An Exact Synthesis Algorithm for Multi-loop Array Computations on Field Programmable Gate Arrays (FPGAs)

Nagaraj Shenoy, Prithviraj Banerjee, Alok Choudhary
Northwestern University
Mahmut Kandemir
Pennsylvania State University

Technical Report No. CPDC-TR-2000-03-003

March 2000

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

An Exact Synthesis Algorithm for Multi-loop Array Computations on Field Programmable Gate Arrays (FPGAs)

Nagaraj Shenoy, Prithviraj Banerjee, Alok Choudhary
Northwestern University

Mahmut Kandemir
Pennsylvania State University

March 2000

Abstract

Array intensive computations are characterized by processing of large arrays stored in the external memory in multiple loops. Synthesizing these computations onto FPGAs involves automatic translation of the behavioral description into state machines controlled by a clock such that the execution time of the program as a whole is the minimum and area requirement does not exceed a predefined limit. The synthesis algorithm also needs to efficiently sequence the array accesses taking into account memory access requirements such as pipelining.

In this report we present an MILP based algorithm that can optimally synthesize behavioral descriptions. It tries to minimize both execution time and area. Our algorithm not only looks at individual loops to exploit parallelism but also considers them together while deciding the clock. The overall execution time is minimized and not just the number of cycles or the cycle time. It also efficiently synthesizes memory accesses to fully exploit the memory pipelining. We compare the results of this algorithm with the the results generated by a heuristic based algorithm.

1 Introduction

With the commercial availability of high capacity FPGA boards such as [1], it is now possible to use reconfigurable compute engines to accelerate more general purpose computations. The main challenge though is the tools that can automatically synthesize behavioral level programs onto the hardware. The class of applications that involve computations on data stored in large arrays - the domain of this paper - poses several challenges in achieving the best performance from the accelerator.

These arrays are often stored in external memories requiring one or more cycles to access the data. This latency needs to be hidden by overlapping computation with data access. Further, as is the case in [1], the memories can be pipelined and keeping the pipeline busy by appropriately ordering different data accesses is an important issue.

Apart from the parallelism between data accesses and computation, an efficient code needs to extract parallelism across different operations in an expression, across different statements, across iterations in a loop and even across loop nests. This parallelism is made more explicit by performing a control/data flow analysis of the program and by representing the computations in terms of what are known as *control data flow graphs (CDFG)*. Figure 1 shows the data flow graph of a simple program with two loops, which we use in this section to highlight some of the synthesis issues.

The task of the synthesis algorithm is to automatically map these loops into a corresponding state machine controlled by a clock. Figure 2 shows one possible mapping.

Several observations can be made by experimenting with various alternate implementations of this simple example program on to FPGAs. First, synthesizing this program fragment to run at the highest possible clock is not necessarily the best way. For example, the first loop can be synthesized to run at 48 MHz (assuming Xilinx Virtex FPGA). But actually the loop runs most efficiently (in terms of execution time) at 42 MHz. This has to do with how the operations can be packed into different states and how evenly the computation is distributed across the states. A higher clock could introduce greater *holes* in the states whereas a lower clock could result in a tighter packing, in addition to reducing the number of required states. If we consider the second loop alone, it can be run at 114 MHz which is also the optimal clock for

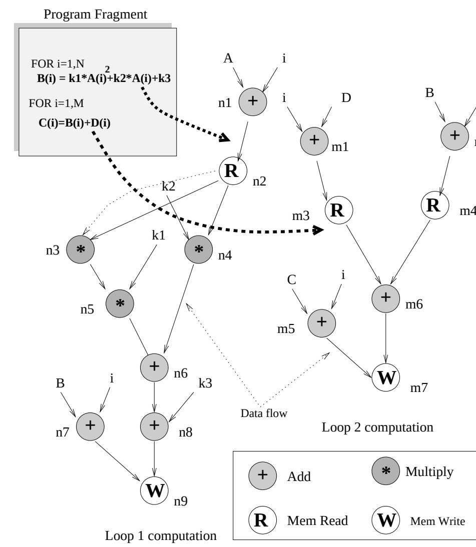

Figure 1: An example program to illustrate various synthesis issues (only the loop bodies are shown in the CDFG).

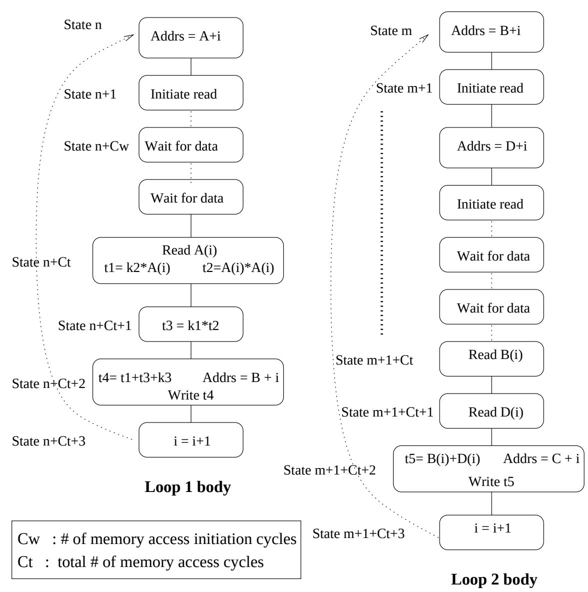

Figure 2: A possible mapping of the operations in the example program into different states in a state machine.

that loop.

However, the optimal clock happens to be 42 MHz when both loops are considered and synthesized into a single state machine. This shows the importance of synthesizing multiple loops by considering them together rather than in isolation.

A further observation can be made when we consider the relative values of the loop bounds `M` and `N`.

It is interesting to note that only when the ratio M/N is less than 3 the optimal clock for this example is 42 MHz. For any other ratio the optimal clock happens to be 48 MHz. So, we not only need to consider loops together but also consider the number of iterations in each loop.

Now let us consider that each memory access requires C_w initiation cycles (for setting up the address and other handshake signals) and C_t total number of cycles before the data is ready (for read access). The total time spent by each loop in accessing the memory would be $N*(C_t+C_w)+M*(2*C_t+C_w)$. However, by utilizing the pipelining supported by the memory we can bring this down to $N*(C_t+C_w)+M*(C_w+C_t+C_w)$ by overlapping the memory accesses in nodes m_3 and m_4. Further reduction may be possible if we pipeline the accesses across loop iterations.

The number of CLBs needed to synthesize a given program can also be an important issue and often there can be an upper bound on the CLBs available. A naive synthesis of the example program may need at least the CLBs corresponding to 3 multipliers and 8 adders. But by cleverly *scheduling* the various operations, this can be brought down to 2 multipliers and 2 adders and we can still achieve the same optimal solution in terms of the execution time. One interesting problem could be to find out the minimum resources needed to synthesize a given computation without exceeding the achievable minimum execution time. This could result in large savings in the FPGA resources that can be used to pack more computation on to the same FPGA.

In this paper we propose an algorithm that synthesizes multiple loops considering the interaction between the loops (in terms of performance), parallelism across operations/statements and efficient pipelining of memory accesses while being conservative about the CLB requirements. Our algorithm is based on Mixed Integer Linear Programming (MILP) (section 3) and ensures optimality of the solution. We evaluate our algorithm by comparing it with a heuristic based algorithm in section 5. We start with a brief overview of the related work in the next section.

2 Related Work

High level synthesis in general is a well researched area and a wide range of algorithms have been proposed to address specific synthesis problems [4, 5]. Scheduling has been an important area focused on by many researchers. Commercial behavioral compilers such as [6] can automatically perform scheduling under different conditions. Given a clock frequency and optional demarcation of state boundaries, this tool can schedule the operations across the states so that the resources are efficiently utilized.

In many of the recent works [7, 8, 9, 10], the main focus has also been exploitation of parallelism and efficient resource utilization. [8] for example tries to schedule multiple loop iterations. These works often employ clever heuristic based algorithms that are designed to be fast but may not guarantee optimality. On the other hand, Mixed Integer Linear Programming (MILP) has been used in some of the scheduling and low level synthesis algorithms [11, 12] that guarantee optimality at the expense of higher synthesis times.

In array intensive computations, efficient access of the external memory is also an important issue. [13] deals elaborately with different types of memories and their access modes and the ways of synthesizing the accesses. Memory accesses can lead to scheduling bottlenecks due to data dependency and a technique to overcome these problems is proposed in [15]. An application of pointer analysis to synthesis is proposed by [14]. Many of the FPGA boards support multiple external memories and allocation of arrays to these memories is an important issue. [16] suggests a heuristic based allocation technique.

The algorithm proposed in this paper differs from those listed above in the following sense. Our algorithm is specifically designed with FPGA based boards in mind. Our algorithm tries to find the best clock rate and the structure of the state machines to minimize both the execution time as well as the CLBs needed.

Our algorithm not only looks at individual loops but also considers them together while deciding the clock. The overall execution time is minimized and not just the number of cycles or the cycle time. Our algorithm also efficiently handles external memory accesses by suitably overlapping the pipelined accesses. When possible, it tries to exploit parallelism at various levels in order to minimize the execution time. Currently our algorithm does not exploit parallelism across loop iterations and does not handle multiple memories. These are possible future extensions to our algorithm.

Constants for a given target	
C_w	# of initial cycles for memory access.
C_t	# of total cycles for memory access.
α_t	Area requirement for oper t.
δ_i	Delay associated with oper in node i.
β	Per state control path area requirement.
γ	Per state control path delay.
Constants for a given DAG	
I_d	# of iterations (or an estimate) in loop d.
$\mathcal{U}_i$	Upper bound on the # states in loop i.
μ_i	State mobility interval for node i.
$\mathcal{N}$	An arbitrary large integer constant.
Model Variables	
P	Clock period.
s_i	Start time of node i
g_{ij}	Boolean variable indicating whether node i is mapped to state j.
G_i	State to which node i gets scheduled.
Δ	Total execution time of the synthesized design.
η_t	# of operators of type t.
Θ	Total area requirement of the design.
z_{ij}	An auxiliary boolean variable to ensure non overlap of memory operations i and j.

Table 1: Notation used in our model.

3 The MILP Model

Our Mixed Integer Linear Programming (MILP) model to solve the synthesis problem discussed in section 1 is captured by the set of constraints and objective functions listed below. Table 1 shows the notation used.

$$
\begin{aligned}
s_i &\geq s_k + \delta_k \quad \forall \text{ opr } n_k \prec n_i && (1)\\
&\geq s_k + C_t P \\
&\qquad \forall \text{ memop } n_k \prec n_i && (2)\\
\sum_{j \in \mu_i} g_{ij} &= 1 \quad \forall \text{ opr i} && (3)\\
G_i &= \sum_{j \in \mu_i} j g_{ij} && (4)\\
s_i + \mathcal{N} - \mathcal{N} g_{ij} &\geq (j-1)P + \gamma && (5)\\
s_i - \mathcal{N} + \mathcal{N} g_{ij} &\leq jP - \delta_i \quad \forall\, j \in \mu_i && (6)\\
s_i + \mathcal{N} z_{ij} &\geq s_j + C_w P && (7)\\
s_j - \mathcal{N} z_{ij} + \mathcal{N} &\geq s_i + C_w P && (8)\\
\eta_t &\geq \max_{\forall\, state\ j} \sum_{\forall\, i} g_{ij} \\
&\qquad \forall \text{ node i s.t. i uses } t \\
&\qquad j \in \mu_i && (9)\\
\textbf{Objective 1: } \Theta &= \sum_{\forall\, t} \eta_t \alpha_t + \\
&\qquad \sum_{\forall\, last\ node\ l} G_l \beta && (10)
\end{aligned}
$$

$$
\begin{aligned}
\textbf{Objective 2: } \Delta &= \sum_{\forall\, loop\ d} I_d s_{l_d} \\
&\qquad l_d \text{ being last node in d} && (11)
\end{aligned}
$$

We briefly discuss these constraints under the following groups. For the purposes of this model we assume that the DAG corresponding to each loop has a dummy node which we refer to as *last node* or *end node* which is the successor of all other nodes in the DAG.

3.1 Dependence constraints

The directed data edges in the CDFG indicate that a node at the head of an edge (successor) can start only after the node at the tail (predecessor) has finished. If the predecessor is an operator then the successor can start only after the operator has finished execution (refer to Constraint 1). If the predecessor is a memory operation, then the successor needs to wait for it to be completed; which could take C_t cycles (refer to Constraint 2).

For example, consider node $n6$ in Figure 1. $n6$ adds the results computed by the nodes $n4$ and $n5$. So it can start only when both $n4$ and $n5$ have finished. In other words

$$
\begin{aligned}
s_{n6} &\geq s_{n4} + \delta_* \\
&\geq s_{n5} + \delta_*
\end{aligned}
$$

Similarly, the node $n4$ cannot start execution unless one of its operands is read from memory by the node $n2$. This puts the restriction

$$s_{n4} \geq s_{n2} + C_t P$$

3.2 Mapping of Operators to States

An operator node can be mapped to one and only one of the states among the *mobility interval* (refer to Constraint 3). The *mobility interval* is the range of states onto which a node can possibly map to. In Section 4 we discuss more about how these intervals can be precomputed.

Consider for example the node $n6$ in Figure 1. Let us assume that the mobility interval for this node is states 3 to 5. This node can only be in one of these states and hence one and only one of $< g_{n_6 3}, g_{n_6 4}, g_{n_6 5} >$ can be 1.

Since these boolean state variables g_{ij} are mutually exclusive, we can compute the state to which a node gets scheduled using their weighted sum as shown in Constraint 4. For the node n_6 under the above assumptions

$$G_{n_6} = 3g_{n_63} + 4g_{n_64} + 5g_{n_65}$$

Since we assume that operators are not pipelined, each operator has to start and end in the same state. This is captured by the pair of constraints 5 and 6. The first one forces it to start after the beginning of a state and the second one ensures that it finishes before the state ends. For the node $n6$, these constraints expand to

$$\begin{aligned}
s_{n6} + N - Ng_{n_63} &\geq 2P + \gamma \\
s_{n6} - N + Ng_{n_63} &\leq 3P - \delta \\
s_{n6} + N - Ng_{n_64} &\geq 3P + \gamma \\
s_{n6} - N + Ng_{n_64} &\leq 4P - \delta \\
s_{n6} + N - Ng_{n_63} &\geq 4P + \gamma \\
s_{n6} - N + Ng_{n_63} &\leq 5P - \delta
\end{aligned}$$

The choice of the 'arbitrary constant' N is very crucial in ensuring faster termination of the MILP solver. In general it should be as small as possible but larger than any value s_{n6} could possibly take. As a thumb rule we use some number twice as large as the product of the maximum state the node can occupy and the minimum clock period. Refer to Section 4 for more details on some of the estimations.

3.3 Memory Access Sequencing

Since the memory is pipelined, two or more accesses to the memory can overlap except during the C_w initiation cycles. During this initiation phase (when address/data is asserted etc.) only one of the nodes can be in that phase. Another node can enter that phase either C_w cycles before the node grabs the memory or C_w cycles after the node starts the initiation phase. This is enforced by the pair of constraints 7 and 8 that use an auxiliary boolean variable z_{ij} to achieve this. It does not really matter whether node i or j grabs the memory first (i.e. whether z_{ij} is 1 or 0). But they should not enter the initiation phase together.

For example, the two memory read accesses $m3$ and $m4$ in Figure 1 need to be serialized in such a way that either $m3$ starts the access and $m4$ follows after at least C_w cycles.

3.4 Area Computation

The area requirement Θ of the entire design is the sum of the area requirements of the operator instances used plus other associated costs such as control path and interface logic. Though it is possible to have an elaborate area computation model, in the present work we compute it by counting the number of operators needed for the entire design and the average per state area requirement of the control path. Equation 9 counts the total number of operators of each type needed to synthesize the whole program.

In Figure 1, let us assume that $n3$ and $n4$ get mapped to say, state 3 ($g_{n_33} = g_{n_43} = 1$) and $n5$ and $n6$ get mapped to state 4 ($g_{n_54} = g_{n_64} = 1$). The total number of multipliers needed would be $\max((g_{n_33} + g_{n_43}), g_{n_54}) = 2$. The number of adders needed can be computed in a similar fashion.

The total area required is computed by Equation 10 where the first term accounts for the area requirements of the operators and the second term takes care of the control paths. The area needed by control paths is estimated as the product of the number of states and the average per state requirement. The number of states in a loop is same as the state to which the *last node* in a loop gets mapped.

3.5 Execution time

The total execution time Δ of the program is the sum of the products of per iteration time of each loop and the number of iterations in the loop as shown by Equation 11. The per iteration time is given by the start time of the *last node* with respect to the start time of the *start node* in each loop body.

3.6 Objective Function

Since our primary objective is to synthesize the design so that execution time is minimum, our objective function is just Δ that needs to be minimized. We could impose constraints on the area by specifying $\Theta \leq$ *limit*. Or we could even use area minimization as a *secondary objective* and minimize both Δ and Θ.

Multiple objective functions could be mutually conflicting in general but it is possible to prioritize or associate appropriate weights to these objectives. The priority or the weight indicates the relative importance of the goals.

If the multiple objective functions are prioritized, then we are effectively telling the MILP solver to minimize the highest priority objective function by 'relaxing' the other objective functions and then minimize the second priority objective function (relaxing the rest) in such a way that the first one is not violated and so on. For example, if COST1 and COST2 are two objective functions prioritized in that order, one way to achieve multiple objectives is to first ignore the

COST2, minimize the problem for COST1, then minimize the original problem for COST2 with an added constraint that COST1 should not be greater than the minimum value achieved in previous pass. The order in which the objectives are minimized depends on the order of priority.

In contrast to this, in a weighted multi-objective function case we associate weights with each objective function and create a new objective function which is a linear combination of original objective functions and minimize that. In the previous example we would have COST $= \alpha$ COST1$+\beta$ COST2 as the new objective function to be minimized, where α, β are suitable weights.

Some of the commercial MILP solvers [2] readily support both types of multi-objective functions and many more variations there of.

4 Improving the Solution Time

The number of Integer variables used in an MILP model often decide the solution time. In our MILP model, in the worst case, the number of integer variables required is $\mathcal{O}(N_o * S) + \mathcal{O}({N_m}^2)$ where N_o is the total number of operator nodes in the CDFG, S is the maximum possible states and N_m is the total number of memory access nodes. The first term is contributed by the state variables g_{ij} and the second term is contributed by the auxiliary variables z_{ij}. However, the average case complexity can be improved as follows.

4.1 Reducing the number of State variables g_{ij}

Since the problem we are addressing in this paper is not time constrained, there is really no upper bound on the number of states. However, there exists an upper bound on the number of states beyond which any increase in the number of states may not improve the execution time. We estimate this bound on the optimal number of states by adding the corresponding bounds for each of the loops.

Let us assume that we use a fast heuristic to synthesize a given loop and find that the execution time of the loop is T. Let us also assume that the minimum clock period (based on the slowest operator in the entire program) is P_{min}. Now, the upper bound S on the number of states in the loop is given by T/P_{min}. The reason is obvious.

We further reduce the number of state variables by eliminating the need for some of them. Assume that we estimated the upper bound on the number of states in loop 1 in Figure 1 as 7 and $C_t = 4$ for read and $C_t = 1$ for write operations. Let us say that we somehow found that the node $n4$ in Figure 1 can never be mapped to any state less than 5 and greater than 6. This range $< 5..6 >$ of possible states is called the mobility interval of the node $n4$. Knowing this interval, we don't really need the variables $g_{n_4 1}, g_{n_4 2}, g_{n_4 3}, g_{n_4 4}, g_{n_4 7}$ since they will be zero anyway. We effectively managed to get rid of 70% of the state variables needed for the node $n4$.

We compute this mobility interval as follows. We add the delays (in terms of cycles) corresponding to all the memory access predecessors of node $n4$ (i.e. $n2$ in our case). This addition is performed conservatively taking into consideration the fact that memory is accessed sequentially, memory is pipelined and two or more memory accesses can be overlapped. This sum gives the lower bound of the interval. Similarly, the upper bound is computed by summing the memory cycles corresponding to the successors and subtracting that from the upper bound on the number of states.

4.2 Reducing the Number of Auxiliary variables z_{ij}

In the extreme case we need a variable z_{ij} for each pair of memory access nodes i and j. However, we assume that the loops are executed in the same order as in the behavioral description (we don't try to exploit inter loop parallelism). This will automatically exclude the possibility of two memory operations in two different loops conflicting with each other. Similarly any pair of memory access nodes in a given loop that are *related* by a successor-predecessor relationship can never conflict. For example, in Figure 1, while nodes $m3$ and $m4$ can conflict, neither of them can conflict with $m7$ since $m7$ is the successor of both. This allows us to use as few as 1 auxiliary variable $z_{m_3 m_4}$ in our example.

It is possible to further cutdown the number integer variables needed by performing a more detailed preprocessing of the CDFGs. However, in the present work we restrict to the simple but effective methods described in the sections above.

5 Experimental Evaluation of Our Algorithm

We have implemented our algorithm in Java. The MILP model is solved using the XPRESS solver [2]. Synplify [3] is used for RTL level synthesis (all operations are assumed to be fixed point). Xilinx Virtex FPGA based board was chosen to be the target. In this section we present some of the results and compare and contrast our MILP based algorithm with a list scheduling based heuristic algorithm. The various benchmarks used are shown in Table 2.

Table 2: Benchmarks used in our evaluation of our algorithm.

	# of loop nests	Source
advc	2	SPEC95
dpdx	3	SPEC95
fcdf22	3	Honeywell ACS suite
fftgen	1	Honeywell ACS suite
glre	1	Livermore Loops
hydro	1	Livermore Loops
jacobi	1	RAW (MIT)
ex	2	Synthetic

5.1 No Resource Constraint Case

To start with, we assume single cycle non pipelined memory and no resource constraints. The synthesis results for both the algorithms under these conditions are shown in Table 3. In many cases the heuristic algorithm does indeed manage to find a solution within 10% of that found by MILP based algorithm.

Table 3: Synthesis results with no resource constraints and single cycle memory (M: MILP, H: Heuristic, PDT: % decrease in exec time, PDA: % decrease in area. Clock in MHz, Syn.Time is solution time in secs).

	Clock		# of States				Syn.Time	
	M	H	M	H	PDT	PDA	M	H
advc	45	48	9,6	10,7	6	30	*65	<1
dpdx	45	48	4,7,4	4,8,5	6	50	<1	<1
fcdf22	48	48	6,6,5	6,7,5	6	0	*67	<1
fftgen	48	48	11	11	0	20	*171	<1
glre	45	48	4	5	15	0	<1	<1
hydro	42	48	5	8	29	11	2	<1
jacobi	48	48	7	7	0	68	105	<1
ex	42	48	4,4	6,4	9	6	1	<1

However, this is at a price as we can see from the greater area used by the heuristic. On an average, MILP found 9% faster solutions using 23% less resources. We can also see that the heuristic always tries to synthesize at 48 MHz (the slowest operator being '*' in all cases), while the MILP often chooses a lower clock and still manages to achieve lower execution times. This is because, the number of states in the MILP solution are generally less than those in the heuristic. We talk about the synthesis times (last columns in Tables 3-6) later in section 5.4.

5.2 Identical Resource Constraint Case

The total number of resources used for a design is also important and it is only fair to compare the two algorithms under similar resource constraints. If we repeat the same experiment under identical resource conditions (i.e. the resources used by MILP), we can clearly see the superiority of the MILP over heuristic as shown in Table 4.

Table 4: Synthesis results with equal resources and single cycle non pipelined memory (M: MILP, H: Heuristic, PDT: % decrease in exec time. Clock in MHz, Syn.Time is solution time in secs).

	Clock		# of States			Syn.Time	
	M	H	M	H	PDT	M	H
advc	45	48	9,6	12,8	20	*65	<1
dpdx	45	48	4,7,4	5,9,5	16	<1	<1
fcdf22	48	48	6,6,5	6,7,5	6	*67	<1
fftgen	48	48	11	11	21	*171	<1
glre	45	48	4	5	15	<1	<1
hydro	42	48	5	9	37	2	<1
jacobi	48	48	7	10	30	105	<1
ex	42	48	4,4	6,5	17	1	<1

As can be seen, on an average MILP solution is 20% faster. The number of states in heuristic solution also increases further due to the resource constraint.

5.3 Multi-cycle Pipelined Memories

We now compare the two algorithms in the context of multi-cycle pipelined memories.

As can be seen in Tables 5 and 6, the MILP algorithm still fares better by roughly 18% in both the cases. As in the case of non-pipelined memory, MILP solution generally uses less number of states and employs a lower clock frequency.

Table 5: Synthesis results with equal resources and two cycle pipelined memory ((M: MILP, H: Heuristic, PDT: % decrease in exec time. Clock in MHz, Syn.Time is solution time in secs).

	Clock		# of States			Syn.Time	
	M	H	M	H	PDT	M	H
advc	45	48	10,7	11,8	5	*170	< 1
dpdx	45	48	5,8,5	7,12,7	26	39	< 1
fcdf22	48	48	6,7,5	6,7,6	5	14	< 1
fftgen	45	48	11	15	22	*2	< 1
glre	45	48	5	7	24	< 1	< 1
hydro	42	48	6	9	24	1	< 1
jacobi	48	48	8	11	27	127	< 1
ex	48	48	5,5	7,5	5	< 1	< 1

Table 6: Synthesis results with equal resources and four cycle pipelined memory ((M: MILP, H: Heuristic, PDT: % decrease in exec time. Clock in MHz, Syn.Time is solution time in secs).

	Clock		# of States			Syn.Time	
	M	H	M	H	PDT	M	H
advc	45	48	12,9	15,10	10	*179	< 1
dpdx	45	48	7,10,7	7,11,8	20	*52	< 1
fcdf22	48	48	8,9,7	8,9,7	4	8	< 1
fftgen	45	48	13	14	37	*67	< 1
glre	45	48	7	8	17	< 1	< 1
hydro	42	48	8	11	30	< 1	< 1
jacobi	48	48	10	10	23	140	< 1
ex	42	48	7,7	9,7	1	1	< 1

5.4 The Synthesis Time

In the previous sections, it was evident that our MILP based algorithm is superior to the heuristic algorithm both in terms of the execution time of the synthesized design as well as its resource requirement. However, if we consider the synthesis time itself (last columns in Tables 3-6), the heuristic algorithm has an upper hand. In all cases, the heuristic algorithm took less than a second to arrive at a solution. Whereas the MILP algorithm took much longer and did not terminate in many cases. This is one of the serious drawbacks of MILP based solutions.

Fortunately, XPRESS MILP solver we used allows us to specify a timeout after which the solver terminates with whatever best solution was found. These cases are marked by the the asterisk in the last columns of Tables 3-6. The time indicated in these cases was the time at which the solution was found. It is interesting to note that good solutions were found early on and they were far superior to those found by the heuristic. We also observed that the solution found in such cases differed by less than 10% from the bound computed by the solver making it very close to the optimal. It is quite possible that a tighter bound could have terminated the search and the solution found was indeed optimal.

6 Conclusion

In this paper we made some interesting observations in the context of synthesizing array intensive computations onto FPGAs. We proposed an algorithm based on MILP. Our algorithm tries to minimize execution time as well as area while arriving at the best clock, number of states and the memory access sequence. We compared and contrasted the relative strengths of our algorithm with a heuristic based algorithm.

References

[1] Annapolis Micro Systems, *WILDSTAR^{TM} Reconfigurable Computing Engines.*

[2] Dash Associates, *XPRESS-MP : User Guide and Reference Manual*, 1997.

[3] Synplicity Inc, *Synplify : VHDL and Verilog Synthesis for FPGAs and CPLDs.*

[4] D.D. Gajski, N. Dutt, A. Wu and S.Lin, *High Level Synthesis : Introduction to Chip and System Design*, Kluwer Academic Publications.

[5] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill Inc.

[6] D. Knapp, T. Ly, D. MacMillen and R.Miller, *Behavioral Synthesis Methodology for HDL-Based Specification and Validation*, Proc. of DAC'95.

[7] J.Babb et.al., *Parallelizing Applications into Silicon*, Proc. of FCCM'99.

[8] G. Lakshminarayana, K. Khouri and N. Jha, *Wavesched : A Novel Scheduling Technique for Control-flow Intensive Behavioral Description*, Proc. of ICCAD'97.

[9] O. Bringmann and W. Rosentiel, *Resource Sharing in Hierarchical Synthesis*, Proc. of ICCAD'97.

[10] S. Raje and R. Bergamaschi, *Generalized Resource Sharing*, Proc. of ICCAD'97.

[11] A. Chowdhary and J. Hayes, *Technology Mapping for Field-Programmable Gate Arrays Using Integer Programming*, Proc. of ICCAD'95.

[12] M. Kaul, R. Vemuri, S. Govindarajan and I. Quaiss, *An Automated Temporal Partitioning and Loop Fission Approach for FPGA based Reconfigurable synthesis of DSP Applications*, Proc. of DAC'99.

[13] P. Panda, N. Dutt and A. Nicolau, *Exploiting off-chip Memory Access Modes in High-Level Synthesis*, Proc. of ICCAD'97.

[14] L. Semeria and G. Micheli, *SpC : Synthesis of Pointers in C : Application of pointer analysis to the behavioral synthesis from C*, Proc. of ICCAD'98.

[15] S. Ravi, G. Lakshminarayana and N. Jha, *Removal of Memory Access Bottlenecks for Scheduling control-flow Intensive Behavioral Descriptions*, Porc. of ICCAD'98.

[16] M. Gokhale and J. Stone, *Automatic Allocation of Arrays to Memories in FPGA Processors with Multiple Memory Banks*, Sarnoff Corporation.