Heterogeneous Technology Mapping for FPGAs with Dual-Port Embedded Memory Arrays

Steven J.E. Wilton
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada, V6T 1Z4
stevew@ece.ubc.ca [*]

Abstract

It has become clear that on-chip storage is an essential component of high-density FPGAs. These arrays were originally intended to implement storage, but recent work has shown that they can also be used to implement logic very efficiently. This previous work has only considered single-port arrays. Many current FPGAs, however, contain dual-port arrays. In this paper we present an algorithm that maps logic to these dual-port arrays. Our algorithm can either optimize area with no regard for circuit speed, or optimize area under the constraint that the combinational depth of the circuit does not increase. Experimental results show that, on average, our algorithm packs between 29% and 35% more logic than an algorithm that targets single-port arrays. We also show, however, that even with this algorithm, dual-port arrays are still not as area-efficient as single-port arrays when implementing logic.

1 Introduction

On-chip storage has become an essential component of high-density FPGAs. The large systems that will be implemented on these FPGAs often require storage; implementing this storage on-chip results in faster clock frequencies and lower system costs.

Most recent FPGAs implement the memory portion of circuits using large embedded memory arrays [1, 2, 3, 4, 5, 6, 7, 8]. These memory arrays result in very efficient memory implementations, since the per-bit overhead is small. Unfortunately, the use of these arrays 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, the chip area devoted to the unused arrays is wasted.

[*]This work was supported by Cypress Semiconductor, the British Columbia Advanced Systems Institute, and the Natural Sciences and Engineering Research Council of Canada.

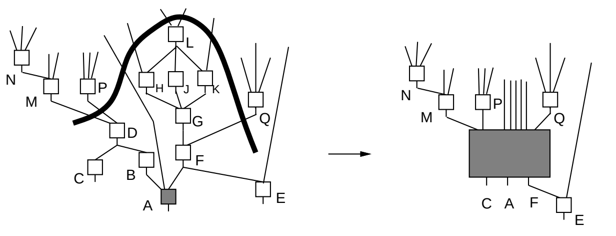


Figure 1: Example Mapping to a 8-Input, 3-Output Memory

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 can very efficiently implement some logic circuits. As an example, consider the circuit in Figure 1(a). Each node represents a four-input lookup table. If a 8-input 3-output memory is available, the implementation in Figure 1(b) is possible, in which 10 of the lookup-tables in the original circuit have been packed into the memory array. In [9] and [10], two algorithms, SMAP and EMB_Pack, were presented that pack as much circuit information as possible into the available memory arrays. In both papers, it was shown that this technique results in extremely dense logic implementations for many circuits. Thus, even customers that do not require storage can benefit from embedded memory arrays.

The algorithms presented in [9] and [10] both target single-port embedded memory arrays (arrays in which only one access can be performed at a time). Many recent FPGAs, however, contain dual-port arrays (so that two accesses can be performed in each array concurrently) [2, 4, 7]. In this paper, we present a heterogeneous technology mapping algorithm which targets FPGAs with dual-port embedded arrays. Although we can simply use SMAP or EMB_Pack, and leave one port of each array unused, this paper shows that significant density improvements can be obtained by taking into account the dual-port nature of the arrays.

The algorithm presented here partitions each memory array into two sub-arrays, each of which is filled with logic, and accessed through one of the array's two ports. Since there are many ways to partition each array into sub-arrays, and since it is impossible to know which partition will lead to the best results before filling each sub-array with logic, we have developed a *simultaneous partitioning/packing algorithm* which partitions each array at the same time as packing logic into each sub-array. This algorithm is novel, and is a key contribution of this paper.

In addition to presenting the new algorithm, this paper answers two questions:

1. Given an FPGA with dual-port arrays, how much better does the proposed algorithm perform than an algorithm originally developed to target single-port arrays?

2. Which is more area-efficient when implementing logic: dual-port arrays or single-port arrays?

We answer these questions in two contexts: one in which the goal is simply to minimize the area required to implement a circuit without regard for circuit speed, and one in which the goal is to minimize the circuit area with the constraint that the combinational depth of the circuit does not increase (the former was the goal of SMAP [9], while the latter was the goal of EMB_Pack [10]).

This paper is organized as follows. Section 2 outlines the terminology used and the architectural assumptions made in this paper. Section 3 then describes the area minimization algorithm, and Section 4 describes the area minimization with depth constraint algorithm. Section 5 then evaluates the algorithms and answers the first question listed above. Finally, Section 6 compares single- and dual-port arrays, and answers the second question listed above.

2 Preliminaries

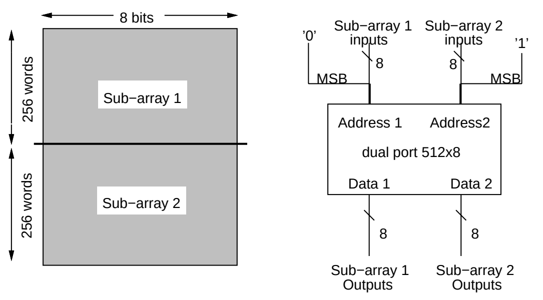
2.1 Terminology

In this paper, we will use the following terminology (primarily from [11]). 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, $depth(v)$, is the maximum path length from any primary input to node v. The depth of a graph $depth(G)$ is the maximum $depth(v)$ 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. Given a set of nodes W, a *fanin-subnetwork* rooted at W is a subnetwork containing each node in W along with one or more 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. Similarly, a *fanout-free subnetwork* is a fanin-subnetwork in which no node in the subnetwork (except the root nodes) drives a node not in the fanin-subnetwork. The *maximum-fanout free cone (MFFC)* for a node v is the fanout-free cone rooted at v containing the largest number of nodes. The *maximum-fanout free subnetwork (MFFS)* for a set of nodes W is the largest fanout-free subnetwork rooted at W. Given a cone or fanin-subnetwork 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 or fanin-subnetwork 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'.

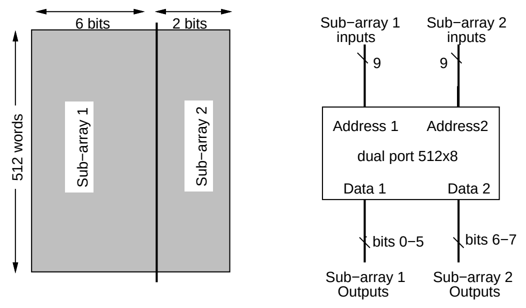
2.2 Architectural Assumptions

We assume an FPGA consisting of many 4-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 is fixed, a wider word means there are fewer words in the array). 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.

In this paper, we assume each array contains two independent access ports. Note that some FPGA architectures, such as the Altera FLEX10KE, contain two independent ports, but one port is a dedicated read port and one port is a dedicated write port. This works well for many applications (such as a first-in first-out buffer that is used to temporarily hold data in a communication system), but there are many applications for which this is insufficient (a dual-port register file in a processor which must be read by two functional units simultaneously, for example). To implement these sorts of circuits, true dual-port memory arrays are required, in which the two accesses are independent, and either can be a read or write. With the increasing importance of embedded memory in FPGAs, and since true dual-port arrays appear to be a natural evolution from the restricted dual-port model used in many architectures today, we feel that true dual-port memories will be available in future devices. Thus, we focus our efforts on studying algorithms that target true dual-port memories. In particular, the algorithms presented in this paper assumes both ports can be used as read ports.



a) Horizontal Partitioning



b) Vertical Partitioning

Figure 2: Partitioning a 512x8 dual-port array

3 Algorithm Description: Area Minimization Algorithm

In this section, we consider strict area minimization. That is, the goal is to pack as much logic into the available arrays as possible (since the more logic that is implemented in the arrays, the less that must be implemented using the logic resources of the FPGA), without regard for the speed of the final circuit.

As described in the introduction, our approach is to logically partition each array into two sub-arrays, each of which can be filled with logic, and accessed through one of the array's two ports. Since there are many ways to partition each array, and since it is impossible to know which partition will lead to the best results before filling each sub-array with logic, we use a simultaneous partitioning/packing algorithm to perform these tasks together.

The following subsection discusses the partitioning of the arrays. Subsection 3.2 then describes the new simultaneous partitioning/packing algorithm. The discussion assumes only a single memory array with a fixed data width is available; in Subsection 3.3, we show how the algorithms can be extended to handle multiple arrays, each with several possible aspect ratios.

3.1 Partitioning Arrays

Each array can be partitioned in one of two ways: *horizontally* and *vertically*, as shown in Figure 2. In horizontal partitioning, two sub-arrays are created, each containing exactly half of the words in the original array. In the example of Figure 2(a), a dual-port 512x8 memory is divided into two 256x8 memories. Since the memory is dual-port, there are two address ports. One of the address bits in Port 1 is tied to 0, while the same address bit in Port 2 is tied to 1. This ensures that addresses supplied to the remaining 8 address bits of Port 1 map to words 0 through 255 of the memory, while the address supplied to the remaining 8 address bits of Port 2 map to words 256 to 512 of the memory. Thus, each address port sees an independent 256x8 memory space.

Figure 2(b) shows an example of vertical partitioning. In this case, In this case, the 512x8 dual-port memory is divided into one 512x6 memory and one 512x2 memory. All 9 bits of each address port are used, but only bits 0 to 5 of Data Port 1 and bits 6 to 7 of Data Port 2 are used. Again, the two sub-arrays are independent.

Note that with horizontal partitioning, it only makes sense to divide the array exactly in half. If an array is cut anywhere other than its midpoint, one of the sub-arrays will have a depth that is not a power-of-two; such a sub-array will not be able to implement all functions of its inputs, a property that the packing algorithm described below relies upon. On the other hand, with vertical partitioning, we do not require each array to be the same size. If the original array is w bits wide, there are w-1 ways to partition the array vertically.

Since there is only one way to horizontally partition an array, the partitioning and packing can be done as separate steps if only horizontal partitioning is considered. An algorithm that employs vertical partitioning, on the other hand, must perform the partitioning and packing simultaneously, since it is impossible to know before-hand which partitioning will lead to the best packing. In the algorithm described below, we focus on vertical partitioning, since it is more flexible; experiments have shown that in all but a few cases, vertical partitioning gives better packing results.

3.2 Simultaneous Partitioning/Packing Algorithm

The algorithm presented in this section simultaneously divides the physical array into two sub-arrays, and packs each sub-array with logic.

The algorithm is outlined in Figure 3. It iterates over each node in the network, considering each as a possible seed node. For a given seed node v, the algorithm first finds the maximum-volume n-feasible cut of the cone rooted at v, where n is the number of inputs to the array. Figure 1(a) shows an example circuit along with the the maximum 8-feasible cut for seed node A. Note that the selection of input nodes is independent of the partitioning of the array, since, with vertical partitioning, the number of input signals does not depend on the partition chosen.

Given a seed node v and a cut, the algorithm then determines the set P_v of all nodes which can be expressed entirely

```
output_selection($P_v$)
    $r_v = \emptyset$
    for each node $v'$ in $P_v$
        MFFCsize[$v'$] = |MFFC($v'$)|
    for $i = 1$ to $w$
        $r_{v,i}$ = $r_{v,i-1} \cup$ node with next largest MFFC
    return $r_v$

top_level:
    best = $\emptyset$
    $s_i$ = $\emptyset$ for $1 \leq i \leq w$
    for each seed $v$
        cutset$_v$ = max-vol $n$-feasible cut
        $P_v$ = set of Potential Nodes driven by cutset$_v$
        $r_v$ = output_selection($P_v$)
        update_solution(best,$s$,$r_v$)
    return best

update_solution(best,$s$,$r_v$)
    for $i = 1$ to $w - 1$
        temp = $r_{v,w-i}$
        if (|MFFS(temp)|+|MFFS($s_i$)|>|MFFS(best)|)
            best = temp $\cup s_i$
    for $i = 1$ to $w$
        if (|MFFS($r_{v,i}$)|>|MFFS($s_i$)|)
            $s_i = r_{v,i}$
```

Figure 3: Area Minimization algorithm for one array with n inputs and w outputs

as a function of the cut-nodes. From this set, the algorithm then selects which nodes will become the memory array outputs. The selection of the outputs is an optimization problem, since different combination of outputs will lead to different numbers of nodes that can be packed into the arrays. In selecting the outputs, the algorithm constructs a set of possible solutions r_v (this is different than SMAP, which constructs only a single solution for each seed node v). Each element $r_{v,i}$ (for $1 \leq i \leq w$) is the set of output nodes that would be chosen if the array had i outputs (the actual number of outputs is not yet known, because the partition has not yet been chosen). Each element $r_{v,i}$ contains the i nodes in P_v with the largest MFFC's.

As the algorithm iterates over each seed node v, it maintains two variables: *best* and s. The variable s is a vector containing the best solutions for each possible output width seen so far. Each element, s_i (for $1 \leq i \leq w$), contains the best single-port mapping if the array has i outputs. The best mapping is the one in which the selected outputs has the largest MFFS. More precisely, if V is the set of seed nodes visited so far,

$$s_i = r_{v',i} \text{ such that } \mid \text{MFFS}(r_{v',i}) \mid = \max_{v'' \varepsilon V} \mid \text{MFFS}(r_{v'',i}) \mid$$

The variable *best* contains the single best combination of two single-port mappings seen so far, such that the mappings can mapped to a single dual-port array partitioned vertically. If the array has w outputs, this means the sum of the output widths of the two components is at most w. More precisely,

$$\textit{best} = s_i \cup s_j \text{ such that } \mid \text{MFFS}(s_i) \mid + \mid \text{MFFS}(s_j) \mid = \max_{\forall i,j: i+j \leq w} (\mid \text{MFFS}(s_i) \mid + \mid \text{MFFS}(s_j) \mid)$$

Figure 3 shows how these these quantities can be computed incrementally. After visiting all nodes, the variable *best* contains the final dual-port mapping solution.

Note that this algorithm is not guaranteed to find the optimum solution, since it is possible that the MFFCs of one selected output node may contain another selected output node. However, experiments have shown that this heuristic works well.

3.3 Multiple Arrays and Aspect Ratios

In most FPGAs, there are several memory arrays, and each can be used in one of several aspect ratios. If there is more than one array available, we map to each array separately. Once the first array has been filled, those logic blocks that can be packed into that array are removed from the circuit, and the process is repeated with the remaining network. This is done until logic has been packed into all available arrays.

If there is more than one aspect ratio in which each array can be used, we first choose the best seed node using the algorithm from Figure 3 assuming the widest available word width. We then test that seed node using all narrower array configurations, and choose the configuration that results in the best overall mapping. This is the same technique used in SMAP [9].

4 Algorithm Description: Area Minimization with Depth Constraint

In this section, we describe a modification to the algorithm presented above that solves the area minimization with depth constraint problem. This 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 lookup tables. 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.

```
output_selection($P_v$, $cutset$)
    $r_v = \emptyset$
    for each node $v'$ in $P_v$
        MFFCsize[$v'$] = |MFFC($v'$)|
    for $i = 1$ to $w$
        $r_{v,i}$ = $r_{v,i-1}$ $\cup$ node with next largest MFFC
              such that $depth(v) + slack(v) \geq$
                        $h_m + \max_{x \in cutset}(depth(x))$
    return $r_v$

top_level:
    best = $\emptyset$
    $s_i$ = $\emptyset$ for $1 \leq i \leq w$
    calculate $depth(v)$ and $slack(v)$ for each node $v$
    for each seed $v$
        cutset$_v$ = max-vol $n$-feasible cut
        $P_v$ = set of Potential Nodes driven by cutset$_v$
        $r_v$ = output_selection($P_v$,cutset$_v$)
        update_solution(best,$s$,$r_v$)
    return best

update_solution(best,$s$,$r_v$)
    for $i = 1$ to $w - 1$
        temp = $r_{v,w-i}$
        if (|MFFS(temp)|+|MFFS($s_i$)|$>$|MFFS(best)|)
            best = temp $\cup s_i$
    for $i = 1$ to $w$
        if (|MFFS($r_{v,i}$)|$>$|MFFS($s_i$)|)
            $s_i = r_{v,i}$
```

Figure 4: Area Minimization with Depth Constraint Algorithm for one array with n inputs and w outputs

Circuit Name	Number of 4-LUTS	SMAP	New Algorithm
i10	994	8.0	24.0
apex4	1262	20.9	56.4
diffeq	1494	12.6	35.0
s38584	6211	127	367
iir16	3612	70	180

Table 3: Run times (seconds) on a 250MHz UltraSparc

Figure 4 outlines this algorithm. The first step is to compute the *slack* at each node. 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 [12].

Once the slacks have been computed, the algorithm proceeds as in the area minimization algorithm. The only difference is in the selection of the array outputs (the set r_v). In the area minimization algorithm, each node in P_v is a potential output. Here, however, only the nodes in P_v that satisfy the following inequality are potential outputs:

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

where I is the set of nodes in the cut set (the memory array inputs). Intuitively, v is only considered as an output node if the minimum depth between v and any node in the cut set (the memory array inputs) is not less than h_m.

5 Results: Packing into Dual-Port Arrays

In this section, we evaluate how well the proposed algorithms are able to pack logic into dual-port arrays. To perform this evaluation, we used 27 large benchmark circuits, each containing between 527 and 6598 4-LUTs. Sixteen 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 [13] and technology-mapped to 4-input lookup tables using Flowmap and Flowpack [14].

5.1 Area Minimization

We first consider strict area minimization. The fourth and fifth columns of Table 1 show the number of 4-LUTs that can be packed into a single memory (and hence not implemented using the logic resources of the FPGA) using a previous algorithm, SMAP, and the new algorithm from Section 3. The previous algorithm, SMAP, targets single-port arrays; here, we use this algorithm and leave one port of each array unused. In both cases, a dual-port 2Kbit array that can take on a word width of 1, 2, 4, or 8 was assumed.

As the table shows, the new algorithm performs 35% better than the original SMAP algorithm (35% more 4-LUTs can be packed into the same memory array). The geometric average of the results is 54% higher for the new algorithm than the SMAP algorithm. The final two columns of the table shows the results if eight memory arrays are available. In this case, a 32% improvement is obtained.

Notice two anomolies: for the circuits *ex5p* and *apex4*, the new algorithm performs worse than SMAP. This highlights the fact that both algorithms are heuristics, and neither is guaranteed to find the optimum solution. As described in Section 3.1, the algorithm attempts to find the solution such that each memory array output has the largest MFFC. In some

Circuit Name	Original Circuit		One Array		Eight Arrays	
	Number of 4-LUTS	Number of Flip Flops	SMAP	New Algorithm	SMAP	New Algorithm
pair	641	0	13	24	81	147
apex1	696	0	14	21	87	124
cps	749	0	46	52	173	224
C5315	596	0	12	23	76	140
C6288	527	0	19	29	93	121
apex3	867	0	26	32	119	166
C7552	679	0	15	33	94	178
i10	994	0	18	32	91	160
ex5p	1064	0	198	200	1043	579
spla	3690	0	67	116	349	590
pdc	4575	0	88	192	480	903
apex4	1262	0	319	324	1261	1225
tseng	1046	385	10	20	69	127
bigkey	1707	224	18	23	75	116
s38417	6096	1463	26	55	193	346
diffeq	1494	377	22	39	120	213
frisc	3539	886	62	73	118	189
dsip	1370	224	18	22	60	105
s5378	572	160	16	25	84	139
s38584	6211	1260	64	84	241	384
iir16	3612	522	37	46	164	249
fir16	6598	847	84	117	250	371
ralu32	3659	590	20	36	118	197
spsdes	3356	949	26	33	91	135
mac64	4307	64	15	32	85	189
mips64	2226	438	10	19	66	135
sort8	1861	184	24	33	82	148
avg.			47.7	64.3	213.4	281.5
geo.avg.			29.1	44.7	138.6	217.7

Table 1: Results assuming 2048-bit memory arrays: Area Minimization

cases, this does not give the overall optimum solution.

The execution time of each algorithm for five of the benchmark circuits is shown in Table 3. The execution time of the new algorithm is roughly three times that of SMAP. Although this is a significant increase in execution time, these times are still considerably smaller than the times required for technology mapping, placement and routing of the same circuits.

5.2 Area Minimization with Depth Constraint

Section 4 showed how the algorithms can be extended to minimize area under the constraint that the combinational depth of the circuit does not increase. Table 2 shows the number of 4-LUTs that can be packed for the two algorithms under this constraint. Geometric averages are not shown because of the 0 entries. In gathering these results, it was assumed that the read time of a dual-port memory is 5 times the propagation time of a 4-LUT ($h_m = 5$); this value was obtained from a delay model.

As the table shows, the new algorithm gives a 32% improvement if there is only one array available, and a 29% improvement if there are eight arrays available.

Note that the use of a depth constraint to constrain delay is only an approximation. In [15] and [10], results from the original SMAP and EMB_Pack were fed into MAX+plusII, and their delays measured. Although there were large variations, in general, it was found that constraining depth in this way did, on average, keep the critical path of the circuit from increasing. We have not yet performed similar experiments for dual-port arrays.

6 Results: Comparison of Single- and Dual-Port Arrays

The previous section showed that, if an FPGA contains dual-port arrays, the proposed algorithm uses these arrays more effectively than SMAP does. In this section, we answer a related, but different question: If the arrays are to be used to implement logic, are single- or dual-port arrays more efficient? In our comparisons, we use SMAP to map to single-

Circuit	Original Circuit		One Array		Eight Arrays	
Name	Number of 4-LUTS	Number of Flip Flops	SMAP	New Algorithm	SMAP	New Algorithm
pair	641	0	13	24	58	88
apex1	696	0	10	16	49	70
cps	749	0	19	21	25	25
C5315	596	0	10	17	56	136
C6288	527	0	14	26	73	91
apex3	867	0	22	24	81	93
C7552	679	0	11	24	62	116
i10	994	0	17	30	92	138
ex5p	1064	0	198	200	1036	1036
spla	3690	0	67	103	260	298
pdc	4575	0	88	178	473	655
apex4	1262	0	319	319	1261	1261
tseng	1046	385	10	19	64	109
bigkey	1707	224	0	0	0	0
s38417	6096	1463	26	51	189	330
diffeq	1494	377	15	28	94	182
frisc	3539	886	8	23	59	122
dsip	1370	224	0	0	0	0
s5378	572	160	10	14	59	90
s38584	6211	1260	64	84	241	382
iir16	3612	522	37	46	164	249
fir16	6598	847	84	117	225	314
ralu32	3659	590	20	36	113	174
spsdes	3356	949	26	33	88	132
mac64	4307	64	10	17	56	108
mips64	2226	438	8	17	63	126
sort8	1861	184	24	28	63	95
avg.			41.9	55.4	185	238

Table 2: Results assuming 2048-bit memory arrays: Area Minimization with Depth Constraint

port arrays and the new algorithm from this paper to map to dual-port arrays.

First consider strict area minimization. In this case, Table 1 shows that 35% more logic can be packed in a 2048-bit dual-port array, using the new algorithm, than a 2048-bit single-port array, using SMAP. However, we must balance this with the fact that a dual-port array consumes more chip area than a single-port array. Using a detailed area model, which includes terms for decoders, the core array, multiplexors, and sense amplifiers, we have estimated that a 2048-bit dual-port array requires 5478 bit-areas (one bit-area is the area required to implement one single-port SRAM bit) while the area required to implement a 2048-bit single-port array requires only 3229 bit-areas. The ratio of logic blocks packed per chip area for a single port array is 0.0148, while the same ratio for a dual-port array is 0.0117. Thus, the single-port memory array can implement logic 21% more efficiently than a dual-port array. If eight arrays are available, the difference is 22%. These numbers and calculations are summarized in Table 4.

Second, consider area minimization with the constraint that the combinational depth of the circuit does not increase. In this case, we must also take into account the fact that, in addition to being larger, a dual-port array is also slower than a single-port array. To be fair, we have compared results from SMAP targeting a single-port array with $h_m = 3$ to results from the new algorithm targeting a dual-port array with $h_m = 5$ (again, these numbers were obtained from detailed delay models). Table 5 shows the results; the single-port memory array can implement logic 31% more efficiently than a dual-port array. If eight arrays are available, the difference is 34%.

Thus, in both cases we conclude that dual-port arrays are not as efficient as single-port arrays when used to implement logic. This does not mean dual-port arrays are a bad idea: the primary motivation for including dual-port arrays is to efficiently implement dual-port storage requirements of circuits, such as FIFOs. Dual-port storage is vital for many applications. Thus, we still expect to see dual-port arrays in future FPGAs.

	One Array Available		Eight Arrays Available	
	Single Port	Dual Port	Single Port	Dual Port
Avg. logic blocks packed (a)	47.7	64.3	213.4	281.5
Area in bit-areas (b)	3229	5478	25832	43826
Ratio (a)/(b)	0.0148	0.0117	0.00826	0.00642

Table 4: Architecture Comparison for Area-Minimization

	One Array Available		Eight Arrays Available	
	Single Port	Dual Port	Single Port	Dual Port
Avg. logic blocks packed (a)	47.2	55.4	211	238
Area in bit-areas (b)	3229	5478	25832	43826
Ratio (a)/(b)	0.0146	0.0101	0.00819	0.00543

Table 5: Architecture Comparison for Area-Minimization with Depth Constraint

7 Conclusions

In this paper, we have presented a heterogeneous technology mapping algorithm that maps logic to dual-port embedded FPGA memory arrays. Two versions of the algorithm were presented: one that optimized circuit area with no regard for circuit speed, and one that optimizes circuit area under the constraint that the combinational depth of the circuit is no larger than that if the circuit was implemented entirely with 4-LUTs.

Experimental results have shown that, given an FPGA with dual-port arrays, our new algorithm packs between 29% and 35% more logic than the previous algorithm. Since many current (and future) FPGAs contain dual-port arrays, this is an important result.

However, we have also shown that, even with this new algorithm, dual-port arrays are not as area-efficient as single-port arrays when used to implement logic. This is because the dual-port arrays are larger and slower than their single-port counterparts.

References

[1] Altera Corporation, *FLEX 10K Embedded Programmable Logic Family Data Sheet, ver. 4.01*, June 1999.

[2] Altera Corporation, *FLEX 10KE Embedded Programmable Logic Family Data Sheet, ver. 2.01*, June 1999.

[3] Altera Corporation, *APEX 20K Programmable Logic Device Family Data Sheet, ver. 2.0*, May 1999.

[4] Xilinx, Inc., *Virtex 2.5 V Field Programmable Gate Arrays, ver. 1.6*, July 1999.

[5] Lattice Semiconductor, *Vantis VF1 FPGA Data Sheet*, November 1998.

[6] Actel Corporation, *Data sheet: ProASIC 500K Family*, June 1999.

[7] Actel Corporation, *MX FPGA Data Sheet*, January 1999.

[8] Actel Corporation, *Datasheet: Integrator Series FPGAs: 1200XL and 3200DX Families*, January 1998.

[9] S. J. E. Wilton, "SMAP: heterogeneous technology mapping for FPGAs with embedded memory arrays," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 171–178, February 1998.

[10] J. Cong and S. Xu, "Technology mapping for FPGAs with embedded memory blocks," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 179–187, February 1998.

[11] 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.

[12] R. Hitchcock, G. Smith, and D. Cheng, "Timing analysis of computer hardware," *IBM Journal of Research and Development*, pp. 100–105, January 1983.

[13] E. Sentovich, "SIS: A system for sequential circuit analysis," Tech. Rep. UCB/ERL M92/41, Electronics Research Laboratory, University of California, Berkeley, May 1992.

[14] 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.

[15] S. J. E. Wilton, "Heterogeneous technology mapping for area reduction in fpgas with embedded memory arrays," *to appear in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2000.