Balancing FPGA Resource Utilities

Xuejun Liang[*], Jeffrey S. Vetter[**], Melissa C. Smith[**], and Arthur S. Bland[**]
xuejun.liang@jsums.edu, vetterjs@ornl.gov, smithmc@ornl.gov, and blandas@ornl.gov
*Department of Computer Science, Jackson State University, Jackson, MS 39217, USA
**Oak Ridge National Laboratory, Oak Ridge, NT 37831, USA

ABSTRACT: Balancing the use of FGPA resources such as FPGA slices, block RAMs, and block multipliers is desirable in many FPGA applications. This task can be carried out manually by experienced hardware designers with the use of hardware description languages, such as Verilog and VHDL. However, many users of reconfigurable computers are software developers who depend on hardware synthesis tools or even high-level synthesis tools to deal with the details beneath the application logic. In this paper, a motivating example of balancing FPGA resource utilities is given first. A module selection optimization problem is then formulated, in which, balancing FPGA resource utilities is treated as a constraint, so that the solution to the module selection problem is the balanced use of the FPGA resources. Several variations of the problem formulation are discussed. A naïve algorithm and an efficient greedy algorithm to solve the problem are provided and compared. Some experimental results are also presented.

Keywords: FPGA, Reconfigurable Computing, High-Level Synthesis, Module Selection

1. Introduction

Accelerating applications with reconfigurable computers has been successfully shown in many fields, such as digital signal processing, image processing, cryptography, and high-end scientific applications. However, the programming of reconfigurable computers is extremely cumbersome, demanding that software developers also assume the role of hardware designers. Thus, one of the keys to unlocking the full potential of these systems is developing truly automatic and optimal mapping tools. To this end, there have been many research projects on design environments for reconfigurable systems.

The MAP complier for the SRC-6E reconfigurable computer [1], which is based on the Xilinx Virtex-II FPGA [2], can translate a user function in a high-level programming language (C or Fortran) into an FPGA hardware function unit in the Verilog hardware description language [3]. The MAP compiler always binds multiplications in a user function to multipliers that are built based on the FPGA block (built-in) multipliers. As a result, a user function that contains a large number of multiplications may not fit in an FPGA because it requires too many block multipliers, even though the FPGA still has plenty of slices. Since a multiplier can also be implemented with FPGA slices, it is possible that the user function would fit in the FPGA if some of the multiplications were bound to multipliers that are built based on the FPGA slices. Balancing the use of FPGA resources of a hardware function unit is also desirable when multiple copies of the function unit are needed on a single FPGA device.

Motivated by the above observations, this paper intends to add the resource-balancing feature to the MAP compiler. So, a module selection optimization problem is formulated, in which, different types of multiplier implementations are selected, and balancing FPGA resource utilities is treated as a constraint, so that the solution to the module selection problem is the balanced use of the FPGA resources.

There have been several studies that are related to balancing the FPGA resource usage. The authors [4] developed a methodology to optimize the usage of the different components in a heterogeneous FPGA specifically for 2D FIR filters, by using Singular Value Decomposition to approximate a 2D filter. A technique [5] was proposed to transfer FPGA resource usages between FPGA block RAMs and block multipliers, based on polynomial approximation. In [6] a new technique was proposed that makes use of FPGA block RAMs and maps variables to them rather than registers.

This memory-binding technique resulted in large savings in FPGA slices at the expense of increasing FPGA block RAM utilities. In [7] the author proposed an algorithm that identifies part of the circuit that can be implemented in embedded RAMs.

The rest of the paper is organized as follows. Section 2 gives an overview of SRC-6E reconfigurable computer. Section 3 presents an example of mapping the forward complex FFT onto the SRC-6E reconfigurable computer, which motivates the study and serves as the test case. Section 4 describes the problem formulation and its variations. Section 5 provides two algorithms to solve the problem. Section 6 gives some experimental results. Section 7 concludes the paper.

2. SRC-6E Platform Overview

The SRC-6E reconfigurable computer consists of two dual-microprocessor boards and two MAP® processors, each with two user programmable Xilinx® Vertex II XC2V6000™ FPGA chips and six 4MB banks of on-board memory (OBM). The microprocessors are connected to the MAP processors via SNAP® cards which plug into the DIMM slot on the microprocessor motherboard [1].

The programming model for the SRC-6E is similar to that for conventional microprocessor-based computers, with the additional task of producing logic for the MAP reconfigurable processor. Two types of application source files are needed to target the microprocessor and the MAP processor, respectively.

There are two levels of source files for the MAP processor. At a high level, the user describes the function, called the MAP function designated for hardware, using a high-level programming language such as FORTRAN or C. The MAP complier [3] then converts this high-level language description into a Verilog description for the FPGA. Optimized macros for the hardware are included as a bundled library with the SRC system and can be called from the MAP function. This library includes functions such as DMA calls, accumulators, counters, etc. Additionally, at a low level, the MAP compiler allows users to integrate their own custom VHDL/Verilog functions or macros to extend the built-in set included with the SRC-6E platform.

It is noticed that the MAP compiler will pipeline every innermost loop in a MAP function. Therefore, the user should try to avoid loop-carried dependencies and to reduce OBM accesses in an innermost loop in order to achieve optimum performance of a pipelined loop. Moreover, merging a nested loop into a single loop is certainly desirable since a larger loop is pipelined after merging. It is also noticed that the MAP compiler does not perform resource sharing. This is probably because the following fact. In case that a loop is fully pipelined, i.e. a new iteration of the loop is initiated every clock cycle, operations in the loop body cannot share the resource.

3. Motivating Example

A study [8] of porting the Parallel Spectral Transform Shallow Water Model (PSTSWM) parallel benchmark code [9] to the SRC-6E reconfigurable computer demonstrated a need to map the computation of in-place forward FFT over an array of complex vectors onto the MAP processor. The inputs to the FFT computation are a one-dimensional array TRIG for storing the twiddle factors and a two-dimensional array Y (an array of complex vectors), which is also used as the output. Due to the inherent parallelism of the FFT computation, an FPGA function unit is designed that computes the FFT over one vector and then multiple copies of the function unit are placed on the FPGA. Initially, we only consider 32-bit integer operations.

3.1 Designing the Function Unit

The FPGA design for the SRC-6E reconfigurable computer is developed by simply writing the MAP function in either the C or FORTRAN programming language. The user decides whether to allocate arrays to the OBM banks or to the block RAMs inside the FPGA chip. The user also must take care to optimize the MAP function code as mentioned in Section 2.

In the function unit design, one OBM bank is used for the array of complex vectors Y. Since both the real part and the imaginary part of a complex number are assumed to be 32-bits and the memory cell of the OBM is 64-bits, the real part and the imaginary part of a complex number are packed together into one memory cell. The array TRIG is allocated to block RAMs.

The FPGA resource utilities of this design labeled as Design 1 are shown in Table 1. Note that the FPGA chip used on the MAP processor contains 33,792 slices, 144 block RAMs, and 144 block multipliers (MULT18×18s).

Table 1: FPGA Resource Utilities of MAP Function Designs

FPGA Resources	Design 1		Design 2	
Slices (33,792)	9,367	27%	8,846	26%
Block RAMs (144)	6	4%	6	4%
MULT18×18s (144)	56	38%	52	36%

Design 1 is further optimized to create Design 2. First, a nested two inner loops are combined into a single loop. Second, an unnecessary loop is eliminated. Third, a multiplication operation is replaced by additions. Fourth, a code block is rewritten so as to avoid the data dependency and therefore reduce the latency. The

FPGA resource utilities of this optimized design labeled as Design 2 are also shown in Table 1.

3.2 Balancing the FPGA Resource Utilities

From Table 1, it is expected that if three copies of Design 2, each consuming one OBM bank, could fit on a single FPGA chip, one MAP processor, with two FPGA chips and six OBM banks, could hold six copies of Design 2. The six hardware copies can work on different vectors in parallel. But, unfortunately, three copies of Design 2 consume more block multipliers than those available in the FPGA. So the three copies do not fit in the FPGA although there are many unused slices. Thus there is a need to balance the FPGA resources. By implementing a multiplier using FPGA slices rather than FPGA block multipliers, it is possible to fit more multiplication operations into a single FPGA and more efficiently utilize the FPGA resources.

Since the SRC-6E computing environment allows users to use commercial hardware design tools to design their own macros, which can be called from within a MAP function, a slice-based multiplier macro is built using slices rather than block multipliers. In Design 3, four multiplication operations in the MAP function are bound to the slice-based multiplier macros. The FPGA resource utilities of Design 3 are shown in Table 2.

Table 2: FPGA Resource Utilities of MAP Function Designs

FPGA Resources	Design 3		Design 4	
Slices (33,792)	9,816	29%	26,098	77%
Block RAMs (144)	6	4%	18	12%
MULT18×18s (144)	40	27%	120	83%

Now, three hardware copies of Design 3 can fit on a single FPGA chip as shown in Design 4 in Table 2. Note that the interface design in Design 3 including the DMA data transfer from the host memory to the OBM banks remains the same in Design 4. That is, only the computation part of Design 3 is tripled in Design 4. Therefore, the number of slices used in Design 4 is less than the triple of that used in Design 3.

Note that all hardware macros in the SRC-6E system must satisfy the 100 MHz system clock rate constraint. Both slice-based multipliers and block multiplier-based multipliers are fully pipelined, and the former needs few more pipeline stages than the latter. So the former will have a little longer latency than the latter. But they have the same throughput. Therefore, a pipelined loop design with either the former or the latter will have almost the same latency.

4. Problem Formulation

The FPGA resource-balancing problem discussed in this paper is actually a particular module selection problem [10] where a decision on which multiplier implementation should be selected for a given multiplication needs to be made. It will be formulated as a constrained optimization problem in this section. One of these constraints will balance the use of FPGA resources.

FPGA resources considered in this paper include FPGA slices, block multipliers, and block RAMs. But, at this time, only FPGA slices and block multipliers will be considered to balance in the problem formulation. Note that some interconnect is shared between the block RAMs and the block multipliers [2] thus, the block RAM can be used only up to 18-bits wide when the block multiplier is used. In another words, the number of available block multipliers to a FPGA design will be decreased by one when a block RAM configured with 36-bits is used in the design.

4.1 Enumerating Related Multiplier Macros

Since up to 32-bit integer operations are considered in this paper, only three types of multipliers are relevant. Type A has two 32-bit inputs and one 32-bit output. Type B has one 32-bit input and one 16-bit input and one 32-bit output. Type C has two 16-bit inputs and one 32-bit output.

A multiplier of Type A can be implemented using up to three block multipliers. Thus, four multiplier macros of Type A can be implemented using zero, one, two, or three block multipliers, respectively. The FPGA resource utilities of these macros are shown in Table 3.

Table 3: FPGA Resource Utilities of Type A Multiplier Macros

FPGA Resources	MA0	MA1	MA2	MA3
Slices (33,792)	355	236	185	118
MULT18×18s (144)	0	1	2	3

A multiplier of type B can be implemented using up to two block multipliers. Thus, three multiplier macros of Type B can be implemented using zero, one, or two block multipliers, respectively. The FPGA resource utilities of these macros are shown in Table 4.

Table 4: FPGA Resource Utilities of Type B Multiplier Macros

FPGA Resources	MB0	MB1	MB2
Slices (33,792)	257	124	62
MULT18×18s (144)	0	1	2

Similarly, a multiplier of type C can be implemented using either zero or one block multiplier, respectively. The FPGA resource utilities of the two macros are shown in Table 5.

Table 5: FPGA Resource Utilities of Type C Multiplier Macros

FPGA Resources	MC0	MC1
Slices (33,792)	163	57
MULT18×18s (144)	0	1

Note that all these macros are fully pipelined and satisfy the 100 MHz system clock rate constraint.

4.2 Notations and Initial Conditions

Some notations used in the formulation are introduced as follows. Let *TNS*, *TNBM*, and *TNBR* denote the total number of FPGA slices, block multipliers, and block RAMs, respectively. Let SA_n (n=0, 1, 2, 3), SB_n (n=0, 1, 2), and SC_n (n=0, 1) denote the number of FPGA slices used for the multiplier macro of type A, B, and C using n block multipliers, respectively. Notice that the following inequalities hold.

$$SA_0 > SA_1 > SA_2 > SA_3 \geq 0 \tag{1}$$

$$SB_0 > SB_1 > SB_2 \geq 0 \tag{2}$$

$$SC_0 > SC_1 \geq 0 \tag{3}$$

In an FPGA design, let NA_n (n=0, 1, 2, 3), NB_n (n=0, 1, 2), and NC_n (n=0, 1) stand for the number of multiplier macro calls of type A, B, and C using n block multipliers, respectively. Let *NA*, *NB*, and *NC* stand for the number of multiplications of type A, B, and C, respectively. Note that every multiplication should be bound to the same type multiplier. Therefore, the following conditions must be satisfied.

$$NA_n \geq 0, n = 0, 1, 2, 3 \tag{4}$$

$$NB_n \geq 0, n = 0, 1, 2 \tag{5}$$

$$NC_n \geq 0, n = 0, 1 \tag{6}$$

$$NA_0 + NA_1 + NA_2 + NA_3 = NA \tag{7}$$

$$NB_0 + NB_1 + NB_2 = NB \tag{8}$$

$$NC_0 + NC_1 = NC \tag{9}$$

Let *NBM*, *NBR18*, and *NBR36* stand for the number of block multipliers, block RAMs configured with up to 18-bits, and block RAMs configured with 36-bits, respectively. Let *NS4I* and *NS4C* stand for the number of FPGA slices used for the interface and FPGA slices used for the computation, respectively.

The sum of *NS4I* and *NS4C* should be equal to the number of FPGA slices used for the whole FPGA design. It is assumed that *NS4I* is not going to change when multiple copies of the computation unit of the FPGA design are used. It is also assumed that all block multipliers and all block RAMs used in the FPGA design belong to the computation unit. Therefore, *NS4C*, *NBM*, *NBR18*, and *NBR36* will be doubled when two copies of the computation unit of the FPGA design are used.

It is assumed that every multiplication is initially bound to the same type multiplier that uses maximum number of block multipliers in an initial synthesis without considering the resource balancing like the current MAP compiler. This initial conditions are represented by equalities (10), (11), and (12).

The values of *NA*, *NB*, *NC*, *NS4I*, *NBR18*, and *NBR36* are assumed known after the initial synthesis and unchanged during the process of balancing FPGA resources. The initial values of *NS4C* and *NBM* are also known after the initial synthesis and denoted by $NS4C_0$ and NBM_0 respectively.

$$NA_0 = NA_1 = NA_2 = 0 \text{ and } NA_3 = NA \tag{10}$$

$$NB_0 = NB_1 = 0 \text{ and } NB_2 = NB \tag{11}$$

$$NC_0 = 0 \text{ and } NC_1 = NC \tag{12}$$

When the initial values for NA_n (n=0, 1, 2, 3), NB_n (n=0, 1, 2), and NC_n (n=0, 1) are changed, the change of the number of block multipliers used for the computation ($NBM-NBM_0$) and the change of the number of FPGA slices used for the computation ($NS4C-NS4C_0$) can be calculated by the following two formulae, respectively.

$$\begin{aligned} NBM - NBM_0 = {} & (NA_1 + 2 \times NA_2 + 3 \times (NA_3 - NA)) \\ & + (NB_1 + 2 \times (NB_2 - NB)) \\ & + (NC_1 - NC) \end{aligned} \tag{13}$$

$$\begin{aligned} & NS4C - NS4C_0 = \\ & (SA_0 \times NA_0 + SA_1 \times NA_1 + SA_2 \times NA_2 + SA_3 \times (NA_3 - NA)) + \\ & (SB_0 \times NB_0 + SB_1 \times NB_1 + SB_2 \times (NB_2 - NB)) + \\ & (SC_0 \times NC_0 + SC_1 \times (NC_1 - NC)) \end{aligned} \tag{14}$$

4.3 Formulating the Problem

Now the module selection problem for balancing FPGA resources discussed in this paper is to determine proper values for NA_n (n=0, 1, 2, 3), NB_n (n=0, 1, 2), and NC_n (n=0, 1) so as to have a balanced use of slices and block multipliers. We define

$$CComp(NS4C) = Floor\left(\frac{TNS - NS4I}{NS4C}\right) \tag{15}$$

$$CMult(NBM) = Floor\left(\frac{TNBM}{NBM + NBR36}\right) \tag{16}$$

$$CRams = Floor\left(\frac{TNBR}{NBR18 + NBR36}\right) \tag{17}$$

In the above definition, *TNS-NS4I* is the slices available for the computation unit and *NS4C* is the slices actually used for the computation unit. *TNBM* is the block multipliers available for the computation unit and *NBM+NBR36* is the equivalent block multipliers used actually for the computation unit, because if a block RAM is configured with 36-bits wide, then the corresponding block multiplier cannot be used any more. *TNBR* is the block RAMs available for the computation unit and *NBR18+NBR36* is the block RAMs actually used for the computation unit. So it can be seen that the maximum number of copies of the computation unit that can fit on a single FPGA chip will be the minimum value of *CComp*(*NS4C*), *CMult*(*NBM*) and *CRams*.

Because *CRams* is fixed after the initial synthesis but *CComp*(*NS4C*) and *CMult*(*NBM*) are going to change during the process of the resource balancing, the maximum number of copies of the computation unit will be achieved when the following, called the balance constraint, holds.

$$CMult(NBM) = CComp(NS4C) \tag{18}$$

Proposition 1: The minimum value of *NS4C* is $NS4C_0$ and the maximum value of *NBM* is NBM_0. The maximum value of *CComp*(*NS4C*) is $CComp(NS4C_0)$ and the minimum value of *CMult*(*NBM*) is $CMult(NBM_0)$.

Proposition 2: Minimizing *NS4C* implies maximizing *CComp*(*NS4C*). Minimizing *NBM* implies maximizing *CMult*(*NBM*).

Now the resource-balancing problem can be formulated as follows: Determine NA_n (n=0, 1, 2, 3), NB_n (n=0, 1, 2), and NC_n (n=0, 1), to

minimize *NS4C*
subject to the constraints (4)-(9) and
the balance constraint (18). $(P1)$

Note that the balance constraint (18) is too strong. It is very likely that (*P1*) does not have a solution to some problems. Two released balance constraints: slice-bound constraint (19) and block multiplier-bound constraint (20) are defined as follows.

$$CMult(NBM) \geq CComp(NS4C) \tag{19}$$

$$CMult(NBM) \leq CComp(NS4C) \tag{20}$$

Then a released version of (P1) formulation can be

minimize *NS4C*
subject to the constraints (4)-(9) and
the slice-bound constraint (19). $(P2)$

Proposition 3: If (*P1*) has a solution to a problem, then it must be the solution of *(P2*) to the same problem.

Moreover, a dual formulation of (*P1*) and a dual formulation of (*P2*) can be as follows.

minimize *NBM*
subject to the constraints (4)-(9) and
the balance constraint (18). $(Q1)$

minimize *NBM*
subject to the constraints (4)-(9) and
the multiplier-bound constraint (20). $(Q2)$

Proposition 4: If (*Q1*) has a solution to a problem, then it must be the solution of *(Q2*) to the same problem.

It can be noticed that either (*P2*) or (*Q2*) must have a solution for a common problem. If both (*P2*) and (*Q2*) have solutions to the same problem, then both solutions must satisfy the balance constraint, and then the solution of (*P2*) is also the solution of (*P1*) and the solution of (*Q2*) is also the solution of (*Q1*).

A solution of (*P2*) for one problem may end up with *CComp*(*NS4C*) = 2 and *CMult*(*NBM*) = 3. This means that we can have two copies of computation unit, even though we have enough block multipliers for three. On the other hand, a solution of (*Q2*) for another distinct problem may end up with *CComp*(*NS4C*) = 3 and *CMult*(*NBM*) = 2. This means that we can have two copies of computation unit, even though we have enough slices for three. Please note that these two situations will not occur for the same problem.

5. Two Algorithms

5.1 A Naïve Algorithm

The formulations (*P1*), (*P2*), (*Q1*), and (*Q2*) can be very easily solved by a naïve algorithm, which is described for the formulation (*P1*) as follows. Enumerate all possible values of NA_n (n=0, 1, 2, 3), NB_n (n=0, 1, 2), and NC_n (n=0, 1). For those values that satisfy the constraints (4)-(9), compute *NBM*, *NS4C*, *CMult*(*NBM*) and *CComp*(*NS4C*) based on (13), (14), (16) and (15). If the balance constraint (18) is satisfied, keep track of those values that make *NS4C* smaller than before.

Note that this naïve algorithm performs the exhaustive search and thus will produce an exact solution to the constrained optimization problem. However, the worst-case complexity of the naïve algorithm is very high. It can be computed by $(NA+1)^3\times(NB+1)^2\times(NC+1)^1$, where the exponents 3, 2, and 1 are the number of different macros of Type A, B, and C multipliers less 1, respectively.

5.2 A Greedy Algorithm

An efficient greedy algorithm that solves (*P2*) is given in this section. A similar discussion may apply to (*Q2*), which is omitted in this paper owing to the space limit.

From $CMult(0) \geq CMult(NBM)$ and $CComp(NS4C_0) \geq CComp(NS4C)$ for any values of NA_n (n=0, 1, 2, 3), NB_n (n=0, 1, 2), and NC_n (n=0, 1) that satisfy the constraints (4)-(9), the following condition is sufficient for (*P2*) to have a solution, and then is assumed true in this section.

$$CMult(0) \geq CComp(NS4C_0) \tag{21}$$

When $CMult(NBM_0) \geq CComp(NS4C_0)$, the initial settings of NA_n, NB_n, and NC_n already provide an optimal solution, and, in this case, $NS4C = NS4C_0$. Now, consider

$$CMult(NBM_0) < CComp(NS4C_0) \tag{22}$$

This condition indicates that there are relatively more block multipliers initially used in the function unit design than expected.

The basic idea of the greedy algorithm to solve (*P2*) is to decreased *NBM* by one at one step and to keep *NS4C* with a minimum increase at each step until *CMult*(*NBM*) ≥ *CComp* (*NS4C*).

Note that there are six different schemes to reduce *NBM* by one as shown in Table 6. The action for each scheme is listed in the action column. Each scheme action must satisfy a condition so as to satisfy the problem constraints. The condition for each scheme is listed in the condition column. Note that when *NBM* is decreased, *NS4C* will increase. The increase of *NS4C* for each scheme, called cost, is listed in the cost column.

Table 6: Six Schemes to Reduce *NBM* by One

Scheme	Action	Condition	Cost
0	NA_1-- & NA_0++	$NA_1>0$	SA_0-SA_1
1	NA_2-- & NA_1++	$NA_2>0$	SA_1-SA_2
2	NA_3-- & NA_2++	$NA_3>0$	SA_2-SA_3
3	NB_1-- & NB_0++	$NB_1>0$	SB_0-SB_1
4	NB_2-- & NB_1++	$NB_2>0$	SB_1-SB_2
5	NC_1-- & NC_0++	$NC_1>0$	SC_0-SC_1

A scheme with a smaller cost will have a higher priority to apply. For example, when the values of SA_n (n=0, 1, 2, 3), SB_n (n=0, 1, 2) and SC_n (n=0, 1) are taken from Tables 3, 4, and 5, respectively, the costs of each scheme are listed in Table 7. It can be seen that Scheme 1 has the highest priority and Scheme 3 has the lowest priority. The six schemes sorted from the smallest cost to the largest cost form an array, called priority array.

Table 7: An Example of Costs of Each Scheme

Scheme	0	1	2	3	4	5
Cost	119	51	67	133	62	106

A pseudo code of the greedy algorithm in the C++ context is shown below. In each iteration *NBM* is decreased by one and *NS4C* is updated accordingly by taking one possible scheme action in Table 6 with the highest priority until *CMult*(*NBM*) ≥ *CComp*(*NS4C*). Note that the loop will terminate because (21) is assumed true.

```
while(CMult(NBM) < CComp(NS4C))
{
    index = 0;
    update = 0;
    while(update == 0)
    {
        update = Update(Priority[index], &NS4C);
        index++;
    }
    NBM--;
}
```

The worst-case complexity of the greedy algorithm is (*NA*×3+*NB*×2+*NC*)×6, where the factor 6 (=3+2+1) is the number of schemes that can reduce *NBM* by one, and the addends 3, 2, and 1 are the number of different macros of Type A, B, and C multipliers less 1, respectively.

6. Experiments

The naïve algorithm and the greedy algorithm are both implemented using C++. The data from the motivating example in Section 3 are used as the program inputs as listed in Table 8.

Table 8: Program Inputs

TNS	33792	*NA*	15	SB_0	257
TNBM	144	*NB*	3	SB_1	124
TNBR	144	*NC*	1	SB_2	62
$NS4C_0$	8141	SA_0	355	SC_0	163
NS4I	1675	SA_1	236	SC_1	57
NBR18	0	SA_2	185		
NBR36	6	SA_3	118		

The results produced by both algorithms are listed in Table 9. In this example, (*P1*) and (*P2*) have the same solution, and (*Q1*) and (*Q2*) also have the same solution. They are solved by the naïve algorithm. (*P2*) is also solved by the greedy algorithm. It can be seen that the results from the greedy algorithm are very close to the exact results from the naïve algorithm. But the complexity of the naïve algorithm is much higher that that of the greedy algorithm.

The last column of Table 9 shows results obtained manually from the motivating example. The *NBM* and *NS4C* are computed from (13) and (14) respectively. Note that *NS4C* +*NS4I* = 9089 + 1657 = 10746, which is greater than 9,816, which is obtained from the Xilinx's Place and Routing tool for Design 3 in Section 3. This is because the Xilinx's tool simplifies the user logic and prunes the unused user logic.

Table 9: Experimental Results

Parameter	Naïve		Greedy P2	Manual
	P1 or P2	Q1 or Q2		
NA_0	0	0	0	4
NA_1	5	1	3	0
NA_2	0	14	1	0
NA_3	10	0	11	11
NB_0	0	2	0	0
NB_1	0	1	3	0
NB_2	3	0	0	3
NC_0	0	0	0	0
NC_1	1	1	1	1
NS4C	8731	9649	8748	9089
NBM	42	31	42	40
Max Copies	3	3	3	3
Complexity	795906	795906	318	N/A

Please notice that in order to plug the module selection algorithm into the SRC compilation environment, the

MAP compiler must be able to estimate the FPGA utilities for a given MAP function. Otherwise, we have to use this algorithm off-line as follows. First, use the MAP compiler to translate a given MAP function from Fortran or C into Verilog. Second, use commercial tools to synthesize the Verilog file and to obtain the FPGA resource utilities. Third, use this algorithm to get the balanced use of FPGA slices and block RAMs, and then change the MAP function to allocate (call) a proper multiplier macro for a given multiplication. Finally, the modified MAP function code needs to go through the MAP compiler and commercial synthesis tools again.

7. Conclusion

An FPGA module selection problem is formulated to deal with balancing FPGA resources. In the problem formulation a decision on which multiplier implementation should be bound to a given multiplication needs to be made. Several variations of the problem formulation are also studied. A naïve algorithm and an efficient greedy algorithm to solve the problem are provided. The naïve algorithm is able to produce exact results and the greedy algorithm produces accurate results at a much lower cost. The worst-case complexities of the two algorithms are studied, and the complexity of the greedy algorithm is very low. The two algorithms are also implemented and tested. The results produced by the two algorithms are very close, and compared with results from a manual FGPA design.

The FPGA resource-balancing problem studied in this paper is far from complete. The balancing of FPGA block RAMs with FPGA slices and FPGA block multipliers is not considered. Additionally, the block RAMs may be replaced with the distributed RAMs built from FPGA lookup tables inside FPGA slices freeing block RAMs and in the process, additional block multipliers. To accommodate more block multipliers, a block RAM configured with 36-bits can be replaced with two block RAMs configured with 18-bits. Therefore, the FPGA resource-balancing problem becomes more challenging when block RAMs are also considered.

Moreover, only three types of multipliers are considered in this paper. In general, a parameterized multiplier should be considered. The implementations of the multipliers should also be optimized. The FPGA slices required by the multiplier macros used in this research may be reduced. These macros are fully pipelined and can run at 100 MHz clock rate or above because the SRC reconfigurable computer demands 100 MHz clock rate.

8. Acknowledgments

This research was supported in part by an appointment to the Oak Ridge National Laboratory/Oak Ridge Associated Universities Historically Black Colleges and Universities and Minority Education Institutes Summer Faculty Research Program at the Oak Ridge National Laboratory administered by the Oak Ridge Institute for Science and Education.

Moreover, the SRC Computers, Inc. provided the authors with the remote accesses of its SRC-6E reconfigurable computer. All the FPGA designs including all the macros were developed under the SRC computing environment.

9. References

[1] SRC Computers, Inc., http://www.srccomp.com/, 2004.

[2] Xilinx, Inc., Virtex-II Platform FPGAs: Complete Data Sheet, June 2004.

[3] SRC-6 Fortran Programming Environment v1.7 Guide, SRC Computers, Inc. 2004.

[4] Christos Bouganis, George Constantinides, and Peter Cheung, A Novel 2D Filter Design Methodology For Heterogeneous Devices, in IEEE Symposium on Field Programmable Custom Computing Machines, April 2005

[5] Gareth Morris, George Constantinides, and Peter Cheung, Migrating Functionality From ROMs to Embedded Multipliers, in IEEE Symposium on Field Programmable Custom Computing Machines, p.287-288, April 2004

[6] Hassan Al Atat and Iyad Ouaiss, Register Binding for FPGAs with Embedded Memory, in IEEE Symposium on Field Programmable Custom Computing Machines, p.167-175, 2004

[7] S. Wilton, SMAP: Heterogeneous Technology Mapping for Area Reduction in FPGAs with Embedded Memory Arrays, ACM/SIGDN International Symposium on Field Programmable Arrays, February 1998

[8] M. C. Smith, J. S. Vetter, and X. Liang, Accelerating Scientific Applications with the SRC-6E Reconfigurable Computer: Methodologies and Analysis, The 12th Reconfigurable Architectures Workshop, Denver, Colorado, USA, April 2005

[9] P. H. Worley and B. Toonen, A USERS' GUIDE TO PSTAWM, ORNL Technical Report ORNL/TM-12779, July 1995

[10] Giovanni de Micheli, Synthesis and Optimization of Digital Circuits, Mcgraw-Hill, Inc., 1994.