Data Partitioning for Reconfigurable Architectures with Distributed block RAM

Wenrui Gong Yan Meng Gang Wang Ryan Kastner
Department of Electrical and Computer Engineering
University of California, Santa Barbara
{gong, yanmeng, wanggang, kastner}@ece.ucsb.edu

Timothy Sherwood
Department of Computer Science
University of California, Santa Barbara
{sherwood}@cs.ucsb.edu

Abstract

Contemporary reconfigurable architectures integrate distributed block RAM modules on-chip to provide ample storage for DSP, wireless, and image processing applications. Synthesizing applications to these complex systems requires an effective and efficient approach to conduct data partitioning and storage assignment. This work showed that different data partition schemes and storage assignment solutions, integrated with other memory optimization techniques, dramatically improve overall system performance. Experimental results indicated that partitioned designs could meet design goals and achieve much better performance.

1 Introduction

Reconfigurable systems are a novel computing paradigm, which allows different tradeoffs between flexibility and performance [2, 5]. In order to offer greater computing capabilities, high-performance commercial reconfigurable architectures have integrated a number of fixed components, including microprocessor cores, DSPs, custom hardware, and on-chip distributed memory.

Reconfigurable devices 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 as modern reconfigurable architectures have a complex memory hierarchy. Memory optimizations for these reconfigurable architectures differs significantly to previous memory optimizations in parallelizing compilation for multiprocessor architectures, and closely interfere with high-level synthesis efforts, and physical synthesis results. This paper focuses on seeking a partitioning-based solution to the storage assignment problem for reconfigurable architecture with distributed block RAM modules.

The central contribution of this paper is an novel approach of deriving an appropriate data partitioning and synthesizing the program behavior to reconfigurable architectures. By intensive research on interferences of data partitions and architectural synthesis decisions, such as scheduling and binding, the synthesized designs are expected to meet the design goals, and minimize the execution time (or maximize the system throughput) under resource constraints.

2 Target Reconfigurable Architectures

Contemporary reconfigurable architectures usually integrate a number of distributed block memories. According to the RAM blocks, these architectures could be divided into heterogeneous architectures and heterogeneous ones. Figure 1 presents such a *homogeneous* architecture. A number of block RAMs are evenly distributed on the chip, and connected with CLBs using reprogrammable interconnect. Each block RAM has the same capacity as others. Usually there is an embedded multiplier located baside the block RAM for DSP applications.

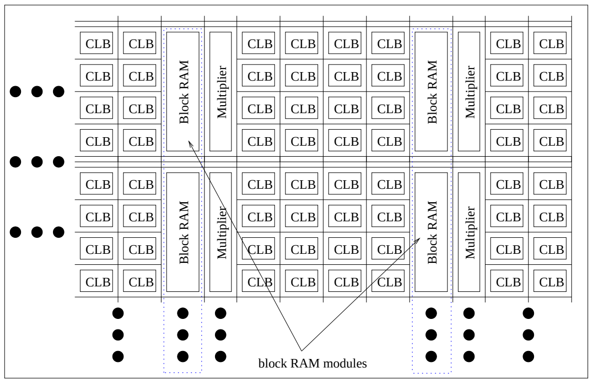

Figure 1. FPGA with distributed Block RAMs

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 -6 FPGA. And it takes an extra ε ns to transfer data from the memory port to the accessing CLB.

In this paper, we aim for data partitioning and assigning distributed block RAM modules to data partitions. Compared to off-chip global memory and using CLBs as distributed RAM, this approach is an effective and efficient solution to most DSP, image, and wireless applications.

3 Related Work

Data partitioning and storage assignment problem was well studied in the field of parallelizing compilation [1, 7, 10]. Early efforts developed effective analysis techniques and program transformations to reduce global communications and hence improve system performance. Shih and Sheu [9], and Ramanujam and Sadayappan [8] addressed the methodology to achieve *communication-free* iteration space and data partitioning problem. Pande [6] presented an *communication-efficient* data partitioning solution when it is impossible to get a communication-free partitioning.

However, the following differences between multiprocessor architecture and reconfigurable architecture with distributed block RAM modules make it impossible to directly migrate them in our system compiler.

- The target architectures are different. Multiprocessor parallel system has a fixed number of microprocessors. They usually exhibit-the non uniform memory access (NUMA) attributes. In reconfigurable architectures, through the number of block RAM modules are fixed. There is no determinate CLBs associated with a particular block RAM. Hence the boundaries between local and remote memory are indistinct.
- Programs are executed sequentially or with limited instruction level parallelism (ILP) on multiprocessor parallel systems, though parallelizing compiler exploits coarse-grained parallelism. However, computing tasks runs in a fully parallelized and concurrent manner on reconfigurable architectures.

Most of the previous efforts assumed that global communications or latencies to remote memory are an order of magnitude slower than access latencies to local memory. This make it reasonable to simplify the objective function to simply reduce the amount of global communications. This assumption is no longer true in the context of reconfigurable architectures. As previously described, access latencies to block RAM modules depends on the distance between the accessing CLBs and the memory ports. It is difficult to determine the exact delay before obtained placement and routing results.

Second, data partition and storage assignment have more compound effects on system performance. While synthesizing programs into reconfigurable systems, it is extremely difficult to determine the execution time before physical synthesis. Given a scheduled design, there could be a 30-50% variation in execution time, even though numbers of clock cycles are almost the same. Not only the memory access delays but also the control logics and computations are affected.

In summary, our architectures differ from traditional NUMA machines. It is difficult to estimate candidate solutions during the early stage of synthesis. Flexibilities in configuring block RAM modules greatly enlarge the solution spaces, and hence make the problem more challenging.

4 Data Partitioning and Storage Assignment

This section discusses the data partitioning and storage assignment problem and some techniques we use to reduce memory accesses and improve system performance for FPGA-based reconfigurable architectures with distributed block RAM modules.

4.1 Data Partitioning

At the current stage, we mainly focus on data-intensive applications in DSP and image processing. 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 will be assign to block RAM modules; and *e*) each data element will be assigned one and only one single block RAM modules, i.e. no duplicate data.We further assume that all data types are fix-point numbers due to the current capability of our system compiler and synthesis tools.

The problem is to minimize the total execution time (or maximize the system throughput) under the resource constraints of specific reconfigurable architectures by partitioning data arrays and assigning these portions into the distributed RAM modules.

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.

As discussed before, in reconfigurable architectures, the boundaries between local and remote accesses are indistinct. In our preliminary experiments, we found that, given the same datapath with memory accesses to block RAM modules with different locations, the lengths of critical path achieved after placement and routing have a 30-50% variation. And a limited number of datapaths could be placed near the block RAM modules which they access.

Therefore, we could still assume that, once the data space are partitioned, we could obtain a corresponding partitioning of the iteration space, or the computations. Each portion of the data space could be mapped to one portion of the iteration space. Then we divide all memory accesses into local accesses and remote ones (or communications). However, these local and remote memory accesses distinguish from those in parallel multiprocessor systems on that the differences of access latencies are usually in the same magnitude rather than in orders of magnitude.

Based on this further assumption, we adapt some concepts and analysis techniques in tradition parallelizing compilation. *Communication-free* partitioning refers to a situation that each partition of the iteration space only access the associated partition of data space. If we could not find a communication-free partitioning, we look for a *communication-efficient* partitioning to minimize the execution time.

Our proposed approach integrates traditional program test and transformation techniques in parallelizing compilation into our system compiler framework. In order to tackle the performance estimation during data space partitioning, we use our specific behavioral-level synthesis techniques, such as resource allocation, scheduling and binding.

4.2 Performance Estimation and Optimizations

In order to evaluate our data partition and storage assignment solutions, we apply architectural-level synthesis techniques to each portion of the partitioned design. Sophisticated algorithms during scheduling and binding are applied to benefit some critical computations and controls.

Besides traditional architectural-level synthesis techniques, we apply other optimization techniques, especially those ones taking advantages of FPGA-based reconfigurable architectures, such as scalar replacement, and input prefetching. These optimization techniques could be utilized to reduce memory access, and improve overall performance.

4.2.1 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.

4.2.2 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, which is most useful that access large array sequentially. However, in FPGA-based reconfigurable architectures with block RAM modules, there is no 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, especially in some high-speed designs. 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. Hence the critical path may be reduced and the data will available on time.

5 Experimental Results

This section presents experimental setup and results. Two examples are intensively examined. The first benchmark is a bank of correltors, which is very popular in DSP and wireless applications, such as Kalman filters, RLS, and MMSE [4]. The second benchmark is Sobel edge detection, which applies horizontal and vertical Sobel edge detection masks to an input image.

The target architecture is Xilinx Virtex II FPGA series, which contains evenly distributed block RAM modules. Target frequency was set to 200 MHz for the DSP application, and 150 MHz for the image processing application. We partitioned the arrays using the approach proposed in Section 4.2, and performed program transformations, and then used commercial tools to obtain area and timing results. Experiments results are collected after RTL synthesis and placement and routing.

5.1 Communication-free: 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. In this case for the matching pursuit algorithm, both l and m equal to 88. If the S matrix is kept packed, the most advanced commecial 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.

The original data space could be partitioned by column or by row direction. Our proposed approach showed us that column-wise partition could achieve communication-free partitioning. In a column-wise partition, computations of each correlator are conducted using embedded multipliers beside the block RAM in a multiplication and accumulation (MAC) manner. For each correlator, control logic and computation resources could be treated as local to the block RAM module. Table 2 presents experimental results of the column-wise data partitioning.

Table 2 also presented area and timing trends of different granularity for the column-wise scheme. 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.

To evaluate different partitioning schemes, we also obtained performance results for row-wise partitions, as shown in Table 1, we found more interesting results. In the term of numbers of clock cycles, the differences are really small. However, if we check the achieved maximal frequencies, or the latencies for the whole bank of correlators, 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 great amount of global communications.

Data per BRAM	# of cycles	Pre-layout Timing		Post-layout Timing	
		F(MHz)	L(ns)	F(MHz)	L(ns)
1 row	184	140.5	1309	133.5	1378
4 rows	710	157.0	4520	129.4	5486
8 rows	1413	147.1	9602	138.7	10183

Table 1. Experimental results of row-wise partition

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
4 columns	706	205.0	3436	178.2	3961
8 columns	1410	198.6	7099	161	8752

Table 2. Experimental results of column-wise partition

In summary, different partitions of the array S deliver a wide variety of candidate solutions. Synthesized designs showed that data partitioning and storage assignment not only affect the number of clock cycles, but also affect the achieved clock frequencies. Generally speaking, the design with less remote accesses or less communications could achieve better performance.

5.2 Efficient communication: Sobel

Sobel edge detection applies horizontal and vertical Sobel edge detection masks to an input image. This application is a 2-level nested loop. A number of image application have the same control structure and memory access patterns, such as texture smoothing, and convolution [3].

```
for (i=1; i<N-1; i++)
  for (j=1; j<M-1; j++){
    ...
    i00=in[i-1][j-1]; i01=in[i-1][j]; i02=in[i-1][j+1];
    i10=in[i  ][j-1];               ; i12=in[i  ][j+1];
    i20=in[i+1][j-1]; i21=in[i+1][j]; i22=in[i+1][j+1];
    ...
  }
```

Figure 2. Memory accesses in Sobel edge detection

Based on results from code analysis stage, we could not obtain a communication-free partition. Now the task is to find a communication efficient partition which could meet design goals.

Table 3 and 4 showed timing results for two Sobel edge detection with different input 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.

256× 8	# of	Pre-layout Timing		Post-layout Timing	
Sobel	cycles	F(MHz)	L(ns)	F(MHz)	L(ns)
original	12,196	159.5	76,481	152.2	80,444
partitioned	2,032	150.4	13,514	140.7	14,445
+scalar replacement	771	166.1	4,642	145.7	5,291
+prefetching	263	185.0	1,421	150.8	1,744

Table 3. Comparing optimization techniques (1)

256× 16	# of	Pre-layout Timing		Post-layout Timing	
Sobel	cycles	F(MHz)	L(ns)	F(MHz)	L(ns)
partitioned	2,032	145.9	13,925	105.6	19,155
+scalar replacement	7,71	153.4	5,026	118.2	6,522
+prefetching	263	185.0	1,421	125.9	2,088

Table 4. Comparing optimization techniques (2)

In summary, different optimization techniques could be utilized to increase memory bandwidth, reduce memory access, and improve overall performance. When the sizes of designs increase, it becomes more difficult to achieve design goals since it lacks the support from down-stream tools, especially physical design tools.

6 Concluding Remarks

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 or system throughput. Experimental results indicated that partitioned designs achieve much better performance.

In future work, we plan to investigate analysis and transformation techniques to deal with heterogeneous architectures and generate heterogeneous partitions. It will also be interesting to handle irregular iteration space and control constructs in iteration bodies. Furthermore, we wish to integrate layout information during our architectural-level synthesis. It would be promising to obtain more accurate estimation of interconnect delay and direct physical design tools using our architectural-level synthesis.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Karfmann Publishers, San Francisco, CA, 2002.

[2] K. Bondalapati and V. K. Prasanna. Reconfigurable Computing Systems. *Proc. of the IEEE*, 90(7):1201–17, July 2002.

[3] R. C. Gonzalez and R. E. Woods. *Digital Image Processing, 2nd Edition*. Prentice Hall, Englewood Cliffs, NJ, 2002.

[4] S. Haykin. *Adaptive Filter Theory, Fourth Edition*. Prentice Hall, Englewood Cliffs, NJ, 2001.

[5] R. Kastner, A. Kaplan, and M. Sarrafzadeh. *Synthesis Techniques and Optimizations for Reconfigurable Systems*. Kluwer Academic, Boston.

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

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

[8] J. Ramanujam and P. Sadayappan. Compile-time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–82, October 1991.

[9] 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.

[10] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.