Register Requirement Minimization of Fixed-Depth Pipelines for Streaming Data Applications

Thomas Goldbrunner*, Nguyen Anh Vu Doan*, Diogo Poças†, Thomas Wild*, and Andreas Herkersdorf*
*Chair of Integrated Systems
†Chair of Operations Research
Technical University of Munich
Arcisstr. 21, 80333 Munich, Germany
Email: {thomas.goldbrunner, anhvu.doan, diogo.pocas, thomas.wild, herkersdorf}@tum.de

***Abstract*—We present a method that can be used to map control/data flow graphs into fixed-depth pipelines targeted at FPGA design. The main objective for the design is to reduce the register resources which are needed to forward data within the processing pipeline. We show that these requirements can be reduced by appropriate task scheduling. Starting from an intuitive network flow based scheduling approach, we develop a linear programming model of the task scheduling problem. This allows us to efficiently create schedules which are provably optimal with regard to the objective of minimal register usage.**

I. Introduction

Ever since their introduction to the market, FPGAs have gained importance in the field of computing architectures. As Trimberger shows in detail in [1], FPGAs have been used to create specialized deeply-pipelined massively parallel designs for the processing of streaming data, especially since the early 2000s. The application domains range from classical communications applications to the control of adaptive optic systems for telescopes such as the Very Large Telescope (VLT) operated by the European Southern Observatory (ESO) [2].

Typically, the complete design for one of these applications consists of several processing pipelines that are partly executed in parallel and partly consecutively to perform the desired operation on the data that is streamed through these pipelines. Pipelines executed in parallel need to be synchronized so that their respective output, i.e. the input for the following pipeline, is produced from the same input data. While this can be achieved using synchronization FIFOs, for our designs we chose to enforce synchronization by a fixed depth of all parallel pipelines. Also, for pipelines that hold application parts that consist of several branches the fixed depth pipelines make sure that the results are always created after a fixed amount of clock cycles, regardless of which branch is actually executed. Therefore, to create an architecture with predictable timing, in this work we assume that application (sub-)graphs need to be mapped to fixed depth pipelines.

Such deeply pipelined designs can satisfy high demands on throughput and computational density. For example a pipeline with 256 stages and an input data width of $256\,\mathrm{bits}$ clocked at a non-aggressive $125\,\mathrm{MHz}$ yields up to $32\,\mathrm{Gbps}$ I/O throughput and $8\,\mathrm{Tbit/s}$ of processed data, corresponding to a $256\,\mathrm{GOPS}$ equivalence of 32 bit RISC instructions.

Another trend in recent years is that the amount of data that needs to be processed is increasing significantly faster than the computational density, which leads to the so called *data deluge* [3]. For the computing architectures, this means that memory is becoming the bottleneck and that new data processing applications need to be developed to handle the sheer amount of data [4]. In order to exploit the advantages that deeply-pipelined FPGA designs have over other compute platforms, the data needs to be kept close to the processing. A typical modern high-end FPGA provides several layers of on-chip and in-package memory ranging from flip-flops in the FPGA fabric over block RAM to high-bandwidth memory (HBM). Compared to the increasing data load that needs to be processed, the memory resources closest to the FPGA fabric, including flip-flops, are relatively limited, so one goal for the system design is to use the available memory resources as efficiently as possible.

In this paper, we propose a linear program (LP) model for the scheduling problem in fixed-depth pipelines and detail how it can be optimally solved so that tasks are placed within the pipeline in such a way that the overall register demand for data forwarding within the pipeline is minimized. The experimental results show that our approach leads to a significant reduction of the register requirements, when compared to classical scheduling approaches. So, for data heavy applications where performance is more often limited due to bottlenecks when accessing data than through computational performance, this reduction of register requirements combined with efficient data placement methods should lead to an improved performance.

The rest of the paper is structured as follows. In section II we present related work. The problem statement and observations from preliminary analysis are presented in III. Our proposed method for register minimization is introduced in section IV. The feasibility of our method is demonstrated with the experimental results in section V. Section VI concludes the paper and presents an outlook on future work.

II. Related work

A. Using network flow techniques for FPGA design

Cong and Ding made use of network flow techniques in their Flow-Map technology mapping algorithm [5]. They represented the Boolean network that needs to be mapped to the FPGA lookup tables (LUT) as a directed acyclic graph. On this graph, they used a two-phased mapping algorithm which applies a graph cut based approach to find an optimal mapping of graph nodes to LUTs.

In [6] Yang and Wong exploited the similarity between the problems faced in scheduling of network flow processing and the partitioning of FPGAs. They proposed a method to model the netlist of an FPGA design by a network flow and a heuristic that allows to use the max-flow min-cut technique to create a *min-cut balanced bi-partition*, which is a partition that splits a circuit into two equally sized disjoint sub-components and minimizes the nets that are needed to connect the two sub-components.

Tan et.al. [7] presented a scheduling approach for the use in high-level synthesis (HLS) based on cuts of acyclic graphs. They used this approach to generate schedules that minimize latency and reduce area requirements. They achieved this by simultaneously considering the scheduling and the technology mapping, which provides more accurate estimates of an operations execution time and therefore allows a more aggressive operation chaining.

While these approaches operate on different levels of FPGA design and are not directly related to register optimization, they show that it can be a valid approach to use network flow techniques for partitioning or, in our case, scheduling problems. Indeed, this idea is not new: results in this direction can already be found in the work of Fulkerson [8], Kelley [9], and Levner and Lemirovsky [10]. The mathematical formulation of our problem is also very similar to the just-in-time project scheduling problem [11, Sec. 19.5].

B. Optimizing memory aspects of FPGA designs

The work presented in [12] is related to memory optimization for HLS for FPGAs. However, Gallo et.al. focused on the specific problem of partitioning data arrays into memory banks and proposed a partitioning technique that allows to minimize the area overhead that is introduced due to control logic for memory accesses.

The work presented by Atat and Ouaiss in [13] shows how on-chip embedded memory of modern FPGAs can be used as registers instead of the flip-flops in logic cells. Their technique, called *Memory Binding*, uses the memory banks of the embedded memory to store data in a manner that avoids conflicting accesses to the memory banks. With this approach they increase the amount of logic cells which are available to implement processing while increasing the utilization of the embedded memory.

What these approaches have in common is that they focus on the actual data placement in the FPGA design, whereas our approach aims to reduce the amount of data that needs to be stored in the first place.

III. Problem Statement and Preliminary Analysis

As already stated in section I, our goal is to create register-optimized fixed-depth pipeline implementations on a FPGA target for a given task graph. The approach we consider in this paper is the reduction of used register resources due to appropriate task scheduling within the pipeline. We will now formalize the problem statement and present the findings of preliminary analysis before introducing our scheduling approach in section IV.

A. Problem statement

Let $G = \{T, E\}$ be a directed graph consisting of a set of n tasks $T = \{T_i \mid i = 0, 1, 2, \ldots, n-1\}$ and a set of edges $E = \{E_{ij}\}$ where E_{ij} exists between two tasks (T_i, T_j) if these tasks exchange data with a certain probability $p_{ij} > 0$. This therefore defines the control/data flow graph (CDFG) of an application. The exchanged data between two tasks T_i and T_j is denoted D_{ij} and the corresponding data width $w\left(D_{ij}\right)$.

In our pipelined design, each task T_i will be executed in one pipeline stage which is expressed as $S\left(T_i\right) \in \{0, 1, 2 \ldots d-1\}$ where d is the depth of the pipeline which we consider as an application specific constraint given by the target implementation. Due to the inherent parallel architecture of the FPGA, there is no hard limit (except for the overall number of available resources on the FPGA) for the amount of tasks that can be executed in parallel in one pipeline stage. In contrast, the amount of tasks that can be executed sequentially within one pipeline stage is strictly limited by the maximum depth of the combinational path allowed by the operating frequency, i.e. the setup constraint of register timing. The sum of execution times $t\left(T_i\right)$ of all T_i that are executed sequentially in one pipeline stage must not exceed this limit.

From these definitions, the amount of register resources that are needed to forward the data between pipeline stages can be expressed as:

$$\#D = \sum_{i=0}^{n} \sum_{j=0}^{n} \Delta T\left(D_{ij}\right) \cdot w\left(D_{ij}\right), \tag{1}$$

where $\Delta T\left(D_{ij}\right) = S\left(T_j\right) - S\left(T_i\right)$ is the *lifetime* of the data exchanged between T_i and T_j, i.e. the number of pipeline stages that D_{ij} needs to be propagated through. As defined earlier, $w\left(D_{ij}\right)$ is the bit width of the data that is exchanged over edge E_{ij}, so for all combinations of i and j for which no edge exists this width is 0.

The goal of our work is to minimize $\#D$ for a given graph G by choosing a proper $S\left(T_i\right)$ for all tasks in G, while maintaining a valid schedule which respects all data dependencies between the tasks inside the pipeline.

B. Scheduling Basics

The two extremes, when statically scheduling a task graph to a fixed-depth pipeline, are the *as soon as possible* (ASAP) and the *as late as possible* (ALAP) schedules. In the ASAP case the scheduling starts from pipeline stage 1. For each task T_i, it is checked whether all tasks that produce input data for this task have already been scheduled. If this is the case, it is checked whether the current task can be added to the current stage without violating the constraint of maximum sequential execution time. Then the task will either be scheduled in the current stage or, if this is not possible, in the next stage.

This process is repeated iteratively until all tasks have been assigned to a pipeline stage. In our case of fixed-depth pipelines, the task(s) that produce the graphs output are always scheduled in stage d.

For illustrative purposes, we use the graph shown in figure 1a. It consists of the tasks $\{T_i \mid i = 0, 1, \ldots, 8\}$ with the corresponding edges between the tasks. Furthermore, without loss of generality, we assume that each task T_i has an execution time $t\left(T_i\right)$ that does not allow to execute more than one task consecutively within a given pipeline stage. The data width $w_{i,j}$ for all edges is given in the figure.

The ASAP schedule created for the graph under the given assumptions for a pipeline depth of $d = 8$ is listed in table I.

The ALAP schedule can be built similarly to the ASAP schedule with the difference that the process is started with the last stage and for each task it is checked whether all succeeding tasks have already been scheduled. The ALAP schedule for the example graph can also be found in table I.

The mobility $\mu_i = \mu\left(T_i\right)$ of a task T_i is defined as the number of possible pipeline stages where T_i can be executed in a valid schedule. It can be expressed as $\mu_i = S_{\mathrm{ALAP}}\left(T_i\right) - S_{\mathrm{ASAP}}\left(T_i\right)$. The implication of a $\mu_i > 0$ is that, when creating a schedule, there are several pipeline stages that T_i can be mapped to. So we can influence $S\left(T_i\right)$ and therefore the ΔT for all data that is either produced or consumed by T_i. As shown in (1) the overall amount of data that needs to be propagated within the pipeline is given by the sum of $\Delta T\left(D_{ij}\right) \cdot w\left(D_{ij}\right)$ for all data that is exchanged between the tasks of the graph. As the widths of the edges are fixed, we can only minimize $\#D$ through the appropriate selection of $S\left(T_i\right)$ for each task.

C. Minimum Cut Scheduling

A first rather intuitive approach that can be considered comes from the *max-flow min-cut theorem* [14]. This provides us with two sub-graphs that are chosen in such a way that the weights of the edges between the two sub-graphs are minimal. We then scheduled the first sub-graph in an ASAP fashion and the second in ALAP fashion, so that the data forwarded through empty pipeline stages has minimal width.

This approach is very similar to the multiprocessor scheduling solution presented in [15] where it was proven that the max-flow min-cut theorem can be used to find an optimal scheduling of a task graph to two different processors based on the edge weights and also similar to methods such as [16], [17] and [18]. However, we are not scheduling tasks to fixed processors, but interpret the two sub-graphs as *schedule groups* which we then use to create a register-optimized schedule for a pipelined FPGA architecture.

When we apply this method to the example graph from figure 1a, the min-cut leaves us with the following two task sets:

1) `T0, T1, T2, T3, T4, T5`
2) `T6, T7, T8`

The first set which contains the tasks *above* the cut will be scheduled in ASAP fashion and the tasks *below* the cut will be scheduled ALAP. This gives us the *min-cut schedule* as shown in table II. Using the definition in (1), we can obtain the amount of register resources needed for each of the schedules as $\#D_{\mathrm{ASAP}} = 416\,\mathrm{bits}$, $\#D_{\mathrm{ALAP}} = 760\,\mathrm{bits}$ and $\#D_{\mathrm{min-cut}} = 400\,\mathrm{bits}$.

i	0	1	2	3	4	5	6	7	8
$S_{\mathrm{ASAP}}\left(T_i\right)$	0	1	1	2	2	3	3	4	7
$S_{\mathrm{ALAP}}\left(T_i\right)$	0	6	3	4	4	5	5	6	7

TABLE I
ASAP AND ALAP SCHEDULE FOR FIG. 1A ($d = 8$)

i	0	1	2	3	4	5	6	7	8
$S_{\mathrm{min-cut}}\left(T_i\right)$	0	1	1	2	2	3	5	6	7

TABLE II
MIN-CUT SCHEDULE FOR FIG. 1A ($d = 8$)

However, it turned out that the min-cut approach can lead to schedules that are invalid as they do not respect data dependencies. The graph shown in figure 1b and the corresponding min-cut schedule in table III illustrate this problem. For example, task T_1 is scheduled in stage 3 whereas task T_2, which depends on input data from T_1, is scheduled in stage 1, so data dependencies cannot be fulfilled.

i	0	1	2	3	4	5	6	7
$S_{\mathrm{min-cut}}\left(T_i\right)$	0	3	1	4	2	5	3	6

TABLE III
MIN-CUT SCHEDULE FOR FIG. 1B ($d = 7$)

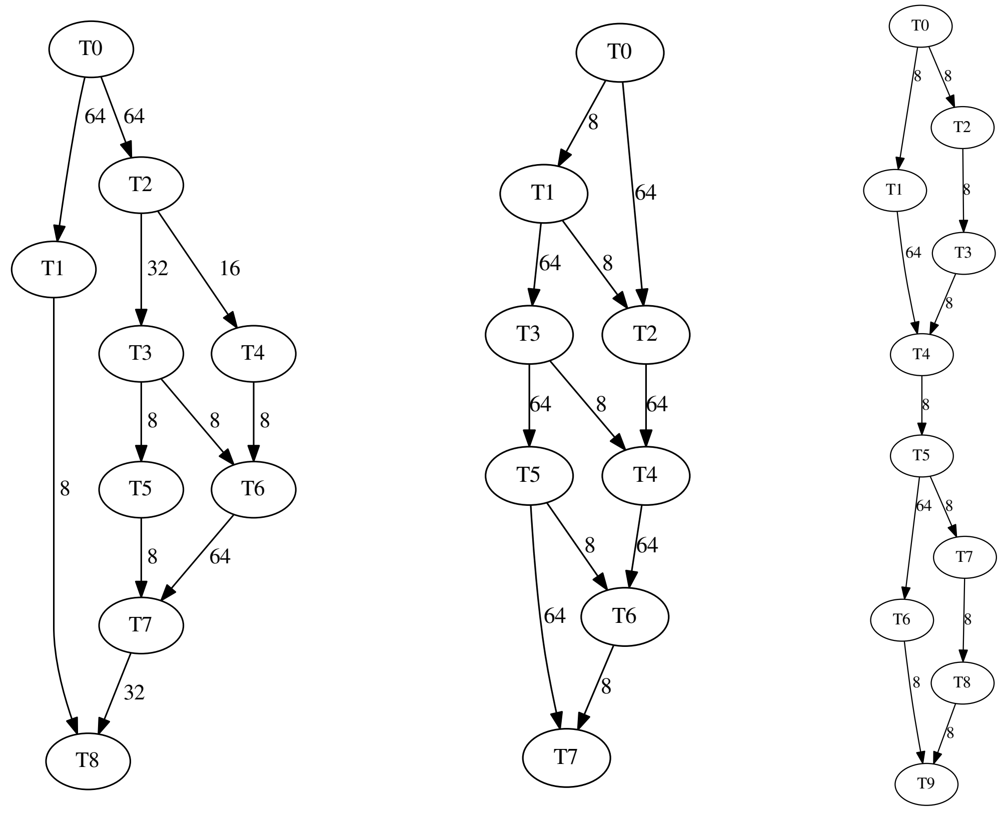
D. Minimum Dicut Scheduling

To prevent the creation of these invalid schedules, the *minimum dicut* instead of the minimum cut can be used [19]. The difference between a cut and a dicut is that the dicut takes into account the direction of edges and does not allow the creation of backwards edges. Using the min-dicut we can obtain a valid schedule for the graph from figure 1b as shown in table IV.

i	0	1	2	3	4	5	6	7
$S_{\mathrm{min-dicut}}\left(T_i\right)$	0	1	2	2	3	3	4	6

TABLE IV
MIN-DICUT SCHEDULE FOR FIG. 1B ($d = 7$)

While the min-dicut solves this problem, unfortunately for certain graphs it can lead to schedules that in the end use more register resources than the simple ASAP or ALAP schedules. For example the min-dicut schedule for the graph shown in figure 1c leads to a register requirement of $336\,\mathrm{bits}$ whereas the ASAP schedule needs $288\,\mathrm{bits}$. The problem here is that while the min-dicut minimizes the edge widths between the two subgraphs generated by the cut, the data propagation within the subgraphs is not optimal. One approach to handle this problem would be to apply the min-dicut scheduling method in a recursive fashion. However, we chose another path which enables us to formally prove the optimality of the created schedules.



(a) (b) (c)

Fig. 1. Example graphs (edge widths in bit)

IV. Proposed Method

With a deeper analysis, it appears that we can exploit structural properties of our problem in order to design algorithms which are both efficient and provably optimal. If we introduce a variable $S(T_i)$ denoting the execution stage of the i-th task as in Section III-A, then our goal is to solve the following integer program.

$$\min \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} w(D_{ij})(S(T_j) - S(T_i)) \quad \text{s.t.} \tag{2}$$

$$S(T_j) - S(T_i) \geq 1 \quad \text{for each edge } E_{ij}, \tag{3}$$

$$S(T_0) = 0, \tag{4}$$

$$S(T_{n-1}) = d - 1, \tag{5}$$

$$S(T_i) \in \{0, \ldots, d-1\} \quad \text{for each task } T_i. \tag{6}$$

Notice that we have a constraint of type (3) for each edge in the directed graph G, which enforces that task T_j is executed after task T_i. Constraints (4) and (5) enforce the scheduling stage of the first and last tasks, whereas (6) enforces that each task must be executed at some pipeline stage within the depth limit.

Using classical results in the theory of combinatorial optimization (see e.g. [20, Ch. 19] for a detailed exposition of this reasoning), we can first observe that the constraint matrix associated with (3) is the (transpose of the) node-arc incidence matrix of G; thus, it is a totally unimodular matrix (i.e. each of its subdeterminants is 0, $+1$ or -1).

Now let us consider the linear relaxation of our problem, which is obtained by replacing the constraints of type (6) with

$$0 \leq S(T_i) \leq d - 1 \quad \text{for each task } T_i. \tag{6'}$$

Since (3) defines a totally unimodular matrix, so does the system (3)+(4)+(5)+(6'). As an immediate consequence, all vertices in the corresponding polytope are integral. This implies that an optimal vertex of the linear relaxation will also be an optimal solution to the original problem. As linear programming can be solved efficiently, this approach is both efficient and provably optimal (since it can be solved with the simplex algorithm which provides guarantee of optimality).

It is also worth mentioning that one could design combinatorial algorithms via linear programming duality. Indeed, by considering the dual program of our linear relaxation, we can write it down as a minimum cost flow problem, for which efficient network simplex algorithms are known to exist [11, Ch.11].

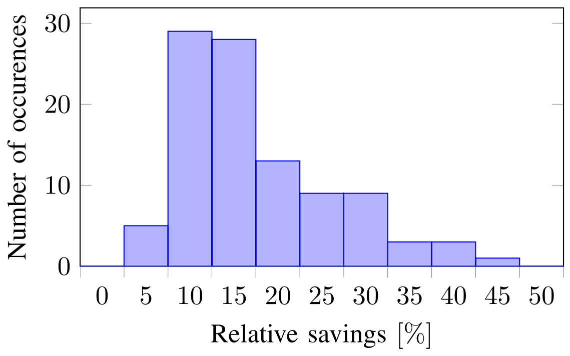


Fig. 2. Relative savings for 100 random graphs $(n = 50, p = 0.1, d = 64)$

V. Experimental results

A. Experimental setup

To illustrate the effect of the register minimizing scheduling we implemented a simulation framework in Python. This framework provides methods to create ASAP, ALAP and optimized schedules for a given pipeline depth and graph. For our tests, we used random directed graphs generated with the Erdős-Rényi $G(n, p)$ model [21]. The graphs generated by this model can be tuned using the two parameters n and p, where n is the number of nodes in the graph and p defines the probability with which each of the possible edges between the graph nodes is included in the graph. The data width for each edge is randomly selected from the set $\{8\,\text{bits}, 16\,\text{bits}, 32\,\text{bits}, 64\,\text{bits}\}$. By carefully selecting the values for n and especially p it is possible to create random graphs that have properties similar to typical CDFGs. Besides, without loss of generality, we consider the graphs to have a single source and a single sink node. We ran simulations for several values of n, p and the pipeline depth (while making sure that the pipeline depth is sufficient to hold the graph). In the following subsections we present our findings regarding the effects of the different parameters on the performance of our approach. In each subsection we give further details on how we chose the variables for our investigations.

B. Comparison to ASAP

The graph in figure 2 shows the distribution of relative savings regarding the number of register bits compared to ASAP schedules. The numbers were obtained for a set of 100 different random graphs. The graphs have been generated with the parameters $n = 50$ and $p = 0.1$ and the schedules for these graphs have been created for a pipeline depth of 64 stages.

Statistical observations show that the maximum difference found in this set is a requirement for the optimal LP approach of 10 576 bits compared to 18 152 bits in the ASAP case, which is a relative reduction of 41%. The minimal observed reduction for a given graph was found when comparing the LP approach (12 248 bits) with the ASAP case (12 600 bits), which is a relative reduction of 2.7%. The average reduction for this sample set of graphs is 15.3% with a standard deviation of 8.7%.

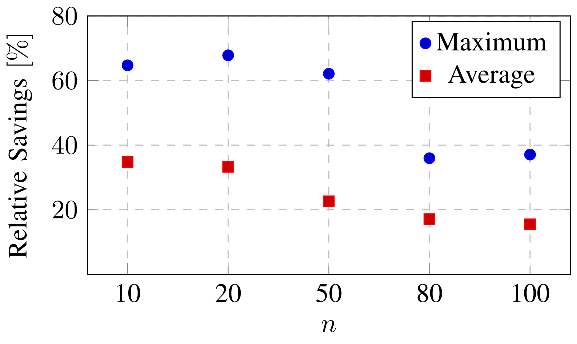


Fig. 3. Relative savings for different values of n with $p = 0.1$

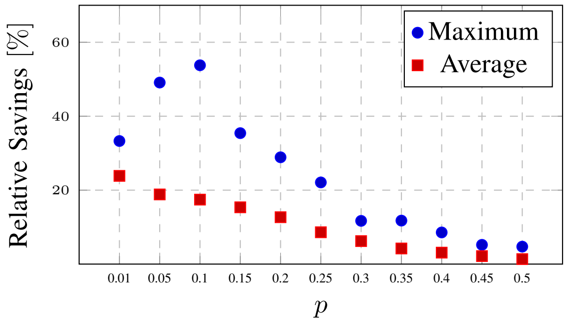


Fig. 4. Relative savings for different values of p with $n = 50, d = 64$

C. Influence of the number of nodes in a graph

To compare the performance of our approach for graphs of different sizes, we analyze the results shown in figure 3. The depicted information was obtained for graphs created with different values of n. To make sure that each graph can be scheduled, we also had to adapt the pipeline depth accordingly. So the numbers shown here are for the following (n, d)-tuples: $(n = 10, d = 16)$, $(n = 20, d = 32)$, $n = 50, d = 64)$, $(n = 80, d = 96)$, $(n = 100, d = 128)$. The value of $p = 0.1$ was kept constant. For each parameter set we performed simulations for 100 random graphs. The data shows a trend that the maximum and average relative resource savings are larger for smaller numbers of n.

D. Influence of the graph density

To inspect the effect that the graph density, i.e. the amount of edges in the graph has on the register optimized scheduling, we performed simulations for different values of p. As mentioned before in the $G(n, p)$ model, the parameter p is the probability with which each of the possible edges between the graphs nodes is actually included in the graph. So the lower the value for p, the lower is the number of edges in the graph. For example with $p = 0.01$ the average amount of edges for a graph with 50 nodes is 12.5 whereas for $p = 0.5$ the average number of edges is 611.

Figure 4 shows the relative savings obtained for different values of p in the range from 0.01 to 0.5. Once again the relative reduction is compared to the ASAP schedule. The

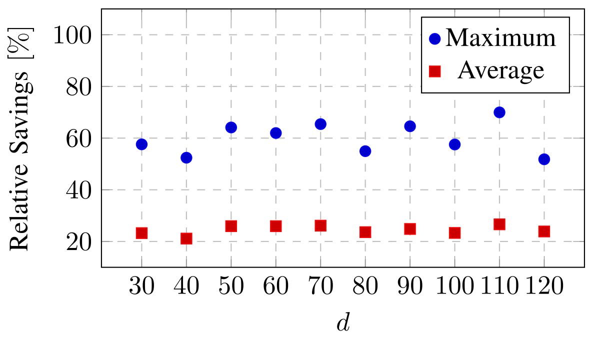


Fig. 5. Relative savings for different values of d with $n = 20,\ p = 0.1$

values presented here are based on simulations with 100 different graphs per value of p.

It is clear from the figure that the LP-based schedule performs much better for graphs with a low density. However, this is exactly the kind of graphs that a typical CDFG is a representative of. In these graphs the nodes are typically not connected to a large number of others, but each node has a very limited number (< 10) of predecessors and successors. Therefore, the proposed approach is well-suited to create pipeline schedules for real-life application graphs that will minimize the register resource requirements.

E. Influence of the pipeline depth

Figure 5 shows the relative savings obtained for random graphs with 20 nodes, an edge probability of $p = 0.1$ and 10 different pipeline depths ranging from 30 to 120. The maximum saving is always above 50% for the different pipeline depths and, in average, the savings are above 20%. Overall the pipeline depth does not have a significant impact on the proposed scheduling method.

In conclusion, with the findings from our experimental results, we can say that our proposed scheduling method leads to significant savings in register resources compared to ASAP and ALAP schedules and is well-suited for graphs with a limited number of nodes and a low graph density, both of which are properties found in typical CDFGs.

VI. Conclusion

In order to enable efficient FPGA implementations of streaming applications that operate on high-volume data, it is key to make efficient use of the limited memory resources available on the FPGA. In this paper we presented a linear model which can be solved optimally to reduce the register resource requirements for fixed-depth pipeline designs. This can be seen as a first step towards an overall memory optimized FPGA implementation. As a next step in our ongoing research, we strive to optimize the mapping of data to the different types of memory available on the FPGA board (Flip-Flops, BRAM, DRAM). This mapping needs to be performed in a way that optimizes the utilization of the available bandwidth for each memory type. The combination of reduced data forwarding and optimized bandwidth utilization will then lead to an overall memory optimized design.

VII. Acknowledgements

The research of Diogo Poças was supported by the Alexander von Humboldt Foundation with funds from the German Federal Ministry of Education and Research (BMBF).

References

[1] S. M. Trimberger, “Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology,” *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, March 2015.

[2] S. Goodsell, E. Fedrigo, N. Dipper, R. Donaldson, D. Geng, R. Myers, C. Saunter, and C. Soenke, “FPGA developments for the SPARTA project,” in *Astronomical Adaptive Optics Systems and Applications II*, vol. 5903. International Society for Optics and Photonics, 2005, p. 59030G.

[3] I. Gorton, P. Greenfield, A. Szalay, and R. Williams, “Data-intensive computing in the 21st century,” *Computer*, vol. 41, no. 4, pp. 30–32, 2008.

[4] K. Slavakis, G. B. Giannakis, and G. Mateos, “Modeling and optimization for big data analytics: (statistical) learning tools for our era of data deluge,” *IEEE Signal Processing Magazine*, vol. 31, no. 5, pp. 18–31, 2014.

[5] J. Cong and Y. Ding, “An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs,” in *The Best of ICCAD*. Springer, 2003, pp. 235–248.

[6] H. H. Yang and D. Wong, “Efficient network flow based min-cut balanced partitioning,” in *The Best of ICCAD*. Springer, 2003, pp. 521–534.

[7] M. Tan, S. Dai, U. Gupta, and Z. Zhang, “Mapping-aware constrained scheduling for LUT-based FPGAs,” in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 190–199.

[8] D. R. Fulkerson, “A network flow computation for project cost curves,” *Management Science*, vol. 7, no. 2, pp. 167–178, 1961.

[9] J. E. Kelley Jr, “Critical-path planning and scheduling: Mathematical basis,” *Operations research*, vol. 9, no. 3, pp. 296–320, 1961.

[10] E. V. Levner and A. Nemirovsky, “A network flow algorithm for just-in-time project scheduling,” *European Journal of Operational Research*, vol. 79, no. 2, pp. 167–175, 1994.

[11] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows*. Prentice-Hall, 1988.

[12] L. Gallo, A. Cilardo, D. Thomas, S. Bayliss, and G. A. Constantinides, “Area implications of memory partitioning for high-level synthesis on FPGAs,” in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014, pp. 1–4.

[13] H. A. Atat and I. Ouaiss, “Register binding for FPGAs with embedded memory,” in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2004, pp. 165–175.

[14] J. Edmonds and R. M. Karp, “Theoretical improvements in algorithmic efficiency for network flow problems,” *Journal of the ACM (JACM)*, vol. 19, no. 2, pp. 248–264, 1972.

[15] H. S. Stone, “Multiprocessor scheduling with the aid of network flow algorithms,” *IEEE Transactions on Software Engineering*, vol. SE-3, no. 1, pp. 85–93, Jan 1977.

[16] T. A. Johnson, R. Eigenmann, and T. Vijaykumar, “Min-cut program decomposition for thread-level speculation,” in *ACM Sigplan Notices*, vol. 39, no. 6. ACM, 2004, pp. 59–70.

[17] T. Kis, “A branch-and-cut algorithm for scheduling of projects with variable-intensity activities,” *Mathematical programming*, vol. 103, no. 3, pp. 515–539, 2005.

[18] R. H. Etkin, F. Parvaresh, I. Shomorony, and A. S. Avestimehr, “Computing half-duplex schedules in gaussian relay networks via min-cut approximations,” *IEEE Transactions on Information Theory*, vol. 60, no. 11, pp. 7204–7220, 2014.

[19] M. Grötschel, L. Lovász, and A. Schrijver, *Geometric algorithms and combinatorial optimization*. Springer, 1988.

[20] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[21] P. Erdős and A. Rényi, “On random graphs I,” *Publ. Math. Debrecen*, vol. 6, pp. 290–297, 1959.