

# SMAP: Heterogeneous Technology Mapping for Area Reduction in FPGAs with Embedded Memory Arrays

Steven J.E. Wilton

Department of Electrical and Computer Engineering

University of British Columbia

Vancouver, BC, Canada, V6T 1Z4

steve@ee.ubc.ca \*

## Abstract

*It has become clear that large embedded configurable memory arrays will be essential in future FPGAs. 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 large multi-output ROMs, and used to implement logic.*

*In order to efficiently use the embedded arrays in this way, a technology mapping algorithm that identifies parts of circuits that can be efficiently mapped to an embedded array is required. In this paper, we describe such an algorithm. The new tool, called SMAP, packs as much circuit information as possible into the available memory arrays, and maps the rest of the circuit into four-input lookup-tables. On a set of 29 sequential and combinational benchmarks, the tool is able to map, on average, 60 4-LUTs into a single 2-Kbit memory array. If there are 16 arrays available, it can map, on average, 358 4-LUTs to the 16 arrays.*

## 1 Introduction

It has become clear that on-chip storage is critical in large FPGAs. As FPGAs grow, they are being used to implement entire systems, rather than the small logic

subcircuits that have traditionally been targeted to FPGAs. 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 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 FPGAs have emerged: fine-grained and coarse-grained. In FPGAs employing fine-grained memory, such as the Xilinx 4000 series FPGAs, each lookup-table can be configured as a small RAM, and these RAMs can be combined to implement larger user memories [1]. The coarse-grained approach is used in the Altera 10K devices [2], the Actel 3200DX and SPGA parts [3, 4], and the Lattice ispLSI 6192 FPGAs [5]. In these devices, large arrays are embedded onto the FPGA. Devices in the Altera 10K family contain between 3 and 16 two-Kbit arrays, the Actel 3200DX parts contain between 8 and 16 256-bit arrays, Actel SPGAs contain between 2 and 32 two-Kbit arrays, and the Lattice ispLSI 6192 devices contain a single 4608 bit array. Several academic studies have also focused on coarse-grained memory architectures [6, 7, 8].

The coarse-grained approach results in significantly denser memory implementations, since the per-bit overhead is much smaller [9]. 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 work was supported by Cadence Design Systems, the Natural Sciences and Engineering Research Council of Canada, and UBC's Centre for Integrated Computer Systems Research.

This chip area need not be wasted, however, if the unused memory arrays are used to implement logic. Configuring the arrays as ROMs results in large multi-output 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 [10]. 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 efficiently mapped to the available arrays, and implements the rest using normal look-up tables is key to using the available memory arrays effectively. Such an algorithm is the focus of this paper.

Implementing logic in memory arrays has been studied by Murgai [11]. 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 also studied by He and Rose [12]. Their algorithm targets FPGAs with two sizes of single-output lookup-tables, and is not immediately extendible to lookup-tables with multiple outputs. Technology mapping to multiple-output lookup-tables has been studied [13, 15]. 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 PLAs [16, 17]. Unlike PLAs, however, memories can implement any function of their inputs, without regard for the number of product terms.

In this paper, we will use the following terminology (from [18]). 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. 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. 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  $X'$ .

## 2 Problem Definition

Technology-mapping algorithms generally attempt to minimize either the area required to implement circuits (or equivalently, maximize the size of circuits that can be implemented on a given FPGA) or minimize the delay of the longest combinational path in the circuit (and hence maximize the achievable clock frequency). In this paper, we focus on the first of these goals. The result of the algorithm is a circuit mapped to both memory arrays and small lookup-tables. We assume a fixed number of available arrays, and minimize the number of lookup-tables that are required to implement the parts of the circuit that can not be implemented in the available arrays. Thus, assuming an FPGA with  $N$  available memory arrays, the problem can be defined as:

Given a circuit, find a mapping to  $n$  memory arrays and  $m$   $k$ -input lookup tables ( $k$ -LUTs) where  $n \leq N$  and  $m$  is minimum.

In this paper, we assume  $k = 4$ . In the next section, we will focus on a restricted version of this problem, in which  $N = 1$ . In Section 5, we extend the algorithm for  $N > 1$ .

## 3 Single Array Algorithm

There are several possible approaches to solving the problem specified in the previous section. Our solution is as follows. We first map the entire circuit to  $k$ -feasible nodes using an existing technology mapper (we use Flowmap/Flowpack [19]). 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  $k$ -LUTs.

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 3.1 and 3.2 will describe the selection of the memory input and output signals; Section 3.3 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 3.4,

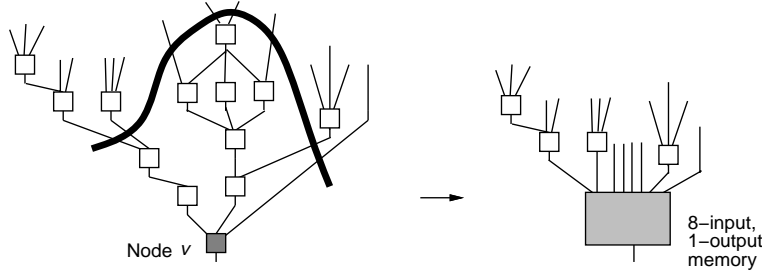


Figure 1: Mapping a Circuit to a 8-Input, 1-Output Memory Block

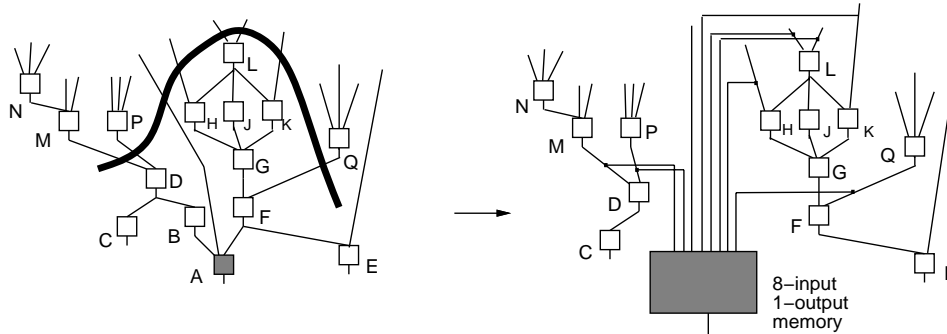


Figure 2: Mapping a Circuit to a 8-Input, 1-Output Memory Block: Poor Solution

we will show how the algorithm can be extended to support arrays with a configurable data width.

### 3.1 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 efficient heuristic algorithm to find such a cut was presented in [19]; we use the same algorithm here.

### 3.2 Memory Output Signals

If there is only one memory output ( $w = 1$ ), we can select the seed node output as the memory array output. Figure 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 [19].

For the large memory sizes considered in this paper, this simple extension of Flowpack does not work well for some circuits. It is also not clear how to extend the algorithm to support multi-output 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 multi-output blocks.

In Figure 1, all predecessors of the seed node that are below the cut can be packed into the memory array, and the LUTs 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 Figure 1, all nodes below the cut were also in the MFFC of  $v$ . Figure 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.

Figure 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

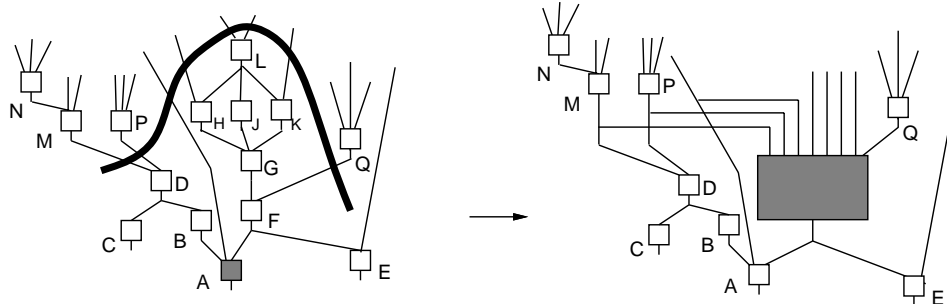


Figure 3: Mapping a Circuit to a 8-Input, 1-Output Memory Block: Better Solution

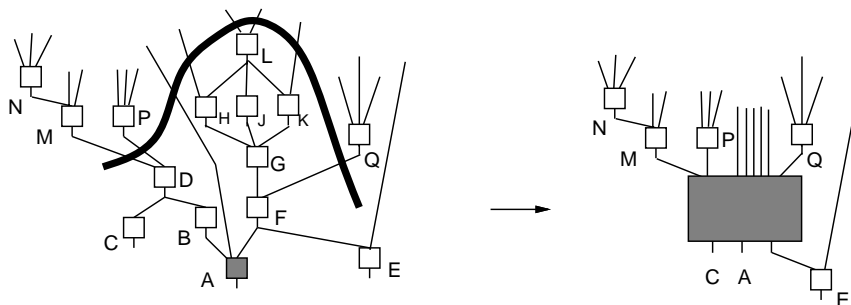


Figure 4: Mapping to a 8-Input, 3-Output Memory Block

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

If the memory array has 3 outputs instead of 1, the solution in Figure 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 Figures 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 Figure 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  $\frac{|P|!}{w!(|P|-w)!}$  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 MFFCs. In Section 4, however, we will show that this heuristic works well.

Figure 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.

### 3.3 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.

```

find_this_solution(seed,  $N_{in}$ ,  $N_{out}$ ) {
  cut_set = max-vol  $N_{in}$ -feasible cut
   $P$  = set of Potential Nodes driven by cut_set
  output_nodes = output_selection( $P$ ,  $N_{out}$ )
  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
}

```

Figure 5: Summary of Algorithm Given Seed Node and Array Configuration

```

SMAP{
  for each node  $v$  {
    sol = find_this_solution( $v$ , 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 = find_this_solution( $u$ , number inputs for config  $i$ , 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
}

```

Figure 6: Summary of SMAP Algorithm for One Memory Array.

### 3.4 Arrays with a Configurable Width

Many FPGAs 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 2048x1, a 1024x2, a 512x4 or a 256x8 memory. Thus, each array can be used as either a 11-input,1-output, a 10-input,2-output, a 9-input,4-output, or a 8-input,8-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 Figure 6.

## 4 Results and Discussion

To evaluate the proposed algorithm, we used 29 large benchmark circuits. As shown in Table 1, each circuit contained between 527 and 6598 4-LUTs. Seventeen of the circuits were sequential. The combinational circuits and 9 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) [20] and technology-mapped to 4-input lookup tables using Flowmap and Flowpack [19].

The third column of Table 1 shows the number of 4-LUTs that can be deleted from the circuit due to a single 2-Kbit array that can take on data widths of 1, 2, 4, or 8; this is the size of the memory array used in the Altera 10K CPLD [2]. As the table shows, SMAP reduces the number of 4-LUTs required to implement each circuit by 60, on average. Execution times for two medium-sized combinational circuits and two large sequential circuits are shown in the third column of Table 2.

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 K-bit memory array is the same as the area required to implement 16 4-LUTs (including routing) [21]. This area would be wasted if an unused array is not used to implement logic. Using SMAP, however, not only is this area

Circuit Name	Original Circuit		Nodes that can be deleted						
	Number of 4-LUTs	Number of Flip Flops	1 array	4 arrays		8 arrays		16 arrays	
				BF=1	BF=4	BF=1	BF=8	BF=1	BF=16
pair	641	0	13	45	23	81	23	148	23
apex1	696	0	14	47	36	87	68	159	88
cps	749	0	46	117	108	173	147	249	204
C5315	596	0	12	42	23	76	26	141	27
C6288	527	0	19	61	29	93	35	140	43
apex3	867	0	11	75	74	119	109	199	173
C7552	679	0	15	57	20	94	29	163	13
i10	994	0	18	55	42	94	74	170	77
ex5p	1064	0	198	810	780	1043	1044	1056	1064
spla	3690	0	67	200	86	349	137	569	213
pdc	4575	0	88	277	138	480	331	816	568
apex4	1262	0	319	1205	1205	1261	1261	1261	1261
tseng	1046	385	14	41	15	75	16	130	21
bigkey	1707	224	18	48	30	88	31	155	33
s38417	6096	1463	26	102	50	193	63	346	63
s298	1930	8	434	1445	1445	1878	1819	1930	1906
diffeq	1494	377	22	70	26	120	15	206	7
frisc	3539	886	62	94	63	129	63	183	63
dsip	1370	224	18	48	25	87	25	167	25
s5378	572	160	16	52	29	91	31	155	37
s38584	6211	1260	64	159	62	259	68	415	179
iir16	3612	522	38	95	54	165	54	293	54
fir16	6598	847	84	180	123	252	139	384	156
ralu32	3659	590	20	65	32	118	64	216	128
spsdes	3356	949	31	61	55	92	62	152	75
mac64	4307	64	15	50	25	86	31	152	42
mips64	2226	438	10	36	17	68	23	130	24
sort8	1861	184	28	54	52	83	64	143	66
ochip64	3314	3665	10	40	15	80	23	160	30
average			60	194	162	269	202	358	230

Table 1: Results assuming 2048-bit memory arrays

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-LUTs of his/her circuit in that chip area, while, using SMAP, the user can implement 60 4-LUTs of circuitry in the same chip area. Thus, the presence of embedded memory blocks leads to density improvements even if the user’s circuit requires no storage at all.

The results presented in this section assume only a single memory array is available. In the next section, we examine how the algorithm can be applied to FPGAs with more than one available memory array.

## 5 Extension to Multiple Arrays

The previous discussion assumed that only a single memory array is available. We have investigated three methods of extending the algorithm to multiple arrays;

they are summarized in Figure 7. In Algorithm 1, after mapping to the first array, 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 line in Figure 8(a); if there are 16 arrays available, the algorithm is able to delete, on average, 358 logic blocks (22 logic blocks per array).

The problem with Algorithm 1 is that it is slow for large circuits. Execution times for four of the circuits are shown in columns 4 and 8 of Table 2. The majority of the algorithm’s execution time is spent choosing a seed node, and this decision must be repeated  $N$  times. Algorithms 2 and 3 are attempts to reduce the execution time. In Algorithm 2, we perform the first step of the algorithm (selecting the seed node) only once. Rather than selecting a single seed, we select the  $N$  best seed nodes, where  $N$  is the number of available arrays. Each seed node is

Circuit Name	Number 4-LUTs	1 array	8 arrays				16 arrays			
			Alg 1	Alg 2	Alg 3		Alg 1	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 2: Sample CPU Run Times (in seconds) on a 143MHz UltraSparc

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	find the $N$ best seed nodes for $i=1$ to $N$ { connect memory array $i$ and delete all nodes no longer needed due to array $i$ } b) Algorithm 2	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 } c) Algorithm 3
--	--	--

Figure 7: Three Possible Algorithms for Multiple Memory Arrays

then used for one array, and the nodes implemented by all  $N$  arrays are then removed at once. Although this is faster than Algorithm 1 (see Table 2), it gives worse results since which seed node is best may change as nodes are removed. The lower line in Figure 8(a) shows this; for 16 arrays, Algorithm 2 is able to delete only 154 logic blocks (9.6 logic blocks per array).

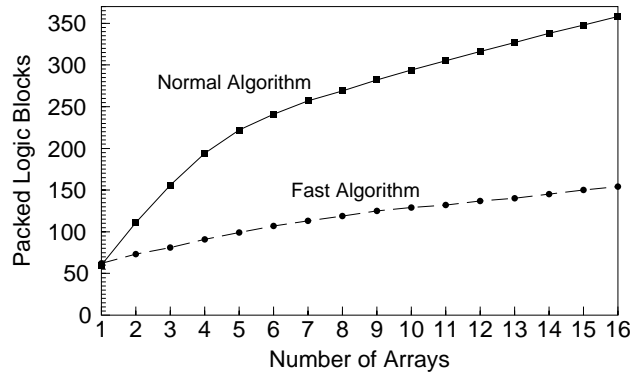
When arrays are used to implement storage, several arrays can be combined to create deeper or wider memories. For example, two Altera EABs can be combined to form a single 4096x1, 2048x2, 1024x4, 512x8, or 256x16 memory. The same can be done when implementing logic in a memory array. In Algorithm 3, 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  $\frac{N}{BF}$  super-arrays. If  $BF = 1$ , this reverts to Algorithm 1. If  $BF$  is large, the CPU requirements are significantly reduced (see Table 2), since only  $\frac{N}{BF}$  seed nodes need be chosen. Figure 8(b) shows the results for Algorithm 3 with various Blocking Factors. Note that as  $BF$  increases, the number of nodes that can be deleted drops, but not nearly as much as it did for Algorithm 2. Thus, we conclude that for large  $N$ , a high blocking factor is appropriate. Numerical results for selected  $N$  and  $BF$  are shown in the right-most six columns of Table 1.

## 6 Conclusions

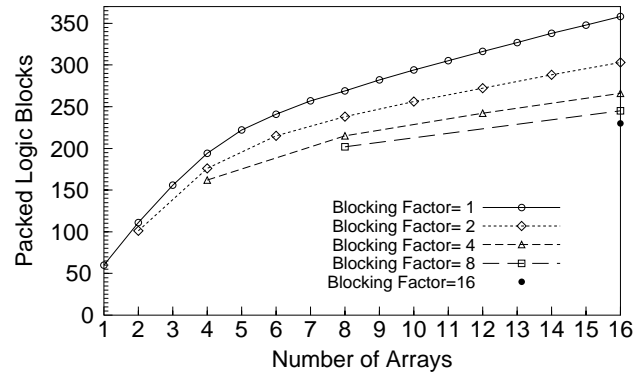
In this paper, we have described a new technology mapping algorithm, SMAP, that maps logic circuits to FPGAs with large embedded memory arrays. Although these embedded arrays were originally conceived to implement the storage parts of circuits, we have shown that if not all arrays are needed to implement storage, the unused arrays need not be wasted. Instead, they can be used to implement logic. Our algorithm packs as much circuit information as possible into the available memory arrays, and maps the rest of the circuit to lookup-tables.

On a set of 29 sequential and combinational benchmarks, the tool is able to map, on average, 60 4-LUTs into a single 2-Kbit memory array. If there are 16 arrays available, it can map, on average, 358 4-LUTs to the 16 arrays. These densities are better than would have been obtained had the FPGA contained nothing but 4-LUTs. 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.

The algorithm presented here has focused on minimizing the area required to implement circuits. It is likely that embedded memories can also be used to shorten the critical path of circuits, thereby raising the achievable clock frequency. The extension of this algorithm to delay reduction is an area of future work.



a) Results from Algorithms 1 and 2



b) Algorithm 3 with Various Blocking Factors

Figure 8: Results for Multiple 2048-bit Arrays.

## References

- [1] Xilinx, Inc., *XC4000 Series (E/L/EX/XL) Field Programmable Gate Arrays v1.04*, September 1996.
- [2] Altera Corporation, *Databook*, June 1996.
- [3] Actel Corporation, *Datasheet: 3200DX Field-Programmable Gate Arrays*, 1995.
- [4] Actel Corporation, *Actel's Reprogrammable SPGAs*, 1996.
- [5] Lattice Semiconductor Corporation, *Datasheet: ispLSI and pLSI 6192 High Density Programmable Logic with Dedicated Memory and Register/Counter Modules*, July 1996.
- [6] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, "Architecture of centralized field-configurable memory," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 97–103, 1995.
- [7] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, "Memory/logic interconnect flexibility in FPGAs with large embedded memory arrays," in *Proceedings of the IEEE 1996 Custom Integrated Circuits Conference*, pp. 144–147, May 1996.
- [8] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, "Memory-to-memory connection structures in FPGAs with embedded memory arrays," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1997.
- [9] T. Ngai, J. Rose, and S. J. E. Wilton, "An SRAM-Programmable field-configurable memory," in *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, pp. 499–502, May 1995.
- [10] 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*, pp. 94–99, June 1996.
- [11] R. Murgai, F. Hirose, and M. Fujita, "Logic synthesis for a single large look-up table," in *Proceedings of the International Workshop on Logic Synthesis*, May 1995.
- [12] J. He and J. Rose, "Technology mapping for heterogeneous FPGAs," in *Proceedings of the ACM International Workshop on Field Programmable Gate Arrays*, Feb 1994.
- [13] 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 *Proceedings of the European Conference on Design Automation*, pp. 534–538, Feb. 1991.
- [14] R. Francis, J. Rose, and Z. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based fpgas," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 227–233, June 1991.
- [15] R. Murgai, Y. Nishizaki, N. Shenoy, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 620–625, June 1990.
- [16] 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.
- [17] M. Ciesielski and S. Yang, "Plade: A two-stage PLA decomposition," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 8, pp. 943–954, 1992.
- [18] J. Cong and Y. Ding, "Combinational logic synthesis for LUT based field programmable gate arrays," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, pp. 145–204, April 1996.
- [19] 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, January 1994.
- [20] E. Sentovich, "SIS: A system for sequential circuit analysis," Tech. Rep. UCB/ERL M92/41, Electronics Research Laboratory, University of California, Berkeley, May 1992.
- [21] S. J. E. Wilton, *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*. PhD thesis, University of Toronto, 1997.