Automatic Technology Mapping for Generalized Fundamental-Mode Asynchronous Designs

Polly Siegel   Giovanni De Micheli   David Dill

Technical Report: CSL-TR-93-XXX
June 1993

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055

Abstract

The generalized fundamental-mode asynchronous design style is one in which the combinational portions of the circuit design are separated from the storage elements, as with synchronous design styles. Synchronous technology mapping techniques can be adapted to work for this asynchronous design style if hazards are taken into account. First, we examine each step of algorithmic technology mapping for its influence on the hazard behavior of the modified network. We then present modifications to an existing synchronous technology mapper to work for this asynchronous design style. We present efficient algorithms for hazard analysis that are used during the mapping process. These algorithms have been implemented and incorporated into the program CERES to produce a technology mapper suitable for asynchronous designs.

Key Words and Phrases: Technology Mapping, Asynchronous Design, Hazard Analysis

Contents

Automatic Technology Mapping for Generalized Fundamental-Mode Asynchronous Designs

Polly Siegel   Giovanni De Micheli   David Dill

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford CA 94305

1 Introduction

Asynchronous design styles have been increasing in popularity as device sizes shrink and concurrency is exploited to increase system performance. However, asynchronous designs are difficult to implement correctly because the presence of *hazards*, which are of no consequence to synchronous systems, can cause improper circuit operation. Many asynchronous design styles, together with accompanying automated synthesis algorithms, address the issues of design complexity and correctness. Typically, these synthesis systems [1, 2, 3] take a high-level description of an asynchronous system and produce a logic-level description of the resultant design that is hazard-free for transitions of interest. The designer then must manually translate this logic-level description into a technology-specific implementation. At this stage, the designer must be careful not to introduce new hazards into the design. The size of designs is limited in part by the inability to safely (and reliably) map the technology-independent description into an implementation.

Automatic technology mapping techniques have been employed over the past decade for synchronous design styles [4, 5, 6]. These algorithms allow translation of a technology-independent logic description into a library-specific (technology-dependent) implementation. However, these techniques by themselves are not suitable for asynchronous design styles because they do not take hazards into account.

In this paper we look at the problem of technology mapping for asynchronous designs. In particular, we concentrate on the *generalized fundamental-mode* asynchronous design style [1], since we can easily separate the combinational portions of the design from the storage elements, as with synchronous design styles. First, we present some background information on fundamental-mode operation, Boolean operations and hazards. In section 3, we examine each step of algorithmic technology mapping for its influence on the hazard behavior of the modified network. We then present modifications to an existing synchronous technology mapper, CERES, to adapt it to work for generalized fundamental-mode designs. In section 4, we present efficient algorithms for hazard analysis that are used by the modified technology mapper during the mapping process. Section 5 presents the results obtained by applying the technology mapper to some benchmark circuits. Finally, section 6 presents conclusions and future work.

2 Definitions

Before delving into the technology mapping problem, it is important to describe the general design style that we are addressing. This section also reviews some terminology related to Boolean algebra and hazards.

2.1 Design Style

Most readers are familiar with single-input change fundamental-mode design styles [7]. In these design styles the input must remain stable until the outputs and feedback variables have settled in response to an input change. Because the operation of the machines is asynchronous, single-input change hazards in the combinational logic can cause incorrect circuit behavior and are thus to be avoided.

Several popular asynchronous design styles extend the single-input change fundamental-mode assumption [7] to allow multiple-input change bursts in a particular state. The synthesis methods [1, 2] that incorporate this burst-mode—or *generalized fundamental-mode*—design style produce logic under the assumption that a burst of input changes can occur in any order and that the outputs and feedback variables of the combinational portion will settle before the next set of input changes are applied. As in the case of single-input change fundamental-mode operation, no hazards can be tolerated during the input bursts. However, in this design style both single-input change hazards and multi-input change hazards must be considered.

Figure 1 shows a simple burst-mode specification, along with the architecture to which it will be mapped by an automatic synthesis method [1]. The output of the automatic synthesis method is a set of technology-independent combinational logic equations along with a set of latches which implement the state machine. The job of the technology mapper is then to implement the combinational portions of this design using, for example, parts from a library of standard cells, in such a way that no hazards are introduced during the mapping process.

This paper addresses the problem of technology mapping for the burst-mode fundamental-mode design style. Because single-input change fundamental-mode is a subset of generalized fundamental-mode, the single-input change case is automatically handled if we take care of the more complicated

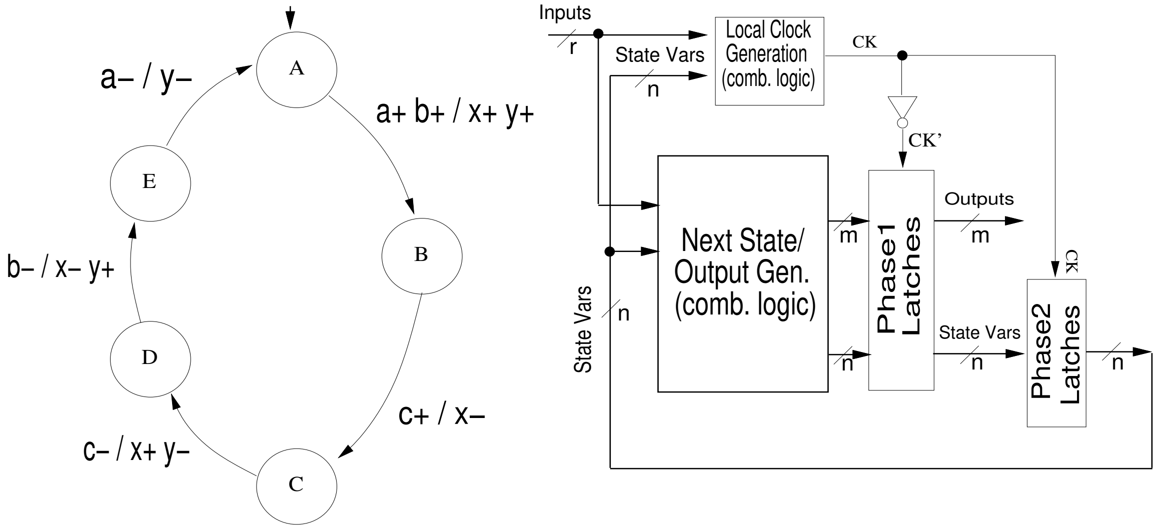

Figure 1: Burst-mode state description and corresponding high-level block diagram.

burst-mode design style.

2.2 Terminology

Although we assume the reader has a basic familiarity with Boolean terminology, for clarity this section defines some terms that appear frequently in the paper. For more detail, please see [8, 7].

A *literal* is an instance of a Boolean variable or its complement. For example, if a is a Boolean variable, in the equation $y = ab + a'$, each occurrence of a is a literal (i.e., a and a').

An *implicant* is a product of literals within a sum-of-products (SOP) expression. The term *cube*, which refers to the mapping of Boolean variables onto an n-cube, is used interchangeably with implicant.

A *prime implicant* is an implicant that is not contained by any other implicant of the function.

A *minterm* is a product of literals that contains all input variables of the function.

A function expressed by a two-level sum-of-products equation can be directly mapped into a two-level gate implementation of AND gates feeding into an OR gate, where the inputs to the AND gates have a one-to-one correspondence with the variables in the product terms. We will use SOP expressions and their two-level gate implementations interchangeably throughout the paper.

2.3 Hazards

A *hazard*, in the most general sense, is an unwanted output glitch in response to a change in some input or inputs. The presence of hazards may cause the design to operate incorrectly. The initial design may or may not be hazard-free, depending upon the manual or automated synthesis

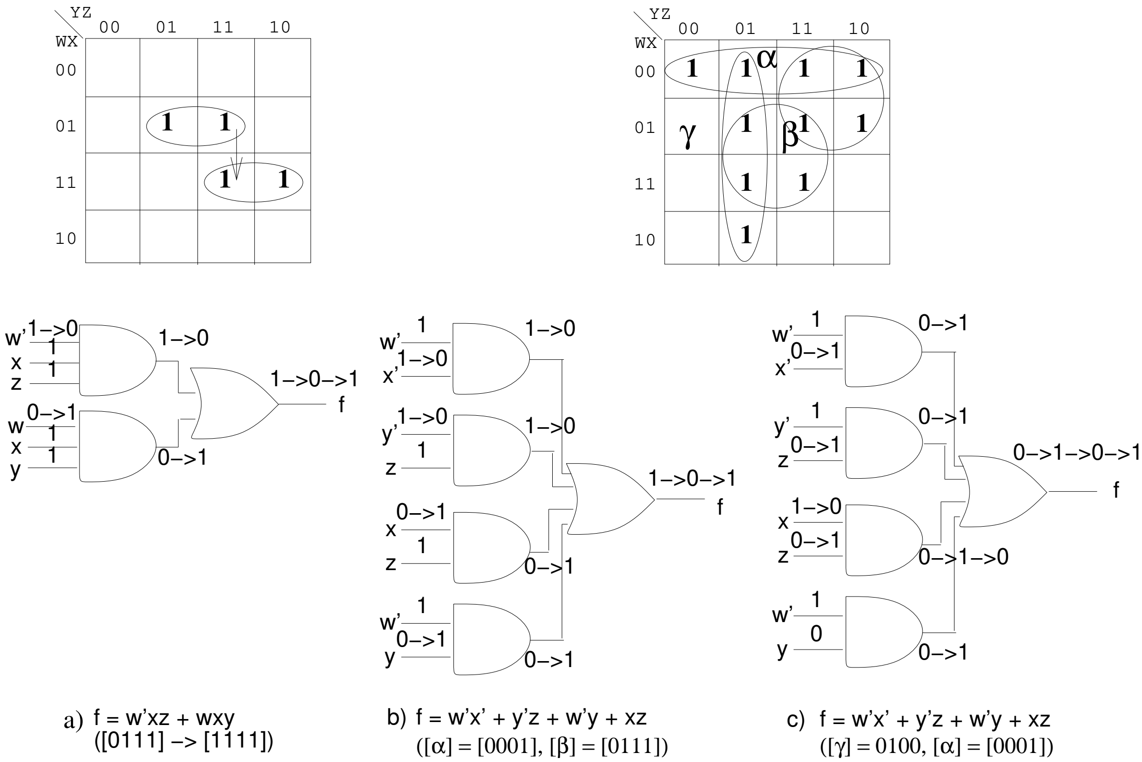

Figure 2: Example of various types of hazards.

procedure used to create it; it is possible that the initial network contains hazards that are never exercised during the circuit's operation. As a result, for most designs it is enough to insure that procedures that manipulate the design do not introduce new hazards.

For the technology mapping problem, we are interested in the hazards present in the combinational portion of both the mapped and unmapped network. For the generalized fundamental-mode design style that we consider in this paper, both single-input change and multi-input change logic hazards are of interest.

There are two basic classes of combinational hazards: *function* and *logic* hazards. Function hazards are a property of the logic function and can only be eliminated through appropriate placement of delay elements, whereas logic hazards are purely a property of the implementation. If a network has a function hazard for a given transition, then it cannot also have a logic hazard for that same transition. Within the class of logic hazards, there are *single-input change* (s.i.c.) hazards and *multi-input change* (m.i.c.) hazards. Finally, each class of hazards (function and logic) includes both *static* and *dynamic* hazards. Given that a transition is being made between two points α and β in the input space $\{0,1\}^n$, static hazards apply to transitions where $f(\alpha) = f(\beta)$, and dynamic hazards apply to cases where $f(\alpha) \neq f(\beta)$.

Static logic hazards occur whenever a transition is not properly covered by a single gate; that is, whenever the implementation does not contain a single gate that maintains the output value throughout the input transition. In Figure 2a, the transition from $w'xyz$ to $wxyz$ in the Boolean space is not covered by a single gate in the implementation. It is thus possible, through some set of gate delays, for both AND gates to be off momentarily, and for the output to make a transition through 0 before settling at its final value, resulting in a static 1-hazard. This problem can be eliminated if an additional AND gate with inputs xyz is added to hold the output high during that transition.

Figure 2b illustrates a multi-input change static logic hazard. During the transition from point α to point β, there is no single gate that holds the output high during the transition. So if gates $w'x'$ and $y'z$ are sufficiently fast and the gates $w'y$ and xz are sufficiently slow, then the output can momentarily take on a 0 value during the transition, resulting in a static hazard. For two-level sum-of-products expressions, it is necessary and sufficient that all prime implicants be included to ensure that there are no multi-input change static logic hazards for any transitions [9]. For multi-level expressions, the conditions are more complex and are discussed in [10].

Dynamic hazards are applicable to both single-input change and multi-input change situations. A dynamic hazard occurs when, during an expected $0 \rightarrow 1$ ($1 \rightarrow 0$) transition of the output, a $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ ($1 \rightarrow 0 \rightarrow 1 \rightarrow 0$) transition occurs. For single-input change conditions, this corresponds to a situation where a literal and its complement fan out to several paths. For multi-input change conditions, a dynamic hazard can occur when a single gate is turned on momentarily during the transition. For example, in Figure 2c a transition between points γ and α can result in gate xz turning on and off before gate $y'z$ turns on, creating a dynamic hazard. Section 4 discusses dynamic hazards and their detection in more detail.

For the generalized fundamental-mode asynchronous design style, it is important to consider all logic hazards, and thus we must make sure that new logic hazards are not introduced during the technology mapping operation.

3 Technology Mapping for Asynchronous Fundamental-Mode Designs

This section describes the approach taken by many recent synchronous algorithmic-based technology mapping programs. In it, we examine each step of the synchronous technology mapper for its effects on the hazard behavior of the transformed network. We then propose modifications to the synchronous technology mapper to allow correct mapping of asynchronous fundamental-mode networks.

The technology mapping step takes as input a technology-independent description of the logic to be implemented. The input to the mapping step typically comes from a logic optimization tool such as MIS [11] in the synchronous case, or an asynchronous logic optimizer [12] in the asynchronous case. Thus, the technology mapping step is primarily concerned with mapping the logic to a technology-specific library in an optimal way, and not with the logic optimization process itself.

To adapt a synchronous technology mapper to the generalized fundamental-mode asynchronous design style, each step in the technology mapping process must be examined and modified to ensure that the step does not introduce new hazards into the network.

3.1 Hazard Analysis of the Technology Mapping Algorithms

The approach to technology mapping taken by heuristic algorithmic technology mappers, such as DAGON, MIS, and CERES, divides the problem into three major steps: decomposition, partitioning, and matching/covering. First, the initial network, which is represented as a directed acyclic graph (DAG), is decomposed into a multi-level network composed of simple gates (e.g., 2-input AND/OR gates or 2-input NAND/NOR gates), whose corresponding representative functions are called *base functions*. Next, the circuit is partitioned into sets of single-output *cones* of logic, where a cone of logic represents a subnetwork of a partition of the network obtained by cutting the network at points of multi-fanout. The mapper then treats each cone independently. All possible matches to library elements are then found for subnetworks within each logic cone. Finally, an optimal set of matching library elements is selected from the set of matches to realize the network.

Thus, the basic procedure is as follows:

```
procedure tmap(network, library) {
    decomposed_network = tech_decomp(network);
    cones = partition(decomposed_network);
    foreach output in cones {
        find_best_cover(output, library);
    }
}
```

Throughout this paper, we use procedure `tmap` and *synchronous mapping procedure* interchangeably.

3.1.1 Decomposition

The decomposition step transforms the network into an equivalent network composed of two-input, one-output base gates. This process can be performed by recursively applying DeMorgan's theorem and the associative law to the network. Both operations have been shown to be hazard-preserving for all logic hazards [13]. Thus, the modified network composed of two-input, one-output gates has identical hazard behavior to that of the original network.

Note that within MIS's technology mapper, some simplification is also done during the decomposition step. This can introduce some static 1-hazards if redundant cubes are eliminated by the simplification algorithm. Therefore, we define a procedure, `async_tech_decomp`, that decomposes a circuit using only the associative and DeMorgan's laws. This procedure must be used by the asynchronous technology mapper during the decomposition step.

3.1.2 Partitioning

The partitioning step breaks the decomposed network at points of multiple fanout into single-output cones of logic. This heuristic simplification is required to convert a multi-output logic network into a collection of single-output cones of logic, so that simpler algorithms can be employed to find the best cover for the network [4, 6]. Given that we start with a hazard-free network and preserve this behavior (within the partitions) in the covering step, the partitioning step does not alter the hazard behavior of the network.

3.1.3 Matching and Covering

The matching and covering steps involve identifying equivalence between a subnetwork and a library element, and replacing that subnetwork with the equivalent library element. This step must not introduce any new hazards into the design.

Different algorithms are used during the matching and covering step. Keutzer [4] and Rudell [5] use tree pattern matching techniques for matching elements within the library to portions of the circuit. MIS generates a complete set of patterns consisting of different decompositions of two-input, one-output gates for each library element. Standard pattern matching techniques are used to compare the library element with a portion of the network to be mapped. As long as the subnetworks represented by the patterns do not have hazards, these techniques work to preserve the hazard behavior of the circuit, provided that the initial circuit decomposition is hazard-preserving and that the library elements do not have hazards. However, if a hazardous library element is selected, then the hazard behavior of the subcircuit must be examined before the match is accepted.

Some mappers, such as CERES, use Boolean techniques to detect equivalent networks. These techniques decouple the structure of the subnetwork from the matching process, which means we cannot reason about the transformations of one subnetwork into the other. However, we can build on theorems presented by Unger in [13] to show that if we replace a portion of the circuit with an equivalent circuit with similar hazard behavior, the resulting circuit is still hazard-free for the transitions of interest.

Figure 3 shows a simple case in which the hazard behavior of the design must be taken into account to get a correct hazard-preserving cover. With Boolean matching, structural information is not taken into account during the mapping process. In this case, the better match from a

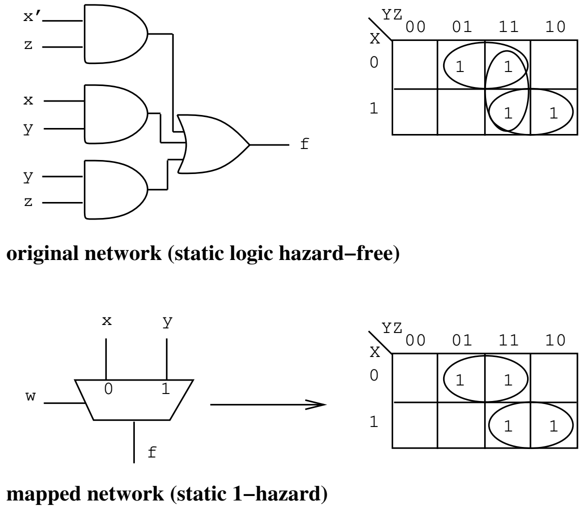

Figure 3: Different structures for the same function can result in different hazard behaviors.

synchronous point of view yields a cover that has more hazards than the original network.

Theorem 3.1 *Given a sum-of-products circuit that has a given hazard behavior, and library elements that consist of AND's, OR's, NAND's, NOR's, and INV's, if we map this circuit with the synchronous mapping procedure to a (possibly multilevel) circuit implemented with the given library elements, the resulting mapped network will have the same hazard behavior as the original network.*

Proof: The decomposition process makes use of DeMorgan's law and the associative laws to transform the initial SOP expression into a multilevel circuit composed of two-input one-output gates. These operations are hazard-preserving, as shown by Unger, so the resulting decomposed network has the same hazard behavior as the original expression. Since replacing any portion of the circuit with a single AND, OR, NAND or NOR gate is equivalent to applying the associative law and/or DeMorgan's law, these operations are also hazard-preserving. Additionally, mapping of the INV to a portion of the circuit cannot introduce new hazards. The covering process simply selects the best set of these mappings that covers the entire network, which is equivalent to successively replacing individual portions of the network with a library element, which we have just shown to be hazard-preserving. Therefore, the resulting mapped network has the same hazard behavior as the original network. □

We can extend this result to include other classes of library elements as well. However, the operations cannot be extended quite so simply when Boolean matching is used. In particular, with Boolean matching we have a problem with static 1-hazards: if a redundant cube is required to eliminate static hazards in the original design, then the matching must not eliminate that redundant cube. Additionally, we may have problems with m.i.c. dynamic hazards.

We must first restate Lemma 4.5 from Unger [13], before we can solve the problem of how to handle more general hazardous elements during the matching step.

Lemma 4.5 (Unger)

A transformation of a circuit C that consists of applying a hazard-preserving transformation to a subcircuit of C is itself hazard-preserving.

This leads to the following key theorem:

Theorem 3.2 *Given a multilevel network with a given hazard behavior, replacement of a subcircuit C by an equivalent subcircuit C^* that contains a subset of the hazards present in C will not introduce new hazards into the network.*

Proof: If C^* has identical logic hazard behavior to C, then for any set of inputs that might cause a hazard in C, the same inputs will cause a hazard in C^* and will have the same effect on the overall network. Quite clearly then, if there is a hazard that is present in C and missing from C^*, and if the signal transition(s) that excited the hazard in C resulted in a hazard that was visible at the outputs of the network, then replacement of C by C^* will eliminate the hazard in the network, since C^* does not have the original hazard. Therefore, the resulting network has a subset of the hazards present in the original network, and thus the replacement does not introduce any new hazards. $\Box$

Even though transforming a network according to theorem 3.2 may eliminate some hazards in the mapped network that were present in the unmapped network, but this will not adversely affect the operation of the network.

The following corollary is a direct result of the theorem.

Corollary 3.1 *Given a logic-hazard-free combinational network, replacement of a subnetwork by a logic-hazard-free equivalent subnetwork will result in a logic-hazard-free network.*

3.2 Modified Technology Mapping Procedure for Generalized Fundamental-Mode Asynchronous Designs

Given the theorems in the previous section, if we start with a set of hazard-free library elements, the covering step and the resulting cover do not introduce new hazards. However, if we have hazardous library elements, we need to make sure that the hazards they contain are a subset of the hazards in the portion of the network that is being matched.

As the starting point for the technology mapping procedure, let us assume that the initial design has no hazards for the transitions of interest. The problem of technology mapping, then, is to map the set of logic equations representing the design to an implementation composed of elements from a specific library, such that the implementation is hazard-free for the transitions of interest.

We can modify our synchronous technology mapping procedure to work for asynchronous fundamental-mode designs (both single- and multi-input change) as follows:

```
procedure async_tmap(network, library) {
    augment_library_with_hazard_info(library);
    decomposed_network = async_tech_decomp(network);
    cones = partition(decomposed_network);
    foreach output in cones {
        find_best_async_cover(output, library);
    }
}
```

The covering routine for `async_tmap` has modifications in the matching routine as follows:

```
async_matching_routine(subckt, library) {
    best_match = nil;
    if (matching_elements = find_matches(library, subckt)) {
        foreach match in matching_elements {
            if (has_hazards(match)) {                    /* library element has hazards */
                if (hazards(match) ⊆ hazards(subckt))
                    accept_match(matching_elements, match);
                else
                    reject_match(matching_elements, match);
            }
            else                                         /* library element is hazard-free */
                accept_match(matching_elements, match);
        }
        best_match = compute_best_match(matching_elements);
    }
    return best_match;
}
```

3.2.1 Representing the Structure of Library Elements

The functionality of each library element is expressed in Boolean factored form (BFF). We use the BFF expression as an accurate and convenient representation for both the functionality and structure of the particular library element. This BFF expression is then analyzed for logic hazards when the library is read into the mapper, and the logic hazard behavior of each library element is added as an annotation to the library element for later use during the matching phase.

For example, the BFF expression for the library element in Figure 4a indicates that the library element is implemented as a sum of two cubes, resulting in a dynamic logic hazard when inputs w and x change with $y = 1$. However, if this element were implemented as shown in Figure 4b, then there would be no dynamic hazard for that input burst.

Note that other representations are possible as long as they adequately express both the structure and the functionality of the library element. Boolean factored form is simply a convenient

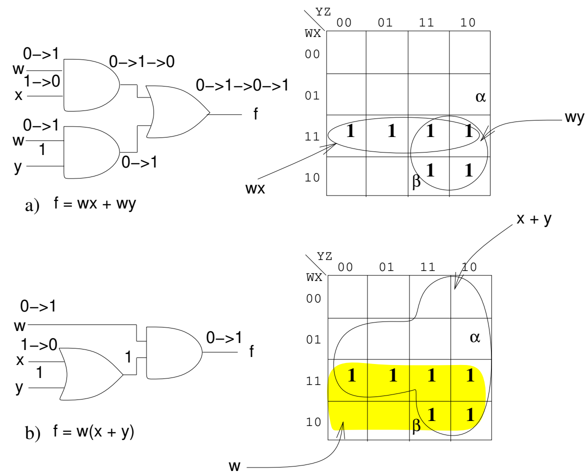

Figure 4: Different structures for the same function can result in different hazard behaviors.

notation which meets these requirements for CMOS circuits. This implies that the structure of each library element must be accurately abstracted from the transistor-level description of the library element, and represented as its equivalent Boolean factored form in the library description.

3.2.2 Modification to the Matching Algorithm

We must now modify the matching algorithm to take the logic hazard information into account. If a hazardous library element is selected as a match for a particular subnetwork, then the subnetwork of interest must be examined to see if the same logic hazards exist. We are not, however, interested in the function hazard behavior of either the library element or the subnetwork, since a function hazard is purely a property of the function itself and thus is independent of implementation.

The procedure for doing the comparison is much easier than the initial hazard analysis of each library element because we already know which transitions are of interest. For each logic hazard in the library element, we must look at the subnetwork to see if the same logic hazard exists. As soon as we find a hazardous transition in the library element that is not in the subnetwork we can stop, because that library element cannot safely be used. At this point the library element is eliminated from consideration as a match for this subnetwork.

We examined some typical commercial standard cell and gate array libraries to see how many elements contained logic hazards. For the standard cell libraries, only the multiplexers, which represented a small fraction of the library, contained hazards. So, for those libraries, most matching elements are logic-hazard-free and the normal synchronous algorithms can be used with negligible overhead.

Table 1 shows the (logic) hazardous elements that are present in several libraries. The LSI and CMOS3 libraries are commercial CMOS ASIC libraries [14]. The GDT library is a CMOS standard-cell library that was produced specifically for a particular chip, and includes many complex AOI gates. For these libraries, the only library elements with logic hazards were multiplexers. Examining a subset of the Actel Act1 library, we found many hazards, primarily in the AOI and OAI gates. (Many of these elements had several instances with different drive capability, for brevity, only one is shown.)

Library	**Hazardous Elements**	**#**	**Total Elements**	**% Hazardous**
LSI9K	Muxes	12	86	14%
CMOS3	Muxes	1	30	3%
GDT	None	0	72	0%
Actel	AOI's,OAI's, Muxes	24	84	29%

Table 1: Libraries and their hazardous elements

The remaining problems we must solve, then, are how to efficiently characterize the hazard behavior of the library elements, and how to easily determine whether the subcircuit has the same hazards. The next section addresses these problems.

4 Hazard Analysis Algorithms

This section describes the hazard analysis algorithms used by the modified technology mapping procedure to characterize both the hazard behavior of the library elements during initialization and the hazard behavior of the subnetwork during the matching and covering process. These analysis algorithms can also be extended to hazard-removal algorithms.

Unless otherwise noted, we present the algorithms for m.i.c. logic hazards, because these algorithms will also identify the s.i.c. hazards.

4.1 Static Logic Hazard Analysis of Combinational Logic

4.1.1 Static Logic 1-Hazard Analysis

From Theorem 4.3 of Unger [13], we know that a multi-level expression can be transformed into a sum-of-products expression in a static hazard-preserving manner using the associative, distributive and DeMorgan laws. Therefore, without loss of generality, we will assume that this has been done and we will work with the two-level SOP form of the expression for static 1-hazard analysis.

For a given function, any missing prime implicants uniquely identify the static logic 1-hazards present in the circuit. Since our goal is not to generate all prime implicants of the function, but to identify the static logic 1-hazards, we must come up with an efficient procedure that doesn't involve prime implicant generation. If all cubes in the network are prime, we can simply identify the cube adjacencies that are not covered. If we encounter a cube that is not prime, we then expand the cube into a prime and add it to the list of cubes to be checked by the adjacency checking algorithm, after having flagged the hazard.

The algorithm works as follows:

```
static_1_analysis(multilevelExpr) {
    SOPexpr = xformTwolevel(multilevelExpr);
    foreach cube in SOPexpr {
        /* Any uncovered non-primes represent hazards ... look at those first */
        if (not prime(cube)) {
            if (prime(cube) not in SOPexpr) {
                addToHazards(primeCube, static1Hazards);
            }
            primeCube = replace_with_prime(cube, SOPexpr);
        }
    }
        /* Generate all cube adjacencies */
    foreach cube1, cube2 in SOPexpr {
        if (numAdjVars(cube1,cube2) == 1) {
            generateAdjCubes(cube1,cube2,cubeAdjacencies);
        }
    }
        /* If the adjacency isn't covered, we have a hazard */
    foreach cube in cubeAdjacencies {
        if (not cubeContainedInExpr(cube, SOPexpr)) {
            addToHazards(cube, static1Hazards);
        }
    }
    return static1Hazards;
}
```

If we only want to test for s.i.c. static logic 1-hazards, the problem is simpler—we need only check that each cube adjacency is covered by some cube in the expression.

Implementation

The data structure we use to represent the logic equation and its cubes is similar to the meta-product structure used in [15]. Two bit-vectors (USED and PHASE) are used to represent each cube, with each bit position in the bit-vector corresponding to a unique variable. For a given cube, if a bit is set in the USED vector, then its corresponding variable appears in the cube. Similarly, for the same cube, for each variable that appears in the vector, the corresponding bit in the PHASE

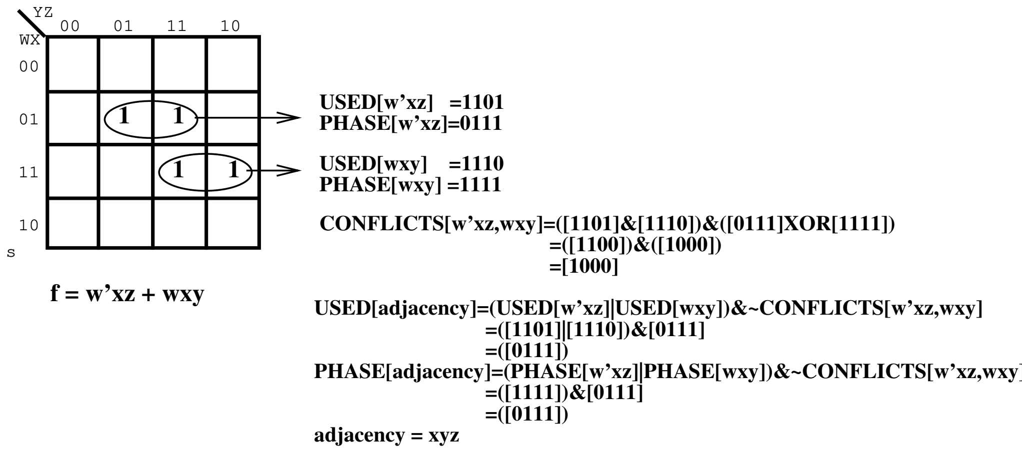

Figure 5: Detection of static 1-hazards.

vector is set if the variable appears in its uncomplemented form in the cube. If the corresponding bit is not set, then the variable appears in its complemented phase. Figure 5 illustrates the mapping of a function to this data structure.

The complexity of generating the list of cube adjacencies is $O(n^2)$ in the number of cubes in the expression. For every pair of cubes in the equation, the cube adjacency is generated by simple bit operations. Obviously, only a small subset of the cube pairs will be found to be adjacent.

Two cubes are adjacent if and only if a single bit is set in the expression:

$$\mathtt{CONFLICTS} = (\mathtt{CUBE1}_{\mathtt{USED}} \& \mathtt{CUBE2}_{\mathtt{USED}}) \& (\mathtt{CUBE1}_{\mathtt{PHASE}} \oplus \mathtt{CUBE2}_{\mathtt{PHASE}})$$

We can decompose this expression to gain some insight. For two cubes to be adjacent, they must share exactly one variable that differs in its phase in the two cubes. So the first part of the expression indicates that variables are present in both cubes. If the two cubes share no variables, then they're not adjacent, so there cannot be a static hazard as a result of those two cubes. The resulting bit vector, then, represents the variable(s) that the two cubes have in common. The second part of the expression represents the variables that differ in phase in the two cubes. The CONFLICTS bit vector will thus have a bit set for each variable that is present in both cubes, but differs in phase between the two. If there is more than one variable that is present in both in differing phases, then the two cubes are not adjacent. If there is only a single bit set in the vector, then the two cubes are adjacent. Figure 5 illustrates the formation of the CONFLICTS vector from the cubes of a function.

Once two cubes have been found to be adjacent, the adjacency is easily generated by generating the OR of the two cubes while masking out the literal that expresses the adjacency. This is done for both the USED and PHASE vectors, as can be seen in the figure.

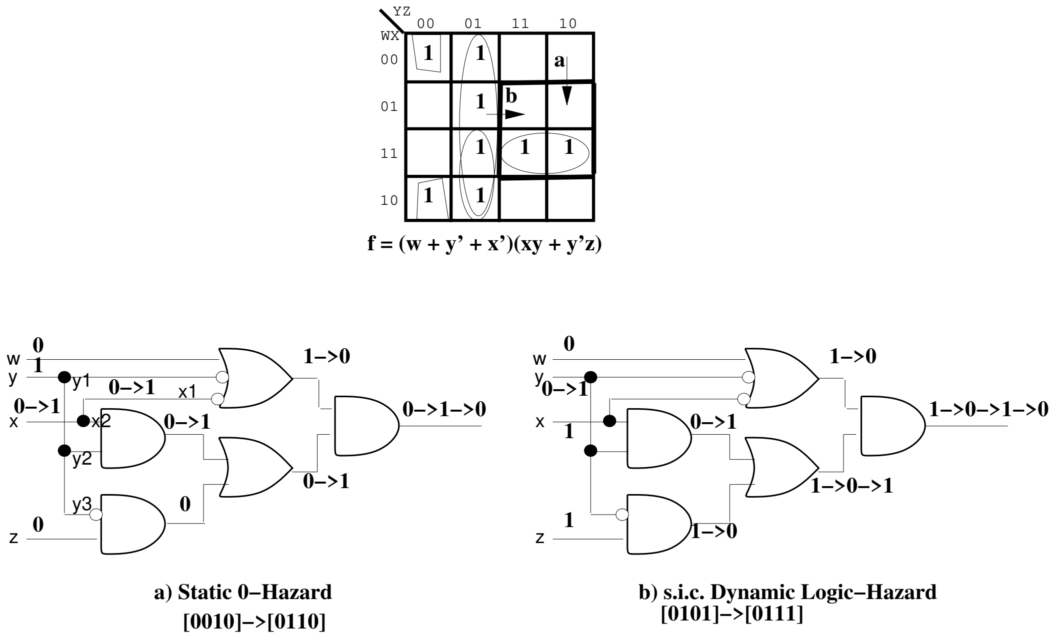

Figure 6: Static 0-hazards and s.i.c. dynamic hazards (from McCluskey,p. 91).

The result of the cube adjacency generation routine is a list of adjacent cubes. Typically, the list of adjacent cubes is much smaller than the $n(n-1)/2$ cube pairs.

4.1.2 Static Logic 0-Hazard Analysis

Static 0-hazards are present when both a product term in the sum-of-products form of an expression contains a vacuous term (i.e. a term that contains a variable and its complement, such as $xx'y$, and thus contributes nothing to the expression in the steady-state), and the circuit can be sensitized to view that term at its outputs. For example, in Figure 6a, a static 0-hazard is present when $w = 0, y = 1, z = 0$, and x is changing.

The detection of static 0-hazards simply requires that the different paths through the circuit be distinguished so as to mark vacuous terms in the SOP form of the expression. These vacuous terms typically represent the reconvergence of a variable and its complement in a multilevel network, which can lead to hazards caused by different delays through the different paths. The detection procedure is a subset of the detection procedure for single-input change dynamic hazards, which is described in the next section.

4.2 Dynamic Logic Hazard Analysis of Combinational Logic

To properly analyze the dynamic logic hazard-behavior of a given multilevel network, it is necessary to uniquely identify the individual paths each signal takes. Past work has used ternary simulation to identify dynamic hazards for specific transitions [9]. However, we are interested in characterizing the dynamic logic hazard behavior of an entire subnetwork or expression, rather than the dynamic logic hazard behavior of a specific input burst. Applying ternary simulation to a network to characterize its dynamic hazard behavior is exponential in the number of input variables, and is thus impractical in the general case. Therefore, we have formulated more efficient hazard analysis techniques for use during the technology mapping process, because hazard analysis is performed frequently during the course of operation.

The remaining subsections focus on the dynamic hazard analysis techniques, starting with the more complicated multi-input change dynamic logic hazard detection procedure, and then describing a simpler procedure for single-input change dynamic logic hazard detection.

4.2.1 Multi-Input Change Dynamic Logic Hazard Analysis of Two-Level Networks

In order to analyze the dynamic logic hazard behavior of a network, we will need to begin with a few definitions.

Definition 4.1 *Let $\alpha, \beta \in \{0,1\}^n$ be points in the input space such that $f(\alpha) = 0$, and $f(\beta) = 1$. Then, a* dynamic hazard *exists if during an input burst which results in a transition between α and β the output goes through a $0 \to 1 \to 0$ transition before settling at 1.*

We must further refine this definition to distinguish between dynamic *function* hazards and dynamic *logic* hazards. Dynamic function hazards are purely a property of the function itself and thus are independent of implementation. We do not need to take dynamic function hazards into account, since, for any acceptable match, both the subcircuit to be mapped and any matching library element have the same function hazards. However, because dynamic logic hazards are a property of the implementation, we must characterize them for all library elements, and for any subnetwork that is matched by a hazardous library element to compare the hazard behaviors.

Definition 4.2 *A* transition space*, $T[\alpha, \beta]$, is the smallest Boolean subspace that contains α and β, where $f(\alpha) = 0$ and $f(\beta) = 1$. (This is described as a* transition subcube *in [16], and is also equivalent to the* supercube(α, β) *[17]).*

Example 4.2.1 In Figure 7, the transition space $T[\alpha, \beta]$ is shown in the non-shaded areas of the Karnaugh maps. Within that transition space, the input variables may change in any order, thus tracing a path between α and β. For example, variables can change in the order $W \uparrow \rightarrow Y \uparrow \rightarrow X \uparrow$ tracing path 1 as shown. Or, the variables could change in the order $Y \uparrow \rightarrow X \uparrow \rightarrow W \uparrow$, depending upon the delays in the network. The first path exercises no hazards, whereas the second path exercises a dynamic logic hazard. A third path, represented by the transitions $X \uparrow \rightarrow W \uparrow \rightarrow Y \uparrow$ excites a dynamic function hazard. Note that this transition space is not function hazard-free, a property which will later be important for our dynamic logic hazard detection algorithm. □

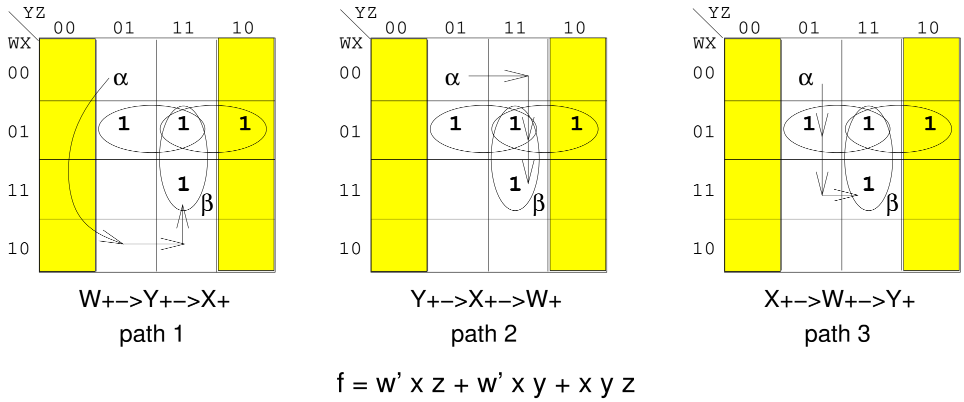

Figure 7: Hazards in transition spaces.

Given these definitions, we can present the necessary conditions for existence of a dynamic logic hazard in a two-level SOP circuit (these were also presented in [16, 18, 9]):

Theorem 4.1 *Given a two-level sum-of-products representation for a circuit, an implementation of a function* f *has a dynamic logic hazard for a given transition $\alpha \rightarrow \beta$, where $\alpha, \beta \in \{0,1\}^n$, if and only if*

1. *There is no function hazard over the transition space $T[\alpha, \beta]$ (where $f(\alpha) = 0, f(\beta) = 1$).*
2. *There exists a cube $c \in f$ that intersects $T[\alpha, \beta]$ but does not contain β.*

Before proving this theorem, let us look at a simple example that illustrates the conditions.

Example 4.2.2 Looking at the transition space $T[\beta, \gamma]$ in the Karnaugh map in Figure 8, we can see that if the order of the transitions is $X \uparrow \rightarrow Z \downarrow \rightarrow Y \uparrow$, then the output will make a $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ transition independent of the implementation of the function, and that therefore a function hazard exists for that transition. The existence of a dynamic function hazard within that

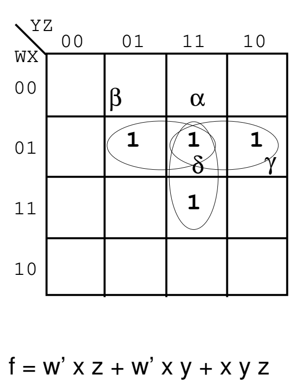

Figure 8: Dynamic hazards within a transition space.

transition space implies that some combination of gate delays can cause that function hazard to be exercised during the input burst, independently of the implementation.

Condition 2 can be seen by examining the transition space $T[\alpha, \gamma]$ in the figure. If the inputs change in the order $X \uparrow \rightarrow Z \downarrow$, then there can be some set of gate delays such that cubes $w'xz$ and xyz turn on and off before cube $w'xy$ turns on, thus yielding a hazardous transition on the output of $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$. (This particular hazard can only be eliminated by implementing the function with a single gate.) Note that if we pick the transition space $T[\beta, \delta]$, then by Condition 2, a dynamic logic hazard does not exist, since there is no cube which intersects $T[\beta, \delta]$ that does not also intersect δ. More intuitively, the first gate which turns on during the transition will hold the output high while the rest of the gates are settling. □

Proof of Theorem 4.1: Condition 1 follows from the definition of a dynamic logic hazard.

Condition 2: $\Rightarrow$ If a dynamic logic hazard exists, then the output must make a transition from $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ during the transition between α and β. For this to happen, at least one cube must make the transition from $0 \rightarrow 1 \rightarrow 0$, and another cube must make the transition from $0 \rightarrow 1$. Since the first cube has value 0 at the endpoint (β), it clearly cannot intersect β. And since that same cube makes a transition to 1 during the change, it must intersect $T[\alpha, \beta]$. Therefore condition 2 is satisfied.

$\Leftarrow$ If there is a cube that intersects the transition space but does not intersect β, then there exists a path starting at β and ending somewhere at the border of the subspace where the intersecting cube makes a $1 \rightarrow 0 \rightarrow 1$ transition. This implies that a $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$ transition will occur during the transition between α and β. This transition will be visible at the output of the circuit if other gates are sufficiently slow. Therefore a dynamic logic hazard exists. □

Let Υ denote the set of transition spaces for a given function f. Let $\Upsilon_{min} \subseteq \Upsilon$ denote the set of minimal *function hazard-free* transition spaces for f, where a minimal transition space is one that is not properly contained by any other transition space for some $\alpha, \beta \in \{0,1\}^n$. We claim that we can detect dynamic hazards by focusing only on the minimal function hazard-free transition spaces, represented by the set Υ_{min}. The following is an outline of the basic steps:

1. Form the set of *minimal function hazard-free* transition spaces,Υ_{min}.

2. For each transition space $T_{min}[\alpha, \beta] \in \Upsilon_{min}$, find the cubes in f that intersect the transition space.

3. If any $T_{min}[\alpha, \beta] \in \Upsilon_{min}$ intersects a cube $c \in f$ that does not also intersect the transition space at β, then a dynamic logic hazard exists for the transition defined by that transition space.

Having observed that any dynamic logic hazard that results from a static logic 1-hazard is fully characterized by the static logic 1-hazard, we can just look for those dynamic logic hazards that are a result of intersecting cubes. The example below illustrates this condition.

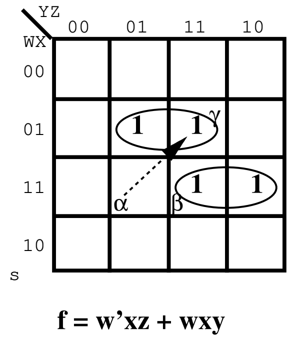

Figure 9: M.i.c. hazard that is the result of a static 1-hazard.

Example 4.2.3 In Figure 9, there is a multi-input change dynamic hazard when traversing from point α to point γ via point β. In that case, the gate corresponding to cube wxy can turn on and off before the output is eventually held high by the gate corresponding to cube $w'xz$. However, this dynamic logic hazard is fully characterized by the static logic 1-hazard which exists in the transition between $wxyz$ and $w'xyz$. $\square$

So, instead of trying to restrict the search space to transition spaces that are function hazard-free, we can start with each cube intersection and form the minimal function hazard-free transition space that contains it, which will give us the dynamic hazards (if any) for that cube intersection.

This leads to the following, more efficient procedure:

procedure findMicDynHaz2level(f) {
  hazards = $\emptyset$;
  I = { irredundant cube intersections of f };
  for all $c \in I$ {   /* complement one variable at a time in c */
    $J_c = \{$cubes adjacent to $c\}$;
    $\alpha_{\mathbf{c}} = \emptyset; \beta_{\mathbf{c}} = \emptyset$;
    for all $d \in J_c$ {   /* Look at each cube adjacent to the cube intersection. */
      if ($f(d) == 0$)   /* If the value of the function at this point is 0, */
        $\alpha_{\mathbf{c}} = \alpha_{\mathbf{c}} \cup d$ /* then the cube belongs to the set α_c, */
      else
        $\beta_{\mathbf{c}} = \beta_{\mathbf{c}} \cup d$ /* otherwise it belongs to the set β_c. */
    }
    /* hazards are defined by all possible pairs of minterms from $\alpha_{\mathbf{c}}$ and $\beta_{\mathbf{c}}$ */
    hazards = hazards $\cup \{T[i,j] \mid i,j \in \alpha_{\mathbf{c}} \times \beta_{\mathbf{c}}\}$;
  }
}

The procedure thus finds the remaining dynamic logic hazards that are necessary for complete characterization of the logic hazard behavior of the subnetwork, as illustrated in the following

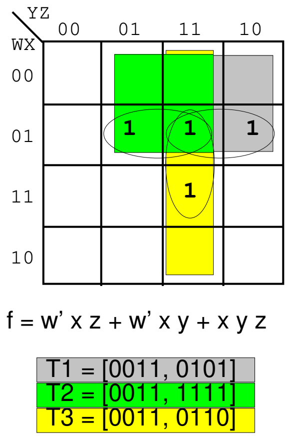

Figure 10: Illustration of procedure `findMicDynHaz2level`.

example.

Example 4.2.4 In Figure 10, the only irredundant cube intersection is $c = w'xyz$. Taking the complement of each *care* variable in this cube we get the set $J_c = \{wxyz, w'x'yz, w'xy'z, w'xyz'\}$. Examining the value of f at each of these locations, we can see that we have one minterm in $\alpha_{\mathbf{c}}$ ($w'x'yz$), and three minterms in $\beta_{\mathbf{c}}$ ($w'xy'z$,$wxyz$, and $w'xyz'$). The minimal function hazard-free transition spaces are then as shown by the shaded areas. Inspection reveals that each of these transition spaces has a dynamic logic hazard. □

We now must show that the set of dynamic logic hazards resulting from this procedure represents all possible dynamic logic hazards that are not the result of a static logic 1-hazard, and that the transition spaces generated by the procedure are function hazard-free. (Note that this second part is not as crucial, because although extra work would be required in the matching step if the procedure produced transition spaces with function hazards, the results would still be correct, since equivalent functions have the same function hazards.)

Theorem 4.2 *Procedure* `findMicDynHaz2level` *finds all possible m.i.c. dynamic logic hazards which are not the result of a static 1-hazard for the function f.*

Proof: Two different cubes must intersect the transition space for a m.i.c. dynamic logic hazard to exist (Theorem 4.1). There are two ways that two cubes can intersect a transition space and not introduce function hazards: two cubes can be adjacent, or they can intersect. In the case where the cubes are adjacent, either the adjacency represents a static logic hazard, which has already been characterized by other algorithms, or the adjacency is covered by another cube that intersects both cubes, in which case they are characterized by the procedure. If two cubes overlap, then the procedure will select them, so we will not miss a possible dynamic logic hazard. Picking minterms adjacent to the intersection ensures that we will satisfy condition 2 of Theorem 4.1—that is, the point β will not intersect the cube intersection. Picking points beyond the adjacencies of the cube intersections as possible candidates for α or β may result in selection of a Boolean space that contains function hazards. More importantly, these spaces contain any transition space, and thus any dynamic hazards, which would be identified by our procedure. □

4.2.2 Multi-Input Change Dynamic Logic Hazard Analysis of Multi-Level Networks

In many cases the library elements and the subnetworks we select during the matching process will be multilevel. (See Figure 4 for an example in which this is a problem.) In this case, we can use the two-level procedure as a filtering process to narrow down the sets of transitions we need to examine for dynamic logic hazards in the multi-level expression, leading to the procedure below:

procedure findMicDynHazMultiLevel:

1. Transform the network into a two-level SOP expression by applying static hazard-preserving transformations.
2. Perform algorithm `findMicDynHaz2level` on the expression generated by step 1.
3. Examine the original multi-level network for dynamic logic hazards on those transitions produced by step 2. Throw away any hazards that aren't found in the multi-level network.

For step 3, we can label the paths as we do for single-input change dynamic hazards and use the transformed two-level expression together with the information we have on the potentially hazardous transitions to identify the real dynamic hazards. We can then eliminate from consideration any hazards that are found to be false hazards. Alternatively, ternary simulation can be used on the specific transitions to eliminate any false hazards.

4.2.3 Single-Input Change Dynamic Logic Hazard Analysis

In the previous subsection we ignored any dynamic logic hazards which are the result of vacuous terms in the two-level expression, as illustrated in Figure 6. We must now address this case.

Given a multi-level network, the network can be transformed into a sum-of-products expression suitable for dynamic hazard analysis by first relabeling the variables so that each distinct path the variable takes is identified, and then transforming the expression into an SOP form through hazard-preserving operations. A single-input change dynamic hazard will be present whenever a variable appears within a product term in both its complemented and uncomplemented forms, and the remaining variables in that product term remain constant, while another product term containing that variable changes from 0 to 1. [13]

Figure 6b shows that there is a dynamic hazard when $w = 0, x = 1, z = 1$ and y is changing. The transformed expression is $f = wx_2y_2 + wy_3'z + y_1'x_2y_2 + y_1'y_3'z + x_1'x_2y_2 + x_1'y_3'z$. With $w = 0, x = z = 1$ the expression reduces to $f = y_1'y_2 + y_1'y_3'$. With y changing, the first term in the reduced expression will make a $0 \rightarrow 1 \rightarrow 0$ transition, while the second term will make a monotonic change, which can result in a dynamic hazard.

The algorithm for detecting the presence of the hazard is straightforward. It involves transforming the literals in the expression to keep track of different paths for each literal, transforming the new expression to a two-level SOP form, and then statically analyzing the expression with simple bit operations to see if any of the necessary conditions apply. Once a variable is identified as a candidate for exciting a dynamic hazard, the rest of the expression can easily be examined to determine whether the dynamic hazard exists. As a by-product of the transformation into two-level SOP form, static 0-hazards are automatically identified by identifying those p-terms that contain pairs of complementary literals.

5 Results

The hazard analysis routines described in the previous section were incorporated into the CERES technology mapping program, and the modifications to the matching algorithms were implemented as described in Section 3.

The technology library is analyzed and annotated with hazard information only when the library is initially read in. Hazards in the subnetwork to be mapped are analyzed only when a hazardous library element is selected as a potential match.

Table 2 compares the run times of the library initialization for both the synchronous mapper and the asynchronous mapper, where the asynchronous mapper must also analyze the library elements for hazards during initialization. The mappers were run on three ASIC libraries and one custom library. The hazard analysis routines did not add appreciable run-time overhead for most libraries. The GDT library took longer to analyze mainly because of the complexity of its library elements.

Library	**Sync**	**Async**	**# Elements**
LSI	.6 sec	1.2 sec	86
Actel	.6 sec	1.1 sec	94
CMOS3	.2 sec	.4 sec	28
GDT	.6 sec	16.7 sec	72

Table 2: Hazard analysis run times for various libraries. All times are user times reported in seconds. All benchmarks were run on a DEC 5000.

In addition to some standard benchmark circuits, two real asynchronous controllers were mapped with the asynchronous mapper, and their results compare favorably against hand-mapped implementations (where available), as can be seen in Table 3. An asynchronous implementation of a SCSI controller was synthesized using the locally-clocked synthesis method [19] and then mapped with the asynchronous mapper. Since the logic equations were never mapped by hand, the hand-mapped entry is empty. The ABCS design is a part of the control logic for an asynchronous infrared communications chip currently under development at Stanford in collaboration with Hewlett-Packard. It was mapped from logic equations synthesized with the 3D synthesis method [2]. The automatically mapped version is less than 13% smaller than the hand-mapped version, where in the table, each transistor in the pulldown network of a cell is assigned an area cost of 1. (The hand-mapped results do not include the cost for buffers which is included in the automatically mapped results.)

Design	**Library**	**How Mapped**	**Cost (area)**	**Time (sec)**
SCSI	LSI	hand-mapped		
		async tmap	168	28.1
ABCS	GDT	hand-mapped	312	–
		async tmap	272	28.1

Table 3: A comparison of automatically-mapped and hand-mapped designs in terms of area (depth of 5). Benchmarks were run on a DEC 5000.

Run times of both the synchronous and asynchronous mapping procedures are shown in Table 4 for mapping the two designs to four different libraries. The asynchronous mapper took from 50%-60% longer than the synchronous mapper in most cases. The overhead is very dependent upon the number of hazardous elements present in the library.

Design	**Mapper**	**Library**			
		Actel	LSI	CMOS3	GDT
SCSI	Synchronous	14.4	17.8	14.0	31.7
	Asynchronous	22.9	28.1	20.7	44.2
ABCS	Synchronous	6.3	8.7	5.7	22.9
	Asynchronous	10.2	13.5	9.0	28.1

Table 4: Comparison between the run-times of the synchronous and asynchronous mappers (depth of 5).

Table 5 shows mapping results and mapping runtimes for some asynchronous benchmark circuits for two libraries. The hazard analysis of each library typically took a only fraction of a second. The area costs are relative to the particular library and will not compare between libraries.

Design	Library					
	CMOS3			**LSI9K**		
	CPU	Delay	Area	CPU	Delay	Area
chu-ad-opt	.6s	24ns	152	3.7s	2.1ns	9
dme-fast-opt	.8s	11.8ns	232	3.8s	1.7ns	13
dme-fast	.6s	11ns	136	3.7s	1.7ns	9
dme-opt	.7s	22.4ns	184	3.9s	2ns	10
dme	.6s	17.8ns	168	3.7s	1.9ns	10
oscsi-ctrl	10.6s	96.5ns	3552	16.1s	7.3ns	172
pe-send-ifc	2.3s	47.2ns	864	5.3s	3.5ns	40
vanbek-opt	.6s	19.5ns	144	3.7s	2.6ns	9
dean-ctrl	33.6s	126ns	11320	43.5s	10.3ns	565
scsi	20.7s	95ns	6888	27.9s	7.2ns	330
abcs	9s	74.7ns	3288	13.3s	6.8ns	168

Table 5: Mapping results for the asynchronous mapper run on various examples (depth of 5). Benchmarks were run on a DEC 5000/240.

6 Conclusions and Future Work

Technology mapping algorithms for synchronous designs can be adapted to work for generalized fundamental-mode asynchronous designs by detecting the hazards in the library elements and using this information to ensure that no new hazards are introduced during the mapping process. The resulting mapped network will then have no new hazards. We proved that in characterizing the hazard behavior of the mapped and unmapped network, only logic hazards need to be taken into account. Based on that information we have created efficient algorithms for hazard detection and incorporated these algorithms into an existing technology mapper to produce an asynchronous technology mapper that works well for generalized fundamental-mode asynchronous designs. A crucial part of the algorithm assumes that the library is properly characterized, and that its hazard behavior is properly expressed through the structure of a Boolean factored form.

We are currently analyzing in more detail the transistor level hazard-behavior of library cells. In particular, we are developing a model for the representation and hazard analysis of pass-transistor networks, such as those employed in MUX-based FPGAs such as the Actel Act2, which do not exhibit the same hazard behavior as complementary CMOS networks. We are also exploring the use of *hazard don't care* information during technology mapping as a means to improve the quality of the mapped circuit.

A more difficult problem is to produce a technology mapper for speed-independent asynchronous circuits. Because the notion of hazards at the logic and implementation levels is different and not as well characterized, we do not expect the synchronous technology mapping procedures to adapt as well to speed-independent design styles. We are currently working on algorithms to address this problem.

7 Acknowledgments

The authors would like to thank to Steve Nowick for many useful discussions related to asynchronous system design and hazards. Frédéric Mailhot wrote the original CERES technology mapper, and was very helpful while it was being adapted for asynchronous design. Ken Yun wrote the 3D synthesis system which was used to generate many of the example circuits.

This work was supported by the Semiconductor Research Corporation, Contract no. 92-DJ-205, and a trust grant from the Center for Integrated Systems, Stanford University.

References

[1] S. M. Nowick and D. L. Dill, "Synthesis of asynchronous state machines using a local clock," in *ICCD, Proceedings of the International Conference on Computer Design*, IEEE Computer Society Press, 1991.

[2] K. Yun and D. Dill, "Automatic synthesis of 3D asynchronous finite-state machines," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 576–580, Nov. 1992.

[3] W. S. Coates, A. L. Davis, and K. S. Stevens, "Automatic synthesis of fast compact self-timed control circuits," in *IFIP Workshop on Asynchronous Circuits*, (Manchester, UK), 1993.

[4] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *24th Design Automation Conference*, pp. 341–347, IEEE/ACM, 1987.

[5] R. Rudell, *Logic Synthesis for VLSI Design.* PhD thesis, U. C. Berkeley, Apr. 1989. Memorandum UCB/ERL M89/49.

[6] F. Mailhot and G. De Micheli, "Algorithms for technology mapping based on binary decision diagrams and on boolean operations," *IEEE Transactions on CAD/ICAS*, in press.

[7] E. J. McCluskey, *Logic Design Principles With Emphasis on Testable Semicustom Circuits.* Prentice-Hall, 1986.

[8] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic, 1984.

[9] E. B. Eichelberger, "Hazard detection in combinational and sequential switching circuits," *IBM Journal*, Mar. 1965.

[10] J. Bredeson, "Synthesis of multiple input change hazard-free combinational switching circuits without feedback," *International Journal of Electronics*, vol. 39, no. 6, pp. 615–624, Dec. 1975.

[11] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Transactions on CAD/ICAS*, vol. 6, no. 6, pp. 1062–1081, Nov. 1987.

[12] S. M. Nowick and D. L. Dill, "Exact two-level minimization of hazard-free logic with multiple-input changes," in *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 626–630, 1992.

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

[14] D. V. Heinbuch, *CMOS3 Cell Library*. Addison-Wesley, 1987.

[15] O. Coudert and J. Madre, "Implicit and incremental computation of primes and essential primes of Boolean functions," in *DAC, Proceedings of the Design Automation Conference*, pp. 36–39, June 1992.

[16] J. Beister, "A unified approach to combinational hazards," *IEEE Transactions on Computers*, vol. 23, no. 6, pp. 566–575, June 1974.

[17] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Transactions on CAD/ICAS*, vol. 6, no. 5, pp. 727–750, Sept. 1987.

[18] J. Bredeson and P. Hulina, "Elimination of static and dynamic hazards for multiple input changes in combinational switching circuits," *Information and Control*, vol. 20, no. 2, pp. 114–124, Mar. 1972.

[19] S. M. Nowick, K. Yun, and D. L. Dill, "Practical asynchronous controller design," in *ICCD, Proceedings of the International Conference on Computer Design*, IEEE Computer Society Press, 1992.