FNOCEE: A Framework for NoC Evaluation by FPGA-based Emulation

Daniel Pfefferkorn, Achim Schmider, Guillermo Payá-Vayá, Martin Neuenhahn, and Holger Blume
Institute of Microelectronic Systems, Leibniz Universität Hannover
Appelstr. 4, 30167 Hannover, Germany
Email: {pfefferk|schmider|guipava|neuenhahn|blume}@ims.uni-hannover.de

***Abstract*—This paper introduces FNOCEE, a framework for the evaluation of NoC-based many-cores systems by FPGA-based emulation. It uses a task graph-oriented approach to model applications, while a hardware-accelerated genetic algorithm is employed to find close-to-optimal solutions to the task mapping problem. The proposed genetic algorithm is analyzed in detail, e.g., in terms of mutation rate and number of elite individuals. In order to illustrate the framework's capabilities, several case studies have been performed, wherein scalability of relevant parallel applications is investigated with regard to the number and type of available processing cores and the generated traffic load as a result of inter-task communication.**

I. Introduction

The number of processing cores on a chip has been steadily increasing over the last years. This is caused by technology improvements, which have been prospected by Moore's Law one the one hand and by the ever increasing computational demands of applications (e.g. multimedia applications) on the other hand. These applications typically consist of several computational tasks, which can be executed in parallel on different cores.

With the increased number of processing cores on a chip, the communication among them becomes more and more critical. Classic communication structures like busses or point-to-point connections either cannot fulfill communication demands or are simply too costly to implement. One solution to this problem is the Network-on-Chip (NoC). Designing a NoC is not a simple task, as the design space for NoCs is huge, due to the large number of design parameters (e.g. topology, routing algorithms). Therefore, a design methodology and framework is needed to identify the optimal NoC parameter combination for a certain application or application class.

One essential part in designing a NoC, is the assessment of performance and power consumption of a given NoC defined by a set of parameters. A major factor which influences these performance metrics is the mapping of computational tasks onto specific computational cores. As for each mapping of an application onto a NoC the resulting performance has to be determined, the identification of a mapping which fulfills the communication requirements at minimal costs is complex. Thus, fast evaluation of a large number of mappings is crucial for finding good mappings. Classic simulation-based approaches cannot fulfill this demand or have to be performed with reduced temporal resolution.

In this work, a framework for the design of application-specific NoCs is presented. Performance evaluation is done using an FPGA-based emulator. To find a near-optimal mapping, a genetic algorithm is used, which is implemented on the FPGA as well. The viability of this framework is demonstrated in three case studies.

II. Related work

In the last years, many Network-on-Chip architectures have been proposed [1]. Most of these NoCs are derived from computer networks and adapted to the needs of SoCs. Examples of some of these first prototypes and even products featuring NoCs are the 80-Tile 1.28 TFLOPS Network-on-Chip, presented in [2], or the TILE-Gx or TILE-Mx Processors [3]. The performance and the hardware costs (measured in power consumption or silicon area) of these NoCs strongly depend on NoC-specific parameters, like topology [4], data word length, routing algorithm, and many more [5]. These NoC parameters span a huge design space for NoCs. Due to the fact that each application or application-class has different communication requirements, the appropriate NoC-parameter combination which fulfills these requirements with minimal hardware costs has to be found [6], [7].

The mapping of an application on the processor cores available in the NoC is still a challenge [8], which highly influences the search of the NoC with minimal hardware costs. It is a complex, NP-hard problem [9] and cannot be performed manually for relevant problem sizes. An evaluation and classification of mapping algorithms is given in [10]. In recent years, genetic algorithms have been used and evaluated, showing good performance results [11], [12], [13], [14], [15].

In order to evaluate the performance of a specific task mapping on a specific NoC architecture, different approaches from simulation [16] to FPGA emulation [17], [18], [19] can be used. However, the task mapping is still performed off-line involving transfers of each individual mapping and, therefore, extremely increases the exploration time required to fairly evaluate a given NoC architecture. In this paper, the search of an optimal static mapping based on a genetic algorithm is included in the NoC emulation system, highly decreasing the time required for NoC evaluation.

978-1-4673-7311-1/15/$31.00 ©2015 IEEE

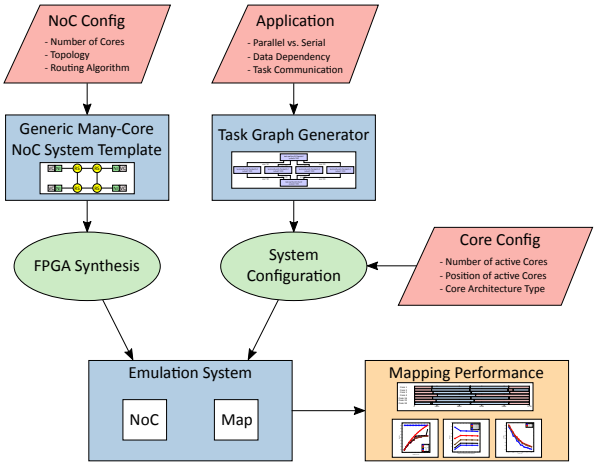


Fig. 1: Overview of FNOCEE's capabilities with regard to the problem space

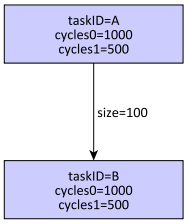


Fig. 2: Simple task graph with two tasks, both of which need 1000 cycles to execute on a core of architecture type 0 or 500 cycles on a core of architecture type 1. To execute task B, a message with a duration of 100 cycles has to be sent from the core executing task A to core executing task B.

III. Framework for NoC evaluation by emulation

In order to examine the effects of different topologies and NoC-sizes on application execution the "framework for NoC evaluation by emulation" (FNOCEE) has been developed.

A. General Problem Overview

When investigating the benefit of executing a program on a many-core architecture, it is important to consider multiple influencing parameters. Software applications feature a certain inherent degree of parallelism. This can be either data or task parallelism. Choosing tasks as the atomic unit of computation within an application leads directly to the problem of mapping them efficiently onto available execution units (processor cores).

This mapping problem is influenced by application aspects, for example number and duration of tasks and their interdependency, but also by hardware aspects, such as the number and type of available processing cores, as well as the properties of the available communication channels between these cores (Fig. 1). Since task mapping is of utmost importance to properly exploit many-core architectures, this problem has to be solved. With FNOCEE, we use a generic but hardware-supported approach to model aforementioned properties and to solve the problem of task mapping.

B. Tasks & Task Mapping

A necessary first step to evaluate application performance on NoC-based many-core systems is the ability to model them. Section III-B1 will explain the chosen approach and its advantages and limitations. In the next step, the application model needs to be combined with the system-specific information to produce a suitable, if not necessarily optimized, mapping. This process is described in section III-B2.

1) Task Graphs: An application can usually be described as the conjunction of a set of tasks and their dependencies. These dependencies are the result of data dependencies present within the application. A common way to describe the aforementioned tasks and their order of execution is a directed acyclic graph. Tasks are identified by a task ID and form the graph's nodes. The edges between the nodes represent dependencies. The direction of the dependency is identical to the direction of the edge indicated by its arrow.

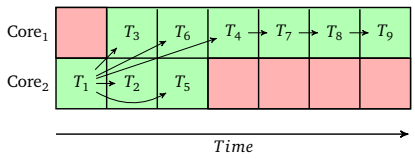
Nodes and edges of the task graph can be annotated with attributes to provide additional information. Generally, nodes are annotated with an execution time measured in cycles. In order to precisely specify the execution times for specific architectures types, within FNOCEE each task is annotated with the amount of cycles necessary to complete the task on a specific core type. Edges are labeled with the amount of data that needs to be transferred in order to allow processing of subsequent tasks. An example for such a graph is given in Fig. 2.

Task graphs are stored in the XML-based GraphML format [20], which is human-readable and allows easy manipulation as well as automated generation. With the help of the graphical editor yEd [21] it is possible to visualize, edit and create task graphs in the GraphML format.

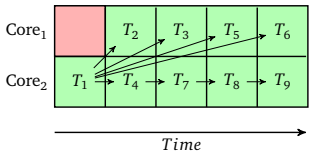
2) Task Mapping: Task mapping can be described as the non-surjective mapping $T \rightarrow C$ of a set of tasks T onto a set of processing cores C. Redundant mappings, where a task is assigned to more than one core, are not within the scope of this publication.

An application can only properly benefit from a multi- or many-core architecture, if its tasks are mapped efficiently. Fig. 3 illustrates two possibilities to map nine tasks onto two processor cores. Due to their dependency on task 1, tasks 2 through 6 can only be executed after task 1 has completed. However, the tasks 7 through 9 have a serial dependency on task 4. In order to exploit the parallelism of the two available processing cores, it is essential to execute the independent tasks in parallel to the serially dependent ones. This results in a reduction of execution time by 28.6 % and increases the utilization to 93.3 %.

In order to solve the mapping problem, numerous algorithms have been proposed. [9] presents a survey on various instances of the mapping problem. Many of these prove to be NP-hard, making it impossible to find an optimal solution with



(a) Inefficient task mapping



(b) Efficient task mapping

Fig. 3: Comparison of two possible ways to map 9 tasks onto 2 processor cores

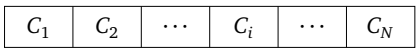


Fig. 4: Coding of a task mapping with N tasks (genome). Each C_i represents a core ID.

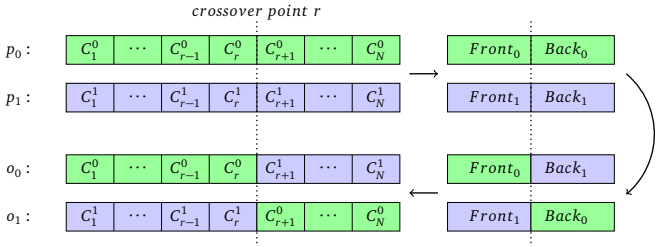


Fig. 5: Recombination of two parents p to two offsprings o by exchanging the partial genomes at the crossover point r

a reasonable amount of resources (e.g. time, computational and/or storage capability). Depending on the problem, approximation algorithms can help to achieve a "close-to-optimal" solution, but usually with far less effort. Various algorithms are presented and discussed in [10]. One of the most capable of the discussed ones is the genetic algorithm.

Genetic algorithms belong to the class of metaheuristics [22] and multiple variants are applicable to the problem of task mapping [11]. Initially, a start population is generated randomly or with the help of other algorithms. A population consists of individuals, each representing a candidate solution within the solution space and described by their genome. A fitness function determines the quality of these solutions. When generating the next generation, the fittest parent individuals are chosen for reproduction and new individuals are bred through crossover (i.e. recombination of the parents' genomes) and subsequent mutation (i.e. random gene variation) operations. The least fit individuals are replaced by the new individuals. This process is repeated, until a termination criterion (e.g. satisfying solution, maximum number of generations, computation time, etc.) has been met.

The fitness function has a significant influence on the effectiveness of the genetic algorithm. In the special case of application-oriented NoC optimization, the execution time shall be minimized. Therefore, the fitness function has to transform the application's overall execution time in such a way, that a shorter execution time results in a higher fitness value. The authors in [12] use the reciprocal of the execution time as fitness value. In order to avoid computationally costly divisions, the method proposed in [13] and [14] has been implemented. For every generation, the maximum execution time c_{max} is determined. The fitness function $f : N \rightarrow N$ is defined as: $f(c_i) = c_{max} - c_i + 1$. As a result, all implementation-specific details are abstracted. In order to achieve acceptable solutions with analytical optimization algorithms, the systems behavior would have to be modeled precisely first, which requires significant additional effort.

A population consists of a set of possible solutions, which is in our case a set of viable task mappings. Their genomes are coded in such a fashion that the i-th gene contains the ID of the core onto which the task with the ID i is mapped (Fig. 4). Choosing this kind of coding yields potential problems, as two general principles have to be followed [23]:

1) Genes which are related, are close to one another in the genome
2) Genes only have a small influence on each other

The first principle can be addressed by placing tasks which are interdependent next to each other in the coding. As mapping the task graph is a discrete problem, minor changes in the mapping might result in large differences in the overall execution time. It is therefore a task-graph-dependent question, whether the second principle can be abided by.

During *recombination*, parents shall be selected with a probability proportional to their fitness value. For a population of N candidates, a uniformly distributed random value r is chosen from the right-open interval $[0, \sum_{t=1}^{N} f(c_t))$. The mapping with the smallest index p fulfilling the inequality $r < \sum_{t=1}^{p} f(c_t)$ is selected. Two parents p_0 and p_1 are selected to generate two offsprings o_0 and o_1. The genomes of these parents are recombined according to the one-point crossover method. For this, again a uniformly distributed random value r is chosen from the right-open interval $[1, L)$, where L represents the length of the parents' genomes. Offspring o_j inherits the genes i for $1 \leq i \leq r$ of parent p_j and the genes i' for $r < i' \leq L$ of parent p_{1-j} (see Fig. 5).

For the mutation of a genome with length L, random values $r_i, 1 \leq i \leq L$ from the interval $[0, 1)$ are generated. Gene i of the genome is mutated if r_i is less than the set mutation probability P_M. In this case, task T_i is assigned a new core for execution. Candidates are all cores of an architecture type for which T_i defines an execution time. From this set of candidates, a random core is chosen with uniformly distributed probability.

The genetic algorithm cannot guarantee that the quality of a generation's best solution does not decrease. Therefore, it is possible to automatically inherit the N_E best mappings of the former generation. This ensures the guaranteed "survival" of

the best solutions. Otherwise, these *elite* mappings could be removed due to an unlucky selection during recombination. Additionally, findings in [24] indicate that inheritance of the best solution(s) improves the algorithm's convergence.

In order to accelerate the task mapping optimization, the genetic algorithm has been implemented as a synthesizable hardware description ($mapgen$), which is described in section III-C2. In combination with cycle-accurate execution to evaluate a mapping's fitness, this results in a tremendous overall speed-up.

C. Hard- and Software Components of FNOCEE

In order to enable evaluation by emulation, soft- and hardware components are necessary. Therefore, FNOCEE consists of four elements: a task graph defining the application, an emulation hardware component realizing the NoC and the surrounding components, a mapping unit assigning each task a physical execution unit, and the host software which transmits task graph and cores/core types to use, controls the evaluation process, and collects the results (Fig. 6). Communication between host and emulation system is established via the unified emulation framework (UEMU), which is described in [25].

1) Virtual Cores: Each task needs to be executed on a processing unit. Typically, this will be an individual core in a multi- or many-core system. In order to maintain a high abstraction level, a concept that can be best described as "virtual cores" was implemented.

Before starting an experiment, each virtual core is configured to be of a specific architecture type. In order to execute a task, it will check whether the task's dependencies have been resolved, i.e. all other tasks on which the task to be executed depends have been executed and their results have been received. If this is the case, an internal counter will be loaded with the architecture-specific cycle count. It will then simply decrement this cycle count until it is zero, meaning that the task has been processed completely. Afterwards, all virtual cores with depending tasks will be notified by sending a message through the NoC. The size of this message equals the value annotated along the edge connecting the two tasks within the task graph.

Additionally, each virtual core features performance counters to count the number of cycles the core spends in each of the following states:

Blocked	at least one necessary message has not been received yet
Executable	all necessary data are locally available
Processing	the task is being processed by the core
Post-processing	task has been processed and messages to the successors are being created
Completed	all outgoing messages have been sent and the core is ready to process the next task

In order to reduce the overhead incurred by task handling (dependency checking and updating, post-processing/message generation) and to avoid blockage due to interaction with the NoC's network interface controller (NIC), the virtual core has been split into five independent submodules as illustrated in Fig. 7.

The module $ctrl$ connects the virtual core to a serial control bus, which allows configuration of the core, writing of task information, requesting the current task's or the core's status information. Operation of this module is independent of the NoC clock source.

The submodule $worker$ is responsible for processing a task. After startup, it will receive a list of runnable tasks from $ctrl$. It will then start to process the first task on this list by copying the architecture-specific cycle count to an internal register. The register value will then be decremented in synchronization with the NoC's clock. Once the register value reaches zero, the task has been processed and its successors need to be notified. These successors are stored in a list attached to the task. For each successor task in this, task ID the core ID of the core onto which the successor task has been mapped and the message size (edge-annotated in the task graph) can be extracted. A message is generated accordingly and transferred to the nic submodule for transmission via the NoC. To decouple task processing from message transmission, all messages will be stored temporarily in the nic submodule.

All messages received by the nic from the NoC must be handled, as they contain task completion information, possibly allowing a blocked task to be processed. Handling of received or locally generated messages is the responsibility of the submodule $updater$. It will extract the task ID of the completed task, perform a lookup for tasks with a matching predecessor in the list of currently blocked tasks and mark the dependency for this predecessor as resolved. In case there are no more unresolved predecessor dependencies, the blocked task will be labeled as runnable and handed over to the submodule $worker$ for processing. Once the list of blocked tasks is empty, the $ctrl$ will be notified of the overall completion of all tasks.

As stated earlier, both nic and $worker$ might be sources for task completion messages. Although this does not change the operation of $updater$, it is worth mentioning that hence two messages might arrive at the same time. Therefore the $updater$ submodule is able to store one message per source. Subsequent messages will only be acknowledged after the previous message from the same source has been processed. Messages originating from a different virtual core are handled with higher priority.

In order to send messages across the NoC, each virtual core has a nic submodule. Its transmission and reception data paths are independent, allowing full-duplex transfers, thus further reducing the task handling delay. After task completion, the $worker$ submodule sends the nic an information quadruple containing the task ID of the completed task (T_S), the task ID of the depending task (T_D), the recipient core ID (C_D) and the message size (M). Once the egress path is idle, this quadruple will be used to establish a connection with the recipient's nic through the NoC. Afterwards a message header containing T_S, T_D and C_D is sent followed by $M \times wordwidth$ byte of pseudorandom data.

The receiving nic will accept the incoming connection

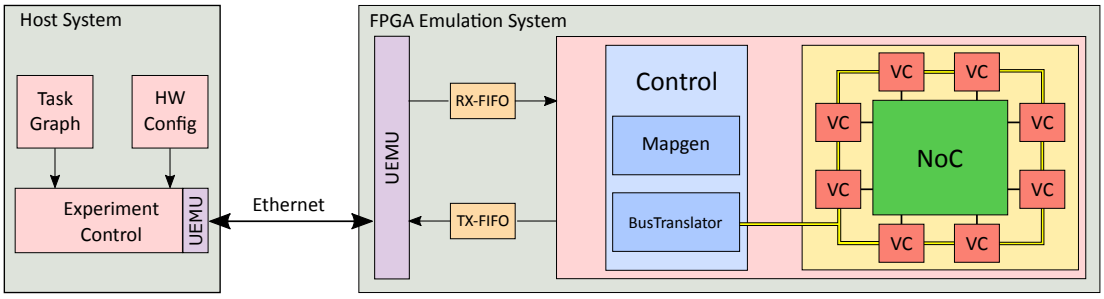


Fig. 6: General overview of FNOCEE. Task mapping optimization with the help of the genetic algorithm is performed within the module $mapgen$. All virtual cores (VC) are connected via a serial control bus.

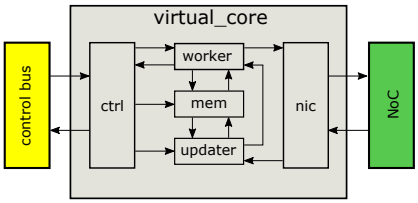


Fig. 7: Structure of a virtual core

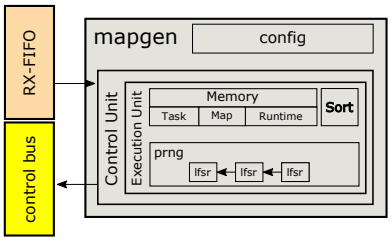


Fig. 8: Hierarchy of the $mapgen$ module

request if it is idle, and prepares the reception of the message header. Once message header and subsequent data have been received, T_S, T_D and C_D are handed over to the $updater$ submodule and the connection is closed.

2) HW-Accelerated Genetic Algorithm: The module responsible for the generation of task mappings contains multiple submodules, such as a control and an execution unit $mapgen\_cu$/$mapgen\_eu$, a pseudorandom number generator $mapgen\_prng$ and a sorting unit $mapgen\_sort$ (see Fig. 8).

$mapgen\_cu$ receives all incoming data and commands addressed to $mapgen$. It allows configuration of the genetic algorithm's parameters (mutation probability, number of elites and seed value of the PRNG) and is able to receive a list of tasks to be mapped as well as a list of available virtual cores including their architecture types. After being provided with these lists and the necessary configuration data, the task mapping optimization can be started. The module $mapgen\_cu$ will then control the execution of the following steps:

1) Evaluate all task mappings of current generation
2) Copy elite individuals
3) Calculate sum of all fitness values
4) Perform selection of parents
5) Perform crossover recombination
6) Perform mutation
7) If generation limit not reached, restart at 1, else experiment is complete

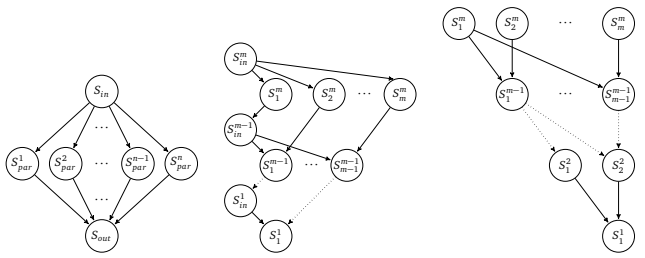
Note that during evaluation, all task mappings of a generation are evaluated by processing the entire task graph once. This will result in message transfers across the NoC which may lead to congestive behavior and therefore delayed execution of blocked tasks.

The $mapgen\_cu$ will instruct the module $mapgen\_eu$ to perform recombination, mutation, elite copying and sorting operations. All possible mappings and their individual execution times are stored inside a local memory. Evaluated mappings have to be sorted according to their execution time prior to choosing the N best. Therefore, the sorting unit ($mapgen\_sort$), based on a parallel shift sort [26], arranges the stored candidate entries in ascending execution time order. During recombination and mutation operations, multiple random values are necessary to choose the crossover point or decide whether a gene is mutated or not. These random values are generated by the submodule $mapgen\_prng$. Since a seed value may be set, it is possible to exactly re-create past experiments or perform an experiment with different random values. The PRNG is implemented via linear feedback shift registers (LFSRs), which are very suitable for FPGA-emulation [27].

As stated earlier, a fundamental advantage of our approach is the precise capture of the dynamic communication behavior as actual messages are transferred across the NoC. Since task mapping optimization and task graph execution are performed via emulation on an FPGA, execution times are extremely short compared to implementations where the genetic algorithm is implemented in software or the behavior is analyzed via simulation. With the current setup, it is possible to evaluate significantly more than one thousand mappings per second cycle-true. Since the evaluation involves the actual execution of the task graph, this number is strongly dependent on the number of execution cycles defined in the task graph itself.

IV. Case Study

To investigate the viability and potential of NoC evaluation with FNOCEE, we created multiple synthetic task graphs and



(a) Parsync (b) LU (c) GJ

Fig. 9: Type of task graphs used for individual experiments. LU denotes graphs inspired by LU decomposition, GJ graphs are inspired by the Gauß-Jordan algorithm.

NoC varieties. The conducted experiments are individually discussed in sections IV-A, IV-B and IV-C.

Fig. 10 shows two basic examples of NoC topologies. All virtual cores (VC) are connected to the NoC via network interfaces (NI), which provide a transmit as well as a receive port independent of each other. Each NI is connected to a routing switch (RS) via two independent unidirectional links. The routing switch has a topology-dependent number of ports, each consisting of the aforementioned two unidirectional links. The links are 16 bit wide and can introduce errors based on a configurable bit error probability. Circuit switching in combination with a "send-and-wait" protocol augmented with a sliding window is used to transfer data among the NIs. The routing switches choose the egress ports for data according to the deterministic XY-routing algorithm, which is known to be deadlock free in meshes [28].

The NoC components can be connected to form networks of different topologies. For reasons of simplicity, we chose to investigate different variations of two basic mesh topologies:

- a 2D mesh, where a core is connected to every RS (intern)
- a 2D mesh, where cores are connected only to peripheral RS (extern)

Tables I and II give an overview on the basic NoC configurations used and the overall resource usage when synthesizing for a Xilinx™ XC6VLX240T FPGA. The design is logic-dominated and uses a maximum of $69.47\,\%$ of the available block RAM resources and $0.39\,\%$ of the DSP48E Slices. Additionally, the overhead for the necessary communication components and mapping generation (1,509 slices ($4.03\,\%$) and 43 RAMB36E1 ($14.87\,\%$) for τ_7) is comparably small, which allows the evaluation of large NoCs structures on commonly available emulation platforms.

A. Performance Scaling with Number of Cores

The main motivation for multi-/many-core systems is to accelerate program execution through means of parallelization. However, there are limits to the achievable speedup. In 1967, Gene Amdahl stated what has become known as Amdahl's Law [29]. Parallel execution of application with a known serial

TABLE I: Overview of the synthesized NoC configurations. Dimensions for a 2D mesh result from the number of rows and columns of routing switches. Network interfaces are included in the row/column count for "extern" variants.

Config	Topology	VCs	Width	Height	RSs	Links
τ_0	2D-Mesh (int.)	9	3	3	9	21
τ_1	2D-Mesh (ext.)	16	3	9	7	22
τ_2	2D-Mesh (int.)	16	4	4	16	40
τ_3	2D-Mesh (ext.)	16	6	6	16	40
τ_4	2D-Mesh (int.)	25	5	5	25	65
τ_5	2D-Mesh (ext.)	36	3	19	17	52
τ_6	2D-Mesh (int.)	36	6	6	36	96
τ_7	2D-Mesh (ext.)	36	11	11	81	180

TABLE II: Total resource usage when synthesizing different NoC configurations for a Xilinx™ XC6VLX240T FPGA

Config	Slices	RAMB36E1	DSP48E1
τ_0	$19,220$ ($51.10\,\%$)	127 ($40.63\,\%$)	3 ($0.39\,\%$)
τ_1	$23,556$ ($62.52\,\%$)	169 ($40.63\,\%$)	3 ($0.39\,\%$)
τ_2	$26,188$ ($69.50\,\%$)	169 ($40.63\,\%$)	3 ($0.39\,\%$)
τ_3	$26,388$ ($70.03\,\%$)	169 ($40.63\,\%$)	3 ($0.39\,\%$)
τ_4	$32,644$ ($86.63\,\%$)	223 ($53.61\,\%$)	3 ($0.39\,\%$)
τ_5	$35,812$ ($95.04\,\%$)	289 ($69.47\,\%$)	3 ($0.39\,\%$)
τ_6	$36,817$ ($97.71\,\%$)	289 ($69.47\,\%$)	3 ($0.39\,\%$)
τ_7	$37,430$ ($99.33\,\%$)	289 ($69.47\,\%$)	3 ($0.39\,\%$)

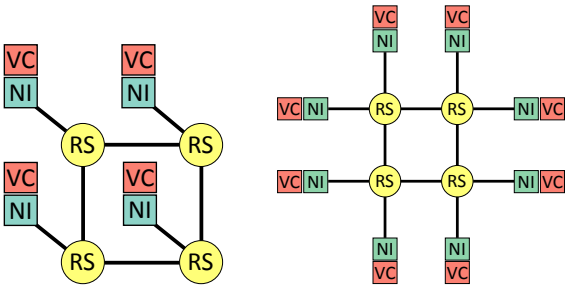
part f by N processor cores will result in a theoretical speedup of S_A according to:

$$S_A = \frac{1}{f + \frac{1-f}{N}} \tag{1}$$

Three aspects complicate the direct application of this law. For most applications, it is difficult if not impossible to determine the serial part. Furthermore, Amdahl's law is continuous, whereas most applications are divided into small tasks and hence discrete in nature. The achievable speedup is therefore limited and reached when the number of processor cores is identical to the number of tasks. A third limitation is neglect of communication overhead in case of interdependent tasks which are mapped onto different processor cores. Thus it may be beneficial to execute interdependent tasks on the same core although others are available.

In order to investigate the influence of discretization in tasks and communication cost, a parameterized task graph called *Parsync* was created. It consists of an entry node S_{in} (fork) and an exit node S_{out} (join) forming the serial part and n parallel and independent nodes S^i_{par}(execute) forming the parallel part. Cycle counts can be adapted to result in a desired serial to overall ratio, which will be appended as numerical suffix, e.g. *Parsync5* for a $5\,\%$ amount of serial cycles. These task graphs were then mapped to different NoC configurations with the help of our genetic algorithm hardware module.

As a next step, the achieved speedup was examined as a function of the number of virtual cores. *Parsync5* was mapped to multiple NoCs featuring 16 virtual cores, but with different topologies. The number of parallel tasks in *Parsync5* was set to 16 as well. For each combination the genetic algorithm was run 20 times with different seed values for the PRNG and the best task mapping was chosen. The results are illustrated in Fig. 11.



(a) Variant: 2D mesh intern

(b) Variant: 2D mesh extern

Fig. 10: Topology variants of a 2D mesh NoC. Each virtual core (VC) is connected to the NoC via a network interface (NI), which itself is connected to a routing switch (RS). Routing switches and network interfaces are connected via links. The NoC in (a) has a size of 2×2, the NoC in (b) a size of 4×4, since the coordinate grid is based on the NI positions.

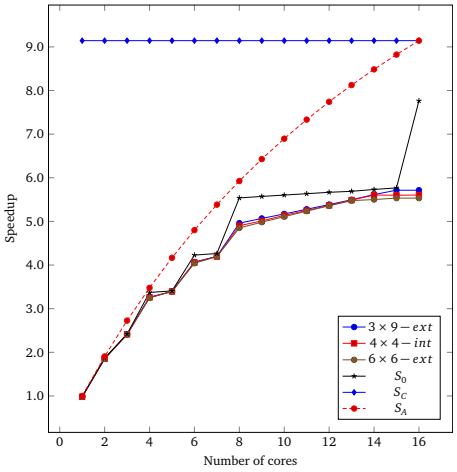


Fig. 11: Speed-up for execution of the task graph *Parsync5*. The experiment has been conducted for NoC configurations τ_1, τ_2 and τ_3.

In addition to the achieved speedups, the achievable speedup according to Amdahl S_A, the achievable speedup based on the length of critical path S_C (the longest serial path through in the task graph) and the ideal speedup with a message size of zero S_0 are included. The task graph contains 16 parallel tasks, therefore the S_A and S_C converge for a number of 16 virtual cores. Being almost identical for two cores, S_A and all other speedup curves start to diverge increasingly with higher numbers of available cores. This illustrates the quantization effect or - in other words - task granularity. S_0 takes this effect into account and illustrates losses due to communication. Its steplike behavior results from domination of the overall execution time by the longest chain of tasks mapped onto a single virtual core. A mapping of the 16 parallel tasks onto 10 cores will result in at least one core with two mapped tasks. Once 16 cores are available, the number of mapped tasks per cores instantly drops to one, resulting in a jump in speedup.

Interestingly, S_0 shows a steady increase for the interval $[8, 15]$. After the entry task S_{in} has been processed, 16 messages are generated to inform the successor tasks. These are sent serially resulting in a delayed start of the parallel task's execution, increasing with the task's ID. Although the message size is zero, a connection is established along all involved routing switches, which will result in a few cycles of transmission delay. Cores that have received this message earlier will complete task execution earlier, making them very suitable to process one of the remaining tasks, hence reducing the overall execution time. The result is a linear correlation between the number of available cores and the resulting speedup. Using the detailed analysis mode of FNOCEE, we can visualize this staggered execution in Fig. 12.

A third effect illustrated by Fig. 11 is the speedup stagnation for 14 and more available processor cores as the message-induced overhead outweighs the additional parallelization benefit.

B. Performance Scaling with Traffic Load

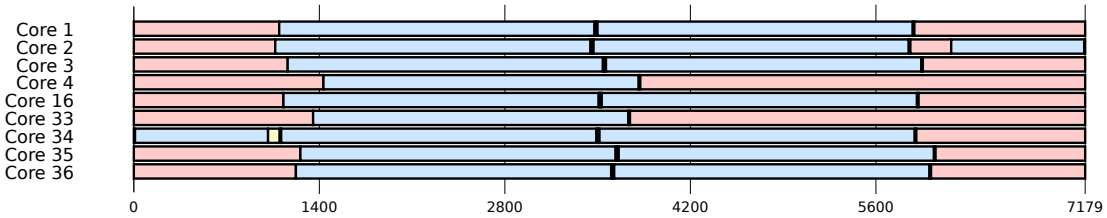
An outcome of the previous experiment was the lack of separation between NoC configurations. As the transmitted messages for a single task graph were very limited in size and number, each task graph was instantiated multiple times in parallel (see III for more information). This could easily be achieved through graph replication using the graphical editor.

TABLE III: Task graphs used to investigate NoC-utilization effects. Every graph consists of four identical and independent partitions. c_T represents the overall amount of execution cycles, c_{CP} the length of the longest serial path (critical path) in cycles. For LU and GJ graphs the numerical suffix indicates the number of stages (their structure is illustrated in Fig.9)

Name	Parameter	Tasks	Edges	c_T	c_{CP}
Parsync5x4	$f = 0.05$	72	128	39,952	1,093
Parsync30x4	$f = 0.30$	72	128	39,968	3,437
LU7x4	7 Levels	140	220	39,944	3,328
GJ8x4	8 Levels	144	224	40,320	2,240

With these combined task graphs, the resulting solution space is much larger and the genetic algorithm was not always able to find solutions of constant quality. This becomes evident when plotting the resulting speedup as a function of the message size. Graphs in Fig. 13 show a constant decline with increasing message size with several outliers caused by GA's inability to find close to optimal mappings.

Again, speedup for different NoC topologies did not differ significantly for the task graphs *Parsync5x4* and *Parsync30x4*. A very distinct separation was visible when using the task graph *LU7x4* (Fig.14). With increased message lengths, the topology *4x4-int* shows the largest speedup values, the topology *3x9-ext* the smallest. As a result of the increased traffic load this NoC with its linear layout is now suffering from transmission delays due to blocked routing switches increasing the overall execution time. The difference between topologies τ_2 and τ_3 originates in differing average path lengths of 3.91



(a) Task mapping for 9 virtual cores

Fig. 12: Processing of the task graph *Parsync5M0* plotted over time on 9 cores. The number of cycles is given in the bottom axis. Core idle cycles are depicted in red. Tasks are processed and messages are generated in the blue and yellow segments, respectively.

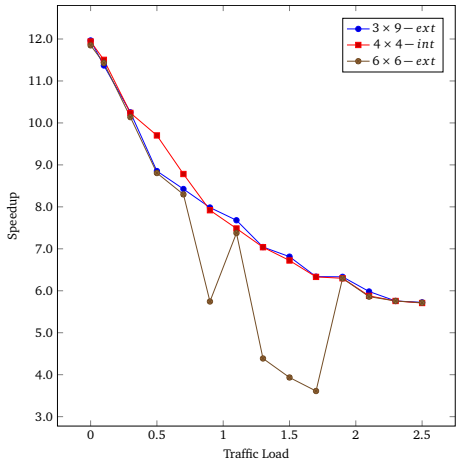


Fig. 13: Speedup for execution of the task graph *Parsync5x4*. The experiment has been conducted for NoC configurations τ_1, τ_2 and τ_3.

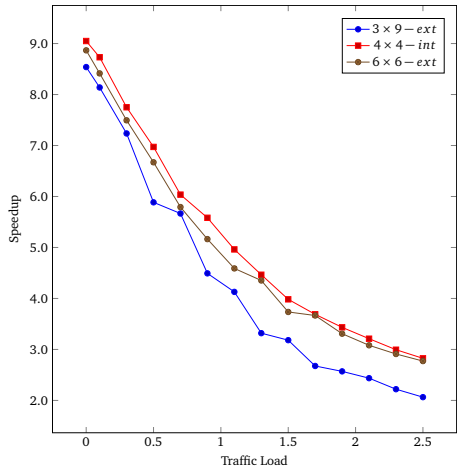


Fig. 14: Speedup for execution of the task graph *LUx4*. The experiment has been conducted for NoC configurations τ_1, τ_2 and τ_3.

and 4.34 routing switches, but more research is necessary regarding the exact behavior of the NoC itself.

C. Performance Scaling for specialized Core Architectures (heterogeneous SoC)

With the help of FNOCEE, it is also possible to investigate the overall execution behavior when different core architectures are available. A core with a specialized architecture can be employed to accelerate execution of serial tasks, which limit speedup, although this could increase the amount of necessary hardware resources. This has been described by Hill & Marty [30].

In order to investigate the impact of specialized core architectures on the achievable speedup, two additional experiments were conducted. First, the overall speedup was measured depending on the number of specialized cores, then core performance and area were scaled. Again, task graphs of the *Parsync* type were used with a varying serial/parallel cycles ratio. Message size for all task graphs was set to zero (*M0*). All 18 tasks (2 serial, 16 parallel) were executed on a NoC with virtual cores of the two architecture types A_0 and A_1. In accordance with [30], architecture type A_1 represents an improvement of A_0 accelerating task execution by a factor of two, but with a four-fold increase in required hardware resources. The resulting NoC could hence feature 16 virtual cores of type A_0, four virtual cores of type A_1 or all remaining integer combinations. Fig. 15 shows the resulting speedup for all possible numbers of improved virtual cores. It is evident that the integration of a single core of architecture type A_1 can accelerate the overall execution, but the inclusion of more than one improved core does not increase it further in this scenario, as more and more A_0-type cores are sacrificed, lowering the parallel execution capability. In the case of *Parsync5M0*, the serial part is too small to benefit from accelerated execution, but parallel execution is slowed due to the lower number of available A_0-type cores.

In the second experiment, the architecture A_1 was scaled to increase performance and area requirement according to $\text{perf}(s) = \sqrt{s}$ [30]. The resulting speedups for task graphs with a high serial portion can be found in Fig. 16. It is noticeable that an optimal trade-off point of size and performance exists depending on the graph's degree of parallelism. For *Parsync30M0* and *Parsync50M0*, this point is reached when A_1 provides a performance increase of factor $\sqrt{11}$ requiring 11 times the size of A_0.

It can be concluded that task granularity and inter-task

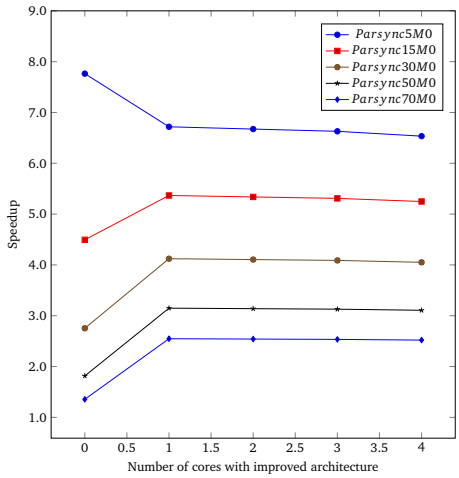


Fig. 15: Speed-up as a function of the number of specialized virtual cores

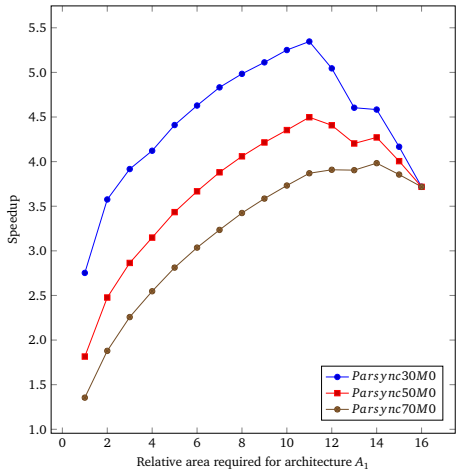


Fig. 16: Speed-up as a function of the size of an improved core. Overall speedup is plotted against the relative size of a core with improved architecture type A_1 compared with the base architecture A_0. A chip size of 16 A_0 units is assumed.

communication have a significant impact effectively limiting scalability. Specialized core architectures can be employed to fill the gap at the cost of silicon area increase. With the help of FNOCEE it is possible to quantify the aforementioned effects and investigate the impact of changes within the application, NoC or the executing cores.

V. Conclusions

In this publication FNOCEE, a framework for the evaluation by emulation of NoC-based many-core systems has been introduced. Applications are represented as task graphs and executed on virtual cores, allowing a closed-loop task mapping optimization using a hardware-accelerated genetic algorithm. After problem-matched parameter values for the genetic algorithm had been determined, different experiments were conducted to investigate scalability effects for an increased

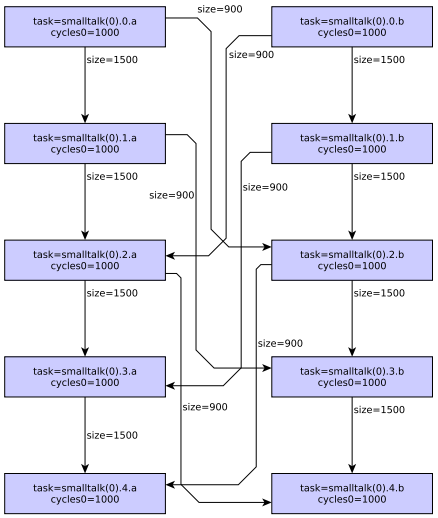


Fig. 17: Task graph *Smalltalk*: For an optimal mapping, all tasks of a column have to be mapped onto the same processor, as otherwise transfer times exceed the tasks' computation time.

number of virtual cores, increased traffic load and for varying core architectures.

FNOCEE proved to be a light-weight, yet high-performance framework able to evaluate more several thousand possible mappings cycle-true. Its logic-dominated design allows usage also on smaller FPGA-based emulation platforms. The high abstraction level for application modeling enables design space exploration for a wide range of parameters, while preserving the capability to observe dynamic execution effects on a cycle-true basis. Therefore, an early analysis of a specific NoC architecture can be performed during an early design phase without requiring the implementation of the complete system. Application behavior can be predicted with regard to scalability, the influence of inter-task communication and the use of heterogeneous SoCs.

Future works include the automated task graph generation based on application profiling information and the extension of FNOCEE's NoC component to allow broader comparisons of routing algorithms, topologies and introduce online-monitoring. Additionally, it needs to be determined, how genome coding for the genetic algorithm can be improved or whether a different optimization algorithm should be used.

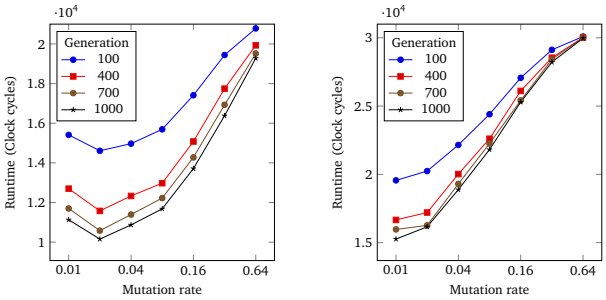
Appendix
Configuration of the Genetic Algorithm

Before performing actual experiments, optimal values for the genetic algorithm's parameters mutation rate and elite size had to be determined. For this purpose the task graph *Smalltalk* with 10 interdependent tasks was created. Each task needs 1000 cycles of execution time to be processed and generates two messages with a size of 1500 and 900 transfer cycles respectively (see Fig. 17).

In order to increase complexity and communication load, the task graph *Smalltalk* was instantiated four times in a new task

graph called *Smalltalk4*. With a fixed value of 6 for the number of elite individuals, the mutation rate was varied from 1 % to 64 % for a population size of 100. An increase in generations after which the optimization is terminated results in a lower overall execution time. A mutation rate of 2 % resulted in the lowest execution times. Each of the experiments were repeated 100 times to provide a more reliable statistical basis, therefore average values for minimal and average overall execution time are given in Fig. 18.

Due to emulation and hardware acceleration of the genetic algorithm it was possible to evaluate a 100k mappings each containing 40k cycles of execution and 96k cycles of communication time in 58 s, which results in an evaluation rate of 1,724 $\frac{\text{mappings}}{s}$. The performance is primarily limited by network-induced communication latencies between host pc and the emulation system.



(a) Average minimal execution time depending on mutation rate for 100 experiments

(b) Average mean execution time depending on mutation rate for 100 experiments

Fig. 18: Mutation rates from 1 % to 64 % were evaluated and are plotted here for selected generations

Afterwards the optimal value of elite individuals were determined identically. It was found that a higher number of elite individuals per generation lead to lower overall execution times, but a value of 10 for a population of 100 individuals resulted in the best mappings.

References

[1] L. Benini and G. De Micheli, "Networks on chips: a new soc paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[2] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob *et al.*, "An 80-tile 1.28 tflops network-on-chip in 65nm cmos," in *IEEE International Solid-State Circuits Conference, San Fransisco, USA, 2007*. IEEE, 2007, pp. 98–99.

[3] T. Corporation, "Tile-gx family of multicore processors," http://tilera.com/products/?ezchip=585&spage=614, Tilera Corporation, 2014.

[4] M. Neuenhahn, H. Blume, and T. Noll, "Quantitative analysis of network topologies for noc-architectures on an fpga-based emulator," *Proceedings of the URSI Advances in Radio Science–Kleinheubacher Berichte, Miltenberg, Germany*, 2006.

[5] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys (CSUR)*, vol. 38, no. 1, p. 1, 2006.

[6] L. Benini, "Application specific noc design," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*. European Design and Automation Association, 2006, pp. 491–495.

[7] T. Ahonen, S. Virtanen, J. Kylliäainen, D. Truscan, T. Kasanko, D. Sigäuenza-Tortosa, T. Ristimäaki, J. Paakkulainen, T. Nurmi, I. Saastamoinen *et al.*, "A brunch from the coffee table-case study in noc platform design," in *Interconnect-centric design for advanced SoC and NoC*. Springer, 2005, pp. 425–453.

[8] S. Chai, Y. Li, J. Wang, and C. Wu, "A list simulated annealing algorithm for task scheduling on network-on-chip," *Journal of Computers*, vol. 9, no. 1, pp. 176–182, 2014.

[9] M. Drozdowski, *Scheduling for parallel processing*. Springer, 2009.

[10] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.

[11] S. Gupta, G. Agarwal, and V. Kumar, "Task scheduling in multiprocessor system using genetic algorithm," in *Machine Learning and Computing (ICMLC), 2010 2nd Int. Conf. on*. IEEE, 2010, pp. 267–271.

[12] R. Hwang, M. Gen, and H. Katayama, "A comparison of multiprocessor task scheduling algorithms with communication costs," *Computers & Operations Research*, vol. 35, no. 3, pp. 976–993, 2008.

[13] M. R. Mohamed and M. H. Awadalla, "Hybrid algorithm for multiprocessor task scheduling," *IJCSI International Journal of Computer Science Issues*, vol. 8, no. 3, 2011.

[14] Y. Qu, J.-P. Soininen, and J. Nurmi, "Static scheduling techniques for dependent tasks on dynamically reconfigurable devices," *Journal of Systems Architecture*, vol. 53, no. 11, pp. 861–876, 2007.

[15] Z. Shi, "Scheduling tasks with precedence constraints on heterogeneous distributed computing systems," 2006.

[16] A. Wieferink, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, G. Braun, and A. Nohl, "System level processor/communication co-exploration methodology for multiprocessor system-on-chip platforms," in *Computers and Digital Techniques, IEE Proceedings-*, vol. 152, no. 1. IET, 2005, pp. 3–11.

[17] M. Neuenhahn, J. Schleifer, H. Blume, and T. Noll, "Quantitative comparison of performance analysis techniques for modular and generic network-on-chip," *Advances in Radio Science*, vol. 7, no. 8, pp. 107–112, 2009.

[18] P. G. Del Valle, D. Atienza, I. Magan, J. G. Flores, E. A. Perez, J. M. Mendias, L. Benini, and G. De Micheli, "A complete multi-processor system-on-chip fpga-based emulation framework," in *Very Large Scale Integration, 2006 IFIP Int. Conf. on*. IEEE, 2006, pp. 140–145.

[19] B. Senouci, F. Petrot *et al.*, "Large scale on-chip networks: an accurate multi-fpga emulation platform," in *Digital System Design Architectures, Methods and Tools, 2008. DSD'08. 11th EUROMICRO Conference on*. IEEE, 2008, pp. 3–9.

[20] G. Team, "The graphml file format," http://www.graphml.graphdrawing.org/index.html, GraphML Project Group, 2015.

[21] yWorks GmbH, "yed graph editor," http://www.yworks.com/en/products/yfilesd/yed, yWorks GmbH, 2015.

[22] O. Roeva, S. Fidanova, and M. Paprzycki, "Influence of the population size on the genetic algorithm performance in case of cultivation process modelling," in *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*. IEEE, 2013, pp. 371–376.

[23] D. Beasley, R. Martin, and D. Bull, "An overview of genetic algorithms: Part 1. fundamentals," *University computing*, vol. 15, pp. 58–58, 1993.

[24] S. Jin, G. Schiavone, and D. Turgut, "A performance study of multiprocessor task scheduling algorithms," *The Journal of Supercomputing*, vol. 43, no. 1, pp. 77–97, 2008.

[25] M. Kock, S. Hesselbarth, M. Pfitzner, and H. Blume, "Hardware-accelerated design space exploration framework for communication systems," *Analog Integrated Circuits and Signal Processing*, vol. 78, no. 3, pp. 557–571, 2014.

[26] K. Ø. Arisland, A. C. Aasbø, and A. Nundal, "Vlsi parallel shift sort algorithm and design," *INTEGRATION, the VLSI journal*, vol. 2, no. 4, pp. 331–347, 1984.

[27] C. Maxfield, *Bebop to the Boolean boogie: An unconventional guide to electronics*. Newnes, 2008.

[28] M. Li, Q.-A. Zeng, and W.-B. Jone, "Dyxy: a proximity congestion-aware deadlock-free dynamic routing method for network on chip," in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006, pp. 849–852.

[29] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[30] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, no. 7, pp. 33–38, 2008.