Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 453970, 14 pages
doi:10.1155/2009/453970

Research Article

A Decentralised Task Mapping Approach for Homogeneous Multiprocessor Network-On-Chips

Peter Zipf,[1] Gilles Sassatelli,[2] Nurten Utlu,[3] Nicolas Saint-Jean,[2] Pascal Benoit,[2] and Manfred Glesner[3]

[1] *Digital Technology Lab, University of Kassel, Wilhelmshöher Allee 73, 34121 Kassel, Germany*
[2] *Laboratoire d'Informatique, de Robotique et de Microélectroniqe de Montpellier (LIRMM), University of Montpellier II, UMR CNRS 5506, 161 rue ADA, 34392 Montpellier Cedex 5, France*
[3] *Institute of Microelectronic Systems, Darmstadt University of Technology, Karlstrasse 15, 64283 Darmstadt, Germany*

Correspondence should be addressed to Peter Zipf, zipf@uni-kassel.de

Received 27 December 2008; Accepted 25 May 2009

Recommended by Michael Huebner

We present a heuristic algorithm for the run-time distribution of task sets in a homogeneous Multiprocessor network-on-chip. The algorithm is itself distributed over the processors and thus can be applied to systems of arbitrary size. Also, tasks added at run-time can be handled without any difficulty, allowing for inline optimisation. Based on local information on processor workload, task size, communication requirements, and link contention, iterative decisions on task migrations to other processors are made. The mapping results for several example task sets are first compared with those of an exact (enumeration) algorithm with global information for a 3×3 processor array. The results show that the mapping quality achieved by our distributed algorithm is within 25% of that of the exact algorithm. For larger array sizes, simulated annealing is used as a reference and the behaviour of our algorithm is investigated. The mapping quality of the algorithm can be shown to be within a reasonable range (below 30% mostly) of the reference. This adaptability and the low computation and communication overhead of the distributed heuristic clearly indicate that decentralised algorithms are a favourable solution for an automatic task distribution.

1. Introduction

On-chip multiprocessing provides the computing power and parallelism required for many of today's real-world applications with high data rates. The diminishing returns of Instruction Level Parallelism (ILP) point the interest to higher levels of applications, where explicit Thread Level Parallelism (TLP) can be exploited [1]. A logical consequence of increasing performance demands is to use both ILP and TLP simultaneously by integrating a large number of processors in one Multiprocessor System-on-Chip (MPSoC). At the same time, reduced clock frequencies for the individual processor cores enable a large reduction of the overall power consumption while keeping the system performance up.

Multiprocessor systems can only be utilised sufficiently, if the software running on them can be separated into sets of communicating tasks working in parallel. These tasks are then distributed over a set of processors sharing the workload. For a well-known set of tasks and workloads, the distribution can be precalculated for an optimal mapping. For applications with unpredictable workload like, for example, user-induced multimedia processing, and subsequently unpredictable changes in active tasks and communication requirements, a run-time task mapping depending on the actual resource utilisation must be applied to balance the processor loads.

In this paper we present a decentralised task mapping heuristic for task sets on an MPSoC. The heuristic running on each processor is capable of reconfiguring the system by migrating individual tasks to neighbouring processors based on the local workload, task sizes, and communication requirements of the tasks to be migrated. It is not restricted to a final set of tasks but can also handle task sets added during operation, thus supporting a reconfiguration at task level. Due to its scalability, a homogeneous Network-on-Chip (NoC) structure is used as the underlying hardware

architecture, which is essential for the developed task mapping heuristic. An experimental implementation of the multiprocessor platform based on interconnected FPGA prototyping boards is used to investigate the potential of decentralised task distribution and workload balancing algorithms.

1.1. Multiprocessor Network-on-Chips. An MPSoC is a special form of SoC; where the functional modules are all processor modules. Due to the advantages on-chip design offers, like a free choice of bus bit widths or high data transfer rates, such systems can be adapted very well to their specific requirements. Based on the envisioned application scenario of multimedia workloads, which are characterised by structured and regular computations, some additional desired properties for the system can be derived. This refers mainly to the communication model, the processor types, and the physical interconnect architecture. Multiprocessor systems can be based on shared memory or message passing communication. For large high-performance systems with up to several hundred processors, only a communication based on message passing is reasonable [2, 3], combined with distributed local memory. To enable a simple task distribution, a homogeneous MPSoC should be preferred, where each node consists of an identical processor to present a uniform (homogeneous) array.

The components of MPSoCs are usually connected by point-to-point or bus-based structures. Both interconnect concepts cannot be scaled well for larger numbers of processors, for example, exceeding 50. A Multiprocessor Network-on-Chip (MPNoC) uses a Network-on-Chip [4] structure to interconnect its processor modules. A set of interconnection segments is combined to a network by routers. Data sent from one Processor is then relayed from one router to the next until it reaches its destination [5]. Such a MPNoC called HS-Scale [6] is used in our work.

1.2. The Task Mapping Problem. For the envisioned data flow applications, a high overall system throughput is the dominant requirement, surpassing short latencies as needed, for example, in closed loop control systems. In order to improve throughput, tasks must be mapped in the right way. The main question to be answered for a task mapping is: what makes one mapping better than another? Consequently, the objective is to reduce (a) the average distance of travelling data packets and (b) the workload on the individual processors. In addition, the maximum bandwidth on the communication links should not be exceeded. These objectives are specific for on-chip scenarios where individual interconnects are not the most dominant limitation and the network topology including all its parameters is fixed and known in advance.

Two major concepts in developing task mapping strategies are the graph theoretic approach and the mathematical programming approach [7]. Although rapid advances in both the methodology and application of graph theoretic models have been realised, many models actually are special types of linear programming problems [8]. Task mapping considering traffic generation is a nonlinear problem, which limits the usability of common graph theoretic approaches. Due to the unsatisfactory support of nonlinear task mapping by graph-based methods, in this work flexible mathematical programming for developing an algorithm is used.

1.3. Section Overview. Section 2 introduces some relevant previous work on the task mapping problem for multiprocessor systems. Section 3 describes the heuristic algorithm developed and an exact algorithm used for comparison. Section 4 discusses some experimental results obtained by running the algorithms on a set of example task sets. In Section 5 the performance of the heuristic algorithm for larger network processing unit (NPU) arrays is investigated based on a large number of random task sets. Section 6 concludes with a summary and some final remarks.

2. Related Work

The aim of this work is to develop a run-time task mapping algorithm for MPNoCs to balance the system throughput. This is done by considering the two conflicting requirements *maximisation of average processor utilisation* and *minimisation of the contention on links* caused by intertask communication. A classification of some relevant related work on task mapping is given in Table 1. The main categories are the factors taken into account for the mapping (computation and/or traffic), the flexibility of the mapping process (static or dynamic), and the way it is implemented (centralised or decentralised).

The first category is based on the target factors taken into account to achieve the mapping goal. In [9], only the network bandwidth is considered but not the computing requirements of the applications. The aim of [10] is the minimisation of total communication time for sets of similar tasks. Other factors like congestion are not considered. Also, workload balancing is only done by mapping exactly one task to one processor. A more general load balancing model considering job and resource migration is used in [11]. As communication bandwidth is assumed to be sufficient, the mapping depends only on the communication distance and is independent of the network traffic. In contrast, a mapping optimisation regarding computation and traffic is given in [12]. The goal is to minimise the total execution and communication costs. Communication costs are used as an attracting force between tasks, causing them to be assigned to the same processor. The costs of incompatibilities between tasks are used as a repulsive force, causing a task distribution over several processors. Communication costs occur if two tasks are assigned to different processors and are independent of the congestion on the links. They are not explicitly specified, but occur as the multiplication of the communication flow between two tasks and the distance between the processors they are mapped to. The mapping problem is solved by a Max Flow/Min Cut algorithm in combination with a greedy algorithm. In [13] also the total execution time is minimised by weighting the computation of each task and each interaction between tasks. The resulting

Table 1: Classification of related work on task mapping.

Group	Description	Examples
Computation or Traffic:	Maximum utilisation of processors or minimum traffic generation	[9–11]
Computation and Traffic:	Maximum utilisation of processors and minimum traffic generation	[12–14]
Static Mapping:	Offline or predictive mapping at design time	[12, 15–17]
Dynamic Mapping:	Supports run-time task migration (and injection)	[18–20]
Central Mapping:	Global view, Master Slave	[12, 14, 19–21]
Decentrlised Mapping:	Local view	[22]

cost function is minimised by a hybrid of a genetic algorithm and mean field annealing. The turnaround time is improved in [14]. After defining execution and communication costs simulated annealing is used.

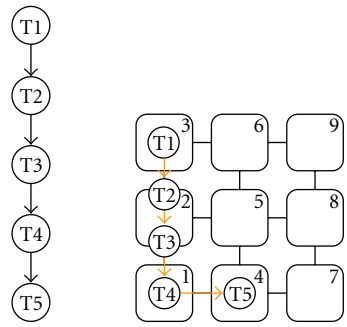
While workload balancing tries to exploit parallel execution in space by distributing all tasks regarding the computation demand evenly among the processors, intertask communication tends to exploit computation in time, by mapping the whole application to a single processor in order to save bandwidth of communication links [23]. Task Mapping differs regarding the time at which assignment decisions are made. Most authors propose the use of static mapping [12, 15–17], according to most of the current real-time operating systems of embedded systems [18]. Static mapping is less complex and easier to design than dynamic mapping. The assignment is defined prior to the application execution at design time and is not changed any more later on. To improve the performance of dynamic workloads at run-time, task migration has been used [18–20] to relocate tasks in order to distribute the workload more homogeneously among the resources. Differently from task migration, dynamic mapping can insert new tasks into a system at run time [24].

For the decision-making policy of task mapping, the two fundamental models centralised and decentralised can be considered. In a centralised model [12, 14, 19–21], one specialised master processor and an arbitrary number of slave processors are used [20]. The master has global knowledge of the application characteristics and of the distributed system [12]. It performs task mapping, aiming at an equal distribution of the load among the slave processors and communication links. The centralised task mapping allows a globally coordinated and hence efficient placement mechanism, however at the cost of scalability. An increasing number of processors in future systems or a great number of tasks will overload the master. In decentralised models, the authority for task mapping is shared among all processors. Because of the absence of a global view, knowledge of application and processor characteristics is shared by the exchange of messages. All decisions for the task mapping are made from local interaction laws.

Typical applications running in MPNoCs like multimedia and networking display a dynamic workload of tasks. This implies a varying number of tasks running simultaneously [24]. It is impossible to foresee and specify an appropriate response for every potential run-time scenario before the application execution. Therefore, unpredictable information like task arrival times, workload of processors, and contention on the links must be gathered during execution. This work considers dynamic mapping for MPNoCs, which supports varying workloads by task injection and target load distribution by task migration. Tasks are mapped on the fly, according to computation and communication requirements, following a distributed (decentralised) mapping scheme, considering both computation (workload) and traffic data.

3. The Heuristic Task Mapping Algorithm

Since scalability of the platform architecture and programming model will be a major challenge for MPSoC designs in the years to come, a platform providing a large number of processors must discard all non-scalable properties. Our hardware platform HS-Scale [6] is a homogeneous MPSoC based on programmable RISC processors, small distributed memories, and an asynchronous Network-on-Chip (NoC). The software model is a multithreaded sequential programming model with communication primitives handled at run-time by a simple multitasking operating system specifically developed for the platform; the threads are described in C language. The HS-Scale framework guarantees any application to be executed independent of the platform settings, specifically the number of processing elements (PE) and the chosen task mapping. The communication is abstracted via communication primitives, so that tasks can communicate with each other without knowing their position in the system. The communication primitives were derived from 5 of the 7 layers of the OSI model, allowing transparent data communications between tasks either locally or remotely: the routing is done following a dynamic routing table. If the task is local, the writing of data is done on a local software FIFO. If it is a remote task, the operating system must assure that there is enough space for the remote software FIFO to avoid deadlocks on the network. This is done using dedicated functions. As soon as the OS gets a positive answer, it can start encapsulating and sending the data packets to the



(a) Task graph (b) Task placement

Figure 1: A task graph and its placement on an NPU array.

Task	T1	T2	T3	T4	T5
NPU	3	2	2	1	4

Figure 2: Task placement representation.

remote task while the remote task can de-encapsulate and receive the data packets and write them to its local software FIFO. A lightweight operating system has been developed for the specific needs of the MPNoC platform. The OS provides preemptive task switching and communication support for task interactions using the communication primitives [6].

Load balancing, the overall communication bandwidth, and the local communication bandwidth have to be considered for the task mapping. This section introduce the implemented algorithms after defining the underlying model and a mathematical problem formulation.

3.1. Problem Definition and Model Formulation. To reduce the average distance of travelling data, the number of data packets and the distance between the communicating network processing units (NPUs) must be known.

The mapping alternatives are determined by using an appropriate solution representation and by modifying representations (solutions). Every possible solution can be represented by a Table with two rows (see also Figure 2). The first row is an ID list of all existing tasks without repetition. This constraint results from the fact that each task must only be mapped exactly once. The second row contains the IDs of used NPUs. Because each NPU has multitasking capabilities which enables a time-sliced execution of tasks, a repetition of NPU IDs is allowed. Not all NPUs need to be used and thus some need not appear in the second row. As an example, the task graph of Figure 1(a) with five consecutive tasks is mapped on the array with 3×3 NPUs shown in Figure 1(b). The according solution representation Table is shown in Figure 2.

The target hardware architecture is a homogenous array. Therefore it is possible to assign any task to any NPU. New solutions can easily be generated by exchanging NPUs in the second line of the solution representation by other existing NPUs. This is equivalent to the combinatoric variation with repetition, where order matters and an object can be chosen more than once. The number of possible variations with repetition is given by

$$\mathrm{NPU}^{\mathrm{Task}}, \tag{1}$$

where NPU is the number of available NPUs to be chosen from and Task is the number of tasks to be placed.

The problem can now be formulated as follows. Given the computation time for every task and the data flow between communicating tasks, find a task placement that reduces the distance through which data travels and balances computation load. Each task has to be assigned to a single NPU and each NPU can execute multiple tasks.

The communication costs $f_{i,k}$ between task i and task k depend on the distance $d_{j,l}$, determined by the position of NPU j to which task i is assigned ($x_{i,j} = 1$) and NPU l on which task k is assigned ($x_{k,l} = 1$). The problem is a quadratic assignment problem (QAP) [8]. The formulation of the overall bandwidth minimisation can be given as.

$$\text{minimise } z = \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{(k=1,k\neq i)}^{n}\sum_{(l=1)}^{m} f_{i,k}\cdot d_{j,l}\cdot x_{i,j}\cdot x_{k,l} \tag{2}$$

The load balancing between NPUs can be considered as the linear assignment problem (LAP), where each task i in the task graph has been assigned a constant computational complexity t_i, where $t_{i,j}$ is this cost when task i is assigned to NPU j:

$$\text{minimise } z = \sum_{i=1}^{n}\sum_{j=1}^{m} t_{i,j}\cdot x_{i,j}. \tag{3}$$

subject to

$$\sum_{j=1}^{m} x_{i,j} = 1, \quad i = 1,\ldots,n, \tag{4}$$

where n is the number of tasks, and m is the number of NPUs. This constraint guarantees that task i is assigned to exactly one NPU.

To consider local bandwidth, the congestion $c_{j,l}$ on the links between NPU j and NPU l must also be included. A complete formulation of the objective function can be to minimise z, where

$$\begin{aligned} z = {} & \alpha\cdot\sum_{i=1}^{n}\sum_{j=1}^{m} t_{i,j}\cdot x_{i,j} \\ & + \beta\cdot\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{(k=1,k\neq i)}^{n}\sum_{(l=1)}^{m} f_{i,k}\cdot d_{j,l}\cdot x_{i,j}\cdot x_{k,l} \\ & + \gamma\cdot\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{(k=1,k\neq i)}^{n}\sum_{l=1}^{m} c_{j,l}\cdot x_{i,j}\cdot x_{k,l} \end{aligned} \tag{5}$$

subject to (4).

Equation (5) considers load balancing, overall bandwidth and local bandwidth, weighted by the scaling factors $0 \leq \alpha, \beta, \gamma \leq 1$.

Table 2: Area scalability and power consumption results for 90 nm technology.

No. of NPUs	1	2	2×2	3×3
Area (mm^2)	18.22	36.63	73.61	165.30
Power (mW/MHz)	2.56	5.14	10.34	23.26

Table 3: FPGA synthesis results.

Module	NPU	Router	Processor	Other
No. of Slices	2496	683	1462	351
% of Slices	14.4	3.9	8.5	2.0

3.2. The Task Mapping Algorithms. Three task mapping algorithms have been implemented. The first one is an exact algorithm based on complete enumeration. It delivers one solution which is guaranteed to be as good as any other objective function value. This algorithm is only used for small examples (up to 9 NPUs and 11 tasks) and as a reference, because (5) contains a modified QAP formulation (QAP problems have been shown to be NP-hard [25]) and for a complete enumeration NPU^{Task} solutions have to be generated. The program flow is shown in Figure 3. All solutions are generated, evaluated, and the best value encountered is returned as the result.

The second algorithm is a constructive algorithm. Its results are used as the starting point for the main improvement heuristic. to produce a feasible initial mapping solution, the constructive algorithm is run on one task injection (boundary) NPU. Initially only, global information is available, because no task is running on any NPU. Also, the 2D mesh structure of the hardware is used based on a reachability measure.

All NPUs are evaluated regarding their reachability. For illustration the array given in Figure 4 is used. The distance between two NPUs is given by the number of required hops, as shown in Figure 4(a). The sum of hops from NPU 1 to all other NPUs is 18. Applying this procedure to all NPUs gives their reachability. It can be seen in Figure 4(b) that the NPU with the best reachability is in the centre. To avoid overloading NPUs with good reachability, the reachability of NPUs which run tasks is penalised proportional to their computation time.

The program flow of the constructive algorithm is shown in Figure 5. Output tasks are sinks in the task graph. All tasks on the injection NPU are mapped to the remaining NPUs. This is done starting with the input task of each application's task set and continued by moving on to its successor tasks.

The constructive algorithm is activated once. Later the improvement algorithm is started on all NPUs until a steady state is reached. The closer the initial solution to the optimum, the fewer the number of required operations during the following improvement procedure. However, a good solution usually requires a complex algorithm and high computational effort. The proposed constructive heuristics balances the desire for a high quality initial solution and a simple algorithm, which is easy to implement and does not require extensive computations.

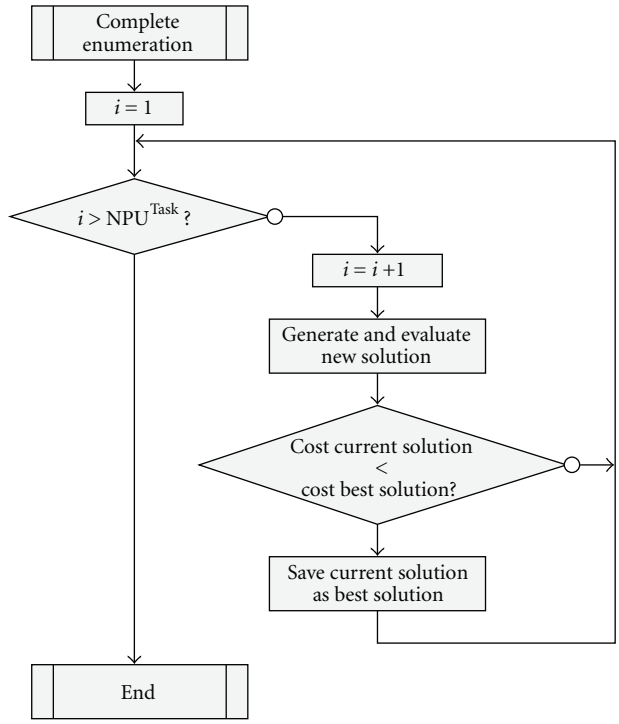
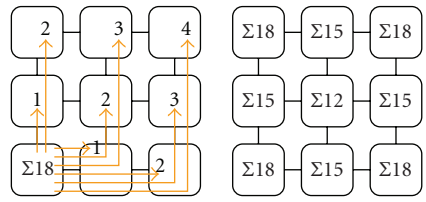


Figure 3: Program flow: exact algorithm.



(a) Reachability of NPU 1 (b) Reachability of all NPUs

Figure 4: Reachability calculation.

The third algorithm is a hybrid tabu search and force directed improvement algorithm. It is a distributed algorithm meant to run on each NPU if required. A model of spring-connected weights is used as its basis. Weights correspond to tasks and springs to communication between the tasks. A spring will try to pull its tasks closer together or push them apart, depending on its stiffness which is proportional to the quality (objective function rating) of the considered neighbourhood. The algorithm starts with a stiffness of distance 0 and 1. The NPU of the sending task and its neighbours which have a distance of one hop are considered first. If the objective function value worsens, the stiffness value is incremented to consider a growing neighbourhood. Assignment of tasks with high communication demands are prioritised. In order to achieve proper tradeoffs between time spent looking for solutions and the quality of the solutions found, a feature of tabu search, the candidate list strategy feature of the tabu search is applied [26]. The candidate list is used as a penalty Table

Table 4: Evaluation results for the task graph (TG) examples for $\alpha = \beta = \gamma = 1$ in (5) for constructive algorithm (CA), improvement algorithm (IA), and exact algorithm (EA).

Examples	Value	CA	IA	EA
TG 1	OF	1,161,862	941,508	941,380
	%	123.4	100.0	100.0
	LB	256	384	320
	OB	256	384	320
	CL	1,161,350	940,740	940,740
TG 2	OF	1,657,440	942,148	941,764
	%	176,0	100,0	100.0
	LB	832	704	512
	OB	448	704	512
	CL	1,656,160	940,740	940,740
TG 6	OF	1,359,998	762,921	762,157
	%	178.4	100.1	100.0
	LB	6,720	7,932	6,848
	OB	2,240	2,688	3,008
	CL	1,351,038	752,301	752,301
TG 7	OF	1,378,832	909,235	732,787
	%	188.2	124.1	100.0
	LB	8,568	7,680	5,376
	OB	2,432	2,624	2,176
	CL	1,367,832	898,931	725,235
TG 8	OF	2,565,628	2,108,464	2,009,720
	%	127.7	104.9	100.0
	LB	704	832	640
	OB	704	832	640
	CL	2,564,220	2,106,800	2,008,440
TG 9	OF	1,591,952	869,200	868,816
	%	183.2	100.0	100.0
	LB	384	640	448
	OB	128	640	448
	CL	1,591,440	867,920	867,920
TG 15	OF	2,958,737	2,334,168	2,033,084
	%	145.5	114.8	100.0
	LB	768	1,024	832
	OB	704	1,024	832
	CL	2,957,265	2,332,120	2,031,420
TG 17	OF	2,819,944	2,406,374	2,008,914
	%	140.4	119.8	100.0
	LB	896	1,280	512
	OB	768	704	512
	CL	2,818,280	2,404,390	2,007,890
TG 24	OF	359,000	232,869	230,629
	%	155.7	101.0	100.0
	LB	960	2,880	1,216
	OB	896	1,472	896
	CL	357,144	228,517	228,517
TG 26	OF	1,820,626	942,789	942,597
	%	193.1	100.0	100.0
	LB	1,952	1,088	928
	OB	672	960	928
	CL	1,818,002	940,741	940,741

Table 5: Comparison of the constructive algorithm (CA), improvement algorithm (IA) and exact algorithm (EA).

Algorithm		CA	IA	EA
HW throughput (bytes/s)		133,118	240,365	266,237
% of Exact		49.99	90.28	100
Contention on links	256 byte link	1	0	0
	128 byte link	1	2	2
	64 byte link	1	7	4
Overall Communication OB		448	704	512
Computational Load CL		1,656,160	940,740	940,740

Table 6: Relative OF-minima (values in % of random mapping).

	Minima [NPUs]		Relative value [%]	
Task count	Annealing	Heuristic	Annealing	Heuristic
10	9	16	55.4	64.5
20	16	16	49.9	63.8
30	25	25	52.5	65.3
40	36	36	56.2	64.9
50	36	49	61.9	66.6
60	49	49	65.5	67.4
70	49	64	68.1	68.8
80	49	81	71.8	69.8
90	64	64	71.9	69.7

which includes one element for each NPU. For example, a penalty is applied if the algorithm could not attain a better objective function value. After a certain value in the candidate list is reached, for example, a certain number of unsuccessful repetitions of the algorithm, the corresponding NPU is marked tabu and is no longer allowed to run the task mapping algorithm. This procedure provides three mechanisms.

(i) Avoid cycling by setting NPUs tabu if the improvement algorithm repeatedly cannot reach a better objective function value.

(ii) Intensification of the search by remaining NPUs, excluding nonpromising regions.

(iii) Termination criterion for the algorithm: eventually, all NPUs will be marked tabu.

The program flow of the improvement algorithm is shown in Figure 6. First, the algorithm checks whether a task is assigned to the NPU on which it runs. If no task is available, the tabu candidate originally set to zero will be incremented by one. Otherwise, all successor tasks are determined, except those with output flow which are not allowed to migrate (sinks). If no valid successor exists, the tabu candidate value is increased by one and the task is excluded from consideration. If successors exist, the successor with maximum receiving flow will be selected. According to the objective function, the NPU costs consist of the computation workload on the sending NPU, the computation workload on the receiving NPU, the distance between task and considered successor multiplied by the flow, and finally of the congestion between sending and receiving NPU. The flow to the successor is then checked to determine if it is so high that it is worth assigning both tasks to execute on the same NPU, despite the increasing computational demand. If this is not the case, the successor will be assigned to the neighbour NPU of distance 0 and 1, with minimum sum of congestion and NPU workload. If the NPU costs worsen, the neighbourhood is expanded to a distance of 2 and the value of the tabu candidate is incremented. This procedure of neighbourhood expansion will be continued until the NPU costs are at least equal to the old NPU costs. The improvement algorithm is repeated on the considered NPU as long as its repetition is not forbidden by the tabu list.

4. Experimental Results

A complete synthesisable RTL model of the HS-Scale hardware has been designed. The VHDL model was synthesised with a 90 nm ST Microelectronics design kit. The NPU clock has been constrained to 3 nanoseconds allowing a 300 MHz clock frequency. Table 2 summarizes the results. The model has been placed and routed with a 64 KB local memory, which occupies 87% of the total NPU area. Of the remaining 13%, the processor occupies 54%, the router 38%. Other elements (UART, interrupt controller, network interface, etc.) occupy about 7%. Table 2 clearly shows the areascalability of the MPNoC hardware platform and gives the estimated power consumption.

The very first validations of the system were performed using RTL simulations. Since this method is too slow for

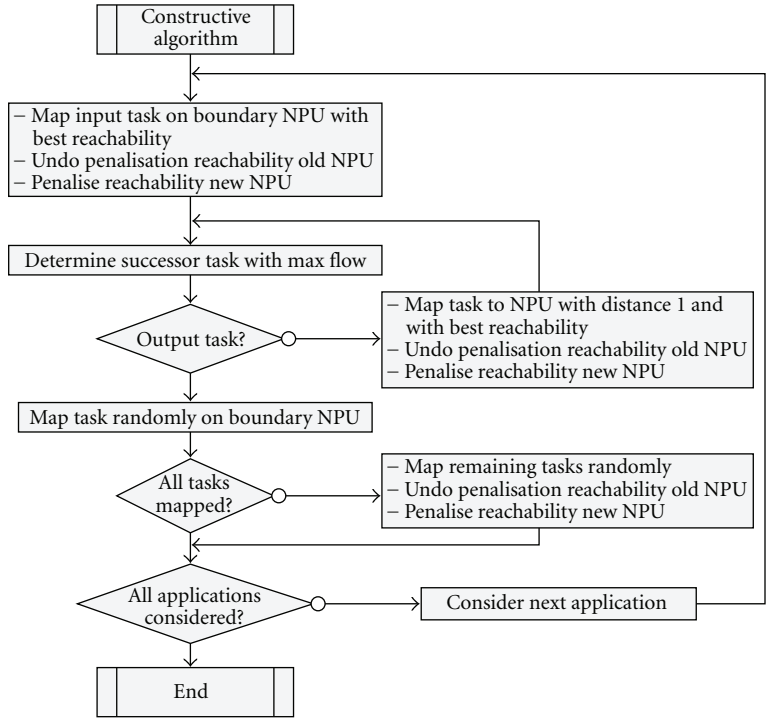


Figure 5: Program flow: constructive algorithm.

running realistic application scenarios, a prototype system using Xilinx Spartan-3 XC3S1000 FPGAs on Xilinx Starter Kit FPGA boards was realised. Table 3 gives the device utilisation for a single NPU on a single XC3S1000 FPGA providing 17,280 logic cells. Each NPU is placed on one board. The complete prototype is then composed of several prototyping boards connected by ribbon cables. This allows for easy extension of the system by adding further FPGA boards.

A set of 27 task graphs was used as examples to evaluate the quality of the constructive and improvement algorithms. The properties of the graphs, that is, computational and communication requirements, are taken from real applications, for example, Motion JPEG video-codec. Variations were generated by duplicating tasks to enable load sharing or by iterative execution of tasks. The example task sets range between 5–11 tasks, distributed to 1–4 independent applications, that is, independent data flows. Figures 7(a), 7(b), and 8(a) show examples of the task graphs used, including computational and communication requirements (given in clock cycles and bytes resp.). In task graph 6 (Figure 7(a)) the tasks 2, 3, and 4 have been replicated twice for load sharing, while at the same time also increasing communication (arrows). Due to the problem complexity for the exact mapping solution, the target array was limited to 3×3 NPUs. Table 4 shows a representative selection of the data obtained from the evaluation. The rows for local bandwidth or link contention (LB), overall bandwidth (OB), and computational load (CL: z from (3)) show the respective algorithm representation of these values. The individual values for the objective function (OF) of the calculated mappings and their relation to the exact results are given.

The average deviation between the results of the improvement algorithm and the exact solution is 6.47% for the given examples, and the maximum difference is below 25%. TG 1 contains task 2 with a computational requirement of 494,810. TG 2 is a parallelised version of TG 1, where task 2 has been replicated once (task 20), resulting in a computational requirement of 247,405 for each of task 2 and 20. It can also be seen that a load balancing can easily be done at the cost of increased communication (OB of the exact algorithm increases from 320 to 512, corresponding to the two additional communication links with costs of 128 and 64 bytes per block calculation). The CL of the exact algorithm for TG 1 and TG 2 are identical because no changes in the computational complexity arises by duplicating tasks. From the viewpoint of the computational loadbalancing, it can be seen by comparing the CL values of TG 2, that the construction algorithm provides an inferior solution, whereas the improvement algorithm and the exact algorithm provide solutions with equal quality (visualised in Figure 9).

Figure 8 shows the task graph model of application example 2 with 6 tasks and a 3×3 NPUs graph. The task graph of Figure 8(a) was mapped by all three algorithms on an FPGA-based NPU array implementing the 3×3 array of Figure 8(b). Figure 9 shows the mapping results for the three algorithms. Table 5 gives the corresponding throughput numbers measured on a VHDL simulation of the hardware platform running at 7 MHz. It can be seen that the result of the improvement algorithm for the example

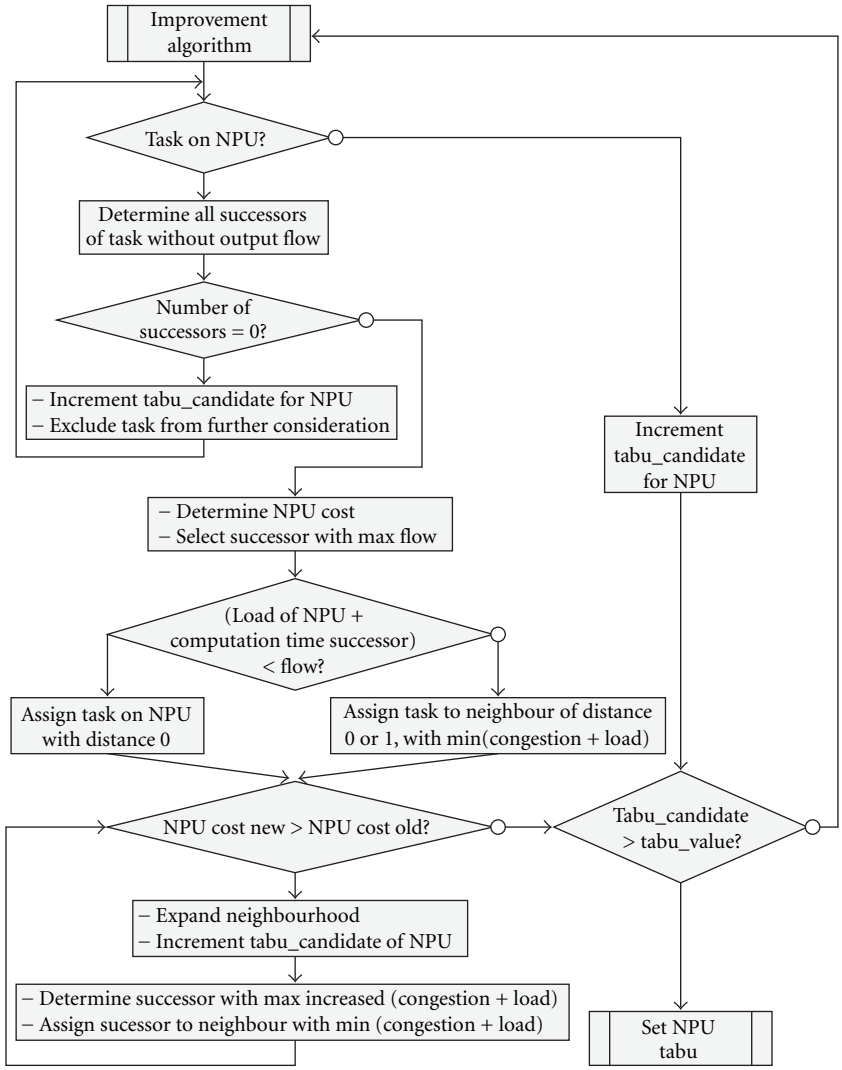


Figure 6: Program flow: improvement algorithm.

is within 10% (90.28%) of the best solution. The local and overall communication requirements (abstracted values for the objective function) and the computational load of the NPUs as computed by the three algorithms are also given.

5. Results for Larger Arrays and Task Sets

The previous results indicate the feasibility of the proposed decentralised placement heuristic. The general performance of the heuristic can only be evaluated by considering a larger range of array sizes and task counts. The exact enumeration algorithm cannot be used as a reference for array sizes above 3×3 and more than about 12 tasks because of the high complexity of $O(\text{NPU}^{\text{Task}})$. Instead, we use a simulated annealing algorithm to optimise the task mapping problem with global knowledge for larger arrays and higher numbers of tasks. These results can be compared to the results of our heuristic.

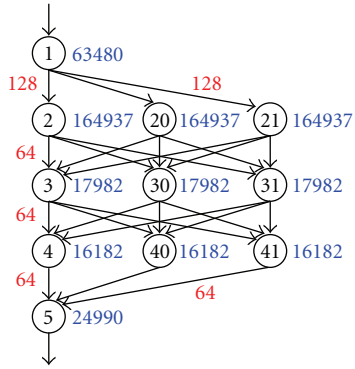
5.1. Experimental Settings and Data. To gain significant information on the behaviour of the algorithms, a large number of experiments must be made for different array sizes and task counts. A task graph generator was implemented to produce random task graphs. Each task graph is characterised by the number of nodes (tasks) it contains, the number of unconnected subgraphs (task groups or processes), and the specific values for the computational load of each task and the communication bandwidth of each edge (data communication between tasks). For our experiments the following parameters are varied.

(i) Array size: array sizes from 1×1 to 9×9, that is, from 1 to 81 NPUs.

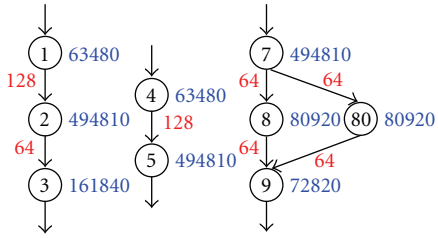
(ii) Task count: task sets with between 10 and 90 tasks.

(iii) Process count: values between 2 and 8 have been used.

The graph generator software produces a number of samples for each parameter combination, for example, 100 graphs

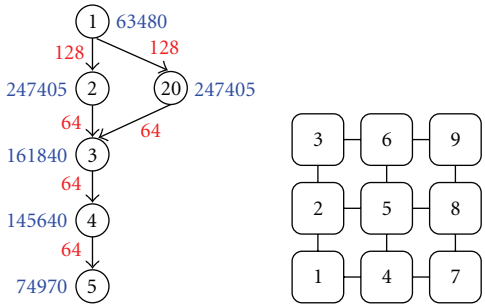


(a) Task graph 6



(b) Task graph 17

Figure 7: Examples of the task graphs used.

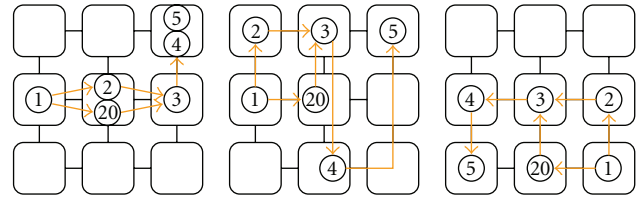


(a) Task graph 2 (b) Processor graph

Figure 8: Modelling example for an application and NoC processor array.

with 25 tasks, and 4 independent processes, with a randomly distributed number of tasks per process. Figure 10 shows one of the task graphs generated during the experiments. It contains 14 tasks arranged in 3 independent groups (processes) which are meant to run in parallel on the NPU array. Each task graph is then handed to the heuristic and the simulated annealing algorithm for placement. Additionally, a random placement is also generated. The resulting objective function values for all three obtained placements are saved as average, minimum, and maximum values over all samples for each parameter combination.

Simulated annealing is known as a good heuristic approach for problems with a largely unknown solution space structure and should produce reasonable reference results.



(a) Constructive (b) Improvement (c) Exact

Figure 9: Comparison of constructive, improvement and exact algorithm.

To get some general information about the design space, two considerations can be made. Firstly, we will assume that input and output tasks must be placed on a boundary NPU. For array sizes above 2×2, the number b of boundary NPUs grows linearly with the square root of the NPU count ($b = 4(\sqrt{n} - 1) = 4\sqrt{n} - 4$, $n \geq 4$, with n being the (square) number of NPUs), while the number i of internal NPUs grows linearly with the NPU count ($i = (\sqrt{n} - 2)^2 = n - 4\sqrt{n} + 4 = n - b$, $n \geq 4$), that is, the fraction of boundary NPUs $b/i = b/(n - b)$ shrinks. While this does not reduce the complexity class of the problem, it still reduces the number of valid mappings due to the fact that input and output tasks must be mapped to boundary NPUs. The number of valid mappings m_v is given by

$$m_v = \left(4(\sqrt{n} - 1)\right)^{2g} \cdot (\sqrt{n} - 2)^{2(t-2g)}, \tag{6}$$

where n is the number of NPUs, and t is the number of tasks, and assuming that each task group g must have at most one input and one output task (or one-task processes, this can be the same task, so $2g$ is an upper limit). The first term gives the number of boundary NPUs, while the second term gives the number of "inner" NPUs of the array. The break even point of b and i is between array sizes of 6×6 and 7×7 (36 and 49 NPUs). For 9×9 array, there are 32 boundary NPUs and 49 inner NPUs, so a large predominance of inner NPUs needs not to be considered.

Secondly, some information about the objective function values to be expected can be obtained by examining random mappings or by using the simulated annealing algorithm to search for worst case solutions. Figure 11 shows the values of random task mappings for 50 tasks and different array sizes, averaged over 1000 samples each (please note the logarithmic scale for the y-axis). The error bars give the range between the best and the worst mapping value found within the samples. It can be seen that larger arrays allow for more efficient mappings according to the objective function. Also, a saturation effect can be observed towards larger arrays.

5.2. Mapping Evaluation. The obtained data can be analysed and the quality of the heuristic mapping results can be rated in relation to the simulated annealing results. Figure 12

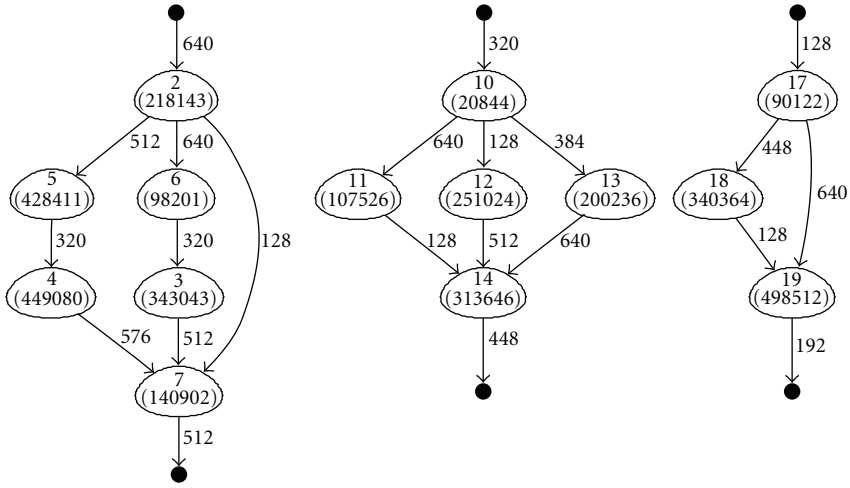


Figure 10: Example: generated task graph.

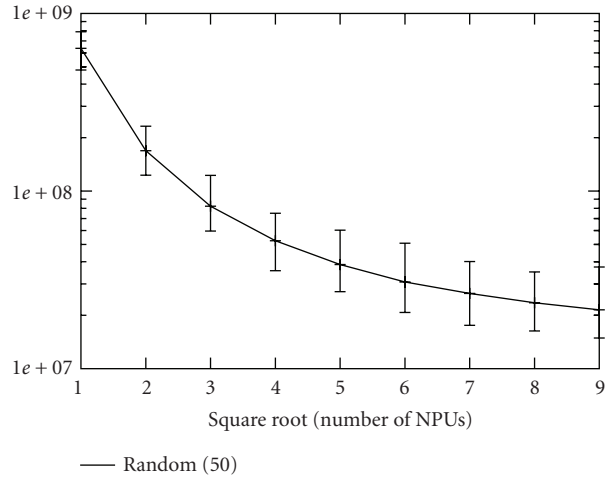


Figure 11: Objective function value space for 50 tasks and random mapping.

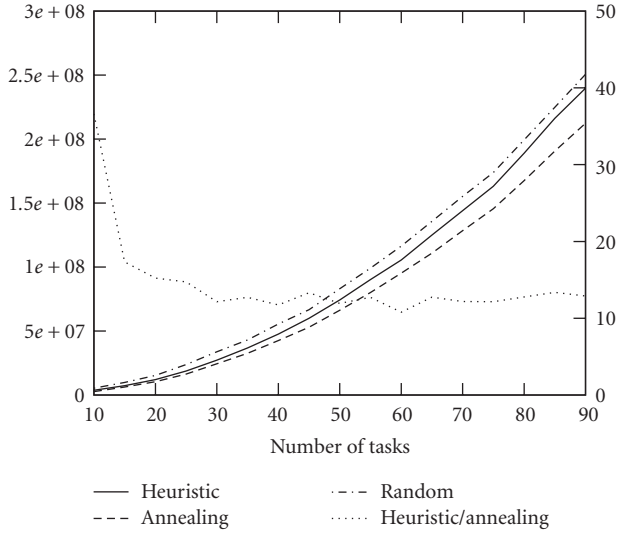


Figure 12: Objective function values (left y-axis) on a 3×3 NPU array and ratio in percent (right y-axis).

shows the objective function results for the heuristic and those for the simulated annealing algorithm for the same task graphs for a 3×3 array (4 processes); Figure 13 shows the same data for a 6×6 array (8 processes). The dotted line (values in %, right y-axis) gives the ratio between simulated annealing and the heuristic. It can be seen that the heuristic performs better for higher numbers of processes. For 8 processes, the heuristic delivers never more than 30% less well results than simulated annealing for arrays between size 3×3 to 5×5, while even keeping below 20% for arrays larger than that. For 4 processes, the heuristic results are never more than 45% worse than simulated annealing, but less than 35% in the great majority of examples. This is true over all data sets, that is, for all array sizes.

Figures 14 and 15 show the mapping development for a fixed task size of 30 and 60 tasks, respectively, and different array sizes (data shown for 8 processes). For the figures, the y-axis range is fixed. It can be seen that there is a diminishing tendency for saturation towards larger NPU arrays, which was already visible in the data for random placement (see Figure 11).

Looking at the relative difference between the heuristic and simulated annealing on the one hand and the random placement on the other hand, it can be seen that there is an distinct minimum in both results at array sizes specific to the task count considered. Figures 16 and 17 show this for 20 and 70 tasks respectively (data shown for 8 processes). The relative minimum for 20 tasks is at 4×4 arrays while it is at 49 and 64 NPUs for 70 tasks. It also becomes clear that the specificity of the minimum diminishes for higher task counts. More specific, while simulated annealing can get down to about 50% of the objective function values of random placement for 20 tasks, it can only accomplish little below 70% of it for 70 tasks. At the same time, the results for

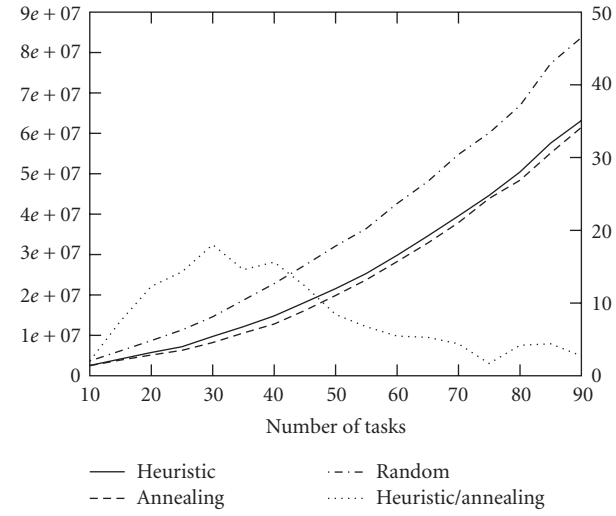


Figure 13: Objective function values (left y-axis)on a 6×6 NPU array and ratio in percent (right y-axis).

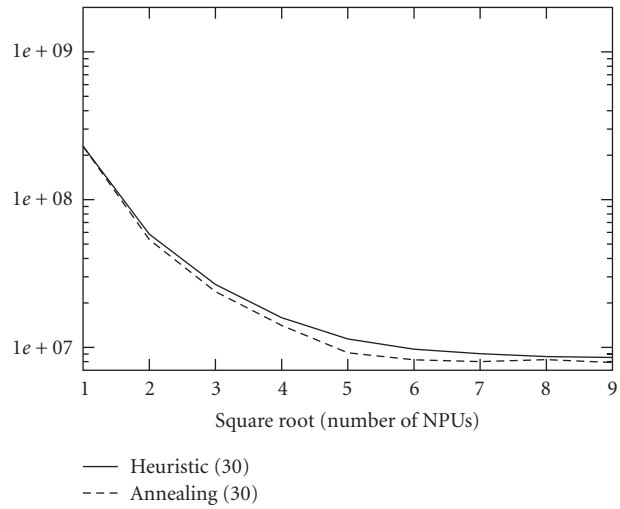


Figure 14: Objective function values for a fixed task count (30) and different array sizes.

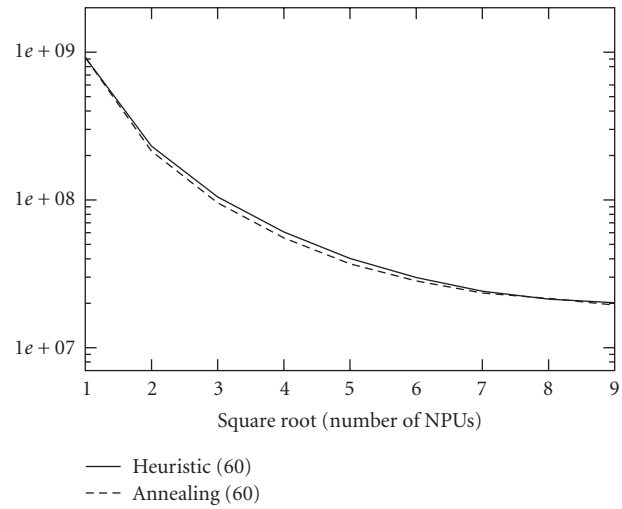


Figure 15: Objective function values for a fixed task count (60) and different array sizes.

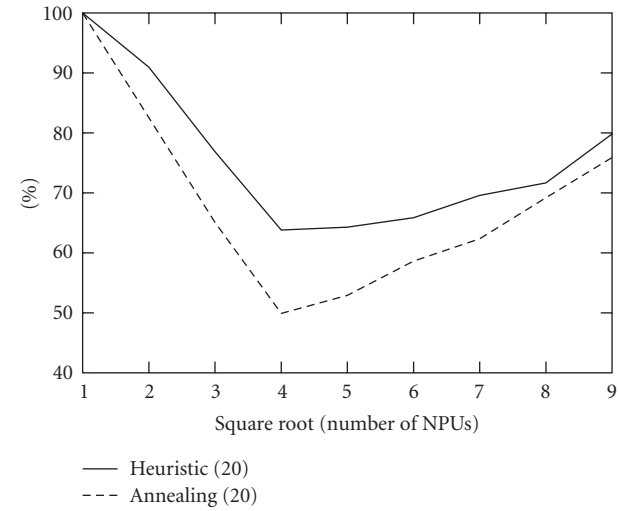


Figure 16: Relative quality of a fixed task count (20) and different array sizes.

the heuristic get closer to that of simulated annealing, already apparent in Figures 12 and 13. Table 6 gives an overview on the minima found for selected task counts. It can be seen that for higher task counts, the heuristic tends to require larger processor arrays than simulated annealing to accomplish its best results.

Finally, it is interesting to look at a specific placement of tasks produced by the heuristic and the simulated annealing algorithm, to see the basic differences. Figure 18 shows the task mapping of the task graph from Figure 10 on a 5×5 array as produced by the heuristic. The three processes are composed of tasks 2–7, 10–14, and 17–19, respectively, with the first and last tasks of each process being input and output tasks. It becomes obvious that the placement produced by the heuristic is limited by the initial distribution of the input and output tasks. In two cases, NPUs 5 and 24, two tasks share the same processor. Apart from this, all other tasks could be placed on their own NPU, thus distributing their workload evenly. The same holds for the simulated annealing result where no processor sharing occurs. It can be seen that the processes are better clustered by the simulated annealing algorithm, while the far-apart input and output tasks as initially placed by the heuristic disrupt a close clustering. Nevertheless, the overall result of the heuristic (OF = 4,616,314) is only 31% worse than that of simulated annealing (OF = 3,516,371) which is quite a good result in the light of the missing global information for the heuristic.

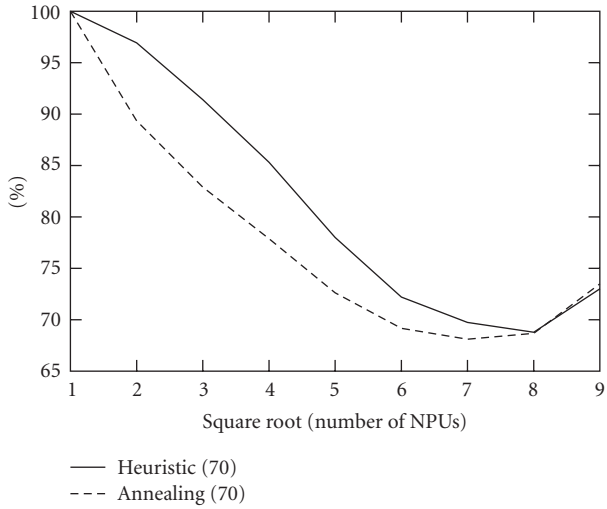


Figure 17: Relative quality of a fixed task count (70) and different array sizes.

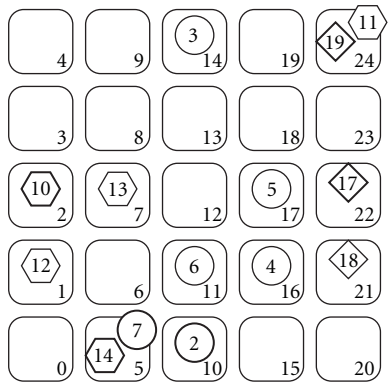


Figure 18: Heuristic placement of task graph from Figure 10 on a 5×5 array.

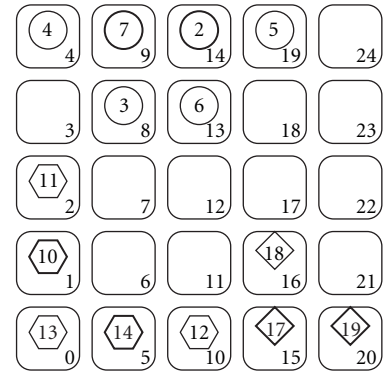


Figure 19: Annealing placement of task graph from Figure 10 on a 5×5 array.

6. Conclusion

This paper describes a distributed task mapping heuristic for homogeneous MPNoCs, derived from a mathematical model. It is based on an initial placement of tasks and a distributed improvement strategy locally implemented on the processing elements. Task sets belonging to different initial applications can be handled as well as tasks added during system operation. For the mapping improvement, only local information available at the affected NPUs and its close vicinity is used, thus avoiding additional communication overhead. Also, the low computational load of the algorithm itself makes its application very attractive.

Running the heuristic for a selected set of example applications shows the good results of the heuristic compared to the exact solution. The accuracy of the results is supported by a system simulation of the VHDL hardware model. For larger array sizes, the heuristic was compared to a simulated annealing algorithm and random placement. It can be seen from the obtained data, that for larger process counts not only do the achieved results of the heuristic come closer to those of simulated annealing—in fact, for large array sizes and high task counts they are even better in some cases—but also the difference of both towards the random placement results get much better. This means, the heuristic delivers increasingly better results for increasing process and task counts. Thus, the heuristic appears to be well suited for future challenges. In summary, the combination of the constructive and the distributed improvement algorithms in the final system appears as a promising decision eliminating many potential scaling problems.

The presented algorithm implementation is a first approach to the problem of efficiently using homogeneous multiprocessor NoC platforms with a large number of processors. Dynamic workloads pose a heavy problem on such systems, for example, because task migration costs will not be negligible any more and must be included into the optimisation algorithms. We believe that the answer to this challenge can only be a scalable solution (like that presented in the paper) which is mainly based on distributed algorithms, using only local information, like the one presented. There is a large design space waiting to be discovered, for example, looking at biologically -inspired algorithms that have proved to be very successful already in nature.

References

[1] L. Benini and D. Bertozzi, "Network-on-chip architectures and design methods," *IEE Proceedings: Computers and Digital Techniques*, vol. 152, no. 2, pp. 261–272, 2005.

[2] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06)*, vol. 1, pp. 1–6, Munich, Germany, March 2006.

[3] T. D. Braun, H. J. Siegel, N. Beck, et al., "A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems," in *Proceedings of the 8th Heterogeneous Computing Workshop (HCW '99)*, pp. 15–29, San Juan, Puerto Rico, April 1999.

[4] E. Carvalho, N. Calazans, and F. Moraes, "Congestion-aware task mapping in NoC-based MPSoCs with dynamic workload," in *Proceedings of IEEE Computer Society Annual*

Symposium on VLSI (ISVLSI '07), pp. 459–460, Porto Alegre, Brazil, March 2007.

[5] J. Chakrapani and J. Skorin-Kapov, "Mapping tasks to processors to minimize communication time in a multiprocessor system," in *The Impact of Emerging Technologies of Computer Science and Operations Research*, pp. 45–64, Kluwer Academic Publishers, Boston, Mass, USA, 1995.

[6] W. W. Chu, L. J. Holloway, M.-T. Lan, and K. Efe, "Task allocation in distributed data processing," *Computer*, vol. 13, pp. 57–69, 1980.

[7] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *Computer*, vol. 15, no. 6, pp. 50–56, 1982.

[8] F. Glover, M. Laguna, and R. Martí, "Fundamentals of scatter search and path relinking," *Control and Cybernetics*, vol. 29, no. 3, pp. 653–684, 2000.

[9] J. Henkel, W. Wolf, and S. Chakradhar, "On-chip networks: a scalable, communication-centric embedded system design paradigm," in *Proceedings of the 17th IEEE International Conference on VLSI Design*, pp. 845–851, Mumbai, India, January 2004.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, Calif, USA, 2003.

[11] F. S. Hillier and G. J. Lieberman, *Introduction to Operations Research*, McGraw-Hill, Boston, Mass, USA, 7th edition, 2001.

[12] B. Hong and V. K. Prasanna, "Performance optimization of a de-centralized task allocation protocol via bandwidth and buffer management," in *Proceedings of the 2nd International Workshop on Challenges of Large Applications in Distributed Environments (CLADE '04)*, pp. 108–117, Honolulu, Hawaii, USA, June 2004.

[13] J. Hu and R. Marculescu, "Energy- and performance-aware mapping for regular NoC architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 4, pp. 551–562, 2005.

[14] J. A. Keane, A. J. Grant, and M. Q. Xu, "Comparing distributed memory and virtual shared memory parallel programming models," *Future Generation Computer Systems*, vol. 11, no. 2, pp. 233–243, 1995.

[15] F.-T. Lin and C.-C. Hsu, "Task assignment scheduling by simulated annealing," in *Proceedings of the 10th Conference on Computer and Communication Systems*, pp. 279–283, Hong Kong, September 1990.

[16] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Transactions on Computers*, vol. 37, no. 11, pp. 1384–1397, 1988.

[17] C. Marcon, A. Borin, A. Susin, L. Carro, and F. Wagner, "Time and energy efficient mapping of embedded applications onto NoCs," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '05)*, vol. 1, pp. 33–38, 2005.

[18] A. Ngouanga, G. Sassatelli, L. Torres, T. Gil, A. Soares, and A. Susin, "A contextual resources use: a proof of concept through the APACHES' platform," in *Proceedings of IEEE Design and Diagnostics of Electronic Circuits and Systems*, pp. 42–47, Prague, Czech Republic, April 2006.

[19] V. Nollet, T. Marescaux, P. Avasare, D. Verkest, and J.-Y. Mignolet, "Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*, vol. 1, pp. 234–239, Munich, Germany, March 2005.

[20] J. M. Orduna, F. Silla, and J. Duato, "A new task mapping technique for communication-aware scheduling strategies," in *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW '01)*, pp. 349–354, 2001.

[21] K. Park, "A heuristic approach to task assignment optimization in distributed systems," in *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, vol. 2, pp. 1838–1842, Orlando, Fla, USA, October 1997.

[22] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, Morgan Kaufmann, San Francisco, Calif, USA, 2004.

[23] S. Sahni and T. Gonzalez, "P-complete approximation problems," *Journal of the Association for Computing Machinery*, vol. 23, no. 3, pp. 555–565, 1976.

[24] N. Saint-Jean, G. Sassatelli, P. Benoit, L. Torres, and M. Robert, "HS-Scale: a hardware-software scalable mpsoc architecture for embedded systems," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07)*, pp. 21–28, Porto Alegre, Brazil, March 2007.

[25] R. Varadarajan, "An efficient approximation algorithm for load balancing with resource migration in distributed systems," Tech. Rep., 1992, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.6072.

[26] P. Yang and F. Catthoor, "Dynamic mapping and ordering tasks of embedded real-time systems on multiprocessor platforms," in *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES '04)*, pp. 167–181, 2004.