Global Memory Mapping for FPGA-Based Reconfigurable Systems*

Iyad Ouaiss and Ranga Vemuri
Digital Design Environments Lab, University of Cincinnati
Cincinnati, OH 45221-0030, USA
{iouaiss, ranga}@ececs.uc.edu

Abstract

Synthesizing designs for FPGA-based reconfigurable systems involves the task of mapping variables and data structures of the application onto RAMs of the reconfigurable board. The variety in types and performance of on-board and on-chip RAMs, their proximity to the processing units, and the interconnection scheme of the reconfigurable system, all contribute to an intricate memory mapping problem. An intelligent memory assignment minimizes the total latency of the design and the interconnection requirements due to memory accesses. A complete Integer Linear Programming (ILP) formulation of the problem results in an optimized memory mapping; however, the formulation is complex and takes a very long time to produce a solution. In order to efficiently solve the problem, the concept of global/detailed memory mapping is introduced in this paper. An ILP formulation of the global mapping process is described. This formulation is simpler and faster than the complete formulation, and it leaves the task of detailed mapping to a post-ILP tool that does not affect the optimality of the memory assignment. As a result, larger designs can be handled at a faster rate and more constraints can be introduced to the formulation.

1 Introduction

Whether an application is to be implemented as an ASIC or on an FPGA, variables in the design have to be assigned onto physical memory banks and operations onto hardware logic. In both technologies, memory mapping is the task of assigning each data structure in the design to one or more physical RAM. Mapping schemes vary widely in complexity: A simple scheme might not consider splitting a data structure across two physical banks or might cater solely to single-port memories. A more complex scheme might dynamically partition a data structure onto several possibly different memory banks, might support overlapping of data structures onto the same physical memory space, or might take into account the application's access patterns to the data structures, while performing the assignment.

With signal and image processing applications, memory mapping becomes a crucial step in the synthesis process: The performance of these data-intensive applications is heavily affected by the quality of the memory assignment. In image and speech processing aplications, physical RAMs can easily occupy more than half the ASIC implementation. Thus, if data structures are not cleverly mapped, congestion in routing and memory access degradation can occur.

For ASICs, memory mapping consists of selecting memory components from a library, selecting where the components are placed, and selecting the way in which they are connected to the hardware logic. Whereas in the case of FPGAs, and of reconfigurable computing (RC) systems in general, the mapping consists of assigning the data structures to a fixed hardware platform. The ASIC scenario assumes that the memory banks are picked to match the data structures of the application; whereas in the RC scenario, the data structures are manipulated to fit on the pre-existing physical banks.

RC boards with modern FPGAs not only have off-chip physical banks but also offer a large number of on-chip memories. Hence, due to the abundance of physical memory banks as well as the amount and importance of data structures in signal processing algorithms, it becomes difficult to manually perform memory mapping. The on-chip memory banks of Xilinx Virtex devices [18], called *BlockRAMs*, vary from 8 BlockRAMs for the XCV-50 device up to 208 BlockRAMs for the XCV-3200E device. On-chip memory banks of Altera FLEX 10K devices [2], called *Embedded Array Blocks (*EABs*)*, vary from 9 EABs for the EPF10K70 device up to 20 EABs for the EPF10K250A device. On-chip memory banks of Altera APEX E devices [1], called *Embedded System Blocks (*ESBs*)*, vary from 12 ESBs for the EP20K30E device up to 216 ESBs for the EP20K1500E device. Table 1 summarizes the onchip RAMs

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

Device	RAM Name	RAMs (# banks)	Size (# bits)	Configurations
Xilinx Virtex	BlockRAM	8 → 208	4096	4096x1 2048x2 1024x4 512x8 256x16
Altera Flex 10K	Embedded Array Block	9 → 20	2048	2048x1 1024x2 512x4 256x8 128x16
Altera Apex E	Embedded System Block	12 → 216	2048	2048x1 1024x2 512x4 256x8 128x16

Table 1. FPGA On-chip RAMs

available in today's FPGAs.

In addition to the number of available RAMs on an FPGA, the number of memory ports of each on-chip memory bank could be greater than one. Both Xilinx and Altera devices offer dual-ported memories, each port accessing the same physical space. Finally, the depth/width ratio of each memory bank could be variable. Both Xilinx and Altera devices have five configurations depicted in Table 1.

Little work has been done to automatically assign data structures to complex RC systems; furthermore, features such as multiple configurations per memory bank, have not yet been incorporated in the synthesis process. This paper proposes a modification to memory mapping: the process is divided into *global mapping* followed by *detailed mapping*. An ILP formulation is used to optimize the solution by performing global mapping. Since detailed mapping does not affect the quality of the assignment, it can be performed after global mapping hence reducing the complexity of the ILP formulation.

The rest of this paper is organized as follows: Section 2 discusses previous work performed in memory mapping. Section 3 describes the inputs to the environment in which memory mapping is performed. Section 4 depicts the Integer Linear Programming approach for memory mapping; it first briefly states the *complete* ILP formulation and then introduce the global v/s detailed memory mapping paradigm. Section 5 shows some results obtained. Finally, Section 6 concludes the paper and presents future work.

2 Previous Work

The majority of the memory mapping studies focus on the ASIC implementation. The tools pick a set of physical memory modules from a library of available banks and select the interconnection structure to connect the processing units to the memory banks. Very few studies target reconfigurable boards where memory banks and interconnection structure are fixed *before* synthesizing the application.

In ASIC implementations, since the hardware is custom built, the aim is to *minimize* the interconnection cost and the number of required physical memory banks. However, with on-chip memory and fixed external memory banks and interconnection structures in RC systems, the problem is different. Off-chip interconnections are reduced when using on-chip memory. Furthermore, given a fixed memory structure on an RC board, minimizing the number of required memory banks might not produce the optimal solution; As long as the mapper does not exceed the physical storage area, it should be allowed to use as many banks as it sees fit.

Integer linear programming models were used in [8, 11] to group registers and form multi-port memory modules in ASIC implementations. Several studies were conducted in the realm of high-level synthesis where cliques of the design variables are partitioned to form data segments. Some researchers performed this task without taking into consideration the interconnection structure of the hardware [3], while others consider the cost of interconnection during variable grouping for multi-port memory banks [16].

In [15], memory mapping for FPGAs with on-chip memories is addressed; however, only single-ported memory banks are assumed. The same technique was improved in [17] so that the mapping caters to recent FPGAs containing dual-ported on-chip banks. In both works, the focus is on hardware containing a single type of memory banks (either single or dual ported) and does not simultaneously consider off-chip memories that exhibit different performance numbers.

Memory access optimizations are targeted in [12], where the goal is to optimize memory accesses in pipelined designs. Synthesis transformations take advantage of on-chip RAMs and intelligent schedules are produced to maximize parallel accesses. Therefore, memory mapping takes place *during* synthesis and does not address hierarchical memory banks.

Given data structures and access constraints to these structures, [5] finds a legal packing of the logical segments into the physical segments while minimizing the area. Again, since the storage area in the RC framework is fixed, it might not be beneficial to minimize the occupied area.

An analysis of several memory mapping studies is presented in [13]. The authors compare the techniques based on the number and the type of logical memories and physical banks considered simultaneously. In addition, the book by Catthoor et al. [6] and the book by Panda, Dutt, and Nicolau [14] provide an excellent source for topics in memory system optimization, exploration, and management.

In a previous report [9], we implemented a memory mapping technique that performs a complete memory assignment in a single step. However, the formulation becomes

quite lengthy and the solution time explodes for large problems. Instead of this "flat view" solution, this paper divides the logical-to-physical memory mapping process into two steps: first, *global memory mapping* assigns each data structure in the application to one *type* of physical memory banks. A bank type refers to a collection of physical memories that share the same architectural properties and the same access performances. Second, *detailed memory mapping* performs the lower-level assignment; it restructures the data segments and assigns them to specific banks of the type that was dictated by global mapping.

Since all banks of the same type share the same performance and architecture specifications, detailed memory mapping does not affect the overall memory mapping cost. The optimization goal is thus sought after during global memory mapping, and the complexity of the global mapping is reduced by avoiding detailed mapping. By reducing the size of the problem, an ILP formulation becomes simpler and the solution is obtained faster. As a result, designs with several hundred data structures and memory banks are efficiently handled.

3 Problem Formulation

Several features exist for different types of RC boards. This section generalizes the approach of memory mapping by targeting a flexible hierarchical memory structure.

A memory mapper must take as an input both the architecture of the target RC board as well as a description of the design to be synthesized and mapped onto the board. For this paper, it is assumed that the RC board contains only one processing unit. As part of future enhancement, this work will be extended to multi-processing units where logic placement and pin constraints during routing will be addressed.

3.1 Architecture Description

As introduced in Section 2, The RC board architecture is described by a collection of *memory types*. There could be several instances of each memory type, but all instances share the same storage and access speed specifications, and share the same proximity and ease of access from the processing unit.

For each memory type, a *number of instances* tells the mapper how many instances of each type exist on the board. The *number of ports* of a type is one if the memory is a single-ported memory, two if dual-ported, etc. As shown in Figure 1, the depth/width ratio of a memory could be variable; The *number of configurations* for each type is the number of possible settings of each port of that type.

The *number of words* (depth) and the *number of bits per word* (width) of a type are unique numbers if only one configuration exists. Otherwise, these are equal-length lists of numbers describing the possible configurations of the depth/width ratio. Entry *i* of the depth list together with entry *i* of the width list correspond to configuration *i*. It is assumed that the capacity of each configuration is a constant.

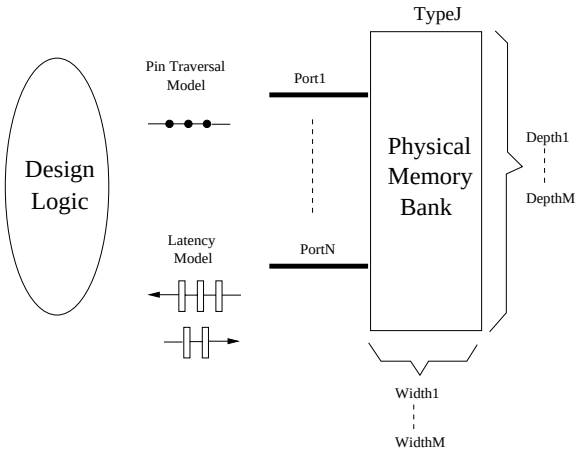

Figure 1. Generic Memory Bank

The access latency for each type of memory is variable; The *read latency* is the number of clock cycles required after performing a read and before getting valid data out of the memory bank. Similarly, the *write latency* is the number of clock cycles required after performing a write operation and before the data is correctly stored in the memory bank.

Finally, with respect to the physical location of the memory bank, the *number of pins traversed* depicts the proximity of the physical memory bank to the processing unit. If a bank is on-chip, zero pins are traversed. If an off-chip bank is directly connected to the memory, two pins are traversed. If an indirect connection exist between the processing unit and the external memory bank, then additional pins are traversed. In general, the aim is to map data structures to physical banks that are as close as possible to the processing unit; The further away they are, the larger the impact on the overall memory access performance.

A generic physical bank is shown in Figure 1. The latency model captures the number of read and write clock delays and the pin traversal model captures the number of pins traversed between the processing unit and the memory bank. In the Figure, bank type *J* is an *N*-ported memory, has *M* different configurations: depth*1*/width*1*, depth*2*/width*2*, ..., depth*M*/width*M*.

3.2 Task graph Description

On the design side, a description of the data structures is required. Since this work focuses on placing the data structures on the physical banks, it is assumed that the structures

are already formed.

For each data segment in the design, the *number of words* (depth) in the segment and the *number of bits per word* (width) are required. A footprint analysis of the memory accesses could tremendously help in guiding the mapping process: e.g. data segments that are extensively accessed should be assigned to faster and closer physical banks.

3.3 Conflict Description

During synthesis of a design, scheduling determines the *life times* [7, 4] of the variables and data structures. This life cycle analysis could further improve the memory mapping since segments that can overlap could be placed in the same storage area, thus decreasing the total storage requirement. For this purpose, the mapper needs to know which data segments life cycles overlap. A set of *conflicting pairs* captures this requirement; Pair (L1, L2) means that data segment L1 cannot share storage space with segment L2.

4 ILP Formulation

For the formulation presented in this section, we assume the following notation. There are M data structures:

$DS = \{DS_1, DS_2, ..., DS_M\}$ to be mapped onto N different types of physical memory banks:

$PB = \{PB_1, PB_2, ..., PB_N\}$.

There could be multiple instances of each type of memory bank.

For each logical data structure d, we have:

$$\begin{cases} D_d & \text{Number of words in segment } d. \\ W_d & \text{Number of bits per word in segment } d. \end{cases}$$

For each type of physical memory bank t, we have:

$$\begin{cases} I_t & \text{Number of banks of type } t. \\ P_t & \text{Number of ports in a bank of type } t. \\ C_t & \text{Number of depth/width configurations in a bank of type } t. \\ D_t & \text{Array of number of words in a bank of type } t. \\ W_t & \text{Array of number of bits per word in a bank of type } t. \\ RL_t & \text{Read latency in number of clock cycles.} \\ WL_t & \text{Write latency in number of clock cycles.} \\ T_t & \text{Number of pins traversed from the processing unit to a bank of type } t. \end{cases}$$

where the depth/width ratio variables are:

$D_t = \{d_1, d_2, ..., d_{C_t}\}$ and $W_t = \{w_1, w_2, ..., w_{C_t}\}$.

Finally, there are Q conflict pairs in the design where each associates two logical structures:

(DS_x, DS_y), where $x \neq y$.

Finally, the remaining notations pertain to the 0-1 variables used in the model. Z_{dt} associates a data structure to a memory type:

$$Z_{dt} = \begin{cases} 1 & \text{if data structure } d \text{ is assigned to some instance of bank type } t. \\ 0 & \text{otherwise.} \end{cases}$$

Z_{dt} is used to force an oversized data structure to be split across banks of the *same type*. Similarly, X_{dtip} associates a data structure to a specific physical bank:

$$X_{dtip} = \begin{cases} 1 & \text{if data structure } d \text{ is assigned to port } p \text{ of instance } i \text{ of bank type } t. \\ 0 & \text{otherwise.} \end{cases}$$

And, only for multi-configuration physical banks (i.e. $C_t > 1$), Y_{tipc} sets a specific configuration to a port of a memory bank:

$$Y_{tipc} = \begin{cases} 1 & \text{if configuration } c \text{ is selected for port } p \text{ of instance } i \text{ of bank type } t. \\ 0 & \text{otherwise.} \end{cases}$$

4.1 Global Memory Mapping

Global mapping only considers the task of assigning a data structure to exactly one type of memory bank. It does not deal with the assignment of the data structure to *specific* instances and ports of the type. However, global mapping ensures a successful detailed mapping by taking into account the architecture specification while avoiding non-optimizing factors in the formulation.

While a complete memory mapper makes use of all three X_{dtip}, Z_{dt}, and Y_{tipc} parameters, a global memory mapper requires only the Z_{dt} parameter. Z_{dt} assigns a data structure to a memory type, whereas X_{dtip} and Y_{tipc} assign data structures and configurations to specific bank instances.

The execution time savings obtained by using a global mapper could be lost if the detailed mapper fails. If this occurs, the global and detailed mappers need to execute multiple times until a solution is found. Thus, it is very important to ensure that the global mapper produces an assignment that can be successfully detailed mapped. In an ILP formulation, this translates to having constraints in the global mapper that take into account the number of instances of each type of memory banks, the number of ports of each instance, and the available width/depth configurations of the type.

4.1.1 ILP Pre-processing

For each design being mapped, the global mapper initially pre-processes some information in order to produce an ILP formulation of the problem. This formulation will result in

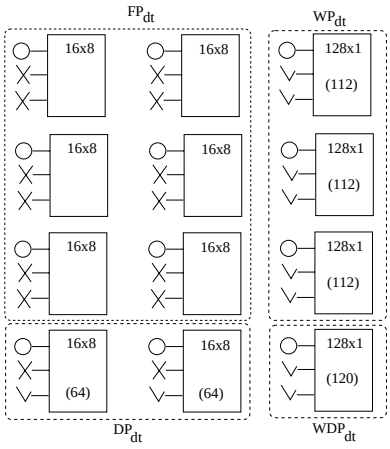

Figure 2. Space and Ports Allocation Example

a fast ILP solution that will successfully go through detailed mapping.

Three main parameters need to be computed to allow a simple yet powerful constraint formulation: CP_{dt} or the total number of consumed ports of memory type t if data structure d is assigned to it. CW_{dt} or the "ceiling" value of the width of data structure d if assigned to bank type t. And finally, CD_{dt} or the "ceiling" value of the depth of data structure d if assigned to bank type t.

First, the total number of consumed ports CP_{dt} depends on the size of data structure d with respect to the size and number of ports of bank type t. There are four components to CP_{dt} and they are illustrated by the following example.

A 55x17 data structure is to be mapped onto one type of memory bank that has 3 ports, and four ratio configurations: 128x1, 64x2, 32x4, 16x8. Since the data structure requires more than one instance, Figure 2 shows a mapping where the assigned instances can be visualized as a rectangular area. The width, 17, of the data structure will be divided into: 8, 8, and 1; and the depth will thus have to be 16 to match the 16x8 configuration. Hence, the upper left instances in the rectangle are *fully* utilized with configuration 16x8 selected. The upper right instances, that form a single column, are *partially* utilized with configuration 128x1 selected since there remained one bit from the width of the structure. The lower left instances, that form a single row, are *partially* utilized with configuration 16x8 selected. Finally, the lower right instance, that is a single instance, is *partially* utilized with configuration 128x1 selected. Note also that, since the memory type has 3 ports, *all* ports in the upper left instances are consumed, and *some* ports in all other instances are consumed. The "O" next to a port signifies that the port is used by this data structure. The "X" next to a port denotes that the port is wasted. And, the check mark next to a port indicates that the port is available for other data structures. Finally, the number between parentheses next to available ports indicates the total number of bits that are still unused in the instances.

Thus, following the scheme of Figure 2:

$$\forall d \in DS, \quad \forall t \in PB,$$

$$CP_{dt} = FP_{dt} + WP_{dt} + DP_{dt} + WDP_{dt}$$

where

$$FP_{dt} = \left\lfloor \frac{D_d}{D_{t\alpha}} \right\rfloor * \left\lfloor \frac{W_d}{W_{t\alpha}} \right\rfloor * P_t$$

α refers to the configuration with the smallest width such that $W_{t\alpha}$ is greater than or equal to W_d. If W_d is larger than all configuration widths, then α is the configuration with the largest width. For multi-configuration banks, the best configuration yields the smallest numbers of instances used while trying to match the width of the bank with the width of the data structure.

if $((W_d mod W_{t\alpha}) == 0)$ then $WP_{dt} = 0$ else:

$$WP_{dt} = \left\lfloor \frac{D_d}{D_{t\alpha}} \right\rfloor * consumed\_ports\left(D_{t\alpha}, D_{t\beta}, P_t\right)$$

where β refers to the configuration with the smallest width such that:

$$W_{t\beta} \geq W_d mod W_{t\alpha}$$

consumed_ports() is defined in Figure 3. And:

$$DP_{dt} = \left\lfloor \frac{W_d}{W_{t\alpha}} \right\rfloor * consumed\_ports\left((D_d mod D_{t\alpha}), D_{t\alpha}, P_t\right)$$

Finally, if $((W_d mod W_{t\alpha}) == 0)$ then $WDP_{dt} = 0$ else:

$$WDP_{dt} = consumed\_ports\left((D_d mod D_{t\alpha}), D_{t\beta}, P_t\right)$$

The second parameter, CW_{dt}, indicates the total width that is consumed by data structure d if assigned to bank type t. After finding configuration α and β as described above:

$$CW_{dt} = \left\lfloor \frac{W_d}{W_{t\alpha}} \right\rfloor * W_{t\alpha} + W_{t\beta}$$

Similarly, the third parameter, CD_{dt}, indicates the total depth that is consumed by data structure d if assigned to bank type t:

$$CD_{dt} = \left\lfloor \frac{D_d}{D_{t\alpha}} \right\rfloor * D_{t\alpha} + \lceil D_d mod D_{t\alpha} \rceil_{pow(2)}$$

For data structures to co-exist on the same instance of a bank type (on different ports of the bank), $consumed\_ports()$ is used to compute the fractional number of ports consumed by a data structure. If the entire data structure fits on a single instance, then $consumed\_ports()$ actually is the total number of ports consumed by the data

function *consumed_ports*(D_d, D_t, P_t)
begin
$depth = round(D_d, pow(2))$
$fraction = \frac{depth}{D_t}$
$EP = \lceil fraction * P_t \rceil$
returns EP
end

Figure 3. Fractional Port Consumption

structure; However, when multiple instances are required, *consumed_ports*() returns the number of ports consumed on each used instance.

Thus, for an n-ported memory, the algorithm in Figure 3 computes *consumed_ports*(). The main purpose of this algorithm is to make sure that once data structures are mapped onto physical banks, no adders or other logic would be required to perform memory accesses. To do so, each assigned fraction in an instance is rounded to the closest power-of-two depth that corresponds to the configuration with the largest width. In addition, the port assignment follows the order of decreasing fraction sizes. This ensures that several fractions can be assigned to different ports of an instance without the need of extra logic for base address generation. It also ensures that the memory space for each fraction is mutually exclusive, thus avoiding unwanted conflicts.

Note that *consumed_ports*() in Figure 3 is optimal for $P_t = 2$. There is a waste of ports when $P_t > 2$; but since the majority of on-board and on-chip memory banks are either single or dual ported, this problem is not very pronounced currently. Improvement to this algorithm is part of future work.

Hence, a multi-ported memory bank can only be divided in a fixed number of ways. For instance, the general space allocation for a 3-port memory with 16-word depth is shown in Table 2. The algorithm in Figure 3 rejects the (8, 8, 0) configuration since it estimates that 8 words require two ports each thus requiring a total of 4 ports. This overestimation does not occur when a bank type has only two ports.

4.1.2 Constraint Formulation

Next, the ILP formulation is constructed based on the following constraints:

- **Uniqueness constraints:** Each data structure should be mapped to exactly one *type* of physical bank:

$$\forall d \in DS, \quad \sum_{t \in PB} Z_{dt} = 1$$

3-port 16-word bank		
Port 1 (# words)	Port 2 (# words)	Port 3 (# words)
16	0	0
8	8	0
8	4	4,2,1,0
8	2	2,1,0
8	1	1,0
8	0	0
4	4	4,2,1,0
4	2	2,1,0
4	1	1,0
4	0	0
2	2	2,1,0
2	1	1,0
2	0	0
1	1	0,1
1	0	0
0	0	0

Table 2. Example on Allocation Options

- **Port constraints:** Each memory type should have enough ports for all the data structures assigned to it:

$$\forall t \in PB, \quad \sum_{d \in DS} Z_{dt} * CP_{dt} \leq P_t * I_t$$

From the pre-processing step, the number of ports that each data structure would consume from each type of memory is computed. The sum of all consumed ports of a type should be less than or equal to the total number of available ports of the type.

- **Capacity constraints:** Each bank type must have enough space to contain all data structures assigned to it.

$$\forall t \in PB, \quad \sum_{d \in DS} Z_{dt} * (CW_{dt} * CD_{dt}) \leq I_t * W_{t[1]} * D_{t[1]}$$

In the event where the life-cycles of different data structures do not conflict, the capacity constraint is slightly modified to allow overlapping in the memory space.

Note: The uniqueness constraint ignores the specifications of the memory banks; whereas the port and capacity constraints cater to the specific features of each type. It is these latter constraints in addition to some parameters computed in the pre-processing step, that ensure successful detailed mapping.

4.1.3 Objective Formulation

The objective of the ILP model is to optimize the performance and minimize the interconnection cost of the memory assignment. The cost function takes the form:

$$minimize[Cost_1 * \alpha_1 + Cost_2 * \alpha_2 + ... + Cost_n * \alpha_n]$$

where α_i is a weight coefficient used to normalize $Cost_i$ with respect to all other cost components.

Three cost components are depicted below.

- **Latency cost:** Assuming the number of reads is equal to the number of writes for every data structure:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * D_d * [RL_t + WL_t]$$

- **Pin delay cost:** Assuming the number of pins traversed from the processing unit to reach the memory bank is inversely proportional to the clock speed:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * D_d * T_t$$

- **Pin I/O cost:** The larger the depth and width of a data structure the more pins it will need in the event of off-chip physical banks:

$$\sum_{d \in DS} \sum_{t \in PB} Z_{dt} * (\lceil log_2(CD_{dt}) \rceil + CW_{dt}) * T_t$$

4.2 Detailed Memory Mapping

Once global mapping is complete, the task of detailed mapping is to consider each bank type at a time and all data structures assigned to it. It might re-shape the data structures in case they are larger than a single instance of the bank type, it might assign configurations for each port of each bank instance of the type, and it will assign specific instances for each data structure segment. Thus, the entire problem is serialized into one main formulation (global mapping) followed by a collection of formulations (detailed mapping on each bank type).

Since global memory mapping ensures a successful detailed assignment, the task of the detailed mapper is simplified. It cannot further optimize the assignment based on the optimization criteria established in global mapping. Instead, it can aim at optimizing the detailed assignment based on different optimization factors.

An ILP-based formulation for the detailed memory mapper was developed. For brevity, the mathematical formulation is not reproduced in this paper. For every type of memory bank, an ILP problem is formed and solved. The aim is to assign data structures to specific ports of specific instances of the bank, possibly requiring to fragment the data structure to fit on several instances. Optimization factors include trying to reduce on-chip interconnection congestion and reducing data structure fragmentation.

5 Results

The ILP model presented in the previous section was executed for designs of different sizes. Also, a complete memory mapper [9] was executed on the same designs. CPLEX, a commercial linear programming solver [10], was used. Both the number and types of logical segments as well as physical banks were varied. Table 3 shows the execution time for both the complete formulation approach and the global/detailed formulation approach. The platform was a SUN Ultra-30 (248MHz with 128MB RAM) for designs of various sizes: For logical memories, the number of segments represents the main complexity parameter in the ILP formulation. Similarly, for physical memories, the three complexity parameters are: the total number of physical banks, the total number of ports summed over all instances of all bank types, and the total number of possible configuration settings summed over all multi-configuration ports of all bank types. The execution time is given in seconds.

Data Structures	Physical Banks			Complete Approach	Global Approach
#segments	Total #banks	Total #ports	Total #configs	Execution Time (sec)	Execution Time (sec)
22	13	25	50	8.1	7.8
32	23	45	100	29.4	25.3
32	45	77	150	99.3	50.7
42	45	77	150	130.4	59.2
32	65	105	150	172.7	105.1
62	65	105	150	411.0	140.4
32	180	265	375	518.3	216.4
62	180	265	375	1225.0	309.0
132	180	265	375	2989.0	489.0

Table 3. ILP Execution Times

It is clear that for large designs, the complete approach becomes impractical compared to the global/detailed approach. Note that the execution times shown for the global/detailed formulation include all pre-processing steps.

It can be seen that for relatively small designs, the difference in the two approaches decreases. This is because for the global/detailed technique, the setup time required for pre-processing and for ILP-processing becomes the dominating factor.

The plot in Figure 4 gives a visual rendering of the results. The X-axis represents the different design points that are ordered in the increasing size of the problem (corresponding to the increasing row number in Table 3). For the sake of clarity, a line is plotted to connect the data points; it does not represent the performance of the algorithm between the test cases.

6 Ongoing Work

The global versus detailed memory mapping paradigm, introduced in this work, eased the ILP formulation while conserving the quality of the resulting assignment. Large designs can be mapped faster and more efficiently than in the case of complete ILP formulation. In addition, the pre-processing of the data simplified the complexity of the

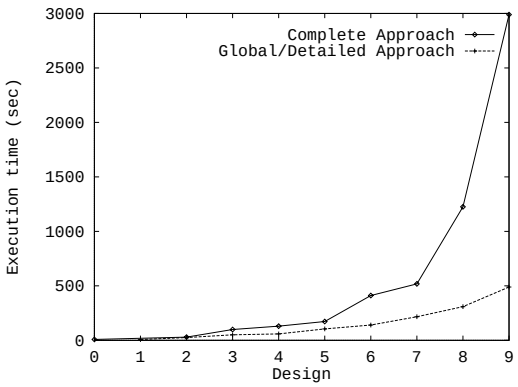

Figure 4. Complete versus Global/Detailed Execution Times

global mapping formulation and ensured a successful detailed mapping.

As part of ongoing and future work, the following issues are being considered. First, the *consumed_ports*() algorithm needs to be improved for memory banks with more than two ports. Second, in the case of a single processing unit, all design logic is mapped onto one hardware area, and all logic areas are assumed equidistant from each physical bank. The model needs to be enhanced to support multiple processing units. Finally, arbitration is not taken into consideration in this paper; in other words, two logical segments will not be mapped onto the same port. In the event of RAM limitation, the model could allow data structures to overlap at the price of adding conflict resolution to the objective function.

References

[1] Altera Corporation. "APEX 20K Programmable Logic Device Family Data Sheet", March 2000.

[2] Altera Corporation. "FLEX 10K Embedded Programmable Logic Family Data Sheet", May 2000.

[3] C. J. Tseng and D. Siewiorek. "Automated Synthesis of Data Paths in Digital Systems". In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 5, pages 379–395, July 1986.

[4] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin. *"High-Level Synthesis, Introduction to Chip and System Design"*. Kluwer Academic Publishers, 1992.

[5] 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.

[6] F. Catthoor, et al. *"Custom Memory Management Methodology"*. Kluwer, 1998.

[7] G. De Micheli. *"Synthesis and Optimization of Digital Circuits"*. McGraw-Hill, 1994.

[8] 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.

[9] I. Ouaiss and R. Vemuri. "Hierarchical Memory Mapping During Synthesis in FPGA-Based Reconfigurable Computers". In *Proceedings of Design Automation and Test in Europe (to appear)*. IEEE Computer Society Press, March 2001.

[10] ILOG Incorporation. *"Using the CPLEX Callable Library"*. http://www.cplex.com.

[11] M. Balakrishnan, et al. "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 1988.

[12] M. Weinhardt and W. Luk. "Memory Access Optimization and RAM Inference for Pipeline Vectorization". In *Proceedings of International Workshop on Field-Programmable Logic and Applications*, pages 61–70. Springer, September 1999.

[13] 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.

[14] P. R. Panda, N. Dutt, A. Nicalau. *"Memory Issues In Embedded Systems-On-Chip"*. Kluwer, 1999.

[15] S. Wilton. *"Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory"*. PhD thesis, University of Toronto, 1997.

[16] T. Kim and C. L. Liu. "Utilization of Multiport Memories in Data Path Synthesis". In *Proceedings of the 30th Design Automation Conference*, pages 298–302. ACM Press, June 1993.

[17] W. Ho and S. Wilton. "Logical-to-Physical Memory Mapping for FPGAs with Dual-Port Embedded Arrays". In *Proceedings of International Workshop on Field-Programmable Logic and Applications*, pages 111–123. Springer, September 1999.

[18] Xilinx, Inc. "Virtex 2.5V Field Programmable Gate Arrays", September 2000.