Universal Boolean Functional Vectors

Jesse Bingham
Intel Corporation
Hillsboro, Oregon, U.S.A.
Email: jesse.d.bingham@intel.com

***Abstract*—In the simplest setting, one represents a boolean function using expressions over variables, where each variable corresponds to a function input. So-called *parametric representations*, used to represent a function in some restricted subspace of its domain, break this correspondence by allowing inputs to be associated with *functions*. This can lead to more succinct representations, for example when using binary decision diagrams (BDDs). Here we introduce *Universal Boolean Functional Vectors* (UBFVs), which also break the correspondence, but done so such that all input vectors are accounted for. Intelligent choice of a UBFV can have a dramatic impact on BDD size; for instance we show how the hidden weighted bit function can be efficiently represented using UBFVs, whereas without UBFVs BDDs are known to be exponential for any variable order. We show several industrial examples where the UBFV approach has a huge impact on proof performance, the "killer app" being floating point addition, wherein the wide case-split used in the state-of-the-art approach is entirely done away with, resulting in 70-fold reduction in proof runtime. We give other theoretical and experimental results, and also provide two approaches to verifying the crucial "universality" aspect of a proposed UBFV. Finally, we suggest several interesting avenues of future research stemming from this program.**

I. Introduction

Binary Decision Diagram (BDD) techniques for formal verification (FV) have fallen out of fashion as a research topic in the past decade. Nevertheless, the data structure is still widely used in industry to solve real-world hardware verification problems, for example at companies such as Intel [23], IBM [24], [29], and Centaur [28]. Furthermore, contemporary commercial FV tools also include BDDs in their spectrum of technologies. Perhaps one of the most successful domains for these techniques is FV of arithmetic data-path hardware designs. The efficacy of BDDs stems from the fact that they constitute a canonical representation of boolean functions, and for many functions of practical importance, the BDDs are of tractable size. It is well-known that the variable order can drastically influence the size of a BDD. Unfortunately, there are many functions, both artificial and practical, that have been shown to have exponentially large BDDs regardless of the ordering. To combat these roadblocks, techniques such as *case-splitting* and proof *decomposition* have been explored.

In this paper, we propose an orthogonal approach to avoiding blow up. Typically, one constructs a BDD for a function f in a setting wherein BDD variables and the inputs of f are in one-to-one correspondence. This correspondence can be broken when using parametric substitutions [3] to perform case-splitting; there, (not necessarily variable) functions are associated with f's inputs, with the goal of restricting the space in which f is represented. Our approach, called *Universal Boolean Functional Vectors* (UBFV) is similar to parametric substitutions in that *functions* are associated with f's inputs, however, unlike parametric substitutions, a UBFV representation of f does not restrict the space of representation (ergo, "universal"). In other words, all assignments to f's inputs are implicitly represented in the UBFV representation.

To illustrate this concept, consider the function $f(v_1, v_2, v_3) = v_1 \vee v_2\overline{v_3}$. Suppose we perform the substitution $(v_1, v_2, v_3) \mapsto (a \vee b, d, \overline{bc})$, where a, b, c, and d are fresh variables. Applying this substitution to f yields a new function $f' = a \vee b \vee dc$. Even though f' bears no syntactic resemblance to f, the former completely characterizes the latter in the sense that one can evaluate f for any input using only f' and the substitution. The complete characterization is possible only because $(a \vee b, d, \overline{bc})$ is *universal* in the sense that all 2^3 of the possible boolean assignments to (v_1, v_2, v_3) can be realized via assignments to (a, b, c, d); such a substitution is what we call a UBFV.

But what advantage can be achieved by performing these UBFV substitutions? We show that there exists functions, both theoretical examples and those arising in practical FV, with BDD representations being exponentially more compact when one selects an appropriate UBFV substitution. For the practical FV problems, this approach has a profound impact on proof runtime requirements. This is because where the current solution to avoid the exponential blow up involves performing many case-splits, by employing UBFVs we can reduce the number of cases drastically, often eliminating the need for case-splitting altogether. We also give a practical example where using UBFVs removes the need for proof decomposition – this can reduce proof *development* effort.

Our contributions are as follows. We lay down the groundwork for the theory of universal boolean functional vectors. and prove a key result (Theorem 2) showing that those functions that have small *partitioned BDDs* [22] also have UBFV representations with small BDDs. As a corollary, we prove that the hidden weighted bit function [5] has a cubic BDD for an appropriate UBFV. Deciding if a given substitution constitutes a UBFV is NP-hard. However we provide a BDD-based algorithm that is sufficient for the UBFV for the hidden weighted bit function and for our industrial examples, and also a user-assisted approach with low complexity. Finally, we report very encouraging performance speed-ups for real industrial proofs, namely floating point (FP) addition (FADD) and FP fused-multiply add (FMA) instructions.

II. Related Work

The idea of using parametric representations of boolean functions (what we term BFVs) goes back to the early 1970s [6], [9]. Such a representation, generated by the *generalized co-factor* operation (GCF) (a.k.a. *constrain*) was introduced in the setting of hardware model checking by Coudert and Madre [11]. Later, Jones *et al.* [3], [18] proposed a specialized variation of GCF called *param*, for use in symbolic simulation, e.g. STE [26]. Similar to us, Jain and Gopalakrishnan [17] employ problem-specific recipes (rather than use a generic algorithm) to create parametric representations for hardware verification. However, a common theme in these works is the restriction of the space of a boolean function to simplify its representation (often BDDs); we believe ours is the first published approach that strives to simplify the representation *without* a space restriction.[1]

The industrial example where our approach is extremely effective is the verification of FADD. Published FADD proofs [10], [3], and those solving the related problem of FMA verification [16], [21], [27] require wide case-splitting, which we altogether eliminate (or at least drastically reduce, in the case of FMA).

A good introduction to the use of BDDs in hardware FV is the paper by Hu [14].

III. Mathematical Foundations

A. Boolean Functions

Let $\mathbb{B}$ be the boolean constants $\{0, 1\}$ and let V be a finite set of *boolean variables*. A *V-assignment* (or simply *assignment* if V is understood) is a function $\alpha : V \to \mathbb{B}$. A *(boolean) function (over V)* is a function f taking V-assignments to $\mathbb{B}$, i.e. $f : (V \to \mathbb{B}) \to \mathbb{B}$. An assignment α is said to *satisfy* f if $f(\alpha) = 1$; f is said to be *satisfiable* (resp. *tautological*) if f is satisfied by some (resp. by all) assignments. We denote a tautological function as $\mathbf{1}$ and an unsatisfiable function by $\mathbf{0}$. We will employ overbar for boolean negation, juxtaposition or $\wedge$ for conjunction, $\vee$ for disjunction, and $\leftrightarrow$ for boolean equality. As is well known, any function over V can be represented as a formula using these operators and V. We will often leave α implicit, for instance $xy \vee z$ represents the function f over $\{x, y, z\}$ such that $f(\alpha) = \alpha(x)\alpha(y) \vee \alpha(z)$. Also, if $V \subseteq V'$, and f is defined to be a function over V, we can freely employ f as a function over V' in the obvious way.Finally, we say the function f over V is a *variable* if there exists $v \in V$ such that $f(\alpha) = \alpha(v)$ for all assignments α.

Much of what follows involves placing a total order $\preceq$ on sets of variables; if the set of variables is subscripted with integers, we say the *natural order* is the ordering that simply applies $\leq$ to the subscripts.

[1] Anecdotes within the walls of Intel recall similar *ad-hoc* trickery done in the past [15], though the idea was not explored thoroughly as we do in this paper, nor was the application to FADD/FMA known.

B. (Universal) Boolean Functional Vectors

A common operation on boolean functions is that of *substitution*, in which functions are substituted for the variables of another function. In this paper, the objects that describe substitutions are called *boolean functional vectors* (BFV) [13]. Given (not necessarily disjoint) sets of variables V and V', a BFV over (V, V') is a function $\psi : V \to ((V' \to \mathbb{B}) \to \mathbb{B})$ that takes the variables V to boolean functions over the other set of variables V'. We transpose the arguments of ψ to obtain the function $\psi^* : (V' \to \mathbb{B}) \to (V \to \mathbb{B})$ defined by $\psi^*(\alpha')(v) = \psi(v)(\alpha')$. Now applying a BFV ψ over (V, V') as a substitution against any function f over V is achieved via the composition

$$f \circ \psi^* : (V' \to \mathbb{B}) \to \mathbb{B}$$

Aside from being employable as a substitution for functions over V, a BFV ψ is also characterized by a particular V-function c. We say that c is the *characteristic* of the BFV ψ if for all V-assignments α we have that α satisfies c if and only if there exists a V'-assignment α' such that $\alpha = \psi^*(\alpha')$. Said another way, as we range across all possible V'-assignments α', $\psi^*(\alpha')$ ranges across exactly the set of V-assignments that satisfy c. When necessary to distinguish between the two variable sets V and V', we will respectively refer to them as the *primary* and the *secondary* variables.

Now we may define the central concept of this paper: a *universal BFV* (UBFV) is simply a BFV that has as the characteristic the tautological function $\mathbf{1}$. Equivalently, ψ is universal iff ψ^* is surjective. The key observation about UBFVs is that when one is used as a substitution against a function f, the result incurs no loss of information regarding f. This is formalized by the following lemma.

Lemma 1. *Let ψ be a UBFV over (V, V') and let f be a function over V. Then for any $\alpha : V \to \mathbb{B}$, there exists $\alpha' : V' \to \mathbb{B}$ such that $\alpha = \psi^*(\alpha')$ and thus $(f \circ \psi^*)(\alpha') = f(\alpha)$*

.

Proof. Follows from the fact that ψ is a UBFV; note however that α' is not necessarily unique. □

Lemma 1 says that $f \circ \psi^*$ is a legitimate representation of f; given only $f \circ \psi^*$ and ψ, we can evaluate f for any V-assignment. Since BFV substitution commutes with any boolean connective, we can apply boolean connectives in the "domain" of a UBFV ψ, for instance for two functions f_1 and f_2 and a boolean connective $\odot$, we have

$$(f_1 \circ \psi^*) \odot (f_2 \circ \psi^*) \;=\; (f_1 \odot f_2) \circ \psi^*$$

It follows from Lemma 1, importantly, that we can check for function equality in this domain as well — f_1 and f_2 are identical functions iff $f_1 \circ \psi^*$ and $f_2 \circ \psi^*$ are also identical.

C. Binary Decision Diagrams

A *branching program* (BP) [4] is an acyclic, labelled, directed graph with labelling function $\ell : N \to (V \cup \{1, 0\})$, where N is a set of nodes and V is a set of boolean variables.

A BP has exactly one source node, called the *root*. The labelling function is such that $\ell(\sigma) \in \{1, 0\}$ if and only if σ is a sink; the sinks are called *terminal nodes*. Each non-terminal node σ has exactly two direct successors, called $lo(\sigma)$ and $hi(\sigma)$. Given a V-assignment α and non-terminal node σ, the *active* child of σ is $lo(\sigma)$ if $\alpha(\ell(\sigma)) = 0$ and $hi(\sigma)$ otherwise. A BP represents the boolean function f over V defined so that $f(\alpha)$ is the label of the terminal node found by starting at the root, and following the path of active children according to α. The number of non-terminal nodes in a BP is called its *size*.

We now introduce two sub-classes of BPs that involve placing a total order $\preceq$ on V. A *$\preceq$-ordered binary decision diagram* ($\preceq$-OBDD, or OBDD if $\preceq$ is understood) is a BP such that for all non-terminal nodes σ and σ' where σ' is a successor of σ, we have that $\ell(\sigma) \neq \ell(\sigma')$ and $\ell(\sigma) \preceq \ell(\sigma')$. In other words, all paths from the root to a sink respect $\preceq$. The *sub-OBDD* rooted at a node σ is the OBDD formed by deleting all nodes other than σ and its descendants. We say two BPs are *isomorphic* if they are isomorphic in the traditional graph theoretic sense, and the isomorphism preserves ℓ, lo, and hi. A *reduced OBDD*, which we simply call a *BDD*, is an OBDD such that no two sub-OBDDs are isomorphic, and no node σ has $lo(\sigma) = hi(\sigma)$.

Theorem 1 (Bryant [7])**.** *For any function f and $\preceq$, there exists a $\preceq$-BDD that represents f and it is unique up to isomorphism.*

Thanks to Theorem 1, we can refer to a $\preceq$-BDD that represents f as *the $\preceq$-BDD for f*; we denote the size of this BDD by $\textsc{Sz}(f, \preceq)$. For many functions that arise in practice, the BDD provides a compact representation. Also it is well-known that the choice of $\preceq$ can often make the difference between having an exponentially- or compactly-sized BDD. Furthermore, there are some practical functions that have been proven to have exponentially sized BDDs for *any* choice of $\preceq$. The middle bit of the output of an integer multiplier is a standard such example [8], as too is the so-called *hidden weighted bit function* [5], which we define in Sect. IV-C.

It is well-known that BDDs are the most compact OBDDs:

Lemma 2. *For any function f and $\preceq$, $\textsc{Sz}(f, \preceq)$ is not greater than the size of any $\preceq$-OBDD for f.*

D. Generalized Cofactor

Given a total order $\preceq$ over a set of indexed variables $\{x_1, \ldots, x_n\}$, the *permutation induced by* $\preceq$ is the permutation π on $\{1, \ldots, n\}$ defined by $\pi(i) = j$ iff x_j is the ith element in the order $\preceq$, thus $x_{\pi(1)} \preceq \cdots \preceq x_{\pi(n)}$. Given functions f and h over $\{x_1, \ldots, x_n\}$, and a variable order $\preceq$, the *generalized cofactor* [11] of f and h is the function defined by $\mathit{GCF}(f, h, \preceq)(\alpha_1) = f(\alpha_2)$, where α_2 is the unique assignment such that $h(\alpha_2) = 1$, and the following distance d between α_1 and α_2 is minimized. Here π is the permutation induced by $\preceq$, and $\oplus$ is exclusive-OR.

$$d(\alpha_1, \alpha_2) \;=\; \sum_{i=1}^{n} 2^{n-i}(\alpha_1(x_{\pi(i)}) \oplus \alpha_2(x_{\pi(i)})) \qquad (1)$$

The intuition behind (1) is that differences between α_1 and α_2 are weighted greater for variables that are smaller in $\preceq$. Although d depends on π and thus $\preceq$, we leave this implicit.

There are many papers that employ the generalized cofactor operation, but we could not find an explicit statement of the following lemma (that we use):

Lemma 3. $\textsc{Sz}(\mathit{GCF}(f, h, \preceq), \preceq) \;\leq\; \textsc{Sz}(f, \preceq)\textsc{Sz}(h, \preceq)$

Proof. By inspection of the pseudo-code for $gcf(f, h)$ of Franco and Weaver [12], we see that a new node is generated at most once per recursive call to $gcf(f', h')$. A recursive call is done at most once on each pair of nodes (f', h') where f' is a node of f and h' is a node of h; the result follows. □

As noted by Jones [18], the *Param* operation [3] can be synthesized using *GCF*. We will be effectively using *Param* on several occasions, but will express it using *GCF* and hence not mention *Param* explicitly.

IV. Partitioned BDD and UBFV Size Complexity

A partitioned BDD for a function f is a set of functions that each characterize f in a subspace of assignments, and each subspace can use a different variable order. This freedom can result in smaller BDDs than a "monolithic" BDD representation. In this section we show that if f has a compact representation as a partitioned BDD, then there exists a UBFV ψ for f such that $\psi(v)$ has a compact BDD for each v, and so too does $f \circ \psi^*$. Hence, we needn't exploit disparate variable orderings as afforded by partitioned BDDs; UBFVs allow for compact representations using a single order. Consequently, whereas techniques for using partitioned BDDs in proofs effectively involve doing the proof once for each partition, we need only run the proof once using an UBFV representation.

As just stated, our result is almost obvious: one could create a per-partition copy of each primary variable, and construct the UBFV representation using an order that preserves each partition's ordering on its copy. This would totally eliminate the possibility of inter-partition sharing of sub-BDDs, and provide no interesting advantage over doing per-partition proofs. On the contrary, we use the same set of variables for each partition in the UBFV representation, which allows us to prove our result while only introducing a logarithmic increase in the number of secondary variables. Though this enables our per-partition sub-BDDs to share BDD nodes, our upper-bound result assumes no sharing. However, we show in Sect. V empirically and by example that sharing can have a profound effect; this is also evident in our experimental results of Sect. VII.

A. Partitioned BDDs

We now formalize partitioned BDDs, more or less following Narayan *et al.* [22], with one material difference.[2]

[2]The difference being that [22] requires $f_j = w_j f$ while we use the weaker condition $w_j f_j = w_j f$. We effectively give f_i the freedom to behave arbitrarily in the "don't care" space $\overline{w_i}$. This allows Theorem 2 to potentially yield tighter bounds than if we used the definitions from [22] verbatim.

A *partitioned BDD* for V-function f is a set of triples $\{(w_j, f_j, \preceq_j) : 1 \leq j \leq k\}$ where

1) each w_j and f_j are boolean functions over V such that $w_j f_j = w_j f$ and $w_j \neq \mathbf{0}$
2) each $\preceq_j$ is a V-order
3) $w_1 \vee \cdots \vee w_k = \mathbf{1}$

Typically, partitioned BDDs are of interest when f does not have a sufficiently small BDD representation, but each w_j and f_j do have compact $\preceq_j$-BDDs.

B. UBFV Construction

Suppose we have a partitioned BDD for f as in Sect. IV-A, over the variables $V = \{v_1, \ldots, v_n\}$. Our construction of a corresponding UBFV representation involves secondary variables $C = \{c_{\log(k)}, \ldots, c_0\}$ that, when assigned to, uniquely select a partition. Two key insights come into play:

1) We employ secondary variables $Y = \{y_1, \ldots, y_n\}$ such that, once we have selected partition j, the variable y_i represents the ith element of the V-order $\preceq_j$. In other words, the variable v_q represented by y_i differs according to the selected partition. This trickery allows us to use the *same* order on Y across all partitions and preserve the BDD sizes in the partitioned BDD.
2) Our UBFV is constructed so that once we select partition j, the V-assignment that is induced satisfies w_j. To achieve this we utilize the generalized co-factor operation, explained in Sect. III-D.

Formalizing this, we define the *repartitioned BFV* γ. Let $c[j]$ denote the condition that $c_{\log(k)} \ldots c_0$, as a binary number, is equal to j. Let π_j be the permutation induced by $\preceq_j$, and in a slight abuse, for any V-function h we let $\pi_j(h)$ be the Y-function formed by substituting $y_{\pi_j(i)}$ for v_i, $1 \leq i \leq n$, in h. Letting $\preceq$ be the natural order over Y, γ is the BFV over $(V, C \cup Y)$ defined by

$$\gamma(v_i) \;=\; \bigvee_{j=1}^{k} c[j] \mathit{GCF}(\pi_j(v_i), \pi_j(w_j), \preceq) \qquad (2)$$

The above expression integrates both of our key insights. The use of π_j to turn V-functions v_i and w_j into Y-functions is the manifestation of the first insight, while the use of *GCF* is that of the second. Lemma 4 below, which is proven in the appendix of the web version [1] asserts that the repartitioned BFV γ is in fact universal, and also gives an expression for $f \circ \gamma^*$.

Lemma 4. *γ is a* universal *BFV, and furthermore*

$$f \circ \gamma^* \;=\; \bigvee_{j=1}^{k} c[j] \mathit{GCF}(\pi_j(f_j), \pi_j(w_j), \preceq) \qquad (3)$$

Lemma 4 is instrumental in proving our upper bound result Theorem 2 below. The BDD for $f \circ \gamma^*$ described in the proof is shown in Figure 1.

Theorem 2. *Suppose f is a function over V with a partitioned BDD $\{(w_1, f_1, \preceq_1), \ldots, (w_k, f_k, \preceq_k)\}$. Then there exists a UBFV ψ for f with secondary variables V' and total order $\preceq$ on V' such that*

(a) For all $v \in V$, $\mathrm{Sz}(\psi(v), \preceq) = \mathcal{O}\left(\sum_{j=1}^{k} \mathrm{Sz}(w_j, \preceq_j)\right)$
(b) $\mathrm{Sz}(f \circ \psi^*, \preceq) = \mathcal{O}\left(\sum_{j=1}^{k} \mathrm{Sz}(w_j, \preceq_j)\mathrm{Sz}(f_j, \preceq_j)\right)$
(c) $|V'| = \lceil \log_2 k \rceil + |V|$

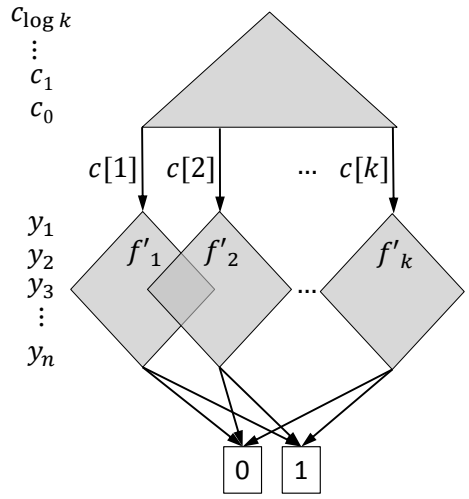

Fig. 1. The OBDD for $f \circ \gamma^*$ described in the proof of Theorem 2. The variable order is shown on the left; the triangular part at the top of the OBDD is the incomplete OBDD that determines which $c[j]$ holds. Each f'_j is the BDD for $\mathit{GCF}(\pi_j(f_j), \pi_j(w_j), \preceq)$, which is the GCF of f_j with respect to w_j, but with variables renamed according to π_j. Thus each f'_j has the exact same structure as the $\preceq_j$-BDD of $\mathit{GCF}(f_j, w_j, \preceq_j)$. The overlap between f'_1 and f'_2 emphasizes that sub-BDD sharing is possible between all the f'_j's.

Proof. (Sketch) We choose the repartitioned BFV γ to serve as the witness. Thanks to Lemma 4, we have that γ is universal; Also condition (c) holds of γ by definition. To prove conditions (a) and (b), let us employ the notion of an *incomplete OBDD* as an OBDD wherein some sink nodes are not terminal nodes, but rather unlabelled *placeholders*; by appropriately plugging in other OBDDs at placeholders, an incomplete OBDD can be turned into an OBDD.

For each v_i, an $\preceq$-OBDD for $\gamma(v_i)$ (2) can be constructed as follows. Start with an incomplete OBDD over variables C with k placeholder nodes, such that the path to the jth placeholder is active when $c[j]$ holds of an assignment. Note that this incomplete OBDD has size $\mathcal{O}(k)$. For each $1 \leq j \leq k$, at the jth placeholder we insert the $\preceq$-BDD of $\mathit{GCF}(\pi_j(v_i), \pi_j(w_j), \preceq)$. From Lemma 3 and the fact that the size of a variable function is 1, the size of this BDD is bounded by $\mathrm{Sz}(\pi_j(w_j), \preceq) = \mathrm{Sz}(w_j, \preceq_j)$; condition (a) follows.

We build an $\preceq$-OBDD for $f \circ \gamma^*$ by starting with the same incomplete OBDD as above. At the jth placeholder, we insert the $\preceq$-BDD of $\mathit{GCF}(\pi_j(f_j), \pi_j(w_j), \preceq)$. Again appealing to Lemma 3 we find that size of this OBDD to be bounded by

$$\mathrm{Sz}(\pi_j(f_j), \preceq)\mathrm{Sz}(\pi_j(w_j), \preceq) = \mathrm{Sz}(f_j, \preceq_j)\mathrm{Sz}(w_j, \preceq_j)$$

□

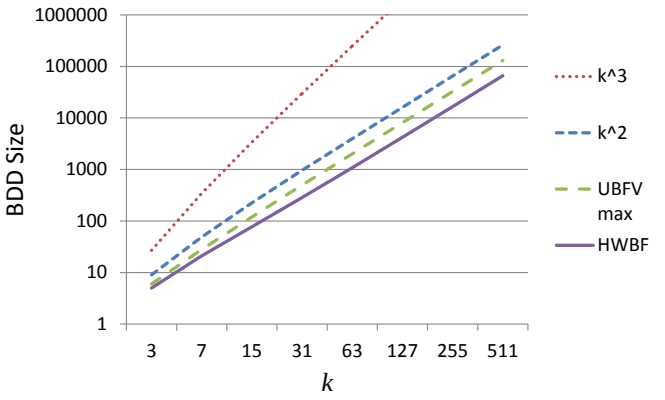

Fig. 2. Computed BDD sizes for the hidden weighted bit function using UBFVs (log-log scale). Data points are for $k = 2^e - 1$ with $2 \leq e \leq 9$. "UBFV max" is the largest BDD in the UBFV ψ, while "HWBF" is the size of $HWB_k \circ \psi^*$. The curves k^3 and k^2 are plotted for comparison, from which it seems evident that the BDDs grow only quadratically, an improvement over our proven cubic bounds.

C. Application to Hidden Weighted Bit Function

As a corollary to Theorem 2, we can prove the existence of a good UBFV representation for the hidden weighted bit function. For any $k \geq 1$, HWB_k is the function over the k variables $\{x_1, \ldots, x_k\}$ defined by $HWB_k = x_w$, where $w = \textsc{Weight}(\{x_1, \ldots, x_k\})$ and $\textsc{Weight}$ counts the number of variables assigned to 1 in its argument. For the case that $w = 0$, we set $HWB_k = 0$.

Corollary 1. *For $k \geq 1$ there exists a UBFV ψ for HWB_k and secondary variable ordering $\preceq$ such that for each $1 \leq i \leq k$, $\textsc{Sz}(\psi(x_i), \preceq) = \mathcal{O}(k^3)$, and $\textsc{Sz}(HWB_k \circ \psi^*, \preceq) = \mathcal{O}(k^3)$.*

Proof. Let $\{(w_0, f_0, \preceq), \ldots, (w_k, f_k, \preceq)\}$ be the partitioned BDD for HWB_k such that

- $w_j = 1$ iff $j = \textsc{Weight}(x_k, \ldots, x_1)$
- $f_0 = \mathbf{0}$ and $f_j = x_j$ for $1 \leq j \leq k$
- $\preceq$ is any ordering.

Since each w_j is a totally symmetric function, $\textsc{Sz}(w_j, \preceq) = \mathcal{O}(k^2)$ [7], and clearly $\textsc{Sz}(f_j, \preceq) = \mathcal{O}(1)$. The result then follows by Theorem 2. □

V. Discussion

In practice, due to the reduction and sharing inherent in BDDs, the bounds afforded by Theorem 2 can be quite loose. For example, empirically we observe quadratically sized UBFV representations for HWB_k, as plotted in Figure V. Here we give a couple other arm-chair constructions that illustrate the versatility afforded by UBFVs.

A. Inverting an Adder

In this section we give an example of a UBFV that does not resemble the repartitioned UBFV of Sect. IV-B and demonstrates how counter-intuitive some UBFV representations can be. Consider an unsigned modulo-2^n adder; in our formalism, this is a list of n functions $\textsc{Add} = \textsc{Add}_{n-1}, \ldots, \textsc{Add}_0$ over $X \cup Y$, where $X = \{x_{n-1}, \ldots, x_0\}$ and $Y = \{y_{n-1}, \ldots, y_0\}$. An assignment α encodes a binary number on X and Y, when we evaluate the functions of $\textsc{Add}$ at α we obtain a binary encoding of $(X+Y) \mod 2^n$. It is well-known that by using a variable ordering that interleaves X and Y, the BDDs of $\textsc{Add}$ are of linear size. However, one can formulate a UBFV representation of $\textsc{Add}$ with *constant* size simply by exploiting the fact that modulo addition is invertible.

Letting $Z = \{z_{n-1}, \ldots, z_0\}$ be fresh variables, we create a BFV ψ over $(X \cup Y, Z \cup Y)$ such that $\psi(y_i) = y_i$ for each $y_i \in Y$, and $\psi(x_i)$ is the function representing the ith bit of the binary number $Z - Y \mod 2^n$. Universality of ψ follows from the fact that given any naturals $x, y < 2^n$, there exists a natural $z < 2^n$ such that $x = z - y \mod 2^n$. More interestingly, consider $\textsc{Add} \circ \psi^*$ (by which we mean ψ applied to each element of $\textsc{Add}$ piecemeal). Now, slightly abusing notation, $\textsc{Add} = X + Y \mod 2^n$, hence

$$\begin{aligned} \textsc{Add} \circ \psi^* &= (X \circ \psi^*) + (Y \circ \psi^*) \mod 2^n \\ &= (Z - Y + Y) \mod 2^n \\ &= Z \end{aligned}$$

Hence the ith bit of $\textsc{Add} \circ \psi^*$ is simply the variable z_i.

This is rather surprising; we claim that a non-trivial arithmetic function is represented simply by a vector of unique variables. But we must keep in mind that $\textsc{Add} \circ \psi^*$ only represents $\textsc{Add}$ when we know what ψ is. And in a sense, we have simply transposed complexity from the outputs of $\textsc{Add}$ to (half of) its inputs.[3]

B. Disguising a Function as a Variable

Sect. V-A showed how in a particular case we can construct a UBFV that reduces a list of functions to a list of unique variables. In general this is not always possible, but when we consider the output of a single function, we have the following result.

Theorem 3. *Let f be a function that is not identically* **1** *or* **0**. *Then there exists a UBFV ψ for f such that $f \circ \psi^*$ is a variable.*

Proof. (Sketch) Let $V = \{v_1, \ldots, v_n\}$ be the variables of f, let $\preceq$ be the natural V-order, and let y be a fresh variable. For all $1 \leq i \leq n$ we define

$$\psi(v_i) \;=\; (y \wedge \mathit{GCF}(v_i, f, \preceq)) \vee \left(\overline{y} \wedge \mathit{GCF}(v_i, \overline{f}, \preceq)\right)$$

It can be seen that ψ is a UBFV, and that $f \circ \psi^* = y$; the condition that f is not **1** or **0** is necessary since the generalized co-factor cannot be taken of **0**. □

Of course Theorem 3 does not directly save us any BDD complexity since the largest BDD in ψ is the same size as the BDD for f; the UBFV ψ simply moves the complexity of the "output" to the "inputs". However, if f is actually an intermediate function involved in symbolically simulating a

[3]The result is not quite so elegant if we use a non-modulo adder, i.e. one with an $(n+1)$th bit of output. In this case we can construct a similar UBFV such that the UBFV representation of bit i of the output will again be z_i, except when $i = n$, in which case it is a nontrivial function.

specification and/or implementation, and having a nontrivial BDD on f leads to downstream blow-up, it an UBFV along the lines of Theorem 3 could be a remedy.

VI. Checking BFV Universality

So far we have ignored a crucial question: given a BFV ψ, how does one verify that it is a *universal* BFV? This is an important problem to solve; we propose to allow users to concoct intricate, problem-specific UBFVs, and soundness of any proof involving UBFVs hinges on them being universal, hence we require a way to certify BFV universality. This issue is not merely theoretical — when experimenting with UBFVs for the case studies of this paper, on more than one occasion the author ended up with a BFV that subtly failed to be universal. In this section we give several solutions to this problem, varying in automation.

A. Algorithmic Approach

Let us denote the elements of V and V' respectively by $\{v_1, \ldots, v_n\}$ and $\{u_1, \ldots, u_m\}$, and for this section we will consider V and V' to be disjoint. The V-function that is the characteristic of ψ can be expressed as:

$$\exists u_1, \ldots, u_m. \bigwedge_{i=1}^{n} (v_i \leftrightarrow \psi(v_i)) \tag{4}$$

This is simply a logical expression of the set represented by a BFV from Goel and Bryant's paper [13]. It follows that ψ is a UBFV iff (4) is the function $\mathbf{1}$. This can be checked using BDD techniques, although we have found that often computing the BDD for the n-way conjunction or the subsequent existential quantification caused BDD blow-up.

B. Semi-automatic Inverse Approach

A non-automatic, but computationally less demanding approach to checking universality can use BDDs or SAT-solving as a propositional engine. The user provides a proof of universality; the engine then checks the proof via computations that are much simpler than a direct computation of (4). The proof takes the form of an inverse $\psi^{-1} : V' \to ((V \to \mathbb{B}) \to \mathbb{B})$ of ψ. We say "an inverse" since ψ^{-1} is usually not unique. The idea is that for each $v' \in V'$ and V-assignment α, the V'-assignment α' defined by $\alpha'(v') = \psi^{-1}(v')(\alpha)$ is such that $\alpha = \psi^*(\alpha')$. In this way, ψ^{-1} maps α to a "witness" α' such that $\alpha = \psi^*(\alpha')$, the existence of such a witness for each α is tantamount to universality of ψ.

Expounding on this, we employ ψ^{-1} to simplify (4) by using it to pick values for the existentially quantified variables of V', which allows us to strip off the quantifier:

$$\bigwedge_{i=1}^{n} \left(v_i \leftrightarrow (\psi(v_i) \circ \psi^{-1^*}) \right) \tag{5}$$

Since we aspire to prove that (5) is tautological, which is the case iff each conjunct is tautological, this reduces to checking tautology of each $v_i \leftrightarrow \psi(v_i) \circ \psi^{-1^*}$ individually.

Recollecting the example in the introduction of this paper, suppose we have $V = \{v_1, v_2, v_3\}$, $V' = \{a, b, c, d\}$, and $\psi = \{v_1 \mapsto a \vee b, v_2 \mapsto d, v_3 \mapsto \overline{b}\overline{c}\}$. Then a suitable inverse is $\psi^{-1} = \{a \mapsto v_1, b \mapsto 0, c \mapsto \overline{v_3}, d \mapsto v_2\}$; the reader can check that (5) holds.

instruction	case-splits		runtime		memory	
	classic	UBFV	classic	UBFV	classic	UBFV
SP FADD	113	1	22.1	0.2	2.5	2.4
DP FADD	231	1	51.5	0.5	2.6	6.7
SP FMA	173	5	40.1	2.6	12.8	11.7
DP FMA	2374	28	497.6	47.4	10.3	37.5
PCMPISTRI	82	1	0.7	1.3	2.0	17.1
SP FDIV Pre	335	16	17.2	0.9	4.1	4.1

TABLE I
Results comparing our UBFV approach to the classical approach. Times are in hours and memory usage is the maximum number of GB used by any case-split.

VII. Case Studies

In this section we report on application of the UBFV approach to the problem of verifying hardware implementations of several instructions in recent CPU designs done at Intel. All examples use symbolic simulation with BDDs, specifically using the rSTE symbolic simulator [23] and the underlying forte [25] tool. We contrast our results against the "classic" proofs, which use the same tool suite and were done by Intel FV experts other than the author.[4] The results are summarized in Table VII; note that all results involving a case-split include the time taken to prove that the cases are exhaustive. Note that the UBFVs for these case studies were developed manually prior to the theory of Sect. IV, and thus do not explicitly use the repartitioned BFV construction of that section.

A. Floating Point Addition

Floating Point Addition (FADD) is a family of instructions that perform addition of FP numbers. BDDs for the outputs of *integer* addition are linear in the bit-width of the operands, using the variable order that simply interleaves the operand variables. For FADD, however, the BDDs are exponential. This is because the input mantissas must be aligned, according to the exponent difference *expdiff*, prior to performing the (integral) addition [10], [3], which means there is no good ordering that covers all values of *expdiff* at once. The current state of the art thus involves case-splitting based on *expdiff*. For double precision (DP), there are roughly 2^{12} possible *expdiff*'s, however it is common practice to reduce this significantly by bucketing near-by and extremal *expdiff*'s into the same case. For example, the classic DP proof ran 4 consecutive *expdiff*'s per case, yielding 231 cases total.

We constructed a UBFV for FADD where the mantissa of the second operand m is effectively symbolically shifted by the exponent difference, except in the opposite direction as the FADD alignment. This UBFV pre-shifting has the result (in both the hardware and the specification code) that after

[4]It should be noted that the classic proofs might not be as optimized with regard to case-splitting as possible — as the work was done under project schedule constraints, when the verification engineer achieves a reasonable proof configuration, he or she moves on to other work. Nevertheless, they provide a meaningful benchmark against which to compare our approach.

alignment, the i bit of m collapses to a relatively simple BDD involving just a variable m_i and variables from the exponents. We can think of m_i as representing the bit of m that is weighted the same as the ith bit of the first operand, *after alignment*. Thus, when the symbolic addition is performed and we use an interleaving variable order, the exponential blow-up one faces in the classic proof (without case-splitting) is avoided. Table VII shows the incredible impact this has; for double precision, the 231-way case-split that takes 38 hours is reduced to a single case that takes but half an hour.

B. Fused Multiply-Add

Fused Multiply-Add (FMA) is a three operand FP instruction that computes $x+yz$ in one fell-swoop, incurring only a single rounding error. FVing an FMA design with symbolic simulation requires both the decomposition involving in verifying a multiplier [20], and the wide case-split for FADD discussed above. Our FMA proofs generally follow the decomposition described by Slobodová [27], the final stage of which uses a cut-point at the product p of the mantissas of y and z, and proves that p is added to x and rounded correctly, given the exponents and signs of x, y, and z. This looks more or less like an FADD, with the exception that p is roughly twice as wide as the input mantissa width, which increases BDD complexity and doubles the number of (non-extremal) $expdiff$'s. Thus the case-splitting is more extensive than for FADD.

An aspect of our FMA hardware that proved to be a challenge is the support for *denormal* FP inputs. As a result, the product p can too be denormal in the sense that its leading one can be in any position. We have yet to pin down why this constituted a challenge for our UBVF approach, especially since our FADD examples also supported denormals but they weren't problematic. As a result we needed to employ some case-splitting on top of the UBVF; 5-way for SP and 28-way for DP. This case-splitting involved the conditions $expdiff < -1$, $expdiff \in \{-1, 0\}$, $0 < expdiff$, as well as additional splitting based on the position of the leading one when p is denormal. Nevertheless, we find our results extremely encouraging — even with dozens of machines on which to concurrently run these cases, the classic DP FMA proof still took several days to run, and would often fail due to a machine going down or infrastructure issues. Reducing to under 2 days of compute time is a huge win; the 28 cases with 12 concurrent worker threads only took 6.5 hours of real time. Also, further splitting could reduce the memory footprint significantly; although the maximum memory was 37.5 GB, the average across all 28 cases was only 17.3 GB[5]

Apart from the using the $expdiff$ pre-shifting discussed in Sect. VII-A, our FMA study employed a second trick afforded by the UBFV framework. The mantissa product p, from the specification's point of view, is a (binary encoding of) a single positive integer. However, this number is never calculated explicitly in most hardware designs; but rather appears as a *sum/carry pair* of values. A verification engineer constructs a mapping that appropriately sums these two vectors; the result of which serves as the p input to the specification. Using a trick very similar to that of Sect. V-A, our UBFV was set up so after summing, p collapses to more or less a vector of unique variables, hence simplifying the downstream computations in the specification and the design as well.

C. SSE4 String Instruction

Our third case study is an example string processing instruction from Intel's SSE4 instruction set [2] called PCMPISTRI. Logically[6], this instruction takes two arrays $s1$ and $s2$, each having 8 entries, and each entry being a 16-bit word, and returns $ind \in \{0, \ldots, 8\}$. Let $len1$ (resp. $len2$) be the smallest $i < 8$ such that the $s1[i] = 0$ (resp. $s2[i] = 0$), or 8 if no such i exists. Then the returned value ind is the smallest such that $ind < len1$ and $s1[ind] = s2[j]$ for some $0 \leq j < len2$, or $ind = 8$ if no such ind exists.

The classic proof we looked at involves a decomposition point; there is an internal 8-bit vector $IntRes1$ that is calculated such that $IntRes1[i] = 1$ iff $s1[i] = s2[j]$ for some $0 \leq j < len2$ and $i \leq len1$. Once $IntRes1$ is computed, the return value ind is simply the index of the lowest set bit of $IntRes1$, or $ind = 8$ if it is all 0s. The first stage of the decomposition uses an 81-way case-split, according to the possible values of $(len1, len2)$. Referring to Table VII, we note that although the classic proof was a bit faster and used significantly less memory, we still see this result as a "win" for the UBFV approach; we have eliminated the need to decompose the proof, which introduces lots of human effort into the proof (decomposing the specification, mapping to the cut-point in the RTL, etc). Finally, to show that the UBFV win was not simply an artifact of it using more memory, we ran the class proof for the case $(len1, len2) = (8, 8)$ but *without* decomposition – the memory footprint grew to 100 GB after 20 hours of runtime and we killed the process.

D. Floating Point Division

FP Division (FDIV) proofs are highly decomposed [19]. One part of this decomposition involves proving that a pre-processing step (Pre), prior to the main iterative algorithm, is correct. The Pre proof originally required holding 4 bits of the mantissa to constants, yielding 16 cases. In a subsequent chip, support for denormal inputs was added. To perform the analogous case-split for denormals, a second level of case-splitting based on the position of the leading one in the denormal mantissa was needed, resulting in roughly 22×16 additional cases. We employed a pre-shifted UBFV that allowed us to handle both normal and denormal inputs using only the original 16 cases. This pre-shifting was based on a vector of secondary variables that point to the leading-one position.

We originally deemed UBFV to be overkill for this problem since one can play pre-shifting trickery when crafting the

[5]For the classic FMA approach, we were unable to compile definitive data on maximum memory usage; the given numbers are the memory footprint of the case-split that involved the most BDD nodes, which isn't necessarily the one that used the most memory. Hence they are lower bounds.

[6]Actually what we describe here is how the instruction behaves when the immediate bits are set appropriately.

case-split. However, doing this lead to non-trivial BDDs on the *non-constant* mantissa bits, which incurred BDD blow up. Using our UBFV, these BDDs involve only a single mantissa variable, which makes the BDDs behave very similarly to the purely normal cases. Finally we mention that the same UBFV approach also had a dramatic impact on double precision FDIV Pre, however we were not able to compile data for Table VII in time for this submission.

VIII. Conclusions and Future Work

We have proposed Universal Functional Boolean Vectors as a means of alleviating BDD complexity and demonstrated a profound impact of this approach on difficult hardware datapath FV problems. Though concocting a good UBFV requires human insight, like a BDD variable order, the recipe is often *specification* specific (rather than *implementation* specific) — thus the effort is amortized over many generations of hardware designs. Here we now ponder directions for future work.

One of our case studies showed that it is possible to use UBFVs to make it unnecessary to decompose a proof, for a rather esoteric string processing instruction. But what about classically complex arithmetic functions, for instance multiplication? We conjecture that there does not exist a polynomial-sized UBFV that would elicit a polynomial-sized representation for the list of functions that define the output of integer multiplication, but can this be proven?

The focus here has been on BDD-representations. The other propositional reasoning workhorse is the SAT solver – can UBFV representations be employed to alleviate time complexity in this domain? Are there algorithms and heuristics for generating good UBFVs? Finally, are their applications in other domains, for instance fix-point BDD model checking?

Acknowledgement

Thanks to Sava Krstić and the anonymous reviewers for their suggestions, and to John Franco and Sean Weaver for answering some of our questions.

References

[1] Web version of this paper. http://www.cs.ubc.ca/~jbingham/fmcad2015.pdf.

[2] *Intel SSE4 Programming Reference*. Intel Corporation, 2007.

[3] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Formal verification using parametric representations of Boolean constraints. In *Design Automation Conference (DAC 1999)*, July 1999.

[4] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, 1978.

[5] B. Bollig, M. Lobbing, M. Sauerhoff, and I. Wegener. On the complexity of the hidden weighted bit function for various BDD models. *Theoretical Informatics and Applications*, 33:33–103, 1998.

[6] F. M. Brown. Reduced solutions of boolean equations. *IEEE Trans. Comput.*, 19(10):976–981, Oct. 1970.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677–691, 1986.

[8] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. Comput.*, 40(2):205–213, 1991.

[9] E. Cerny and M. A. Marin. A computer algorithm for the synthesis of memoryless logic circuits. *Computers, IEEE Transactions on*, C-23(5):455–465, May 1974.

[10] Y.-A. Chen and R. Bryant. Verification of floating-point adders. In *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 488–499. 1998.

[11] O. Coudert and J. Madre. A unified framework for the formal verification of sequential circuits. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 126–129, Nov 1990.

[12] J. Franco and S. Weaver. *Handbook of Combinatorial Optimization*, chapter Algorithms for the Satisfiability Problem. Springer, New York, 2013.

[13] A. Goel and R. E. Bryant. Set manipulation with boolean functional vectors for symbolic reachability analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, 2003.

[14] A. J. Hu. Formal hardware verification with BDDs: An introduction. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*, pages 677–682, 1997.

[15] S. Huddleston, R. Kaivola, and C. Seger. personal communication, 2015.

[16] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner. Automatic formal verification of fused-multiply-add fpus. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 1298–1303, 2005.

[17] P. Jain and G. Gopalakrishnan. Efficient symbolic simulation-based verification using the parametric form of boolean expressions. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 13(8):1005–1015, 2006.

[18] R. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Springer US, 2002.

[19] R. Kaivola and K. R. Kohatsu. Proof engineering in the large: formal verification of pentium?4 floating-point divider. *Software Tools for Technology Transfer*, 4(3):323–334, 2003.

[20] R. Kaivola and N. Narasimhan. Formal verification of the pentium 4 floating-point multiplier. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '02, pages 20–. IEEE Computer Society, 2002.

[21] V. KiranKumar, A. Gupta, and R. Ghughal. Symbolic trajectory evaluation: The primary validation vehicle for next generation Intel® processor graphics fpu. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 149–156, Oct 2012.

[22] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned ROBDDs-a compact, canonical and efficiently manipulable representation for boolean functions. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pages 547–554, Nov 1996.

[23] J. O'Leary, R. Kaivola, and T. Melham. Relational STE and theorem proving for formal verification of industrial circuit designs. In B. Jobstmann and S. Ray, editors, *Formal Methods in Computer-Aided Design (FMCAD 2013)*, pages 97–104. IEEE, Oct. 2013.

[24] V. Paruthi, C. Jacobi, and K. Weber. Efficient symbolic simulation via dynamic scheduling, don't caring, and case splitting. In *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, pages 114–128, 2005.

[25] C. J. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 24(9):1381–1405, 2006.

[26] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.

[27] A. Slobodová. Challenges for formal verification in industrial setting. In *FMICS/PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2006.

[28] A. Slobodová, J. Davis, S. Swords, and W. A. Hunt. A flexible formal verification framework for industrial scale validation. In S. Singh, B. Jobstmann, M. Kishinevsky, and J. Brandt, editors, *MEMOCODE*, pages 89–97. IEEE, 2011.

[29] J. Xu, M. Williams, H. Mony, and J. Baumgartner. Enhanced reachability analysis via automated dynamic netlist-based hint generation. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 157–164, Oct 2012.