

Heterogeneous Technology Mapping for Area Reduction in FPGA's with Embedded Memory Arrays

Steven J. E. Wilton

Abstract—It has become clear that large embedded configurable memory arrays will be essential in future field programmable gate arrays (FPGA's). Embedded arrays provide high-density high-speed implementations of the storage parts of circuits. Unfortunately, they require the FPGA vendor to partition the device into memory and logic resources at manufacture-time. This leads to a waste of chip area for customers that do not use all of the storage provided. This chip area need not be wasted, and can in fact be used very efficiently, if the arrays are configured as multioutput ROM's, and used to implement logic.

In this paper, we describe two versions of a new technology mapping algorithm that identifies parts of circuits that can be efficiently mapped to an embedded array and performs this mapping. The first version of the algorithm places no constraints on the depth of the final circuit; on a set of 29 sequential and combinational benchmarks, the tool is able to map, on average, 59.7 4-LUT's into a single 2-Kbit memory array, while increasing the critical path by 7%. The second version of the algorithm places a constraint on the depth of the final circuit; it maps, on average, 56.7 4-LUT's into the same memory array, while increasing the critical path by only 2.3%.

This paper also considers the effect of the memory array architecture on the ability of the algorithm to pack logic into memory. It is shown that the algorithm performs best when each array has between 512 and 2048 bits, and has a word width that can be configured as 1, 2, 4, or 8.

Index Terms—Embedded memories, FPGA, technology mapping.

I. INTRODUCTION

IT has become clear that on-chip storage is critical in large field programmable gate arrays (FPGA's). As FPGA's grow, they are being used to implement entire systems, rather than the small logic subcircuits that have traditionally been targeted to FPGA's. One of the key differences between these large systems and smaller logic subcircuits is that the large systems often require storage. Although this storage could be implemented off-chip, on-chip storage has a number of advantages. Besides the obvious advantages of integration, on-chip storage will often lead to higher clock frequencies, since input-output (I/O) pins need not be driven with each memory access. In

addition, on-chip storage will relax I/O pin requirements, since pins need not be devoted to external memory connections.

Two implementations of on-chip storage in FPGA's have emerged: fine-grained and coarse-grained. In FPGA's employing fine-grained memory, such as the Xilinx 4000 and DynaChip FPGA's, each lookup-table can be configured as a small RAM, and these RAM's can be combined to implement larger user memories [1], [2]. The coarse-grained approach is used in the Altera FLEX 10K, FLEX 10KE and APEX 20K devices [3]–[5], the Xilinx Virtex FPGA's [6], the Cypress Delta 39K devices, the Vantis VF1 FPGA's [7], and the Actel ProASIC, 42MX, and 3200DX families [8]–[10]. In these devices, large arrays are embedded onto the FPGA. Devices in the Altera FLEX 10K family contain 2-Kbit arrays, Altera FLEX 10KE and Xilinx Virtex devices contain 4-Kbit arrays, Actel's ProASIC parts contain 2304-bit arrays, Actel's 42MX and 3200DX devices contain 256-bit arrays, and the Cypress Delta 39K contains both four-Kbit arrays and eight-Kbit arrays. Several academic studies have also focused on coarse-grained memory architectures [11]–[13].

The coarse-grained approach results in significantly denser memory implementations, since the per-bit overhead is much smaller [14]. Unfortunately, it also requires the FPGA vendor to partition the chip into memory and logic regions when the FPGA is designed. Since circuits have widely-varying memory requirements, this "average-case" partitioning may result in poor device utilizations for logic-intensive or memory-intensive circuits. In particular, if a circuit does not use all the available memory arrays to implement storage (or in the worst case, uses none at all), the chip area devoted to the unused arrays is wasted.

This chip area need not be wasted, however, if the unused memory arrays are used to implement logic. Configuring the arrays as ROM's results in large multioutput lookup-tables that might very efficiently implement some logic circuits. In many combinational subcircuits, a small set of inputs are combined to produce a large number of intermediate results; these intermediate results are then often combined to produce only a few outputs [15]. If the available memory arrays are large enough to implement the output signals directly as functions of the input signals, then the intermediate nodes are not needed. This could result in significant area savings. Similarly, if several combinational levels can be packed into a single memory array, significant speed improvements may be obtained. Of course, rarely will an entire circuit fit into the available memory arrays. An algorithm that identifies the parts of circuits that can be effi-

Manuscript received June 15, 1998; revised August 17, 1999. This work was supported by Cadence Design Systems, the British Columbia Advanced Systems Institute, and the Natural Science and Engineering Research Council of Canada. This paper was recommended by Associate Editor L. Stok.

The author is with the Department of Electrical and Computer Engineering at the University of British Columbia, Vancouver, BC, V6T 1Z4, Canada.

Publisher Item Identifier S 0278-0070(00)01369-5.

ciently mapped to the available arrays, and implements the rest using normal look-up tables (LUT's) is key to using the available memory arrays effectively. Such an algorithm is the focus of this paper.

Many technology mapping algorithms which target LUT's have been developed [16], [17]. There are three major differences between the problem of lookup-table technology mapping and our problem. First, unlike the LUT's available on most of today's FPGA's, memory arrays have multiple outputs (in the Altera APEX20K, each block can have up to 16 outputs). Second, the arrays available on most FPGA's can be used in one of several aspect ratios (width/depth combinations). Finally, there is usually only a small number of arrays available; the goal is to use the limited number of arrays as effectively as possible. Thus, algorithms developed for lookup-table technology mapping are not suitable to solve our problem.

Implementing logic in memory arrays has been studied by Murgai [18]. Murgai's application was a circuit emulation system, however, in which it was permissible to take more than one clock cycle to evaluate complex functions. Heterogeneous technology mapping was studied by He and Rose [19]. Their algorithm targets FPGA's with two sizes of single-output lookup-tables, and is not immediately extendible to lookup-tables with multiple outputs. Cong and Xu focused on delay-optimal heterogeneous technology mapping, but again their algorithms assume single-output arrays [20], [21]. Technology mapping to multiple-output lookup-tables has been studied [22], [23]. Most such algorithms are targeted to lookup-tables with two-outputs; the matching techniques used in these algorithms are not suitable for arrays with eight or more outputs. Finally, this problem is similar to that of mapping logic to programmable logic arrays (PLA's) [24], [25]. Unlike PLA's, however, memories can implement any function of their inputs, without regard for the number of product terms.

Two algorithms aimed at solving the heterogeneous technology mapping problem for arrays with multiple outputs were described at the 1998 International Symposium on Field-Programmable Gate Arrays [26], [27]. In this paper, we review and extend the discussion from [26]. We also present a new version of the algorithm which minimizes area without increasing the depth of the final circuit, and compare the performance of that algorithm with the results presented in [27]. Finally, this paper also investigates the performance of the algorithm across a wide variety of FPGA memory architectures. An early version of this latter material appears in [28].

II. TERMINOLOGY AND PROBLEM DEFINITION

In this paper, we will use the following terminology (primarily from [29]). The combinational part of a circuit is represented by a directed acyclic graph $G(V, E)$ where the vertices V represent combinational nodes, and the edges E represent dependencies between the nodes. V also contains nodes representing each primary input and output of the circuit. Flip-flop inputs and outputs are treated as primary outputs and inputs. The depth of a node v , d_v , is the maximum path length from any primary input to node v . The depth of a graph d_G is the maximum

d_v for all nodes v in G . A network is k -feasible if the number of inputs to each node is no more than k . Given a node v , a cone rooted at v is a subnetwork containing v and some of its predecessors. We extend this and define a cone rooted at a set of nodes W to be a subnetwork containing each node in W along with nodes that are predecessors of at least one node in W . A *fanout-free cone* is a cone in which no node in the cone (except the root) drives a node not in the cone. The *maximum-fanout free cone (MFFC)* for a node v (or set of nodes W) is the fanout-free cone rooted at v (or W) containing the largest number of nodes [30], [31]. Given a cone C rooted at v , a *cut* (X, X') is a partitioning of nodes such that $X' = C$. A *cut-set* of a cut is the set of all nodes v such that $v \in X$ and v drives a node in X' . If the size of the cut set is no more than d , the cut is said to be d -feasible. Given a cone C rooted at v , the *maximum-volume d -feasible cut* is the d -feasible cut (X, X') with the largest number of nodes in X' .

We assume an FPGA consisting of many k -input lookup-tables and N fixed-size memory arrays. Each memory array is configurable, allowing the user to select one of several word widths for that array (since the total number of bits in each array B is fixed, a wider word means there are fewer words in the array). The set of word widths that each array can take on is denoted by w_{eff} . The read access time of the memory is assumed to be fixed; the ratio of the read access time to the propagation time of a lookup-table is denoted by h_m . These parameters are summarized in Table I, along with values for commercial FPGA's. In addition, the table shows the value or range of values for each parameter for which we present experimental results. Note that the Cypress and Actel parts do not use lookup-tables as their basic logic element. We do not quote values for h_m for the commercial devices; this ratio depends on the specific path taken into and out of the memory or logic block. In the experiments, we assume $h_m = 3$, which was estimated using access time models (and is also used in [27]).

We present two algorithms, each of which maps logic circuits to an FPGA defined by the parameters in Table I. The goal of the first algorithm is to minimize the area required to implement the logic circuit without regard for circuit delay. The algorithm packs logic into the N memory arrays, and implements the rest using lookup-tables. Since N is fixed, the area is minimized when the number of lookup-tables required to implement the parts of the circuit that can not be mapped to the memories is minimized. This can be written as follows.

1) *Area-Minimization Problem:* Given a circuit C and an FPGA defined by the parameters in Table I, find an implementation of C (denoted C') using up to n memory arrays and m k -input LUT's (k -LUT's) such that $n \leq N$ and m is minimum.

The goal of the second algorithm is to minimize the area required to implement the logic circuit without increasing the critical path of the final circuit. Since, before place and route, it is difficult to predict which paths will be slow, we solve an approximation to this problem:

2) *Area-Minimization with Depth Constraint Problem:* Given a circuit C and an FPGA defined by the parameters in Table I, find an implementation of C (denoted C') with combinational depth $d_{C'}$ using up to n memory

TABLE I
ARCHITECTURAL PARAMETERS

Parameter	Meaning	Commercial Architectures									Range in this paper
		Altera 10K	Altera 10KE	Altera 20K	Xilinx Virtex	Cypress Delta39K	Vantis VFI	Actel ProASIC	Actel 42MX	Actel 3200DX	
N	Number of Arrays	3-20	6-24	16-228	8-32	6-16	28-48	6-60	10	8-16	1-16
B	Bits per Array	2048	4096	2048	4096	4096, 8192	128	2304	256	256	256-16384
w_{eff}	Data Widths	{1,2,4,8}	{2,4,8,16}	{1,2,4,8,16}	{1,2,4,8,16}	{1,2,4,8}	{4}	{9}	{4,8}	{4,8}	several
h_m	Ratio of memory read time to LUT prop. Delay	See text									3
k	Inputs per LUT	4	4	4	4	-	3	-	-	-	4

arrays and m k -input LUT's (k -LUT's) such that $n \leq N$, $d_C \leq d_{C_0}$ and m is minimum, where d_{C_0} is the combinational depth of the circuit implemented using only LUT's.

In other words, the goal is to find an implementation of C with a maximum combinational depth that is *no larger* than the optimum combinational depth of C implemented using only lookup-tables (this latter quantity can be computed using Flowmap [17]). In calculating the combinational depths, the delay of each lookup table is taken as one, while the delay of each memory block is taken as h_m . In Section III, we will focus on a restricted version of these problems, in which $N = 1$. In Section V, we extend the algorithms for $N > 1$.

III. SOLUTION FOR ONE MEMORY ARRAY

A. Overall Approach

There are at least three possible approaches to solving the problems specified in the previous section. One possibility is to pack as much logic as possible into each memory array, and subsequently implement the rest of the logic using LUT's (using an algorithm such as Flowmap [17]). A second possibility is to map the logic to LUT's using Flowmap first, and then pack as many of these pre-packed LUT's into the available memory arrays as possible. A third possibility is to develop an algorithm that packs to both types of blocks simultaneously.

We have chosen the second approach listed above. The advantage of the second approach over the first approach is that by giving the packer a circuit that has already been technology mapped, the packer can evaluate precisely how various packing decisions will affect the number of LUT's in the final implementation. In the first approach, since the logic has not yet been mapped to LUT's, the packer can only guess at which packing decision is best, since it does not know how the remaining logic will be divided into lookup-tables.

The third approach, simultaneous mapping, leads to a very difficult optimization problem. Previous work has considered only single-output blocks and a fixed ratio of large blocks to small blocks [19]–[21]. The extension of such an algorithm to

map logic to large, scarce, multiple-output blocks considered is still an open problem.

B. Area Minimization Algorithm: SMAP

As described in the previous subsection, we first map the entire circuit to k -feasible nodes using an existing technology mapper (we use Flowmap/Flowpack [17]). We then pack as many of the k -feasible nodes as possible into the available memory arrays. Those nodes that could not be packed into memory arrays are implemented using k -LUT's.

There are three steps to the packing algorithm: (1) one node is chosen as a *seed node*, (2) the signals that will drive the memory array inputs are chosen, and (3) the signals that will be produced by the memory array outputs are chosen. The goal is to choose the memory array inputs and outputs such that the number of nodes that can be replaced by the memory array is as large as possible. Sections III-B.1 and III-B.2 will describe the selection of the memory input and output signals; Section III-B.3 will discuss the selection of the seed node. The discussion in these sections will assume an array with d inputs (address lines) and w outputs (data lines). In Section III-B.4, we will show how the algorithm can be extended to support arrays with a configurable word width.

1) *Memory Input Signals*: Given a seed node, the signals that will drive the memory array inputs are chosen by finding the maximum-volume d -feasible cut of the seed node's fanin network, where d is the number of inputs of the memory array. The signals that make up the cut become the memory array inputs. An algorithm to find such a cut was presented in [17]; we use the same algorithm here.

2) *Memory Output Signals*: If there is only one memory output ($w = 1$), we can select the seed node output as the memory array output. Fig. 1 shows an example in which $d = 8$ and $w = 1$. In this case, all nodes below the cut can be replaced with the memory array. In this case, our algorithm is the same as the Flowpack algorithm presented in [17].

For the large memory sizes considered in this paper, this simple extension of Flowpack does not work well for some

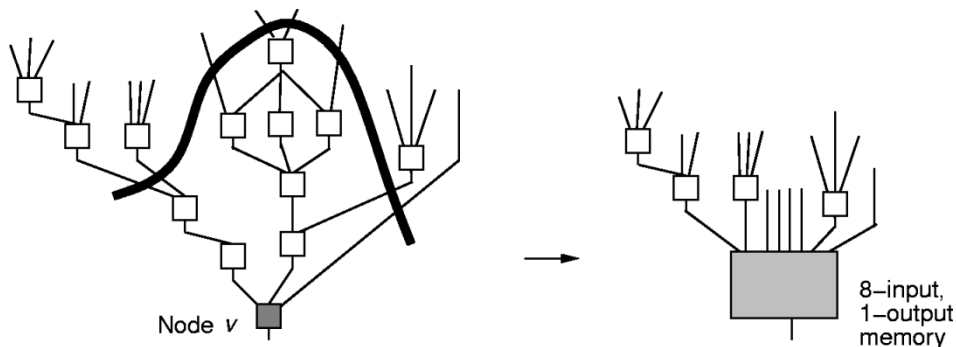


Fig. 1. Mapping a circuit to an eight-input, one-output memory block. Inputs are at the top and outputs are at the bottom.

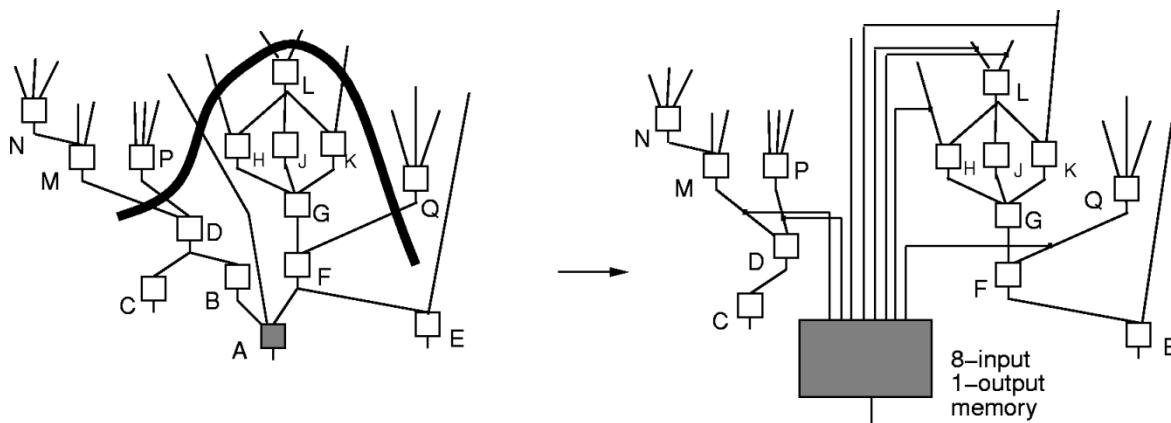


Fig. 2. Mapping a circuit to an eight-input, one-output memory block: Poor solution. Inputs are at the top, and outputs are at the bottom.

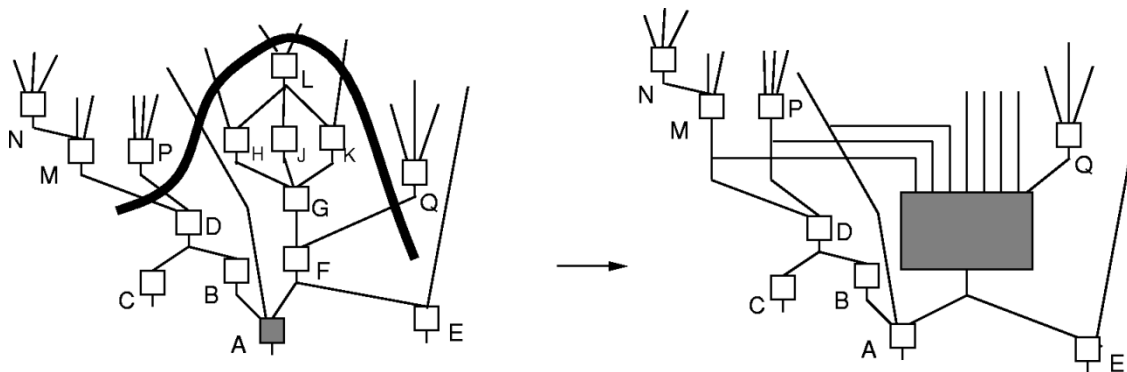


Fig. 3. Mapping a circuit to an eight-input, one-output memory block: Better solution. Inputs are at the top, outputs are at the bottom.

circuits. It is also not clear how to extend the algorithm to support multioutput blocks. By intelligently selecting which nodes will become the output of the memory array, we not only get better results for large single-output blocks, but also handle multioutput blocks.

In Fig. 1, all predecessors of the seed node that are below the cut can be packed into the memory array, and the LUT's implementing these nodes can be deleted. In general, this is not true. Define F to be the set of nodes in the fan-in network of the seed node v , and let F' be the maximum fanout-free cone (MFFC) rooted at v (clearly $F' \subseteq F$). Represent the cut of F found by the Flowpack algorithm as (X, X') where $v \in X'$. Then, only those nodes in the intersection of F' and X' can be deleted. In other words, of all the nodes below the cut, only those in the maximum fanout-free cone of v can be deleted.

In Fig. 1, all nodes below the cut were also in the MFFC of v . Fig. 2 shows an example where this is not true. In this circuit, the maximum fanout-free cone of v consists only of nodes A and B. Thus, only these two nodes can be deleted when the memory array is used, as shown in the right side of the figure. The signals generated by nodes D and F (and all their predecessors) are needed to drive nodes C and E respectively. Node E can not be packed into the memory array since it also depends on an input signal that is not part of the cut set. Node C can not be packed in the memory array, since it would require another memory array output.

Fig. 3 shows a better solution. Although the seed node, the cut, and the memory array inputs, are the same, we have now chosen the signal produced by node F as the output of the memory array. Node F and all its predecessors make up

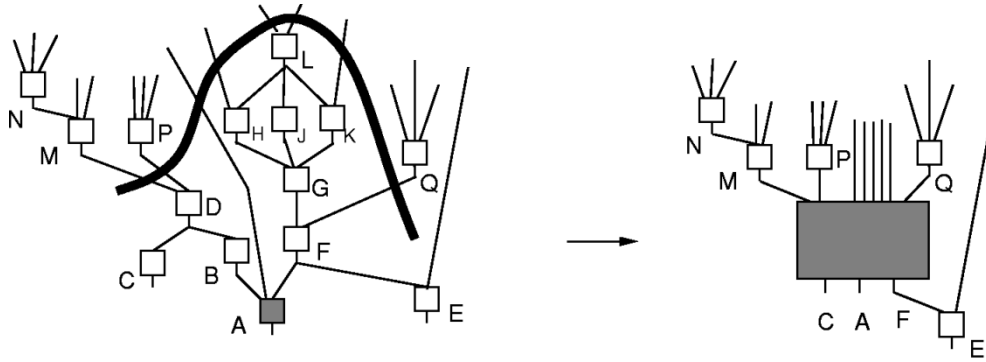


Fig. 4. Mapping to an eight-input, three-output memory block. Inputs are at the top, outputs are at the bottom.

```

find_this_solution(seed, Nin, Nout) {
    cut_set = max-vol Nin-feasible cut
    P = set of Potential Nodes driven by cut_set
    output_nodes = output_selection(P, Nout)
    retval.memory_inputs = cut_set
    retval.memory_outputs = output_nodes
    retval.nodes_to_delete = MFFC of all output_nodes
    return retval
}

output_selection(P, w) {
    for each node v in P {
        score[v] = number of nodes in MFFC of v
    }
    return w highest scoring nodes
}

```

Fig. 5. Summary of algorithm given seed node and array configuration.

the output's MFFC, and can be deleted and replaced by the memory array.

If the memory array has three outputs instead of one, the solution in Fig. 4 is possible. Here, the memory array produces the signals that had been produced by nodes C, A, and F. We can now delete all nodes in the MFFC of the three output signals.

In the examples of Figs. 3 and 4, we have intelligently chosen which signal(s) in the original network will be implemented as outputs of the memory array. We call this process *output selection*. Consider a circuit in which we have selected a seed node v , and found the cut (X, X') where $v \in X'$. The cut-nodes are the memory array inputs. All the nodes in X' are potential memory array outputs; the goal of the output selection algorithm is to select up to w of these nodes, such as to maximize the number of nodes in the MFFC of the selected nodes.

Rather than choosing the w nodes from X' , we can expand the solution space somewhat by choosing the nodes from P , where P is the set of nodes for which there is no path connecting the node to any primary input without traversing one of the cut edges. In other words, P is the set of all nodes that can be expressed as a function of only the cut signals. We call the nodes in P the *Potential Nodes*. Clearly, $X' \subseteq P$. As an example, had node C and A in Fig. 4 driven another node u , then u would be in P , even though it is not part of X' . Node u would be a legal choice for one of the w memory array outputs.

An exhaustive algorithm that checks all combinations of w nodes in P to find the best set of output signals would require $|P|!/w!(|P| - w)!$ checks; this is infeasible for even moderate w and $|P|$. Thus, we have developed a heuristic algorithm to perform output selection. In our algorithm, we visit all nodes in P , and assign a score equal to the number of nodes in that

node's MFFC. We then choose the w highest-scoring nodes as the memory array outputs. Note that this does not guarantee the optimum solution, since the MFFC rooted at the set of output nodes may be larger than the union of the individual MFFC's. In Section IV, however, we will show that this heuristic works well.

Note that in choosing the output nodes in this way, it may occasionally turn out that some of the original cut nodes are no longer needed. For example, in Fig. 3, the left three inputs are not needed to produce the function at node F. This could cause false-paths during timing analysis later in the design flow. To prevent this, we prune all inputs that are not actually needed to produce the output signals. Pruning also reduces the number of nets that must be routed. A more aggressive algorithm might try to make use of these unused inputs (perhaps by removing one of the chosen output nodes and packing in unrelated logic), however experimentation has shown us that very little can be gained by doing this, and it adds significant complexity to the algorithm.

Fig. 5 summarizes this part of the algorithm. The function `find_this_solution` finds a solution given a seed node and the number of memory array inputs and outputs. The solution returned consists of three parts: the nodes that make up the memory array inputs (the cut-nodes), the nodes that are replaced by memory array outputs (the nodes returned by the output selection algorithm) and the nodes that can be deleted.

3) *Seed Selection*: The selection of the seed node is critical. In order to ensure we are choosing intelligently, we apply the above algorithm once with each node in the circuit as the seed node. The seed node that leads to the maximum number of deleted nodes is then chosen.

```

SMAP{
  for each node  $v$  {
     $sol = \text{find\_this\_solution}(v, \text{maximum allowable array inputs, minimum allowable array outputs})$ 
    if  $|sol.nodes\_to\_delete|$  is largest so far,  $u = v$ 
  }
  for each array configuration  $i$  {
     $sol = \text{find\_this\_solution}(u, \text{number inputs for config } i, \text{number outputs for config } i)$ 
    if  $|sol.nodes\_to\_delete|$  is the largest so far,  $result = sol$ 
  }
  connect array outputs to nodes driven by  $result.memory\_outputs$ 
  connect array inputs to  $result.memory\_inputs$ 
  delete nodes in  $result.nodes\_to\_delete$ 
}

```

Fig. 6. Summary of SMAP algorithm for one memory array.

4) *Arrays with a Configurable Width:* Many FPGA's allow the user to configure the width of each memory array, trading width for depth. For example, in the Altera FLEX10K, each array can be configured as a 2048×1 , a 1024×2 , a 512×4 , or a 256×8 memory. Thus, each array can be used as either an eleven-input, one-output; a ten-input, two-output; a nine-input, four-output; or an eight-input, eight-output block. The above algorithm, however, requires us to know the number of inputs so that an appropriate cut-set can be found.

Our solution is to select the seed node using the narrowest array configuration, and then repeat the inner loop for all possible array configurations using the selected seed node. The configuration that gives the best results is then chosen. This is summarized in Fig. 6.

C. Area Minimization with Depth Constraint: SMAP-dc

In this section, we describe a variation of the SMAP algorithm which solves the *Area Minimization with Depth Constraint* problem described in Section II. The new algorithm minimizes the area of the circuit under the constraint that the depth of the circuit is no longer than it would be if the circuit was implemented using only LUT's. Since the input of the algorithm is a depth-optimal lookup-table implementation of the circuit (it is generated using Flowmap), it is enough to ensure that the depth of the final implementation is no larger than the depth of the input circuit. In other words, the depth does not increase as mapping progresses. This algorithm will be referred to as SMAP-dc.

The first step in SMAP-dc is to compute the *slack* at each node [32]. The slack for node v , denoted $slack(v)$, is the maximum amount of delay that can be added to the output of node v without increasing the delay of the entire circuit. The slack for all nodes in the input circuit can be computed efficiently using two breadth-first traversals of the graph: one forward from the input pins and one back from the output pins [32].

Once the slacks have been computed, the algorithm proceeds the same as SMAP. The only difference is in the output selection algorithm (the right side of Fig. 5). In SMAP, each node in P is

a potential output. In SMAP-dc, however, only the nodes in P that satisfy the following inequality are potential outputs:

$$d_v + slack(v) \geq h_m + \max_{x \in I} (d_x)$$

where I is the set of nodes in the cut set (the memory array inputs). If the memory array has w outputs, SMAP-dc finds the size of the MFFC rooted at each node in P that satisfies the above constraint, and chooses the w nodes with the largest MFFC. These nodes become the memory array outputs.

It can easily be shown that this guarantees that the combinational depth of the circuit does not increase. One would expect, however, that the amount of logic that is packed in each array is reduced somewhat, since there are fewer choices for each output node. The next section investigates this experimentally.

D. Complexity and Implementation Efficiency

There are two main parts of the algorithm: the input signal selection and the output signal selection. The input signal selection uses the algorithm from [17], which has a complexity of $O(d^3 m)$ where d is the number of inputs to the memory array and m is the number of edges below the cut. The output signal selection involves finding up to n MFFC's, each of which can be done in $O(m)$ time. This is repeated for each seed, leading to a complexity of $O(mn + d^3 m)$. Although this complexity is high, it is rarely seen in practice. In the next section, we present experimental results along with sample run times for several large circuits; even for one of our largest circuits (which contains 6211 lookup-tables), the run time for mapping to a single array was roughly two minutes, which compares favorably to the times required for lookup-table technology mapping as well as place and route.

In our implementation of the algorithm, we included two optimizations that, although they do not reduce the worst case complexity, speed up the algorithm significantly in practice. First, we save all cuts in a hash table as they are found. If, after finding a cut, it is discovered that this cut is in the hash table already, it is immediately concluded that this cut could produce a result no better than the previous best result, and the processing for that seed ends. Second, after finding the set of potential nodes P , if $|P|$ is less than or equal to the number of nodes to delete in the best

TABLE II
RESULTS ASSUMING 2048-BIT MEMORY ARRAYS

Circuit Name	Original Circuit		Nodes deleted: SMAP			Nodes Deleted: SMAP-dc		
	Number of 4-LUTS	Number of Flip Flops	1 array	16 arrays		1 array	16 arrays	
				BF=1	BF=4		BF=1	BF=4
pair	641	0	13	147	84	13	115	55
apex1	696	0	14	158	108	13	100	70
cps	749	0	46	249	228	27	82	76
C5315	596	0	12	139	74	12	126	62
C6288	527	0	19	134	82	19	158	85
apex3	867	0	26	196	169	23	147	100
C7552	679	0	15	163	83	15	119	62
i10	994	0	18	168	123	17	164	115
ex5p	1064	0	198	1056	1050	198	1037	1064
spla	3690	0	67	569	396	67	533	341
pdcc	4575	0	88	816	601	88	835	605
apex4	1262	0	319	1261	1261	319	1261	1261
tseng	1046	385	10	123	67	10	113	60
bigkey	1707	224	18	131	54	0	0	0
s38417	6096	1463	26	346	160	26	339	154
s298	1930	8	434	1930	1930	434	1930	1929
diffeq	1494	377	22	205	87	15	184	96
frisc	3539	886	62	176	121	57	158	103
dsip	1370	224	18	108	47	0	0	0
s5378	572	160	16	140	91	16	128	72
s38584	6211	1260	64	393	257	64	393	245
iir16	3612	522	37	292	127	37	292	127
fir16	6598	847	84	374	228	84	441	232
ralu32	3659	590	20	216	128	20	210	128
spsdes	3356	949	26	150	96	26	136	94
mac64	4307	64	15	151	96	11	108	62
mips64	2226	438	10	127	68	8	120	64
sort8	1861	184	24	142	90	24	102	69
ochip64	3314	3665	10	160	48	0	0	0
average			59.7	358	274	56.7	322	253

solution found so far, then processing for that seed terminates. Neither of these optimizations change the final solution, but both significantly speed up the code on our benchmark circuits.

IV. RESULTS FOR ONE MEMORY ARRAY

To evaluate the proposed algorithms, we used 29 large benchmark circuits. As shown in Table II, each circuit contained between 527 and 6598 4-LUT's. Seventeen of the circuits were sequential. The combinational circuits and nine of the sequential circuits were obtained from the Microelectronics Corporation of North Carolina (MCNC) benchmark suite, while the remaining sequential circuits were obtained from the University of Toronto, and were the result of synthesis from VHDL and Verilog. All circuits were optimized using SIS (both script.rugged

and script.algebraic were attempted, and the better result for each circuit was used) [33] and technology-mapped to four-input LUT's (4-LUT's) using Flowmap and Flowpack [17]. Four-input LUT's were assumed since these are common on many of today's commercial FPGA's (see Table I).

The fifth column of Table II shows the number of 4-LUT's that SMAP deletes from the circuit due to a single 2-Kbit array that can take on word widths of 1, 2, 4, or 8; this is the size of the memory array used in the Altera FLEX 10K CPLD [3]. As the table shows, SMAP reduces the number of 4-LUT's required to implement each circuit by 59.7, on average. Column eight shows the results for SMAP-dc. In this case, the number of 4-LUT's that can be deleted is reduced only slightly to 56.7 (only 5% less than SMAP). Execution times for SMAP using two medium-sized combinational circuits and two large sequential circuits

TABLE III
DEPTH RESULTS ASSUMING ONE 2048-BIT MEMORY ARRAY

Circuit Name	Original Depth	After SMAP		After SMAP-dc	
		Depth	% Increase	Depth	% Increase
pair	7	7	0%	7	0%
apex1	7	8	14.3%	7	0%
cps	5	6	20%	5	0%
C5315	10	10	0%	10	0%
C6288	28	30	7.14%	28	0%
apex3	6	7	16.7%	6	0%
C7552	9	9	0%	9	0%
i10 *	13	16	23.1%	13	0%
ex5p	7	7	0%	7	0%
spla	8	8	0%	8	0%
pdv	9	9	0%	9	0%
apex4	6	6	0%	6	0%
tseng	13	13	0%	13	0%
bigkey	3	6	100%	3	0%
s38417	11	11	0%	11	0%
s298	15	15	0%	15	0%
diffeq	14	14	0%	14	0%
frisc	23	23	0%	23	0%
dsip *	3	6	100%	3	0%
s5378	6	9	50%	6	0%
s38584	9	9	0%	9	0%
iir16	32	32	0%	32	0%
fir16	24	24	0%	24	0%
ralu32	17	17	0%	17	0%
spsdes	34	34	0%	34	0%
mac64	52	55	5.77%	52	0%
mips64	54	55	1.85%	54	0%
sort8	25	25	0%	25	0%
ochip64	3	6	100%	3	0%
average			15.1%		0%

TABLE IV
DELAY RESULTS ASSUMING ONE 2048-BIT MEMORY ARRAY

Circuit Name	Original Depth	After SMAP		After SMAP-dc	
		Depth	% Increase	Depth	% Increase
pair	48.1 ns	53.3 ns	10.8%	53.3 ns	10.8%
apex1	47.3 ns	51.2 ns	8.25%	50.9 ns	8.25%
cps	46.2 ns	46.0 ns	-0.43%	48.4 ns	4.76%
C5315	59.0 ns	60.4 ns	2.37%	60.4 ns	2.37%
C6288	118 ns	116.8 ns	-0.77%	113.6 ns	-3.73%
apex3	52.7 ns	52.6 ns	-0.19%	56.2 ns	6.64%
C7552	96.9 ns	91.4 ns	-5.19%	91.4 ns	-5.19%
i10 *	151 ns	156.1 ns	3.51%	145.6 ns	-3.58%
ex5p	59.5 ns	63.3 ns	6.39%	63.3 ns	6.39%
spla	87.1 ns	87.1 ns	0%	87.1 ns	0%
pdv	106 ns	104.8 ns	-1.41%	104.8 ns	-1.41%
apex4	63.8 ns	51.3 ns	-19.6%	51.3 ns	-19.6%
tseng	66.2 ns	67.6 ns	2.11%	67.6 ns	2.11%
bigkey	16 ns	24.4 ns	52.5%	16 ns	0%
s38417	99.9 ns	100.6 ns	0.701%	100.6 ns	0.70%
s298	122.8 ns	100.6 ns	-18.1%	100.6 ns	-18.1%
diffeq	67.8 ns	95.8 ns	41.3%	66.3 ns	-2.21%
frisc	106.8 ns	128.9 ns	20.7%	111 ns	3.93%
dsip *	15.7 ns	14.9 ns	-5.10%	15.7 ns	0%
s5378	27.7 ns	37.6 ns	35.7%	31.8 ns	14.8%
s38584	81.7 ns	85.5 ns	5.88%	85.5 ns	5.88%
iir16	154.9 ns	165.1 ns	6.58%	165.1 ns	6.58%
fir16	213.7 ns	290.5 ns	35.9%	290.5 ns	35.9%
ralu32	101.0 ns	107.3 ns	6.24%	107.3 ns	6.24%
spsdes	130.2 ns	129.9 ns	-0.23%	129.9 ns	-0.23%
mac64	256.4 ns	234.6 ns	-8.50%	223.2 ns	-12.9%
mips64	264.6 ns	272.7 ns	3.06%	265.5 ns	0.3%
sort8	162.0 ns	191.5 ns	18.21%	191.5 ns	18.21%
ochip64	26.9 ns	27.2 ns	1.12%	26.9 ns	0%
average			6.96%		2.31%

are shown in the third column of Table V (the execution times for SMAP and SMAP-dc are roughly the same).

To appreciate the significance of these results, consider the following. Using a detailed area model, we have estimated that the chip area required by a single 2-Kbit memory array is the same as the area required to implement 16 4-LUT's (including routing) [34]. This area would be wasted if an unused array is not used to implement logic. Using SMAP (or SMAP-dc), however, not only is this area not wasted, but it is used *more efficiently that it would have been used if the array was replaced by logic blocks*. Had the array not been present, the user would be able to implement 16 4-LUT's of his/her circuit in that chip area, while, using SMAP, the user can implement 59.7 4-LUT's of circuitry in the same chip area (56.7 using SMAP-dc). Thus, the presence of embedded memory blocks leads to density improvements even if the user's circuit requires no storage at all.

Table III shows the effect of our algorithms on the combinatorial depth of each circuit. The results in the table assume that the depth of a lookup-table is one, while the depth of a memory block is three (ie. $h_m = 3$). This is the same ratio used in [27], and closely reflects the ratios between memory read time and lookup-table access time found in commercial devices. As can be seen, SMAP increases the combinatorial delay by (on average) 15.1%. SMAP-d, on the other hand, does not increase the combinatorial delay at all.

To investigate the effect that this would have on the critical path of the final circuit after place and route, we mapped our benchmark circuits to an Altera FLEX 10K using the MAX+PlusII version 8.3 software and measured the achievable clock frequencies after placement and routing (for combinatorial circuits, we measured the maximum combinatorial path).

```

for  $i=1$  to  $N$  {
  find the best seed node as before
  connect memory array  $i$  and
  delete all nodes no longer
  needed due to this array
}

```

a) Algorithm 1

```

for  $i=1$  to  $N/BF$  {
  find the best seed node using  $BF$ 
  arrays combined into "super-array"
  connect memory arrays and delete
  nodes no longer needed
}

```

b) Algorithm 2

Fig. 7. Two possible algorithms for multiple memory arrays.

Table IV shows the delay of the original circuit and the circuit after having mapped by SMAP and SMAP-dc. The percentage increase in the critical path delay is also shown; those circuits where the critical path decreased are shown as negative. Those circuits marked with a * could not be mapped to a single device and so were split between two devices. As shown in the table, the critical path increased by, on average, 6.96% using SMAP, and 2.3% using SMAP-d.

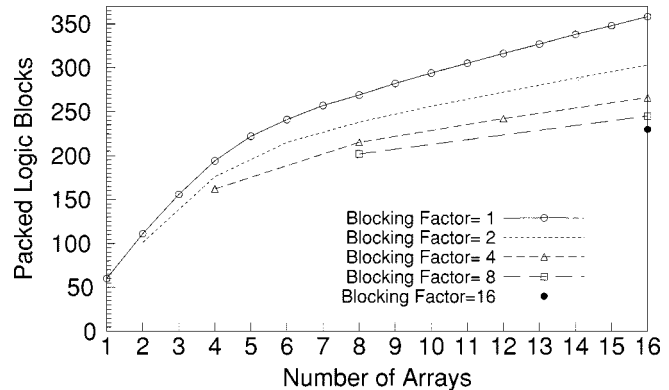
The results presented in this section assume only a single memory array is available. In the next section, we examine how the algorithms can be applied to FPGA's with more than one available memory array. Comparisons to the algorithm described in [27] will be presented in Section VI.

V. EXTENSION TO MULTIPLE ARRAYS

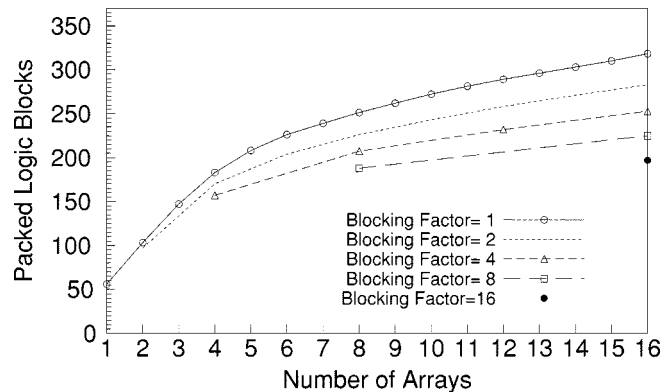
The previous discussion assumed that only a single memory array is available. One way to extend the algorithm to multiple arrays is shown in Fig. 7(a). In this algorithm, after mapping to the first array (using either SMAP or SMAP-dc), we remove the nodes implemented by that array, and repeat the entire algorithm for the second array. This is repeated for each available array. The results are shown by the upper lines in Fig. 8(a) and (b); if there are 16 arrays available, the SMAP is able to delete, on average, 358 logic blocks (22 logic blocks per array) while SMAP-dc can delete 318 logic blocks on average.

The problem with this algorithm is that it is slow for large circuits. Execution times for four of the circuits are shown in columns four and seven of Table V. The majority of the algorithm's execution time is spent choosing a seed node, and this decision must be repeated N times. Fig. 7(b) shows an alternative. We partition the N available arrays into N/BF larger "super-arrays," each containing BF arrays ($BF \leq N$). We refer to BF as the *Blocking Factor*. We then enumerate all possible width/depth configurations for each super-array, and apply SMAP to each of the N/BF super-arrays. If $BF = 1$, this reverts to the first algorithm. If BF is large, the CPU requirements are significantly reduced (see Table V), since only N/BF seed nodes need be chosen. Fig. 8(a) and (b) shows the results for SMAP and SMAP-dc with various Blocking Factors. Numerical results for two values of BF and $N = 16$ are shown in Table II.

Note that in SMAP-dc, it is also necessary to recompute the slacks before processing each memory array. The time required to recompute the slacks, however, is small in comparison with the rest of the mapping procedure.



(a)



(b)

Fig. 8. Results for multiple 2048-bit arrays.

VI. COMPARISONS TO OTHER PUBLISHED ALGORITHMS

In [27], an alternative algorithm, called EMB_Pack, is described. The goal of EMB_Pack is the same as that of SMAP-dc in that EMB_Pack tries to minimize the area with the constraint that the depth of the circuit must not increase.

To compare our results to those presented in [27], we tested SMAP-dc on the mapped benchmark circuits in that paper. These circuits are combinational, and require between 83 and 1367 four-input LUT's. Since EMB_Pack may not choose to use all available memory arrays, we used the number of memory arrays actually used by EMB_Pack as input to SMAP-dc. The circuit duke2 is not included in the table, since EMB_Pack chose not to use any memory arrays at all for that circuit. The results are shown in Table VI. As can be seen, SMAP-dc is able to pack, on average, 88.3% more logic into the available memory arrays than EMB_Pack. The delay of the

TABLE V
SAMPLE CPU RUN TIMES (IN SECONDS) ON A 143 MHz ULTRASPARC

Circuit Name	Number 4-LUTs	1 array	8 arrays				16 arrays			
			Alg 1	Alg 2	Alg 3		Alg1	Alg 2	Alg 3	
					BF=4	BF=8			BF=8	BF=16
ex5p	1064	7.7	21.8	10.1	11.9	11.7	22.0	10.2	11.7	12.5
apex4	1262	31.2	60.6	33.5	48.1	49.1	61.2	33.9	49.5	50.9
s38584	6211	127	1010	128.4	286	158	1906	128.4	299	163
iir16	3612	98.3	826	99.2	283	164	1693	99.9	335	194

TABLE VI
COMPARISON TO EMB_PACK ALGORITHM

Circuit Name	Number 4-LUTs in Original Circuit	Number of Arrays Available	Number 4-LUTs Packed			Increase in Critical Path	
			EMB.Pack [27]	SMAP-dc	% Diff.	EMB.Pack [27]	SMAP-dc
C5315	673	3	48	48	0%	7.73%	12.2%
C6288	555	3	22	45	105%	-5.19%	-3.52%
C7552	850	5	56	76	35.7%	-9.01%	-13.8%
C880	142	3	28	31	10.7%	17.6%	0.17%
9sym	90	1	90	90	0%	-20.3%	-28.2%
9symml	94	1	94	94	0%	-18.0%	-18.0%
alu2	192	3	18	126	600%	16.8%	-27.7%
alu4	326	3	22	106	382%	7.52%	-6.68%
apex6	300	3	21	27	28.6%	21.5%	11.1%
apex7	88	3	14	20	42.9%	4.22%	2.76%
des	1367	5	38	29	-23.7%	5.43%	8.66%
i10	1166	8	84	113	34.5%	1.36%	-0.21%
pair	641	3	70	84	20%	-11.9%	-1.43%
rd84	83	1	83	83	0%	-13.6%	-21.9%
average			49.1	69.4	88.3%	0.297%	-6.18%

circuits produced by SMAP-dc is slightly smaller, on average, than those produced by EMB_Pack.

VII. EFFECT OF MEMORY BLOCK ARCHITECTURE

The results presented in the last section assume an FPGA with N 2048-bit arrays, each of which can be configured as 2048×1 , 1024×2 , 512×4 , or 256×8 . This is array size and flexibility used in the Altera FLEX 10K devices. In this section, we present results showing the ability of SMAP to map logic to FPGA's with different array dimensions. In doing so, we identify which memory array architecture can be most efficiently used by SMAP.

The memory architectures of several commercial devices are summarized in Table I. In this section, we vary B and w_{eff} .

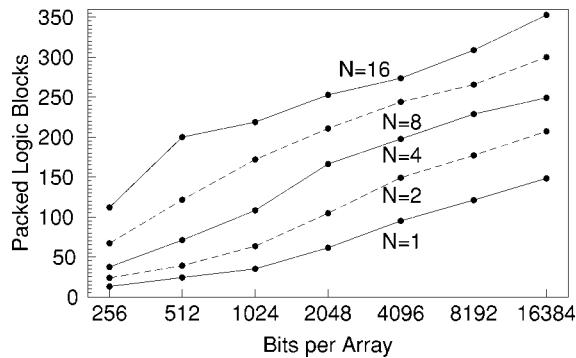
A. Array Size

Fig. 9 shows how the effectiveness of each memory block in implementing logic depends on the array size, B . Fig. 9(a) shows the number of logic blocks that can be packed into the

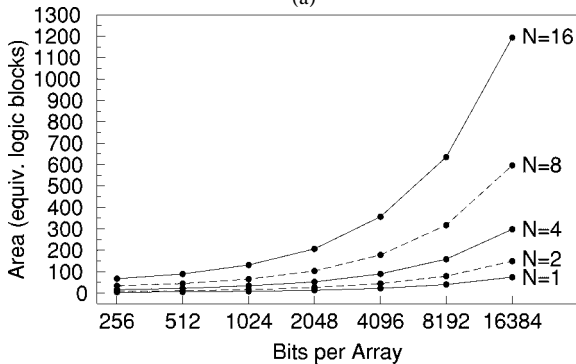
arrays (averaged over our benchmark circuits) vs. B for several values of N . For $N > 1$, a blocking factor of N was used.

In order to determine which architecture is most efficiently used by the algorithm, we need to consider the chip area required by each architecture. Fig. 9(b) shows the estimated chip area of each memory block as a function of B , again for various values of N . The area estimates were obtained from a detailed area model [34] and are expressed in *logic block equivalents* (LBE). One LBE is the area required to implement one logic block. For $N > 1$, these area estimates include the area due to the multiplexors needed to combine the arrays into a single super-array (these multiplexors are assumed to be implemented using the FPGA logic resources).

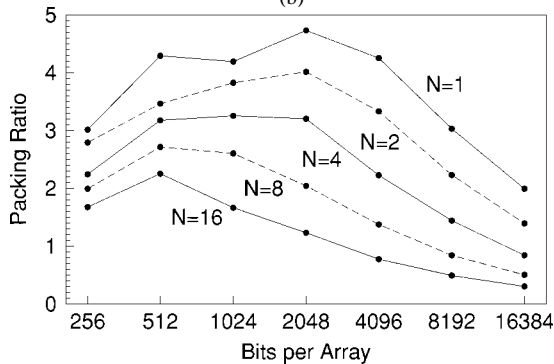
Fig. 9(c) shows the *packing density* as a function of B for several values of N . The packing density is defined as the ratio of the number of logic blocks that can be packed into the available memory arrays over the area required to implement the memory arrays (in LBE's). A packing density of one means that the density of logic implemented in memory arrays is equal to that if the logic was implemented in logic blocks. A packing density



(a)



(b)



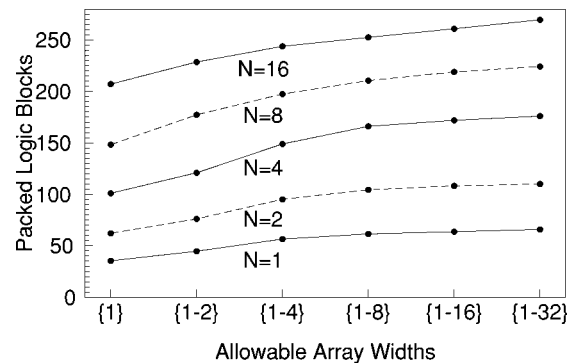
(c)

Fig. 9. Effects of memory array size.

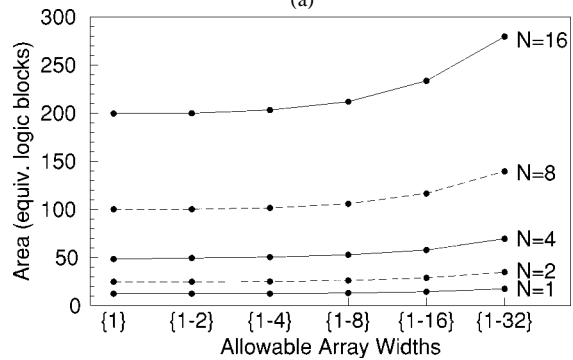
greater than one means that the density of logic implemented in memory arrays is *greater* than that if logic blocks were used. As Fig. 9(c) shows, the packing density is greater than one for all but the largest memory arrays. The highest packing density occurs for $B = 2048$ if there are two or fewer arrays; if more arrays are available, the optimum value of B is slightly smaller.

B. Array Width

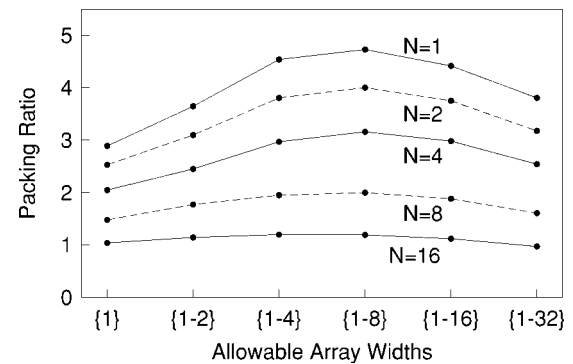
As described above, most FPGA memory architectures allow the user to select the word width of each array from a predetermined set, w_{eff} . Fig. 10(a) shows how the effectiveness of the memory arrays in implementing logic depends on w_{eff} . Fig. 10 shows the number of logic blocks packed, the area requirements, and the packing density as a function of the maximum allowable width (highest value in w_{eff}) for each memory array. In all cases, it is assumed that each array contains 2048 bits, and that the user can configure the memory width to be any power-of-two between one and the maximum allowable width. Thus, for the



(a)



(b)



(c)

Fig. 10. Effects of maximum allowable width.

left-most point on the graph, only a 2048×1 configuration is available, while for the right-most point, each array can be configured to any configuration between 2048×1 and 64×32 . As the figures show, the highest packing density occurs if the maximum available width is eight. Arrays with a larger width require more area (more sense amplifiers and routing) while arrays with a smaller maximum width can not implement as much logic. The optimum choice for the maximum allowable width is independent of N . Fig. 11 shows the results as a function of the minimum allowable width, assuming that $B = 2048$ and that the maximum allowable width is eight. Thus, for the left-most point in the graphs, the user can configure each array to be one of 2048×1 , 1024×2 , 512×4 , or 256×8 , while in the architecture corresponding to the right-most point, only the 256×8 configuration is available. As shown in Fig. 11(a), more circuit information can be packed into the more flexible arrays. The area results in Fig. 11(b) appear counterintuitive; according to this figure, the more flexible an architecture is, the less area it

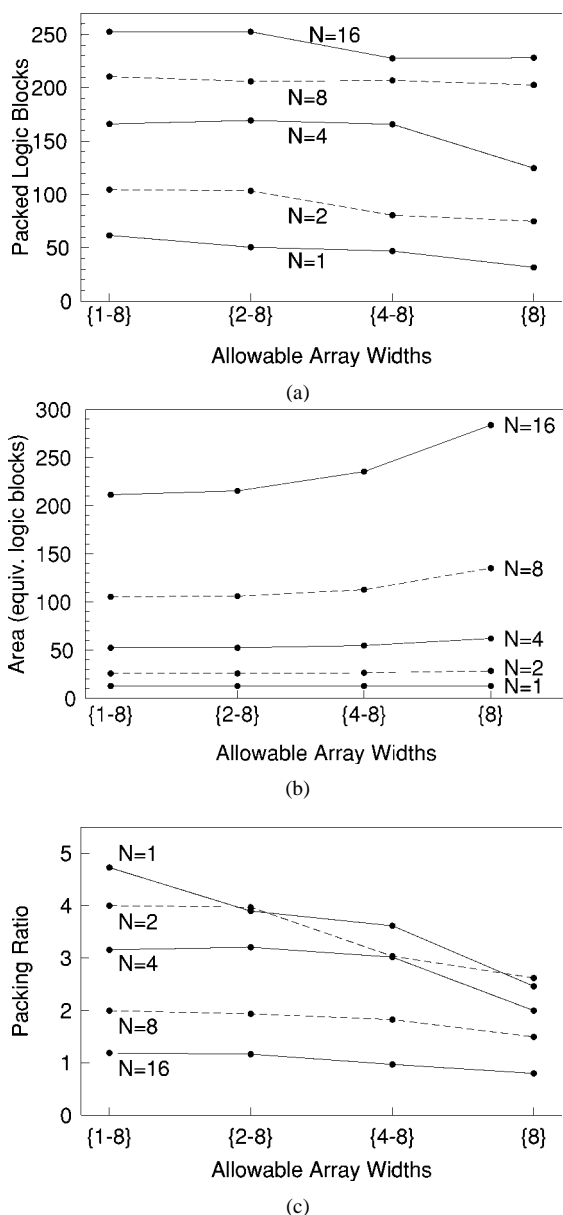


Fig. 11. Effects of minimum allowable width.

requires. This is because the area measurements in this graph include the area required to implement the multiplexers needed to combine the N arrays into a super-array. The multiplexers are larger if an array is used in the “ $\times 8$ ” configuration than in the “ $\times 1$ ” configuration, since more bits must be multiplexed; thus, an architecture in which only a “ $\times 8$ ” configuration is allowed ($w_{\text{eff}} = \{8\}$) will require larger multiplexers, on average. This effect is more pronounced for larger values of N ; this is because the more arrays there are to combine, the more multiplexers are required. The packing ratio is shown in Fig. 11(c); in all cases, the more flexible architecture is the best choice.

Overall, the best choice for w_{eff} is $\{1, 2, 4, 8\}$.

VIII. CONCLUSION

In this paper, we have described two versions of an algorithm that maps logic circuits to FPGA's with large embedded memory arrays. Although these embedded arrays were orig-

inally conceived to implement the storage parts of circuits, we have shown that if not all arrays are needed to implement storage, the unused arrays need not be wasted. Instead, they can be used to implement logic. Our first algorithm, SMAP, packs as much circuit information as possible into the available memory arrays, and maps the rest of the circuit to lookup-tables. The second algorithm does the same, but under the constraint that the depth of the circuit does not increase beyond the depth of the circuit implemented using only LUT's.

On a set of 29 sequential and combinational benchmarks, the first algorithm is able to map, on average, 59.7 4-LUT's into a single 2-Kbit memory array. The second algorithm is able to map 56.7 4-LUT's without increasing the depth of the circuit. If there are 16 arrays available, the first algorithm can map 358 4-LUT's to the 16 arrays, while the second can map 318 4-LUT's. These densities are better than would have been obtained had the FPGA contained nothing but 4-LUT's. Thus, not only are the arrays not wasted, but, their area is used more efficiently than if the arrays were replaced by logic blocks. The presence of embedded memory blocks leads to density improvements even if the user's circuit requires no storage at all.

We have also shown that the algorithm works well over a wide variety of architectures. Overall, we found that the algorithm works best when each memory array contains between 512 and 2048 bits, and has a word width that can be configured to be 1, 2, 4, or 8.

ACKNOWLEDGMENT

The author would like to thank S. Xu and J. Cong of the University of California, Los Angeles, for providing the benchmark circuits used to evaluate EMB_Pack.

REFERENCES

- [1] *XC4000E and XC4000X Series Field Programmable Gate Arrays, ver. 1.6*, Xilinx, Inc., May 1999.
- [2] DynaChip DY8000 Family Datasheet, DynaChip Corp., June 1999.
- [3] FLEX 10K Embedded Programmable Logic Family Data Sheet, ver. 4.01, Altera Corporation, June 1999.
- [4] FLEX 10KE Embedded Programmable Logic Family Data Sheet, ver. 2.01, Altera Corporation, June 1999.
- [5] APEX 20K Programmable Logic Device Family Data Sheet, ver. 2.0, Altera Corporation, May 1999.
- [6] *Virtex 2.5 V Field Programmable Gate Arrays, ver. 1.6*, Xilinx, Inc., July 1999.
- [7] Vantis VF1 FPGA Data Sheet, Lattice Semiconductor, Nov. 1998.
- [8] Data sheet: ProASIC 500K Family, Actel Corporation, June 1999.
- [9] MX FPGA Data Sheet, Actel Corporation, Jan. 1999.
- [10] Datasheet: Integrator Series FPGAs: 1200XL and 3200DX Families, Actel Corporation, June 1998.
- [11] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, “Architecture of centralized field-configurable memory,” in *Proc. ACM/SIGD Int. Symp. Field-Programmable Gate Arrays*, 1995, pp. 97–103.
- [12] —, “Memory/logic interconnect flexibility in FPGA's with large embedded memory arrays,” in *Proc. IEEE 1996 Custom Integrated Circuits Conference*, May 1996, pp. 144–147.
- [13] —, “Memory-to-Memory connection structures in FPGA's with embedded memory arrays,” in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Feb. 1997, pp. 10–16.
- [14] T. Ngai, J. Rose, and S. J. E. Wilton, “An SRAM-Programmable field-configurable memory,” in *Proc. IEEE 1995 Custom Integrated Circuits Conference*, May 1995, pp. 499–502.
- [15] M. Hutton, J. Grossman, J. Rose, and D. Corneil, “Characterization and parameterized random generation of digital circuits,” in *Proceedings of ACM/IEEE Design Automation Conference*, June 1996, pp. 94–99.

- [16] R. Francis, J. Rose, and Z. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based FPGAs," in *Proc. ACM/IEEE Design Automation Conference*, June 1991, pp. 227–233.
- [17] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 1–12, Jan. 1994.
- [18] R. Murgai, F. Hirose, and M. Fujita, "Logic synthesis for a single large look-up table," in *Proc. Int. Workshop on Logic Synthesis*, May 1995.
- [19] J. He and J. Rose, "Technology mapping for heterogeneous FPGAs," in *Proc. ACM Int. Workshop on Field Programmable Gate Arrays*, Feb. 1994.
- [20] J. Cong and S. Xu, "Delay-optimal technology mapping for FPGA's with heterogeneous LUTs," in *Proc. 35th Design Automation Conference*, June 1998, pp. 704–707.
- [21] —, "Delay-oriented technology mapping for heterogeneous FPGA's with bounded resources," in *Proc. ACM/IEEE Int. Conference on Computer-Aided Design*, Nov. 1998, pp. 40–45.
- [22] D. Filo, J. C.-Y. Yang, F. Mailhot, and G. De Micheli, "Technology mapping for a two-output RAM-based field-programmable gate array," in *Proc. Eur. Conf. Design Automation*, Feb. 1991, pp. 534–538.
- [23] R. Murgai, Y. Nishizaki, N. Shenoy, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," in *Proc. ACM/IEEE Design Automation Conf.*, June 1990, pp. 620–625.
- [24] G. D. Micheli and M. Santomauro, "Smile: A computer program for partitioning of programmed logic arrays," *Computer-Aided Design*, vol. 15, no. 2, pp. 89–97, 1983.
- [25] M. Ciesielski and S. Yang, "Plade: A two-stage PLA decomposition," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 943–954, Aug. 1992.
- [26] S. J. E. Wilton, "SMAP: Heterogeneous technology mapping for FPGA's with embedded memory arrays," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Feb. 1998, pp. 171–178.
- [27] J. Cong and S. Xu, "Technology mapping for FPGA's with embedded memory blocks," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Feb. 1998, pp. 179–187.
- [28] S. J. E. Wilton, "Implementing logic in FPGA embedded memory arrays: Architectural implications," in *Proc IEEE Custom Integrated Circuits Conference*, May 1998, pp. 12.4.1–12.4.4.
- [29] J. Cong and Y. Ding, "Combinational logic synthesis for LUT based field programmable gate arrays," *ACM Trans. Design Automation Electron. Syst.*, vol. 1, pp. 145–204, Apr. 1996.
- [30] —, "On area/depth trade-off in LUT-based FPGA technology mapping," in *Proc. 30th Design Automation Conference*, June 1993, pp. 213–218.
- [31] J. Cong, P. Li, S. Lim, T. Shibuya, and D. Xu, "Large scale circuit partitioning with loose/stable net removal and signal flow based clustering," in *Proc. IEEE/ACM Int. Conference on Computer-Aided Design*, Nov. 1997, pp. 441–446.
- [32] R. Hitchcock, G. Smith, and D. Cheng, "Timing analysis of computer hardware," *IBM J. Res. Develop.*, pp. 100–105, Jan. 1983.
- [33] E. Sentovich, "SIS: A system for sequential circuit analysis," Electronics Research Laboratory, University of California, Berkeley, CA, Tech. Rep. UCB/ERL M92/41, May 1992.
- [34] S. J. E. Wilton, "Architectures and algorithms for field-programmable gate arrays with embedded memory," Ph.D. dissertation, Univ. Toronto, Toronto, ON, Canada, 1997.



Steven J. E. Wilton received the B.Eng. degree in computer engineering from the University of Victoria, Victoria, B.C., Canada in 1990 and the M.A.Sc. and Ph.D. degree from the University of Toronto, Toronto, Ont., Canada in 1992 and 1997 respectively.

Since 1997, he has been an Assistant Professor in the Department of Electrical and Computer Engineering at the University of British Columbia in Vancouver, B.C., Canada. In 1999, he was appointed a Fellow of the British Columbia Advanced Systems

Institute. His research interests include FPGA architectures, algorithms for FPGA's, and VLSI design.

In 1998, Dr. Wilton won the Douglas R. Colton Medal for Research Excellence for his work in FPGA architectures and their associated CAD tools.