In Proceedings of the IEEE International Workshop on System on Chip for Real Time Processing, Banff, Canada, July 2004

Interconnect Synthesis for Systems on Chip

Neal K. Bambha and Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering
and Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland, USA
{nbambha,ssb}@eng.umd.edu

Abstract

We describe an algorithm for performing a joint scheduling/interconnect synthesis optimization for System-on-Chip (SoC) architectures. The algorithm is able to account for different distributions of long vs. short interconnect routes in an architecture. It is based on a genetic algorithm, and utilizes a graph isomorphism test to significantly pare the search space and increase the search efficiency.

1. Introduction

Interconnect synthesis is important for today's system-on-chip (SoC) designs. As transistor density increases, more functional units can be placed on a single chip, and the number of possible interconnections (links) between them increases. The longest wires on the chip are usually due to these links. These wires contribute to delay and limit the maximum achievable clock rate. Also, routing these interconnections is a significant challenge for the EDA tools. A number of today's architectures for SoC provide special routing tracks for long interconnects. Future architectures may even incorporate optical interconnects. In this paper, we develop methods for deriving efficient interconnection networks in these architectures. The idea is that if we can incorporate routing constraints in the high level front end design stage, placement and routing can be improved in the back end of the design process and performance will increase.

Embedded systems typically run a limited and fixed set of applications. We can use this application-specific information to optimize the interconnection network. For our purposes, an optimal network is defined in the context of a set of applications and constraints. The constraints may include the latency, throughput, and power consumption for the given applications, along with cost and area constraints of the overall system. A key distinguishing feature to our algorithm is that we perform the application scheduling and interconnect synthesis jointly—most existing interconnect synthesis algorithms assume a given schedule.

2. Routing Constraints

Most FPGA designs use a heirarchy of interconnect segments of differing lengths. In the Atmel AT40K, for example, each logic cell connects directly to its nearest neighbors through fast dedicated local paths. These cells are nested in a mesh of longer-range interconnect buses spanning eight cells.

Several research groups have proposed *pipelined* FPGA architectures [3, 9] which provide for long interconnects through a large number of registers. For example, the RaPid architecture is targeted to high-throughput applications like those found in DSP. The interconnect structure consists of both long track and short track interconnects. Short tracks are used to achieve local connectivity between functional units, while long tracks traverse longer distances along the datapath. It is shown in [7] that the area-delay product in this architecture is sensitive to the short track / long track ratio.

In a system utilizing optical interconnects, cost and area constraints dictate the total number of transmitters and receivers in the system (i.e., total number of optical links). Routing constraints from local partitions to their associated VCSEL transmitters and detectors dictate a maximum fanout for each local partition. An optimum interconnect is then one that minimizes the number of links while enabling the application to meet the power, latency, and throughput constraints.

Our general model for a system-on-chip (SoC) is one in which the chip is partitioned into regions that are connected with local interconnects, and these local regions are then connected through longer global interconnects. The applications consist of *task graphs* [11], where the individual tasks must fit fully into a local region. The graph vertices (*tasks* or *nodes*) in the acyclic task graphs represent computations while the edges represent the communication of a packet of data from a source task to a sink task. Previous work [2] has addressed global/local partitioning, scheduling onto arbitrary interconnect topologies, and a preliminary interconnect synthesis algorithm. We express the relative distribution of local interconnects to global interconnects as a *fanout constraint*, which is simply the number of long (global) interconnects available for each local region. The strength of the interconnect synthesis algorithm described in this paper is its ability to handle interconnect fanout constraints.

3. Link Synthesis using Genetic Algorithm

Our interconnect synthesis method utilizes a genetic algorithm (GA) operating in conjuction with a list scheduling algorithm. The scheduling algorithm is a dynamic level scheduling (DLS) algorithm [1] modified for arbitrary interconnection networks. The algorithm takes into account constraints on the total number of links $l_{\max}$ and a maximum fanout for each processor or functional unit $f_{\max}$, as described earlier and motivated by area and cost constraints for the system.

3.1. Genetic Algorithm Overview

We give a brief overview of genetic algorithms here in order to explain our link synthesis algorithm. When a genetic algorithm is used to solve an optimization problem, it is necessary to be able to represent a single solution to the problem with a single data structure. This representation is often called a *chromosome* or an *individual*. The quality or *fitness* of a given solution is evaluated using an *objective function*. Genetic algorithms are capable of both broad search (exploration) and local search (exploitation) of a search space. They are often preferred than gradient search methods because they avoid local minima, and do not require a smooth search space.

The genetic algorithm creates an initial *population* of candidate solutions using an initialization operator. Often the initial population is distributed randomly over the search space. The genetic algorithm first selects individuals from the population and performs *crossover* and *mutation* operations on these individuals. Traditional crossover generates two *children* from two *parents* in a population. This is depicted in Figure 1 for a chromosome whose representative data structure is an array. A *crossover point* is chosen, shown by the dashed vertical line in Figure 1, and the child chromosome is formed by the elements from the first parent chromosome to the left of the crossover point and the elements from the second parent to the right of the

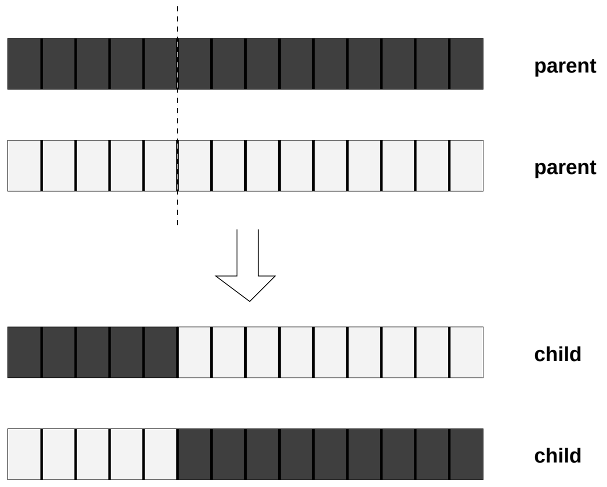


Figure 1: Crossover operator applied to array chromosome.

crossover point. The mutation operator specifies a procedure for changing (mutating) an individual. The specifics of the mutation depend on the data structure used to represent an individual. A typical mutation operator for an individual represented by a binary string flips the bits in the string with a given probability (the *mutation probability*). One *generation* of a genetic algorithm consists of performing crossover and mutation on individuals in the population. There are many possibilities for evolving the population. A *simple* GA uses non-overlapping populations. Each generation creates an entirely new population of individuals. A *steady state* GA uses overlapping populations, in which a fraction of the population is replaced in each generation. In an *incremental* GA each generation consists of only one or two children.

3.2. Problem representation

In our algorithm, the individuals are bit vectors corresponding to a given interconnect topology. The fitness function for a chromosome in our interconnect synthesis algorithm is described by

$$\text{fitness} = M(1 + P_f + P_l) \tag{1}$$

where M is the makespan (latency) calculated by the modified DLS algorithm for the interconnect topology of the chromosome, P_f (equation 6) is a penalty based on violating the fanout constraint $f_{\max}$, and P_l (equation 7) is a penalty based on violating the maximum link constraint $l_{\max}$.

We assume that the links are directional—that is, each link has one dedicated transmitter side and one dedicated receiver side. Optical links, for example, consist of a laser on one side of the link and a detector on the other side.

This assumption does not limit our model, however, because any bidirectional link can be represented as two directional links. We define a *link vector* as a bit vector with one entry for each possible interconnection between two processors. For a system with N processors, there are $N(N-1)$ entries in the link vector. The link vector for a four processor system would be denoted as

$$\vec{l} = (l_{01} l_{02} l_{03} l_{10} l_{12} l_{13} l_{20} l_{21} l_{23} l_{30} l_{31} l_{32}) \tag{2}$$

where l_{ij} equals one if there is a connection from processor i to processor j and zero otherwise. We define $l_{ij} \equiv 0$ if $i = j$. We also write $\vec{l}$ as

$$\vec{l} = (\vec{l_0} \vec{l_1} \ldots \vec{l}_{N-1}) \tag{3}$$

where $\vec{l_k}$ describes the (outgoing) connections for processor k. We will refer to the $\vec{l_k}$ as *processor link vectors*. We define the fanout of processor i by

$$f_i = \sum_{j=0}^{N-1} l_{ij} \doteq \|\vec{l_i}\| \tag{4}$$

Then the number of links is given as

$$n_l = \sum_{i=0}^{N-1} f_i \tag{5}$$

while the fanout penalty is given by

$$P_f = \sum_{i=0}^{N-1} P_i \tag{6}$$

where $P_i = \max(0, (f_i - f_{\max}))$. The link penalty is given by

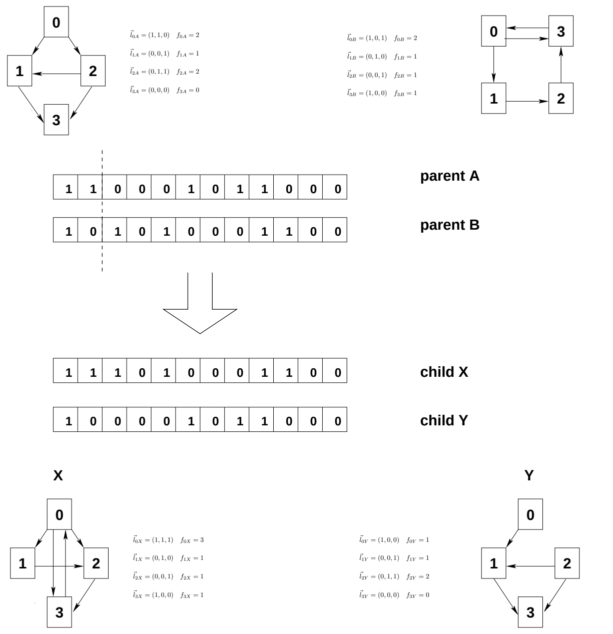
$$P_l = \max(0, (n_l - l_{\max})). \tag{7}$$

3.3. Fanout Constraints

In a real system, cost and area constraints will place a limit on the processor fanout. Therefore, as mentioned above, it is important to have a link synthesis algorithm that can conform to fanout constraints. Our GA is able to incorporate these constraints in a straightforward manner by implementing the initialization, crossover, and mutation operators as described below.

3.4. Crossover and Mutation Operators

We first note that if an individual topology is represented as a binary string as in equation 2, then the typical crossover operations like array one-point crossover (Figure 1) or two-point crossover will not preserve the fanout constraint. This is illustrated in Figure 2 where both parents obey a fanout constraint $f_{\max} = 2$, but processor 0 of child X has fanout $f_{0X} = 3$. This is because the crossover point can be chosen at any point. If we instead choose to represent the topology by the vector representation of Equation 3, fanout constraints are preserved in the crossover operation, since the processor link vectors $\vec{l_i}$ are never altered. The crossover operation only rearranges the relative position of these processor link vectors. This is illustrated in Figure 3.



$\vec{l}_{0A} = (1,1,0) \quad f_{0A} = 2$
$\vec{l}_{1A} = (0,0,1) \quad f_{1A} = 1$
$\vec{l}_{2A} = (0,1,1) \quad f_{2A} = 2$
$\vec{l}_{3A} = (0,0,0) \quad f_{3A} = 0$
$\vec{l}_{0B} = (1,0,1) \quad f_{0B} = 2$
$\vec{l}_{1B} = (0,1,0) \quad f_{1B} = 1$
$\vec{l}_{2B} = (0,0,1) \quad f_{2B} = 1$
$\vec{l}_{3B} = (1,0,0) \quad f_{3B} = 1$
1 1 0 0 0 1 0 1 1 0 0 0
parent A
1 0 1 0 1 0 0 0 1 1 0 0
parent B
1 1 1 0 1 0 0 0 1 1 0 0
child X
1 0 0 0 0 1 0 1 1 0 0 0
child Y
X
Y
$\vec{l}_{0X} = (1,1,1) \quad f_{0X} = 3$
$\vec{l}_{1X} = (0,1,0) \quad f_{1X} = 1$
$\vec{l}_{2X} = (0,0,1) \quad f_{2X} = 1$
$\vec{l}_{3X} = (1,0,0) \quad f_{3X} = 1$
$\vec{l}_{0Y} = (1,0,0) \quad f_{0Y} = 1$
$\vec{l}_{1Y} = (0,0,1) \quad f_{1Y} = 1$
$\vec{l}_{2Y} = (0,1,1) \quad f_{2Y} = 2$
$\vec{l}_{3Y} = (0,0,0) \quad f_{3Y} = 0$

Figure 2: Crossover operation for link synthesis using the binary string representation Equation 2. Link fanout constraint is not preserved for child X, where the fanout of processor 0 is $f_{0X} = 3$.

We also must ensure that the initial population obeys the link constraint. The initialization operator generates random processor link vectors which each satisfy the fanout constraint Equation 4. $N - 1$ of these vectors are then concatenated to form the link vector.

The mutation operator simply chooses a random bit in the link vector, and sets its value to zero. This removes a link if one existed at this point. Since the mutation operator only removes links, the fanout constraint is preserved.

4. Using Graph Isomorphism

If we consider systems in which all the processors are identical (homogeneous processor set), then we can pare the design space significantly if we only consider isomorphically unique topology graphs. Two graphs $G = (V, E)$ and $G'(V', E')$ are isomorphic if we can relabel the vertices of G to be vertices of G', maintaining the correspond-

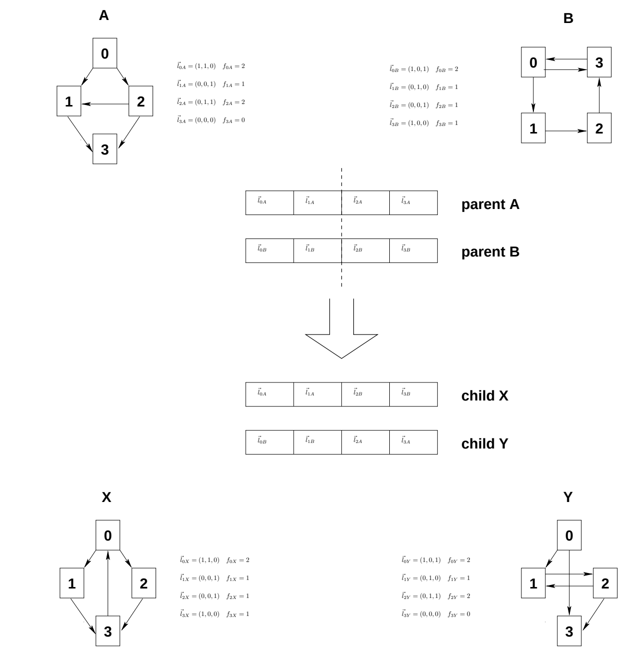


Figure 3: Crossover operation for link synthesis using the vector representation Equation 3. The fanout constraint $f_{\max} = 2$ is preserved in the children.

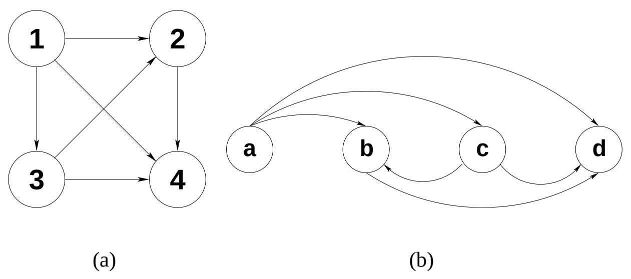


Figure 4: Example of two isomorphic graphs.

ing edges in G and G'. For example, the graphs in Figures 4(a) and 4(b) are isomorphic with the vertices relabelled as follows: $1 \to a$, $2 \to b$, $3 \to c$, and $4 \to d$.

Consider a topology graph G with e edges and n nodes where each node corresponds to a processor and each edge corresponds to a link between two processors. The maximum number of edges in G is $e_{\max} = n(n-1)$ corresponding to a fully connected graph (full crossbar interconnect). If all links are bidirectional, the topology graph is undirected and $e_{\max} = n(n-1)/2$. We can represent the graphs with either an adjacency list or adjacency matrix and label each different representation. Then for a graph with e edges the

$N = 4$ processors, all links bidirectional

$E_{\max} = 6$

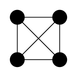
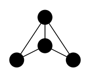
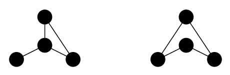
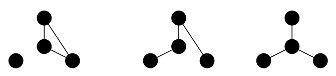
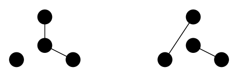
E	n_g	n_{unique}	
6	1	1	
5	6	1	
4	15	2	
3	20	3	
2	15	2	

Figure 5: Isomorphically unique graphs containing E edges for $n = 4$ processors. Here we only consider undirected graphs representing bidirectional links in order to make the figure clearer.

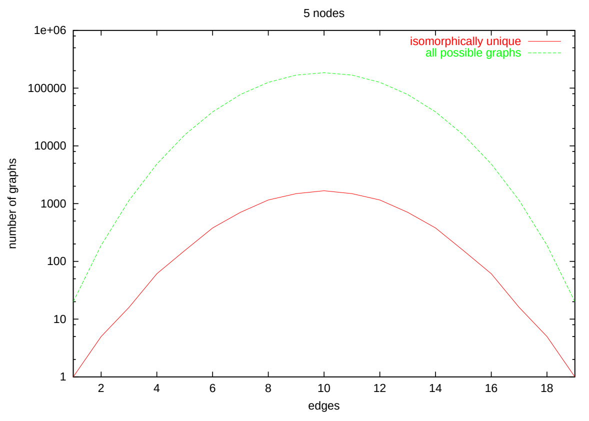
number of different labellings is given by

$$n_g = \binom{e_{\max}}{e} = \frac{e_{\max}!}{e!(e_{\max} - e)!} = \frac{n(n-1)!}{e!(n(n-1) - e)!} \tag{8}$$

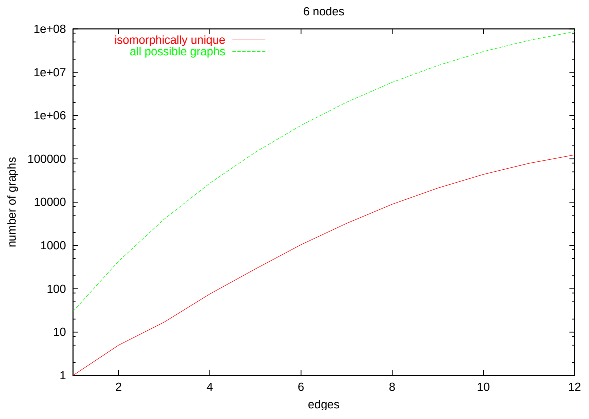
which increases exponentially with n. The maximum value of n_g occurs at $e = e_{\max}/2$. However, the number of isomorphically unique graphs n_{unique} is much less than n_g. For very small n, we can enumerate the different possibilities to show this. Figure 5 depicts the different isomorphic graphs for $n = 4$ processors and $e = 3$ bidirectional links. There are 20 different graph labellings, but we observe that most are isomorphic—only 3 are isomorphically unique.

For larger n, n_g increases rapidly according to Equation 8. We enumerated the possibilities and tested for isomorphism for $n = 5$ and $n = 6$ using Brendan McKays's *nauty* program [5], which is currently the fastest published graph isomorphism testing program. The results are shown in Figure 6 For $n = 6$ and $e = 12$ we observe that there is a 3 order magnitude difference between the number of graph labellings n_g and those that are unique (n_{unique}). Also, this ratio increases with n_g.

We exploit this property to improve our genetic algorithm. As mentioned earlier, each generation in the GA consists of a predetermined number of individuals derived from the previous generation by crossover and mutation operators. The initial population is generated randomly. However, the results from Figure 6 imply that a large fraction of the in-



(a) $n = 5$



(b) $n = 6$

Figure 6: A comparison of the number of possible graph labellings n_g given by Equation 8 with the number of these graphs that are isomorphically unique.

dividuals in any randomly generated population will be isomorphically equivalent, and that we would be wasting computation time by operating on equivalent interconnection topologies. Instead, we employ an efficient online graph isomorphism test when generating the population, and only accept new individuals that are isomorphically unique. By reducing the solution space by orders of magnitude, the GA can search it more throroughly in a given amount of computation time, and produce better results.

The graph isomorphism test is advantageous for the link synthesis algorithm only if the isomorphism testing is efficient. The complexity of the graph isomorphism problem is still an open problem—there exists no known polynomial-time algorithm for graph isomorphism testing, although the problem has also not been shown to be NP-complete. It is thought that the problem falls in the area between P and NP-complete, if such an area exists [10]. However, McKay's nauty [5] program has been proven to be very efficient in practice. Although its worst case run time is exponential [6], an empirical test of a large number of randomly generated graphs produced run times of $1.2p^2$ ns on a 1 GHz Pentium III machine where p is the number of nodes in the graph [5] (p equals the number of processors in our case). By comparison, the DLS scheduling algorithm has complexity $O(v^3p)$ where v is the number of nodes in the *task graph* [8]. We modify the DLS scheduling algorithm by adding a flexibility calculation at each scheduling step. The complexity of the flexibility algorithm is $O(v(v+e)p\log p)$, so the overall complexity scheduling an arbitrary graph using the modified DLS scheduling algorithm is

$$O(v^4(v+e)p^2\log p). \tag{9}$$

The number of tasks in the application will be much greater than the number of processors in practice, so $v >> p$ and $e >> p$. For randomly generated graphs, the nauty program is therefore much faster than the modified DLS scheduling algorithm and we achieve significant speedup by detecting and exploiting graph isomorphism.

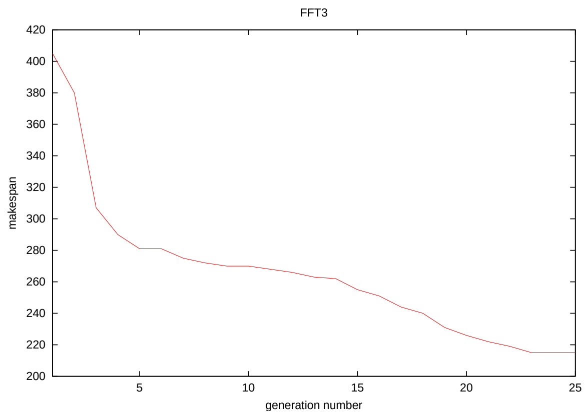


Figure 7: GA output versus generation for the FFT3 application.

4.1. Experiments

We evaluated our interconnect synthesis algorithm on several DSP benchmark application graphs. Figure 7 shows the convergence of the GA vs. generation number for an FFT application, with population size $N = 100$. This application graph, representing the computation of the fast Fourier transform, was taken from [4]. In this plot the y-axis refers to the schedule makespan of the best interconnection topology found. Figure 8 shows how the makespan

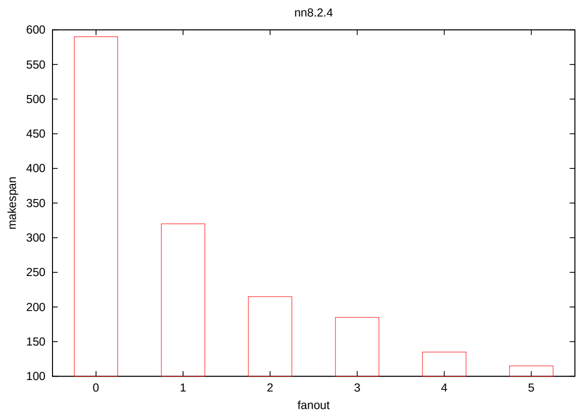


Figure 8: GA output for different processor fanout constraints.

improves with increasing processor fanout. As explained in [7], there is an increasing area overhead associated with a greater number of long interconnects. In our model, the number of long interconnects is equal to the number of processors times the average fanout per processor.

5. Conclusion

Interconnect synthesis is becoming an increasingly important problem for designers of systems-on-chip as the designs become larger. We presented a genetic algorithm for synthesizing efficient interconnection networks for SoC. The algorithm works in conjuction with a list scheduling algorithm to jointly optimize both the schedule and the interconnect topology. The algorithm is able to account for different distributions of local vs. global (long) interconnect routing tracks via a processor fanout constraint. It uses graph isomorphism to significantly pare the search space in order to search more efficiently.

Acknowledgment

This research was supported in part by the DARPA funded Optoelectronics Center for Innovative Photonic Chipscale Technologies (contract number MDA972-00-1-0023, through Brown University).

References

[1] N. Bambha and S. S. Bhattacharyya, *System synthesis for optically-connected, multiprocessors on-chip*, System on Chip for Real-time Systems (W. Badawy and G. A. Julien, eds.), Kluwer Academic Publishers, 2002, pp. 339–448.

[2] N. K. Bambha, S. S. Bhattacharyya, and G. Euliss, *Design considerations for optically connected systems on chip*, Proceedings of the International Workshop on System on Chip for Real Time Processing (Calgary, Canada), June 2003, pp. 299–303.

[3] C. Ebeling, D. Cronquist, and P. Franklin, *RaPiD: reconfigurable pipelined datapath*, Proceedings of the 6$^{\text{th}}$ International Workshop on Field-Programmable Logic and Applications, 1996, pp. 126–135.

[4] C. L. McCreary, A. A. Kahn, J. J. Thompson, and M. E. McArdle, *A comparison of heuristics for scheduling DAGs on multiprocessors*, Proceedings of International Paralel Processing Symposium, 1994.

[5] B. McKay, *Nauty user's guide*, Tech. Report Technical Report TR-CS-90-02, Australian National University, 1990.

[6] T. Miyazaki, *The complexity of McKay's canonical labeling algorithm*, Groups and Computation II **28** (1997), 239–256.

[7] A. Sharma, C. Ebeling, and S. Hauck, *Piperoute: a pipelining-aware router for FPGAs*, Proceedings of the 11$^{\text{th}}$ ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2003, pp. 68–77.

[8] G. C. Sih and E. A. Lee, *A compile-time scheduling heuristic for inerconnection-constrained heterogeneous processor architectures*, IEEE Transactions on Parallel and Distributed Systems **4** (1993), no. 2, 75–87.

[9] A. Singh, A. Mukherjee, and M. Marek-Sadowska, *Interconnect pipelining in a throughput-intensive FPGA architecture*, ACM/SIGDA 9$^{\text{th}}$ International Symposium on Field-Programmable Gate Arrays, 2001, pp. 153–160.

[10] S. Skiena, *Graph isomorphism, implementing discrete mathematics: Combinatorics and graph theory with mathematica*, Addison-Wesley, Reading MA, 1990.

[11] S. Sriram and S. S. Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*, Marcel Dekker Inc., 2000.