# Xtmap: Generate-and-Test Mapper for Table-Lookup Gate Arrays

Kevin Karplus*

Board of Studies in Computer Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
(408)459-4250 Internet: karplus@ce.ucsc.edu

## Abstract

*Technology mapping is the process of taking a fine-grain network describing a multiple-output logic function, and covering it with cells to get a network that is legal in a given technology. The goals of the mapping are to produce small, fast, and testable circuits.*

*This paper introduces Xtmap, a new technology mapper for f-input table-lookup cells based on a generate-and-test paradigm. Xtmap can optimize for area and delay simultaneously , and produces smaller circuits than previous mappers that considered delay, while matching their delay values.*

*Tables of benchmark results compare Xtmap with Xmap, Dagmap, FlowMap, chortle-d, and mis-pga.*

**Keywords:** Combinational logic synthesis, technology mapping, FPGAs, Xilinx

## 1 Technology mapping

Most logic minimizers have two parts: a technology-independent minimizer that tries to simplify the logic without directly considering what technology will be used, and a technology-dependent minimizer that tries to find a good implementation in a specific technology. The technology-dependent phase of minimization usually consists of a decomposition into a fine-grain network, followed by a technology mapper that covers the fine-grain network with legal cells.

This paper will concentrate only on the last part—covering the fine-grain network. The logic minimizer that the technology mapper Xtmap is embedded in (ITEM) uses a fine-grain network representation for the higher-level minimization steps, and so further decomposition is not needed. To ensure that the technology mapping will always complete successfully, the target technology must be able to implement each gate of the fine-grain network as a single cell. That is, the initial network must already be a *feasible* (if not particularly good) mapping to the target technology.

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

The Xtmap mapper (like Xmap and Xcmap) is a *faithful* mapper, in that it makes no rearrangement of the fine-grain network. However, it does allow limited duplication of nodes in the network, in that cells are allowed to overlap.

Early work in technology mapping concentrated on mapping to cell libraries [14, 2, 7], but more recent work has focussed on mapping to the uniform cells of field-programmable gate arrays (FPGAs) [15, 8, 9, 5, 4, 3, 12, 11].

Cell-library-based mappers do not work particularly well when mapping to the flexible cells of FPGAs. The usual technique for using them is to create dummy cell libraries, where each library entry is one way to configure a cell in the gate array. The cell-library approach allows older 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. If only a restricted subset of the possible configurations is used, the size of the library can be controlled, but the quality of the mapping suffers.

## 2 Generate-and-test mapping

Covering algorithms have generally fallen into three groups: input-first greedy algorithms (Xmap [12]) output-first greedy algorithms (Amap [11]), dynamic programming (Dagon [14], mis [7], chortle [8], Dagmap [5], FlowMap [4], Rmap [18]). An *input-first* algorithm maps the inputs to a gate before choosing the gate, while an *output-first* algorithm chooses a gate before mapping its inputs. A greedy algorithm is one that enumerates a few alternatives, and commits to one on the basis of simple heuristics. A dynamic programming algorithm chooses between several different local choices, based on the value of different partial implementations (for a good description of dynamic programming, see [6, Chapter 16]).

This paper introduces a mapper using a slightly different paradigm: generate-and-test.

Xtmap is an output-first algorithm based on a generate-and-test paradigm. The algorithm starts at an output, and generates many ways for that output to be implemented in a single cell, covering some portion of the fine-grain network. Each of the candidate implementations is evaluated (tested) using a simple heuristic, and the best one chosen. The inputs of the new cell then are added to the set of functions that need to be mapped. Figure 1 gives simplified pseudocode for the algorithm. In the actual implementation, mappings are chosen recursively for the inputs of the new cell, and the ToMap set is not explicitly represented.

To complete the description of a generate-and-test mapper, three subroutines have to be described:

- the method for choosing which node of the fine-grain network to map next (Section 2.1),
- the heuristics for determining the value of a cell (Section 2.2), and
- the generator for single cells (Section 2.3).

## 2.1 Choosing the next node to map

Xtmap has a top-level loop that iterates over the principal outputs, and uses a recursive call to the mapper for each input to a newly created cell. Because the ToMap set is implicitly represented on the recursion stack, it need not be explicitly represented. No attempt is made to choose the order in which the principal outputs or the inputs to a new cell are mapped.

## 2.2 Value of an FPGA cell

The basic assumption of Xtmap, as with any greedy mapper, is that a series of locally optimal choices (with respect to the heuristic functions) will result in a globally good solution. The heuristic functions need to capture any interaction between cells that is likely to be important in the final solution.

One of the biggest advantages in mapping to FPGAs, rather than to cell library technologies, is that all cells cost the same, and so we only need to look at how well we cover the fine-grain network.

One key simplification of the generate-and-test paradigm is that only the generator needs to know about the function of a cell—the function itself does not matter in determining the value of the cell. Instead, the cell can be evaluated only on the basis of which nodes in the fine-grain networks are inputs to the cell, and which are *hidden*—that is, which nodes are covered by the cell but are neither inputs nor outputs of the cell.

The value of a cell is computed as the sum of weight functions computed on the inputs and hidden set, as shown in Figure 2.

One would expect it to be more useful to use already mapped nodes in the input set, than to have to map new nodes, particularly new ones with low fanout. This means that we would expect to set the weights to meet the constraints $a_1 > a_2$, $a_2 < 0$, and $a_3 < 0$. Fanout appears in a denominator, because the difference between a fanout of 1 and 2 is much more significant than a difference between 11 and 12, this means that a negative value for $a_3$ gives a fairly large penalty for a mapping with low-fanout inputs, and a much smaller penalty to ones with high-fanout inputs. Furthermore, a mapping for a cell is generally better if the size of the portion of the network remaining to map is less. If we use the sum of the area estimates for the inputs as an estimate of how much remains to be mapped, we would expect $a_4 < 0$.

There seems to be little advantage to using up part of a cell to re-implement already mapped nodes, and so we would expect to have $b_1 < 0$. For hidden nodes, one would expect it to be best to hide nodes that haven't been implemented, particularly if they have low fanout. Thus one would expect to have $b_2 > 0$ and $b_3 > 0$.

To minimize delay in the circuit, the DelayWeight should be set to a negative number. The expected constraints on the weights were usually met by sets of parameters found by the learning algorithms described in Section 3, but all were violated by at least one of the weight sets found by the random neighborhood search learning technique. The values used in the results section ($a_1 = 0$, $a_2 = -30$, $a_3 = -14$, $a_4 = -11$, $b_1 = 1$, $b_2 = 5$, $b_3 = 5$, DelayWeight $= -256$) were found by learning on one network, and discovered to work reasonably well on many networks. The only expected constraint that is violated is that $b_1$ is positive, instead of negative. No attempt has been made to see if the results are improved by reducing $b_1$.

The quality of the delay estimate *arrival* is important for producing good mappings. At first, the algorithm just used the longest distance from a principal input to the node in the fine-grain network, but this did not prove to be a particularly good predictor of delay, and the circuits produced had larger delays than those produced by Chortle and Dagmap.

An improved delay estimate was obtained by doing an initial mapping of the fine-grain network using Xcmap (a re-implementation of the main algorithm of Dagmap), and recording the *lutheight*: the height in the mapped circuit. By setting the delay weight to a sufficiently large negative number, the generate-and-test algorithm can be forced to produce circuits whose delay (as measured by lutheight) is as small as that produced by Xcmap, as long as the generator for the node is guaranteed to try the set of inputs used by Xcmap (Xtmap does try that set of inputs, as described in Section 2.3).

```
ToMap ← Principal outputs
while ToMap ≠ ∅
    Choose node n from ToMap and delete it
    For each generated single-cell implementation of n
        Compute value for implementation of n
    Implement highest value cell
    Add inputs of cell (that aren't already mapped) to ToMap
```

Figure 1: Pseudocode for the generate-and-test technology mapping paradigm.

$$\begin{aligned}
\mathrm{Value}(\mathrm{in_set}, \mathrm{hid_set}) &= \sum_{i \in \mathrm{in_set}} \mathrm{inweight}(i) + \sum_{h \in \mathrm{hid_set}} \mathrm{hidweight}(h) \\
&\quad + \mathrm{DelayWeight} \max_{i \in \mathrm{in_set}} \mathrm{arrival}(i) \\
\mathrm{inweight}(i) &= a_4 \,\mathrm{areaest}(i) + a_3/\mathrm{fanout}(i) + \begin{cases} a_1 & \text{if } i \text{ is already mapped} \\ a_2 & \textit{otherwise} \end{cases} \\
\mathrm{hidweight}(h) &= b_3/\mathrm{fanout}(h) + \begin{cases} b_1 & \text{if } h \text{ is already mapped} \\ b_2 & \textit{otherwise} \end{cases}
\end{aligned}$$

Figure 2: Formula for computing the value of a cell. Of the many candidates for implementing a particular node in the network, the one with the highest value will be selected. The function arrival($i$) estimates the arrival time for the signal at node $i$; the fanout($i$) function counts the number of nodes that use $i$ as an input in the fine-grain network; and the areaest($i$) estimates the area needed to map the input. Currently, arrival is hardwired to be lutheight, and areaest is hardwired to be the count metric (an area estimator described in [10]).

### 2.3 Generator for table-lookup cells

The generator needs to produce a number of feasible single-cell implementations for a node in the fine-grain network. The approach I have taken is to write a simple generator for each different FPGA technology, rather than a general-purpose generator to use with a variety of technologies. However, this is not essential to the concept, and a general-purpose generator could be written, perhaps along the lines of Proserpine [1]. A generator has been written for $f$-input table-lookup functions, and other generators are being written for other target FPGA technologies.

For table-lookup cells, the generator can be fairly simple, as the function of the cell is irrelevant—all that matters is how many inputs are needed. The generator for Xtmap tries to produce all possible vertex cut sets of size $f$ or less separating the node being mapped from the primary inputs using a simple depth-first search algorithm.

Simplified pseudocode for the generator is given in Figure 3. The recursive procedure takes two parameters: a set of nodes to consider as inputs to the cell and a set of nodes that are not to be expanded in the search. The nodes in the input set are replaced one at a time by their own inputs, until the complete input set gets too large. After a node has been *expanded* in this way, it is marked so that it will not be expanded again, to avoid repeated enumeration of the same cut sets. The set of hidden nodes (hid_set) keeps track of the nodes that have been removed from the input set, for use in determining the value of the cell (see Section 2.2).

The actual algorithm used in Xtmap is slightly more complicated. One simple but important modification is that unnecessary inputs are removed from in_set, before in_set is counted and the cell is evaluated. An input $i$ is considered to be *unnecessary* if inputs($i$) $\subset$ in_set. This optimization is inefficiently implemented, and accounts for a large part of Xtmap's running time.

Note that not all vertex cut sets with $|\mathrm{in\_set}| \leq f$ are enumerated by this simple algorithm, as some vertex cut sets may require intermediate cut sets with more than $f$ nodes, and these would be missed.

Another, more important improvement can be made to the generator by adding an alternative definition of the inputs of a node. Both the Xmap and Xcmap

```
TryExpansions(set in_set, set noexpand, set hid_set)
    if |in_set| > f then return
    EvaluateCell(in_set, hid_set)
    for each input i in in_set−noexpand
        TryExpansions(in_set − {i} ∪ inputs(i), noexpand, hid_set ∪ {i})
        noexpand ← noexpand ∪ {i}
```

Figure 3: Simplified pseudocode for generating trial cells for $f$-input lookup tables. The generator is called initially with TryExpansions(inputs($n$), $\emptyset$, $\emptyset$). EvaluateCell is called to test each trial cell.

greedy mappers compute legal ($|\text{in\_set}| \leq f$) vertex cut sets for every node in the fine-grain network. These cut sets can be used as alternatives to inputs($i$) both in the TryExpansions algorithm and in defining the unnecessary nodes of in_set. Updating the set of hidden nodes requires a little more work than before, as all the nodes between $i$ and the stored vertex cut are hidden, not just $i$.

Even with the improvements,the algorithm misses some $f$-cuts, and these missed cuts are often the most interesting ones for high-quality mapping, since they are generally lower in the network than the cuts that are enumerated. The algorithm could be improved by using the techniques of FlowMap [4] or Rmap [18] to ensure that all $f$-cuts are enumerated.

The pseudocode for the version of TryExpansions used to produce numbers for the benchmarks is shown in Figure 4. The function lutinputs($i$) looks up a legal cut set for $i$ found by the algorithm for Xcmap (unless Xmap found a smaller delay for the node, in which case the cut set found by Xmap is used).

Because the Xcmap (Dagmap) algorithm guarantees minimum delay implementation for tree circuits and for nodes that can be implemented in a single cell directly from the principal inputs, adding the lutinputs expansions causes the generator to catch many of the interesting possible cells that it otherwise would have missed.

## 3 Learning the weights for the heuristics

My original hope was that a good set of weights could be chosen and fixed for all circuits, but I was unsuccessful in finding a good set of weights manually, and so I implemented learning algorithms to try to find good weights. Two different learning algorithms have been implemented: random neighborhood search and learning from an existing mapping.

The random neighborhood search algorithm is slow, but effective. Xtmap is run repeatedly with different values for the weights, and the goodness of each mapping determined. New values are generated by perturbing the values that produced the best mapping so far.

Goodness may be number of cells, delay, or area-delay product, or any similar function—I usually use the number of Xilinx cells multiplied by the delay of the circuit with a unit-delay model. Other area measures, such as the number of lookup tables before merging or the pin count, are available in ITEM, as are other delay measures (such as the measure proposed in [17]). The goodness measures available in the Xtmap implementation are any of the area or delay measures available in ITEM, or the product of any two measures.

After tuning the weights on a few circuits, I added a further feature that kept a list of good settings for weights. For each new circuit, all the settings already recorded were tried first, and then some number of random searches were made, adding a new setting to the list if a better was found by the random search. I had hoped that pruning the list would leave me with a few good settings that would work for many circuits. Unfortunately, running 45 benchmark circuits resulted in 25 different weight settings, each of which was better than any of the others for at least one circuit.

The second learning approach was based on a very different philosophy. Instead of blindly searching for a setting by looking at the results of the mapping, I tried to derive a setting from a good mapping. I started with an area-efficient mapping from Xmap, then ran the cell-generation procedure for each node that was used in that mapping. Instead of evaluating each potential cell, it was compared with the cell found by Xmap. Weights were increased or decreased so as to favor the cell found by Xmap over the candidate one. For example, if the generated cell had more hidden nodes that were already mapped, then $b_1$ would be decreased, but if it had fewer, $b_1$ would be increased.

Running Xtmap with the parameters learned in this way produced circuits similar to those found by Xmap (small, but not fast), but was, of course, much more expensive to run. Similarly, running Xtmap with the delay weight set to a large negative number duplicated

```
TryExpansions(set in_set, set noexpand, set hid_set)
    remove unnecessary nodes from in_set
    if |in_set| > f then return
    EvaluateCell(in_set, hid_set)
    for each input i in in_set−noexpand
        TryExpansions(in_set − {i} ∪ inputs(i), noexpand, hid_set ∪ {i})
        TryExpansions(in_set − {i} ∪ lutinputs(i), noexpand, hid_set ∪ hidden(i, lutinputs(i)))
        noexpand ← noexpand ∪ {i}
```

Figure 4: Pseudocode for generating trial cells for $f$-input lookup tables. It is called initially with TryExpansions(inputs($n$), $\emptyset$, $\emptyset$) and TryExpansions(lutinputs($n$), $\emptyset$, $\emptyset$). The function lutinputs($i$) looks up a legal cut set for $i$ found by a greedy mapper (such as Xcmap or Xmap).

the mapping found by Xcmap (fast, but not small). The advantage of Xtmap is that the delay weight can be set to intermediate values, getting circuits that are both small and fast.

In order to create repeatable results, the benchmark results in this paper did not use learning. Instead, a single parameter set was used, and the penalty for delay increased by factors of two until the unit-delay from Xtmap was as small as from the Xcmap mapping (see Table 1). This single parameter set seems to produce results very similar to (but slightly better than) doing the deterministic learning procedure for each benchmark. Based on earlier benchmark runs, following this with 10 repetitions of random search would probably have improved the results slightly, but increased the running time substantially.

One interesting observation was that running Xtmap twice for the `des` benchmark with the same parameters produced different results. The explanation is that the arrival time estimate `lutheight` is not just the result of running the Xcmap algorithm, but is updated after every mapping, so that it represents the lowest known height. For some nodes in `des`, Xtmap finds a better mapping, and updating the lutheight for these nodes results in an improved mapping when Xtmap is re-run.

## 4 Conclusions and Future Work

Table 1 gives some benchmark results for some of the recently published technology mappers, all being run on identical networks. Xtmap was run with the parameters $a_1 = 0$, $a_2 = -30$, $a_3 = -14$, $a_4 = -11$, $b_1 = 1$, $b_2 = 5$, $b_3 = 5$, DelayWeight $= -256$. For each mapper, the number of 5-input lookup tables and the unit delay are given. Although Xtmap appears to be about as good as the best previously published mapper (FlowMap), the unpublished results from FlowMap-r seem to be better.

For delay minimization, the best previous results were reported for FlowMap [4], and for area minimization mis-pga [15], but mis-pga gets much of its area optimization from high-level optimization and decomposition, rather than from mapping per se. Although the Xcmap (Dagmap) algorithm produces optimal delay for trees, on one highly reconvergent example (not in the table) Xmap produces a circuit with fewer levels of lookup tables. FlowMap produces optimal delay on arbitrary networks.

Xtmap usually gets delay values as good as FlowMap, and with about the same area penalty.

The cpu time reported in Table 1 is for reading the BLIF or EQN input file and doing all three mappings on a Sparcstation SLC. The slowest operation is Xtmap itself, which is much slower than simple greedy mappers, primarily because of the cost of evaluating all the potential cells that are not used. Xtmap is significantly slower than Xcmap or Xmap, but not as slow as chortle-d. (I do not have timing information for FlowMap, and so cannot compare execution times of Xtmap and FlowMap with it.) Improving Xtmap's cut-enumeration algorithm should improve both Xtmap's speed and performance. Improving the arrival time estimates (perhaps using FlowMap to compute lutheight) would also improve performance.

I believe that technology mapping efforts should focus on producing *faster* mappers, rather than *better* mappers, because it is probably more useful to spend the cpu time on improving higher-level optimization.

To illustrate this, some results are reported for techniques that high-level optimization before (or during) mapping. Table 2 reports results for the same benchmarks as Table 1. The cputime reported in Table 2 does not include the optimization time, just the mapping time. The three "X" mappers were run on networks created by running two iterations of the ITEM script shown in Figure 5 on the benchmark files used for the FlowMap results, which is a better starting point than used for the comparisons in Table 1. Those

| file | Xcmap | Xmap | Xtmap | cpu | FlowMap-r [3] | FlowMap [4] | Dagmap [5] | chortle-d [9] |
|---|---|---|---|---|---|---|---|---|
| 5xp1 | 30:3 | 25:4 | 25:3 | 2.0 | *23:3* | 25:3 | 28:3 | 26:3 |
| 9sym | 75:5 | 66:9 | 74:5 | 4.3 | *61:5* | *61:5* | 63:5 | 63:5 |
| 9symml | 66:5 | 62:9 | 65:5 | 4.0 | *58:5* | *58:5* | 61:5 | 59:5 |
| C499 | 252:5 | 136:7 | 186:5 | 220.5 | *151:5* | 154:5 | 204:5 | 382:6 |
| C880 | 263:8 | 181:13 | 231:8 | 64.9 | *211:8* | 232:8 | 246:8 | 329:8 |
| alu2 | 181:8 | 136:14 | 154:8 | 63.0 | *148:8* | 162:8 | 199:9 | 227:9 |
| alu4 | 334:10 | 238:17 | 270:10 | 216.3 | *245:10* | 268:10 | 303:10 | 500:10 |
| apex6 | 319:4 | 238:7 | 267:4 | 32.5 | *232:4* | 257:4 | 284:5 | 308:4 |
| apex7 | 114:4 | 80:6 | 88:4 | 6.5 | *80:4* | 89:4 | 95:4 | 108:4 |
| count | 100:4 | 65:5 | *71:3* | 5.1 | 73:4 | 76:3 | 87:3 | 91:4 |
| des | 1690:6 | 1054:10 | 1212:6 | 458.9 | *1087:5* | 1308:5 | 1480:6 | 2086:6 |
| duke2 | 218:4 | 158:8 | *187:4* | 17.6 | *187:4* | *187:4* | 195:4 | 241:4 |
| misex1 | 19:2 | 18:3 | 19:2 | 1.5 | *15:2* | *15:2* | 17:2 | 19:2 |
| rd84 | 48:4 | 45:7 | 49:4 | 5.7 | *43:4* | *43:4* | 48:4 | 61:4 |
| rot | 325:6 | 233:11 | 269:6 | 17.9 | *243:6* | 268:6 | 328:6 | 326:6 |
| vg2 | 54:4 | 43:5 | 49:4 | 4.1 | 38:4 | 45:4 | *42:3* | 55:4 |
| z4ml | 20:3 | 12:4 | 16:3 | 1.2 | *13:3* | *13:3* | 17:3 | 25:3 |
| total | 4108:85 | 3107:139 | 3227:84 | | *2908:83* | 3261:83 | 3543:85 | 4906:87 |

Table 1: The columns headed with mapper names report the number of 5-input lookup tables and unit-delay estimate of delay for that mapper. The cpu time (on a Sparcstation SLC) is the total time for the all three mappings. Xcmap is a re-implementation of the main algorithm of Dagmap, while the Dagmap column has the numbers from the original paper [5]—the numbers differ because Xcmap is not allowed to rearrange associative operators. Xtmap was set to choose the minimum product of unit-delay and the number of tables, while sweeping the delay penalty. All the algorithms were run on the same initial network. Italics are used to mark the best area-delay product in each row.

benchmarks had been created by optimizing the benchmarks for delay, then decomposing with the DMIG algorithm [5].

The ITEM optimization script does not have any notion of critical paths, and attempts to reduce delay on all outputs, even non-critical ones. An optimization technique that concentrated on area reduction off the critical paths, and delay reduction near the critical path would probably produce better area-delay products—even so, this technique did produce better mappings for several benchmarks than any previously reported results.

Although simple greedy algorithms have been getting fairly good results, other paradigms for technology mapping remain to be examined. I have a student now investigating using ratio-cut partitioning to map to $f$-input lookup tables, so that more global information can be used in deciding which nodes in the fine-grain network are worth implementing as cell outputs. We hope to get more area-efficient mappings with this technique.

## 4.1 Mapping for routability

Most mapper research has focussed on area or delay minimization, but on the Xilinx chips, the routing resources are usually the limiting factor on any design. There has been some previous work on mapping for routability (notably Rmap [18]), but the available place-and-route tools have random enough behavior that it is difficult to compare the qualities of different mappers just by placing and routing the circuits. One measure that I believe will be useful is the total pin count (number of connections to cells) of a mapping (after merging into two-output cells). Table 3 reports this measure, and a few other potentially useful ones for the Xtmap mappings reported in Table 1. Placement and routing experiments have not yet been done, but I believe that using Xtmap to optimize for pin count and fanout-weighted delay will produce faster, more routable circuits than available mappers (other than Rmap).

Pak Chan believes that pins per CLB is a more important measure, and that the mappings produced by my mappers and most other published mappers are too dense to be routed consistently on Xilinx 3000 chips [personal communication]. Using the information from Table 3, Xtmap averages 6.1 pins per CLB (the pin count column includes primary inputs and outputs, which must be subtracted out before dividing by the CLB columns). According to Dr. Chan, mappings with more than 6 pins per CLB are nearly always unroutable, but ones with 5 or fewer are nearly always routable. If further experimentation bears out his observation, then Xtmap will modified to control the pin per CLB ratio as well. Some simple controls are already possible, such as lying to ITEM about the

```
# use best variable ordering heuristic (expensive)
            order split -c count
# use local factor to minimize edge count* height
            transform -m local -a edges -d height
# do block covering to minimize delay (lutheight of ite DAG)
            bcov -m tox -f lutheight bcdelay -s bcvalue bcarea
# redo local factor this time minimizing edge count*lutheight
            transform -m local -a edges -d lutheight
```

Figure 5: Optimization script used with ITEM to create the networks mapped by the "X" mappers in Table 2. Two iterations of this optimization script were run on each network. The block-covering technique is described in [19], and a rough description of local factor is in [13], but the variable ordering heuristic has not been published yet [20].

| file | Xcmap | Xmap | Xtmap | cpu | FlowMap [4] | mis-pga [16] |
|---|---|---|---|---|---|---|
| 5xp1 | 21:3 | 20:4 | 20:3 | 1.2 | 22:3 | *21:2* |
| 9sym | 8:3 | 8:3 | 8:3 | 0.6 | 60:5 | *7:3* |
| 9symml | 8:3 | 8:3 | 8:3 | 0.6 | 55:5 | *7:3* |
| †C499 | 90:4 | 78:6 | 84:4 | 9.7 | *68:4* | 199:8 |
| C880 | 151:9 | 112:15 | 143:9 | 10.3 | *124:8* | 259:9 |
| alu2 | 58:4 | 61:7 | *56:4* | 3.0 | 155:9 | 122:6 |
| alu4 | 182:7 | 156:8 | *167:7* | 8.1 | 253:9 | 155:11 |
| apex6 | 263:4 | 194:7 | *234:4* | 12.8 | 238:5 | 274:5 |
| apex7 | 94:4 | 64:6 | 87:4 | 4.1 | *79:4* | 95:4 |
| count | 48:3 | 33:4 | *45:3* | 1.9 | 31:5 | 81:4 |
| †des | 1494:6 | 909:11 | *1020:5* | 149.7 | 1310:5 | 1397:11 |
| duke2 | 190:4 | 144:8 | *166:4* | 15.3 | 174:4 | 164:6 |
| misex1 | 17:2 | *14:2* | 15:2 | 1.3 | 16:2 | 17:2 |
| rd84 | 16:3 | 14:3 | 16:3 | 0.9 | 46:4 | *13:3* |
| rot | 309:6 | 223:11 | 275:6 | 15.9 | *234:7* | 322:7 |
| vg2 | 33:4 | 23:6 | 32:4 | 1.9 | *29:3* | 39:4 |
| z4ml | *5:2* | *5:2* | *5:2* | 0.4 | *5:2* | 10:2 |
| total | 2987:71 | 2066:106 | *2381:70* | | 2899:84 | 3182:90 |

Table 2: This table reports the results of mapping after high-level optimization for delay. The Xcmap, Xmap, and Xtmap results are from optimizing the same starting points as used for the FlowMap results using an ITEM script that optimizes for the product of edge count and lutheight. Results for des are for running the optimization script on the same starting point as Table 1, because the minimizer ran out of memory minimizing on the DMIG-optimized network. Results for C499 are from a less expensive script that did not require canonical forms, because the script of Figure 5 was unable to complete on C499. The cpu time column (seconds on a Sparcstation SLC) counts only the mapping, not the optimization, which took anywhere from seconds to 2 hours. FlowMap and mis-pga were run with different starting points, and so the quality of the optimization is really being compared, rather than the quality of the mappers. Italics are used to mark the best area-delay product in each row. Note that for rot the best result in Table 1 is better. The result for count can be improved to *39:3* and C880 to 145:7 by running the optimization script on the original benchmark network, rather than an already optimized and decomposed one; apex7 to *100:3* by optimizing the network used in Table 1; and vg2 to *28:3* by using the cheap optimization script used for C499, rather than the expensive one.

| | | | from Table 1 | | | | | from Table 2 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| file | inputs | outputs | tables | CLBs | depth | delay estimate | pin count | tables | CLBs | depth | delay estimate | pin count |
| 5xp1 | 7 | 10 | 25 | 19 | 3 | 73.50 | 134 | 20 | 14 | 3 | 64.50 | 103 |
| 9sym | 9 | 1 | 74 | 52 | 5 | 120.90 | 332 | 8 | 8 | 3 | 61.70 | 58 |
| 9symml | 9 | 1 | 65 | 52 | 5 | 132.90 | 324 | 8 | 8 | 3 | 61.70 | 58 |
| C499 | 41 | 32 | 186 | 158 | 5 | 121.15 | 1019 | 84 | 74 | 4 | 87.90 | 492 |
| C880 | 60 | 26 | 231 | 173 | 8 | 140.15 | 1147 | 143 | 107 | 9 | 144.65 | 742 |
| alu2 | 10 | 6 | 154 | 119 | 8 | 215.60 | 743 | 56 | 49 | 4 | 131.70 | 303 |
| alu4 | 14 | 8 | 270 | 211 | 10 | 276.65 | 1302 | 167 | 116 | 7 | 231.00 | 740 |
| apex6 | 135 | 99 | 267 | 210 | 4 | 292.10 | 1502 | 234 | 193 | 4 | 304.20 | 1375 |
| apex7 | 49 | 37 | 88 | 61 | 4 | 127.30 | 460 | 87 | 58 | 4 | 119.70 | 442 |
| count | 35 | 16 | 71 | 52 | 3 | 121.20 | 367 | 45 | 33 | 3 | 92.20 | 249 |
| des | 256 | 245 | 1212 | 964 | 6 | 1131.20 | 6403 | 1020 | 716 | 5 | 749.00 | 4915 |
| duke2 | 22 | 29 | 187 | 134 | 4 | 127.80 | 877 | 166 | 127 | 4 | 146.40 | 825 |
| misex1 | 8 | 7 | 19 | 13 | 2 | 57.30 | 93 | 15 | 12 | 2 | 57.30 | 87 |
| rd84 | 8 | 4 | 49 | 38 | 4 | 91.60 | 240 | 16 | 12 | 3 | 62.70 | 83 |
| rot | 135 | 107 | 269 | 183 | 6 | 144.50 | 1388 | 275 | 189 | 6 | 142.05 | 1412 |
| vg2 | 25 | 8 | 44 | 29 | 4 | 72.75 | 205 | 32 | 22 | 4 | 70.00 | 166 |
| z4ml | 7 | 4 | 16 | 11 | 3 | 62.20 | 71 | 5 | 4 | 2 | 48.60 | 34 |
| total | 830 | 640 | 3227 | 2479 | 84 | 3308.80 | 16607 | 2381 | 1742 | 70 | 2575.30 | 12084 |

Table 3: More metrics for the mappings produced by Xtmap for Tables 1 and 2 The second and third columns are the number of inputs and outputs from the circuit. Within each section the first column is the number of lookup tables used in the mapping. The second column is the number of 2-output Xilinx 3000 cells needed after merging tables. The third is the longest path from an input to an output. The fourth is a fanout-weighted delay estimate [17]. The final column is the total number of inputs and outputs used on all CLBs, plus the inputs and outputs. Averaged over all the examples, the Xtmap mappings use 1.33 outputs per cell, and 4.77 inputs per cell.

number of inputs allowed to a CLB, or having Xtmap searches minimize the pin count (ignoring the number of CLBs).

Another direction for future research is to explore simultaneous technology mapping and placement, so that routable circuits are produced.

### 4.2 Mapping to other cells

I plan to implement generators for generate-and-test mapping to Actel's Act1 and Act2 cells. A preliminary version was written over a year ago, and looked quite promising, but has not yet been translated into C++ for inclusion in the new system.

When mapping to cells other than table-lookup cells, the input set becomes a multiset (since inputs may need to be duplicated). I've also added a few more weights to the heuristics, to account for constant inputs, duplicate inputs, and inputs whose complements are mapped, even though the inputs themselves are not. That is, the definition of inweight has been modified as shown in Figure 6.

Other generators are planned, including one for Quicklogic's multiple-output cell. For multiple output cells, I will probably add three more weights for the output nodes of the cell, similar to the weights for the hidden nodes.

## Acknowledgements

I would like to thanks the students who have helped create ITEM, the logic minimization system in which Xtmap is embedded, especially Søren Søe and Mehrdad Parsa. I would also like to thank Jason Cong and Eugene Ding for sharing their starting networks and unpublished results.

## References

[1] Alessandro Bedarida, Silvia Ercolani, and Giovanni De Micheli. A new technology mapping algorithm for the design and evaluation of fuse/antifuse-based field-programmable gate arrays. In *FPGA'92: First International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, pages 103–108, Berkeley, CA, 16–18 February 1992.

[2] M. R. C. M. Berkelaar and J. A. G. Jess. Technology mapping for standard-cell generators. In *IEEE International Conference on Computer-Aided Design ICCAD-88*, pages 470–473, Santa Clara, CA, 7–10 November 1988.

[3] Jason Cong and Yuzheng Ding. On area/depth trade-off in LUT-based FPGA technology mapping. UCLA unpublished, 23 October 1992.

[4] Jason Cong and Yuzheng Ding. An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. In *IEEE International Con-*

$$
\text{inweight}(i) = a_4\,\text{areaest}(i) + a_3/\text{fanout}(i) + \begin{cases} a_5 & \text{if } i \text{ is constant 0 or 1} \\ a_6 & \text{if } i \text{ is a duplicate input} \\ a_1 & \text{if } i \text{ is already mapped} \\ a_7 & \text{if } i' \text{ is already mapped} \\ a_2 & \text{otherwise} \end{cases}
$$

Figure 6: Parametric value of an input to a cell for mapping to cells other than table-lookup ones. The values of all inputs are added together, as in Figure 2.


*ference on Computer-Aided Design ICCAD-92*, pages 48–53, Santa Clara, CA, 8–12 November 1992.

[5] Jason Cong, Andrew Kahng, Peter Trajmar, and Kuang-Chien Chen. Graph based FPGA technology mapping for delay optimization. In *FPGA'92: First International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, pages 77–82, Berkeley, CA, 16–18 February 1992.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill and MIT Press, 1990.

[7] Ewald Detjens, Gary Gannot, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert Wang. Technology mapping in MIS. In *IEEE International Conference on Computer-Aided Design ICCAD-87*, pages 116–119, Santa Clara, CA, 9–12 November 1987. IEEE Computer Society Press.

[8] Robert J. Francis, Jonathan Rose, and Kevin Chung. Chortle: a technology mapping program for lookup table-based field programmable gate arrays. In *ACM IEEE* $27^{th}$ *Design Automation Conference Proceedings*, pages 613–619, Orlando, FL, 24–28 June 1990.

[9] Robert J. Francis, Jonathan Rose, and Zvonko Vranesic. Technology mapping of lookup table-based FPGAs for performance. In *IEEE International Conference on Computer-Aided Design ICCAD-91*, pages 568–571, Santa Clara, CA, 11–14 November 1991.

[10] Kevin Karplus. Using if-then-else DAGs for multi-level logic minimization. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 101–118, Pasadena, CA, 20-22 March 1989.

[11] Kevin Karplus. Amap: a technology mapper for selector-based field-programmable gate arrays. In *ACM IEEE* $28^{th}$ *Design Automation Conference Proceedings*, pages 244–247, San Francisco, CA, 17–21 June 1991.

[12] Kevin Karplus. Xmap: a technology mapper for table-lookup field-programmable gate arrays. In *ACM IEEE* $28^{th}$ *Design Automation Conference Proceedings*, pages 240–243, San Francisco, CA, 17–21 June 1991.

[13] Kevin Karplus. Item: an if-then-else minimizer for logic synthesis. In *EuroASIC92*, pages 2–7, 1–5 June 1992.

[14] Kurt Keutzer. DAGON: Technology binding and local optimization by DAG matching. In *ACM IEEE* $24^{th}$ *Design Automation Conference Proceedings*, pages 341–347, Miami Beach, FL, 28 June–1 July 1987.

[15] Rajeev Murgai, Yoshihito Nishizaki, Narendra Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *ACM IEEE* $27^{th}$ *Design Automation Conference Proceedings*, pages 620–625, Orlando, FL, 24–28 June 1990.

[16] Rajeev Murgai, Narendra Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Performance directed synthesis for table look up programmable gate arrays. In *IEEE International Conference on Computer-Aided Design ICCAD-91*, pages 572–575, Santa Clara, CA, 11–14 November 1991.

[17] Martine Schlag, Pak Chan, and Jackson Kong. Empirical evaluation of multilevel logic minimization tools for a field programmable gate array technology. In *Proceedings of the First International Workshop on Field Programmable Logic and Applications*, pages 201–213, September 1991.

[18] Martine Schlag, Jackson Kong, and Pak K. Chan. Routability-driven technology mapping for lookup-table-based FPGAs. In *1992 IEEE International Conference on Computer Design*, pages 86–90, October 1992.

[19] Søren Søe and Kevin Karplus. Logic minimization using two-column rectangle replacement. In *ACM IEEE* $28^{th}$ *Design Automation Conference Proceedings*, pages 470–473, San Francisco, CA, 17–21 June 1991.

[20] Søren Søe and Kevin Karplus. A new variable-ordering heuristic for binary decision diagrams and if-then-else dags. In *ACM IEEE* $30^{th}$ *Design Automation Conference Proceedings*, Dallas, Texas, June 1993. submitted.