Xmap: a Technology Mapper for Table-lookup Field-Programmable Gate Arrays

Kevin Karplus*

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

This paper presents a new algorithm, Xmap, for doing mapping from multi-level logic to field-programmable gate arrays based on table-lookup gates, such as those used in the Xilinx chip.

The algorithm is based on an if-then-else DAG representation for the functions. The technology mapper differs from previous mappers in that the circuit is not decomposed into fan-out-free trees.

The Xmap algorithm uses 7% fewer cells than Chortle, 11% fewer than misII, and 14% fewer than mis-pga, and is 4.5 times faster than Chortle, 17 times faster than misII, and at least 150 times faster than mis-pga.

1 Why a new technology mapper?

Previous generations of technology mappers have generally worked by choosing from a library of available cells [5, 9,3,1]. Field-programmable gate arrays, such as those made by Xilinx, do not use a library of different cell types, but use an array of identical cells, each of which can be used quite flexibly. The cell-library-based mappers do not work particularly well when mapping to such flexible cells, and so dummy cell libraries are usually created, where each library entry is one way to configure a cell in the gate array. The cell-library approach allows existing technology mappers to be used, but does not scale well as the size of the basic cells increases, because the library tends to grow exponentially with the size of the basic cell.

At the 1990 Design Automation Conference, two technology mappers for programmable gate arrays were presented: Chortle and mis-pga [4,11]. These techniques do not need a cell library, but are still based on decomposing the circuit into fan-out-free trees (non-overlapping rooted sub-DAGs with no internal reconvergence) before mapping. Even if the trees are optimally mapped, the decomposition may be worse than a non-optimal mapping of the original DAG.

The Xmap algorithm maps to the Xilinx chip and other table-lookup-based gate arrays. Xmap is much faster than Chortle or mis-pga, and the results are better, but has no guarantee of optimality on trees. The current version of Xmap handles only combinational logic.

The logic blocks found by Xmap are (possibly overlapping) sub-DAGs of the if-then-else DAG for the entire circuit. Because of this *direct mapping*, Xmap preserves any path-delay-fault testability of the underlying DAG [8].

*This research was funded by NSF grant MIP-8903555.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

2 Conversion from BLIF to If-then-else DAGs

The data structure used to represent the circuits is a multiply-rooted if-then-else directed acyclic graph [6]. Each root corresponds to a primary output of the circuit, and each leaf to a primary input. Intermediate nodes can be thought of as 2-to-1 selectors, with the control input connected to the node on the if-branch, and the "1" and "0" inputs connected to the then- and else-branches. The connections between the selectors can be either wires or inverters—the polarity of a function is stored as a label on the edge pointing to the DAG node, rather than using extra nodes for inversion.

To compare Xmap fairly with existing mappers, it has been run on the output of the misII logic minimizer [2], which is in Berkeley Logic Interchange Format (BLIF). Because BLIF represents a circuit as a network of sum-of-products representations, rather than as an if-then-else DAG, a conversion needs to be made.

Building an if-then-else DAG from a network of gates is easy if each gate is described as an if-then-else DAG—we simply glue together the gate functions to build the DAG for the entire circuit. The only tricky part is converting the sum-of-products descriptions of the gates into if-then-else DAGs.

Our technique for converting sum-of-products to if-then-else DAGs does some simple factoring [6]. First, the terms of the sum-of-products expression are collected in a set T. Then the variables are sorted in decreasing order of the frequency with which they occur in the terms. Finally, a recursive function is applied to T to get an if-then-else DAG E.

The recursive function

- sorts the terms of T, grouping together those that don't use the first input variable (T_d), those that use v_1' (T_0), and those that use v_1 (T_1),
- strips the first variable off the terms in each group, and applies the routine recursively to get expressions E_d, E_0, and E_1, and
- builds the expression E as (**if** E_d **then** TRUE **else** (**if** v_1 **then** E_1 **else** E_0)).

A similar technique creates binary decision diagrams in mis-pga, but binary decision diagrams cannot represent the separation of E_d from E_0 and E_1, and so much of the factorization is lost.

3 Mapping to table-lookup logic blocks

One approach for programmable gate arrays is to provide logic blocks that can implement any Boolean function with up to f variables (typically 4 or 5). The logic blocks are easily implemented as a 2^f-bit ROM or RAM for the truth table and a selector controlled by the f inputs.

Because blocks are often used with fewer than f inputs, Xilinx allows their blocks to be split into two $(f-1)$-input functions, as long as the block has no more than f total

© 1991 ACM 0-89791-395-7/91/0006/0240 $1.50

```
mark(d)
    gatein(d) ← signals(d)
    signals(d) ← {d}
visit(n)
    visit all previously unvisited children
    signals(n) ← ⋃_{c a child of n} signals(c)
    while |signals(n)| > f
        choose d an unmarked descendant of n
        mark(d)
        update signals(n)
```

Figure 3.1: Pseudo-code for marking pass. The `visit` procedure is called for each root of the DAG. The text gives heuristics for choosing d.

inputs. The truth table storage space remains the same, and the selector circuitry is only slightly more complex. Allowing two functions to share a block reduces the number of blocks needed by about 32% for five-input blocks, and about 22% for four-input blocks.

The algorithm for mapping to logic blocks works in four passes:

1. If $f = 2$, we first preprocess the entire DAG, replacing all three-input if-then-else triples (**if** a **then** b **else** c) with either $ab + a'c$ or $(a + c)(a' + b)$. This DeIf procedure[1] guarantees that each node has at most two children, not counting TRUE and FALSE.
2. Mark some nodes as outputs of logic blocks, and record for each a set of f or fewer marked nodes that could be used as gate inputs. A legal set of inputs for a logic block forms a vertex cut of marked nodes separating the node from the primary inputs. Ideally, we would like the cut to be as small as possible, so that the function is more likely to be merged with another into a shared logic block.
3. Choose a polarity for each marked node, and determine the gate function needed to compute the node.
4. Try to merge functions that use fewer than f inputs.

3.1 Xmap algorithm—Marking pass

The marking pass can be thought of as a bottom-up lazy marker. A traversal of the DAG is done, making sure that nodes are visited only after all their inputs have been visited. Nodes that are principal inputs or outputs of the circuit are initially marked, but other nodes are not marked unless some ancestor forces the marking.

For each node we keep track of the *signals* set: a set of marked nodes needed to compute the function of that node. If the node is marked, then the set contains just the node itself; otherwise it contains the signals we would need to use to compute the node. When we mark a node, we save the signals set as the gate inputs, and change the signals set to be the singleton containing just the node. Figure 3.1 gives pseudo-code for the marking phase. Note that the marking phase doesn't care what the function is—only what variables are needed to compute it.

Let's call a node whose signals set is larger than the allowable fan-in an *overflow node*. When we reach an overflow node in the traversal, we have to reduce the fan-in by marking one or more of its descendants, thus hiding the nodes below that descendant. Note that we have to examine descendants only down to the nodes in the signals set—a small piece of the DAG.

The fan-in reduction for overflow nodes is done in two stages.

1. Mark descendants with high fan-out until signals(n) is sufficiently reduced or marking the descendant does not reduce signals(n).
2. Mark children of n, in decreasing order of the size of their signals sets, until signals(n) is sufficiently reduced. Because each if-then-else triple has at most f children, eventually the fan-in limit will be met.

For the first heuristic, we use a recursive procedure, called *reduce_fan-in*(m, h).

1. We mark all children of m that have a large fan-out (fan-out $\geq h$). A new signals set is computed for m, and recorded if it is smaller than the old set. We return success if the signals set of m has been reduced.
2. If marking the high-fan-out children doesn't reduce the signals set of m, we recursively apply *reduce_fan-in*$(child(m), h + 1)$ until one of them succeeds, then recompute signals(m) and return success if $|\text{signals}(m)|$ is reduced. Note that the effects of marking a node are not propagated to all the ancestors, but only to the immediate parent that requested the fan-in reduction.

The *reduce_fan-in*(n, h) procedure is repeated for the overflow node until the signals set is sufficiently reduced or the procedure fails. When it fails, the second heuristic (marking children) is used.

The benchmarks in Section 4 were done with $h = 2$ for the children of n, with h increasing by one for each level of recursion. Using a constant $h = 3$, not increasing it in the recursion, also works well.

The marking algorithm is fast, as each node is visited exactly once by the outer traversal, and the fan-in reduction algorithm visits only a few nodes.

3.2 Xmap algorithm—Building the logic blocks

After nodes have been marked, another traversal is done to choose polarities and assign gate functions. The polarities of the outputs are already set, and the polarities of the inputs are all positive. For all the intermediate nodes the choice of polarity is arbitrary, as long as the choice is consistent in all uses. The only times that inverters are created are when a primary output is the negation of a primary input or both polarities of a primary output are needed.

For each marked node that isn't a primary input we need to create a logic block and choose appropriate inputs for it. We know one legal set of inputs: the signals set that was recorded when the node was marked. However, there may have been other nodes marked that are now in the interior of the gate—this is not a problem, it just means that the some logic has been duplicated. Still, the fan-in of the block could be reduced, if we could find a smaller cut set.

Doing a full min-cut algorithm to find the smallest legal input set seems overly complex, but there is another vertex cut set that is easy to generate: the marked descendants closest to the node. A simple depth-first traversal stopping at marked nodes will find this cut quickly. Whichever of the two cuts (the original signals set or the closest-marked cut) is smaller is the one that is used as the inputs for the gate.

Having chosen the inputs for a logic block, the if-then-else DAG for its function is simply the DAG for the node truncated at the set of inputs, with appropriate inversions of pointers at the inputs and outputs to match the assigned polarities.

[1] DeIf does not preserve testability, and so Xmap does not preserve path-delay-fault testability for $f = 2$.

3.3 Xmap algorithm—Sharing logic blocks

The Xilinx architecture allows two functions to share a single combinational logic block (CLB) if each function uses no more than $f-1$ inputs, and the combined functions have no more than f total inputs. Murgai *et al.* identified the merging problem as finding a maximum matching between mergeable blocks [11], defining two blocks as being *mergeable* if each has at most $f-1$ inputs, the number of common inputs is $f-2$, and the total number of inputs is f. The second requirement is an unnecessary restriction, as two four-input functions with identical inputs can share a five-input CLB.

The matching algorithm used with Xmap is not a true maximum matching algorithm, as such an algorithm would be complicated and could take $n^{2.5}$ steps to do the matching for a network with n logic blocks. Instead, Xmap uses a greedy algorithm to get a good, but usually not maximum, matching.

The algorithm takes a set of logic blocks L and two parameters: i, the maximum number of inputs for each function in a shared CLB, and t, the maximum total number of inputs in a shared CLB. First, all blocks with more than i inputs are removed from L as being unmergeable. The remaining set is sorted by the number of inputs to each block.

The block with the most inputs is removed from L, and mergeability is checked with all remaining blocks, in decreasing order of number of inputs. If a legal merging is found, then the other block is also removed from L. The removal of blocks is continued until L is empty.

Trying to merge blocks with high fan-in first serves two purposes: it helps ensure that easy-to-match blocks (blocks with few inputs) are kept available for later matching, and it reduces routing demands by increasing the number of block inputs that are shared.

Theoretically, the mergeability check could be done as many as $n(n-1)/2$ times, if all n blocks have few enough inputs to look mergeable, but no matches are actually possible. In practice, n is small enough and the inner loop fast enough, that the matching takes insignificant time.

4 Results of benchmarks

This section gives a quick summary of the benchmark results; a full tabulation has been printed in a technical report [7]. Thirty-four benchmark circuits were mapped, including all the benchmarks reported for mis-pga [11] and Chortle [4]. The benchmark circuits were all minimized with misII's standard script before mapping.

Figure 4.1 compares the Xmap algorithm to Chortle [4] for five-input logic functions. The figure plots the ratio of the block counts rather than a simple scatter diagram of the two counts, because the range of sizes would otherwise obscure the differences. The Xmap algorithm consistently outperforms Chortle, even without merging blocks, averaging 6% fewer blocks for five-input functions, and 7% fewer over all benchmarks reported for Chortle.

The Xmap algorithm is significantly faster than Chortle [2]. Overall, Xmap is about 4.5 times faster than Chortle, but Xmap does not slow down as the fan-in limit f increases, while Chortle's time appears to increase linearly with f. For the most useful cases ($f=5$), Xmap is about 7 times faster than Chortle.

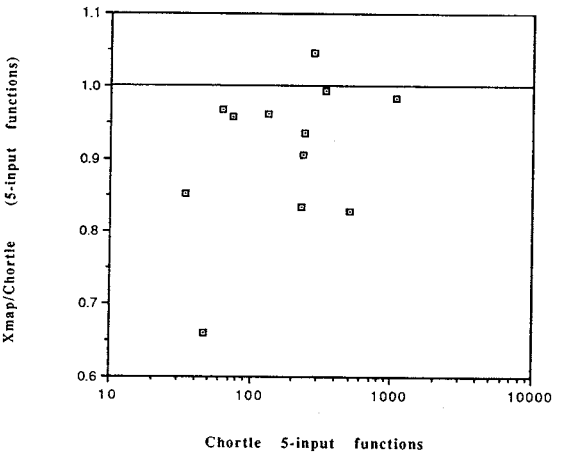

Figure 4.1: The number of five-input functions needed by Xmap before merging divided by the number needed by Chortle for the benchmarks reported in [4].

Francis *et al.* also reported results for using misII to map to a large library of cells that simulated mapping to arbitrary functions [4]. For their benchmarks, Xmap is about 17 times faster than misII, and uses 11% fewer cells.

Figure 4.2 compares Xmap with mis-pga for five-input logic blocks that allow merging [10,11]. The Xmap algorithm uses the same number of blocks as mis-pga before merging, but uses 14% fewer after merging. One might think that Xmap's merging algorithm is better than mis-pga's, but this is unlikely, as mis-pga uses a maximum matcher, and Xmap only a heuristic approximation to one. The difference is probably that Xmap tries to minimize the fan-in of gates, and so more gates are mergeable. Compared to mis-pga, Xmap is about 150 times faster, and manages to complete the merging on all benchmarks, but mis-pga ran out of memory on two examples.

5 Choosing lookup table size

The Xmap algorithm (like Chortle or mis-pga) can be used for choosing gate sizes when designing new table-lookup chips. If bits of truth table cost c_b each, and wires to inputs or outputs cost c_g each, then the cost of using a single-output, f-input block is $c_b 2^f + c_g(f+1)$. Blocks that allow merging have two outputs and cost $c_b 2^f + c_g(f+2)$.

Table 5.1 shows that 5-input blocks with merging are optimal for the arbitrary set of constants chosen. Note that merging becomes more valuable as the block size increases, because more of the blocks are not fully used. The constants were chosen rather arbitrarily, and all the benchmarks were purely combinational logic—other constants and other benchmarks could give quite different results. For example, the results for Chortle favor 4-input rather than 5-input blocks [4].

[2] The CPU times are reported for a SUN 3/80 [7], and include all input and output in BLIF format, as well as construction of the if-then-else DAG and the mapping, but do not include the prior minimization by misII.

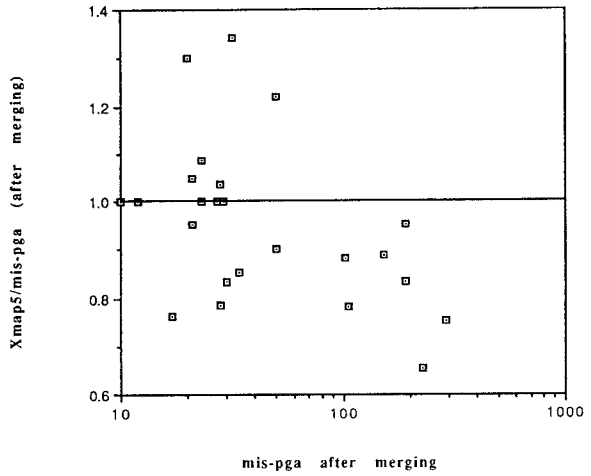

Figure 4.2: The number of five-input functions needed by Xmap after merging divided by the number needed by mis-pga for several benchmarks. Note that Xmap consistently outperforms mis-pga on larger problems.

block type	cost per block	blocks needed	total cost/10^6
2-input	124	12013	1.49
3-input	168	6961	1.17
3-input with merge	208	5983	1.24
4-input	216	5376	1.16
4-input with merge	256	4212	1.08
5-input	272	4411	1.20
5-input with merge	312	3027	0.94
6-input	344	3731	1.28
6-input with merge	384	2479	0.95

Table 5.1: Illustration of method for choosing a logic block size when designing a new gate-array chip. Constants were arbitrarily chosen as $c_b = 1$ and $c_g = 40$. The number of blocks needed is the sum of the numbers found by Xmap for all 34 benchmarks.

6 Future Work

The high speed of Xmap makes it attractive for evaluating minimization techniques. Circuit rearrangement can be done for area or delay reduction, and the entire circuit remapped to evaluate the changes—like the iterative improvement done in mis-pga. The techniques used in Xmap (ignoring the structure of a function, and looking only at how many wires are needed to encode the information) may also be valuable in higher-level logic and state-machine minimization algorithms.

We plan to explore other heuristics for Xmap. For example, the *reduce_fanout* procedure in the first pass could be replaced by a procedure that looks for the minimum vertex cut set of high-fan-out nodes separating the overflow node from its signals set. Better minimum cut and maximal matching algorithms could be used in the final passes.

The results in this paper count only the number of logic blocks used, not estimating routability or delay. Optimizing for cost measures that estimate these parameters would increase the value of the technology mapper. Unfortunately, the delays in programmable gate arrays are heavily dependent on the routing, because of the high resistance of the routing switches or anti-fuses. It will be difficult to find meaningful delay estimates before routing is done, and doing good placement and routing is too expensive to put in the inner loop of a logic minimizer.

Another direction for research is to develop mappers for sequential logic, generating complete mappings including appropriate assignment of register resources. Such a mapper might also try to take advantage of special routing resources, such as Xilinx's internal feedback from the output of a flip-flop to an input of the CLB for the same cell.

Acknowledgements

I'd like to thank Martine Schlag for an afternoon discussion that crystallized some of the ideas in Xmap. Søren Søe read drafts of this article and provided useful comments.

References

[1] M. R. C. M. Berkelaar and J. A. G. Jess. Technology mapping for standard-cell generators. In *ICCAD-88*, pages 470–473, Nov. 1988.

[2] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: a multiple-level logic optimization system. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(6):1062–1081, Nov. 1987.

[3] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology mapping in MIS. In *ICCAD-87*, pages 116–119. IEEE Computer Society Press, Nov. 1987.

[4] R. J. Francis, J. Rose, and K. Chung. Chortle: a technology mapping program for lookup table-based field programmable gate arrays. In 27^{th} *Design Automation Conf.*, pages 613–619, Orlando, FL, June 1990.

[5] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. Socrates: a system for automatically synthesizing and optimizing combinational logic. In 23^{rd} *Design Automation Conf.*, pages 79–85, Las Vegas, NV, 1986.

[6] K. Karplus. Using if-then-else DAGs for multi-level logic minimization. In *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 101–118, March 1989.

[7] K. Karplus. Using if-then-else DAGs to do technology mapping for field-programmable gate arrays. Technical Report UCSC-CRL-90-43, Computer Engineering, Univ. of Calif., Santa Cruz, Sept. 1990.

[8] K. Karplus. Canonical forms of if-then-else dags are robustly path-delay-fault testable. In *International Workshop on Logic Synthesis*, Research Triangle Park, North Carolina, May 1991. submitted.

[9] K. Keutzer. DAGON: Technology binding and local optimization by DAG matching. In 24^{th} *Design Automation Conf.*, pages 341–347, Miami Beach, FL, 1987.

[10] R. Murgai. Electronic mail, August 1990.

[11] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In 27^{th} *Design Automation Conf.*, pages 620–625, Orlando, FL, June 1990.