Storage Assignment during High-level Synthesis for Configurable Architectures

Wenrui Gong Gang Wang Ryan Kastner
Department of Electrical and Computer Engineering
University of California, Santa Barbara
Santa Barbara, CA 93106-9560 USA
Email: {gong, wanggang, kastner}@ece.ucsb.edu

***Abstract*— Modern, high performance configurable architectures integrate on-chip, distributed block RAM modules to provide ample data storage. Synthesizing applications to these complex systems requires an effective and efficient approach to conduct data partitioning and storage assignment. In this paper, we present a data and iteration space partitioning solution that focuses on minimizing remote memory accesses or, equivalently, maximizing the local computation. Using the same code but different data partitionings, we can achieve faster clock frequencies, without increasing the number of cycles, by simply minimizing global memory accesses. Other optimization techniques like scalar replacement, prefetching and buffer insertion can further minimize remote accesses and lead to average 4.8x speedup in overall runtime.**

I. Introduction

Typical configurable computing systems consist of arrays of reconfigurable logic blocks and flexible interconnect. In order to offer greater computing capabilities, high-performance commercial configurable architectures usually incorporate a number of distributed block RAM modules. For instance, the Xilinx Virtex-II Pro FPGA series provides up to 105K logic cells, and 1,456 kilobytes of distributed, embedded block RAM.

These configurable architectures, integrated with ample distributed block RAM modules, exhibit superior computing abilities, storage capacities, and flexibilities over traditional FPGAs. However, they currently lack the tools necessary to provide the application designer efficient synthesis onto these complex architectures. In particular, there is a pressing need for memory optimization techniques in the early stages of the design flow as modern configurable architectures have a complex memory hierarchy, and earlier architectural-level decisions greatly affect the final design qualities.

In traditional design flow of configurable devices, synthesis of block RAM modules are generally handled as a physical problem. They are directly inferred from arrays, or instantiated using vendor macros. They are packed even in placement, and only partitioned when it is difficult to fix the device. In most situations, the memory bandwidth and storage capacities are not efficiently utilized, and hence the generated designs are inefficacious in terms of latencies, throughput, and achieved frequencies.

This paper focuses on seeking a partitioning-based solution to the storage assignment problem at an very early stage of the design flow, and other memory optimization techniques helpful to achieve the design goals, such as latencies and frequencies.

The central contribution of this paper is a novel integrated approach of deriving an appropriate data partitioning, and synthesizing the program behavior to configurable devices. Through intensive research on the interplay between the data partitions and architectural synthesis decisions, such as scheduling and binding, we show that designs that minimize the number of global memory accesses and exhibit local computation can meet the design goals, and minimize the execution time (or maximize the system throughput) under resource constraints. Other optimization techniques, including scalar replacement, and data prefetching and buffer insertion, are applied to improve the overall performance. In particular, these optimizations further reduce latencies, and improve the achieve clock frequencies.

This work is organized as follows. The next section gives details on the target configurable architecture and the following section presents a motivating example. Section IV discusses related work. Section V formally describes the data partitioning and storage assignment problem and provides techniques to minimize the number of remote data memory accesses. Section VI presents our experimental results and we conclude in Section VII.

II. Target Configurable Architecture

Modern reconfigurable architectures incorporate a number of distributed block memories. These architectures can be divided into homogeneous and heterogeneous architectures according to the capacities and distribution of the RAM blocks. The *heterogeneous* architecture contains a variety of block RAM modules with different capacities. Figure 1 presents an example of a *homogeneous* architecture. This roughly corresponds to Xilinx Virtex II FPGA [1]. The block RAMs are evenly distributed on the chip and connected with CLBs using reprogrammable interconnect. Every block RAM has the same capacity. Additionally, there is an embedded multiplier located beside each block RAM.

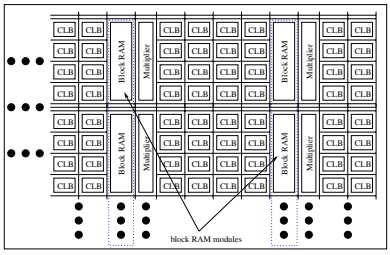


Fig. 1. FPGA with distributed Block RAMs

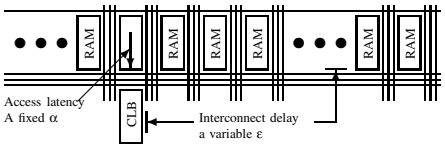


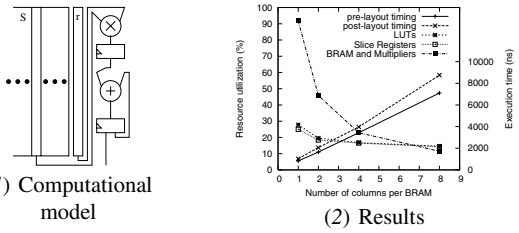
Fig. 2. Total access latencies = $\alpha + \varepsilon$

Access latencies of the on-chip block RAM equals to the propagation delay to the memory port after the positive edge of the `clock` signal. This delay is usually a fixed number α for a specific FPGA architecture. For example, α is 3.7 ns for Xilinx XC2V3000 FPGA. And it takes an extra ε ns to transfer data from the memory port to the accessing CLB. Hence, as to a design running at 200MHz, it could take one clock cycle to access *near* data, or two or more cycles to access data far away from the CLB. On the other hand, it is difficult to distinguish which ones are near and which ones are far.

III. Motivating Example: Correlation

The bank of correlators multiplies each sample of the received vector r with the corresponding sample of a column in an S matrix, i.e. $C_i = \sum_{j=1}^{l} r_j \times S_{j,i}$, where r is a vector of l complex numbers, and S is a $m \times l$ real numbers. l and m will vary based on the application. For instance, if we wish to perform radiolocation in the ISM band (802.11x) using the matching pursuit algorithm, both l and m are equal to 88. If the S matrix is kept packed, the most advanced commercial high-level synthesis tool either generates a design with an extremely slow execution time of about 77,440 ns, or fails to synthesize this design due to the huge S matrix. Obviously the partition of the S matrix to the block RAM modules greatly affects the system overall performance.

The data space can be partitioned by column or by row, or by other means. By simple analysis, column-wise partition achieves a communication-free partitioning. Figure 3(1) presents the control and computations of the column-wise data partitioning. Computations of each correlator are conducted using embedded multipliers beside the block RAM in a multiplication and accumulation (MAC) manner. For each correlator, the control logic and computational resources are local to the block RAM module.



(*1*) Computational model

(*2*) Results

Fig. 3. Implementations and area/timing trade-offs

Figure 3(2) presented area and timing trends of different granularity for the column-wise scheme. As shown in Figure 3(2), when assigning one block RAM to one column, the design takes the shortest execution time, but requires the greatest hardware resources. When more columns are packed into one block RAM, the requirements on hardware decreased. However, the execution time increases linearly to the number of columns in one block RAM.

TABLE I
COMPARISON BETWEEN THE SAME GRANULARITY

Data per BRAM	# of cycles	Pre-layout Timing		Post-layout Timing	
		F(MHz)	L(ns)	F(MHz)	L(ns)
1 column	178	214.7	829	171.6	1037
1 row	184	140.5	1309	133.5	1378
4 columns	706	205.0	3436	178.2	3961
4 rows	710	157.0	4520	129.4	5486
8 columns	1410	198.6	7099	161	8752
8 rows	1413	147.1	9602	138.7	10183

When comparing the two partitioning schemes with the same granularity (i.e. same number of rows/columns), as shown in Table I, we found, in the term of numbers of clock cycles, the differences are very small. However, if we check the maximal achieved frequencies, designs of the column-wise partitioning scheme are 30-50% faster than those of the row-wise partitioning scheme. Deeper analysis showed that the performance gaps are mainly due to the increased amount of global communications needed for the control logic and global memory accesses to block RAM modules.

In summary, different partitions of the array S deliver a wide variety of candidate solutions. Synthesized designs showed that data partitioning not only affect the number of clock cycles, but also affect the achieved clock frequencies.

IV. Related Work

High-level synthesis could dramatically reduce the design time and derive high performance designs. There were also different approaches proposed to synthesize memory modules. Early efforts usually map all data arrays into a single memory modules [2]. And Thomas [23] mentioned assigning each data array a memory module. Comprehensive storage exploration and memory optimizations technologies are presented in IMEC's DTSE work [3]. Panda et al. [4] investigated architectural-level exploration techniques for embedded processors with complicate hierarchical memory systems. Based on the PICO method, Kurdur et al. [5] presented an ILP formulation to solve the storage arrangement problem. Early efforts on utilizing multiple memory modules on FPGA [6] allocated an entire array to a single memory module rather than partitioning data arrays. In most of the above work, they assumed the memory module is capable enough for data arrays and didn't consider memory capacity constraints.

Huang et al. [7] presented their work in high-level synthesis with integrated data partitioning for ASIC design flow. Their work are quite similar to our work in adapting similar code analysis techniques from traditional parallelizing compilation field. However, their work are not limited by the capacities of available memory modules. They started from a fixed number of partitioning. Our proposed work start from the program cores and the resource constraints, and use granularity adjustment to find out the reasonable number of partitions.

As to memory optimization in architectual-level synthesis, Budiu et al. [8], and Diniz et al. [9], respectively, presented effective techniques to reduce memory accesses and benefit high-level synthesis.

Data partitioning and storage assignment problem was well studied in the field of parallelizing compilation [10]. Early efforts developed effective analysis techniques and program transformations to reduce global communications and hence improve system performance. Shih and Sheu [11], and Ramanujam and Sadayappan [12] addressed the methodology to achieve communication-free iteration space and data partitioning problem. Pande [13] presented an communication-efficient data partitioning solution when it is impossible to get a communication-free partitioning.

However, it is impossible to directly migrate these approaches since we are facing different target architectures and unable to estimate the precise timing and area results before logic mapping and physical synthesis. More important, data partition and storage assignment have more compound effects on system performance.

V. Data Partitioning and Storage Assignment

This section formally describes the data partitioning and storage assignment problem, and proposes an approach to computing the number of memory accesses for a given partition. Then, we discuss some of the techniques that we use to reduce memory accesses and improve system performance for FPGA-based configurable architectures with distributed block RAM modules.

A. Problem formulation

We focus on data-intensive DSP applications. These applications usually contain nested loops and multiple data arrays. In order to simplify our problem, we assume that *a*) the input programs are perfectly nested loops; *b*) index expressions of array references are affine functions of loop indices; *c*) there is no indirect array references, or other similar pointer operations; *d*) all data arrays are assigned to block RAM modules; and *e*) each data element is assigned one and only one single block RAM modules, i.e. no duplicate data. Furthermore, we assume that all data types are fixed-point numbers due to the current capability of our system compiler.

The inputs are as follows: *a*) A program d contains an l-level perfectly nested loop $\mathbf{L}$, and a set of n data arrays $\mathbf{N}$; *b*) A specific

Algorithm 1 Partitioning

Input: nested loop $\mathbf{L}$, data arrays $\mathbf{N}$, RAM modules $\mathbf{M}$, and the number of CLBs A
Output: data partitioning $\mathbf{P}$, and iteration partitioning $\mathbf{I_P}$, represented by the direction d and granularity g.
Ensure: $\bigcup_{i=1}^{p} P_i = \mathbf{N}$, and $P_i \bigcap P_j = \emptyset$
Ensure: $|\mathbf{P}| \leq |\mathbf{M}|$
1: **procedure** PARTITIONING
2: Calculate the iteration space $IS(\mathbf{L})$
3: **for** each $N_i \in \mathbf{N}$ calculate the data space $DS(N_i)$
4: $B \leftarrow$ Innermost iteration body
5: Calculate the reference footprints, F, for B using reference functions
6: Analyze $IS(\mathbf{L})$ and F, and obtain a set of partitioning direction $\mathbf{D}$
7: $a \leftarrow A/|\mathbf{M}|$ ▷ # of CLBs associated to each RAM
8: Synthesis$(B, 1, 1, a, u_{ram}, u_{mul}, u_a, T, II)$
9: $g_{min} \leftarrow$ size of $IS(\mathbf{L})/|\mathbf{M}|$ ▷ the finest partition
10: $g_{max} \leftarrow \frac{\text{size of } \sum DS(N_i)}{\text{size of each block RAM}}$ ▷ the coarsest partition
11: $d_{cur} \leftarrow d_0$, $g_{cur} \leftarrow g_{min}$
12: $C_{cur} \leftarrow \infty$
13: **for** each $d_i \in \mathbf{D}$ **do**
14: **for** $g_j \leftarrow g_{min}, g_{max}$ **do**
15: Partition $DS(\mathbf{N})$ following d_i and g_j
16: Estimate the number of memory accesses using reference functions $m_r =\leftarrow$ # of remote accesses, $m_t =\leftarrow$ # of total accesses, $\tau = 2^{\frac{m_r}{m_t}}$ ▷ the choice of 2 depends on the chip size
17: $C \leftarrow \tau \times (\max\{u_r, u_m, u_a\} \times II \times g_j + (T))$
18: **if** $C < C_{cur}$ **then**
19: $d_{cur} \leftarrow d_i$, $g_{cur} \leftarrow g_j$
20: $C_{cur} \leftarrow C$
21: Output d_{cur} and g_{cur}

Algorithm 2 Synthesis

1: **procedure** SYNTHESIS(B, b, m, a, u_r, u_m, u_a, T, II)
2: Generate DFG g from B
3: Schedule and pipeline g to minimize the initial interval, subject to allocated resources, including r block RAM, m multipliers, and a CLBs.
4: Output resource utilization u_r, u_m, and u_a.
5: Output execution time T, and the initial interval II

target architecture, i.e. an FPGA, contains a set of m block RAM modules $\mathbf{M}$. and A CLBs; and c) We set our desired frequency to F, and the maximum execution time to L.

The problem of data partitioning and storage assignment is to partition $\mathbf{N}$ into a set of p data portions $\mathbf{P}$, and seek an assignment $\{\mathbf{P} \rightarrow \mathbf{M}\}$ subject to the following constraints a) $\bigcup_{i=1}^{p} P_i = \mathbf{N}$, and $P_i \bigcap P_j = \emptyset$, i.e. that all data arrays are assigned to block RAM and each data element is assigned to one and only one block RAM module; b) $\forall (P_i, M_j) \in \{\mathbf{P} \rightarrow \mathbf{M}\}$, the memory requirement of P_i is less than the capacity of M_j; and c) The slices of CLBs occupied by synthesized design d is less than A.

The objective is to minimize the total execution time (or maximize the system throughput) under the resource constraints of specific configurable architectures. The desired frequency F and the maximum execution time T among inputs are used as target metrics during compilation and synthesis.

B. Data and iteration space partitioning

Our proposed approach is based on our current efforts on synthesizing C programs into RTL designs. Our system compiler takes C programs, performs necessary transformations and optimizations. By specifying target architecture, and desired performance (throughput), this compiler performs resource allocation, scheduling, and binding tasks, and generates Verilog RTL designs, which can then be synthesized or simulated using commercial tools.

Our proposed solution is shown in Algorithm 1. Before line 7, we adapt existing analysis techniques in parallelization to determine a set of directions to partition. After than, our behavioral-synthesis algorithms synthesize the innermost iteration body. We then evaluate every candidate partitioning, and return the one with the most likelihood achieving good performance subject to the resource constraints.

Given a l-level nested loops, the iteration space is an l-dimensional integer space. The loop bounds of each nested level set the bounds of the iteration space. Each m-dimension data array has a corresponding m-dimensional integer space. Since we assume that index expressions of array references are affine functions of loop indices, footprint of each iteration could be calculated using such affine functions.

With the iteration space $IS(\mathbf{L})$ and the reference footprints F, we can determine a set of directions to partition the iteration spaces. For example, if we have a 2-level nested loop, we usually do column-wise or row-wise partitioning, i.e. we may determine partitioning directions as (0,1) or (1,0). Following these directions, and selecting the proper granularity, we could obtain a good partitioning.

In order to evaluate our candidate solutions, we need to determine their performance on configurable architectures. But it is extremely inefficient to perform synthesis on each candidate solutions.

In our approach, we first synthesize the innermost iteration body with proper resource constraint, obtain performance results for the single iteration, and then use them to evaluate our cost function in line 17. The innermost iteration body is scheduled and pipelined using allocated resources, including 1 block RAM modules, 1 embedded multipliers, and a portion of CLBs, which, by our assumption, are associated with a specific block RAM module. After synthesis, we return resource utilization for the block RAM, multiplier, and the CLBs, respectively. We also output the number of total clock cycles, and the initial interval (II), which describes how great the system throughput could be.

For each partitioning direction, we evaluate every possible partition granularity. Given a specific nested loop and data arrays, and a specific architecture, we can determine the finest and coarsest grain for a homogeneous partitioning (as shown in line 9) With determined partitioning direction and partition granularity, reference functions was used to estimate the total number of memory accesses, and among them, how many are global. As shown in line 16, τ works as a special factor, ranges from 1 to 2, which includes effects of remote memory accesses. When there is no remote memory access, $\tau = 1$, we could achieve a communication-free partitioning; otherwise, we want to minimize it, and then reduce the effects on execution time.

Our cost function, as shown in line 17, give us a good idea how long the execution time will be. Since the iteration body is pipelined, the most utilized components determines the performance (or throughput) of the whole system. For example, after pipelining, $II = 1$, $T = 10$, $u_m = 1$. If there are five iterations in one partition, then the execution time will be $1 \times II \times 5 + (T - II) = 19$ clock cycles, without considering effects of remote memory accesses.

C. Performance Estimation and Optimizations

We apply some optimization techniques, especially those ones taking advantages of FPGA-based configurable architectures, such as scalar replacement, and input prefetching. These optimization techniques could be utilized to reduce memory access, and improve overall performance.

1) Scalar replacement of array elements: Scalar replacement, or register pipelining, is an effective method to reduce the number of memory accesses. This method takes advantage of sequential multiple accesses to array elements by making them available in registers. While executing programs, especially nested loops, one array element may be accessed in different iterations. In order to reduce the amount of memory access, the array element could be stored in registers after

the first memory access, and the following references are replaced by scalar temporaries. Furthermore, registers are much cheaper in FPGA designs compared with ASIC ones.

2) Data prefetching and buffer insertion: Data prefetching was originally introduced to reduce cache miss latencies. The microprocessor issues a prefetching instruction to load a data block which may be accessed very soon. However, in our architecture, there is no such cache hit or misses. We apply a similar prefetching techniques to reduce the delay of critical path, and improve system performance.

Before placement and routing, it is difficult to accurately estimate clock frequency, and to determine how many clock cycles it takes to access a particular block RAM modules. An access to block RAM module far away from the CLB may reduce the system maximal frequency due to the interconnect delay. In order to reduce the memory access time, we schedule the memory access one clock cycle earlier, and insert a register on the data path, if allowed.

VI. Experimental Results

This section presents our experimental setup and results. Our benchmark suite consist of several DSP and image processing applications, including `SOBEL` edge detection, `Bilinear` filtering, `2D_Gauss` and `1D_Gauss` filters. A number of DSP and image applications have the similar control structure and memory access patterns, such as texture smoothing, and convolution. Except the `SOBEL` ones, all other four algorithm cores have the same input size and resource constraints. The target architecture is Xilinx Virtex II FPGA series, which contains evenly distributed block RAM modules. The target frequency was set to 150 MHz for our benchmark suite.

We obtain a data partitioning following our proposed approach, and applied optimizations to further improve system performance, and then used commercial tools to obtain area and timing results. Results are collected after RTL synthesis and placement and routing.

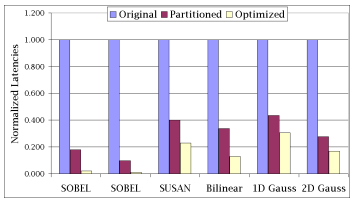


Fig. 4. Normalized Latencies

Figure 4 shows latencies of all designs normalized upon the original un-partitioned designs. We found that the latencies of partitioned designs are greatly smaller that the original ones. After partitioning, the average speedup over the original ones are 2.75 times, and after further optimizations, the average speedup is 4.80 times faster. Since the `SOBEL` applications have different input size and resource constraints, their results are excluded from the average results.

We tested `SOBEL` edge detection with two different input image sizes. If we only partition the data arrays, the number of clock cycles are reduced. However, the maximal frequencies after placement and routing are slower than our desired frequencies. In order to reduce memory accesses, optimization techniques such as scalar replacement for array elements and buffer insertion for data prefetching are utilized. In the smaller design, we finally achieve the 150 MHz design goal, and with a 46x speedup compared to the original design.

However, we could not achieve the design goal in the larger `SOBEL` design. It is interesting that after applying prefetching, both design achieved 185.0 MHz maximal frequency after RTL synthesis. After placement and routing the frequency was drastically reduced to 125.9 MHz. This points to the fact that it extremely important to consider physical attributes of the problem at higher levels of the design.

Figure 5 presents the maximum achievable clock frequencies. In most cases, the partitioned designs are about 10 percent slower than the original ones. However, after applying those optimization techniques, the achievable frequencies are about 7 percent faster than those of partitioned ones. Among half of those designs, the optimized designs could finally achieve the 150 MHz goals. Considering the area of partitioned designs and optimized designs are much larger than the original ones and with more complicate controls, these results is quite good. However, for the `SUSAN`, the frequencies of optimized designs are less than the partitioned one, although the latency of the optimized design is 43 percent lesser.

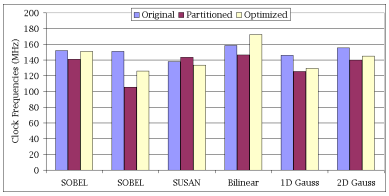


Fig. 5. Maximum achievable frequencies

VII. Conclusion

This work showed that a data and iteration space partitioning approach integrated with existing architectural-level synthesis techniques could parallelize input designs, and dramatically improve system performance. Experimental results indicated that partitioned designs achieve much better performance.

References

[1] Xilinx, *Virtex-II Platform FPGAs: Complete Data Sheet*, October 2003.

[2] R. J. Cloutier and D. E. Thomas, "The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.

[3] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, and T. Omnes, *Data Access and Storage Management for Embedded Programmable Processors.* Kluwer Academic, 2002.

[4] P. R. Panda, N. D. Dutt, and A. Nicolau, "Exploiting Off-Chip Memory Access Modes in High-Level Synthesis," in *Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design*, 1997.

[5] M. Kudlur, K. Fan, M. Chu, and S. Mahlke, "Automatic synthesis of customized local memories for multicluster application accelerators," in *IEEE 15th International Conference on Application-Specific Systems, Architectures and Processors*, 2004.

[6] M. B. Gokhale and J. M. Stone, "Automatic Allocation of Arrays to Memories in FPGA Processors with Multiple Memory Banks," in *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

[7] Z. Huang and S. Malik, "Exploiting Operation Level Parallelism through Dynamically Reconfigurable Datapaths," in *Proceedings of the 39th Conference on Design Automation*, 2002.

[8] M. Budiu and S. C. Goldstein, "Optimizing Memory Accesses For Spatial Computation," in *International Symposium on Code Generation and Optimization*, 2003.

[9] N. Baradaran and P. C. Diniz, "A register allocation algorithm in the presence of scalar replacement for fine-grain architecture," in *the 2005 Conferenc on Design Automation and Testing in Europe*, 2005.

[10] S. Pande and D. P. Agrawal, Eds., *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems*. Springer, 2001.

[11] K.-P. Shih, J.-P. Sheu, and C.-H. Huang, "Statement-Level Communication-Free Partitioning Techniques for Parallelizing Compilers," in *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing*, 1996.

[12] J. Ramanujam and P. Sadayappan, "Compile-time Techniques for Data Distribution in Distributed Memory Machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, October 1991.

[13] S. Pande, "A Compile Time Partitioning Method for DOALL Loops on Distributed Memory Systems," in *Proceedings of 1996 International Conference on Parallel Processing*, 1996.