Fast Heuristic and Exact Algorithms for Two-Level Hazard-Free Logic Minimization*

Michael Theobald Steven M. Nowick

Department of Computer Science

Columbia University

New York, NY 10027

CUCS-001-98

Abstract

None of the available minimizers for 2-level hazard-free logic minimization can synthesize very large circuits. This limitation has forced researchers to resort to manual and automated circuit partitioning techniques. This paper introduces two new 2-level logic minimizers: Espresso-HF*, a heuristic method which is loosely based on* Espresso-II*, and* Impymin*, an exact method based on implicit data structures.*

Both minimizers can solve all currently available examples, which range up to 32 inputs and 33 outputs. These include examples that have never been solved before. For examples that can be solved by other minimizers our methods are several orders of magnitude faster.

As by-products of these algorithms, we also present two additional results. First, we introduce a fast new algorithm to check if a hazard-free covering problem can feasibly be solved. Second, we introduce a novel formulation of the 2-level hazard-free logic minimization problem by capturing hazard-freedom constraints within a synchronous function by adding new variables.

* This work was supported by NSF under Grant no. MIP-9501880 and by an Alfred P. Sloan Research Fellowship. The presented work is an extended version of two recent conference papers [32, 31].

1 Introduction

Asynchronous design has been the focus of much recent research activity. In fact, asynchronous designs have been applied to several large-scale control- and datapath circuits and processors [11, 18, 12, 19, 2, 30, 34, 15, 1].

A number of methods have been developed for the design of hazard-free controllers [22, 20, 37, 13, 27]. These methods have been applied to several large and realistic design examples, including a low-power infrared communications chip [14], a second-level cache-controller [21], a SCSI controller [35], a differential equation solver [36], and an instruction length decoder [4].

An important aspect of these methods is the development of optimized CAD tools. In synchronous design, CAD packages have been critical to the advancement of modern digital design. In asynchronous design, much progress has been made, including tools for exact hazard-free two-level logic minimization [25], optimal state assignment [10, 27] and synthesis-for-testability [24]. However, these tools have been limited in handling large-scale designs.

In particular, hazard-free 2-level logic minimization is an important step in all the above-mentioned CAD tools. However, while the currently used Quine-McCluskey-like exact hazard-free minimization algorithm, HFMIN [10], has been effective on small- and medium-sized examples, it has been unable to produce solutions for several large design problems [13, 27]. This limitation has been a major reason for researchers to invent and apply manual as well as automated techniques for partitioning circuits before hazard-free logic minimization can be performed [13].

Contributions of This Paper

This paper introduces two new and very efficient 2-level hazard-free logic minimizers for multi-output minimization: ESPRESSO-HF and IMPYMIN.

ESPRESSO-HF is an algorithm to solve the heuristic hazard-free two-level logic minimization problem. The method is heuristic solely in terms of the cardinality of solution. In all cases, it guarantees a hazard-free solution. The algorithm is based on ESPRESSO-II[26, 9], but with a number of significant modifications to handle hazard-freedom constraints. It is the first heuristic method based on ESPRESSO-II to solve the hazard-free minimization problem. Espresso-HF also includes a new and much more efficient algorithm to check for existence of a hazard-free solution, without generating all prime implicants.

Impymin is an algorithm to solve the exact hazard-free two-level logic minimization problem. The algorithm uses an implicit approach which makes use of data structures such as BDDs [3] and zero-suppressed BDDs [17]. The algorithm is based on a novel theoretical approach to hazard-free two-level logic minimization. We reformulate the generation of dynamic-hazard-free prime implicants as a *synchronous* prime implicant generation problem. This is achieved by incorporating hazard-freedom constraints within a synchronous function by adding new variables. This technique allows to leverage off an existing method for fast implicit generation of prime implicants. Moreover, our novel approach can be nicely incorporated into a very efficient implicit minimizer for hazard-free logic. In particular, the approach makes it possible to use the implicit set covering solver of Scherzo [8, 6, 5, 7], the state-of-the-art minimization method for synchronous two-level logic, as a black box.

Both Espresso-HF and Impymin can solve all currently available examples, which range up to 32 inputs and 33 outputs. These include examples that have never been previously solved. For examples that can be solved by the currently fastest minimizer Hfmin our two minimizers are typically several orders of magnitude faster. In particular, Impymin can find a minimum-size cover for all benchmark examples in less than 813 seconds, and Espresso-HF can find very good covers – at most 3% larger than a minimum-size cover – in less than 105 seconds.

Espresso-HF and Impymin are somewhat orthogonal. On the one hand Espresso-HF is typically faster than Impymin. On the other hand, Impymin computes a cover of minimum size, whereas Espresso-HF is not guaranteed to find a minimum cover but typically does find a cover of very good quality.

Paper Organization

The paper is organized as follows. Section 2 gives background on circuit models, hazards and hazard-free minimization. Section 3 describes the Espresso-HF algorithm for heuristic hazard-free minimization. Section 4 introduces a new approach to hazard-free minimization where hazard-freedom constraints are captured by a constructed synchronous function, leading to a new method for computing dynamic-hazard-free prime implicants. Based on the results of Section 4, Section 5 introduces our new implicit method for exact hazard-free minimization, called IMPYMIN. Section 6 presents experimental results and compares our approaches with related work, and Section 7 gives conclusions. Background information on BDD, ZBDDs, and implicit logic minimization can be found in the appendix.

2 Background

The material of this section focuses on hazards and hazard-free logic minimization, and is taken from [10] and [25, 23]. For simplicity, we focus on single-output functions. A generalization of these definitions to multi-output functions is straightforward, and is described in [10].

2.1 Circuit Model

This paper considers combinational circuits having arbitrary finite gate and wire delays (an *unbounded wire delay model* [25]). A pure delay model is assumed as well (see [33]).

2.2 Multiple-Input Changes

Definition 2.1 *Let A and B be two minterms. The* **transition cube**, $[A, B]$, *from A to B has* **start point** *A and* **end point** *B, and contains all minterms that can be reached during a transition from A to B. More formally, if A and B are described by products, with i-th literals A_i and B_i, respectively, then the i-th literal for the product of $t = [A, B]$ is the Boolean function $A_i + B_i$ (alternatively, $[A, B]$ is the uniquely defined smallest cube that contains A and B:* supercube(A,B)*). An* **input transition** *or* **multiple-input change** *from input state (minterm) A to B is described by transition cube $[A, B]$.*

A multiple-input change specifies what variables change value and what the corresponding *starting* and *ending* values are. Input variables are assumed to change simultaneously. (Equivalently, since inputs may be skewed arbitrarily by wire delays, inputs can be assumed to change monotonically in any order and at any time.) Once a multiple-input change occurs, no further input changes may occur until the circuit has stabilized. In this paper, we consider only transitions where f is fully defined; that is, for every $X \in [A, B]$, $f(X) \in \{0, 1\}$.

2.3 Function Hazards

A function f which does not change monotonically during an input transition is said to have a *function hazard* in the transition.

Definition 2.2 *A function f contains a* **static function hazard** *for the input transition from A to C if and only if: (1) $f(A) = f(C)$, and (2) there exists some input state $B \in [A, C]$ such that $f(A) \neq f(B)$.*

Definition 2.3 *A function f contains a* **dynamic function hazard** *for the input transition from A to D if and only if: (1) $f(A) \neq f(D)$; and (2) there exist a pair of input states, B and C, such that (a) $B \in [A, D]$ and $C \in [B, D]$, and (b)$f(B) = f(D)$ and $f(A) = f(C)$.*

If a transition has a function hazard, no implementation of the function is guaranteed to avoid a glitch during the transition, assuming arbitrary gate and wire delays [25, 33]. Therefore, we consider only transitions which are free of function hazards [1].

2.4 Logic Hazards

If f is free of function hazards for a transition from input A to B, an implementation may still have hazards due to possible delays in the logic realization.

Definition 2.4 *A circuit implementing function f contains a* **static (dynamic) logic hazard** *for the input transition from minterm A to minterm B if and only if: (1) $f(A) = f(B)$ ($f(A) \neq f(B)$), and (2) for some assignment of delays to gates and wires, the circuit's output is not monotonic during the transition interval.*

That is, a static logic hazard occurs if $f(A) = f(B) = 1$ (0), but the circuit's output makes an unexpected $1 \to 0 \to 1$ ($0 \to 1 \to 0$) transition. A dynamic logic hazard

[1] Sequential synthesis methods, which use hazard-free minimization as a substep, typically include constraints in their algorithms such that no transitions with function hazards are generated [22, 37].

occurs if $f(A) = 1$ and $f(B) = 0$ ($f(A) = 0$ and $f(B) = 1$), but the circuit's output makes an unexpected $1 \to 0 \to 1 \to 0$ ($0 \to 1 \to 0 \to 1$) transition.

2.5 Conditions for a Hazard-Free Transition

We now review conditions to ensure that a sum-of-products implementation, F, is hazard-free for a given input transition (for details, see [25]). Assume that $[A, B]$ is the transition cube corresponding to a *function-hazard-free* transition from input state A to B for a function f. We say that f has a $f(A) \to f(B)$ transition in cube $[A, B]$.

Lemma 2.5 *If f has a $0 \to 0$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B.*

Lemma 2.6 *If f has a $1 \to 1$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B* if and only if *$[A, B]$ is contained in some cube of cover F (i.e.,* some *product must hold its value at 1 throughout the transition).*

The conditions for the $0 \to 1$ and $1 \to 0$ cases are symmetric. Without loss of generality, we consider only a $1 \to 0$ transition [2].

Lemma 2.7 *If f has a $1 \to 0$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B* if and only if *every cube $c \in F$ intersecting $[A, B]$ also contains A (i.e., no product may glitch* in the middle *of a $1 \to 0$ transition).*

Lemma 2.8 *If f has a $1 \to 0$ transition from input state A to B which is hazard-free in the implementation, then, for every input state $X \in [A, B]$ where $f(X) = 1$, the transition subcube $[A, X]$ is contained in some cube of cover F (i.e., every $1 \to 1$ sub-transition must be free of logic hazards).*

$1 \to 1$ transitions and $0 \to 0$ transitions are called *static* transitions. $1 \to 0$ transitions and $0 \to 1$ transitions are called *dynamic* transitions.

[2] A $0 \to 1$ transition from A to B has the same hazards as a $1 \to 0$ transition from B to A.

2.6 Required and Privileged Cubes

The cube $[A, B]$ in Lemma 2.6 and the *maximal* subcubes $[A, X]$ in Lemma 2.8 are called *required cubes.* Each required cube *must* be contained in some cube of cover F to ensure a hazard-free implementation. More formally:

Definition 2.9 *Given a function f, and a set, T, of specified function-hazard-free input transitions of f, every cube $[A, B] \in T$ corresponding to a $1 \to 1$ transition, and every maximal subcube $[A, X] \subset [A, B]$ where f is 1 and $[A, B] \in T$ is a $1 \to 0$ transition, is called a* **required cube**.

Lemma 2.7 constrains the products which may be included in a cover F. Each $1 \to 0$ transition cube is called a *privileged cube*, since no product c in the cover may intersect it unless c also contains its *start point.* If a product intersects a privileged cube but does not contain its start point, it *illegally intersects* the privileged cube and may not be included in the cover. More formally:

Definition 2.10 *Given a function f, and a set, T, of specified function-hazard-free input transitions of f, every cube $[A, B] \in T$ corresponding to a $1 \to 0$ transition is called a* **privileged cube**.

Finally, we define a useful special case. For certain privileged cubes the function is only 1 at the start point and is 0 for all other minterms included in the transition cube. In this case, any product that intersects such a privileged cube always covers the start point, since the cube contains no other ON-set minterms. We call such a privileged cube **trivial**. All trivial privileged cubes can safely be removed from consideration without loss of information.

2.7 Hazard-Free Covers

A *hazard-free cover* of function f is a cover (*i.e.*, set of implicants) of f whose AND-OR implementation is hazard-free for a *given set*, T, of specified input transitions. (It is assumed below that the function is defined for all specified transitions; the function is undefined for all other input states.)

Theorem 2.11 (Hazard-Free Covering [23, 25]) *A sum-of-products F is a hazard-free cover for function f for the set T of specified input transitions if and only if:*

(a.) No product of F intersects the OFF-set of f;

(b.) Each required cube *of f is contained in some product of F; and*

(c.) No product of F intersects any (non-trivial) privileged cube *illegally.*

Theorem 2.11(a) and (c) determine the implicants which may appear in a hazard-free cover of a function f, called *dynamic-hazard-free (dhf-) implicants.*

Definition 2.12 *A* **dhf-implicant** *is an implicant which does not intersect any privileged cube of f illegally. A* **dhf-prime implicant** *is a dhf-implicant contained in no other dhf-implicant. An* **essential dhf-prime implicant** *is a dhf-prime implicant which contains a required cube contained in no other dhf-prime implicant.*

Theorem 2.11(b) defines the covering requirement for a hazard-free cover of f: *every required cube of f must be covered*, that is, contained in some cube of the cover. Thus, the **two-level hazard-free logic minimization problem** is *to find a minimum cost cover of a function using only dhf-prime implicants where every required cube is covered.*

In general, the covering conditions of Theorem 2.11 may not be satisfiable for an arbitrary Boolean function and set of transitions [33, 25]. This case occurs if conditions (b) and (c) cannot be satisfied simultaneously.

A hazard-free minimization example is shown in Figure 1. There are four specified transitions. Transition t_1 is a $1 \to 1$ transition. It gives rise to one required cube (see part (a)). Transition t_2 is a $0 \to 0$ transition. Thus it gives rise neither to required cubes nor privileged cubes. Transition t_3 is a $1 \to 0$ transition. It gives rise to two required cubes (see (a)) and one privileged cube (see (b)). Transition t_4 is also a $1 \to 0$ transition, and gives rise to three required cubes and one privileged cube. A minimum hazard-free cover is shown in part (c). It is apparent that all required cubes are covered, and that no product in the cover illegally intersects any privileged cube. In contrast, the cover in part (d) is not hazard-free since *priv-cube-1* is intersected illegally (shaded region) by product *bd*. In particular, this product may lead to a glitch during transition t_3.

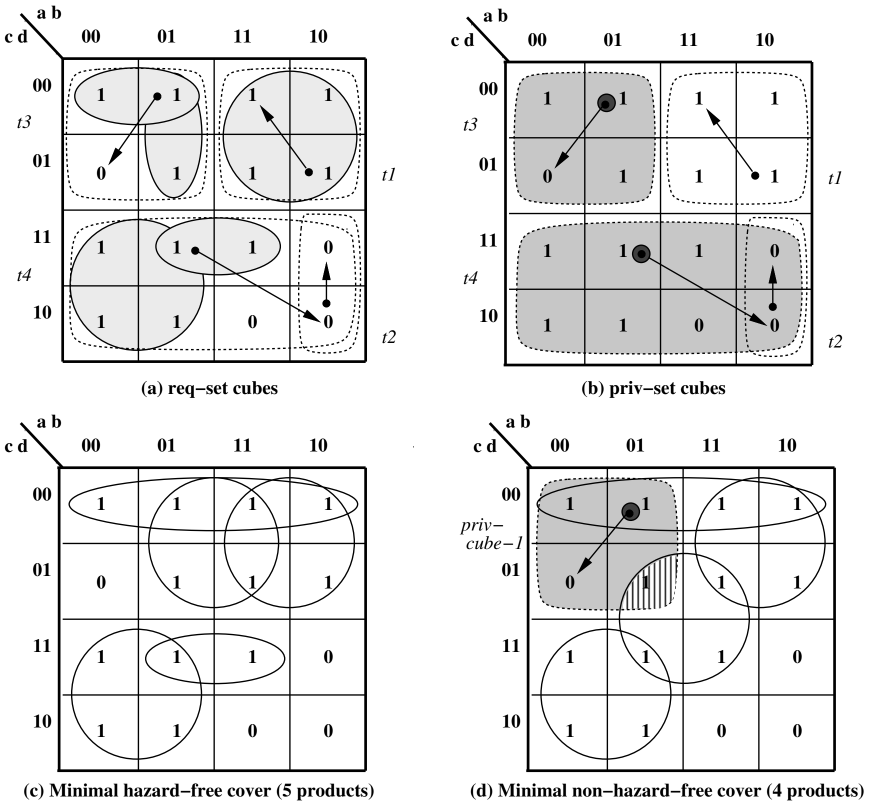

Figure 1: Two-Level Hazard-Free Minimization Example: (a) shows the set of required cubes (shaded); (b) shows the set of privileged cubes (shaded); (c) shows a minimal hazard-free cover; (d) shows a minimum-cost cover that is not hazard-free, since it contains a logic hazard.

2.8 Exact Hazard-Free Minimization Algorithm

A single-output exact hazard-free minimizer has been developed by Nowick and Dill [23, 25]. It has recently been extended to hazard-free multi-valued minimization [3] by Fuhrer, Lin and Nowick [10]. The latter method, called HFMIN, has been the fastest minimizer for exact hazard-free minimization.

HFMIN makes use of ESPRESSO-II to generate all prime implicants, then transforms them into dhf-prime implicants, and finally employs ESPRESSO-II's MINCOV to solve the resulting unate covering problem. Each of the algorithms used in the above three steps is critical, i.e. has a worst-case run-time that is exponential. As a result, HFMIN *cannot* solve several of the more difficult examples.

3 Heuristic Hazard-Free Minimization: ESPRESSO-HF

3.1 Overview

The goal of heuristic hazard-free minimization is to find a very good (but not necessarily exactly minimum) solution to the hazard-free covering problem. The basic minimization strategy of ESPRESSO-HF for hazard-free minimization is similar to the one used by ESPRESSO-II. However, we use additional constraints to ensure that the resulting cover is hazard-free, and the algorithms are significantly different.

One key distinction is in the use of the *unate recursive paradigm* in ESPRESSO-II, i.e. to decompose operations recursively leading to efficiently solvable sub-operations on unate functions. To the best knowledge of the authors, the unate recursive paradigm cannot be applied directly to hazard-free minimization. We therefore follow the basic steps of ESPRESSO-II, modified to incorporate hazard-freedom constraints, but without the use of unate recursive algorithms. However, because of the constraints and granularity of the hazard-free minimization problem, high-quality results are still obtained even for large examples.

[3] It is well-known that multi-output minimization can be regarded as a special case of multi-valued minimization [26].

In this subsection, we describe the basic steps of the algorithm, concentrating on the new constraints that must be incorporated to guarantee a cover to be hazard-free. We then describe the individual steps in detail, in later subsections.

As in ESPRESSO-II, the size of the cover is never increased in size. In addition, after an initial phase, the cover always represents a valid solution, i.e. a cover of f that is also hazard-free. Pseudocode for the algorithm is shown in Figure 2.

The first step of ESPRESSO-HF is to read in PLA files specifying a Boolean function, f, and a set of specified function-hazard-free transitions, T. These inputs are used to generate the set of required cubes Q, the set of privileged cubes P and their corresponding start points S, and the OFF-set R. Generation of these sets is immediate from the earlier lemmas (see also [25]) [4].

The set Q can be regarded both as an initial cover F of the function, and as a set of objects to be covered. Unlike ESPRESSO-II, however, the given initial cover Q does *not* in general represent a valid solution: while Q is a cover of f, it is not necessarily hazard-free. Therefore, processing begins by first expanding each required cube into the *uniquely defined* minimum dhf-implicant covering it. The result is an initial hazard-free cover, F, and set of objects to be covered, Q^f.

The next step is to identify *essential dhf-implicants*, using a modified EXPAND step. This algorithm uses a novel approach to identifying *equivalence classes* of implicants, each of which is treated as a single implicant. Essential implicants, as well as all required cubes covered by them, are then removed from F and Q^f, respectively, resulting in a smaller problem to be solved by the main loop. Before the main loop, the current cover is also made irredundant.

Next, as in ESPRESSO-II, ESPRESSO-HF applies the three operators REDUCE, EXPAND, and IRREDUNDANT to the current cover until no further improvement in the size of the cover is possible. Since the result may be a local minimum, the operator LAST_GASP is then applied to find a better solution using a different method. EXPAND uses new hazard-free notions of *essential parts* and *feasible expansion*. The other steps differ from ESPRESSO-II as well.

At the end, there is an additional step to make the resulting implicants dhf-prime,

[4] The algorithm does not need an explicit cover for the don't-care set because the operations only require the OFF-set to check if a cube is valid.

```
Espresso-HF(f,T)
  Q = generate_set_of_required-cubes(f,T)
  P = generate_set_of_privileged-cubes(f,T)
  S = generate_set_of_start-points(f,T)
  R = OFF-set(f)
  Q^f = {supercube_dhf(q) | q ∈ Q}
  If "undefined" ∈ Q^f then no solution is possible; exit
  Minimize Q^f with respect to single cube containment
  F = Q^f
  (F, E) = expand_and_compute_essentials(F)
  Remove all cubes from Q^f that are already covered by E
  F = F - E
  F = irredundant(F)
  do
      φ2 = |F|
      do
              φ1 = |F|
              F = reduce(F)
              F = expand(F)
              F = irredundant(F)
      while (|F| < φ1)
      F = last_gasp(F)
  while (|F| < φ2)
  F = F ∪ E
  F = make_dhf_prime(F)
```

Figure 2: The Espresso-HF algorithm.

MAKE_DHF_PRIME, since it is desirable to obtain a cover that consists of dhf-prime implicants. The motivation for this step will be made clear in the sequel.

In addition to the steps shown in Figure 2, our implementation has several optional pre- and postprocessing steps.

3.2 Dhf-Canonicalization of Initial Cover

In Espresso-II, the initial cover of a function is provided by its ON-set, F^{ON}. This cover is a seed solution, which is iteratively improved by the algorithm. By analogy, in Espresso-HF, the initial cover is provided by the set of required cubes, Q. However, *unlike* Espresso-II, our initial specification does not in general represent a solution: though Q is a cover, it is not necessarily hazard-free. Therefore, processing begins by expanding each required cube into the *uniquely defined* minimum dhf-implicant containing it. This expansion represents a *canonicalization step*, transforming a potentially hazardous initial cover Q into a hazard-free initial cover Q^f.

Example. Consider the function f in the Karnaugh map of Figure 3. A set T of specified multiple-input transitions is indicated by arrows. There are two $1 \rightarrow 0$ transitions, each corresponding to a privileged cube: $p1 = a'c'$ (start point $p1_{strt} = a'bc'd'$) and $p2 = ad$ (start point $p2_{strt} = abc'd$). The initial cover is given by the set Q of required cubes: $\{a'c'd'$, $a'bc'$, ac', $ac'd$, abd, bcd, $bcd'\}$. This cover is hazardous. In particular, consider the required cube $r = bcd$, corresponding to the $1 \rightarrow 1$ transition from $abcd = 0111$ to 1111. Required cube r illegally intersects privileged cube $p2$, since it intersects $p2$ but does not contain $p2_{strt}$. To avoid illegal intersection, r must be expanded to the smallest cube which also contains $p2_{strt}$: $r^{(1)} = \text{supercube}(\{r, p2_{start}\})$. However, this new cube $r^{(1)} = bd$ now illegally intersects privileged cube $p1$, since it does not contain $p1_{strt}$. Therefore, cube $r^{(1)}$ in turn must be expanded to the smallest cube containing $p1_{strt}$: $r^{(2)} = \text{supercube}(\{r^{(1)}, p1_{start}\})$. The resulting expanded cube, $r^{(2)} = b$, has no illegal intersections and is therefore a dhf-implicant. □

In this example, $r^{(2)}$ is a hazard-free expansion of r, called a **canonical required cube**; it can therefore replace r in the initial cover. (Note that such a canonicalization is feasible if and only if the hazard-free covering problem has a solution; see Section 3.10.)

Thus, an initial set Q of required cubes is replaced by a set Q^f of canonical required cubes (after having been minimized with respect to single cube containment). Q^f is a

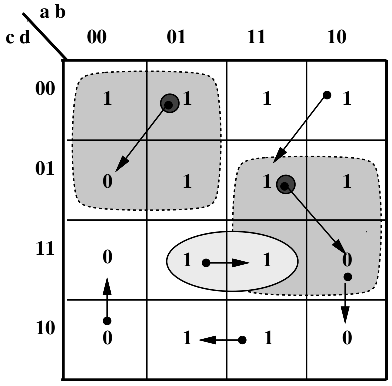

Figure 3: Canonicalization Example

valid hazard-free cover of the function to be minimized, and is used as an initial cover for the minimization process. In fact, Q^f has a second role as well: it is used to simplify the covering problem. In particular, Q^f defines a new covering problem: each cube of Q^f (*not* Q) must be contained in some dhf-implicant. It is straightforward to show that the two covering problems are equivalent: if a dhf-implicant p contains a required cube r in Q, p must also contain the canonical required cube of r in Q^f; if not, p would not be a dhf-implicant.

In the above example, any dhf-implicant which contains required cube $r = bcd$ must also contain canonical required cube $r^{(2)} = b$. Therefore, the hazard-free minimization problem is unchanged, but canonical required cubes are used. An advantage of using Q^f is that it may have smaller size than Q, i.e. being a more efficient representation of the problem. Also, since the cubes in Q^f are in general larger than the corresponding ones in Q, the EXPAND operation may be sped up.

In sum, the set of canonical required cubes Q^f replaces the set of required cubes Q as both (i) the initial cover, and (ii) the set of objects to be covered. Henceforth, the term "set of required cubes" will be used to refer to set Q^f.

We formalize the notion of canonicalization below.

Definition 3.1 *Let f be a Boolean function, T be a set of function hazard-free transitions, and C be a set of implicants. The* **dhf-supercube of** C *with respect to function f and transitions T, indicated as $supercube_{dhf}^{(f,T)}(C)$, is the smallest dhf-implicant containing the cubes of C.*

The superscript (f, T) is omitted when it is clear from the context. $supercube_{dhf}(C)$

$supercube_{dhf}$ (set of cubes $C = \{c_1, \ldots, c_n\}$)

$r = supercube(\{c_1, \ldots, c_n\})$

while (r intersects some privileged cube p_i illegally)

$r = supercube(\{r, s_i\})$ where s_i is the start point of p_i

if r intersects the OFF-set then return "undefined" else return r

Figure 4: $Supercube_{dhf}$ computation

is computed by the simple algorithm shown in Figure 4.

The *canonical required cube* of a required cube r can now be defined as the *dhf-supercube* of the set $C = \{r\}$. The computation of dhf-supercubes for larger sets will be needed to implement some of the operators presented in the sequel.

3.3 Expand

In ESPRESSO-II, the goal of EXPAND is to enlarge each implicant of the current cover in turn into a prime implicant. As an implicant is expanded, it may contain other implicants of the cover which can be removed, hence the cover cardinality is reduced. If the current implicant cannot be expanded to contain another implicant completely, then, as a secondary goal, the implicant is expanded to overlap as many other implicants of the current cover as possible.

In ESPRESSO-HF, the primary goal is similar: to expand a dhf-implicant of the current cover to contain as many other dhf-implicants of the cover as possible. However, EXPAND in ESPRESSO-HF has two major differences. Unlike ESPRESSO-II, expansion in some literal (*i.e.,* "raising of entries") may *imply* that other expansions be performed. That is, raising of entries is now a *binate problem*, not a unate problem. Furthermore, ESPRESSO-HF's EXPAND uses a different strategy for its secondary goal. By the Hazard-Free Covering Theorem, each required cube needs to be contained in some cube of the cover. Therefore, as a secondary goal, an implicant is expanded to contain as many required cubes as possible.

We now describe the implementation of EXPAND in ESPRESSO-HF. Pseudocode for the expansion of a single cube is shown in Figure 5.

Expand_cube(cube a, req-set Q^f, priv-set P, cover-set F, OFF-set R)

$F_a = F - a$

$Q_a = Q^f$

$P_a = P$

$R_a = R$

$free\_entries$ =complement_pos_cube_notation(a)

while ($F_a \neq \oslash$)

update($a, free\_entries, F_a, Q_a, P_a, R_a$)

$F_a = \{c \in F_a | supercube_{dhf}(\{a, c\})$ is defined $\}$

Let c_b be the best candidate in F_a

$a = supercube_{dhf}(\{a, c_b\})$

while ($Q_a \neq \oslash$)

update($a, free\_entries, F_a, Q_a, P_a, R_a$)

$Q_a = \{q \in Q_a | supercube_{dhf}(\{a, q\})$ is defined $\}$

Let q_b be the best candidate in Q_a

$a = supercube_{dhf}(\{a, q_b\})$

Figure 5: Expand (for a cube a)

3.3.1 Determination of Essential Parts and Update of Local Sets

As in Espresso-II, *free entries* are maintained, to accelerate the expansion [26]. The free entries consist of all entries of the current implicant, in positional cube notation [16], which are still candidates to be raised to 1. Initially, a free entry is assigned a 1 (0) if the current implicant to be expanded, a, has a 0 (1) in the corresponding position. An *overexpanded cube* is defined as the cube a where all free entries have been raised simultaneously.

An *essential part* is one which can never, or always, be raised[26]. Our definition of "essential parts" is different from Espresso-II, since a hazard-free cover must be maintained.

First, we determine which entries can *never be raised* and remove them from $free\_entries$. This is achieved by searching for any cube in the OFF-set R that has distance 1 from a, using the same approach as in Espresso-II.

Next, we determine which parts can *always be raised*, raise them and remove them from $free\_entries$. This step differs from Espresso-II. In Espresso-II, a part can always be raised if it is 0 in all cubes of the OFF-set, R. That is, it is guaranteed that the expanded cube will never intersect the OFF-set. In contrast, in ESPRESSO-HF, we must ensure that an implicant is also *hazard-free*: it cannot intersect the OFF-set, nor can it illegally intersect a privileged cube. Unlike in ESPRESSO-II, this is achieved by searching for any column that has only 0s in R AND where each 1 in P implies that the corresponding start point is covered by a.

Example. Figure 1(a) indicates the set of required cubes, which forms an initial hazard-free cover. Consider the cube bcd (11010101, in positional cube notation). As in ESPRESSO-II, the 0-entries for literals b' and d' can never be raised, since the cube would intersect the OFF-set. However, after updating the free entries, ESPRESSO-II indicates that literal c' can *always* be raised, since the resulting cube will never intersect the OFF-set. In contrast, in ESPRESSO-HF, raising c' results in an illegal intersection with privileged cube $a'c'$, so it cannot "always be raised". □

Since the hazard-free minimization is somewhat more constrained, the expansion of a cube a can be accelerated by the following new operations on 3 local sets: P_a, R_a, Q_a. These sets are associated with cube a, and are *updated* as expansion proceeds. (1) Remove privileged cubes from P_a where the corresponding start point is already covered by a (since no further checking for illegal intersection is required). (2) Move privileged cubes from set P_a to the local OFF-set R_a if the overexpanded cube does not include the corresponding start points (since a can never be expanded to include these start points, therefore one must avoid intersection with these privileged cubes entirely). (3) Move privileged cubes from P_a to the local OFF-set R_a where $supercube_{dhf}$({a, start point}) intersects the OFF-set (a can never be expanded to include these start points, therefore one must avoid intersection with the cubes entirely).

3.3.2 Detection of Feasibly Covered Cubes of F

In ESPRESSO-II, a cube in F is expanded through a *supercube* operation. A cube d in F is said to be *feasibly covered* by a if supercube($\{a,d\}$) (the smallest cube containing both a and d) is an implicant. In ESPRESSO-HF, this definition needs to be modified to insure *hazard-free covering*, after expansion of cube a.

Definition 3.2 *A cube d in F is* **dhf-feasibly covered** *by a if $supercube_{dhf}(\{a,d\})$ is defined.*

This definition insures that the resulting expanded cube, $supercube_{dhf}(\{a,d\})$, is (i) an implicant (does not intersect OFF-set), and (ii) is also a dhf-implicant (does not intersect any privileged cube illegally). Effectively, this definition canonicalizes the resulting supercube to produce a dhf-implicant. That is, $supercube_{dhf}(\{a,d\})$ may properly contain supercube($\{a,d\}$), since the former may be expanded through a series of *implications* in order to reach the minimum dhf-implicant which contains both a and d. Using this definition, the following is an algorithm to find dhf-feasibly covered cubes of F.

While there are cubes in F that are dhf-feasibly covered, iterate the following:

> Replace a by $supercube_{dhf}(\{a, d\})$, where d is a dhf-feasibly covered cube such that the resulting cube will cover as many cubes of the cover as possible. Covered cubes are then removed, reducing the cover cardinality. Determine essential parts and update local sets (see above).

3.3.3 Detection of Feasibly Covered Cubes of Q^f

Once cube a can no longer be feasibly expanded to cover any other cube, d, of F, we still continue to expand it. This is motivated by the Hazard-Free Covering Theorem, which states that each required cube needs to be contained in some cube of the cover. Therefore, as a secondary goal, cube a is expanded to contain as many required cubes as possible. The strategy used in this sub-step is similar to the one used in the preceding one, i.e. while there are cubes in Q^f that are dhf-feasibly covered, iterate the following:

> Replace a by $supercube_{dhf}(\{a, q\})$, where q is a dhf-feasibly covered *required cube* such that the resulting cube will cover as many required cubes not already contained in a as possible. Covered required cubes are then removed. Determine essential parts and update local sets (see above).

3.3.4 Constraints on Hazard-Free Expansion

In Espresso-II, an implicant is expanded until no further expansion is possible, i.e. until the implicant is prime. Two steps are used: (i) expansion to overlap a maximum number of cubes still covered by the overexpanded cube; and (ii) raising of entries to find the largest prime implicant covering the cube.

In ESPRESSO-HF, however, we do not implement these remaining EXPAND steps, based on the following observation. The result of our EXPAND steps (cf. 3.3.2 and 3.3.3) guarantees that a dhf-implicant can never be further expanded to contain additional required cubes. Therefore, by the Hazard-Free Covering Theorem, no additional objects (required cubes) can be covered through further expansion. In contrast, in ESPRESSO-II, further expansion steps *may* result in covering additional ON-set minterms. Because of this distinction, the benefits of further expansion are mitigated. Therefore, in general, our algorithm does not transform dhf-implicants into dhf-prime implicants. However, since expansion to dhf-primes is important for literal reduction and testability, it is included as a final post-processing step: MAKE_DHF_PRIME (see Figure 2).

3.4 Essentials

Essential prime implicants are prime implicants that need to be included in any cover of prime implicants. Therefore, it is desirable to identify them as soon as possible to make the resulting problem size smaller. On the one hand, we know of no efficient solution for identifying the essential dhf-primes using the unate recursion paradigm of ESPRESSO-II. On the other hand, the hazard-free minimization problem is highly constrained by the notion of covering of *required cubes*, allowing a powerful new method to classify essentials as equivalence classes.

Example. Consider Figure 6. The required cube, $r = bcd$, is covered by precisely two dhf-prime implicants: $p1 = bd$ and $p2 = cd$. Neither $p1$ nor $p2$ is an essential dhf-prime, since r is covered by both. And yet, clearly, either $p1$ or $p2$ (not both) *must* be included in any cover of dhf-primes. Also, if we assume the standard cost function of cover cardinality, $p1$ and $p2$ are of equal cost. $\square$

Our EXPAND method therefore supports the notion of *equivalence classes*, since implicants are not expanded beyond the required cubes which they cover. In the above example, product r (regarded as a covering object) would not be expanded further, since no feasible required cubes can be found. Cube r therefore represents an *essential equivalence class*, corresponding to the set $\{bd, cd\}$ of dhf-primes. It should be removed from the cover.

ESPRESSO-II computes essentials after an initial EXPAND and IRREDUNDANT. In contrast, ESPRESSO-HF computes essentials as part of a modified EXPAND-step.

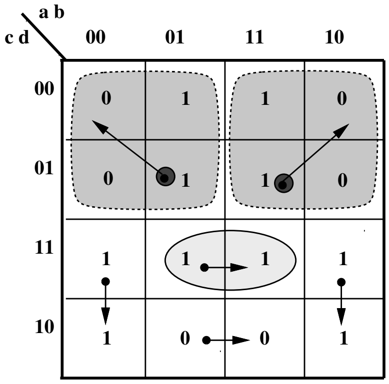

Figure 6: Essential Example

The algorithm is outlined as follows:

The algorithm starts with the initial hazard-free cover, Q^f, of required cubes. To simplify the presentation, assume that one seed cube is selected and expanded greedily, using EXPAND, to a dhf-implicant p. This implicant is characterized by the set, Q^p, of required cubes which it contains. Dhf-implicant p is called an **essential equivalence class** if it contains some required cube, q^f, which cannot be expanded into any other equivalence class. To check if q^f can be expanded into a different equivalence class, a simple pairwise check is used: for each required cube s^f not covered by p, determine if $supercube_{dhf}(\{q^f, s^f\})$ is feasible. If no such feasible expansion exists for q^f, q^f is called a **distinguished required cube**, and therefore p is essential. Otherwise, the process is repeated for every required cube q^f contained in Q^p. Whenever an essential p is identified, all required cubes covered by p are removed, and the covering problem is updated. This step can result in "secondary essential" equivalence classes. The procedure iterates until all essentials are identified.

The above discussion seems to imply that the essentials step is more or less quadratic in the number of required cubes, i.e. very inefficient. However, by making use of techniques similar to the ones described in the EXPAND-section, e.g. by using an overexpanded cube, the number of necessary $supercube_{dhf}$-calls can be reduced dramatically. Therefore, in practice, essentials can be identified efficiently and the problem size is usually significantly reduced (see Section 6).

3.5 Reduce

The goal of the REDUCE operator is to set up a cover that is likely to be made smaller by the following EXPAND step. To achieve this, each cube c in a cover F is maximally reduced in turn to a cube $\tilde{c}$, such that the resulting set of cubes, $\{F - c\} \cup \tilde{c}$ is still a cover.

ESPRESSO-II uses the unate recursive paradigm to maximally reduce each cube. Since ESPRESSO-HF is a required cube covering algorithm, there is no obvious way to use this paradigm. Fortunately, the hazard-free problem is more constrained, making it possible to use an efficient enumerative approach based on required cubes.

Our REDUCE algorithm is as follows. The algorithm reduces each cube c in the cover in order. In particular, a cube c is reduced to the smallest dhf-implicant $\tilde{c}$ that covers all required cubes that are uniquely coverd by c (i.e. contained in no other cube of the cover F). This means, that if $r_1, \ldots, r_l$ is the set of required cubes that are uniquely covered by c, then c is replaced by $\tilde{c} = supercube_{dhf}(\{r_1, \ldots, r_l\})$.

Note that the outcome of this algorithm depends on the order in which the cubes c of the cover F are processed. Suppose c_i is reduced before c_j, and that c_i and c_j cover some required cube r but no other cube of F covers r. If c_i is reduced to a cube $\tilde{c_i}$ that does not cover r, then c_j cannot be reduced to a cube that does not cover r.

3.6 Irredundant

ESPRESSO-II uses the unate recursive paradigm to find an irredundant cover. However, in our case, there is no obvious way to employ this paradigm, since a "redundant cover" (according to covering of minterms) may in fact be irredundant with respect to covering of required cubes.

Therefore, as in REDUCE, our approach is required-cube based. Considering the Hazard-Free Covering Theorem, it is straightforward that IRREDUNDANT can be reduced to a covering problem of the cubes in Q^f by the cubes in F. That is, the problem reduces to a minimum-covering problem of (i) required cubes, using (ii) dhf-implicants in the current cover. In practice, the number of required cubes and cover cubes usually make the covering problem manageable. ESPRESSO-II's MINCOV can be used to solve this covering problem exactly, or heuristically (using its heuristic option).

3.7 Last Gasp

The inner loop of ESPRESSO-HF may lead to a suboptimal local minimum. The goal of LAST_GASP is to use a different approach to attempt to reduce the cover size. In ESPRESSO-II, each cube $c \in F$ is independently reduced to the smallest cube containing all minterms not covered by any other cube of F. In contrast, ESPRESSO-HF computes, for each $c \in F$, the smallest dhf-implicant containing all *required cubes* that are not covered by any other cube in F.

As in ESPRESSO-II, cubes that can actually be reduced by this process are added to an initially empty set G. Each such $g \in G$ is then expanded in turn with the goal to cover at least one other cube of G, using the $supercube_{dhf}$ operator, and if achieved added to F. Finally, the IRREDUNDANT operator is applied to F with the hope to escape the above-mentioned local minimum.

3.8 Make dhf-prime

The cover being constructed so far does not necessarily consist of dhf-primes. It is usually desirable to expand each dhf-implicant of the cover to make it dhf-prime as a last step. This can be achieved by a modified EXPAND step. A simple greedy algorithm will expand an implicant c to a dhf-prime: While dhf-feasible, raise a single entry of c.

3.9 Pre- and postprocessing steps

ESPRESSO-HF includes optional pre- and postprocessing steps. In particular, the efficiency of ESPRESSO-HF depends very much on the size of the ON-set and OFF-set covers that are given to it. Thus, ESPRESSO-HF includes an optional *preprocessing* step which uses ESPRESSO-II to find covers of smaller size for the initial ON-set and OFF-set [5]. ESPRESSO-HF also includes a *postprocessing* step to reduce the literal count of a cover, similar to ESPRESSO-II's MAKE_SPARSE.

[5]ON-set and OFF-set are necessary to form the initial set of required cubes, Q. More importantly, the OFF-set is used to check if a cube expansion is valid, see Figure 4.

3.10 Existence of a hazard-free solution

As indicated earlier, for certain Boolean functions and sets of transitions, no hazard-free cover exists. The currently used exact hazard-free minimization method HFMIN is only able to decide if a hazard-free solution exists after generating all dhf-prime implicants. A solution does not exist if and only if the dhf-prime implicant table includes at least one required cube not covered by any dhf-prime implicant.

Since the generation of all primes may very well be infeasible [6] for even medium-sized examples, it is important to find an alternative approach. We therefore present a new theorem for the existence of a solution, leading directly to a fast and simple algorithm that is incorporated into ESPRESSO-HF.

Theorem 3.3 *A solution of the hazard-free minimization problem exists iff* $supercube_{dhf}(q)$ *is defined for all required cubes* q.

The proof is immediate from the discussion in Section 3.2.

Example. Consider the Boolean function in Figure 7, with four specified input transitions. To check for existence of a hazard-free solution, we compute $supercube_{dhf}(q)$ for each required cube q. Except for abd, it holds that $q = supercube_{dhf}(q)$ since no privileged cube is intersected illegally. To compute $supercube_{dhf}(abd)$, note that privileged cube c is intersected illegally, i.e. $supercube_{dhf}(abd) = supercube_{dhf}(bd)$. Since bd now intersects privileged cube $a'c'$, we get $supercube_{dhf}(abd) = supercube_{dhf}(b)$ leading directly to the fact that $supercube_{dhf}(abd)$ does not exist because b intersects the OFF-set. Thus, there is no hazard-free cover for this example. □

4 A Novel Approach of Incorporating Hazard-Freedom Constraints Within a Synchronous Function

After having discussed the *heuristic hazard-free minimization problem* in the previous section, we will now shift our discussion to the *exact hazard-free minimization problem.*

[6] This refers to "explicit representations"; we will show later that "implicit representations" very often are feasible.

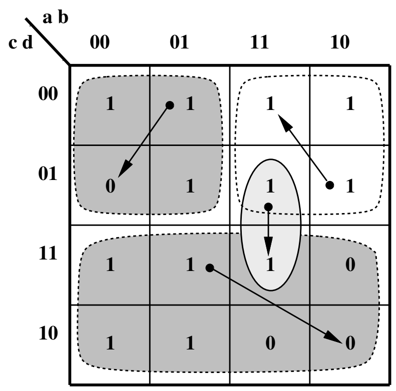

Figure 7: Existence Example

We begin by presenting, in this section, a novel technique which recasts the dhf-prime implicant generation problem into a prime generation problem for a new *synchronous* function, with extra inputs. Based on this approach, we present a new implicit method for exact 2-level hazard-free logic minimization in Section 5.

4.1 Overview and Intuition

In this subsection, we first give a simple overview of our entire method. Details and formal definitions are provided in the remaining subsections.

Our approach is to recast the generation of dhf-prime implicants of an asynchronous function (f, T) into the generation of prime implicants of a synchronous function g. Here, hazard-freedom constraints are incorporated into the function g by adding extra inputs. An overview of the method is best illustrated by a simple example.

Example 4.1 Consider Figure 8. The Karnaugh map in part A represents a function (f, T) defined over the set of 3 variables $\{x_1, x_2, x_3\}$. The shaded area corresponds to the only non-trivial privileged cube of f (the second privileged cube $[101, 100]$ is trivial, cf. Section 2.6). We now define a new *synchronous* function g, shown in part B. g is obtained from f by adding a single new variable z_1. That is, g is defined over 4 variables: $\{x_1, x_2, x_3, z_1\}$. In general, to generate g, one new z-variable is added for each non-trivial privileged cube. Next, the prime implicants of the synchronous function g are computed (shown in part B as ovals). Finally, we use a simple filtering procedure to filter out those prime implicants that correspond to those in f which intersect the

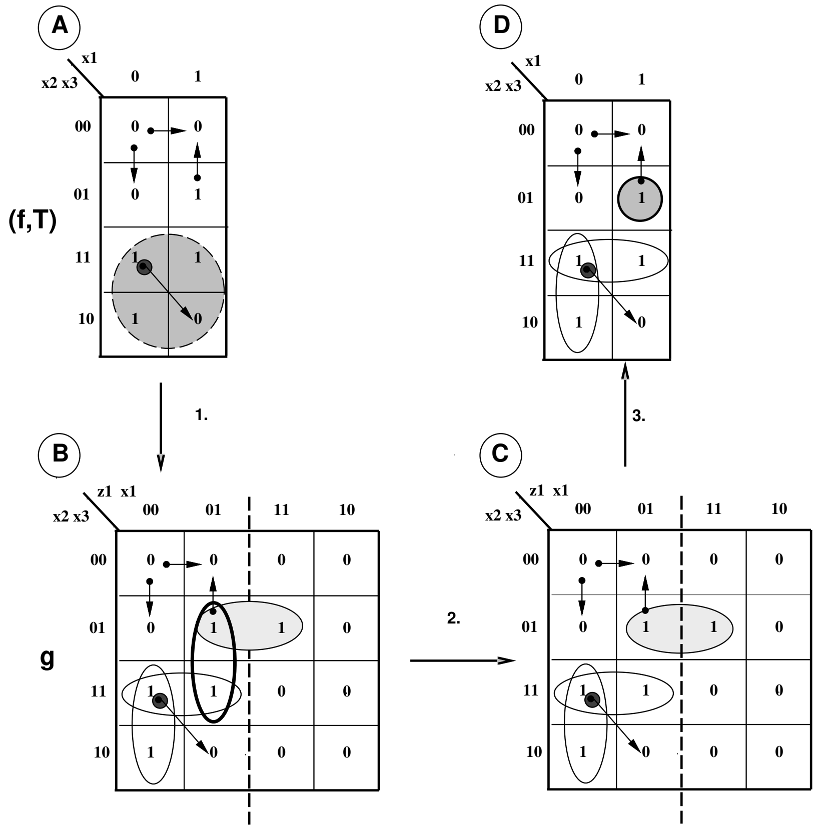

Figure 8: Example for recasting prime generation. A) shows the function (f, T) whose dhf-primes are to be computed. B) shows the auxiliary synchronous function g and its primes. C) shows primes of g that do not intersect illegally. D) shows the final dhf-primes of f, after deleting the z_1 variable.

privileged cube illegally. The remaining prime implicants of g are shown in part C. We then “delete” the z_1-dimension from the prime implicants, and obtain the entire set of dhf-prime implicants of (f, T) (part D). □

Our approach is motivated by the fact that dhf-prime-implicants are more constrained than prime implicants of the *same* function. While prime implicants are maximal implicants that do not intersect the OFF-set of the given function, dhf-prime-implicants, in addition, must also not intersect privileged cubes illegally. This means that there are two different kinds of constraints for dhf-prime-implicants: “maximality” constraints and “avoidance of illegal intersections” constraints. Our idea is now to unify these two types of constraints, i.e. to transform the avoidance constraints into maximality constraints so that dhf-primes can be generated in a uniform way. Intuitively, this can be achieved by adding auxiliary variables, i.e. by lifting the problem into a higher-dimensional Boolean space.

In summary, the big picture is as follows. The definition of g ensures that all dhf-prime implicants of f (*dhf-Prime(f,T)*) can be easily obtained from the set of prime implicants of g ($Prime(g)$). While $Prime(g)$ may also include certain products which are non-hazard-free, these are filtered out easily, using a post-processing step.

4.2 The auxiliary synchronous function g

In this subsection, we explain how the synchronous function g is derived. For simplicity, assume for now that f is a single-output function.

Suppose f is defined over the set of variables $\{x_1, \ldots, x_n\}$, and that the set of transitions T gives rise to the set of non-trivial privileged cubes $PRIV(f,T) = \{p_1, \ldots, p_l\}$. The idea is to define a function g over $\{x_1, \ldots, x_n, z_1, \ldots z_l\}$; that is, *one new variable is added per privileged cube.* Formally, g is defined as follows:

$$g(x_1, \ldots, x_n, z_1, \ldots, z_l) = f \cdot \prod_{1 \le i \le l} (\overline{z_i} + \overline{p_i})$$

That is, the function g is the product of f and some function which depends on the added inputs. The intuition behind the definition of g is that in the $z_i = 0$ half of the domain g is defined as f, while in the $z_i = 1$ half of the domain g is defined as f but with the i-th privileged cube p_i "filled in" with all 0's (i.e., p_i is "masked out").

Example 4.2 As an example, Figure 8A shows a Boolean function (f, T) with privileged cube x_2 (highlighted in gray). Figure 2B shows the corresponding new function g, with added variable z_1. In the $z_1 = 0$ half, function g is identical to f. In the $z_1 = 1$ half, g is identical to f except that g is 0 throughout the cube $z_1 x_2$, which corresponds to the privileged cube in the original function f. In particular, function g is defined as $g = f \cdot (\overline{z_1} + \overline{p_1})$, where $p_1 = x_2$. $\square$

4.3 Prime implicants of function g

To understand the role of function g, we consider its prime implicants $Prime(g)$.

We start by considering a function (f, T) that has only *one* privileged cube p_1. Let q be any implicant of the function g that is contained in the $z_1 = 0$ plane of g. Since the

$z_1 = 0$ plane is defined as f, q also corresponds to an implicant of f. Now, consider the expansion of q into the $z_1 = 1$ plane of function g. There are 2 possibilities: either (i) q can expand into $z_1 = 1$ plane, or (ii) q cannot expand into the $z_1 = 1$ plane. In case (i), expansion of q into the $z_1 = 1$ plane means that g is identical to f in the expanded region. Therefore, q does not intersect privileged cube p_1 in the original function f (if it did, g would have all 0's in p_1 in the $z_1 = 1$ plane, and expansion would be impossible). In case (ii), expansion into the $z_1 = 1$ plane is impossible. In this case, q must intersect p_1 in function f (g has all 0's in p_1).

In summary, q may or may not be able to expand from $z_1 = 0$ into $z_1 = 1$ planes. Expansion can occur precisely if q does not intersect the privileged cube p_1 in the original function.

Example 4.3 Consider the minterm $q_1 = \overline{z_1}x_1\overline{x_2}x_3$ of g in Figure 8B, which corresponds to the minterm $x_1\overline{x_2}x_3$ of f. q can be expanded into the $z_1 = 1$ plane into the prime implicant of g: $x_1\overline{x_2}x_3$ (shaded oval). Intuitively, the expansion is possible since q_1 does not intersect the privileged cube, i.e. the cube $\overline{z_1}x_2$, which corresponds to the privileged cube x_2 of the original function f. However, the implicant $q_2 = \overline{z_1}x_1x_3$ (oval with thick dark border) of g *cannot* be expanded into the $z_1 = 1$ plane: it intersects the privileged cube, and therefore the corresponding region in the $z_1 = 1$ plane is filled with 0's. Note that prime generation is an expansion process until no further expansion is possible. □

Let us now consider the general case, i.e. where (f, T) may have more than one privileged cube. We show that the support variables of each prime of g *precisely* indicate which privileged cubes are intersected by the prime's corresponding implicant in f. Let q be any prime implicant of g:

$$q = x_{i_1} \cdots x_{i_{\hat{n}}} z_{j_1} \cdots z_{j_{\hat{l}}}$$

Here, x_{i_k} is a positive or negative x-literal [7]. However, z_{j_k} can *only* be a negative z-literal. The reason is that g is a negative unate function in z-variables (see definition of g), and therefore prime implicants of g will not include positive z-literals.

We indicate by q^x the *restriction of* q *to the* x*-literals*, i.e. $q^x = x_{i_1} \cdots x_{i_{\hat{n}}}$. Note that q^x is an implicant of f by the definition of g. If q includes the literal $\overline{z_i}$, then q^x

[7] Note that q may not depend on all x-variables.

intersects p_i. The reason is that the primality of q indicates that q cannot be expanded into the $z_i = 1$ plane. As explained above, this is equivalent to the intersection of p_i in the original function f. On the other hand, if q does not include $\overline{z_i}$, then q^x does not intersect p_i. Intuitively, the primes, $Prime(g)$, are maximal in two senses: they are maximally expanded in f, or maximally non-intersecting of privileged cubes, in some combination, which is indicated by the set of support of the primes.

Therefore, the key observation is that the set of support of a prime implicant q of g *precisely* indicates which privileged cubes are intersected by the corresponding implicant q^x in f. This observation will be critical in obtaining the final set of dhf-prime implicants of f, $dhf\text{-}Prime(f,T)$.

4.4 Transforming Prime(g) into dhf-Prime(f,T)

Once $Prime(g)$ is computed, $dhf\text{-}Prime(f,T)$ can be directly computed. The key insight for this computation is that the prime implicants of $Prime(g)$ fall into 3 classes with respect to a specific privileged cube p_i. Each prime q is distinguished based on *if* and *how* it intersects the privileged cube p_i in f, i.e. based on the intersection of q^x with p_i:

- Class 1: Prime implicants q that do not intersect the privileged cube, i.e. q^x does not intersect p_i.
- Class 2: Prime implicants q that intersect the privileged cube legally, i.e. q^x intersects p_i and contains its start point.
- Class 3: Prime implicants q that intersect the privileged cube illegally, i.e. q^x intersects p_i but does not contain the start point.

$Dhf\text{-}Prime(f,T)$ can now be computed as follows. Start with $Prime(g)$. Filter out all prime implicants that fall in Class 3 with respect to the first privileged cube. Then, filter out all prime implicants that fall in Class 3 with respect to the second privileged cube, and so on. Finally, we obtain a set such that each of its elements is a valid dhf-implicant of (f,T) if restricted to the x-variables. The reason is, first, that all primes of g are implicants of f if restricted to x-variables. Second, the filtering removed any element that intersected any privileged cube illegally. Therefore, the set only includes dhf-implicants. In fact, it contains *all* dhf-prime-implicants of (f, T). This will be proven in the next subsection.

Example 4.4 Figure 8B shows function g and its prime implicants, $Prime(g) = \{x_1\overline{x_2}x_3, \overline{z_1}x_1x_3, \overline{z_1}x_2x_3, z_1\overline{x}_1x_2\}$. Part C shows the result of filtering out primes that illegally intersect regions corresponding to privileged cubes in f. In this case, $\overline{z_1}x_1x_3$ (oval with thick dark border) falls into Class 3 with respect to p_1: it is deleted since it has a $\overline{z_1}$-literal, i.e. intersects the region corresponding to privileged cube p_1 and does not contain the start point $\overline{z}_1\overline{x}_1x_2x_3$. However, $x_1\overline{x_2}x_3$ (shaded oval) falls into Class 1: it is not deleted since it does not have a $\overline{z_1}$-literal and therefore does not intersect the region corresponding to the privileged cube p_1. The remaining two primes $\overline{z_1}x_2x_3$ and $\overline{z}_1\overline{x}_1x_2$ fall into Class 2: they intersect the region corresponding to p_1 and contain the start point. Part D shows the result of step 3 which deletes the z-literals in each cube. We obtain $\{x_1\overline{x_2}x_3, x_2x_3, \overline{x_1}x_2\}$, which is $dhf\text{-}Prime(f, T)$. Note that the introduction of the z_1-variable ensures that the dhf-implicant of f, $x_1\overline{x_2}x_3$, which is not a prime implicant of f, since it is contained by the prime implicant, x_1x_3, is nevertheless generated. □

4.5 Formal characterization of dhf-Prime(f,T) in terms of function g

In this subsection, based on above discussion, we present the main result of this section: a new formal characterization of $dhf\text{-}Prime(f, T)$. We use the following notations. g_{z_i} and $g_{\overline{z_i}}$ denote the positive and negative cofactors of g with respect to variable z_i, respectively. $RemZ$ denotes an operator on a set of cubes which removes all z-literals of each cube. As an example, $RemZ\left(\{x_1x_2z_1, x_1x_3\overline{z_2}, x_1x_3z_1z_3\}\right) = \{x_1x_2, x_1x_3\}$. The SCC-operator on a set of cubes (single-cube-containment) removes those cubes contained in other cubes.

Theorem 4.5 *Given* (f, T). *Let* $PRIV(f, T) = \{p_1, \ldots, p_l\}$ *be the set of non-trivial privileged cubes, and* $START(f, T) = \{s_1, \ldots, s_l\}$ *be the set of corresponding start points. Define*

$$g(x_1, \ldots, x_n, z_1, \ldots, z_l) = f \cdot \prod_{1 \leq i \leq l} (\overline{z_i} + \overline{p_i})$$

Then the set dhf-Prime(f, T) can be expressed as follows:

$$SCC\left(\bigcap_{1\le i\le l} \Big[\; RemZ\left(Prime\left(g_{z_i}\right)\right) \cup \left\{q \in RemZ\left(Prime\left(g_{\overline{z_i}}\right)\right) | q \supseteq s_i\right\} \Big] \right)$$

Intuition: $RemZ\left(Prime\left(g_{z_i}\right)\right)$ includes implicants of f that do not intersect the privileged cube p_i. $\left\{q \in RemZ\left(Prime\left(g_{\overline{z_i}}\right)\right) | q \supseteq s_i\right\}$ includes implicants of f that legally intersect p_i, i.e. contain the corresponding start point s_i. The $\bigcap$ ensures that only those implicants remain that are legal with respect to all privileged cubes, *i.e.* that are dhf-implicants. The SCC removes implicants contained in other implicants to yield the final set of dhf-prime-implicants.

Proof: "$\subseteq$" (any product in dhf$-$Prime(f,T) is also contained in the SCC-expression [8]):
Let $q \in dhf\text{-}Prime(f, T)$, then q does not intersect any privileged cube illegally, i.e. for each privileged cube it holds that q either contains the corresponding start point or does not intersect the privileged cube at all.

Suppose q intersects legally $p_1, \ldots, p_{\hat{l}}$, and q does not intersect $p_{\hat{l}+1}, \ldots, p_l$ - i.e. q is an implicant of $\overline{p_{\hat{l}+1}}, \ldots, \overline{p_l}$ -, then $q\overline{z_1}\cdots\overline{z_{\hat{l}}}$ is an implicant of g.

$q\overline{z_1}\cdots\overline{z_{\hat{l}}}$ is a prime implicant of g because:

(i) Removing (any) $\overline{z_i}$ results in a cube which is not an implicant of $\overline{z_i}+\overline{p_i}$, and hence not an implicant of g.

(ii) Removing (any) positive or negative x_j literal (of q) results in a cube such that its restriction to the x-literals, q_{new}, either intersects the OFF-set of f, or intersects for some i privileged cube p_i, $i \in \{\hat{l}+1, \ldots, l\}$ and is therefore no longer an implicant of $\overline{z_i} + \overline{p_i}$. In either case q_{new} is not an implicant of g.

Thus, for each i, q is by construction in at least one of $RemZ\left(Prime\left(g_{z_i}\right)\right)$ or $\left\{q \in RemZ\left(Prime\left(g_{\overline{z_i}}\right)\right) | q \supseteq s_i\right\}$). Therefore, q is contained in the intersection of those l sets. Also, q cannot be filtered out by the SCC-operator since by construction all

[8] "SCC-expression" refers to the entire expression:
$SCC\left(\bigcap_{1\le i\le l} \left[RemZ\left(Prime\left(g_{z_i}\right)\right) \cup \left\{q \in RemZ\left(Prime\left(g_{\overline{z_i}}\right)\right) | q \supseteq s_i\right\} \right] \right)$

cubes contained in the SCC-expression are dhf-implicants. Thus, q is contained in the SCC-expression.

"$\supseteq$" (any product contained in the SCC-expression is also contained in dhf-Prime(f)): Let $q \notin dhf\text{-}Prime(f, T)$. We show that q is not contained in the SCC-expression.

Case (i): q is a dhf-implicant that is strictly contained in some dhf-prime implicant. Then q is filtered out because of the SCC-operator and therefore not contained in the SCC-expression.

Case (ii): q is not a dhf-implicant. Since by construction all cubes contained in the SCC-expression are dhf-implicants, q cannot be contained in the SCC-expression. □

4.6 Multi-output Case

For simplicity of presentation only, it was assumed that f is a single-output function. However, it is well-known [29] that multi-output logic minimization can be reduced to single-output minimization. Based on this theorem, the above characterization carries over in a straightforward way to multi-output functions. All examples given later in the experimental results section are multi-output functions.

5 Exact hazard-free minimization: Impymin

Based on the ideas of the previous section, we are now able to present a new exact minimization algorithm for multi-output 2-level hazard-free logic. We will show in the next section that our implicit method outperforms existing minimizers by a large factor.

Nowick/Dill reduced 2-level hazard-free optimization to a unate covering problem (see Section 2) where each required cube has to be covered by at least one dhf-prime implicant. As with synchronous logic minimization in Scherzo [9], hazard-free logic minimization can also be considered over the lattice of the set of products (over the set of literals). The major difference to synchronous two-level logic minimization is the setting up of the covering problem, i.e. we need to find a method that computes the set dhf-Prime(f,T) efficiently, i.e. preferably in an implicit manner. Fortunately, this can be done using the new characterization of dhf-Prime(f,T) of Section 4. Our algorithm is

[9] An introductory discussion of Scherzo can be found in the appendix.

as follows.

Algorithm: Implicit hazard-free logic minimization
Input: Boolean function f, set of input transitions T.
Output: All minimum hazard-free 2-level implementations of (f, T).

1. Compute the ZBDD [10] $P^{(init)}$ of $dhf\text{-}Prime(f, T)$.

2. Compute the ZBDD $Q^{(init)}$ of $REQ(f, T)$ (set of required cubes of (f, T)).

3. Solve the implicit unate set covering problem $\langle Q^{(init)}, P^{(init)}, \subseteq \rangle$.

We now explain each of the steps in detail.

5.1 Computation of the ZBDD of dhf-Prime(f,T)

Suppose that f is given as a BDD (if f is given as a set of cubes, we first compute its BDD). From the BDD representing f, we can easily compute the BDD representing g, and then the ZBDD of $Prime(g)$ using an existing recursive algorithm [5]. From the ZBDD of $Prime(g)$, we compute the ZBDD of $dhf\text{-}Prime(f, T)$ using Theorem 4.5. It remains to show that the necessary operations $Prime(g_{z_i})$, $Prime(g_{\overline{z_i}})$, $RemZ$, and SCC can be implemented efficiently on ZBDDs:

- *Computing* $Prime(g_{z_i})$: Assuming that positive and negative literal nodes of the same variable are always adjacent in the ZBDD, we only need to traverse the ZBDD of $Prime(g)$. We apply at each z_i variable the following operation. We compute the set union of the two successors corresponding to those products that include positive literal z_i and to those products that do not depend on z_i.

- *Computing the ZBDD of* $Prime(g_{\overline{z_i}})$: Analogously.

- *Computing the ZBDD of* $RemZ$: RemZ deletes all z-literals in the ZBDD. We traverse the ZBDD, and at each z_i- or $\overline{z_i}$-literal, we replace the corresponding node with the ZBDD corresponding to the union of the two successors.

- *SCC (Single-Cube Containment)*: The last task, the application of the SCC-operator, which removes cubes contained in other cubes, is actually not done in this step, since it is automatically taken care of in step 3.

[10] Background on BDDs and ZBDDs is provided in the appendix.

To summarize, based on Theorem 4.5 we can compute the covering objects, *dhf-Prime*(f, T), in an implicit manner.

5.2 Computation of the ZBDD of REQ(f,T)

From the set of input transitions, T, the set of required cubes can be easily computed (see [25]). The set of required cubes can then be stored as a ZBDD.

5.3 Solving the Implicit Covering Problem

The implicit set covering problem $\langle Q^{(init)}, P^{(init)}, \subseteq\rangle$ can be solved analogously to Step 3 of SCHERZO (i.e. passed to the unate set covering solver of SCHERZO).

One subtle difference regarding the correctness is worth considering. SCHERZO's τ operators map products onto other products (for details, see the Appendix). It is possible that a product which is a dhf-implicant is mapped, by τ, onto a non-dhf implicant. This does *not* do any harm because we are ensured that all products of the final solution produced by the solver are products that were given to the solver, i.e. dynamic-hazard-free, through a re-mapping operation (see Step 3(c) in the Appendix). Hence, it is fine to use SCHERZO's set covering solver as a black box.

5.4 A Note on the Efficiency of IMPYMIN

It is worth pointing out that appending z-variables for dhf-prime generation is only a small change to the corresponding synchronous problem. In particular, the BDD for g is not much larger than the BDD for f. Thus, the generation of *dhf-Prime*(f, T) can be done nearly as fast as the generation of primes *without* hazard-freedom considerations. Moreover, the resulting covering problem is unlikely to be much harder than the corresponding synchronous problem. To summarize, the proposed method performs hazard-free logic minimization nearly as efficient as synchronous logic minimization by incorporating state-of-the-art techniques for implicit prime generation and implicit set covering solving. However, note that this could only be achieved based on the presented new and non-trivial formulation of the set of dhf-prime implicants, presented in Section 4.

6 Experimental Results and Comparison with Related Work

Prototype versions of our two new minimizers ESPRESSO-HF[11] and IMPYMIN were run on several well-known benchmark circuits [10, 32] on an ULTRA-SPARC 140 workstation (Memory: 89 MB real/ 230 MB virtual).

6.1 Comparison of exact minimizers: IMPYMIN vs. HFMIN

The table in Figure 9 compares our new exact minimizer IMPYMIN with the currently fastest available exact minimizer, HFMIN, by Robert Fuhrer et al. [10].

For the smaller problems, HFMIN is faster, since our implementation is not yet optimized [12]. However, the bottleneck of HFMIN becomes clearly visible already for medium-sized examples. For examples *sd-control* and *stetson-p2*, IMPYMIN is more than three times faster; for the benchmark *pscsi-pscsi* even more than fifteen times.

For very large examples, IMPYMIN outperforms HFMIN by a large factor. While HFMIN cannot solve *stetson-p1* within 20 hours, we can solve it in just 813 seconds. The superiority of implicit techniques becomes very apparent for the benchmark *cache-ctrl*. While HFMIN gives up (after many minutes of run-time) because the 230MB of virtual memory are exceeded, our method can minimize the benchmark in just 301 seconds.

6.2 Comparison of our new methods: IMPYMIN vs. ESPRESSO-HF

Figure 10 compares our two new minimizers ESPRESSO-HF and IMPYMIN. Besides run-time and size of solution, the table also reports the number of essentials (for ESPRESSO-

[11]Our implementation is not a simple modification of the ESPRESSO-II code. We do *not* re-use any ESPRESSO-II code. The reason is that while we use the same set of main operators - EXPAND, REDUCE, IRREDUNDANT - the algorithms that implement these operators, as explained in detail in Section 3, are actually very different from ESPRESSO-II.

[12]Our BDD package is still very inefficient. In particular, it includes a static (i.e. not a dynamic) hashtable. The hashtable for small examples is unnecessarily large. In fact, the run-time is completely dominated by initializing the hashtables. If we use an appropriate-sized hashtable for smaller examples, experiments indicate that IMPYMIN can solve the small examples as fast as HFMIN.

			HFMIN [**FLN**]	IMPYMIN
name	*i/o*	*#c*	*time(s)*	*time(s)*
cache-ctrl	20/23	97	impossible	301
dram-ctrl	9/8	22	1	13
pe-send-ifc	12/10	27	9	16
pscsi-ircv	8/7	12	1	10
pscsi-isend	11/10	23	3	15
pscsi-pscsi	16/11	77	1656	105
pscsi-tsend	11/10	22	3	13
pscsi-tsend-bm	11/11	23	3	13
sd-control	18/22	34	172	52
sscsi-isend-bm	10/9	22	1	11
sscsi-trcv-bm	10/9	24	1	13
sscsi-tsend-bm	11/10	20	2	13
stetson-p1	32/33	60	> 72000	813
stetson-p2	18/22	37	151	49
stetson-p3	6/4	7	1	8

Figure 9: Comparison of exact hazard-free minimizers (#c - number of cubes in minimum solution, time - run-time in seconds)

HF) and the number of variables that need to be added (for IMPYMIN).

The two minimizers are somewhat orthogonal.

On the one hand, IMPYMIN computes a cover of minimum size, whereas ESPRESSO-HF is not guaranteed to find a minimum cover, but typically does find a cover of very good quality. In particular, ESPRESSO-HF finds always a cover that is at most 3% larger than the minimum cover size. It is worth pointing out that many examples were very positively influenced by our notion of essentials. Quite a few examples can be minimized by *just* the essentials step, resulting in a guaranteed minimum solution, see e.g. *dram-ctrl* and *pe-send-ifc*.

On the other hand, ESPRESSO-HF is typically faster than IMPYMIN. However, since neither tool has been highly optimized for speed, we think it is very important to analyze the intrinsic advantages and disadvantages of the two methods. Intuitively, both methods overcome the three bottlenecks of HFMIN—prime implicant generation, transformation of prime implicants to dhf-prime implicants, and solution of the covering problem—each of which being solved by an algorithm with exponential worst-case behavior. However, the way in which ESPRESSO-HF and IMPYMIN overcome the bottlenecks is very different. Whereas IMPYMIN uses implicit data structures (but still follows the same steps as HFMIN), ESPRESSO-HF follows a very different approach. Thus, the two methods are orthogonal in its approach to overcome the bottlenecks. Moreover, while ESPRESSO-HF is faster than IMPYMIN on all of our examples, this does not mean that this is necessarily true for other examples.

In this context, it is important to note that very often the role data structures like BDDs play in obtaining efficient implementations of CAD algorithms is misunderstood. Using BDDs, many CAD problems can now be solved much faster than before the inception of BDDs. However, the naive approach of taking an existing CAD algorithm and augmenting it with BDDs does not necessarily lead to a good tool (see discussion in [5]). In particular, it is impossible to just augment ESPRESSO-HF or HFMIN with BDDs and get a superb tool. That is why we needed a new theoretical result on the characterization of dhf-prime implicants (cf. Section 4.5) on which our new exact implicit minimizer is based.

		Espresso-HF			Impymin		
name	*i/o*	*#c*	*time(s)*	*#e*	*#c*	*time(s)*	*#v*
cache-ctrl	20/23	99	105	50	97	301	39
dram-ctrl	9/8	22	1	22	22	13	6
pe-send-ifc	12/10	27	1	27	27	16	5
pscsi-ircv	8/7	12	1	12	12	10	3
pscsi-isend	11/10	23	1	23	23	15	6
pscsi-pscsi	16/11	78	11	55	77	105	23
pscsi-tsend	11/10	22	1	22	22	13	4
pscsi-tsend-bm	11/11	23	1	23	23	13	4
sd-control	18/22	35	3	23	34	52	0
sscsi-isend-bm	10/9	22	1	22	22	11	3
sscsi-trcv-bm	10/9	24	1	21	24	13	5
sscsi-tsend-bm	11/10	20	1	20	20	13	4
stetson-p1	32/33	60	21	34	60	813	9
stetson-p2	18/22	37	2	26	37	49	0
stetson-p3	6/4	7	1	7	7	8	1

Figure 10: Comparison of the heuristic minimizer Espresso-HF with the exact minimizer Impymin (#c - number of cubes in solution, time - run-time in seconds, #e - number of essentials, #v - number of added variables)

6.3 Comparison with Rutten's Work

An interesting alternative approach to our new characterization of dhf-prime implicants (cf. Section 4.5) was recently presented by Rutten et al. [28]. His new algorithm to computing dhf-prime implicants is very different from ours. His approach follows a divide-and-conquer paradigm. In particular, the problem is split into three sub-problems with respect to a splitting variable. The first (second, third) sub-problem generates those dhf-prime implicants that have a positive literal (negative literal, don't care-literal) for the splitting variable. The underlying idea why this approach may be efficient is that it allows to determine illegal intersections of privileged cubes already during the splitting phase (see [28] for details), which can significantly reduce the recursion tree and lead fast to terminal cases. In the merging phase of the divide-and-conquer approach, the solutions to the sub-problems are combined.

However, it is worth pointing out that a major difference of our work to Rutten's work is that his approach is *not* based on implicit representations. While Rutten's work is nevertheless very promising, it has not been fully evaluated so far. In particular, he only presented run-times for the computation of dhf-prime implicants of *single-output functions*, i.e. only for functions that are *significantly smaller* than those that can be handled by our method (cf. Section 6.1). Moreover, no results for hazard-free 2-level logic minimization, based on his new approach to computing dhf-prime implicants, were presented.

7 Conclusions

We have presented two new minimization methods for multi-output 2-level hazard-free logic minimization: ESPRESSO-HF, a heuristic method based on ESPRESSO-II, and IMPYMIN, an exact method based on implicit data structures.

Both tools can solve all examples that we available. These include several large examples that could not be minimized by previous methods [13]. In particular both tools can solve examples that cannot be solved by the currently fastest minimizer HFMIN. On examples that can be solved by HFMIN, ESPRESSO-HF and IMPYMIN are typically

[13] In publications on the 3D method (see e.g. [37, 35]), note that several of these examples appear but only *single-output* minimization is performed.

orders of magnitude faster.

Although ESPRESSO-HF is a heuristic minimizer, it almost always obtains an absolute minimum-size cover. ESPRESSO-HF also employs a new method to check for existence of solution that does not need to generate all prime implicants.

IMPYMIN performs exact hazard-free logic minimization nearly as efficiently as synchronous logic minimization by incorporating state-of-the-art techniques for implicit prime generation and implicit set covering solving.

IMPYMIN is based on the new idea of incorporating hazard-freedom constraints within a synchronous function by adding extra inputs. We expect that the proposed technique may very well be applicable to other hazard-free optimization problems, too.

Acknowledgment

The authors would like to thank Olivier Coudert for very helpful discussions and for his immense help with the experiments. The authors would also like to thank Bob Fuhrer for providing his tool HFMIN, and Montek Singh for interesting discussions.

References

[1] M. Benes, S.M. Nowick, and A. Wolfe. A fast asynchronous huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1998.

[2] M. Benes, A. Wolfe, and S.M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *Proceedings of the 17th Conference on Advanced Research in VLSI*. IEEE Computer Society Press, Los Alamitos, CA, 1997.

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

[4] Chou, Beerel, Ginosar, Kol, Myers, Rotem, Stevens, and Yun. Optimizing average-case delay in the technology mapping of domino dual-rail circuits: A case study of an asynchronous instruction length decoding pla. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1998.

[5] O. Coudert. Two-level logic minimization: an overview. *Integration, the VLSI journal*, 17:97–140, 1994.

[6] O. Coudert. Doing two-level logic minimization 100 times faster. In *ACM-SIAM Symposium on Discrete Algorithms*, 1995.

[7] O. Coudert. On solving covering problems. In *Proceedings of the 33rd Design Automation Conference*. ACM, 1996.

[8] O. Coudert and J.C. Madre. New ideas for solving covering problems. In *Proceedings of the 32nd Design Automation Conference*. ACM, 1995.

[9] R.K. Brayton et al. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, 1984.

[10] R.M. Fuhrer, B. Lin, and S.M. Nowick. Symbolic hazard-free minimization and encoding of asynchronous finite state machines. In *1995 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society Press, November 1995.

[11] S.B. Furber, J.D. Garside, S. Temple, J. Liu, P. Day, and N.C. Paver. Amulet2e: An asynchronous embedded controller. In *Async97 Symposium*. ACM, April 1997.

[12] J. Kessels and P. Marston. Design asynchronous standby circuits for a low-power pager. In *Async97 Symposium*. ACM, April 1997.

[13] P. Kudva, G. Gopalakrishnan, and H. Jacobson. A technique for synthesizing distributed burst-mode circuits. In *Proceedings of the 33rd Design Automation Conference*. ACM, 1996.

[14] A. Marshall, B. Coates, and P. Siegel. The design of an asynchronous communications chip. *Design and Test*, June 1994.

[15] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. The design of an asynchronous microprocessor. In *1989 Caltech Conference on Very Large Scale Integration*, 1989.

[16] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on CAD*, CAD-4(3):269–285, July 1985.

[17] S. Minato. Zero-Suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th Design Automation Conference*. ACM, 1993.

[18] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: design of a quasi-delay-insensitive microprocessor. *IEEE Design and Test*, 11(2):50–63, Summer 1994.

[19] L.S. Nielsen and J. Sparso. A low-power asynchronous data path for a fir filter bank. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 197–207. IEEE Computer Society Press, November 1996.

[20] S.M. Nowick and B. Coates. UCLOCK: automated design of high-performance unclocked state machines. In *IEEE International Conference on Computer Design*, pages 434–441, October 1994.

[21] S.M. Nowick, M.E. Dean, D.L. Dill, and M. Horowitz. The design of a high-performance cache controller: a case study in asynchronous synthesis. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 419–427. IEEE Computer Society Press, January 1993.

[22] S.M. Nowick and D.L. Dill. Synthesis of asynchronous state machines using a local clock. In *IEEE International Conference on Computer Design*, pages 192–197. IEEE Computer Society Press, October 1991.

[23] S.M. Nowick and D.L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*, pages 626–630. IEEE Computer Society Press, November 1992.

[24] S.M. Nowick, N.K. Jha, and F. Cheng. Synthesis of asynchronous circuits for stuck-at and robust path delay fault testability. In *Proceedings of VLSI Design 95*, January 1995.

[25] Steven M. Nowick and David L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. *IEEE Transactions on CAD*, CAD-14(8):986–997, August 1995.

[26] R. Rudell and A. Sangiovanni Vincentelli. Multiple valued minimization for PLA optimization. *IEEE Transactions on CAD*, CAD-6(5):727–750, September 1987.

[27] J.W.J.M. Rutten and M.R.C.M. Berkelaar. Improved state assignments for burst mode finite state machines. In *Proceedings of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1997.

[28] J.W.J.M. Rutten and M.A.J. Kolsteren. A divide and conquer strategy for hazard free 2-level logic synthesis. In *International Workshop on Logic Synthesis*, 1997.

[29] T. Sasao. An application of multiple-valued logic to a design of programmable logic arrays. In *Proceedings of Int. Symposium on Multiple-Valued Logic*, 1978.

[30] R.F. Sproull, I.E. Sutherland, and C.E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, 1994.

[31] M. Theobald and S.M. Nowick. An implicit method for hazard-free two-level logic minimization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1998.

[32] M. Theobald, S.M. Nowick, and T. Wu. Espresso-HF: A heuristic hazard-free minimizer for two-level logic. In *Proceedings of the 33rd Design Automation Conference*. ACM, 1996.

[33] S.H. Unger. *Asynchronous Sequential Switching Circuits*. New York: Wiley-Interscience, 1969.

[34] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design and Test of Computers*, 11(2):22–32, Summer 1994.

[35] K. Yun and D.L. Dill. A high-performance asynchronous SCSI controller. In *IEEE International Conference on Computer Design*. IEEE Computer Society Press, October 1995.

[36] Kenneth Y. Yun, Ayoob E. Dooply, Julio Arceo, Peter A. Beerel, and Vida Vakilotojar. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[37] K.Y. Yun, D.L. Dill, and S.M. Nowick. Synthesis of 3D asynchronous state machines. In *IEEE International Conference on Computer Design*, pages 346–350. IEEE Computer Society Press, October 1992.

Appendix

A Background on BDDs and ZBDDs

A.1 BDDs

Binary Decision Diagrams (BDDs) [3] are used to efficiently represent Boolean functions. A BDD of a function f is obtained from the Shannon tree representation of f by reduction rules which (i) identify isomorphic subgraphs and (ii) delete each vertex that has the same left and right children.

Example A.1 In Figure 11a) the Shannon tree of the function $f = ab + c$ is shown. To find the function value for a specific assignment to the variables, one follows the path from the root node to a terminal node, taking the left (right) branch if the corresponding variable is assigned the value 0 (1). The corresponding BDD obtained by above reduction rules is shown in part b) of the figure. Note that the BDD of f is just a compact representation of the Shannon tree of f. In particular, the same algorithm can be used to evaluate the function for an assignment to the variables. □

Important properties of BDDs include canonicity of representation (if the variable ordering is fixed), and the efficiency of binary operators, e.g. the Boolean AND of two functions represented by BDDs can be efficiently computed in time proportional to the product of the number of nodes of the two BDDs.

A.2 ZBDDs

Zero-suppressed BDDs [17] are a variant of BDDs which were introduced to efficiently represent *sets of products*, e.g. the set of prime implicants of a function f. A ZBDD of a set of products is obtained from a tree representation of the set of products by reduction rules which (i) identify isomorphic subgraphs and (ii) delete each vertex whose right children points to 0 (i.e. the empty set). Note that to achieve small representations for sparse sets, the second reduction rule differs from the second reduction rule for BDDs. Another difference from BDDs is that a ZBDD makes decisions based on *literals* instead of *variables*.

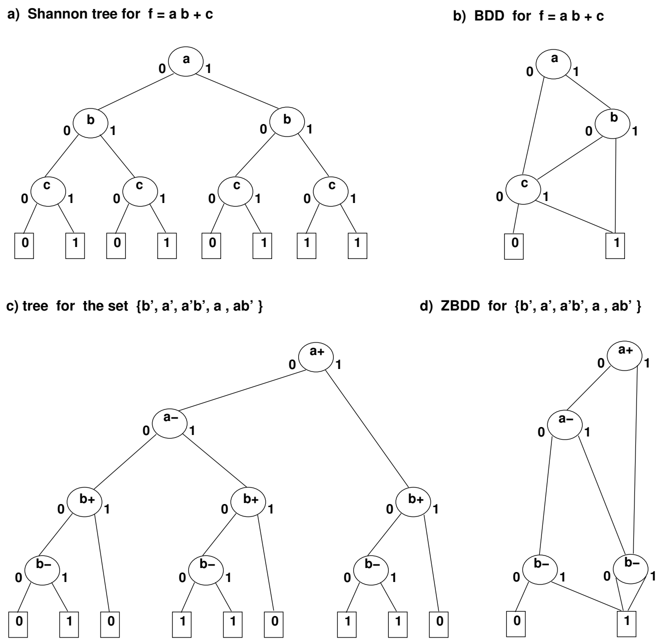

Figure 11: BDDs and ZBDDs

Example A.2 Consider the tree representation of the set of products $\{b', a', a'b', a, ab'\}$ in Figure 11c). Here each path from the root node to a terminal 1 node corresponds to a product in the set. The product consists of those *literals* encountered on taking right branches on the path. Here, positive (negative) literals are denoted by a '+' superscript ('-' superscript). The ZBDD for this set of products obtained by above reduction rules is shown in part d) of the figure. □

Important properties of ZBDDs include canonicity of representation and efficient computation of set-operations, such as union and intersection.

B Implicit 2-Level Logic Minimization: SCHERZO

This section briefly reviews the state-of-the-art synchronous exact two-level logic minimization algorithm, called SCHERZO [8, 6, 5, 7], which forms a basis of our new hazard-free implicit minimization method. Using *implicit* minimization techniques, SCHERZO is 10 to more than 100 times faster than the best previous minimization methods.

SCHERZO has two significant differences from classic minimization algorithms like the well-known Quine-McCluskey algorithm:

- Scherzo uses data structures like BDDs and ZBDDs to represent Boolean functions and sets of products very efficiently (see the Appendix for a review of BDDs and ZBDDs). Thus, the complexity of the minimization problem is shifted, and the cost of the cyclic core computation [14] is independent of the number of products (e.g. the number of prime implicants) that are manipulated.

- Scherzo includes new algorithms that operate on these data structures. The motivation is that the logic minimization problem can be considered as a set covering problem over a lattice. More specifically, both the *covering objects*, P, and the *objects-to-be-covered*, Q, are subsets of the lattice $\mathcal{P}$ of all Boolean products (over the set of literals). A new cyclic core computation algorithm (see below) uses then two endomorphisms τ_P and τ_Q, which operate on Q and P respectively, to capture dominance relations and to compute the fixpoint C, which can be shown to be isomorphic to the cyclic core.

Below is a short description of Scherzo's algorithmic approach [15]. Note that for the understanding of this paper the actual implementation of algorithms is not important. Rather it is of interest which data structures they manipulate and that the algorithms have been very effective in practice.

Algorithm: Scherzo
Input: Boolean function f.
Output: All minimum 2-level implementations of f.

1. Compute the ZBDD $P^{(init)}$ of Prime(f) (the set of all prime implicants of f, or covering objects). Here, f is given as a BDD.

2. Compute the ZBDD $Q^{(init)}$ of the set of ON-set minterms of f, (*i.e.,* the objects to be covered).

3. Solve the implicit set covering problem $\langle Q^{(init)}, P^{(init)}, \subseteq\rangle$ (Note that "$\subseteq$" replaces "$\in$", usually used to describe the relation between the two sorts of objects of a

[14] A set covering problem can be reduced in size by repeated elimination of essential elements and application of dominance relations. The remaining set covering problem (if any) is called the cyclic core.

[15] The ZBDD based recursive algorithms that implement the steps efficiently can be found in [5].

covering problem, since our set covering problem is considered over a lattice, as explained above.)

(a) Determining the cyclic core:

Compute the fixpoint C, which is isomorphic to the cyclic core, produced by the following *rewriting rules* on the implicit set covering problem $\langle Q, P, \subseteq\rangle := \langle Q^{(init)}, P^{(init)}, \subseteq\rangle$,

$$\begin{aligned}
\langle Q, P, \subseteq\rangle &\rightarrow \langle \max_{\subseteq} \tau_P(Q), \max_{\subseteq} \tau_Q(P), \subseteq\rangle \\
\langle Q, P, \subseteq\rangle &\rightarrow \langle Q - E, P - E, \subseteq\rangle, \\
& \quad\;\text{with}\, E = Q \cap P
\end{aligned}$$

where τ_P and τ_Q are defined from $\mathcal{P}$ into $\mathcal{P}$ by:

$$\begin{aligned}
\tau_Q(r) &= \sup_{\subseteq}\{q \in Q | q \subseteq r\} \\
\tau_P(r) &= \inf_{\subseteq}\{p \in P | r \subseteq p\}
\end{aligned}$$

Intuition for τ-operators

To understand the rewriting rules consider first the following examples for *sup* (supremum) and *inf* (infimum): $\sup_{\subseteq}\{x_1x_2, x_2\overline{x_3}\} = x_2$ and $\inf_{\subseteq}\{x_1x_2, x_2\overline{x_3}\} = x_1x_2\overline{x_3}$.

Operator τ_Q maps each product r of the covering objects (initially a prime implicant) onto the supremum of all products (initially on-set minterms) that it covers. Basically, r is mapped onto the smallest cube r' such that r' still covers the same set of products as r. This process often reduces product r.

Operator τ_P maps each product r of the objects-to-be-covered (initially an on-set minterm) onto the infimum of all products (initially prime implicants) by which it is covered. Basically, r is mapped onto the largest cube r' such that r' is still covered by the same set of products as r. This process often enlarges product r.

max removes cubes contained in other cubes. Each non-maximal covering object can be removed since it is included in a “better” cube, i.e. one that covers more. Each non-maximal object-to-be-covered can be removed since the containment in another larger object-to-be-covered ensures its covering.

The intuition behind the τ-operators (together with *max*) is that they are very often not injective, that is, they may reduce the size of the covering problem. Basically, the τ operators capture dominance relations. Also, it can be shown that the *essential elements* [16] (above denoted by E) are those elements that are present in the intersection of P and Q at any iteration.

The rewriting rules for $\langle Q, P, \subseteq \rangle$ are iterated until no change: the fixpoint C is computed, which means that the cyclic core is determined and implicitly represented by Q and P.

(b) Solving the cyclic core:

The resulting fixpoint C is solved using a branch-and-bound method, modified to generate all minimum-cost solutions, and step 3(a).

(c) Solutions to covering problem:

Let F be the union of the sets E found during the computation of the fixpoint C in step 3(a). Let $Sol(C)$ be the set of solutions to C. Then the set of *all* solutions of the 2-level logic minimization of f is:

$$\bigcup_{S \in Sol(C)} \times_{r \in S \cup F} \{p \in P^{(init)} | r \subseteq p\}$$

Intuition: Each $r \in S \cup F$ represents an equivalence class of primes, which is the set of primes that cover r. Each solution includes exactly one of these primes. The Cartesian product therefore gives rise to the set of all solutions.

[16] Essential elements are products that are in every minimum solution