TECHNOLOGY MAPPING FOR CIRCUIT OPTIMIZATION USING CONTENT-ADDRESSABLE MEMORY

by

Joshua M. Lucas

B.S. Computer Engineering, University of Pittsburgh, 2003

Submitted to the Graduate Faculty of

the School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2005

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This thesis was presented

by

Joshua M. Lucas

It was defended on

August 16, 2005

and approved by

Alex K. Jones, Assistant Professor, Electrical and Computer Engineering Department

Raymond R. Hoare, Assistant Professor, Electrical and Computer Engineering Department

Ivan S. Kourtev, Assistant Professor, Electrical and Computer Engineering Department

J.T. Cain, Professor, Electrical and Computer Engineering Department

Thesis Advisor: Alex K. Jones, Assistant Professor, Electrical and Computer Engineering Department

TECHNOLOGY MAPPING FOR CIRCUIT OPTIMIZATION USING CONTENT-ADDRESSABLE MEMORY

Joshua M. Lucas, M.S.

University of Pittsburgh, 2005

The growing complexity of Field Programmable Gate Arrays (FPGA's) is leading to architectures with high input cardinality look-up tables (LUT's). This thesis describes a methodology for area-minimizing technology mapping for combinational logic, specifically designed for such FPGA architectures. This methodology, called LURU, leverages the parallel search capabilities of Content-Addressable Memories (CAM's) to outperform traditional mapping algorithms in both execution time and quality of results. The LURU algorithm is fundamentally different from other techniques for technology mapping in that LURU uses textual string representations of circuit topology in order to efficiently store and search for circuit patterns in a CAM. A circuit is mapped to the target LUT technology using both exact and inexact string matching techniques. Common subcircuit expressions (CSE's) are also identified and used for architectural optimization—a small set of CSE's is shown to effectively cover an average of 96% of the test circuits.

LURU was tested with the ISCAS'85 suite of combinational benchmark circuits and compared with the mapping algorithms FlowMap and CutMap. The area reduction shown by LURU is, on average, 20% better compared to FlowMap and CutMap. The asymptotic runtime complexity of LURU is shown to be better than that of both FlowMap and CutMap.

TABLE OF CONTENTS

1.0 INTRODUCTION 1

1.1 Motivation 1

1.2 Technology Mapping 1

1.3 Content-Addressable Memory 4

1.4 Trends in FPGA Architecture 4

1.5 LURU: A New Approach 5

1.6 Key Contributions 6

2.0 RELATED WORK 8

2.1 Technology Mapping Developments 8

2.2 Circuit Representation Techniques 10

2.3 Applications of Content-Addressable Memory 11

3.0 THEORETICAL FOUNDATIONS 12

3.1 Considerations of Content-Addressable Memory 12

3.2 String Representation of Circuits 13

3.3 The LURU Algorithm 14

3.3.1 LURU Algorithm 14

3.3.2 Inexact Matching Extension 20

3.3.3 LURU Algorithm Complexity 24

4.0 IMPLEMENTATION 26

4.1 Program Functionality 26

4.2 Other Implementation Issues 30

4.2.1 Common Subcircuit Expressions 30

4.2.2 The Search File 30
5.0 RESULTS 32
5.1 Performance of the Exact LURU Technique 32
5.2 Performance of the Inexact LURU Technique 32
5.3 Performance Comparison of the two Techniques 34
6.0 CONCLUSION 39
6.1 Contributions of the Technique 39
6.2 Future Research 40
BIBLIOGRAPHY 41

LIST OF TABLES

1 LUT's required to map the ISCAS'85 circuits. Percentage improvement of LURU (with exact matching) over CutMap is shown in parentheses. 33
2 LUT's required to map the ISCAS'85 circuits. Percentage improvement of LURU (with inexact matching) over CutMap is shown in parentheses. 33
3 Common Subcircuit Expressions, in exact and inexact LURU string forms. . . 36
4 CSE's of partitioned ISCAS'85 benchmark circuits ignoring partitions with a single gate. 37
5 LUT's used in the two phases of LURU's inexact matching extension, shown with percentage of total LUT's for each benchmark ignoring partitions with a single gate. 38

LIST OF FIGURES

1 A synthesis and optimization design flow for digital integrated circuits according to De Micheli [1]. 2

2 FPGA technology mapping design flow using LURU with CAM. 5

3 A simple circuit to illustrate the generation of LURU string representations. . 14

4 Pseudo-code for the LURU algorithm. A list L of LURU strings is generated for a given input circuit, the list is stored into a CAM, and the CAM is searched for subcircuit occurrences. 15

5 The input circuit in (a) is partitioned into subcircuits sc1, sc2, and sc3 in (b) at the point of multiple fanout. Subcircuit sc3 is represented in (c) without the inverter and with generic gate types, as permitted by property P1 in Section 3.1. 16

6 Example circuit to be mapped with LURU. 16

7 Homogeneous LURU circuit topology expressed as an HLS (homogeneous LURU string). 22

8 Example circuit of Fig. 3 with standard LURU and HLS representations. . . . 22

9 Example circuit, similar to that of Fig. 8 (a) standard LURU topology (b) LURU with HLS representation. 23

10 HLS matching the subcircuits of Fig. 8 and Fig. 9. 24

11 Number of CSE's used to cover the ISCAS '85 combinational benchmark circuits (with the coverage shown in Figure 4). 37

1.0 INTRODUCTION

1.1 MOTIVATION

The subject of this thesis is an approach to technology mapping for FPGA's (field programmable gate arrays). The work is motivated by both the state of the art in technology mapping and important trends in FPGA architectures. There is currently no *area-optimal* solution to technology mapping. The best algorithms in use today use heuristics to address the graph-covering problem, known to be NP-hard. Technology mapping is an important phase of design. It has a great effect on both the area requirements and delay characteristics of integrated circuits. Recent trends in FPGA architecture suggest a change in thinking regarding technology mapping. More FPGA's are heterogeneous and include multiple functional units. As technology capabilities improve, FPGA's include more look-up tables (LUT's) with a higher number of inputs.

1.2 TECHNOLOGY MAPPING

Technology mapping is the last in a series of procedures for the synthesis of digital integrated circuits. A design flow for such circuits is shown in Figure 1 [1]. This process begins with *HDL* (hardware description language) descriptions of a system. Such descriptions allow for system modeling at various levels of abstraction, such as the *architectural* and *logic* levels depicted in Figure 1. The task of *scheduling* determines the timing and parallelism of operations in the hardware system [1]. *Binding* is the process of mapping system operations to the hardware resources used by those operations [1]. Binding uses information from the schedul-

ing task, as scheduling places restrictions on resource sharing in the system [1]. *Architectural synthesis* uses scheduling and binding to obtain a resource-bound schedule of operations that satisfies system constraints and is optimal according to some criteria [1]. *Sequential synthesis* translates a control specification (that is, a finite state machine) into a representation consisting of combinational logic and registers [1]. *Combinational synthesis* optimizes the combinational logic of the system using criteria such as speed and area. Finally, technology mapping (also known as *library binding*) maps the combinational logic and registers of the system to a library of hardware elements. For FPGA's, these elements are generally LUT's.

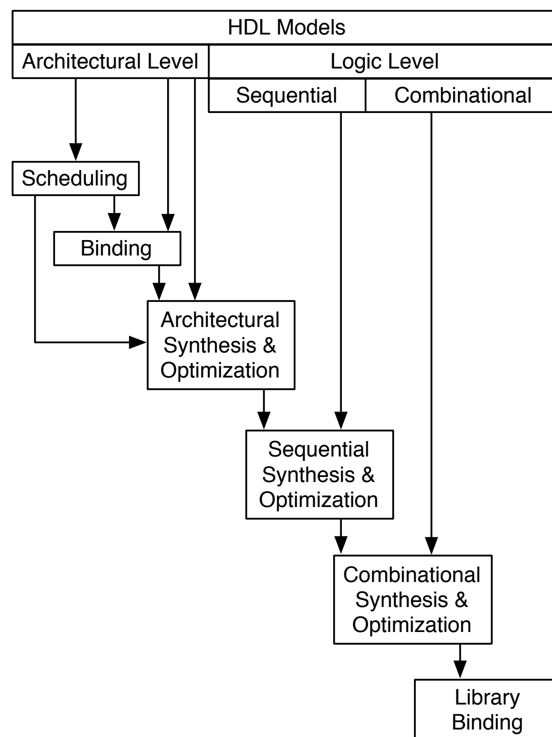


Figure 1: A synthesis and optimization design flow for digital integrated circuits according to De Micheli [1].

Technology mapping is the process of mapping a circuit from a technology-independent form onto hardware elements of a particular technology [2]. Excellent introductions to the topic have been written by DeMicheli [1] as well as Hachtel and Somenzi [3]. Because

hardware elements are often organized as a *library*, this process is also called *library binding* or *cell-library binding*. Techniques for technology mapping can be divided into two broad categories: *rule-based* techniques and *graph-covering* algorithms [3].

Rule-based techniques for technology mapping bind a circuit to a library through stepwise refinement. Local transformations of the circuit, performed according to a database of transformation rules, preserve functionality while improving the efficiency of the mapping. Each transformation effectively replaces a subcircuit with an equivalent subcircuit of library elements that makes best use of the library. Rule-based technology mapping is similar to rule-based Boolean optimization [1].

In the graph-covering paradigm, the circuit (or *subject graph*) is converted to a network of chosen *base functions*. Likewise, each element in the hardware library is converted into a *pattern graph* of the same base functions. The goal of graph-covering algorithms, then, is to find a *cover* - a minimum-cost covering of the subject graph by library elements (pattern graphs). The cost function of graph-covering algorithms could be a sum of the areas of pattern graphs in the cover (in the case of area-optimizing technology mapping) or the critical path delay of the cover (in the case of delay-optimizing technology mapping) [3]. In the *Boolean* approach to graph covering, the circuit graph and the library are represented as Boolean equations. The *structural* approach, on the other hand, uses the graph data structure itself [1]. Graph-covering algorithms use heuristics, as the graph-covering problem is NP-hard [3].

The two approaches have advantages and disadvantages. A rule-based approach may, in fact, be used to combine technology-independent logic optimization and technology mapping into one synthesis phase. However, rule-based systems are tuned to provide locally optimized results, while graph-covering algorithms provide more globally optimal results [3]. Generally, graph-covering algorithms provide more optimal results and have better runtimes. In practice, rule-based techniques are often used to improve the results of graph-covering algorithms [1].

The technology mapping approach outlined in this thesis (called *LURU*) is a graph-covering algorithm. However, as the approach is specifically for FPGA's with LUT's (look-up tables), the *library* consists of all Boolean equations which may be implemented by a LUT.

1.3 CONTENT-ADDRESSABLE MEMORY

A CAM (content-addressable memory) is a hardware search memory implemented in a VLSI circuit. Unlike standard RAM (random access memory), where data is stored in and retrieved from specific addresses, data storage and look-up in CAM is by content. Common applications of CAM are network routing tables and cache memories, among others [4].

In a traditional CAM, the search data must be an exact match of the data stored in the CAM. In a *ternary* CAM, a *mask* may be used to specify matching of only certain data bits [4]. CAM performs a parallel search of its contents in a constant number of cycles. For example, the Micron T-CAM [5] can contain over 16,000 entries with individual masks that can be searched in one clock cycle when data is pipelined and in four cycles using individual searches.

1.4 TRENDS IN FPGA ARCHITECTURE

The capabilities of modern Field Programmable Gate Arrays continue to increase in both logic capacity and performance. The common size of the core logical building block in an FPGA [called a look-up table (LUT)], has until recently been fixed at four (4) inputs and one (1) output. Recently, however, larger LUT's with up to seven (7) inputs have become available [6]. The shift to higher LUT input cardinalities in FPGA's is an important architectural adjustment that permits more complex logic to be implemented using a single LUT. Note that LUT input cardinality may be increased without scaling up the size of the underlying SRAM devices [6], making it possible to improve the performance of critical paths while decreasing area requirements. In order to take advantage of higher input cardinality LUT's, it is important for technology mapping to efficiently map circuits onto these LUT's. Current technology mapping algorithms, while capable of mapping to various LUT cardinalities, are tuned to provide best results for smaller numbers of inputs such as commonly used 4-input LUT's.

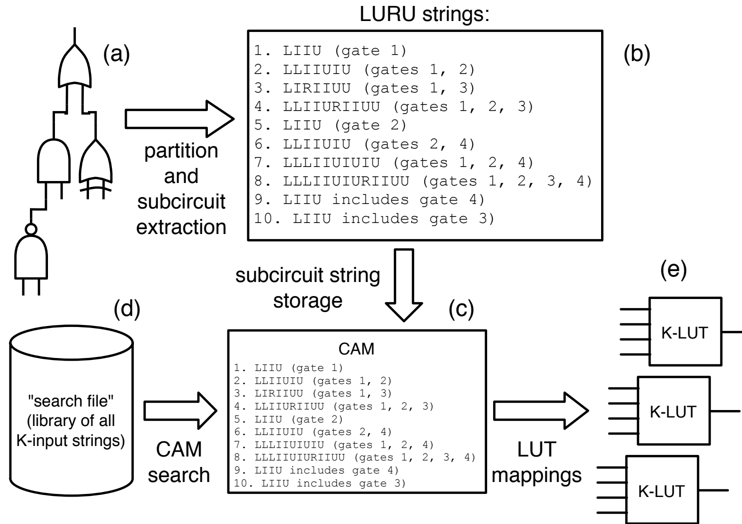


Figure 2: FPGA technology mapping design flow using LURU with CAM.

1.5 LURU: A NEW APPROACH

This thesis describes LURU, a methodology for FPGA technology mapping through the highly parallel search capability provided by CAM. An overview of the LURU flow is shown in Fig. 2. First, the combinational circuit to be mapped is partitioned into a set of subcircuits. The topologies of these subcircuits are then described using textual string representations [note (a) and (b) in Fig. 2]. These string representations need only encode the topology of the circuits because a LUT can compute any Boolean equation with a certain number of inputs. A precomputed set of string representations for the circuit topologies that can be contained in a K-input LUT [illustrated in Fig. 2 (d)] can then be matched against the circuit representation in parallel using the CAM shown in Fig. 2 (c). By using CAM, the search space is increased over traditional technology mapping algorithms. The final mapping is produced for an FPGA device consisting of a network of K-input LUT's as shown in Fig. 2 (e).

Note that CAM's are capable of both exact and inexact matching. Therefore, techniques for technology mapping with both exact matching and inexact matching are proposed and analyzed. Exact matching provides the best mapping density for large cardinality LUT's and is closely related to traditional technology mapping algorithms. Inexact matching takes advantage of frequently occurring *common subcircuit expressions* (CSE's) within a circuit. By optimizing and reusing frequently occurring CSE's, or, by discovering CSE's that match existing architectural features, inexact matching can be used to further improve technology mapping.

Experimental results demonstrate that LURU can improve the quality of mapping results by up to 53%, with an average of 25% improvement over traditional technology mapping techniques such as FlowMap and CutMap [7]. By using inexact matching, it is possible to match approximately 96% of subcircuits to a set of 16 basic CSE's [8].

1.6 KEY CONTRIBUTIONS

The LURU approach to technology mapping offers several additions to the current state of the art in technology mapping. LURU is a dramatic shift from conventional technology mapping algorithms in that it transforms the graph-covering problem into a search problem. Specifically, LURU searches through subcircuits represented as strings. These strings (*LURU strings*) are created via a depth-first search of each subcircuit and encode the *topology* of each subcircuit. As discussed in Section 3.1, the LURU string format permits the circuit to be stored in CAM and permits parallel searches through the circuit in constant time. This string representation is discussed in detail in Sections 3.2 and 3.3. It is used in the exact string matching version of the LURU algorithm, as discussed in Section 3.3.1.

The LURU string representation of subcircuits was generalized through the use of wildcards to create the HLS (homogenous LURU string). Ternary CAMs permit partial string matches based on a mask of wildcards. The HLS format is used in the inexact string matching version of LURU (see the detailed discussion in Section 3.3.2) and allows a ternary CAM to match over 90% of subcircuits with a set of 16 LURU strings, as demonstrated in Section 5.2.

Another significant result of the work is the ability to produce a subcircuit profile for every circuit mapped. This profile is exploited in the LURU technique to identify frequently occurring subcircuits. The profile includes every conceivable subcircuit of the input circuit (that is, it includes all gate combinations that constitute a valid subcircuit). This profiling capability is important for the discovery of circuit patterns for optimization. Frequently occurring subcircuits are encoded as HLSs and are known as Common Subcircuit Expressions (CSEs). Using an inexact matching scheme, a small set of CSEs is used to map a majority of subcircuits. Profiling also allows for identification of CSEs for purposes of circuit optimization. A standard cell implementing a CSE would occur frequently in an implementation. If such a standard cell were optimized for some criterion, the high frequency of the CSE would permit the entire circuit implementation to be improved substantially on that criterion.

The exact string matching version of the LURU technique reduces the LUT requirement over FlowMap and CutMap (by as much as 53%, see Section 5.1). LURU features an algorithmic complexity that scales better than these two algorithms (Section 3.3.3). The inexact string matching version of LURU, which uses CSEs, greatly reduces the number of CAM searches performed by LURU while incurring an increase in area (as much as 17%) over the exact string matching version of LURU (see Section 5.3). The exact version of LURU requires over 5000 CAM searches, while the current inexact version requires just 16 CAM searches.

The remainder of this thesis is organized as follows: Chapter 2 describes work related to and motivating this research. Chapter 3 discusses the LURU technique in detail: Section 3.2 explains the LURU circuit representation while Section 3.3 illustrates the LURU algorithm. Section 3.3.2 describes an inexact matching extension of the technique. The asymptotic complexity of the LURU algorithm is covered in Section 3.3.3. The implementation of LURU is discussed in Chapter 4, which includes a step-by-step description of the program's execution and the details of certain implementation issues. Chapter 5 provides experimental results of the techniques, comparing them with FlowMap and CutMap. Chapter 6 presents conclusions drawn from the work.

2.0 RELATED WORK

2.1 TECHNOLOGY MAPPING DEVELOPMENTS

FPGA technology mapping algorithms have been developed to achieve various objectives: area minimization, depth optimization, routability optimization, and combinations of these [9]. The goal of area minimization, in particular, is to implement a circuit with a minimal number of LUT's. The goal of depth optimization is a solution with minimum delay, and the goal of routability optimization is a solution that simplifies the placement and routing of the implementation. Most of these algorithms use heuristics, as the conventional technology mapping problem for general networks is NP-hard [10].

Keutzer introduced the DAGON technology mapping algorithm [2]. This graph-covering algorithm (recall Section 1.2) reduces the DAG-covering problem by locally and optimally covering divisions of the circuit DAG. The algorithm first partitions the input circuit DAG into trees at gates with multiple fanout. That is, every gate in the DAG with a fanout of greater than one becomes the root node of a tree data structure [2]. A minimal cost mapping to a hardware element (*cell*) in the library is then determined for every tree [2]. The cells (or *tree-covers*) are then formed into a graph structure to implement the functionality of the input circuit [3].

Note that DAGON requires an explicit library of cells. This type of library is required for technology mapping in ASIC (application specific integrated circuit) designs, where a limited set of Boolean equations are implemented in hardware cells. However, FPGA's feature k-input LUT's capable of implementing *any* Boolean equation of k or fewer variables. Thus it is not necessarily desirable to enumerate the entire cell library [1] for FPGA technology mapping. The following two technology mapping algorithms, developed specifically for FPGA's, use an *implicit* library of k-input LUT's.

Cong, *et al.*, developed the FlowMap algorithm for FPGA technology mapping [10]. FlowMap demonstrated that depth minimization in a LUT network can be achieved in polynomial time [11]. A DAG circuit representation is used in FlowMap, where Boolean gates are graph nodes and wires are graph edges. Suppose N_v is a subgraph of the circuit DAG that includes a node v and all predecessors of node v. The label of node v, $l(v)$, is the depth of the depth-optimal K-LUT mapping for N_v. In the first phase of FlowMap, such a label is computed for every node of the DAG. In a second phase, circuit outputs are mapped to LUT's according to the labels (depth-optimal mappings) determined in the first phase. L is the set of circuit outputs to be mapped to LUT's and initially consists of only the primary output nodes of the circuit. The outputs in L are mapped sequentially. Suppose v is an output node in L. A LUT is assigned to implement the Boolean equation of v. L is then updated with new output nodes for those predecessors of v not covered by the assigned LUT. This process continues until L is empty and all nodes of the DAG have been assigned a LUT implementation [10].

The CutMap algorithm is an improvement on FlowMap, guaranteeing depth minimization while using 15% fewer LUTs than FlowMap, on average [12]. The CutMap algorithm is similar to FlowMap. CutMap performs a similar labeling phase, and then computes slack values for each node based on path criticality. The algorithm then iterates through a queue of output nodes from least node slack to most node slack. Suppose v is one such output node. As the node v is processed, a cost is computed for each node in N_v, the subgraph that includes v and the predecessors of v. The costs are used to compute a minimum-cost cut of N_v, which determines how N_v is mapped. As in FlowMap, output nodes are added to the queue for predecessors of v not included in the mapping of N_v [12].

Although LURU is an algorithm for FPGA technology mapping, it uses concepts from DAGON. The input circuit DAG is partitioned into trees, as done in DAGON, and the trees are covered *locally*. However, as discussed in Section 3.3, LURU enumerates *all* conceivable subtrees, thereby achieving a broader scope than that of DAGON. In addition, as seen in Section 4.2.2, the library in LURU is explicitly enumerated for all Boolean equations of eight or fewer variables.

2.2 CIRCUIT REPRESENTATION TECHNIQUES

Note that the computer implementations of mapping algorithms such as FlowMap and CutMap use directed acyclic graph (DAG) representations of circuits. To store DAG's in computer memory, graph nodes (generally corresponding to gates within the circuit) are stored in the RAM memory. Gate connectivity is stored using memory pointers. Pointer use is inherently sequential, thereby requiring that these DAG based mapping algorithms are also sequential in nature.

Many tools for logic synthesis use another circuit representation, the OBDD (ordered binary decision diagram). OBDD's use a tree data structure to store Boolean functions, where tree nodes correspond to input variables of the Boolean function. While OBDD's offer highly efficient Boolean manipulation, they too require the use of memory pointers for computer storage [13].

As discussed in Section 3.1, CAM circuit storage requires a one-dimensional format free of pointer use. The LURU technique presented here uses a new string-based circuit representation discussed thoroughly in Section 3.2.

2.3 APPLICATIONS OF CONTENT-ADDRESSABLE MEMORY

Applications for CAM have been mostly limited to networking applications. For example, CAM has been used to implement a string search engine to speed up network address filtering [14]. This work uses CAM to store 256 workstation addresses. CAM has also been used to implement the memory-intensive components of Asynchronous Transfer Mode (ATM) switch components. A translation table, shared buffer switch, and space switch input buffer are implemented in CAM [15]. CAM has also seen fairly limited utilization outside the networking area. CAM has been used to support a lossless image compression algorithm [16]. This work uses CAM to implement a hash table for a Ziv-Lempel type coder. CAM also shows promise as an efficient database engine [4]. Software based on SQL (Structured Query Language) can interface with CAM to provide for fast database searches.

3.0 THEORETICAL FOUNDATIONS

This chapter details the fundamental ideas that collectively form the LURU technique for technology mapping. Section 3.1 describes how the use of CAM motivates a new, one-dimensional string representation of combinational circuits. Section 3.2 describes this new format in detail. Section 3.3 describes the LURU technology mapping algorithm in detail. This chapter does not discuss implementation issues, which receive their own treatment in the next chapter.

3.1 CONSIDERATIONS OF CONTENT-ADDRESSABLE MEMORY

The use of CAM in LURU is based on a one-dimensional string representation of combinational circuits. The contents of a CAM are a representation of the data in the CAM (not a list of addressable entries) so every CAM entry must be an independently searchable item [4]. Thus, for the textual representation of a circuit to be useful, a string stored in the CAM must uniquely represent the topology of the represented circuit. The challenge of such a representation is to translate circuit topology into a one-dimensional canonical string format.

The constraints of CAM and the needs of technology mapping led to the development of a string circuit format with the following properties:

P1 Boolean gate types are ignored in the LURU string format and only the *topology* of a subcircuit is encoded. The reason for this strategy is that the number of LUT's required to map a subcircuit depends only on the number of primary inputs to this subcircuit.

Circuit *behavior* (the actual Boolean functionality) does not impact the mapping. Single-input gates such as buffers and inverters are ignored because they do not affect the number of primary inputs to a subcircuit.

P2 If x is a subcircuit of a circuit X, then the string representation of x can be found as a substring of the string representation of X. This is because the LURU string for a subcircuit is derived from the depth-first search of a tree. This property facilitates the replacement of one subcircuit with another, as described in Section 3.3.1.

P3 LURU strings are independent and self-contained such that each string contains sufficient information to describe the topology of its corresponding circuit without additional contextual information.

3.2 STRING REPRESENTATION OF CIRCUITS

An example of the construction of a LURU string for a subcircuit via a depth-first search (DFS) is shown in Fig. 3. A DFS starting at the root node in Fig. 3 follows the arrows, recording the direction of traversal as the tree (circuit) is traversed. The traversal is described as follows: '`L`' indicates the left child of a gate (node); '`R`' indicates the right child of a gate; '`U`' indicates upward traversal of the tree (toward the root); and '`I`' indicates a child node that is a primary input signal to the circuit combined with an implicit '`U`.' It is from this textual notation that the name LURU was derived. For gates with more than two inputs, the textual representation is extended to use numerals in addition to the '`L`' and '`R`' children. The first two children are indicated by '`L`' and '`R`,' respectively, while other children are indicated with a numeral corresponding to a left-to-right ordering of children nodes. For example, `LII2IIUIU` describes a circuit consisting of one four-input gate and one two-input gate, with the two-input gate feeding the third input of the four-input gate.

The LURU string representation of the circuit in Fig. 3 is `LLLIIUIURIIUU`. This string is generated by traversing left from the *begin* node and through gate G1 (`L`), then moving left through gate G2 (`LL`), then left again through gate G4 (`LLL`). The two inputs of gate G4 are recorded (`LLLII`), then the traversal continues up from gate G4 (`LLLIIU`). The second input

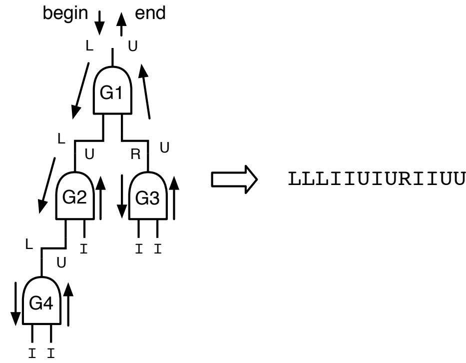


Figure 3: A simple circuit to illustrate the generation of LURU string representations.

of gate G2 is noted (`LLLIIUI`) and the traversal continues up from gate G2 (`LLLIIUIU`), then right through gate G3 (`LLLIIUIUR`) with its two inputs (`LLLIIUIURII`), then up from gate G3 (`LLLIIUIURIIU`). Finally, upward traversal through gate G1 returns to the *end* node, resulting in the overall string `LLLIIUIURIIUU`.

3.3 THE LURU ALGORITHM

3.3.1 LURU Algorithm

The LURU algorithm—shown as pseudo-code in Fig. 4—is conceptually simple and can be summarized as follows. First, a circuit is partitioned into subcircuits and strings representing all subcircuits are generated. These strings are stored into a CAM and represent the set of gate combinations that are candidates for mapping. A pre-generated *search file* containing the LURU strings of all possible circuit topologies that can be implemented by a K-input LUT is used to search against all CAM entries (that is, the entire circuit) in parallel to accomplish the mapping.

A combinational circuit is naturally represented in a tree format where the nodes, edges, and root node of the tree correspond to gates, wires, and the output gate of a subcircuit, respectively. This representation is what requires the partitioning phase of LURU—the input circuit is partitioned into subcircuits such that any gate with a fanout of more than one becomes the root node of a distinct subcircuit tree [2]. This partitioning is illustrated in Fig. 5. Because of property **P1**, inverters are neglected—note the transformation of subcircuit sc3 from Fig. 5 (b) to Fig. 5 (c). Note the generic gates of Fig. 5 (c) as per property **P1**.

```
"BLS" = LURU string for a single gate only
"Active LURU string" = LURU string marked
  to replace 'I' with the BLS of the current gate g
L = an initially empty list of LURU strings
"search file" = file of all LURU strings possible
  for chosen K, sorted by gate count

partition circuit into trees at gate fanout > 1
for each circuit partition p
  perform DFS on p
    g = gate in p currently visited by DFS
    add g's basic LURU string to L,        (step 1)
      marked to replace 'I' with the BLS(s)
      of g's child gate(s)
    for each active LURU string s in L   (step 2)
      if replacement will result in #'I's <= K
        add a copy s' of s to L
        replace 'I' in s' with g's BLS
        if s' now has K inputs
          mark s' as completed
store the list L into the CAM
for each LURU string x in search file
  search for x in the CAM
```

Figure 4: Pseudo-code for the LURU algorithm. A list L of LURU strings is generated for a given input circuit, the list is stored into a CAM, and the CAM is searched for subcircuit occurrences.

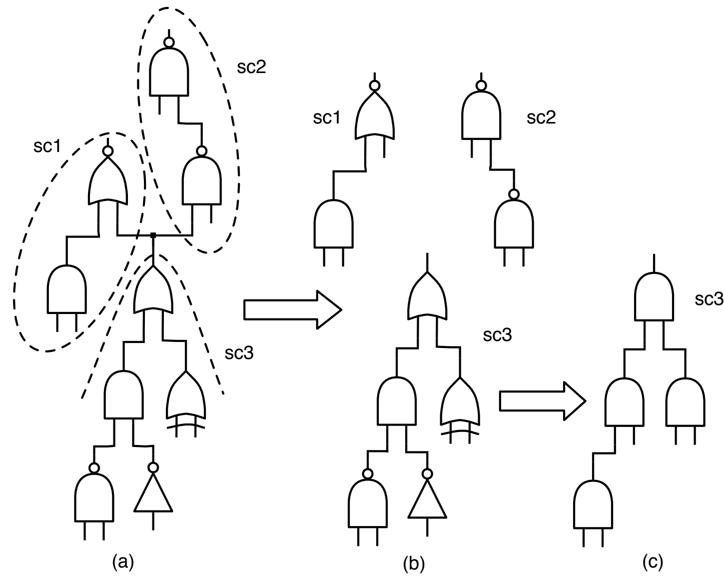


Figure 5: The input circuit in (a) is partitioned into subcircuits sc1, sc2, and sc3 in (b) at the point of multiple fanout. Subcircuit sc3 is represented in (c) without the inverter and with generic gate types, as permitted by property P1 in Section 3.1.

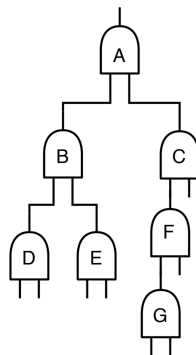


Figure 6: Example circuit to be mapped with LURU.

The LURU algorithm is formally described by the pseudo-code in Fig. 4 and illustrated with the circuit in Fig. 6. Note that for simplicity (and without loss of generality) this circuit does not have any gates with fanout greater than one. Assume that one wishes to map this circuit to LUT's with $K = 4$. A *basic LURU string* (BLS) is defined as a LURU string that describes a single gate. Gate A in Fig. 6 is the root node of the tree (partition p in Fig. 4). Step 1 of Fig. 4 records the BLS of gate A:

1.	`LIIU`	(BC)	[A]

This LURU string describes the subcircuit consisting of gate A (including its two inputs) and constitutes the beginning of a working list of LURU strings. Children (fan-in) gates of gate A (indicated in parentheses) will be used to replace the '`I`'s in string 1. The gate(s) described by a string (in this case gate A) are indicated in brackets to the right.

The DFS visits gate B. Step 1 adds the BLS of gate B to the list. In step B, string 1 is identified as an active LURU string (because B is included in parentheses for string 1). A copy of string 1 is added to the list (string 3 below) and the first '`I`' in the copy is replace with B's BLS:

1.	`LIIU`	(BC)	[A]
2.	`LIIU`	(DE)	[B]
3.	`LLIIUIU`	(DEC)	[AB]

The DFS visits gate D. Step 1 adds D's BLS to the list. Step 2 identifies strings 2 and 3 as active LURU strings, adds copies of them to the list (strings 5 and 6 below), and replaces '`I`' in the copies with D's BLS. The minus signs preceding strings 4 and 6 indicate that these are *completed* strings. Completed strings have no additional opportunity for expansion. String 4 is completed because gate D has no children gates. String 6 is completed because of the choice of LUT input cardinality $K = 4$. This string 6 already has four (4) inputs and any copy-replacement operations would produce invalid strings with too many inputs to fit in a 4-input LUT:

1.	`LIIU`	(BC)	[A]
2.	`LIIU`	(DE)	[B]
3.	`LLIIUIU`	(DEC)	[AB]
4.	`-LIIU`	(00)	[D]
5.	`LLIIUIU`	(00E)	[BD]
6.	`-LLLIIUIUIU`	(00EC)	[ABD]

The DFS then visits gate E. Again, it's BLS is added to the list. Active strings (in this case 2, 3, and 5) are copied and the new BLS is inserted to replace the appropriate "I."

1.	`LIIU`	(BC)	[A]
2.	`-LIIU`	(DE)	[B]
3.	`LLIIUIU`	(DEC)	[AB]
4.	`-LIIU`	(00)	[D]
5.	`-LLIIUIU`	(00E)	[BD]
6.	`-LLLIIUIUIU`	(00EC)	[ABD]
7.	`-LIIU`	(00)	[E]
8.	`LIRIIUU`	(D00)	[BE]
9.	`-LLIRIIUUIU`	(D00C)	[ABE]
10.	`-LLIIURIIUU`	(0000)	[BDE]

At gate C, C's BLS is added to the list. Copy-replacement is performed for active strings 1 and 3:

1.	`LIIU`	(BC)	[A]
2.	`LIIU`	(DE)	[B]
3.	`LLIIUIU`	(DEC)	[AB]
4.	`-LIIU`	(00)	[D]
5.	`LLIIUIU`	(00E)	[BD]
6.	`-LLLIIUIUIU`	(00EC)	[ABD]
7.	`-LIIU`	(00)	[E]
8.	`LIRIIUU`	(D00)	[BE]
9.	`-LLIRIIUUIU`	(D00C)	[ABE]
10.	`-LLIIURIIUU`	(0000)	[BDE]
11.	`LIIU`	(F0)	[C]
12.	`LIRIIUU`	(BF0)	[AC]
13.	`-LLIIURIIUU`	(DEF0)	[ABC]

At gate F, F's BLS is added and copy-replacement is performed for active strings 11 and 12:

1.	`LIIU`	(BC)	[A]
2.	`LIIU`	(DE)	[B]
3.	`LLIIUIU`	(DEC)	[AB]
4.	`-LIIU`	(00)	[D]
5.	`LLIIUIU`	(00E)	[BD]
6.	`-LLLIIUIUIU`	(00EC)	[ABD]
7.	`-LIIU`	(00)	[E]
8.	`LIRIIUU`	(D00)	[BE]
9.	`-LLIRIIUUIU`	(D00C)	[ABE]
10.	`-LLIIURIIUU`	(0000)	[BDE]
11.	`LIIU`	(F0)	[C]
12.	`LIRIIUU`	(BF0)	[AC]
13.	`-LLIIURIIUU`	(DEF0)	[ABC]
14.	`LIIU`	(G0)	[F]
15.	`LLIIUIU`	(G00)	[CF]
16.	`-LIRLIIUIUU`	(BG00)	[ACF]

Finally, the DFS visits gate G where G's BLS is added and copy-replacement is performed for active strings 14 and 15: The DFS subsequently visits gates E, C, F, and G and adds LURU strings in a similar fashion. The final list of LURU strings for the circuit from Fig. 6 is as follows:

1.	`LIIU`	(BC)	[A]
2.	`LIIU`	(DE)	[B]
3.	`LLIIUIU`	(DEC)	[AB]
4.	`-LIIU`	(00)	[D]
5.	`LLIIUIU`	(00E)	[BD]
6.	`-LLLIIUIUIU`	(00EC)	[ABD]
7.	`-LIIU`	(00)	[E]
8.	`LIRIIUU`	(D00)	[BE]
9.	`-LLIRIIUUIU`	(D00C)	[ABE]
10.	`-LLIIURIIUU`	(0000)	[BDE]
11.	`LIIU`	(F0)	[C]
12.	`LIRIIUU`	(BF0)	[AC]
13.	`-LLIIURIIUU`	(DEF0)	[ABC]
14.	`LIIU`	(G0)	[F]
15.	`LLIIUIU`	(G00)	[CF]
16.	`-LIRLIIUIUU`	(BG00)	[ACF]
17.	`-LIIU`	(00)	[G]
18.	`LLIIUIU`	(000)	[FG]
19.	`-LLLIIUIUIU`	(0000)	[CFG]

This final list includes a string representation for every valid subcircuit of the circuit in Fig. 6. This can be verified by checking the gates in brackets in the list above: every gate combination that constitutes a subcircuit of four (4) or fewer inputs is represented. This list, along with lists for other subcircuit partitions of an input circuit, is written into a CAM as illustrated in Fig. 2 (b, c).

The pre-generated *search file* contains the LURU strings for all possible gate combinations than can be implemented with a K-input LUT (*e.g.,* $K = 4$ for the previous example). The strings in the search file are applied sequentially (one at a time) to the CAM, as shown in Fig. 2 (d). The LURU strings in the search file are searched in order from most gates to least gates. This order is easily accomplished by sorting the strings according to their number of '`U`' characters. A LUT is assigned to map each CAM hit, as shown in Fig. 2 (e).

The CAM search technique described above permits a profile to be generated of the frequency of occurrence of a subcircuit within the input circuit. This profile may indicate the dominance of particular subcircuits and help inform circuit designers of possible optimizations. In addition, the profile permits a context for developing inexact CAM searching schemes, as discussed in the next section.

The search file that represents all LURU strings possible for a K-input LUT is built recursively as follows. Two-input LUT topologies are known (the only one being '`LIIU`'). Three-input LUT topologies are generated by inserting copies of the two-input LURU strings into each other at every input location. This insertion results in the three-input strings `LIIIU`, `LLIIUIU`, and `LIRIIUU`. Four-input strings are generated by combining two- and three-input strings and so on. Eventually, all LUT topologies with up to K inputs are generated. A total of 5440 strings are generated and stored in the search file for $K = 8$. Note that the generation occurs only once and imposes no overhead during the execution of the LURU CAM searches.

3.3.2 Inexact Matching Extension

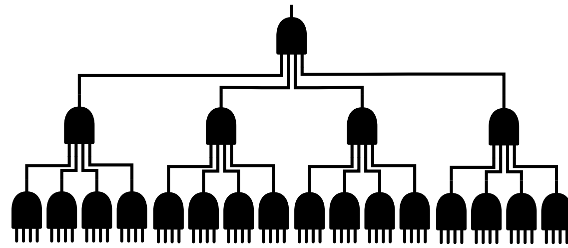
The LURU algorithm described in Section 3.3.1 searches the CAM for the entries in the pre-generated search file, one at a time. This procedure can be time-consuming, because of

the thousands of entries in the search file. As LUT input cardinality increases, it becomes desirable to reduce the number of CAM searches by decreasing the size of the search file, that is, by using a smaller set of LURU strings to search the CAM. However, the randomness of mapped circuits can lead to poor results unless a "good" subset of the search file is chosen. On the other hand, if a given search string matches multiple subcircuit topologies, the corresponding CAM search covers more of the circuit and thus fewer searches are required.

Inexact matching imposes additional requirements on the LURU string format. A standard string length is required for search strings as well as strings stored in the CAM, so that different subcircuits in the CAM can match the same search string. Because the string length is dependent on the topology of a subcircuit, a standard topology of four (4) gate inputs and three (3) logic levels was chosen for all subcircuits. This topology appears in Fig. 7 along with its LURU string representation. This string representation is a *homogeneous LURU string* (HLS).

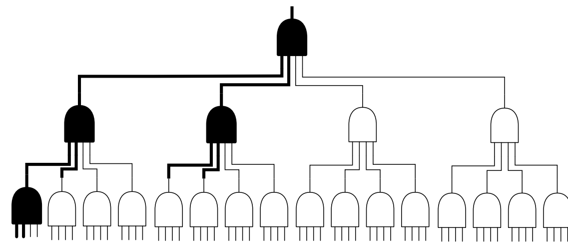
Consider a typical CAM with 32,000 entries of 576 bits each [17]. To support the standard string format described above, three (3) bits per character are required to properly represent the HLS's. Thus, the CAM can support a maximum of $192 = 576/3$ characters per string. Assuming four gate inputs and three logic levels, each of the four children circuits of a subcircuit's output gate requires a maximum of 26 characters. The addition of the initial '`L`' and final '`U`' characters brings the total to 106 characters (as demonstrated in Fig. 7), well within the 192-character CAM limit. The choice of four gate inputs and three logic levels reflects the fact that the majority of logic gates in the ISCAS'85 benchmark suite have four or fewer inputs.

Note that an HLS uses null characters (`0`'s) to represent unused portions of the standard topology. The HLS format handles input signals differently. The input signal representation depends on the logic level (tree level) where the input signal is encountered. An input signal to the topmost-level gate is represented by an '`I`' and 25 nulls. An input at the next level down is represented by an '`I`' and 5 nulls. At the bottom level, a single '`I`' is used. Note that this notation is required to keep a consistent string length. In other words, every subcircuit of the circuit in Fig. 7 will be represented by the constant-length string described above. Fig. 8, for example, shows the HLS of the subcircuit from Fig. 3.



```
L LLIIIIURIIIIU2IIIIU3IIIIUU
  RLIIIIURIIIIU2IIIIU3IIIIUU
  2LIIIIURIIIIU2IIIIU3IIIIUU
  3LIIIIURIIIIU2IIIIU3IIIIUU U
```

Figure 7: Homogeneous LURU circuit topology expressed as an HLS (homogeneous LURU string).



```
LLLIIUIURIIUU
```

```
L LLII00UI00000000000000000U
  RI00000I00000000000000000U
  00000000000000000000000000
  00000000000000000000000000 U
```

Figure 8: Example circuit of Fig. 3 with standard LURU and HLS representations.

(a) (b)

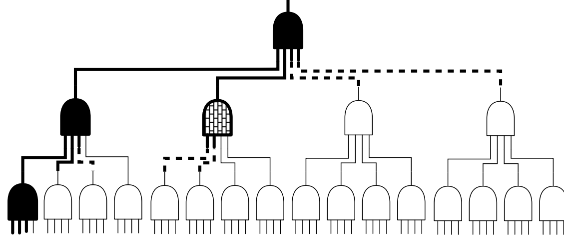
```
LLLIIIUIIUIIIU

L LLIII0UI00000I00000000000U
  I0000000000000000000000000
  I0000000000000000000000000
  I0000000000000000000000000 U
```

Figure 9: Example circuit, similar to that of Fig. 8 (a) standard LURU topology (b) LURU with HLS representation.

Consider the circuit shown in Fig. 9 (a). The LURU string for this circuit is significantly different than the LURU string from Fig. 3. However, the HLS representations shown in Fig. 8 and Fig. 9 (a) are similar. Thus, it is possible to use *wildcards* to construct a string matching both circuits. The string, shown in Fig. 10, is constructed by using wildcards (asterisks) wherever the two strings differ. The first wildcard represents the additional input on the bottom gate of Fig. 9 (a). The second wildcard represents the additional gate input of the middle gate in Fig. 9 (a). The next four wildcards represent the G3 gate of Fig. 3, which is not present in Fig. 9 (a). The final two wildcards represent the additional inputs to the root gate of Fig. 9. In all cases, a wildcard represents a *wildcard* condition.

While the string generation algorithm of the inexact matching extension is nearly identical to that of the standard LURU technique, the CAM search for the extension consists of two separate phases. In the first phase, the set of HLS's representing the input circuit is written into a CAM. Patterns in the small set of CSE's are searched for one at a time in the CAM (note that these are inexact CSE's). For LUT input cardinality $K = 8$, for example,



```
L LLII*0UI00000*00000000000U
  **00000*00000000000000000*
  *0000000000000000000000000
  *0000000000000000000000000 U
```

Figure 10: HLS matching the subcircuits of Fig. 8 and Fig. 9.

the number of inexact searches goes down from 5440 to 16. The coverage of this approach exceeds 90% on average as described in Section 5.

The second phase of the inexact CAM search is required to cover those subcircuits not covered in the first phase. The second phase is required because HLS's can only represent subcircuits with three or fewer levels of logic and four or fewer inputs at their gates. Subcircuits not meeting these criteria generally account for less than 10% of the original circuit and may be mapped using traditional mapping techniques such as FlowMap or CutMap. In this paper, the second phase uses LURU with exact matching.

3.3.3 LURU Algorithm Complexity

The time complexity of LURU is analyzed using the pseudo-code of Fig. 4. Let a given Boolean network have n gates. The maximum input cardinality of the available LUT's is K. As the pseudo-code indicates, the DFS visits each gate in the network twice (upward and downward traversal results in a total of $2n$ gate visits). For each gate visit, a number of strings are added to the list of subcircuits after being copied and undergoing replacement. The number of string additions is limited to K, as K places a limit on the number of valid

subcircuits for mapping to a LUT. By storing LURU strings in a similar format as an HLS, only $K+2$ characters must be changed (*e.g.,* `LIIIIU` for $K=4$). Thus, the time complexity of LURU is $O[2n \times K \times (K+2) + C]$, where C is a constant representing the time required to perform CAM searches. The complexity may be simplified to $O[2nK^2 + C]$. Each CAM search completes in constant time and the required number of CAM searches is independent of n, as described in Section 3.3.1. The time complexity of FlowMap is $O(Kn^2)$ and that of CutMap is $O(2Kn^{\lfloor K/2 \rfloor + 2})$ [11, 12]. In reality, $n \gg K$. Therefore, LURU scales much better than FlowMap or CutMap.

The preceding analysis considers only the time complexity of the LURU algorithm. The efficiency of the algorithm also carries a price of physical CAM hardware. As in the preceding complexity analysis, the number of strings generated is $O[2nK^2]$. For a fixed K, then, the CAM storage requirement is linear in n. Because the algorithm requires that all strings are stored in CAM, the number of required CAM's is $O[2nK^2/M]$, where M is the capacity of a single CAM. Note that there are two alternatives regarding physical CAM use. The first is to use a single CAM to store all strings generated by the algorithm. This option minimizes the CAM space but if the number of strings exceeds the capacity of a single CAM, the CAM must be erased and rewritten with strings not able to be written into the CAM initially. This rewriting of the CAM may need to occur several times, depending on the number of strings generated by the algorithm, and has the drawback of adding memory overhead to the time complexity of the algorithm. The second option, of course, is to use enough CAM devices to store all the generated strings. This option avoids additional time overhead, but is more expensive in terms of hardware.

4.0 IMPLEMENTATION

This section will describe the actual implementation of the LURU technique in detail. To evaluate the propsed LURU algorithm, a simulator was developed for a workstation with a hardware accelerated search memory (CAM). The simulator was built in the Objective-C programming language on an Apple G5 Workstation. The CAM model has 16,000, 576-bit-wide entries following a typical IDT family device [7, 17].

The ISCAS'85 combinational benchmark circuits were partitioned, as described in [2], and used to evaluate the performance of the LURU algorithms. The benchmark circuits were mapped using LURU, FlowMap, and CutMap onto a homogeneous structure of 8-input LUT's. All ISCAS circuits fit into a single CAM. Note, however, that larger circuits that exceed the size of a single CAM can either be depth expanded or computed in two or more sequential steps.

4.1 PROGRAM FUNCTIONALITY

This section discusses the operation and functionality of the LURU implementation. The LURU program accepts four parameters from the user: (1) the ISCAS-format input netlist to be mapped by the algorithm [18], (2) the name of the file that contains all the LUT equations to be searched in the CAM (the "search file"), (3) the name of the file to which a BLIF-format LUT netlist will be written, and (4) the name of the file to which a DOT-format graphics file representing the LUT netlist will be written [19]. After processing these parameters and initializing data structures, the program parses the input netlist.

Each line of the ISCAS-format input file is processed. There are three basic types of lines in the input file [18]. The first type of line, denoted as a *gate* line, contains the following information for a gate of the circuit: unique address, unique name, Boolean type, fanout number, fanin number, and fault information (fault information is unused by LURU). The second type, an *input* line, contains the addresses of nodes that are inputs of the gate described on the previous gate line. The third type, a *fanout* line, is for a special circuit node used by the ISCAS format to specify nodes that have multiple outputs. The line contains the address of the fanout node, its name, the "from" classifier, the name of the gate being fanned out, and fault information. For each node line in the file, the parser creates a new Node object and sets its properties according to those in the file. Nodes listed as inputs of nodes are set as children of the nodes. Each node created is added to an array that constitutes the input network.

The program then determines the actual input and output nodes of the circuit. Input nodes are listed in the input file as type "inpt," while output nodes simply have 0 fanout. Output nodes are added to an array. The nodes in this array will constitute the initial set of root nodes for the trees (SubTrees) which constitute the partitioned circuit. The program then processes the network's "high-fanin" gates (gates with more than K inputs). Such gates cannot be mapped with a K-LUT. So the program removes such gates and replaces them with equivalent networks of gates of K or less inputs. That is, the network is transformed into a K-bounded network.

The program then partitions the network at any gate with multiple fanout. This divides the network into SubTree objects. The program traverses the array of the circuit's output nodes. A depth-first search DFS is performed on the network beginning at each output node. When a gate with more than one fanout is encountered by the DFS, that gate is added to the list of output nodes. That is, the gate becomes the root of a new SubTree. The DFS at each output node x forms a SubTree that is defined as the set of x and x's descendants that have only one fanout. Descendant gates with more than one fanout are excluded from x's SubTree and instead become roots of new SubTrees. In this way, the program forms an array of SubTrees to represent the circuit. The SubTrees do not replicate gates of the input netlist among themselves: each SubTree is an independent portion of the circuit.

Next, the program consolidates SubTrees by resolving “from” nodes. These are nodes of the ISCAS ’85 netlist fle that indicate multiple fanout. Each “from” node has a source node and destination node. The “from” node itself is present merely to indicate a fanout from the source node to the destination node. The program eliminates these “from” nodes from the network, setting source nodes as children of destination nodes.

The program then eliminates inverter gates and buffer gates from the network. These gates do not affect the number of inputs required for a LUT to implement a SubTree, so they are removed from the network. Child gates of such an inverter or buffer are set as children of the parent gate of the inverter or buffer. After this process, some SubTrees (which consisted only of inverters and/or buffers) are now void of gates and are eliminated from the array of SubTrees.

The next major step of the program is to generate SubTree objects for all the possible subcircuits within each SubTree. This step involves exponential growth of data structures, so it is desired to limit this operation to only subcircuits that are realizable with a K-LUT. So before the subcircuit generation occurs, the program further partitions large SubTrees into smaller ones. Any SubTree of greater than K inputs is partitioned. The child gates of the SubTree’s output gate become roots of new SubTrees and the output gate itself is grouped with one of these new SubTrees.

Next the actual generation of subcircuits occurs. This step involves creating an array of SubTree objects (for each original circuit partition, or SubTree) that represents all possible subcircuits within a SubTree. This is done in a recursive manner. Each gate of the SubTree is visited, and a SubTree for that gate is added to an array. Also at each gate visit, any SubTree in the array that contains the parent gate of the current gate is copied and the current gate is added to it. In this way, an array of SubTree objects is built that represents every possible subcircuit within the SubTree.

We now have represented in memory every possible combination of gates that constitutes a valid subcircuit (implementable with a K-LUT). Now the program generates LURU strings via a DFS of each SubTree object. The program simply records the direction of traversal as the DFS proceeds to form the LURU string. Standard LURU strings and homogeneous LURU strings are formed in the same manner, with the exception of input signals ("I" characters, as discussed in Section 3.3.2).

Next the CAM is loaded with all the LURU strings representing subcircuits of the input netlist. The CAM is then searched with a library of LURU strings. In the case of the inexact LURU technique, this library is the set of 16 CSEs. The "CAM" in this implementation is a simple array of SubTrees. A true hardware CAM would store the LURU strings of the circuit. The same is accomplished, at a significant performance penalty, by this array of SubTrees (each SubTree object contains its own LURU string). Each of the 16 CSEs is searched, one at a time, in the CAM. Each CAM "hit" indicates an occurrence in the circuit of one of the CSEs. A LUT object is created for that SubTree and the LUT is added to a network of LUTs that will form the implementation of the circuit. Some gates cannot be covered by CSEs, as they belong only to subcircuits of more than 3 logic levels or subcircuits with gates of more than 4 inputs. That is, such subcircuits cannot be represented with an HLS. Additionally, there may be gates that belong to representable subcircuits that are not mapped simply because the CSEs did not cover a particular subcircuit pattern. Any uncovered gates from this first CAM search are covered in a second CAM search. This CAM search simply searches the CAM with the search file. The search file contains LURU strings for all conceivable K-LUT-implementable subcircuits. So this second CAM search covers the remaining uncovered gates. As before, each CAM hit constitutes a LUT and each LUT is added to the LUT network. The program then prints the LUT network as a DOT graphics file and prints to the screen the number of LUTs required to implement the circuit.

4.2 OTHER IMPLEMENTATION ISSUES

4.2.1 Common Subcircuit Expressions

Common Subcircuit Expressions (CSE's) represent one or more subcircuits that occur frequently in benchmark circuits and can be represented with HLS's. The CSE's in this work were obtained through analysis of the ISCAS'85 circuit benchmarks by storing the frequencies of matches for each LURU string as the exact LURU algorithm proceeded. Intuitively, the longest strings (that is, the largest subcircuits) in each partition are expected to have the greatest mapping impact and are selected to be CSE's. A small set of CSE's was selected and run on the benchmarks. Whenever better coverage was possible with an additional HLS, that string was either added as a CSE or an existing similar CSE was modified with wildcards in the manner of Section 3.3.2 to match the new HLS. In this way, the CSE set was created to cover as many subcircuits as possible while keeping the set small. As noted before, 16 CSE's were found to cover over 90% of subcircuits.

4.2.2 The Search File

The search file that represents all LURU strings possible for a K-input LUT is built recursively as follows. Two-input LUT topologies are known (the only one being '`LIIU`'). Three-input LUT topologies are generated by inserting copies of the two-input LURU strings into each other at every input location. This insertion results in the three-input strings `LIIIU`, `LLIIUIU`, and `LIRIIUU`. Four-input strings are generated by combining two- and three-input strings and so on. Eventually, all LUT topologies with up to K inputs are generated. A total of 5440 strings are generated and stored in the search file for $K = 8$. Note that the generation occurs only once and imposes no overhead during the execution of the LURU CAM searches.

The search file that represents all LURU strings possible for a K-input LUT is built from the bottom up. Two-input LUT topologies are known (the only one being "LIIU"). Three-input LUT topologies are generated by inserting copies of the two-input LURU strings into each other at every input location. This insertion results in the three-input strings LIIIU,

LLIIUIU, and LIRIIUU. Four-input strings are generated by combining two- and three-input strings in the same manner. In the same way, all LUT topologies through K inputs are generated. A total of 5440 strings are generated and stored in the search file for K=8. In general, the number of strings generated grows exponentially with K. However, the generation occurs only once, so the number of LURU strings in the file is constant for all circuits. In turn, the number of CAM searches is constant for all circuits.

5.0 RESULTS

LURU was evaluated with the implementation discussed in Chapter 4. Partitioned version of the ISCAS'85 combinational benchmark circuits were used to measure the effectiveness of both the exact and inexact string matching LURU techniques. Results for both techniques follow, as well as a comparison of the two techniques.

5.1 PERFORMANCE OF THE EXACT LURU TECHNIQUE

Area results in terms of the number LUT's used in mapping the ISCAS'85 circuits with the exact matching version of LURU are presented in Table 1. The circuits were mapped using LURU, FlowMap, and CutMap onto a homogeneous structure of 8-input LUT's. Note that, in all cases, LURU requires fewer LUT's than FlowMap or CutMap to implement a circuit. The average area savings over CutMap is 25%, with a maximum reduction of 53%.

5.2 PERFORMANCE OF THE INEXACT LURU TECHNIQUE

The inexact string matching version of LURU performed similarly to the exact version. Table 2 shows the area results for the ISCAS'85 benchmarks using the inexact version of LURU. The inexact technique requires equal or fewer LUT's than FlowMap and CutMap. However, the inexact technique does not perform as well as the exact technique with regard to area. This disadvantage presents an interesting tradeoff discussed in Section 5.3.

Table 1: LUT's required to map the ISCAS'85 circuits. Percentage improvement of LURU (with exact matching) over CutMap is shown in parentheses.

	FlowMap	CutMap	LURU (exact)
c432	75	75	57 (24%)
c499	66	66	64 (3%)
c880	140	135	129 (4%)
c1355	266	266	264 (1%)
c1908	395	395	206 (48%)
c2670	607	597	283 (53%)
c3540	813	773	590 (24%)
c5315	1232	1182	656 (45%)
c6288	1456	1456	1440 (1%)
c7552	1626	1554	867 (44%)

Table 2: LUT's required to map the ISCAS'85 circuits. Percentage improvement of LURU (with inexact matching) over CutMap is shown in parentheses.

	FlowMap	CutMap	LURU (inexact)
c432	75	75	57 (24%)
c499	66	66	64 (3%)
c880	140	135	135 (0%)
c1355	266	266	264 (1%)
c1908	395	395	203 (49%)
c2670	607	597	324 (46%)
c3540	813	773	600 (22%)
c5315	1232	1182	768 (35%)
c6288	1456	1456	1440 (1%)
c7552	1626	1554	1000 (36%)

The CSE's used for the inexact string matching are presented in Table 3. These 16 LURU strings were constructed based on frequency of subcircuits in the ISCAS'85 benchmark set, as detailed in Section 4.2.

Usage statistics for the 16 CSE's appear in Table 4. This table is helpful in determining which CSE's are most important for covering the subcircuits of each benchmark. Each column of Table 4 corresponds to a benchmark circuit, and each row is a particular CSE. The numbers in the cells indicate how many times a CSE was used in the coverage of a benchmark. Numbers in parentheses show what percentage of the total input circuit was covered with each CSE. Blank fields indicate that the CSE in question did not appear in that circuit. Each column of the table represents a profile of its benchmark circuit, showing the occurrence of various subcircuit patterns in the circuit. It can be seen in Table 4 that certain CSE's have greater matching significance than others (providing higher coverage). The information in Table 4 is an important result of LURU's inexact matching extension and permits designers to quickly identify important subcircuits. If designers wish to optimize a design for speed or power consumption, they know where to focus their optimization effort. For example, optimizations of CSE 6 in Table 4 will benefit 32% of subcircuits in c6288.

5.3 PERFORMANCE COMPARISON OF THE TWO TECHNIQUES

LURU's inexact matching scheme requires a nominal increase in the number of LUT's (compared to LURU). This disadvantage must be balanced against the significant gain of obtaining a small, robust set of optimized search strings (CSE's) rather than searching through all possible topologies for a K-input LUT, as in the exact LURU technique. The potential circuit optimization opportunities provided by inexact matching with CSE's outweigh this relatively minor area increase over exact LURU.

Consider the case in which an optimized CSE implementation provides an $N\%$ improvement in a metric such as area or power versus a LUT. It can be computed using averages obtained from tables 4 and 2 that direct CSE implementations (*e.g.,* standard cells) would need to provide an average improvement of $N = 13.5\%$ over LUT's for the inexact LURU extension to overcome its slight area disadvantage and match standard LURU.

The coverage efficiency of CSE's is due largely to inexact string matching. Figure 11 shows the number of CSE's used in the coverage of the ISCAS '85 circuits. The number of CSE's used in inexact string matching is the same as the numbers in Table 4. The number of CSE's used in exact matching is a result of wildcards - CSE's containing wildcards are generalized representations of several possible subcircuits. A library of all such represented subcircuits may be used to cover each benchmark (with exact string matching) with the same coverage as that achieved with inexact string matching. Note that in Figure 11, the coverage achieved is that shown in the bottom row of Table 4.

The first phase of the inexact LURU technique, which uses CSE's, matches over 90% of subcircuits on average. The second phase, which uses exact LURU, must cover only a few subcircuits not covered in the first phase. Table 5 shows the number of LUT's used in the two phases of the inexact LURU algorithm. The numbers for the first phase (1) are identical to those of Table 4 because each of the LUT's represents one of the 16 CSE's. The numbers for the second phase (2) result from mapping the remaining subcircuits, such as those containing gates with more than four inputs, that do not fit into an HLS. An exact LURU approach requires 5440 CAM searches (for $K = 8$) to map 100% of subcircuits, but the inexact LURU technique requires just 16 CAM searches to map 92.6% of subcircuits (on average). In terms of coverage per search, the inexact LURU technique is more efficient than the exact LURU technique.

Table 3: Common Subcircuit Expressions, in exact and inexact LURU string forms.

0	LLIRIIU2IIU3IIUUIU	8	LIRLIIUIUU
	LLI00000RII00U2II00U3II00UUI0000000000000000000000000 0000000000000000000000000000000000000000000000000000U		LI0000000000000000000000000RLII00UI00000000000000000U 0000000000000000000000000000000000000000000000000000U
1	LLIIURIIU2IIUIU	9	LIRIRIIUUU
	LLI00000I00000000000000000URI00000I00000000000000000U 2I00000I00000000000000000UI0000000000000000000000000U		LI0000000000000000000000000RI00000RII00U000000000000U 0000000000000000000000000000000000000000000000000000U
2	LIII3IIIIUU/LIII3IIIUU/LIII3IIUU/etc.	10	LIRIIU2IIIUU/LIRIIU2IIUU
	LI0000000000000000000000000I0000000000000000000000000 I00000000000000000000000003I00000I00000*00000*00000UU		LI0000000000000000000000000RI00000I00000000000000000U 2I00000I00000*00000000000U00000000000000000000000000U
3	LIRIIUU/LIRIIIUU/LIRIIUIU/LIRIIIUIU/etc.	11	LLIIURLIIURIIUUU
	LI0000000000000000000000000RI00000I00000*00000000000U *0000000000000000000000000*0000000000000000000000000U		LLI00000I00000000000000000URLII00URII00U000000000000U 0000000000000000000000000000000000000000000000000000U
4	LLIIURIRIIUUU	12	LLIRIIIIUUIU
	LLI00000I00000000000000000URI00000RII00U000000000000U 0000000000000000000000000000000000000000000000000000U		LLI00000RIIIIU000000000000UI0000000000000000000000000 0000000000000000000000000000000000000000000000000000U
5	LLIIUIIU/LLIIIUIIIU/etc.	13	LLLIIURIIUUIIIU/etc.
	LL*00000*00000*00000*00000U*0000000000000000000000000 *0000000000000000000000000*0000000000000000000000000U		LLLII00URII00U*00000*00000U*0000000000000000000000000 *0000000000000000000000000*0000000000000000000000000U
6	LLIIURIIUU/LLIIIURIIIUU/LLIIURIIUIIU/etc.	14	LLLIIUIIIUIU/etc.
	LI00000I00000*00000000000URI00000I00000*00000000000U *0000000000000000000000000*0000000000000000000000000U		LLLII00U*00000*00000*00000U*0000000000000000000000000 *0000000000000000000000000*0000000000000000000000000U
7	LIRLIIURIIUUIIU/etc.	15	LII2IIUIU/etc.
	LI0000000000000000000000000RLII00URII00U*00000*00000U *0000000000000000000000000*0000000000000000000000000U		LI0000000000000000000000000I0000000000000000000000000 2I00000I00000*00000*00000U*0000000000000000000000000U

Table 4: CSE's of partitioned ISCAS'85 benchmark circuits ignoring partitions with a single gate.

CSE	c432	c499	c880	c1355	c1908	c2670	c3540	c5315	c6288	c7552
0	1 (3.4%)									
1	8 (27.6%)					1 (0.4%)	2 (0.9%)			
2	2 (6.9%)					1 (0.4%)				
3	9 (31%)	32 (64%)	7 (8.5%)			8 (3.6%)	50 (22.1%)	33 (7.8%)		71 (10.2%)
4	9 (31%)		8 (9.8%)							1 (0.1%)
5		2 (4%)	3 (3.7%)	2 (1.9%)	7 (5.8%)	52 (23.2%)	29 (12.8%)	71 (16.8%)	16 (3.3%)	41 (5.9%)
6		8 (16%)	26 (31.7%)	104 (98.1%)	96 (80%)	114 (50.9%)	63 (27.9%)	236 (55.9%)	464 (96.7%)	417 (60%)
7		8 (16%)	4 (4.9%)			1 (0.4%)	4 (1.8%)			
8			8 (9.8%)			2 (0.9%)	10 (4.4%)	8 (1.9%)		15 (2.2%)
9							1 (0.4%)			
10			2 (2.4%)				1 (0.4%)	10 (2.4%)		20 (2.9%)
11			4 (4.9%)		4 (3.3%)					
12			1 (1.2%)							
13			1 (1.2%)		7 (5.8%)	17 (7.6%)	24 (10.6%)	13 (3.1%)		21 (3%)
14						1 (0.4%)	2 (0.9%)			4 (0.6%)
15						8 (3.6%)	10 (4.4%)			14 (2%)
Total	29 (100%)	50 (100%)	64 (78%)	106 (100%)	114 (95%)	205 (91.5%)	196 (86.7%)	371 (87.9%)	480 (100%)	604 (86.9%)

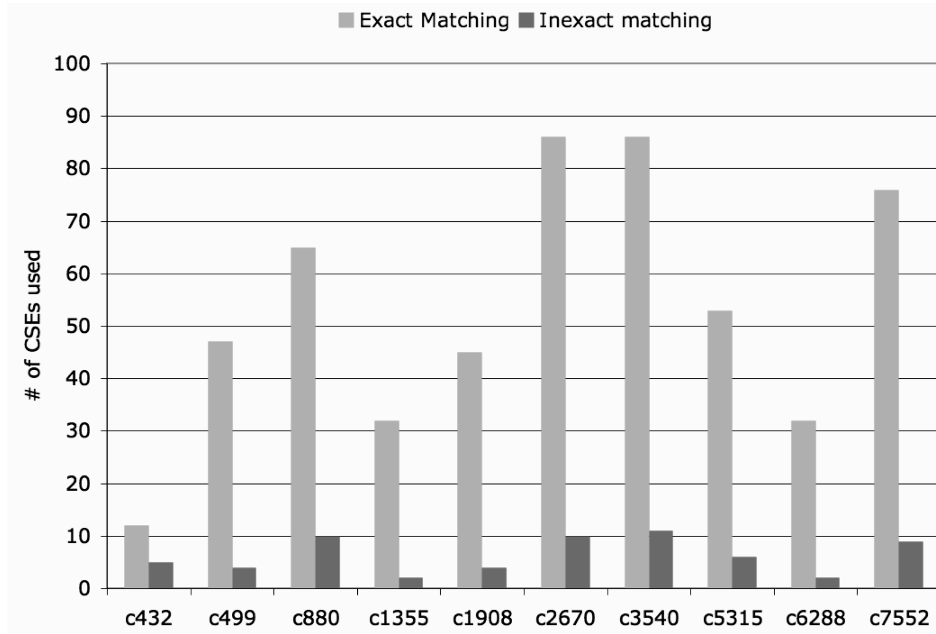


Figure 11: Number of CSE's used to cover the ISCAS '85 combinational benchmark circuits (with the coverage shown in Figure 4).

Table 5: LUT's used in the two phases of LURU's inexact matching extension, shown with percentage of total LUT's for each benchmark ignoring partitions with a single gate.

ISCAS file	LURU phase 1 (#LUT's)	LURU phase 2 (#LUT's)
c432	29 (100%)	0 (0%)
c499	50 (100%)	0 (0%)
c880	64 (78%)	18 (22%)
c1355	106 (100%)	0 (0%)
c1908	114 (95%)	6 (5%)
c2670	205 (91.5%)	19 (8.5%)
c3540	196 (86.7%)	30 (13.3%)
c5315	371 (87.9%)	51 (12.1%)
c6288	480 (100%)	0 (0%)
c7552	604 (86.9%)	91 (13.1%)

6.0 CONCLUSION

6.1 CONTRIBUTIONS OF THE TECHNIQUE

This work demonstrates the effectiveness of CAM for applications outside of the networking area, such as the mapping problem explored in this thesis. The LURU technique introduces a novel string-based circuit representation for use with CAM. The ternary CAM feature of wildcards is used to expand the string representation to facilitate inexact matching. The inexact matching LURU technique is capable of exceeding the area quality of both FlowMap and CutMap, while gaining a significant circuit optimization mechanism to improve upon the original exact matching LURU technique. The profiling feature of LURU allows a user to determine which subcircuits (CSE's) occur most frequently. Standard cells implementing such CSE's could be individually optimized. Entire circuit designs may be optimized through knowledge of their CSE distributions and careful CSE optimization. Highly regular circuits can especially benefit from such optimizations in area, speed, and power consumption.

The LURU technology mapping technique provides several advantages: the exact string matching version of the LURU algorithm provides a reduction in the LUT requirement for technology mapping over FlowMap and CutMap. The average area savings over CutMap is 25%, with a maximum reduction of 53%. The inexact string matching version of the algorithm provides an average 21% reduction, with a maximum of 49% (see Tables 1 and 2). The inexact string matching version is much faster than the exact string matching version, requiring just 16 CAM searches instead of the 5440 required by the exact string matching version. The 16 CSEs used by the inexact string matching version cover 92.6% of the subcircuits (on average) in the ISCAS '85 benchmark set.

6.2 FUTURE RESEARCH

Research should be conducted into the chosen set of CSEs, as this choice affects area results and determines which subcircuits show the most prominence in a circuit. Technology mapping results using other K values (other numbers of LUT inputs) should be generated and analyzed to further quantify the benefit of the LURU technique. Also, the technique should be modified to map to a network of variable-size LUTs. Cleverly choosing LUT sizing in relation to placement may provide for significant improvements. Uses of the LURU technique within the context of a heterogeneous FPGA fabric should be investigated: modern FPGAs that include various functional units, RAMs, and LUTs may take advantage of LURU. Standard cell implementations of CSEs should be developed to evaluate the effectiveness of the optimizations facilitated by the inexact string matching version of the LURU algorithm. Additionally, an effort should be made to amortize the sub-optimal effects of circuit partitioning and the impact of the LURU technique on the packing, placement, and floorplanning stages of design flow need to be explored.

BIBLIOGRAPHY

[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[2] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *Proc. DAC*, June 1987.

[3] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms.* Kluwer Academic Publishers, 1996.

[4] L. Chivin and R. Duckworth, "Content-addressable and associate memory: Alternatives to the ubiquitous RAM," *IEEE Computer*, July 1989.

[5] Micron Inc., "Harmony ternary CAM MT75W16Y136H," Datasheet, www.micron.com.

[6] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff, and J. Rose, "The stratix II logic and routing architecture," in *Proceedings of the ACM Field Programmable Gate Array Conference FPGA*, 2005.

[7] J. Lucas, R. Hoare, I. Kourtev, and A. K. Jones, "LURU: Global scope FPGA technology mapping with content-addressable memories," in *Proceedings of the 11-th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2004)*, 2004, pp. 599–602.

[8] J. Lucas, R. Hoare, and A. K. Jones, "LURU2: Optimizing technology mapping for FPGAs using CAMs," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2005.

[9] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," *IEEE Trans. on VLSI*, June 1994.

[10] ——, "An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," in *Proc. IEEE Int'l Conf. on CAD*, November 1992.

[11] ——, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Trans. on CAD*, January 1994.

[12] J. Cong and Y. Hwang, "Simultaneous depth and area minimization in LUT-based FPGA mapping," in *Proc. ACM/SIGDA Int'l. Symposium on FPGAs*, 1995.

[13] C. Meinel and T. Theobald, *Algorithms and Data Structures in VLSI Design.* Springer, 1998.

[14] H. Yamada, Y. Murata, T. Maeda, R. Ikeda, K. Motohashi, and K. Takahashi, "Real-time string search engine LSI for 800-mbit/sec LANs," in *Proc. IEEE Custom IC Conf.*, 1988.

[15] K. J. Schultz, "CAM-based circuits for ATM switching networks," Ph.D. dissertation, University of Toronto, 1996.

[16] M. Slyz, "Image compression using a ziv-lempel type coder," Master's thesis, University of Michigan School of Engineering, 1991.

[17] Integrated Device Technology, "75P42100 NSE datasheet brief," 2003, http://www1.idt.com/pcms/products.taf? catID=58521&genID=75P42100.

[18] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran," in *Proc. ISCAS*, June 1985.

[19] S. North and E. Koutsofios, "Applications of graph visualization," in *Proc. Graphics Interface Conference*, May 1994.