

Combinational and Sequential Mapping with Priority Cuts

Alan Mishchenko

Sungmin Cho

Satrajit Chatterjee

Robert Brayton

Department of EECS, University of California, Berkeley

{alanmi, smcho, satrajit, brayton}@eecs.berkeley.edu

Abstract

An algorithm for technology mapping of combinational and sequential logic networks is proposed and applied to mapping into K -input lookup-tables (K -LUTs). The new algorithm avoids the hurdle of computing all K -input cuts while preserving the quality of the results, in terms of area and depth. The memory and runtime of the proposed algorithm are linear in circuit size and quite affordable even for large industrial designs. For example, computing a good quality 6-LUT mapping of an AIG with 1M nodes takes 150Mb of RAM and 1 minute on a typical laptop. An extension of the algorithm allows for sequential mapping, which searches the combined space of all possible mappings and retimings. This leads to an 18-22% improvement in depth with a 3-5% LUT count penalty, compared to combinational mapping followed by retiming.

1 Introduction

Technology mapping transforms a technology-independent logic network, called the *subject graph*, into a network of logic nodes. For Field-Programmable Gate Arrays (FPGAs) each logic node is represented using a K -input *look-up table* (LUT) implementing any Boolean function up to K inputs. The subject graph is often represented as an *AND-Inverter Graph* (AIG) composed of two-input ANDs and inverters.

Most structural methods of FPGA mapping [6][12] start by computing all, or nearly all, K -feasible cuts for each AIG node. Similar methods exist for standard cell mapping. The number of such cuts in a network with n nodes is $O(n^K)$ [3]. Next, the AIG nodes are traversed in a topological order and a dynamic programming approach is used to find an optimum-depth LUT mapping of the AIG. This mapping is transformed by applying area-recovery heuristics [3][11][12], which reduce the number of logic nodes while preserving the depth of the LUT network.

It should be noted that some structural FPGA mapping algorithms, e.g. FlowMap [2] and CutMap [4], do not compute all cuts. Instead, one good cut is found at each node using the maximum-flow algorithm, but this approach tends to have higher computational complexity and relatively poor area. As a result, a recent state-of-the-art mapper produced by that same research group [6] is based on cut enumeration rather than maximum flow.

In a large class of programmable architectures, the LUT size, K , varies between 3 and 6. For these relatively small LUT sizes, the traditional methods for LUT mapping based on cut enumeration work well. For K equal to 4 or 5, exhaustive cut enumeration [14][5] can be applied, resulting in an average of 10-40 cuts stored at each node. When the LUT size is 6, exhaustive cut enumeration may lead to 100+ cuts per node. As a result, cut representation takes substantial memory when mapping large Boolean networks. To remedy this, a partial cut enumeration can be used to prune the cuts resulting in reduced memory requirements [5]. However, cut pruning may result in losing good cuts, so that depth-optimality of mapping is not guaranteed.

Another class of modern programmable architectures realizes logic networks using macro-cells, which typically contain LUTs and other logic gates. A straight-forward way of mapping logic into programmable macro-cells starts by computing all K -input cuts for each node where K is the number of macro-cell inputs. Unlike a K -input LUT, a K -input macro-cell cannot implement all logic functions of K inputs. Therefore, the local function of each cut is computed as a function of the cut inputs, and only those cuts whose logic function can be expressed by the macro-cell are used for mapping. However, methods based on cut enumeration cannot be applied because a macro-cell often has 8 or more inputs, and the number of 8-input cuts is extremely large and can be computed only for the smallest benchmarks.

This paper presents a new algorithm for high-quality mapping whose runtime and memory requirements are linear in the number of nodes in the subject graph. The proposed algorithm avoids exhaustive cut enumeration by computing only a small fixed number (typically, 5-10) of “good” K -feasible cuts at each node.

These are called *priority cuts*. The criteria used to prioritize the cuts differ depending on the mapping goals. For example, when mapping for depth, the cuts are prioritized first by depth, then by the number of inputs, and finally by area. Experiments indicate that such prioritization gives a depth-optimum mapping for 95% of all benchmarks and LUT sizes, even if only one cut is stored at each node! Increasing the number of priority cuts to 8 allows the algorithm to avoid area penalty due to not enumerating all cuts, while still offering dramatic improvements in memory and runtime, compared to exhaustive cut enumeration.

For 6-input LUTs, with 8 priority cuts stored, memory is reduced 10x and runtime 5x, compared to previous approaches, while depth and area are comparable or better. For 8-input and larger LUTs, the reduction in memory and runtime is about 50x.

The proposed algorithm is extended to sequential mapping, which searches a combined space of combinational K -LUT mapping and retiming of the resulting LUT network. This integrated mapping leads to a 20% reduction in depth, compared to the combinational LUT mapping followed by retiming performed as a post-processing step.

We emphasize that although this paper was written with FPGA mapping in mind and the experiments were done as such, cut-based mapping for standard cells, macro-cells, super-gates, etc. is similar. The use of priority cuts in this and other applications (e.g. rewriting) should be equally applicable and similar improvements can be expected.

The rest of the paper is organized as follows. Section 2 describes some background. Section 3 reviews the traditional FPGA mapping algorithm. Section 4 describes the new algorithm. Section 5 presents the extension to sequential mapping. Section 6 reports experimental results. Section 7 concludes the paper and outlines future work.

2 Background

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. The terms Boolean network and circuit are used interchangeably in this paper.

A node n has zero or more *fanins*, i.e. nodes that are driving n , and zero or more *fanouts*, i.e. nodes driven by n . The *primary inputs* (PIs) are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network. If the network is sequential, it contains registers whose inputs and output are treated as additional PIs/POs in combinational optimization and mapping. It is assumed that each node has a unique integer called its *node ID*.

A network is *K-bounded* if the number of fanins of any node does not exceed K . A *subject graph* is a K -bounded network used for technology mapping. Any combinational network can be expressed as an AND-INV graph (AIG), composed of two-input ANDs and inverters. In the rest of the paper, the subject graph is assumed to be an AIG.

A *cut* C of node n , called *root*, is a set of nodes of the network, called *leaves*, such that each path from a PI to n passes through at least one leaf. A *trivial cut* of node n is the cut $\{n\}$ composed of the node itself. A non-trivial cut *covers* all the nodes found on the paths from the root to the leaves, including the root and excluding the leaves. A trivial cut does not cover any nodes. A cut is *K-feasible* if the number of nodes in it does not exceed K . A cut is said to be *dominated* if there is another cut of the same node, which is contained, set-theoretically, in the given cut.

A *fanin (fanout) cone* of node n is a subset of all nodes of the network reachable through the fanin (fanout) edges from the given node. A *maximum fanout free cone* (MFFC) of node n is a subset of the fanin cone, such that every path from a node in the subset to the POs passes through n . Informally, the MFFC of a node contains all the logic used exclusively by the node. When a node is removed or substituted, the logic in its MFFC can be removed.

The *level* of a node is the length of the longest path from any PI to the node. The node itself is counted towards the path lengths but the PIs are not. The network *depth* is the largest level of an internal node in the network. The depth and area of FPGA mapping is measured by the depth of the resulting LUT network and the number of LUTs in it.

A *mapping* assigns one K -feasible cut, called *representative cut*, to each non-PI node of the subject graph. The procedure also computes and incrementally updates a subset of nodes whose representative cuts cover all non-PI nodes in the graph. These nodes are said to be *used* in the mapping.

In this paper, a starting mapping is found by assigning one “good” cut at each node in the graph. Next, the mapping is updated by modifying the representative cut of one node at a time. Each such modification may change the set of used nodes. These changes may propagate recursively from the node towards the PIs.

The *area* of the mapping is the number of nodes used in the mapping. Heuristic area recovery discussed in this paper is greedy in the sense that it modifies the representative cuts of the nodes, one at a time, in such a way that the area of the current mapping is reduced or remains the same.

3 Traditional FPGA mapping

Figure 3.0 outlines of the traditional FPGA technology mapping algorithm [2][6], as implemented in [12].

```

traditionalMap( aig, K ) {
    // compute all K-feasible cuts at each node and save them
    traditionalMapCutEnumeration( aig, K );

    // find a min-depth cut and save it as the representative at each node
    traditionalMapDepthOriented( aig, K );

    // update the representative cut at each node to save area
    traditionalMapAreaRecovery( aig, K );

    // return the set of nodes used in the final mapping
    traditionalMapDeriveFinalMapping( aig, K );
}

```

Figure 3.0. The traditional FPGA mapping.

3.1 Cut enumeration

Here we review the cut enumeration method from [14][5].

Let A and B be two sets of cuts. For convenience we define the operation $A \diamond B$ as:

$$A \diamond B = \{ u \cup v \mid u \in A, v \in B, |u \cup v| \leq k \}$$

Let $\Phi(n)$ denote the set of K -feasible cuts of node n . If n is an AND node, let n_1 and n_2 denote its fanins. The set of cuts of node n is computed using sets of cuts of its fanins:

$$\Phi(n) = \left\{ \begin{array}{ll} \{ \{n\} \} & : n \in \text{PI} \\ \{ \{n\} \} \cup \Phi(n_1) \diamond \Phi(n_2) & : \text{otherwise} \end{array} \right\}.$$

Performing cut computation for the nodes in a topological order guarantees that the fanin cuts, $\Phi(n_1)$ and $\Phi(n_2)$, are available when the node cuts, $\Phi(n)$, are computed. The set of computed cuts is filtered by removing dominated cuts. This helps reduce runtime and memory without sacrificing quality of mapping.

3.2 Depth-oriented mapping

Figure 3.1 shows the pseudo-code of depth-oriented mapping performed in the traditional FPGA mapping.

```

traditionalMapDepthOriented( aig, lut_size ) {
    for each aig node n in topological order {
        cut = findCutMinimizingDepth( n );
        setLevel( n, getLevel( cut ); setRepresentativeCut( n, cut );
    }
}

findCutMinimizingDepth( node ) {
    cut_best = NULL;
    for each cut c of node
        if ( cut_best == NULL or getLevel( cut_best ) > getLevel( c ) )
            cut_best = c;
    return cut_best;
}

getLevel( cut ) {
    level_max = -∞;
    for each node m in cut
        level_max = max( level_max, getLevel( m );
    return 1 + level_max;
}

```

Figure 3.1. Depth-oriented traditional FPGA mapping.

The AIG nodes are considered in a topological order. At each node, all cuts are enumerated and an optimum-depth cut is found. This cut along with its level is stored at the node. The level of a cut is computed by adding 1 to the largest level of the cut fanins.

3.3 Area recovery

Figure 3.3.1 shows the pseudo-code of the area recovery performed as part of the traditional FPGA mapping.

```
traditionalMapAreaRecovery( aig, lut_size )
{
  computeRequiredTimes( aig );
  for each aig node  $n$  in topological order {
    cut = findCutMinimizingAreaFlow(  $n$  );
    setLevel(  $n$ , getLevel(cut) ); setRepresentativeCut(  $n$ , cut );
  }
  computeRequiredTimes( aig );
  for each aig node  $n$  in topological order {
    cut = findCutMinimizingExactLocalArea(  $n$  );
    setLevel(  $n$ , getLevel(cut) ); setRepresentativeCut(  $n$ , cut );
  }
}
```

Figure 3.3.1. Area recovery in traditional FPGA mapping.

Previous work [12] has shown that applying two complementary heuristics in a given order produces good practical results. The first heuristic (area flow) has a global view and selects logic cones with more shared logic. The second heuristic (exact local area) provides a missing local view by minimizing the area exactly at each node. The following subsections give an overview of these heuristics.

Figure 3.3.2 shows the pseudo-code of the required time computation used in the above area recovery procedure.

```
computeRequiredTimes( aig )
{
  // find the global required times
  time_max = findLatestPoArrivalTime( aig );

  // initialize the required times
  for each node  $n$ 
    setRequiredTime(  $n$ , +∞ );
  for each PO node  $n$ 
    setRequiredTime(  $n$ , time_max );

  // propagate the required times
  for each aig node  $n$  in reverse topological order {
    time_req_new = getRequiredTime(  $n$  ) - 1;
    cut = getRepresentativeCut(  $n$  );
    for each node  $m$  in cut {
      time_req_old = getRequiredTime(  $m$  );
      setRequiredTime(  $m$ , MIN( time_req_old, time_req_new ) );
    }
  }
}
```

Figure 3.3.2. Required time computation for the mapping.

3.3.1 Global view heuristic

Area flow [11] (effective area [5]) is a useful extension of the notion of area. It can be computed in one pass over the network from the PIs to the POs. Area flow for the PIs is set to 0. Area flow at a node n is:

$$AF(n) = [Area(n) + \sum_i AF(Leaf_i(n))] / NumFanouts(n),$$

where $Area(n)$ is the number of LUTs needed to map the current best cut of node n . $Area(n)$ can be 1 or larger if some of the fanins of the topmost LUT do not have external fanouts. $Leaf_i(n)$ is the i -th leaf of the best cut at n , and $NumFanouts(n)$ is the number of fanouts of node n in the currently selected mapping. If a node is not used in the current mapping, for the purposes of area flow computation, its fanout count is assumed to be 1.

If nodes are processed from the PIs to the POs, computing area flow is fast. Area flow gives a global view of how useful the logic is in the cone for the current mapping. Area flow estimates sharing between cones without the need to re-traverse them.

3.3.2 Local view heuristic

The exact local area of a node is the area added to the mapping by selecting the current node into the mapping. The *exact area of a cut* is defined as the sum of areas of the LUTs in the MFFC of the cut, i.e. the LUTs to be added to the mapping if the cut is selected as the best one.

The exact area of a cut is computed using a fast local DFS traversal of the subject graph starting from the root node of the cut. The *reference counter of a node* in the subject graph is equal to the number of times it is used in the current mapping, i.e. the number of times it appears as a leaf of the best cut at some other node, or as a PO. The exact area computation procedure is called for a cut. It adds the cut area to the local area being computed, dereferences the cut leaves, and recursively calls itself for the best cuts of the leaves whose reference counters are zero. This procedure recurs as many times as there are LUTs in the MFFC of the cut, for which it is called. This number is typically small, which explains why computing the exact area is reasonably quick. Once the exact area is computed, a similar recursive referencing is performed to reset the reference counters to their initial values, before computing the exact area for other cuts.

3.4 Producing a mapped network

The procedure used in the traditional mapping to derive the final LUT network is shown in Figure 3.4 [14]. The procedure assumes that one K-feasible representative cut is assigned at each node. Two sets of AIG nodes are supported: the nodes used in the mapping (M) and the nodes currently present in the frontier (F). While the frontier is not empty, one node (n) is extracted from it, added to the mapping, the representative cut of this node is computed, and the leaves of this cut are explored. If a leaf (m) does not belong to the mapping or is not a PI, this leaf is added to both the mapping and the frontier. When the frontier is empty, the procedure terminates and returns the set M of nodes used in the mapping. Each of these nodes will be implemented by a LUT.

```
fastMapDeriveFinalNetwork( aig, lut_size )
{
  // set the mapped nodes and the frontier to be the set of PO nodes
  M = ∅; F = POs;

  // explore each node in the frontier
  while ( F ≠ ∅ ) {
    n = extractNode( F );
    insertNode( M, n );
    cut = getCut( n );
    for each node  $m$  in cut
      if (  $m \notin M$  or  $m \notin$  PIs )
        F = F ∪  $m$ ;
  }

  // return the set of nodes used in the final mapping
  return M;
}
```

Figure 3.4. Producing the mapped LUT network.

4 Proposed algorithm

The proposed algorithm is similar to the traditional mapping presented in Section 3 in that it considers all nodes in a topological order and optimizes depth, followed by several passes of area recovery. In the end, the mapped LUT network is produced as in the traditional mapping (see above Section 3.4).

The following subsections summarize the differences.

4.1 Priority cuts

The main difference is that the proposed algorithm, instead of computing all K -feasible cuts at each node, computes a small number, C , of K -feasible cuts at each node (typically, $4 \leq C \leq 8$). When priority cuts are computed for a node, the trivial cut is added to the set of at most C non-trivial cuts at each fanin. This allows the cut enumeration procedure from Section 3.1 to produce at most $(C+1)^2$ candidate cuts for the node. Next, the candidate cuts are sorted using a sorting function, and the best C cuts are found and stored at the node. In practice, sorting is done on the fly, by keeping only C best cuts at any time. The fanin cuts are filtered using the delay constraints, which helps reduce the number of cut pairs to be checked during cut enumeration.

The above approach to computing *priority cuts* guarantees that at most C cuts are stored at each node while a mapping pass is taking place and only one cut represents each node at the end of the pass. The best cut may be updated in the next pass. The dynamic update of the cuts during multiple mapping passes makes up for not having all cuts available, by adjusting the current subset of priority cuts to reflect the needs of a particular phase of mapping (e.g. delay in one phase and area in another).

Finally, the following minor modification of the priority cut computation procedure improves the results of mapping. The best cut from the previous pass is always added to the set of at most $(C+1)^2$ candidate cuts derived using the fanin cuts. As a result, the mapping procedure never loses good cuts. If this is not done, the best cut may be lost due the heuristic nature of cut selection.

4.2 Cut sorting

Different sorting functions used in each mapping pass are summarized in Table 4.2. For example, the row labeled “Depth” corresponds to the depth-oriented mapping. In this case, cuts with smaller depths are used; if there is a tie, cuts of smaller size are used; if still there is a tie, cuts with smaller area flow are used. The tie-breaking criterion denoted “fanin refs” means “prefer cuts with smaller average fanin reference counters”.

Table 4.2. Cut sorting criteria in each mapping pass.

Mapping pass	Primary criterion	Tie-breaker 1	Tie-breaker 2
Depth	depth	cut size	area flow
Depth2	depth	area flow	cut size
Area flow	area flow	fanin refs	depth
Exact area	exact area	fanin refs	depth

There is a subtle but important difference between the sorting criteria for mapping passes “Depth” and “Depth2” in Table 4.2. Given two cuts with the same depth, “Depth” chooses the cut with fewer leaves, while “Depth2” chooses the cut with a smaller area flow. When minimization of depth is required, the first option is preferable because it uses cuts with the smallest number of inputs, which allow for mapping as many nodes as possible without increasing the depth of the mapped network.

While experimenting with a large set of industrial benchmarks, it was found that, if an initial mapping is computed using the first sorting option, the depth is optimal for 95% of industrial benchmarks, even if only one cut is used at each node ($C=1$).

4.3 Depth-oriented mapping

The traditional algorithm starts by performing one pass of depth-oriented mapping. This substantially increases area but area recovery brings it down because all cuts are available.

To get a comparable result with priority cuts, it was found helpful to compute several independent mappings while using different sorting functions and stitch them together. Our current implementation starts by computing three independent mappings using sorting criteria “Depth”, “Depth2”, and “Area flow” listed in Table 4.2. The first mapping gives the depth constraint. This mapping is used for the critical and near-critical nodes. The second and third mappings are used on the non-critical paths. This approach minimizes the depth and produces a good starting point for area recovery.

4.4 Area recovery

Area recovery is performed using a sequence of passes over the nodes of the subject graph. The traditional mapping can visit the nodes in any order because it stores all cuts. In the proposed algorithm, only a topological order can be used because a reverse order does not allow re-computing priority cuts in each pass. However, it was found experimentally that restricting the order to a topological one is advantageous for most benchmarks [12].

Several passes of area recovery are based on the sorting functions listed in Table 4.2 as “Area flow” and “Exact area”. In each pass, after the priority cuts are re-computed, the first priority cut is set as the representative cut at each node.

To further improve the quality of area recovery, a simple cut expansion procedure is applied to the best cuts stored at each node between the mapping passes. This procedure moves the cut boundary towards the PIs while maximizing the number of shared fanin as long as the number of cut inputs does not exceed K .

4.5 Memory management

A key observation enabling substantial memory savings in the proposed algorithm is that, no matter how many priority cuts are used and how many mapping passes are performed, the mapper needs to store only one representative (best) cut of each node.

The complete set of C priority cuts should be stored only for the nodes on the *mapping frontier*, which includes the nodes that are already mapped but have at least one unmapped fanout. The largest size of the mapping frontier (called *crosscut*) depends on the subject graph and the topological order used. It was found experimentally that the crosscut size for most of the large industrial benchmarks is less than 1% of the total number of nodes. As a result, the memory needed to store the priority cuts for the nodes on the frontier is typically small compared to the memory needed for the subject graph and the representative cuts.

Our implementation determines the crosscut before mapping and allocates memory upfront for all the representative cuts and priority cuts. During mapping, the priority cut memory is recycled as the nodes are added to and removed from the frontier.

Note that efficient memory management is not only important for reducing the overall memory requirements, but also a smaller memory footprint reduces the number of CPU-cache misses, which improves the speed of mapping large designs.

4.6 Summary of improvements

This section summarizes the heuristics used by the algorithm:

- Compute and use priority cuts (a small subset of all cuts)
- Dynamically update the priority cuts in each mapping pass.
- Use different sorting criteria in each mapping pass
- Include the best cut from the previous pass into the set of candidate cuts of the current pass
- Consider several depth-oriented mappings to get a good starting point for area recovery
- Use complementary heuristics for area recovery

- Perform cut expansion as part of area recovery
- Use efficient memory management

The proposed algorithm often outperforms traditional mapping, which uses the set of all cuts. This confirms that the above heuristics work well together, leading to intelligent tie-breaking and increasing the quality of priority cuts computed. Another reason for superior quality is that the traditional mappers often prune cuts saved at a node when they reach some maximum. Effectively, this results in selecting a subset of cuts randomly.

4.7 Complexity analysis

A previous mapper coming closest to the proposed one, is CutMap [4]. The differences are the following. To find an optimum-depth mapping, CutMap uses FlowMap [2] whose complexity is $O(Knm)$, where K is the LUT size while n and m are the number of nodes and edges in the subject graph. In contrast, the proposed algorithm uses heuristics with complexity $O(Kn)$. Area recovery performed by CutMap uses the maximum-flow algorithm with complexity $O(2Kmn^{\lfloor K/2 \rfloor + 1})$ [4]. In contrast, the proposed algorithm performs heuristic area recovery using several methods with complexity $O(Kn)$. Experimental results confirm this linear runtime.

5 Sequential mapping

So far, we discussed priority-cut-based mapping for combinational networks. Traditionally, if a network has registers, it is divided at the register boundary, the register inputs and outputs are added to the sets of POs and PIs, respectively; only combinational logic is mapped. In some cases, combinational mapping of sequential circuits is followed by retiming, which moves the registers over some combinational nodes to reduce the depth, measured as the longest combinational path after retiming.

However, both combinational mapping alone and combinational mapping followed by retiming often fail to find the best depth feasible for a sequential network when mapping and retiming are synergistically combined [14]. In this case, instead of fixing the network structure by combinational mapping and then retiming the mapped network, a *sequential mapping* is performed. During sequential mapping, depth after retiming is evaluated without fixing the network structure, which often reduces the depth. The resulting mapping has the property that there exists a feasible retiming, such that this mapping followed by this retiming gives the best depth over all possible mappings and retimings.

Below we discuss modifications of the proposed algorithm, which transform it into a robust sequential mapping algorithm that works as described above. Our experiments confirm that, for a large number of benchmarks, a marginal increase in LUTs and registers incurred by the sequential mapping (which includes retiming) is often outweighed by a substantial reduction in depth.

5.1 Computing sequential arrival times

The notion of arrival times was extended to sequential circuits in [14]. Assume that (a) the clock period ϕ is given, and (b) the register boundary is “transparent”. The arrival times of the register outputs are found by subtracting the clock period ϕ from the arrival times of the register inputs. The key observation is that passing over the register boundary delays signals by one clock period. The resulting arrival times at the register output and internal nodes may be negative.

In the combinational case, when the arrival times of the PIs are known, the arrival times of the POs are known at the end of each mapping pass. The situation changes in the sequential case because the arrival times keep changing during several iterations

due to propagating arrival times through the register boundaries. This leads to the procedure shown in Figure 5.1.

This procedure initializes the PI arrival times to the given values (here assumed to be 0) while arrival times of all other nodes are set to $-\infty$. Next, it iterates over the network while computing and updating the arrival times at each node and propagating them over the register boundaries. Note that the cut minimizing depth is updated during each pass since arrival times can change. Since the maximum of the previous and current value is computed, the arrival times increase monotonically at each node. This leads to one of the three outcomes: (a) the arrival times at all nodes converged, (b) the arrival times at a PO exceed ϕ , and (c) an iteration limit is reached.

In the case (a) the clock period is proved to be feasible, meaning that there exists a mapping/retiming such that the longest combinational path does not exceed ϕ levels. The cases (b) and (c) mean that the given clock period is infeasible. In this case ϕ can be increased and sequential arrival times can be re-computed.

A binary search procedure was used for finding the smallest feasible clock period ϕ [14]. An advantage of this procedure is that it reduces the number of sequential arrival time computations. A disadvantage is that each change in ϕ resets the arrival times. A practical alternative is to start with a loose bound on ϕ and gradually tighten it until it becomes infeasible. In this case, the arrival times do not have to be reset, which may lead to faster convergence. Typically, the runtime is affordable in both cases when a tight upper bound on ϕ is available.

```
computeSequentialArrivalTimes(aig,  $\phi$ , iter_limit) {
    // initialize sequential arrival times
    for each aig node  $n$     setLevel( $n$ ,  $-\infty$ );
    for each aig PI  $n$       setLevel( $n$ , 0);

    // iteratively compute sequential arrival times
    for (iter = 0; iter < iter_limit: iter++) {
        changes = 0;
        for each AIG node  $n$  in topological order {
            cut = findCutMinimizingDepth( $n$ );
            if (getLevel( $n$ ) < getLevel(cut)) {
                setLevel( $n$ , getLevel(cut));
                setRepresentativeCut( $n$ , cut);
                changes = 1;
            }
        }
        // check the PO arrival times
        for each aig PO  $n$ 
            if (getLevel( $n$ ) >  $\phi$ )
                return (arrival times at the PO exceeded the clock period)
        // check if the computation converged
        if (!changes)
            return (arrival times are computed);
        // propagate arrival time through the register boundary
        for each aig register  $r$  in topological order
            setLevel(regOutput( $r$ ), getLevel(regInput( $r$ )) -  $\phi$ );
    }
    return (iteration limit has been reached);
}
```

Figure 5.1. Computing sequential arrival times.

5.2 Deriving the resulting network

For a given feasible clock period ϕ , the resulting network can be derived as follows: (1) Apply combinational mapping by setting the arrival times of the additional PIs and POs, which stand for the register outputs and inputs, to be the sequential arrival times computed while checking feasibility of ϕ . (2) Using these arrival times, run minimum-delay retiming to bring the network to its minimum feasible delay ϕ .

We emphasize that sequential mapping is separated into two phases: (1) computation of best clock period ϕ and associated sequential arrival times, and (2) combinational mapping using these arrival times. This separation allows for applying the delay-based priority-cut method in the first phase, and priority cut and combinational mapping with efficient area recovery heuristics in the second phase. Once the mapping is established, efficient incremental retiming is applied, which controls the increase in the number of registers [15]. We know that the mapping is such that it can be retimed to meet clock period ϕ . Thus, the clock period is the same as in a previous work [13] but the numbers of LUTs and registers increase less, because both efficient area recovery can be used as well as area controlled retiming [15]. Also, this two-step computation greatly reduces the implementation complexity.

This two-step computation of the resulting network differs from the one introduced in [14] and generalized in [13], where the final mapping and retiming cannot be separated because of the use of sequential cuts (cuts crossing the current register boundary).

Overall, the increase in quality of results due to using mapping with the “transparent” register boundary was found to outweigh some degradation in quality due to not using sequential cuts. Our experiments indicate that, compared to combinational mapping followed by retiming, the average gain in delay due to the former is about 20% while the average loss in delay due to not using more general sequential cuts is only about 1%.

6 Experimental results

A new LUT-mapper based on priority cuts was implemented in ABC [1] as command “*if*”. The experiments were run on a Windows laptop with a 1.6GHz Pentium-4 CPU and 2GB of RAM. The mapped networks were verified using combinational and sequential equivalence checkers in ABC.

6.1 Comparison with the traditional mapping

To compare against state-of-the-art technology mappers, the same selection of public domain benchmarks was used as in previous work [6][12]. To derive AIGs used as subject graphs, the benchmarks were structurally hashed and balanced.

The summary of mapping results for $K = \{4, 6, 8, 10\}$ in terms of depth, the number of LUTs, memory (in megabytes), and runtime (in seconds) are shown in Table 6.1. Columns “old” refer to the mapper based on exhaustive cut enumeration with improvements [12]. Columns “new” refer to the proposed mapper.

The proposed mapper was allowed to use at most 8 cuts per node. In the traditional mapper, the pruning parameters used for cut enumeration were set as follows. At each node, at most 2,000 cuts are computed. The resulting cuts were sorted in the increasing number of inputs. At most 1,000 of the smallest cuts were stored at each node and allowed to propagate during cut enumeration. For small values of K ($K \leq 6$), the limit of 1000 cuts at a node was never exceeded. In this case, the traditional mapping found an optimum-depth solution.

The first line of Table 6.1 shows that for small LUT sizes ($K = 4$ and $K = 6$) the new mapper produces depth-optimal mappings. The new mapper improves the depth for $K = 8$ and $K = 10$ because the traditional mapper uses pruning to compute a random subset of K -input cuts for large K . For the same reason, the gain in runtime of cut computation for $K = 10$ is less than for $K = 8$.

The second line of Table 6.1 shows that the new mapper produces comparable area. This is a result of applying a well-tuned set of heuristics summarized in Section 4.6.

The last two lines of Table 6.1 show that the new mapper dramatically reduces memory and runtime, compared to a traditional mapper. For 10-input cuts, the gains are 30x in memory

and 20x in runtime. The gains would be larger if pruning was not used during cut enumeration in the traditional method.

In addition to the results shown in Table 6.1, a number of experiments were performed to test the new mapper on large industrial benchmarks. In summary, the new mapper gives the same delay, and reduces memory and runtime 2-100x depending on the LUT size, while area is on average the same or better.

The improvement in area due to priority cuts is more substantial when structural choices are used. Experiments with a diverse set of industrial benchmarks have shown a 2% improvement in area, compared to the mapper [12], which uses choices in the context of exhaustive cut enumeration. The table with these detailed results is not included due to page limitation.

6.2 Performance on large benchmarks

The second set of experiments was designed to show the performance of the new mapper on large benchmarks with large LUT sizes. This experiment was run on multiple timeframes of sequential benchmark *wb_conmax.v* with 1130 PIs, 1416 POs, and 770 registers taken from the IWLS 2005 benchmarks set [9]. Mapping multiple timeframes is relevant to hardware emulation where unrolled designs are emulated to achieve faster simulation.

Table 6.2.1 shows the results of LUT mapping with $K = 10$ and $C = 1$. The first column shows the number of timeframes used to unroll *wb_conmax.v*. The next two columns show the number of levels and nodes in the subject graph. The last four columns show depth, the number of LUTs, memory, and runtime of the proposed algorithm. Note that the runtime is linear in the size of the AIG.

Table 6.2.2 contains the results of mapping with variable K and $C = 1$, applied to 100 frames of *wb_conmax.v*. The first column of the table shows the LUT size. The last four columns show depth, the number of LUTs, memory, and runtime of the proposed algorithm. Note that the runtime is fairly insensitive to cut size – a factor of 4 in cut size increases runtime only about 50%.

Tables 6.2.1 and 6.2.2 report runtimes of the depth-oriented phase of FPGA mapping. Area recovery was not performed in this experiment. These tables demonstrate that the proposed algorithm has reasonable memory and runtime requirements when applied to AIGs with millions of nodes. (The reason why the area of FPGA mapping increased when the LUT size increased from 10 to 12 in Table 6.2.2 will be investigated.)

6.3 Sequential mapping for academic benchmarks

The results of sequential mapping for academic benchmarks are presented in Table 6.3. The proposed algorithm was run with $K = 6$ and $C = 8$. The structural choices [13] were not used during mapping. The circuits selected are a subset of ISCAS’89 benchmarks, for which sequential mapping (which searches a combined space of all combinational mappings and retimings) improved the depth over depth-oriented combinational mapping.

Table 6.3 lists the benchmark name, the number of primary inputs, primary outputs, and AIG nodes in the subject graph. The other sections of the table compare depth and area, expressed in terms of LUTs, and runtime, in seconds, for the following three mapping options: combinational mapping (M), combinational mapping followed by retiming (M+R), and the proposed sequential mapping (MR). Combinational and sequential mapping are performed using ABC command *if* and *if -s*, respectively. Retiming for depth is the heuristic algorithm [15] implemented in ABC as command *retime -M 4*.

Table 6.3 shows that sequential mapping (MR) leads to a substantial (29%) reduction in depth, compared to combinational mapping followed by retiming (M+R) when the reduction is smaller (7%). The LUT area after M+R does not change because

the same logic structure is used, while after MR, area is increased 3%. The number of LUTs and registers is increased by 3% and 8%, respectively, for these benchmarks. The increase in registers could be controlled by using a more sophisticated retiming algorithm. In addition, the algorithm in [15] can be used to create a delay/area tradeoff curve if the optimum delay is not required.

Currently sequential mapping is about five times slower than combinational, due mainly to the binary search for the optimum clock period. Several improvements have been proposed and are waiting to be implemented.

6.4 Sequential mapping for industrial benchmarks

To test if these results are representative, the results of sequential mapping for a diverse set of industrial benchmarks are shown in Table 6.4 with the same notation as in Table 6.3. Before mapping, the benchmarks were optimized using two AIG rewriting scripts, *resyn* followed by *resyn2*.

Table 6.4 confirms that the improvements in depth (25%) and the increase in area (4.5% in LUTs and 8% in registers) after the sequential mapping applied to the industrial benchmarks are close to those observed for the academic benchmarks. For comparison, mapping followed by retiming leads to a 7% reduction in delay, and 0% and 1% increase in LUTs and registers, respectively.

7 Conclusions and future work

This paper presents a new algorithm for technology mapping, which avoids exhaustive cut enumeration.

On close investigation, it is clear that the traditional mapping dramatically over-computes during cut enumeration, wasting memory and runtime. This is especially true for large cuts ($K = 8$ and more) because less than 1% of all computed cuts are selected to represent a node during technology mapping. Therefore, the development of a linear-time algorithm is well motivated.

This paper proposes the first efficient algorithm of this type and may lead to a new class of mapping solutions. The experimental results, applied to LUT mapping, show that the proposed algorithm, although heuristic in nature, almost always finds an optimum-depth mapping while substantially reducing runtime and memory compared to algorithms based on cut enumeration.

Future work will proceed in the following directions:

(1) Extending the algorithm to work for macro-cells and standard cells, which can only implement a subset of the Boolean functions with the given number of inputs; for this purpose, a modified cut computation procedure will be used, which guarantees that the Boolean function of the cut stored at each node can be implemented by the macro-cell.

(2) Combining the mapper with a global placement engine to make mapping placement- and congestion-aware; in this case, the cuts computed by the mapper will be evaluated based on a linear combination of their logic cost as well as their placement cost. A simple way of expressing the placement cost might be to estimate the total change in wirelength needed to implement a cut.

(3) Finally, mapping is a versatile logic synthesis engine customizable by setting the cut area and/or depth to be a user-

specified combination of parameters to be optimized. Several such variations of mapping have been explored, in particular those geared to minimize the number of CNF clauses [7] needed to represent the given logic network. The future work will consider other applications of mapping, such as those minimizing the number of factored form literals or BDD nodes.

Acknowledgements

This work was supported by SRC contracts 1361.001 and 1444.001, and the California Micro Program with our industrial sponsors Altera, Intel, Magma, and Synplicity.

The authors acknowledge helpful discussions with Stephen Jang from Xilinx and his suggestions how to improve logic optimization and technology mapping in ABC.

References

- [1] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 70319. <http://www.eecs.berkeley.edu/~alanmi/abc/>
- [2] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs", *IEEE Trans. CAD*, vol. 13(1), Jan. 1994, pp. 1-12.
- [3] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," *IEEE Trans. VLSI*, vol. 2(2), Jun. 1994, pp. 137-148.
- [4] J. Cong and Y. Hwang, "Simultaneous depth and area minimization in LUT-based FPGA mapping," *Proc. FPGA '95*, pp. 68-74.
- [5] J. Cong, C. Wu and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," *Proc. FPGA '99*, pp. 29-36.
- [6] D. Chen and J. Cong, "DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs," *Proc. ICCAD '04*, pp. 752-757.
- [7] N. Een, A. Mishchenko, and N. Sorensson, "Applying logic synthesis to speedup SAT", *Proc. SAT '07* (to appear).
- [8] A. Farrahi and M. Sarrafzadeh, "Complexity of lookup-table minimization problem for FPGA technology mapping," *IEEE Trans. CAD*, vol. 13(11), Nov. 1994, pp. 1319-1332.
- [9] IWLS 2005 Benchmarks. <http://iwls.org/iwls2005/benchmarks.html>
- [10] C.-C. Kao and Y.-T. Lai, "An efficient algorithm for finding minimum-area FPGA technology mapping". *ACM TODAES*, vol. 10(1), Jan. 2005, pp. 168-186.
- [11] V. Manohara-rajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping," *Proc. IWLS '04*, pp. 14-21.
- [12] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs", *IEEE Trans. CAD*, Vol. 26(2), Feb 2007, pp. 240-253. http://www.eecs.berkeley.edu/~alanmi/publications/2006/tcad06_map.pdf
- [13] A. Mishchenko, S. Chatterjee, R. Brayton, and P. Pan, "Integrating logic synthesis, technology mapping, and retiming", *ERL Technical Report*, EECS Dept., UC Berkeley, Dec. 2006. http://www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_int.pdf
- [14] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," *Proc. FPGA '98*, pp. 35-42.
- [15] D. R. Singh, V. Manohara-rajah, and S. D. Brown, "Incremental retiming for FPGA physical synthesis", *Proc. DAC '05*, pp. 433-438.

Table 6.1. Ratios of improvements of the proposed vs. traditional combinational K-LUT mapping.

Ratio	K = 4		K = 6		K = 8		K = 10	
	old	new	old	new	old	new	old	new
Depth	1.00	1.00	1.00	1.00	1.00	0.93	1.00	0.82
Area	1.00	0.99	1.00	1.00	1.00	0.96	1.00	0.84
Memory	1.00	0.12	1.00	0.06	1.00	0.05	1.00	0.05
Runtime	1.00	0.78	1.00	0.15	1.00	0.02	1.00	0.03

Table 6.2.1. Performance of the proposed algorithm on multiple timeframes of *wb_conmax.v*

Number of frames	AIG statistics		FPGA mapping statistics		Computer resources	
	Levels	Nodes	Depth	Number of LUTs	Memory, Mb	Runtime, sec
1	18	40381	4	11069	2.21	0.02
20	284	808135	61	205143	42.68	0.42
40	564	1616285	121	409149	85.28	0.84
60	844	2424435	181	613155	127.88	1.35
80	1124	3232585	241	817161	170.48	1.77
100	1404	4040735	301	1021167	213.09	2.25

Table 6.2.2. Performance of the proposed algorithm on 100 timeframes of *wb_conmax.v* for different LUT sizes

LUT size	FPGA mapping statistics		Computer resources	
	Depth	Number of LUTs	Memory, Mb	Runtime, sec
4	602	2279062	114.74	1.89
6	451	1704400	147.52	2.00
8	352	1205319	180.30	2.19
10	301	1021167	213.09	2.24
12	276	1044370	245.87	2.50
14	227	799618	278.65	2.55
16	202	694954	311.43	2.62

Table 6.3. Comparison of sequential vs. combinational 6-LUT mapping for academic benchmarks.

Name	Statistics			Depth (LUTs)			Area (LUTs)			Area (registers)			Time, sec	
	PI	PO	AIG	M	M+R	MR	M	M+R	MR	M	M+R	MR	M	MR
s13207	31	121	2136	6	5	4	1047	1047	1056	648	666	733	0.06	0.23
s1423	17	5	441	10	10	9	131	131	146	74	74	80	0.01	0.04
s15850.1	77	150	2755	9	7	6	1012	1012	1042	516	552	533	0.09	0.38
s15850	14	87	2760	9	7	5	1002	1002	1015	563	640	640	0.09	0.43
s35932	35	320	8129	3	3	2	2320	2320	2320	1728	1728	1872	0.19	0.45
s382	3	6	100	3	3	2	36	36	34	21	21	22	0.00	0.04
s38417	28	106	8171	6	6	5	2623	2623	2901	1564	1564	1636	0.28	3.02
s38584.1	38	304	9967	6	6	5	2491	2491	2558	1276	1276	1299	0.31	0.81
s38584	12	278	9989	6	6	5	2504	2504	2517	1301	1301	1327	0.31	0.92
s9234.1	36	39	1349	5	5	3	319	319	332	145	145	171	0.03	0.10
s9234	19	22	1349	5	4	3	321	321	330	160	181	182	0.02	0.14
Ratio				1.00	0.93	0.71	1.00	1.00	1.03	1.00	1.03	1.08	1.00	4.54

Table 6.4. Comparison of sequential vs. combinational 6-LUT mapping for industrial benchmarks.

Name	Statistics		Depth (LUTs)			Area (LUTs)			Area (registers)		
	PI	PO	M	M+R	MR	M	M+R	MR	M	M+R	MR
Ex01	1233	3438	4	4	3	5893	5893	6198	1704	1704	1775
Ex02	1211	5658	5	5	4	8029	8029	8177	1597	1597	1660
Ex03	3431	9646	7	5	4	20021	20021	20213	7930	7944	8050
Ex04	4566	15023	12	11	10	29542	29558	30146	7966	7997	7988
Ex05	827	2099	12	12	11	6198	6198	6591	1677	1677	1805
Ex06	11987	59894	6	6	5	102718	102714	123155	26447	26443	26741
Ex06	526	1627	14	13	10	9024	9024	9414	3061	3126	3885
Ex07	1747	7944	6	5	4	11188	11209	11227	2893	2925	2854
Ex09	4710	11447	7	7	6	22591	22580	23171	7277	7272	7282
Ex10	1859	5347	9	9	7	9265	9265	9853	753	753	1053
Ex11	6112	18898	21	21	19	58258	58258	61750	11772	11772	13248
Ex12	129	4096	3	3	2	34304	34304	33376	19840	19840	20800
Ex13	18037	47101	7	7	5	65589	65636	65710	17532	17578	17539
Ex14	842	3100	21	20	18	13414	13410	15165	5039	5071	5271
Ex15	926	2535	11	10	8	4677	4677	4957	475	548	729
Ex16	350	848	5	5	4	2842	2842	2888	1271	1271	1404
Ex17	153	234	7	7	6	502	502	544	148	148	149
Ex18	2851	10008	7	4	3	18832	18832	18966	7057	7131	7087
Ex19	470	2365	14	14	13	8582	8576	9227	3271	3264	3343
Ex20	519	1346	7	7	5	2374	2374	2431	575	575	623
Ratio			1.000	0.933	0.755	1.000	1.000	1.045	1.000	1.010	1.083