Data Partitioning for Reconfigurable Architectures with Distributed Block RAM

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

Abstract

Modern, high performance reconfigurable 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 formally describe this problem and show how it is much different from the traditional data partitioning problem for compilation to parallel processing systems. 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 up to 50% increase in frequency, without increasing the number of cycles, by simply minimizing remote accesses. Other optimization techniques like memory port configuration, scalar replacement, prefetching and buffer insertion can further minimize remote accesses and lead to 46x speedup in overall runtime.

1 Introduction

Reconfigurable systems are a novel computing paradigm, which allow different tradeoffs between flexibility and performance [3, 10]. Typical reconfigurable computing systems consist of arrays of reprogrammable logic blocks and flexible interconnect. Such architectures distinguish themselves from traditional microprocessor architectures in that reconfigurable computing systems work in a complete parallelized manner, and exhibit an inherent computational density advantage over microprocessors [6].

In order to offer greater computing capabilities, high-performance commercial reconfigurable architectures provide ample configurable logic, and have integrated a number of fixed components, including digital signal processing (DSP) and microprocessor cores, custom hardware, and on-chip distributed memory. For instance, the Xilinx Virtex-II Pro Platform FPGA series provides 3K to 125K logic cells, up to four PowerPC processor cores and 1,738 kilobytes of distributed, embedded block RAM.

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 differ significantly to previous memory optimizations in parallelizing compilation for multiprocessor architectures, as there is a tight coupling with high-level synthesis 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 a novel approach of deriving an appropriate data partitioning, and synthesizing the program behavior to reconfigurable architectures. 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 remote 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 flexible port configuration, scalar replacement, and data prefetching and buffer insertion, are applied to reduce memory accesses and improve the overall performance.

This work is organized as follows. The next section gives details on the target reconfigurable architecture and the following section describes related work. Section 4 formally describes the data partitioning and storage assignment problem and provides techniques to minimize the number of remote data memory accesses. Section 5 presents our experimental results and we conclude in Section 6.

2 Target Reconfigurable Architecture

FPGA architectures consist of an array of lookup tables (LUTs), flip-flops, and programmable interconnect. A fixed number of LUTs and flip-flops are grouped to form a configurable logic block (CLB), or a logic array block (LAB). Programmable switchboxes connect these logic blocks to provide the required interconnect. Modern reconfigurable architectures incorporate a number of dis-

tributed block memories. These architectures can be divided into homogeneous and heterogeneous architectures according to the capacities and distribution of the RAM blocks.

Figure 1 presents an example of a *homogeneous* architecture. This roughly corresponds to Xilinx Virtex II FPGA [16]. 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. A large Virtex II chip contains 168 blocks of 18 Kbits block RAM modules, providing 3,024 Kbits of on-chip memory.

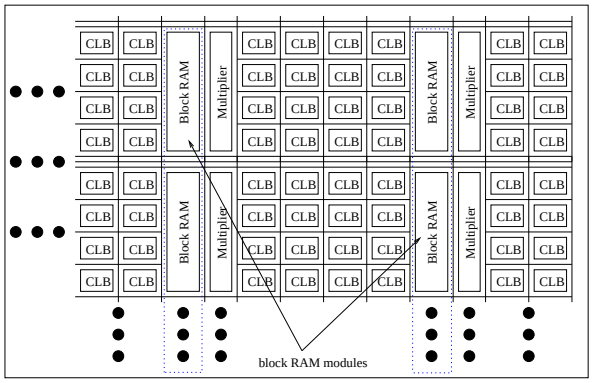


Figure 1. FPGA with distributed Block RAMs

The *heterogeneous* architecture contains a variety of block RAM modules with different capacities. For example, the TriMatrix memory on an Altera Stratix II FPGA chip [2] consists of three types of on-chip block RAM modules: M512, M4K, and M-RAM. Their capacities are 576 bits, 4 Kbits, and 512 Kbits, respectively. A given Stratix II chip may contain a large number of M512 and M4K modules, but only a few M-RAM modules. Currently our work only considers homogeneous architectures.

In order to efficiently support different applications, block RAM modules could be configured as dual-port RAM, single-port RAM, ROM, and FIFO buffers. This feature enables a great flexibility.

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.

In addition to block RAM modules, CLBs can be configured as local memory, which is convenient for storing intermediate results. When CLBs are configured as distributed memory, the access latency, i.e. the logic access time, is quite small. However, if a number of CLBs are assigned to an array, it involves extra delay for MUX selecting the addressed element. For example, the delay for a 512 bit CLB memory is around 3.5 ns for Xilinx XC2V3000 FPGA; the delay for a 16 Kbit CLB memory increases to 6.2 ns.

The FPGA can be complimented by an external, global memory for storing large amounts of data. Access latencies to the external memory depend on the bus protocol and type of memory. The access latencies usually are an order of magnitude slower than those of on-chip block RAM.

In this paper, we develop a methodology for partitioning data to distributed block RAM modules. When compared to off-chip global memory and using CLBs as distributed RAM, this approach is an effective and efficient solution to most applications.

3 Related Work

Data partitioning and storage assignment problem was well studied in the field of parallelizing compilation [1, 12, 15]. Early efforts developed effective analysis techniques and program transformations to reduce global communications and hence improve system performance. Shih and Sheu [14], and Ramanujam and Sadayappan [13] addressed the methodology to achieve *communication-free* iteration space and data partitioning problem. Pande [11] presented an *communication-efficient* data partitioning solution when it is impossible to get a communication-free partitioning.

However, the following differences make it impossible to directly migrate these approaches into a system compiler for reconfigurable architectures with distributed block RAM modules.

- The target architectures are different. Multiprocessor systems have a fixed number of microprocessors. Each microprocessor has its own local memory, and is connected with a different remote memory modules that exhibit non uniform memory access (NUMA) attributes. Reconfigurable architectures execute programs using CLBs rather than microprocessors. 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 each microprocessor, while the parallelizing compiler exploits coarse-grained parallelism. Computing tasks runs in a fully parallelized and concurrent manner on reconfigurable architectures.

Our problem is distinguished from the previous studies as follows. First of all, these differences violate a fundamental assumption held in the previous research. 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 makes 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 data partitioning for reconfigurable architectures. As previously described, the boundaries between local and remote memory are indistinct. Access latencies to block RAM modules depends on the distance between the accessing CLBs and the memory ports. There is no way to determine the exact delay before performing placement and routing.

Second, data partition and storage assignment have more compound effects on system performance. In parallelizing compilation for multiprocessor architectures, once computations and data are partitioned, it will be relatively easy to estimate the execution time since the clock period is fixed, and the number of clock cycles consists of the communication overheads and computation latencies for each instruction. However, it is extremely difficult to determine the execution time in reconfigurable systems before physical synthesis. Our results in Section 5 show that even though number of clock cycles are almost the same, there can be 30-50% deviations in execution time due to variation in frequency. Therefore, the control logic and computation times are effected, and not just the memory access delays.

Moreover, the flexibility to configure block RAM modules make this problem even more difficult. Block RAM modules could be configured with a variety of $width \times depth$ schemes, and the memory ports support can handle different read/write combinations.

Early efforts on utilizing multiple memory modules on FPGA chips [7] allocated an entire array to a single memory module rather than partitioning data arrays. Furthermore, they assumed that the latencies differences had little effect on system throughput.

In summary, reconfigurable architectures are drastically different from traditional NUMA machines, making it 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 even more challenging.

4 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 reconfigurable architectures with distributed block RAM modules.

4.1 Problem formulation

We focus on data-intensive applications in digital signal 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 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 to this data partitioning and storage assignment problem are as follows:

- A program d contains an l-level perfectly nested loop $\mathbf{L} = \{L_1, L_2, \ldots, L_n\}$
- The program d accesses a set of n data arrays $\mathbf{N} = \{N_1, N_2, \ldots, N_n\}$.
- A specific target architecture, i.e. an FPGA, contains a set of m block RAM modules $\mathbf{M} = \{M_1, M_2, \ldots, M_m\}$. This FPGA also contains A CLBs.
- 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 $\mathbf{P} = \{P_1, P_2, \ldots, P_p\}$, where $p \leq m$, and seek an assignment $\{\mathbf{P} \rightarrow \mathbf{M}\}$ subject to the following constraints

- $\bigcup_{i=1}^{p} P_i = \mathbf{N}$, 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.
- $\forall (P_i, M_j) \in \{\mathbf{P} \rightarrow \mathbf{M}\}$, the memory requirement of P_i is less than the capacity of M_j

After obtaining data partitions and storage assignment, we reconstruct the input program d, and conduct behavioral-level synthesis. After RTL and physical synthesis, the synthesized design must satisfy the following constraint.

- 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 reconfigurable architectures. The desired frequency F and the maximum execution time T among inputs are used as target metrics during compilation and synthesis.

4.2 Overview of the proposed approach

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.

4.3 Data and iteration space partitioning

This section discusses our data and iteration space partitioning algorithm in detail. The algorithm is illustrated in Algorithm 1. Before line 7, we adapt existing analysis techniques in parallelizing compilation to determine a set of directions to partition. In line 7 and 8, we call our behavioral-synthesis algorithms to synthesize the innermost iteration body. After that, we evaluate every candidate partitioning, and return the one with the most likelihood achieving the short execution time 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. An integer point in this iteration space solely refers to an iteration, which includes all statements in the innermost iteration body.

Each m-dimension data array has a corresponding m-dimensional integer space. And an integer point refers to a data element with that data index.

In each iteration, data elements in the data space are accessed. 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, i.e. each iteration is mapped to a set of data points in the data space by means of specified array reference.

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}

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 reconfigurable architectures. Since most design problems in behavior synthesis are *NP*-complete, and time-consuming. 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

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

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. The reason why we pipeline our design is, for a large iteration space $IS(\mathbf{L})$, the pipelined iteration body gives the shortest execution time, and the best resource utilization. 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, the finest grained partition depends on at least how many iterations should be put in one block RAM modules. On the other side, the coarsest grained partition depends on how large the capacity of a block RAM module is.

With determined partitioning direction and partition granularity, we could use reference functions to estimate the total number of memory accesses, and among them, how many are remote memory accesses. 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.

4.4 Performance Estimation and Optimizations

In order to evaluate our data partitioning and storage assignment solutions, we apply architectural-level synthesis techniques to each portion of the partitioned design using sophisticated scheduling and binding algorithms. In addition to the traditional architectural-level synthesis techniques, we apply other optimization techniques, in particular those that take advantage of FPGA-based reconfigurable architectures, such as port vectorization, scalar replacement, and input prefetching. These optimization techniques could be utilized to increase memory bandwidth, reduce memory accesses, and improve overall performance.

4.4.1 Flexible port configuration

Different memory bandwidths are available using the flexible port configurations. As described in Section 2, block RAM modules could be configured as dual-port RAM, single-port RAM, and FIFO buffers to support different applications. For example, the 18 Kbit block RAM dual-port memory on Virtex II FPGA [16] consists of an 18 Kbit storage area and two *completely independent* access ports. Data can be written to the first, second or both ports and can be read from either or both of the ports. Each port can be configured in a variety of aspect ratios, e.g. 2K×8 and 512×36.

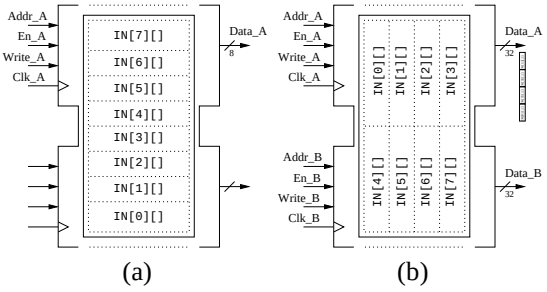


Figure 2. Different port configurations support different memory bandwidths

Figure 2 shows an unsigned char array `IN[8][256]` assigned to an 18 Kbit block RAM. Figure 2(a) illustrates that this block RAM is configured as a 2K×8 single-port RAM. Each memory access could fetch or write 8 bits data. While Figure 2(b) presents a 512×32 dual-port RAM. Each memory access could fetch or write 32 bits data through one memory port. For example, if `Addr_A`= j_1 and `Addr_B`= j_2, one clock cycle later, `Data_A`= $\{\texttt{IN}[0][j_1 - 256], \texttt{IN}[1][j_1 - 256], \texttt{IN}[2][j_1 - 256], \texttt{IN}[3][j_1 - 256]\}$, and `Data_B`= $\{\texttt{IN}[4][j_2], \texttt{IN}[5][j_2], \texttt{IN}[6][j_2], \texttt{IN}[7][j_2]\}$. Therefore, the memory bandwidth in configuration (b) is 8 times greater than that of configuration (a).

4.4.2 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 [4]. When executing a program, especially those with nested loops, one array element may be accessed in different iterations. In order to reduce the amount of memory access, the

array element can be stored in registers after the first memory access, and the following references are replaced by scalar temporaries. This is especially beneficial for reconfigurable systems as registers are much cheaper in FPGAs compared with ASIC designs.

(a)

```
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];
    ...
  }
```

$\Longrightarrow$

(b)

```
...  // initial two iterations
for (i=3; i<N-1; i++)
  for (j=1; j<M-1; j++){
    ...
    i00=i10; i01=i11; i02=i12;     // scalar replacement
    i10=i20; i11=i21; i12=i22;     // scalar replacement
    i20=in[i+1][j-1]; i21=in[i+1][j]; i22=in[i+1][j+1];
    ...
  }
```

Figure 3. Scalar replacement of array elements

Consider the SOBEL edge detection code given in Figure 3. Part of the references to array `in[]` could be replaced by scalar temporaries obtained in the previous iterations. This reduces the number of memory accesses by approximately 62%. If the implementation is pipelined, the design will have better throughput using the same memory ports configuration.

4.4.3 Data prefetching and buffer insertion

Data prefetching was originally introduced to reduce cache miss latencies [5]. The microprocessor issues a prefetching instruction to load a data block that will be accessed in the near future. Prefetching avoids stalling by having the data readily accessible when it is needed. While it is loading data the main memory, the microprocessor executes other computations that are independent of the data being fetched. Prefetching is most useful in programs that access large array sequentially. There are no caches in FPGA-based reconfigurable architectures with block RAM modules. However, we can apply 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 will take to access a particular block RAM module. An access to block RAM module far away from the CLB may reduce the system maximal frequency due to the interconnect delay, especially in high-speed designs. For example, in Figure 4(a), it is faster for CLB (*c*) to access block RAM (*a*) than to access block RAM (*b*).

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 will be reduced and the data will available on time. Figure 4(b) shows a design in which the data in block RAM (*b*) is fetched one clock cycle earlier. This is similar to software prefetching. However, our goal is to reduce the critical path, or maximize the clock frequency.

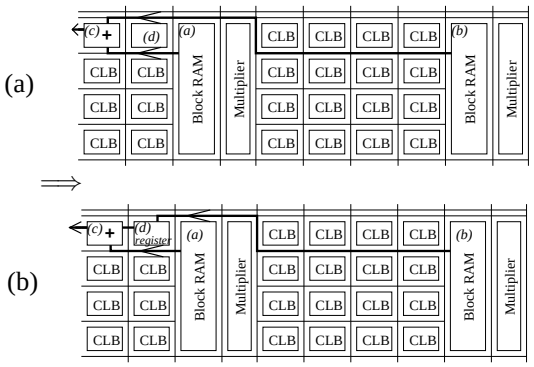


Figure 4. Data prefetching and buffer insertion

5 Experimental Results

This section presents our experimental setup and results. The target architecture is Xilinx Virtex II FPGA series, which contains evenly distributed block RAM modules.

We focus on two applications. The first benchmark is a bank of correlators, which is a commonly occurring operation in DSP applications, e.g. Kalman filters, matching pursuit (MP), recursive least squares (RLS), and minimum mean-square error estimation (MMSE) [9]. The bank of correlators multiplies each sample of a received vector r with the corresponding sample in the correlation matrix S. A communication free partitioning exists for this application. The second benchmark is Sobel edge detection, which applies horizontal and vertical detection masks to an input image. A number of image application have the same control structure and memory access patterns, such as texture smoothing, and convolution [8]. A communication-free partition does not exist in this application.

We obtain a data partitioning following our proposed approach, and applied other optimization to further improve system performance. The target frequency was set to 200 MHz for the bank of correlators and 150 MHz for Sobel edge detection. 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.

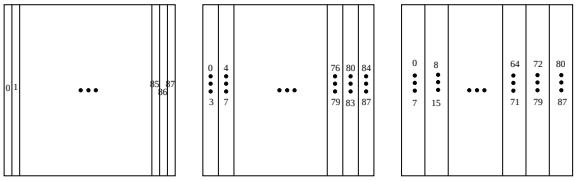
5.1 Communication-free: Correlation

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. 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.

The data space can be partitioned by column or by row. Our proposed approach showed us that column-wise partition achieves a communication-free partitioning. Figure 5 suggests several candidates column-wise partitionings. Figure 5(a), (b), and (c) assign one block RAM to one column, four columns, and eight columns, respectively.

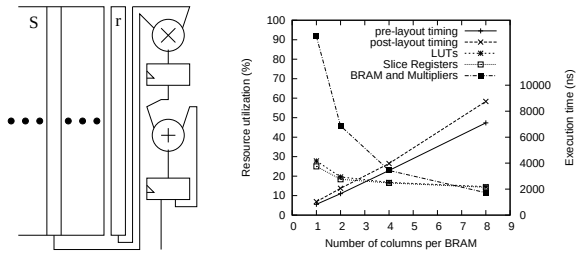


(*a*) (*b*) (*c*)

Figure 5. Candidate communication-free data partitioning

Figure 6(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.

Figure 6(2) presented area and timing trends of different granularity for the column-wise scheme. As shown in Figure 6(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.

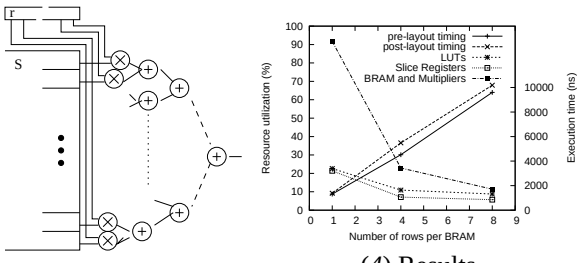


(*1*) Column-wise (*2*) Results

Figure 6. Implementations and area/timing trade-offs

To evaluate different partitioning schemes, we also obtained performance results for row-wise partitions.

Figure 7(1) illustrates the parallel computation scheme, or the *by-row* scheme, where one block RAM was assigned to one or multiple rows. Data at the same column are read and multiplied using the local fixed multiplier, and a



(*1*) Row-wise (*4*) Results

Figure 7. Row-wise partitioning

pipelined adder-tree is used for the summation of the products. The adder tree requires remote accesses to each of the block RAMs, hence this is not a communication-free partitioning. This scheme parallelizes each correlator, and hence requires a global control on the multipliers and the pipelined adder-tree. Figure 7(2) presented area and timing trends of different granularity for both schemes, respectively.

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

Table 1. Comparison between the same granularity

When comparing the two schemes using the same granularity (i.e. same number of rows/columns), as shown in Table 1, we found interesting results. In the term of numbers of clock cycles, the differences are very small. However, if we check the maximal achievable 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 increased amount of global communications needed for the control logic and remote block RAM accesses.

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 fewer remote accesses, or equivalently, reduced global communications achieves better performance.

5.2 Efficient Communication: Edge Detection

Sobel edge detection applies horizontal and vertical detection masks to an input image. This application is a 2-level nested loop as shown in Figure 3. Based on results from code analysis stage, we could not obtain a communication-free partition. Now the task is to find a communication efficient partitioning that meets the design

goals.

Tables 2 and 3 show timing results for 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.

256× 8 Sobel	# of cycles	Pre-layout Timing		Post-layout Timing	
		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 2. Comparing optimization techniques (1)

256× 16 Sobel	# of cycles	Pre-layout Timing		Post-layout Timing	
		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 3. Comparing optimization techniques (2)

However, we could not achieve the design goal in the larger design. It is interesting to note that after applying prefetching, both design achieved 185.0 MHz maximal frequency after RTL synthesis. However, 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.

In summary, different optimization techniques could be utilized to increase memory bandwidth, reduce memory access, and improve the 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 Conclusion

Modern reconfigurable computing systems offer enormous computing capacities, and continue to integrate on-chip computation and storage components. Advanced synthesis tools are required to map large applications to these increasingly complicated chips. More importantly, these tools must be smart and powerful enough to conduct memory optimizations and effectively utilize on-chip distributed block RAM modules.

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.

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.

References

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

[2] Altera Corporation. *Stratix II Device Handbook*, January 2005.

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

[4] D. Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for Subscripted Variables. In *Proceedings of the SIGPLAN '90 Symposium of Programming Language Design and Implementation*, 1990.

[5] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of the 4th International Conference on Architecture Support for Programming Languages and Operating Systems*, 1991.

[6] A. DeHon. The Density Advantage of Configrable Computing. *Computer*, 33(4):41–49, April 2000.

[7] 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.

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

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

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

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

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

[13] 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.

[14] 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.

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

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