Memory Synthesis for FPGA-Based Reconfigurable Computers*

Amit Kasat, Iyad Ouaiss, and Ranga Vemuri

Department of ECECS, ML 30, University of Cincinnati
Cincinnati, OH 45221-0030, U.S.A.
{akasat,iouaiss,ranga}@ececs.uc.edu

Abstract. For data intensive applications like Digital Signal Processing, Image Processing, and Pattern Recognition, memory reads and writes constitute a large portion of the total design execution time. With the advent of on-chip memories, a rich hierarchy of physical memories is now available on a Reconfigurable Computer (RC). An intelligent usage of these memories can lead to a significant improvement in the latency of the overall design. This paper presents an automated heuristic-based memory mapping framework for RCs. We use a Tabu search guided heuristic, *Rectangle Carving*, to map a single data structure onto several instances of a memory type on the RC. We also introduce control logic to resolve potential memory access conflicts and to make the details of memory mapping transparent to the accessing logic.

1 Introduction

FPGAs have been the focus of attention because of the quick turn-around design time. Most designers utilize FPGAs as a prototyping platform where the focus is on functionality. However, with the increasing pressure on time to market, the tremendous increase in the density and complexity of programmable devices, and the flexibility offered by such devices, FPGAs have become a viable alternative to ASIC implementations in many situations. Thus, the performance of designs mapped to FPGA-based platforms has become equally important.

Contemporary FPGA architectures provide a large number of fast physical memories on the device. Xilinx Virtex *BlockRAMs* [1], Altera FLEX 10K *Embedded Array Blocks* [2], and Altera APEX E *Embedded System Blocks* [3] are some of the examples.

With many variations in properties of physical memories, the mapping of data structures in the design is a non-trivial task. Physical memories have multiple ports through which the storage space can be accessed in parallel. There are multiple configurations possible for each port that can be used to minimize wastage of storage space. Also, different types of memories present on a RC provide different read and write latencies. A memory mapper should take all these factors into account.

* This work is supported in part by the US Air Force, Wright Laboratory, WPAFB, under contract number F33615-97-C-1043.

G. Brebner and R. Woods (Eds.): FPL 2001, LNCS 2147, pp. 70–80, 2001.

The rest of the paper has been arranged as follows. Section 2 discusses some related research. Section 3 describes various features of an RC architecture and the model for input design. Section 4 describes what constitutes mapping of data structures and gives a formal definition of the mapping problem. It also presents details about Tabu Search adapted for the problem. Section 5 presents the heuristic algorithm used to perform the actual mapping. Section 6 describes control logic generation for memory mapping. Various constraints in mapping are presented in Section 7 and the components of the cost function for tabu search in Section 8. We present the results in Section 9.

2 Related Work

As outlined in [4], the process of mapping *data structures to physical memory* can be divided into two steps: *(i)* translating the storage requirements into *logical memories (LMs)*, i.e. forming the data structures needed by the design, and *(ii)* mapping the LMs onto the *physical memories* of the hardware; i.e. assigning the data structures to the memory banks.

In memory synthesis for ASICs, the problem is that of mapping various data structures onto a predefined set of library components. The optimization goals include minimizing the number of different physical memory components used, thus minimizing the overall area; and placing the chosen components so as to minimize routing requirements and signal delay. An ILP approach has been used in [5,6] to group registers to form multi-port memory modules. [7] concentrates on minimizing the area while finding a legal packing of the logical segments into the physical segments.

However, in the case of RCs, the resources available are already fixed. The goal is to optimize performance while satisfying the constraints posed by the RC. [8] and [9] consider single and dual ported memories in FPGAs, but not both at the same time.

In [10], an ILP formulation considers all instances simultaneously and gives optimal mapping. However, as the problem size increases, it takes a long time to converge. In [11], mapping is done in two stages (*global* and *detailed*) to simplify the ILP formulation. However, if an LM is bigger than an instance, it is split in some predefined manner during the first stage. This may prevent the mapper to find the optimal solution.

3 Problem Formulation

3.1 Reconfigurable Platform Description

A generic RC has a set of programmable devices (FPGAs), a set of physical memories, and an interconnection network for communication between various FPGAs and memory banks.

The various physical memories on a RC can be grouped into different clusters, called *memory types*, on the basis of similarities in some of their invariant

attributes: A fixed connection already exists between instances of a memory type and its *local_pe*. *Read and Write Latencies* are the number of clock cycles required to perform corresponding operations on that *memory type*. *Num_ports* is the number of ports available on *each* instance of this memory type. The storage space can be accessed through any of these ports in parallel. *Max_storage_bits* is the maximum number of bits of data that can be stored in an instance of this memory type. *Configuration* of a port is the way in which storage space is accessed through that port. It is specified by *(width,depth)* pair. Each word has *#width* bits and there are a total of *#depth* words. *Num_configurations* is the number of different ways in which a port of an instance of this memory type can be configured. *Num_instances* gives the total number of elements of this memory type which are available on the RC. A signal from *local_pe* needs to traverse through *pins_trav* number of pins to access the memory. It represents the proximity of the memory bank to the logic area and is predominant in determining the maximum clock frequency.

3.2 Input Design Description

An LM is modeled as a rectangle; the number of words is represented by the depth of the rectangle while the width represents the number of bits in each word. Each rectangle has a weight, specified by a pair of integers *(num_reads, num_writes)* . The higher the weight of an LM, the greater the influence of the LM on the overall design latency.

4 Tabu Search Formulation

Tabu Search (TS) [12], is a general purpose *meta-heuristic* for solving combinatorial optimization problems. This paper uses Tabu Search to perform memory mapping.

We view an LM as a *complete rectangle*. If split, the complete rectangle can be decomposed into *sub-rectangles*. A mapped sub-rectangle is specified by a tuple of 6 values as shows below:

$$sub\_rectangle = \{phy\_mem\_num, port\_num, depth, width, start\_depth, start\_width\}$$

Physical_mem_num specifies the physical instance to which this sub-rectangle has been mapped. *Port_num* specifies which port of that physical instance will be used to access this sub-rectangle. *depth* and *width* are the dimensions of the sub-rectangle indicating how much of the LM has been mapped to this port. *Start_depth* and *start_width* specify the position of the top-left corner of the sub-rectangle with respect to the complete rectangle. Together, the last four parameters exactly specify which part of the LM is mapped through this sub-rectangle. This information is helpful in automating the initialization of physical instances on the RC. We define the mapping of an LM to be a set of sub-rectangles, one corresponding to each part into which the memory task has been split.

For a mapping to be valid, we require that all its sub-rectangles be mapped to physical instances of the *same memory_type*. We further require that no two sub-rectangles share the same port. If different parts of an LM are mapped to two physical memories whose *local_pes* are not the same, the address and data bus will need to be routed to both PEs. This can greatly deteriorate the quality of the solution. Even if the *local_pe* is the same, memories might differ in read/write latencies. Thus, different parts of the same LM will be available to the logic in different clock cycles. Furthermore, if two sub-rectangles contain different bits of the same word of an LM and get mapped to the same port, more than one read will be required to access the word completely. Our assumption excludes such scenarios without deteriorating the solution.

4.1 Problem Definition

Given

- **Set** $\mathcal{L} = \{l : logical\_memory\}$. $\mathcal{L}$ specifies the input design.
- **Set** $\mathcal{T} = \{t : memory\_type\}$.
- **Set** $\mathcal{P} = \{pm : physical\_mem\_instances | \forall pm : \exists_1 t \in \mathcal{T}\}$. $\mathcal{T}$ and $\mathcal{P}$ are part of the target architecture.
- If logic partitioning information is available, the target architecture specification will also have num_fpgas and **set** $\mathcal{I} = \{i_{f_1 f_2} | 0 \leq f_1, f_2 < num\_fpgas\}$. Each element of $\mathcal{I}$ specifies the number of interconnect pins available between the corresponding FPGA pair.

The objective is to produce another set
$\mathcal{M} = \{m : memory\_mapping \mid \forall l \in \mathcal{L}, \exists_1 l \leftrightarrow m\}$ **such that**
$\{\forall pm \in \mathcal{P}, satisfies(Constraints_{pm})\}$ and $\{\forall i \in \mathcal{I}, satisfies(Constraints_i)\}$

4.2 Memory Mapping Adaption for Tabu Search

The TS operates on an array (of size #LM) of mappings.

- **Neighborhood Moves:** Since the neighborhood space is very large and totally random, we randomly choose a small number of LMs. We heuristically re-map the chosen LMs while mappings for the remaining LMs are retained.
- **Tabu Attributes:** We consider two attributes for each LM re-mapped during an iteration of TS: *from_bank* and *to_bank* are *memory_types* to which the LM was mapped before and after the move. Any move which contemplates to undo a recently made move (by moving the LM to the *from_bank* or from the *to_bank*) will be tabued for the next few iterations, called the *tabu tenure*. Tabu status can be over-ridden if the contemplated solution is better than the best encountered so far (*aspiration criterion*).
- **Residence and Transition Frequency:** These are long-term memories of the TS. Residence Frequency stores the total number of iterations for which a LM was mapped to a bank type. It indicates the suitability of mapping the LM to that bank type. Transition Frequency holds the number of times a

LM was re-mapped from one bank type to another. LMs with high transition frequency generally are smaller in size and their re-mapping causes localized search. Suitably rewarding or penalizing high residence and/or transition frequency can lead to search *intensification* or *diversification* respectively.

– **Restart:** This is a form of medium term memory. After a fixed number of iterations, an average cost of all the solutions explored in the current region is compared with the best cost found so far. If the average cost is significantly higher, chances of finding a new best solution in this region are very small. All memories are reset at restart, and search is restarted with a new random solution.

5 Heuristic: Rectangle Carving

At every iteration, Tabu Search decides which LMs are to be re-mapped and onto which memory type. However, the new mapping for an LM on its memory type is found by a heuristic called *Rectangle Carving* (Figures 1 and 2). The algorithm reads in an LM as a rectangle. It maintains a list of sub-rectangles which are to be mapped. As part of a sub-rectangle is mapped, new ones are carved out of its unmapped part and added to the list. The algorithm continues until all sub-rectangles have been mapped.

```
Algorithm: Map_LM
   Input:
      M : Logical Memory
      B : Bank Type
   Output:
      S : Solution, a set of mapping to ports
begin
   C, R, N : Rectangle
   L :List of Rectangles to map
   P :Physical Port
   U, FORCE :Bool
   R ← Rectangle(M.depth, M.width);
   S ← ∅;
   L ← [R];
   /*initialize L with the rectangle corresponding to M */
   while (L ≠ ∅) loop
      R ← L.first();
      P ← random_port(B);
      U ← port_usable(P, M);
      FORCE ← (max_fails_reached);
      C ← carve_rectangle(R, P, FORCE);
      if (FORCE or (is_valid(C) and U) ) then
         gen_new_subrect (R, C, L);
         L ← L − [R];
         S ← S ∪ {C};
      end if
   end while
end
```

Fig. 1. *Algorithm Map LM*

```
Algorithm: Carve_Rectangle
   Input:
      R :Rectangle to map
      P :Physical Port
      FORCE :Bool
      /*indicates if mapping to be done forcefully*/
   Output:
      C :Rectangle
      /* Carved from R, can be mapped onto P */
begin
   Fitness :Array of fitness, of size #configs
   N :A subset of Port Configurations
   N ← {n|n ∈ Configurations of P
      ∧n.width ≥ P.curr_config.width}
   for each(n ∈ N) loop
      get_assignable_rect(R, P, n);
      Fitness_n ← func(assignable_depth,
         assignable_width, R);
      save_best_assignable_rectangle();
   end for
   if (BestFitness > 0) then
      C ← Rectangle(best_assignable_depth,
         best_assignable_width);
      P.curr_config ← best_config;
   else
         /* Failed to carve out a rectangle */
      if (FORCE) then
         /* Forcefully assign R to C */
         C ← R;
      else
         /* Try in future iterations */
         C ← NULL;
      end if
   end if
end
```

Fig. 2. *Algorithm Carve Rectangle*

In each iteration, the algorithm tries to map the first sub-rectangle in the list. It randomly picks one of the ports of an instance of a given bank. *port_usable*

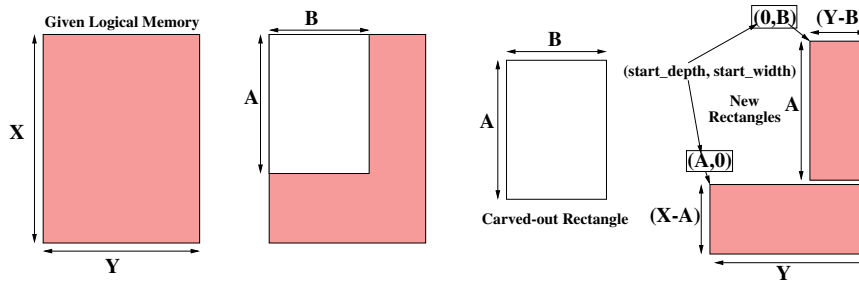

Fig. 3. *Rectangle Carving Process*

performs a check to see if this port has already been assigned to some other sub-rectangle.

Carve_rectangle considers a *subset of configurations* of the selected port and calculates a fitness of mapping the rectangle to each configuration. The fitness depends upon how much of the rectangle can be mapped, whether the rectangle has to be split, and how much storage space would be wasted. We consider only those configurations which have width larger than that required by the already mapped sub-rectangle, if any.

If the carved-out rectangle is smaller than the given rectangle, *at most* two new sub-rectangles, shown in Figure 3, are created out of the unmapped parts. While splitting, we intuitively try to keep all bits of a word in the same rectangle, i.e. depth-wise splitting is preferred over width-wise splitting. The port's configuration is updated to accommodate the new as well as any old rectangles mapped to this port. The overall complexity of mapping an LM to mem_type t is $O\big(\#Ports_t \times (\#LM + \#config_t)\big)$.

6 Control Logic

6.1 Arbitration

If more than one compute_task access a physical port, arbitration logic is required to serialize any parallel accesses. Also, if several LMs are sharing the same port, tasks accessing either of them will have to be arbitered. If an LM is split across ports, all compute_tasks accessing at least one of these ports will have to be arbitered. All ports whose accessors need to be arbitered are combined together to form a *logical port* for arbitration purpose. A scalable arbiter logic for RCs is presented in [13]. Handshake signals (Request/Acknowledge) are introduced between the arbiter and all corresponding arbitered tasks.

6.2 Address Translation and Enable Logic

An address translation mechanism is required to handle the *mismatch* between the *logical address* coming from the task and the *physical address* location where

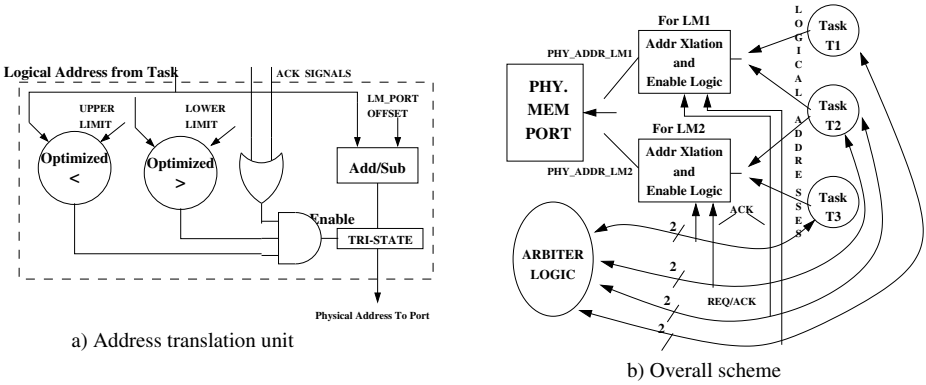

Fig. 4. *Overall Scheme*

that word is placed. If multiple LMs are sharing a port, then only one LM can be placed starting at physical address location 0 of the port. All other LMs will be placed at an *offset*.

One address translation unit (Figure 4a) is required for each *LM-port* pair. *Logical_starting_addr* is the address of the first word of the sub-rectangle of the LM mapped to this port. *Physical_starting_addr* is the address of the physical location where the sub-rectangle is mapped. The difference between the two is added as *lm_port_offset* to the logical address to obtain the actual physical address. Note that the offset value can also be negative. The translated address is given to the input of a tri-state buffer, output of which goes directly to the address port. The enable of this tri-state is controlled by the *lm_port_enable* signal. To assert it, at least one task should be accessing the LM. This is checked by OR*ing* the acknowledge signals generated by the arbiter for tasks accessing the LM. In addition, the logical address coming from the task should be in the range which has been mapped to this port.

6.3 Putting It Together

As shown in Figure 4b, each compute task has address/data ports for each LM it accesses. We assume that when a task is not accessing a particular LM, its corresonding ports are tri-stated. In the case of simultaneous accesses to an LM, the arbiter ensures that only one of the accessing tasks drives the input. If this address is valid for the port and the tri-state logic is enabled, the translated address is sent to the address port. The data bus is directly connected between all accessing tasks and the data port.

7 Constraints

Architectural constraints are dictated by the RC onto which the design has to be mapped.

- **Physical Memory Sizes:** The mapper should ensure that the sum of bits consumed by all LMs assigned to a physical instance does not exceed the maximum capacity of that instance.
- **Interconnect Constraints:** The mapper should ensure that there are enough pins available between each FPGA pair for routing all the address, data, and arbitration signals.
- **Mapping Constraints:** Mapping Constraints are a reflection of the limitations in the ways in which an LM can be mapped. We require that two sub-rectangles of the same LM should not be mapped to the same port.

Design Constraints should be looked upon as broad guidelines being provided by the user based on design requirements. *Sharing of ports between LMs* can lead to better utilization of storage capacity. It also leads to address and data bus sharing between various logical memories. However, the number of compute tasks which need to be arbitered might increase. This will introduce additional delay in memory access, extra arbitration signals to be routed and a bigger area requirement for the arbiter. The user can specify a *Latency Constraint* in the form of an upper limit of the overall read/write latency of the design.

8 Cost Function and Estimation

Tabu Search evaluates a solution by calculating a cost of the current solution. The overall cost function is a weighted sum of the following factors:

- **Latency:** The total number of clock cycles required to perform all read and write operations.
- **Arbitration Cost:** The number of tasks which need to be arbitered at any *logical port* is equal to the sum of number of tasks accessing the LMs. Arbiters of different sizes are pre-synthesized and their areas used while evaluating a solution.
- **Clock Frequency:** Any signal which needs to be routed across chips becomes the bottleneck in operating at high frequency. We take the maximum number of pins traversed by any signal in the design to be an indication of maximum operating clock frequency of the design.
- **Blocks Processed:** Processing multiple sets of data without reconfiguring the device involves introducing a counter for address offset. This is done by packing multiple sets of data in the same memory space. Unused memory instances of the same type can be grouped with used instances to pack more sets of data.
- **Address Translation and Enable Logic:** The size of the logical address bus as well as that of the offset is known. Thus the total area required can be estimated.

Design Name	#LM	Total Words	Exec Time (sec)
DCT1	15	112	17.9
FFT	12	80	15.0
DCT2	10	48	15.4
Laplace	14	509	17.0
MeanValue	13	1200	17.4
LUD	13	1343	17.4
Rand100	100	3513	142

Table 1. *Memory Mapping Results*

Des Num	Num LM	Num Ports	ILP Cost	Heu. Cost	% Diff.
1	8	18	422	422	0.0
2	18	39	770	777	0.9
3	32	63	1292	1319	2.0
4	27	29	1302	1381	5.7
5	27	37	1327	1411	5.9
6	39	95	1584	1622	2.3
7	42	77	1841	1872	1.6
8	49	99	1997	2280	12.4
9	18	52	3139	3143	0.1
10	32	60	4702	4931	4.6

Table 2. *ILP versus Heuristic Results*

– **Address and Data Bus Routing:** Since only one LM can be accessed through a logical port at a time, sharing of the address and data buses between LMs is done. For each logical port, the maximum address and data width required is calculated based on the biggest and widest memory present in it.

9 Results

Table 1 shows results of memory mapping for some benchmark examples. The target architecture is assumed to have one FPGA. There are 3 instances of on-chip memories of size 4096 bits each, with 5 configurations varying in width from 1 to 16 and a read/write latency of 1 clock cycle each. There is one instance of off-chip memory having 2K words, 16-bit wide, single-ported with read latency of 3 and write latency of 1. The total number of words in all LMs is given in Column 3. The last column shows the execution time of the heuristic approach on a SUN Sparc station running at 336MHz with 1344MB of RAM.

Table 2 presents comparison between ILP [10] and heuristic approaches. Designs are characterized by the number of LMs and target architectures by the total number of ports available over all instances of physical memories. As expected, the ILP approach gives better results in terms of lower cost function. However, in many cases, even though a solution was found, the ILP approach did not converge while the heuristic execution time was very small (52 seconds for the largest design). On average, the cost of the heuristic solution is within 3.5% of the ILP cost.

Figure 5 shows different runs of the mapper for the same set of designs and target architectures. Each graph displays two sets of data, one for which sharing of port between LMs was permitted and the other in which it was not. In the portsharing case, bigger logical_ports are formed, leading to sharing of address and data pins between various LMs. This also enabled the mapper to find a constraint satisfying solution very quickly. In fact, for the last 3 designs,

no-portsharing constraint prevented the mapper from finding a constraint satisfying solution. We also observe that in each case where a solution was found, portsharing gives lower read/write latencies; this is expected as more LMs are packed into faster physical memories.

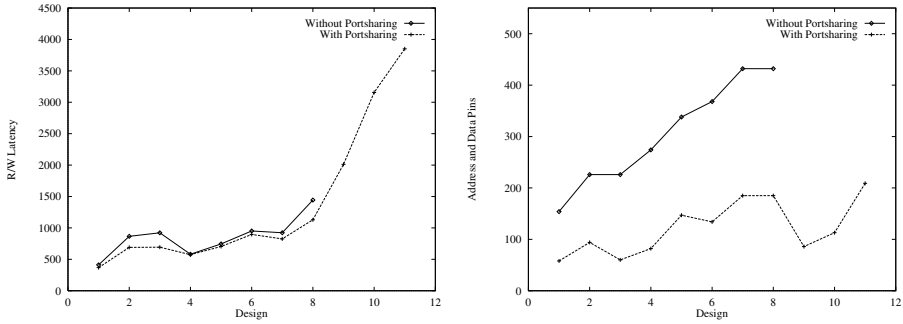

Fig. 5. *Read/Write Latency and Pins Utilized*

References

1. Xilinx Corporation. *"Using Virtex BlockRAMs"*, 1999.
2. Altera Corporation. *"FLEX 10K Embedded Programmable Logic Family Data Sheet"*, May 2000.
3. Altera Corporation. *"APEX 20K Programmable Logic Device Family Data Sheet"*, March 2000.
4. P. Jha and N. Dutt. "High-Level Library Mapping for Memories". In *ACM Transactions on Design Automation of Electronic Systems*, pages 566–603. ACM Press, July 2000.
5. M. Balakrishnan. "Allocation of Multiport Memories in Data Path Synthesis". In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, volume 7*, pages 536–540, April 1998.
6. I. Ahmad and C. Y. Chen. "Post-Process for Data Path Synthesis". In *Proceedings of International Conference on Computer Aided Design*, pages 276–279. ACM Press, 1991.
7. D. Karchmer and J. Rose. "Definition and Solution of the Memory Packing Problem for Field-Programmable Systems". In *Proceedings of International Conference on Computer Aided Design*, pages 20–26. ACM Press, November 1994.
8. S. Wilton. *"Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory"*. PhD thesis, University of Toronto, 1997.
9. W. Ho and S. Wilton. "Logical-to-Physical Memory Mapping for FPGAs with Dual-Port Embedded Arrays". In *International Workshop on Field Programmable Logic and Applications*, pages 111–123, September 1999.
10. I. Ouaiss and R. Vemuri. "Hierarchical Memory Mapping During Synthesis in FPGA-Based Reconfigurable Computers". In *Design Automation and Testing*

Conference of Europe, pages 284–293, Berlin, Germany, September 2000. Springer-Verlag.
11. I. Ouaiss and R. Vemuri. "Global Memory Mapping During Synthesis in FPGA-Based Reconfigurable Computers". In *Reconfigurable Architectures Workshop*, pages 284–293, San Francisco, September 2000. Springer-Verlag.
12. F. Glover and M. Laguna. *"Tabu Search"*. Kluwer Academic Publishers, 1997.
13. I. Ouaiss and R. Vemuri. "Resource Arbitration in Reconfigurable Computing Environments". In *Proceedings of Design Automation and Test in Europe*, pages 560–566. IEEE Computer Society Press, April 2000.