Smart enumeration: a systematic approach to exhaustive search

Tim Todman, Haohan Fu, Brittle Tsoi, Oskar Mencer, and Wayne Luk

Department of Computing,
Imperial College London
180 Queen's Gate, London SW7 2BZ, England
{tjt97, hfu, khtsoi, oskar, wl}@doc.ic.ac.uk

Abstract. This paper explores the potential of smart enumeration: enumeration of a design space giving the effect of exhaustive search, while using heuristics to order and reduce the search space. We characterise smart enumeration as having several key properties, including carefully chosen problem domains and techniques to speed up the search, such as those that exploit symmetry. Our approach has been applied to technology mapping, optimising area and power consumption.

1 Introduction

Many application domains in computer science contain problems which are essentially searches across very large spaces. These searches often take time exponentially proportional to the problem size, so work has concentrated on approximate algorithms which give a good-enough result in reasonable time. This result may be good enough in practice, but it is unlikely to be optimal: only enumeration can give the absolute optimum solution. In this paper we develop smart enumeration: giving the effect of enumeration whilst reducing the cost.

We define *smart enumeration* as enumeration using heuristics to give same effect as exhaustive search with reduced effort. Smart enumeration gives the benefits of enumeration (we can be certain the solution is optimal) while mitigating its drawbacks (time and resources needed). The key is to choose problem domain carefully and apply heuristics to order and reduce the search space without throwing away any potential solutions.

As a case study for our smart enumeration approach, we address the problem of identifying minimal circuits for a function by improving the upper and lower bounds of resources it can use. We find the lower bound using global generation: in principle, generating every possible configuration of a device. In pratice, we use local symmetries to give the effect of exhaustive generation at reduced cost, using Field-Programmable Gate Array devices (FPGAs) for high-speed emulation of configurations and connections of look-up tables (LUTs). By searching the space of LUT configurations and interconnections directly, we combine logic minimisation and technology mapping from Boolean functions to LUTs. We find the upper bound using logic decomposition, applying local generation on the components of the decomposition. If we are lucky, global generation uncovers the minimum possible implementation. Otherwise, we get an improved measure of the bounds within which the optimal design must lie, as well as a locally optimized implementation. We also show how the search space can be reordered to optimise for low power.

Our main contributions in this paper are to:

- Identify key properties of smart enumeration and apply them to technology mapping (section 3)
- Improve the measure of the bounds for optimal solutions (section 3.1)
- Build a framework for circuit generation combining logic minimization and technology mapping (section 3.2)
- Use logic decomposition to guide search, pruning the search space and giving the upper bound (section 3.3)
- Reorder the search to optimise for low power (section 3.4)

In the rest of this section, we enumerate key properties of smart emnumeration and provide an overview of the paper. Several key properties of smart emnumeration can be identified as follows:

1. Restrict to narrow problem domain
2. Preprocess: eliminate hopeless cases
3. Order search space: order by cost metric, then by enumeration effort
4. Canonical forms: search classes of possible solutions, not the solutions
5. Symmetry: relate to other parts of search, save common results
6. Filtering: early elimination of some cases as preprocessing step
7. Early exit: abandon current leaf at earliest opportunity
8. Massive parallelism: use grid computing and custom hardware to exploit parallelism between unrelated searches

Each property is a different way of ordering or reducing the huge search space to be enumerated. We apply each property to the technology mapping problem.

The rest of this paper is structured as follows: Section 2 shows related work Section 3.1 shows our circuit generation framework combining logic minimization and technology mapping. Section 3.2 shows parallel hardware for generating circuits on FPGAs. Section 3.3 uses logic decomposition to guide and speed up the search, finding the upper bound. Section 3.4 shows how the search space can be reordered to optimise for low power. Section 4 gives results and evaluates the use of logic decomposition in our framework for logic minimization and technology mapping. Finally, Section 5 concludes and suggests future work.

2 Related work

Early works on area minimization decompose the circuit into a set of trees, and apply technology mapping on tree structures [1, 2]. Cong et al. concentrate on enumeration of single output, K-input connected subgraphs (fanout-free cones) within the circuit, and prove that the problem can still be optimally solved by decomposing the circuit into maximal fanout-free cones (MFFC), and enumerating separately on each MFFC [3]. The proposed algorithm restricts the solution to duplication-free mappings where each circuit gate must be mapped to exactly one LUT. Later work by Cong et al. [4] introduces heuristics to reduce the runtime, and extends the approach to duplicable mappings.

More recently, Ling et al [5] reformulated the technology mapping problem as a Boolean satisfiability (SAT) problem, showing that state-of-the-art FPGA technology mapping algorithms miss optimal solutions. They also created an algorithm solving the optimal area mapping problem. Safarpour et al. [8] decompose the resulting SAT problem into two easier problems to increase efficiency. Cong et al. [9] derive their SAT formulation from the implicant rather than the minterm representation of the problem, creating a smaller problem which can be solved faster and cover more target problems.

Two recent efforts using enumeration concern an implicit technique for enumerating structural choices in circuit optimization based on rewiring and resubstitution [6], and the adoption of reverse search in enumerative optimization for obtaining, for instance, the k shortest Euclidean spanning trees [7]. Our research complements this work, since we exploit circuit parallelism to speed up generation.

3 Case study: technology mapping

This section shows how we apply our definition of smart enumeration to the problem of technology mapping: mapping from device-independent primitives to device-specific primitives. We apply the properties as follows:

1. Restrict to single-output combinatorial circuits implemented as DAGs of LUTs, in form of truth tables
2. Preprocess: eliminate all disconnected graphs, feedback, redundant inputs, redundant LUTs
3. Order search space: order by optimisation metric (latency / area), then by enumeration effort. Start with potentially most optimal solution, then iterate from easiest to hardest
4. Canonical forms: enumerate NPN-equivalence classes rather than all the possible functions
5. Symmetry: one configuration with one input corresponds to many combinations of inputs and configurations
6. Filtering: for each graph, for each combination, filter configurations passing N inputs, where N < the total number of inputs
7. Early exit: enumerate each combination until first failing input
8. Massive parallelism: parallelise over configuration space, run hundreds of subsearches in parallel over grid. Map to custom hardware exploiting low-level FPGA resources (ROMs, multipliers) to factor out common and constant parts of designs. Generate specific hardware optimised for each search.

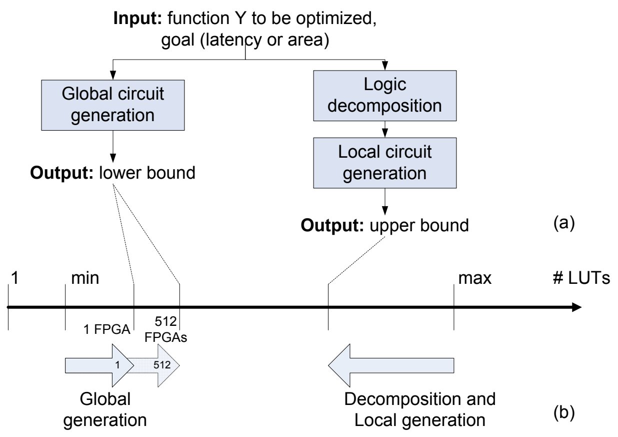

Fig. 1. Improving bounds by generation and decomposition: (a) Process input (logic function) and outputs (upper and lower bounds). (b) Starting with the initial maximum max and minimum min number of LUTs, global circuit generation increases the lower bound, while decomposition and local generation reduce the upper. Generation is parallelizable, so multiple FPGAs can be used for generation, allowing a higher lower bound by generating more circuits in a reasonable time. Ultimately we find either the absolute minimum circuit by global generation, or new, tighter bounds within which it must lie.

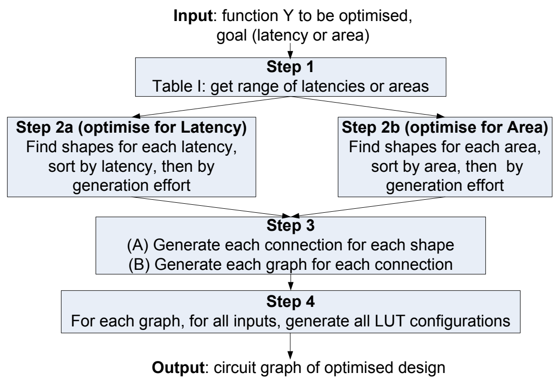
Input: function Y to be optimised, goal (latency or area)
Step 1
Table I: get range of latencies or areas
Step 2a (optimise for Latency)
Find shapes for each latency, sort by latency, then by generation effort
Step 2b (optimise for Area)
Find shapes for each area, sort by area, then by generation effort
Step 3
(A) Generate each connection for each shape
(B) Generate each graph for each connection
Step 4
For each graph, for all inputs, generate all LUT configurations
Output: circuit graph of optimised design

Fig. 2. Circuit generation. Step 2 differs for area (step 2a) and latency (step 2b). Step 4 can run in software or parallel hardware (section 3.2).

3.1 Approach

This section shows our circuit generation framework's four-step approach, which involves developing expressions for the upper and lower bound sizes of mappings from function to graph of LUTs. Our approach improves the initial lower and upper bounds of the number of LUTs required to implement a given logic function (fig. 1), using circuit generation and logic decomposition: global circuit generation, on the entire design, improves the minimum; local generation, on the parts of the decomposed design, improves the maximum.

We break generation into four steps (fig. 2). We implement step 4 in parallel hardware on FPGAs, relying on two key FPGA properties: (a) LUTs: high-speed table look-up. (b) Massive parallelism: many instances in parallel. For pratcitcality, we limit ourselves to single output functions.

Fig. 2 shows how we break the problem into four steps: **Step 1**: given an N-bit input, 1-bit output boolean input function Y and an optimization mode (area or latency), identify observable inputs and limit the search space. **Step 2**: generate all circuit *shapes* (vectors of the numbers of LUTs in each layer) within the search space from step 1; sort by (a) latency or (b) area. **Step 3**: generate all possible interconnections for each shape, **Step 4**: generate all possible LUT configurations for each circuit graph. We generate graphs of 4-input LUTs, with H layers of LUTs, where layer h has L_h LUTs; L_{tot} LUTs in total.

Logic functions with more than four inputs require multiple LUTs. We further refine the four steps (fig. 2) for N-input logic functions.

Step 1. Count observable inputs, index into table 1 to find the area or latency bounds. We define *latency* as the maximal depth in LUTs from design inputs to design output, and *area* as the total number of LUTs. We calculate the initial upper bound by observing that an $n+1$-input design can be implemented using two n-input LUTs multipexed by the $n+1$th input using one more LUT.

Three rules facilitate calculation of minumum area and latency required: (1) each observable design input must connect to at least one LUT input, (2) at least one of the LUT inputs must connect to a LUT output at a previous layer, (3) there is a single LUT at the highest layer. These rules ensure that (1) no input is redundant, (2) no LUT is disconnected (redundant) and (3) there is only one design output.

Step 2. Find all shapes for the bounds from step 1 (table 2). Sort the resulting list of shapes by latency (if optimizing for latency, step 2a) or area (if optimizing for area, step 2b). Within the sorted list, sort equal-area (step 2a) or equal-latency (step 2b) shapes by generation effort: order by size of search space in steps 3 and 4. For example, for an 7-input design for minimum area, first choose the smallest shape that will accept seven inputs: (1,1) in our terminology. If this fails, choose the next smallest shape: (2,1). Similarly, find the minimum latency design by iterating from the minimum latency topology to the maximum. We observe that the number of shapes for a given number of LUTs L_{tot} and layers H is bounded by the binomial coefficient $\binom{L_{tot}}{H}$. Thus the total number of shapes is bounded by $2^{L_{tot}}$ [10], and the total number of shapes for the bounds of areas from step 1 is bounded by: $\sum_{\lfloor (N+1)/3 \rfloor}^{2^{N-3}-1} 2^{L_{tot}} = 2^{2^{N-3}} - 2^{\lfloor (N+1)/3 \rfloor}$

Step 3. Generate all interconnections. Step 3(A): produce a set of *connections* for each shape: topologically distinct trees where the output of each LUT in a layer must connect to the input of a LUT in the next layer. Step 3(B): generate directed acyclic *graphs* for each connection: all combinations of connections from each LUT input unconnected in step 3(A) to each LUT output

Table 1. Step 1: Theoretical upper and lower bounds for latency (maximal depth of LUTs from inputs to output) and area (number of LUTs) for various numbers of inputs.

function	optimize for latency		optimize for area	
#inputs	min	max	min	max
≤ 4	1	1	1	1
5	2	2	2	3
N	$log_4(N)$	$(N-3)$	$\lfloor (N+1)/3 \rfloor$	$2^{N-3}-1$
	$O(logN)$	$O(N)$	$O(N)$	$O(2^N)$

Table 2. Step 2: All the different shapes for one to three 4-LUTs.

	Latency			
	1	2	3	4
Area 1	(1)			
2		(1,1)		
3		(2,1)	(1,1,1)	

in previous layers, and the design inputs. For a LUT at layer h, the number of possible interconnections is: $L_{h-1} * (N + \sum_0^{h-1} L_i)^3$.

Step 4. For all graphs, generate each configuration of each LUT, for each input. The output of the final circuit must match Y for each input over the input space of 2^N. We use parallel hardware to speed up this step (next section).

3.2 Generation Circuit Generation

This section shows our designs for implementing step 4 (fig. 2) by parallel generation on reconfigurable hardware.

We build FPGA circuits using ASC, A Stream Compiler [11]; this means we can write low-level optimizations and high-level structure all within the same C++ description. We build one ASC design per shape:

Step 4. Generate an ASC circuit for each graph output from step 3(B). Instantiate the target hardware, datapath containing LUT emulators and comparators, and a finite state machine to loop through each input until the first failing one, for each configuration, stopping at the first configuration that matches the target Y output for each input. We emulate LUTs, rather than use FPGA LUTs directly, to avoid reconfiguring the design for each set of LUT configurations.

Fig. 3 shows the datapath our parallel generation hardware, which we use for both depth-first and breadth-first approaches. The difference is in the state machine driving the datapath: depth-first tries each input until the first failing one; breadth-first tries only a small set of inputs. Here *failing* means the output of LUT0 does not match the the target (output of Y).

Mapping to Xilinx LUTs. Part of the above design can map explicitly to Xilinx Virtex II CLB resources – similar techniques can apply to other FPGA families. Our hardware design has two properties: (a) for p LUTs emulated in parallel, each parallel configuration for LUT 0 lies in the same arithmetic se-

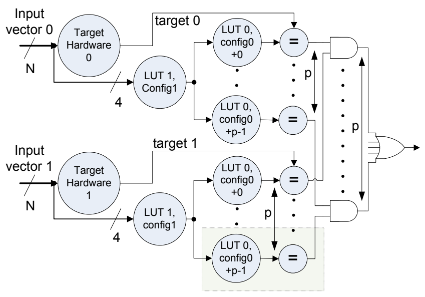

Fig. 3. Hardware for parallel generation of shape (1,1), with $I = 2$ input vectors in parallel. For each input vector, we replicate the target hardware and emulation of LUT 1, and LUT 0 for each of p different configurations. We use this design for both breadth- and depth-first generation. Dotted lines indicate hardware omitted for clarity.

quence $c..c + p$, (b) thus the $log_2(p)$ least-significant bits of each configuration are constant, and can be emulated with ROMs.

3.3 Logic Decomposition

This section shows how we use logic decomposition to improve the measure of the upper bound number of LUTs needed to implement the target design from the initial maximum.

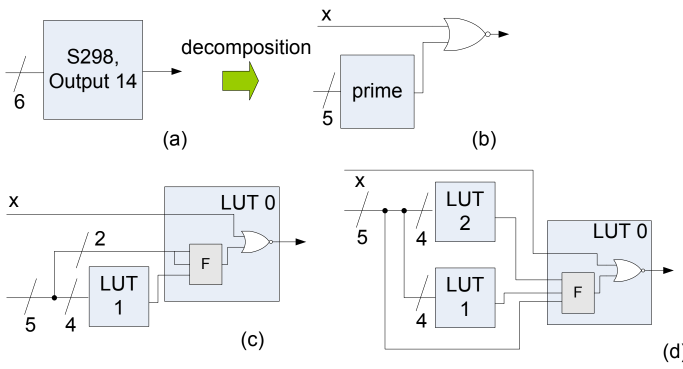

Fig. 4. Using logic decomposition: motivating example. One benchmark (a) is decomposed into a five-input function and a NOR gate with input labelled x (b). We show that only designs (c) and (d) need be considered, a considerably smaller search space than for a general six-input function, and for LUT 0, we need only generate three-input function F.

Logic decomposition breaks a circuit into a collection of subcircuits and their connections. To show the potential benefits of logic decomposition, consider a small example: output 14 of ISCAS benchmark s298. This has six observable inputs: too large to generate on a single CPU or FPGA. The total search space for a six-input function is of order $O(2^{128})$, using up to seven LUTs. Fig. 4 shows the results of decomposing this design (a) into (b): a two-input NOR gate and a five-input prime (non-decomposable) block. After decomposition, we can reduce the search space to (c) and (d) : a five-input function takes at most three LUTs (d), and this design can implement the NOR in LUT 0. Three LUTs is a significant search-space reduction compared to seven without decomposition. Furthermore, because part of the function of LUT 0 is now fixed, its search space reduces to a three-input function F (d) (search space size $2^{2^3} = 2^8$, compared to 2^{16} for a 4-input function). The total search-space reduction is thus $2^{16}/2^8 = 256$. Also, the bounds improve from 2..3 (latency) and 2..7 (area) to 2..2 (latency) and 2..3 (area). Logic decomposition improves the upper bound, generating each subsearch separately. Although the overall result is no longer optimal, each generated subcircuit remains optimal.

Table 3. Range improvement. Shows number of observable inputs, minimal shape found and results from Xilinx XSTv8.1 (X) and for DAOmap (D) and FlowMap (F), using the RASP package from UCLA [13].

Name	Output	#Obs.	#Shapes	Shape	#LUTs			Area bounds		Latency bounds	
		Inputs			X	D	F	(old)	(imp.)	(old)	(imp.)
s27	1	5	2	(1,1)	2	5	5	2..3	2..2	2..2	2..2
	2	5	2	(1,1)	2	2	2	2..3	2..2	2..2	2..2
	3	5	2	(1,1)	2	5	5	2..3	2..2	2..2	2..2
s298	8	5	2	(1,1)	2	3	3	2..3	2..2	2..2	2..2
	10	5	2	(1,1)	2	3	3	2..3	2..2	2..2	2..2
	12	7	268	(2,1)	3	5	5	2..15	3..3	2..2	3..3
	14	6	18	(2,1)	3	3	3	2..7	3..3	2..2	3..3
b01	4	5	2	(1,1)	2	3	3	2..3	2..2	2..2	2..2
	5	5	2	(1,1)	2	3	3	2..3	2..2	2..2	2..2
	6	5	2	(1,1)	2	3	3	2..3	2..2	2..2	2..2
	7	5	2	(1,1)	2	2	3	2..3	2..2	2..2	2..2

Because circuit generation takes time exponential in the number of design inputs, we choose a disjoint decomposition method (decomposed functions have no common inputs); specifically, we use Plaza and Bertacco's STACCATO method and software [12]. Staccato decomposes a logic function into a tree of subfunctions, each with disjoint inputs. Each subfunction is either associative (AND, OR, XOR), or a *prime* function – one that cannot be decomposed further.

Our approach divides into four steps: (1) apply logic decomposition to design, (2) traverse the decomposition tree, separating out the prime (non-decomposable) blocks, (3) generate each prime block, (4) build the output hardware from the generated blocks. Step 3 applies the generation techniques designed in the rest of this paper, using the global optimization goal (latency or area). Note that the decomposed prime blocks may still have too many inputs to practically generate; these cases must rely on conventional tools for optimization.

3.4 Optimising for low power

The proposed search for low area and low latency is also useful for low power consumption, since power is proportional to area and depth of logic. We can further adapt our search for low power consumption as follows: first, we can reorder graph generation so graphs with long wires skipping one or more LUTs are considered after designs without such wires. Long wires tend to use more power in routing resources than neighbour wires between LUTs. Second, designs with high fan-in or fan-out tend to use more power than designs with low fan-in/out and can be ordered after them. Third, some LUT configurations may use more power than others – this requires careful modelling and characterisation of LUTs in the target technology. Finally, power estimation software can be included in the enumeration process, with early exit for designs estimated to use more power (reordered to be searched after lower-power designs).

4 Results and evaluation

This section shows results for software and hardware generation for several ISCAS benchmarks, showing original and improved bounds achieved. We also show estimated power use for replicated units of the circuit graphs found.

Table 3 shows benchmarks chosen from the standard ISCAS 85, 89 and 99 sets, with bounds of LUTs for area and latencies – these worst-case results are the initial upper and lower bounds from table 1. The XST, DAOmap [15] and FlowMap [14] results are for each output individually – we remove hardware for other outputs. The bounds improvement results for these benchmarks show runtime and minimal shapes found (software results run on an Intel Xeon 2Ghz processor). Generation times vary up to an order of magnitude. All the software generation results correspond to an generation rate of roughly 4.8×10^6 configurations per second, about 20% the rate of our hardware. Hardware generation runs on a single Xilinx XC2V6000 FPGA (Celoxica RC2000 board).

Table 4 shows power consumption of designs similar to those found. Since enumeration necessarily considers small designs, we replicate the designs many times to show the general trend. It can be seen that there is a potential power saving of 11% to 22% when a 3-LUT design is optimised to one with 2 LUTs.

5 Conclusion

We define smart enumeration and show how it can apply to technology mapping. We introduce a methodology for optimising circuits for FPGA implementation that combines logic minimization and technology mapping. We develop a four-step process to give the effect of generating all possible circuits ordered by user optimization goal: latency or area. Our reconfigurable hardware implementation speeds up this process by rapidly finding which generated circuits match the target design. We use logic decomposition to guide and speed up our search process, eliminating searches using the resulting decomposition tree. Although

Numbers of inputs	LUTs	FFs	instances				
			500	1000	2000	4000	8000
5	3	6	614	879	1479	2653	4992
5	2	6	543	738	1195	2085	3856
6	2	7	552	755	1238	2180	4054
7	2	8	560	773	1281	2274	4252

Table 4. Estimated power consumption in milliwatts for replicated units of the designs enumerated for two and three LUTs. The results are obtained from Xilinx Virtex-II Web Power Tool Version 8.1.01.

our approach is only globally optimal for small designs, it is still locally optimal for large designs, and can be applied to building blocks of large designs. We show that if the design can fit into fewer LUTs, the power consumption can be reduced substantially.

Current and future work includes porting generation to the *Cube*, a large multiple-FPGA machine developed to exploit massive parallelism. This is ideal for circuit generation as many generators can run in parallel across multiple FPGAs. We would also like to extend generation to cover multiple-output designs, sequential designs and other design elements beyond LUTs. Our ultimate goal is to subsume many traditionally separate optimization steps into one generation step, with results guaranteed to be optimal.

Acknowledgements: The support of UK Engineering and Physical Sciences Research Council (Grant number EP/C509625/1 and EP/C549481/1), European FP6 Project hArtes, Agility, Celoxica and Xilinx is gratefully acknowledged.

References

1. K. Keutzer. “DAGON: Technology Binding and Local Optimization by DAG Matching”. In *Proc. DAC*, pages 341–347, 1987
2. R. Francis, J. Rose, and Z. Vranesic. “Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs”. In *Proc. DAC*, pages 227–233, 1991.
3. J. Cong, and Y. Ding. “On area/depth trade-off in LUT-based FPGA technology mapping”. In *Proc. DAC*, pages 213–218, 1993.
4. J. Cong, C. Wu, and Y. Ding. “Cut ranking and pruning: enabling a general and efficient FPGA mapping solution”. In *Proc. FPGA*, pages 29–35, 1999.
5. A. Ling, D.P. Singh and S.P. Brown, “FPGA Technology Mapping: A Study of Optimality”, In *Proc. DAC*, IEEE, 2005.
6. V.N. Kravets and P. Kudva, “Implicit Enumeration of Structural Changes in Circuit Optimization”, *Proc. DAC*, pp. 438–441, 2004.
7. J. Nievergelt, “Exhaustive Search, Combinatorial Optimization and Enumeration: Exploring the Potential of Raw Computing Power”, *Proc. Conf. on Current Trends in Theory and Practice of Informatics*, LNCS 1963, pp. 18–35, 2000.
8. S. Safarpour, A. Veneris, G. Baeckler and R. Yuan, “Efficient SAT-based Boolean Matching for FPGA Technology Mapping”, in *Proc. DAC*, IEEE, 2006.

9. J. Cong and K. Minkovich, "Improved SAT-based Boolean Matching Using Implicants for LUT-based FPGAs", in *Proc. FPGA*, ACM, 2007.
10. Ronald L Graham, Donald E Knuth, Oren Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, Addison-Wesley, 1989.
11. Oskar Mencer, "ASC, A Stream Compiler for Computing with FPGAs" *IEEE Trans. on CAD*, IEEE, 2006.
12. S. Plaza and V. Bertacco. "STACCATO: Disjoint Support Decompositions from BDDs through Symbolic Kernels". In *Proc. Asia South Pacific Design Conference*, 2005.
13. UCLA VLSI CAD lab., RASP – LUT-Based FPGA Technology Mapping Package, release B1.1, at `http://cadlab.cs.ucla.edu/software_release/rasp/htdocs/` .
14. J. Cong and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-table Based FPGA Designs", in *IEEE Trans. on CAD of ICs and Systems* 13:1, IEEE, Jan. 1994.
15. D. Chen, and J. Cong,"DAOmap : A Depth-Optimal Area Optimization Mapping Algorithm for FPGA Designs", in *Proc. ICCAD*, pp. 752-759, Nov. 2004.