give additional features in Sec. 5.

5 Case Studies

We performed three case studies to evaluate CAL. Each of these case studies has also been verified using ATLAS. Additionally, each case study requires a non-trivial interpretation condition (i.e., an interpretation condition different from **false**). The first case study involves verifying the example shown in Fig. 5. Next, we verify, via correspondence checking, two versions of the the Y86 microprocessor. Finally, we perform equivalence checking between a low-power multiplier and the corresponding non-power-aware version.

All experiments were run on a MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor with 4GB RAM. The term-level verification engine used for the experiments was the UCLID verification system [1, 9] with Minisat2 [15] and Boolector [10] as the SAT and SMT backends, respectively. Random simulation was performed using Icarus Verilog [24]. The decision tree learner we used in the experiments is C5.0 [23].

5.1 The Illustrative Example

In this experiment, we perform equivalence checking between Design A and B shown in Fig. 5. First, we initialize the designs to the same initial state and inject an arbitrary instruction. Then we check whether the designs are in the same state. The precise property that we wish to prove is that the ALU and PC outputs are the same for design A and B. Let $\mathtt{out_A}$ and $\mathtt{out_B}$ denote the ALU outputs and $\mathtt{pc_A}$ and $\mathtt{pc_B}$ denote the PC outputs for designs A and B, respectively. The property we prove is:

$$\mathtt{out_A} = \mathtt{out_B} \wedge \mathtt{pc_A} = \mathtt{pc_B}$$

Aside from the top-level modules, the design consists of only two modules, the instruction memory (IMEM) and the ALU. We do not consider the instruction memory for abstraction because we do not address automatic memory abstraction. The ALU passes the random simulation stage, so it is an abstraction candidate.

For this example, we have the benefit of knowing a priori the exact interpretation condition needed in order to precisely abstract the ALU module. A counterexample is generated during the first verification stage due to the unconditionally abstracted ALU. Due to the rather simple and contrived nature of this example, not many signals need to be considered as features for the decision tree learner. The features we use in this case are arguments to the ALU; the instruction and the data arguments. The interpretation condition learned from the trace data is $\mathsf{op} = \mathsf{JMP}$ where op is the top 4 bits of the instruction.

Interpretation Condition	UCLID Runtime (sec)	
	SAT	SMT
true	28.51	27.01
$\mathsf{op} = \mathsf{JMP}$	**0.31**	**0.01**

Table 1: **Performance comparison** UCLID runtime comparison for the processor fragment shown in 5. The runtime associated with the model abstracted with CAL is shown in **bold**.

5.2 The Y86 Processor

In this experiment, we verify two versions of the well-known Y86 processor model introduced by Bryant and O'Hallaron [12]. The Y86 processor is a pipelined CISC microprocessor styled after the Intel IA32 instruction set. While the Y86 is relatively small for a processor, it contains several realistic features, such as a dual read, dual write register file, separate data and instruction memories, branch prediction, hazard resolution, and an ALU that supports bit-vector arithmetic and logical instructions. Of the several variants of the Y86 processor we focus on two that have different versions of branch prediction logic: NT and BTFNT. In NT branches are predicted as not taken, whereas in BTFNT branches backwards in the address

space are predicted as taken, while branches forward in the address space are predicted as not taken. NT and BTFNT were the designs that the ATLAS approach had the most difficulty abstracting [8].

The property we wish to prove on the Y86 variants is Burch-Dill style correspondence-checking [13]. In correspondence checking, a pipelined version of a processor is checked against a single-cycle version. The main goal of correspondence checking is to verify that the pipeline control logic allows all of the same behaviors that the instruction set architecture (ISA) supports. Each run of correspondence checking involves injecting an instruction into the pipeline and subsequently flushing the pipeline to allow the effects of the injected instruction to update the processor state. A single run of correspondence checking requires over a dozen cycles of symbolic simulation.

Both NT and BTFNT versions have the same module hierarchy and differ only in the logic pertaining to branch prediction. The following modules are candidates for abstraction: register file (RF), condition code (CC), branch function (BCH), arithmetic-logic unit (ALU), instruction memory (IMEM), and data memory (DMEM). The RF module is ruled out as a candidate for abstraction during the random simulation stage due to a large number of failures during verification via simulation. This occurs because an uninterpreted function is unable to accurately model a mutable memory. We do not consider IMEM and DMEM for automatic abstraction because they are memories and we do not address automatic memory abstraction in this work. Instead, we manually model IMEM and DMEM with completely uninterpreted functions. The CC and BCH modules are also removed from consideration due to the relatively simple logic contained within them. Abstracting these modules is unlikely to yield substantial verification gains and may even hurt performance due to the overhead associated with uninterpreted functions. This leaves us with the ALU module.

During the random simulation phase the ALU generates very few, if any, failures. In fact, in order to get a failure, it is usually necessary to simulate for 10,000–100,000 runs of correspondence checking; potentially several hundred thousand to a million individual cycles before obtaining an error. Of course we never run that many cycles of random simulation. Instead, we perform at most 500-1000 runs of correspondence checking and usually do not receive a violation. However, in the first iteration of VERIFYABS we obtain a spurious counterexample which requires the computation of non-trivial interpretation conditions. We use this counterexample to initialize the initial state of the processor models as discussed in Sec. 4.5.

5.2.1 Decision tree feature selection

In the case of both BTFNT and NT using only the arguments of the abstracted ALU is not sufficient to generate a useful decision tree. The ALU takes three arguments, the op-code op and two data arguments a and b. Closer inspection of the data provided to the decision tree learner reveals a problem. In almost every single case in both good and bad traces, the ALU op is equal to ALUADD and the b argument is equal to 0, in almost every cycle of correspondence checking. The underlying cause of this poor data stems from a perfectly reasonable design decision. Many of the Y86 instructions do not require the ALU to perform any operation. In these cases, the default values fed into the ALU are $op = \mathsf{ALUADD}$ and $b = 0$.

In this situation, the arguments to the ALU are not good features by themselves. Conceptually, the unit-of-work that we are performing in a pipelined processor is a sequence of instructions, specifically the instructions that are currently in the pipeline. When we include the instructions that are in the pipeline during the cycle in which the instruction is injected as described earlier in this section, we are able to obtain a much more high quality decision tree, or interpretation condition. In fact, the interpretation condition obtained when considering all instructions currently in the pipeline during the cycle in which the new instruction is injected is:

$$\mathsf{c} := Instr_E = \mathsf{JXX} \wedge Instr_M = \mathsf{RMMOV}$$

The interpretation condition c indicates that we need to interpret the ALU whenever there is a JUMP in-

struction (JXX) in the execute stage and there is a register-to-memory move in the memory stage. While this interpretation condition is an improvement over the original decision tree, it will still lead to further spurious counterexamples. The reason is that we are now including too many features, some of which have no effect on whether the ALU needs to be interpreted. These additional features restrict the situations when the ALU is interpreted and this causes further spurious counterexamples. A logical step is to include as a feature only the instruction that is currently in the ALU. So, when we use the following features: $Instr_E$, op, a, and b we obtain the interpretation condition:

$$\mathsf{c}_{E,b} := Instr_E = \mathsf{JXX} \wedge b = 0$$

This is the best interpretation condition we can hope for. In fact, in previous attempts to manually abstract the ALU in the BTFNT version, we used:

$$\mathsf{c}_{Hand} := op = \mathsf{ALUADD} \wedge b = 0$$

When we compare the runtimes for verification of the Y86-BTFNT processor, we see that verifying BTFNT with the interpretation condition $\mathsf{c}_{E,b}$ outperforms the unabstracted version and the previously best known abstraction condition (c_{Hand}). Table 2 compares the UCLID runtimes for the Y86 BTFNT model with the different versions of the abstracted ALU. The

Interpretation Condition	UCLID Runtime (sec)	
	SAT	SMT
true	> 1200	> 1200
c_{Hand}	133.03	105.34
$\mathsf{c}_{E,b}$	**101.10**	**65.52**

Table 2: **Performance comparison** UCLID runtime comparison for Y86-BTFNT for different interpretation conditions. The runtime associated with the model abstracted with CAL is shown in **bold**.

Note that the Y86 NT model required an additional level of abstraction-refinement regardless of the feature selection.

5.2.2 Abstraction-refinement

The NT version of the Y86 processor requires an additional level of abstraction refinement. In general, requiring multiple versions of abstraction refinement is not interesting by itself. However, it is interesting to see how the interpretation conditions change using this machine learning based approach.

Attempting unconditional abstraction of the ALU in the NT version results in a spurious counterexample. The interpretation condition learned from the traces generated in this step is $\mathsf{c} := a = 0$. It is interesting that the same interpretation condition is generated regardless of whether we consider all of the instructions as features, or only the instruction in the same stage as the ALU. Not surprisingly, the second attempt at verification using the interpretation condition c results in another spurious counterexample. This time, the interpretation condition generated differs depending on whether or not we consider all of the instructions as features or only the instruction in the execute stage. In this case, when all of the instructions in the pipeline are used as features, we obtain the interpretation condition $\mathsf{c}_E := Instr_E = \mathsf{ALUADD}$, which states that we must interpret anytime an addition operation is present in the ALU. Verification is successful when c_E is used as the interpretation condition.

If we were to use only the instruction in the execute stage as a feature, we obtain the trivial decision tree saying to always interpret. The reason we obtain the trivial decision tree is because the arguments to the

ALU and the instruction in the execute stage are exactly the same for every trace that is sent to the decision tree learner. When we include all of the instructions in the pipeline, we obtain a better result because in the good traces, the instructions in stages other than execute are different.

A performance comparison for the NT variant of the Y86 processor is shown in Table 3. Unlike the BTFNT case, the abstraction condition we learn for the NT model is not quite as precise as the previously best known interpretation condition, and the performance isn't as good. However, the runtimes for conditional abstraction, including the time spent in abstraction-refinement, are smaller than that of verifying the original word-level circuit. That is, the runtime when the interpretation condition is c_E is accounting for two runs of UCLID that produce a counterexample and an additional run when the property is proven "Valid". Note that the most precise abstraction condition is the same for both BTFNT and NT. The best performance on the NT version is obtained when the interpretation condition $\mathsf{c}_{BTFNT} := \mathit{Instr}_E = \mathsf{JXX} \wedge b = 0$ is used.

Interpretation Condition	UCLID Runtime (sec)	
	SAT	SMT
$\mathbf{true}$	> 1200	> 1200
c_{Hand}	154.95	89.02
c_E	**191.34**	**187.64**
c_{BTFNT}	94.00	52.76

Table 3: **Performance comparison** UCLID runtime comparison for Y86-NT for different interpretation conditions. The runtime associated with the model abstracted with CAL is shown in **bold**.

The reason for this seemingly disparate behavior between the BTFNT and NT models is due to the way in which the error manifests itself in the counterexample trace. In the case of the BTFNT model, because the ALU is abstracted, the program counter which is supposed to pass through the ALU unaltered gets mangled due to an unconditional abstraction. This causes a discrepancy between the logic that predicts branches and the logic that squashes mispredicted branches. The abstracted ALU causes a properly predicted branch instruction to be squashed, or a mispredicted branch instruction to be allowed to continue through the pipeline. In the NT model, the branch instruction is not in the execute stage, it is the instruction being injected. When the branch is mispredicted, the program counter gets updated with the wrong value. However, by using the instruction in the execute stage as the feature, we are misguiding the decision learning procedure. Instead, we should have used the instruction in the fetch stage, which was the source of the discrepancy.

5.3 Low-Power Multiplier

The next example we experimented with is a low-power multiplier design first introduced in [7]. Consider a design that has a multiplier which can be powered down when no multiply instructions are present to save power. Any time a multiply instruction is issued, the multiplier would have to be powered up in order to compute the multiplication. An optimization to this design is to send any multiply instructions in which at least one operand is a power of 2 to a shifter. Figure 7 illustrates this design. The pow2 module takes inputs `a` and `b` and generates `cond` which is true if either operand is a power of 2. If one or more operand is a power of 2, pow2 ensures that the proper shift amount is computed and sent to the appropriate shifter input. If neither operand is a power of two, `cond` is false, so the inputs to the shifter are don't cares. To safely use such a circuit, we must first verify that the optimized multiplier performs the same computation as a regular multiplier. We consider the unoptimized multiplier module for abstraction, because a) it is the only replicated fblock in the design and b) it passes the random simulation stage.

Note that in the implementation of the design in Fig. 7(b) there are two signals `a_is_pow_2` and `b_is_pow_2` that are $\mathbf{true}$ when `a` or `b`, respectively, are powers of two. When performing unconditional abstraction on the miter constructed between the multiplier designs, a spurious counterexample is generated. This happens

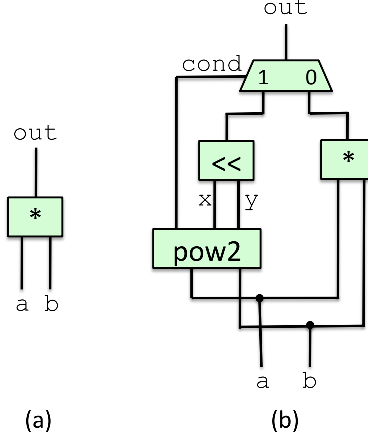

Figure 7: **Low-power design example.** (a) the original, word-level multiplier; (b) multiplier that uses a shifter when an operand is a power of 2.

due to the fact that when either input argument is a power of two, then the output of the low-power multiplier is precise. In order for the verification to succeed, we must capture this behavior in an interpretation condition. Note that because this design is essentially a combinational equivalence checking problem, we will only have a single counterexample trace, regardless of how many times we perform random simulation. The counterexample traces generated will all be the same because a single "run" of this verification problem is checking combinational equivalence.

In this case, we can generate multiple counterexamples by unrolling the circuit many times and constructing a property that is valid if and only if there is never a time frame in which the two circuits differ. A counterexample of this property will be a trace in which the two circuits have different results in every stage. We use this method to generate bad traces for the decision tree learner. To generate good traces, we simply randomly simulate the circuits for some number of cycles. The chance of one of the inputs being a power of two is so small that it is unlikely to occur in a small number of simulation runs. In addition to the arguments to the multiplier circuits, a and b, we consider $\texttt{a_is_pow_2}$ and $\texttt{b_is_pow_2}$ as features. The interpretation condition obtained due to this configuration of features and traces is $\mathsf{c}_{pow2} := \texttt{a_is_pow_2} \vee \texttt{b_is_pow_2}$

The performance comparison for the equivalence checking problem between a standard multiplier and the multiplier optimized for power savings is shown in Table 4. While the abstracted designs show a small speedup in most cases, it is by no means a performance victory. Instead, what this experiment shows is that we are able to learn interpretation conditions for another type of circuit and verification problem. Additionally, performing abstraction on this circuit doesn't cause a performance degradation.

BMC Depth	UCLID Runtime (sec)			
	SAT		SMT	
	No Abs	Abs	No Abs	Abs
1	2.81	**2.55**	1.27	**1.38**
2	12.56	**14.79**	2.80	**2.63**
5	67.43	**22.45**	8.23	**8.16**
10	216.75	**202.25**	21.18	**22.00**

Table 4: **Performance comparison** UCLID runtime comparison for equivalence checking between multiplier and low-power multiplier. The runtime associated with the model abstracted with CAL is shown in **bold**.

5.4 Comparison with ATLAS

ATLAS and CAL compute the same interpretation conditions for the processor fragment described in Sec. 5.1 and the low-power multiplier described in Sec. 5.3. Thus, the only interesting comparison with regard to the interpretation conditions is for the Y86 design.

ATLAS is able to verify both BTFNT and NT Y86 versions with one caveat—the multiplication operator was removed from the ALU to create a more tractable verification problem. When multiplication is present inside the ALU, the ATLAS approach can not verify BTFNT or NT in under 20 minutes. In the case where the multiplication operator is removed, the interpretation conditions generated by ATLAS for both BTFNT and NT are quite large, even though the procedure to generate the conditions is iterated very few times. Running this procedure for more iterations leads to exponential growth of the interpretation condition expressions. Upon closer inspection of the interpretation conditions, it turns out that the expressions simplify to `true` (i.e., no abstraction takes place). In this case, ATLAS actually takes longer to verify BTFNT as shown in [8]. This behavior highlights the main drawback of ATLAS. The static analysis procedure blindly takes into account the structure of the design, giving equal importance to every signal. In reality, bugs stem from very specific situations where only small fragments of the overall design contribute to the buggy behavior. This was the inspiration behind the using machine learning to compute interpretation conditions. Not only is CAL able to verify the BTFNT and NT Y86 versions when multiplication *is* included in the ALU, but it does so with an order of magnitude speedup over the unabstracted version.

5.5 Remarks

We have mainly focused thus far on the runtime taken by UCLID. The remaining runtime taken by the other components of the CAL procedure is, in comparison, negligible. First, the runtime of the decision tree learner is less than 0.1 seconds in every case. Second, the simulation time is quite small. For instance, simulating 1000 correspondence checking runs for the Y86 model takes less than 5 seconds. Whereas we are unable to verify the original word-level Y86 designs within 20 minutes, so the CAL runtime is negligible. Note that the simulation runtime for the low-power multiplier example is even smaller than that of the Y86 design. For instance, it takes less than 0.2 seconds to simulate the low-power multiplier for 1000 cycles. The number of good and bad traces required to produce a quality decision tree for the processor fragment example in Sec. 5.1 and the low-power multiplier example in Sec. 5.3 is 5 (10 total). For the Y86 examples, the number of good and bad traces was 50 (100 total). Thus, in every example, it takes only a fraction of a second to generate enough data for the machine learning algorithm to be able to produce useful results.

A key point to note is that while the entire CAL procedure is automated, human insight can be invaluable in speeding up verification. For instance, if a designer or verification engineer could mark the most important signals in a design, we could give those signals priority when choosing features to give to the decision tree learner. In the context of the Y86 examples, if the designer would specify up front that the instruction code signals for each stage were important, we could fully automate the examples shown in this paper. Noticing that the instruction codes are important signals in a processor design does not require any leap of faith. It is obvious that they are important—they directly affect the operation of the design being verified!

6 Conclusion

In this paper, we present CAL, an automatic abstraction procedure based on a combination of random simulation and machine learning. We evaluate the effectiveness and efficiency of our approach on equivalence and refinement checking problems involving pipelined processors and low-power designs. We have shown that we are able to automatically learn conditional abstractions that lead to better verification performance.

Additionally, we learned abstraction conditions that were better than the previously best known abstraction conditions for two variants of the Y86 microprocessor design.

References

[1] UCLID Verification System. Available at `http://uclid.eecs.berkeley.edu`.

[2] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proceedings of ASP-DAC*, pages 19–24, 2006.

[3] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. CEGAR-based formal hardware verification: A case study. Technical Report CSE-TR-531-07, University of Michigan, May 2007.

[4] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of Verilog models. In *Proceedings of the 41st Design Automation Conference (DAC)*, pages 218–223, 2004.

[5] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[6] P. Bjesse. A practical approach to word level model checking of industrial netlists. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 446–458, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] B. A. Brady. Low-power verification with term-level abstraction. In *Proceedings of TECHCON 2010*, September 2010.

[8] B. A. Brady, R. E. Bryant, S. A. Seshia, and J. W. O'Leary. ATLAS: automatic term-level abstraction of RTL designs. In *Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July 2010. To appear.

[9] B. A. Brady, S. A. Seshia, S. K. Lahiri, and R. E. Bryant. *A User's Guide to UCLID Version 3.0*, October 2008.

[10] R. D. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *In* Proc. of TACAS, pages 174–177, March 2009.

[11] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.

[12] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, 2002. Website: `http://csapp.cs.cmu.edu`.

[13] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In D. L. Dill, editor, *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 68–80. Springer-Verlag, June 1994.

[14] E. M. Clarke, A. Gupta, J. H. Kukula, and O. Strichman. Sat based abstraction-refinement using ilp and machine learning techniques. In *Proc. Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 265–279, 2002.

[15] N. Eén and N. Sörensson. The MiniSAT Page. `http://minisat.se`.

[16] A. Gupta and E. M. Clarke. Reconsidering CEGAR: Learning good abstractions without refinement. In *Proc. International Conference on Computer Design (ICCD)*, pages 591–598, 2005.

[17] W. A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[18] P. Johannesen. BOOSTER: Speeding up RTL property checking of digital designs through word-level abstraction. In *Computer Aided Verification*, 2001.

[19] S. K. Lahiri and R. E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 341–354, 2003.

[20] P. Manolios and S. K. Srinivasan. Refinement maps for efficient verification of processor models. In *Design, Automation, and Test in Europe (DATE)*, pages 1304–1309, 2005.

[21] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[22] L.-S. Peh. *Flow Control and Micro-Architectural Mechanisms for Extending the Performance of Interconnection Networks*. PhD thesis, Stanford University, August 2001.

[23] R. Quinlan. Rulequest research. `http://www.rulequest.com`.

[24] S. Williams. Icarus verilog. `http://www.icarus.com/eda/verilog/`.