This is the author's version of the work. The definitive work will be published in *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip (NOCS 2011), 2011.* This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89)

Dynamic Decentralized Mapping of Tree-Structured Applications on NoC Architectures

Andreas Weichslgartner, Stefan Wildermann, Jürgen Teich
University of Erlangen-Nuremberg, Germany
{andreas.weichslgartner,stefan.wildermann,teich}@cs.fau.de

ABSTRACT

This paper presents a novel application-driven and resource-aware mapping methodology for tree-structured streaming applications onto NoCs. This includes strategies for mapping the source of streaming applications (seed point selection), as well as embedding strategies so that each process autonomously embeds its own succeeding tasks. The proposed embedding strategies only consider the local view of neighboring cells on the NoC which allows to significantly reduce computation and monitoring overhead. Our vision is that this approach facilitates self-organizing embedded systems that provide the flexibility and fault-tolerance required in future silicon technologies. The results provided in this paper show that our local and decentralized algorithms can compete with previously presented global and centralized algorithms.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Algorithms

Keywords

Networks-on-Chip, graph embedding, decentralized mapping

1. INTRODUCTION

Embedded systems used to be devices with limited and often static functionality. For systems with such characteristics, offline design flows are able to explore and synthesize design options that are optimized for specified design goals. It is, however, necessary to rethink this approach due to two main reasons.

©ACM, 2011. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version will be published in *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip,* (May 1-4, 2011 Pittsburgh, PA, USA) .

The first reason is that technology complexity rapidly increases. Soon there will be billions of transistors available on one chip. This requires the need to provide new communication interfaces that scale with the technology. A comparison between (segmented) on-chip buses, which are traditionally applied in heterogeneous multi-processor platforms, and networks-on-chips (NoCs) clearly reveal the scalability advantages of the NoC-approach. [1] analytically derives that with growing number of communicating elements, the clock can be kept constant on a NoC. Area and power dissipation only scale linearly, which is far better than for bus-based communication. Besides the scalability issue, the increase of technology brings further design challenges: due to miniaturization, the number of defects may increase. It is predictable that there will be chips on the market that are not free of defects. This again means that it is nearly impossible to apply offline optimization algorithms that result in a static mapping of functional blocks onto the NoC resources.

The second reason is that embedded applications are getting more and more dynamic. This can especially be observed when looking at the boom of embedded devices which are generally regarded as "smart". Their functionality can be controlled and reorganized by internal mechanisms or by external requests of a user. A "smart phone", for example, might not only be in different operational modes depending on the radio link quality or the user's input. It also runs several applications for entertainment, multimedia or telecommunication and has to support multiple video and audio codecs.

Summarizing these observations, it is necessary to provide novel mapping methods for NoC architectures which are scalable and able to deal with defect processing elements and links, as well as the dynamics of applications. Task and communication mapping onto NoCs with limited processor and link capacities can, however, be reduced to a *bin-packing problem* [2] which is NP-complete. This means that only heuristics are feasible to perform task mapping on larger NoCs. Many related run-time mapping approaches rely on the use of a centralized manager, e.g., [3] [5]. Of course, this centralized approach won't scale and may result in a unsatisfactory performance on a chip with thousands of processors since all the information about the processors and links have to be stored central or gathered in every mapping step. Other work proposes the use of agents which can perform task mapping in a decentralized manner.

In this work, however, we propose a different approach where each application may initiate and perform mapping autonomously. The proposed approach suggests that each

process embeds its own succeeding tasks with data-dependencies, only considering the local view of neighboring nodes in the NoC. This can be done in parallel in contrast to a centralized approach which has to map the tasks sequentially. Especially for streaming applications and applications with a huge amount of periodical communication data, not only the mapping of the functionality, but also of the communication onto the NoC channels is important. Congestions has to be avoided and short paths between the communicating tasks lead to a reduced delay. In this concept, we present several incarnations of an autonomous embedding algorithm, each tailored for the optimization of such objectives. Our vision is that this approach facilitates self-organizing embedded systems that provide the flexibility and fault-tolerance required in future nanoelectronic NoC architecture. The results provided in this paper show that our local algorithms can compete with previously presented algorithms with global knowledge.

2. RELATED WORK

There are several known techniques for dynamic task mapping onto NoCs. Carvalho et al. [3] present several heuristics for dynamic mapping that avoid congestion and reduce the overall network load. They compare simple heuristics without any cost evaluation, such as *First Free* (FF) and *Nearest Neighbor* (NN), with other heuristics that take the link occupation in account, such as *Minimum Average Channel Load* (MAC) or *Path Load* (PL). The tasks are mapped and scheduled by a central manager, which doesn't know the actual topology of the application graphs and maps the occurring tasks dynamically. The application model contains nodes for software and hardware tasks, but there is no multitasking supported and the algorithms run on a global view.

In [5], Chou et al. introduce near convex regions for incremental run-time application mapping. With these areas, the inter-node communication could be minimized and the mapping is divided into two steps: first find a suitable near convex region, and then place the tasks within it. This approach possesses a better scalability than any heuristic that does not consider regions for application mapping. Nevertheless, there is still a global manager which stores the complete system utilization. In [4], this work is extended by comprising the user behavior. By doing this and splitting up the pattern in four subproblems the contention and communication costs could be minimized.

Faruque et al. present an agent-based distributed application mapping which reduces the monitoring traffic comparing to centralized approaches [9]. Agents are small tasks which can be executed on any node in the NoC and perform resource management. They act and negotiate with each other to find processing elements suitable for mapping a task. There are two types of agents to accomplish this: Global Agents (GA) and Cluster Agents (CA). The CAs have knowledge about their cluster. When they get a new task request, they negotiate with the GAs, which have global information about all clusters.

In contrast to above works, our self-embedding approach as proposed in this paper is fully decentralized and autonomous. There is no central manager like in [2], and no agents have to run on several tiles like in [9]. Every NoC unit is considered equal and contributes to the embedding.

Besides the central approaches there is some previous work

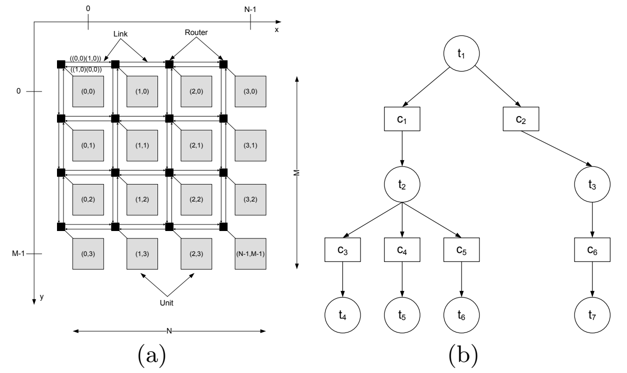


Figure 1: (a) NoC architecture and (b) application tree.

on dynamic and decentralized mapping:

In [12], Li presents a random-walk-based dynamic tree evolution algorithm. Tasks dynamically spawn new tasks. The tree size and further information about the evolution of the tree are not know a priori. The new tasks are embedded totally distributed by a random-walk algorithm. The optimization goal in this approach is to minimize the communication dilation and the ratio between maximal and average resource utilization, which expresses the processor load. They furthermore prove that this ratio converges to 1 if the tree size is infinite. The work is based on theoretical assumptions that assume that there are no resource boundaries and all processing elements are homogeneous.

Hosseinabady and Nunez-Yanez [11] propose a decentralized stochastic and fault-tolerant mapping and routing algorithm. Their methodology incrementally maps the tasks and, concurrently, routes their communication by applying a random walk. This random walk is performed by sending so-called task request flits to randomly chosen neighboring routers. They have the purpose to find a tile which can host the task. To improve this random-walk-based algorithm, they propose additional metrics which rely on information that are distributed by broadcast messages. The fault tolerance is achieved by ignoring faulty ports when performing this stochastic routing scheme.

In our approach we also take the link utilization in account and develop a general scheme for decentralized mapping algorithms. It is also possible to handle dynamically created tasks.

3. PRELIMINARIES

Application mapping onto NoCs involves assigning tasks onto processing elements and communication onto communication links. In the following, we give some formal definitions necessary to describe our mapping approach.

3.1 NoC Architecture

In this paper, we will focus on regular NoCs with a mesh-based structure where units $u \in U$ are connected by links $l \in L$. The NoC can be described formally as follows:

DEFINITION 1. *A* NoC *of width N and the height M is a directed graph $G_{NoC}(U, L)$ with*

$$U = \{(x, y) \mid 0 \leq x < N, 0 \leq y < M\} \quad (1)$$

$$L = \{(u_1, u_2) \mid u_1, u_2 \in U \wedge d_m(u_1, u_2) = 1\}, \quad (2)$$

where d_m defines the Manhattan distance between two units. The distance between two units $u_1 = (x_1, y_1)$, $u_2 = (x_2, y_2) \in U$ is calculated according to:

$$d_m(u_1, u_2) = |x_1 - x_2| + |y_1 - y_2|. \quad (3)$$

Each unit consists of one processing element (PE) and one router that is connected to the local PE (through a Network-Interface (NI)) and the four routers in the cardinal directions. Tasks can be loaded and executed on the PEs. Each unit $u \in U$ provides a limited amount of consumable resources that can be occupied by tasks, e.g., memory for storing data and program code. This limit is denoted as $res(u)$. The maximal available bandwidth of a link $l \in L$ is limited to $band(l)$.
An example of a NoC can be seen in Fig. 1(a).

3.2 Application Graphs

The methodology proposed in this paper is tailored for dataflow-dominated streaming applications, with characteristics typically found in multimedia, telecommunication, and signal processing. For the analysis presented in this work, we make following assumptions:

A1: An application i is executed periodically with period P_i.
A2: Data-dependencies between functional blocks result in a tree-shaped dataflow.
A3: Applications are characterized by bandwidth-oriented one-to-one communications between tasks.

A graph-based, formal model of each applications as it is illustrated in Fig. 1(b) can be given as follows:

DEFINITION 2. *An* tree-structured application *i is modeled as an acyclic, bipartite graph $G_A(V_i, E_i)$. The set $V_i = T_i \cup C_i$ is a union of task vertices T_i and communication vertices C_i. Each edge $e \in E_i$ connects a task vertex $t \in T_i$ to a communication vertex $c \in C_i$ or vice versa. Let $pred(v)$ denote the set of predecessors and $succ(v)$ the set of successors of vertex $v \in V_i$. Then it holds that $|pred(t)| \leq 1$, $\forall t \in T_i$ (**A2**), and that $|pred(c)| = |succ(c)| = 1$, $\forall c \in C_i$ (**A3**). An example of such a tree can be found in Fig. 1(b) where circles describe task vertices and rectangles communication vertices.*

Considering the heterogeneity of the NoC, each task $t \in T_i$ is characterized by its execution time $exec(t, u)$ on PE $u \in U$ and the resources required for successfully executing the task on the PE $req(t, u)$. Each message $c \in C$ is designated with its payload $size(c)$. With the application's execution period P_i, it is possible to furthermore calculate the bandwidth requirement of c :

$$bw(c) = \frac{size(c)}{P_i} \quad (4)$$

and the load imposed by task t to unit u:

$$load(t, u) = \frac{exec(t, u)}{P_i} \quad (5)$$

3.3 Mapping

Application mapping can now be described as embedding the application graph onto the NoC graph. This involves a) task mapping and b) communication routing. Task mapping is denoted as $m : T_i \to U$, i.e., assigning tasks to processing elements. Routing is a function

$$\begin{aligned} r : C_i \quad \to \quad & \{R \mid R \in \prod_{i=1}^{n} L, 0 \leq n \leq N + M - 2 \\ & \wedge \; \forall \; i = 1, ..., n-1; l_i = (u, u') \\ & \wedge \; l_{i+1} = (u', u'')\}, \end{aligned}$$

i.e., each communication node is assigned with a route through communication links. For example, a route with n hops can be described as a sequence $r(c_i) = (l_1, l_2 ... l_n)$. A feasible mapping is give, if the following constraints hold:

$$req(t, u) + \sum_{t' \in T : m(t') = u} req(t', u) \leq res(u) \quad (6)$$

$$load(t, u) + \sum_{t' \in T : m(t') = u} load(t', u) \leq load_{max}(u) \quad (7)$$

where 6 ensures that resource restrictions are respected, and 7 is the schedulability test of resource u. Moreover, a feasible routing has to consider the available bandwidth.

$$bw(c) + \sum_{c' \in C : l \in m(c')} bw(c', l) \leq band(l) \quad (8)$$

In the following, we propose decentralized algorithms for application-driven mapping and routing, which we call *self-embedding.*

4. SELF-EMBEDDING ALGORITHM

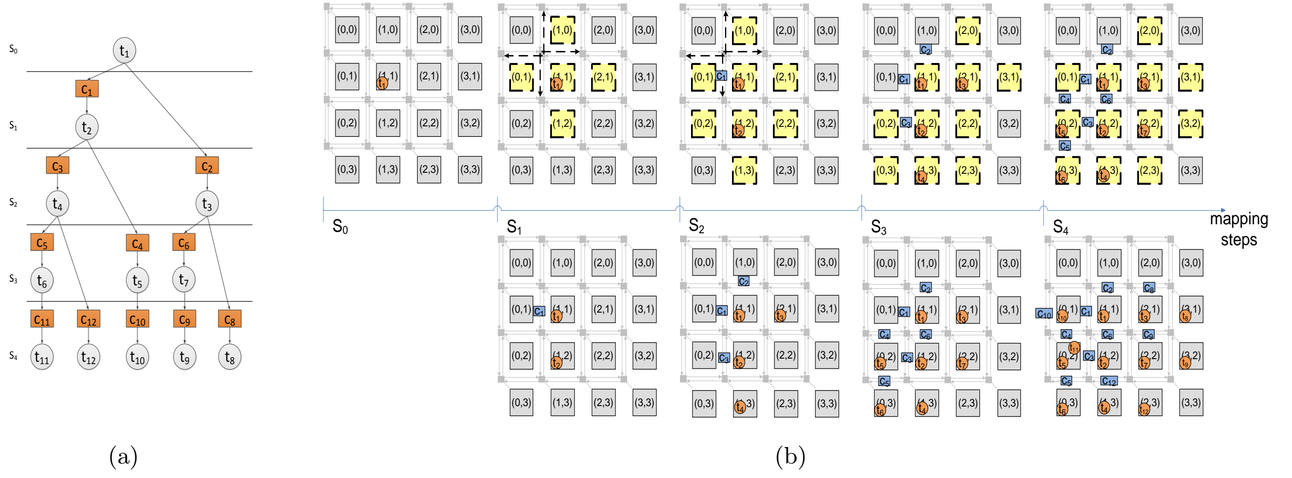
As our inspected application topologies are treestructured, tasks can be embedded incrementally by an already mapped predecessor task. Only the root nodes have to be placed differently, since they have no predecessor tasks. In Section 6, we propose algorithms for this *seed point selection.* Distributing the mapping calculation to the task nodes helps to parallelize the workload and to prevent a single point of failure. Our algorithm is described as follows.

DEFINITION 3. *Given an already placed task t' with mapping $m(t') = u_n$. Then, the incremental* self-embedding algorithm *embAlg searches for a feasible mapping $m(t)$ and $r(c)$ is described as:*

$$embAlg(u_n, t, h, c, z) = \begin{cases} 1, & \text{if Eqs. 6, 7, 8 are fulfilled} \\ 0, & \text{else} \end{cases} \quad (9)$$

- *u_n: current unit ($m(t') = u_n$) that calculates/initiates the embedding algorithm*
- *t: the task that has to be mapped ($t \in succ(c)$)*
- *h: max. search distance in hops defining the size of the search space*
- *c: communication node to map ($c \in succ(t')$)*
- *$z(u, t, R, c)$: cost function which determines the optimization goal by evaluating a mapping option u and routing option R.*
- *$m(t)$: the resulting mapping of t, and*
- *$r(c)$: the route chosen for c.*

This generic formulation allows the use of arbitrary cost functions $z(u, t, R, c)$, even weighted combinations of different optimization goals are possible there. The cost function is applied onto units and links within a search space of a size that is determined by parameter h. It is a matter of



(a) (b)

Figure 2: An example of a self-embedding algorithm with the initial search space of 1 hop. In the upper half the search space is shown and in the lower half the mapping. The root node t_0 is embedded initial with the seedpoint selection in mapping step S_0. In mapping step S_1 it starts the embedding algorithm for t_2 and c_1. In S_2, it starts the embedding for t_3 and c_2. In the same mapping step, t_2 itself can start the embedding of t_4 and c_3, and so on. While embedding the right successor, the already mapped left successor can embed its own successors in parallel.

the implementation of our algorithm-scheme to compute a solution out of the values of the cost function (see Section 5). By means of parameter h, the search space can be limited to units in the neighborhood of the task. This allows to reduce the overhead when a task embeds its successors. Thus, search space can be restricted to a certain area that can range from the direct tile neighbors to all tiles within the NoC. This provides a flexible interface where the difference between a local embedding and a global version is only the matter of the size of the search space. In case no feasible mapping can be found in the local search space, its size can be increased iteratively. An example for this *search space adaption* can be found in Algorithm 1.

Algorithm 1 Pseudocode of search space adaption of the self-embedding algorithm

```
for all t ∈ succ(t') do
    h ← h_init
    repeat
        map ← embAlg(u_n, t, h, c, z);
        h ← h + 1
    until map = 1 ∧ h ≤ h_max
end for
```

Besides the optimization goal, basic constraints have to be considered to provide a feasible mapping. This includes that, for being considered as a mapping candidate, a unit must provide sufficient free resources and the additional work load that would be imposed by mapping the task must not exceed the unit's schedulable utilization. Moreover, units and the links with faults must be ignored.

A motivational example for a decentralized embedding following our scheme is shown in Fig. 2. The figure also illustrates the degree of parallelization achievable with the self-embedding approach. The embedding is performed within 4 mapping steps. In contrast to a sequential embedding which would take 11 mapping steps.

The achievable degree of parallelization is topology-dependent, but we can theoretically derive bounds on how many mapping steps it requires to embed an application. Note that mapping steps may however have varying runtimes. This is due to the fact that processes and monitoring messages might be delayed because of preemption, and the search space adaption might result in different numbers of iterations.

Theorem 1. *For an application with $|T_i|$ tasks, the lower bound on the required mapping steps is $\Omega(\lceil ld(|T_i|)\rceil + 1)$, and the upper bound is $\mathcal{O}(|T_i|)$.*

Proof. The *upper bound* (worst-case) for the parallelization would be a chain of tasks, what means $|pred(t)| \leq 1, |succ(t)| \leq 1,\ \forall t \in T_i$. This is depicted in Fig. 3(a). In this case, every task can only embed one successor and no degree of parallelization is given. The embedding is still distributed, but performed sequentially.

For the *lower bound* (best-case), the application topology would allow that all tasks mapped until step s are able to map a successor in step $s+1$, as illustrated in Fig. 3(b). The bound can be proven by means of induction. The theorem states the relation between number of tasks $|T_i|$ and number of steps s as

$$\lceil log_2(|T_i|)\rceil + 1 = s \tag{10}$$

$$2^{s-2} < |T_i| \leq 2^{s-1}. \tag{11}$$

For $s = 1$, we can only map the root node, i.e.,

$$\lceil log_2(1)\rceil + 1 = 1. \tag{12}$$

For $s \rightarrow s+1$, if the lower bound holds for s, we can map up to $n = 2^{s-1}$ tasks in s steps, as given in Eq. 11. Each of these n tasks can spawn a new successor. This would result in a number of tasks being mapped after $s+1$ steps that

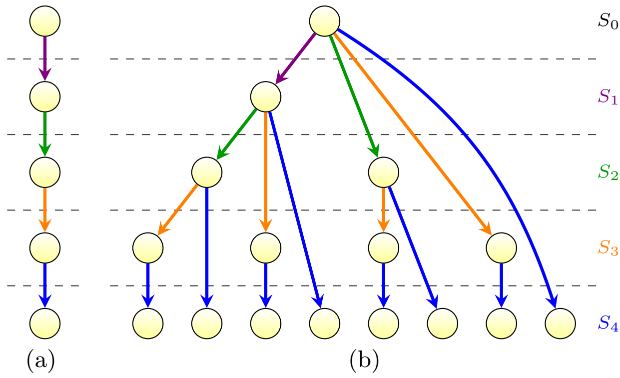


Figure 3: Examples of application topologies leading to worst-case and best-case number of mapping steps. (a) worst-case: tasks are placed sequentially. (b) best-case: all tasks mapped until step S_k map a successor in step S_{k+1}.

lies between $n+1$ and $n+n$, if only one or if all previously mapped tasks spawn a successor, respectively. This leads to

$$\lceil log_2(n+1)\rceil + 1 = \lceil log_2(2^{s-1}+1)\rceil + 1 = s+1 \qquad (13)$$

and, respectively,

$$\lceil log_2(n+n)\rceil + 1 = \lceil log_2(2) + log_2(n)\rceil + 1 = s+1 \qquad (14)$$

□

5. INCARNATIONS OF EMBEDDING ALGORITHMS

According to the generic interface of the embedding algorithm described in Section 4, there are various alternatives to implement self-embedding. This mainly affects how to select a unit from the search space to map the task, and how to perform the routing of the communication between the mapper task and its successor. The implementations range from deterministic to probabilistic approaches. Algorithms can work with a predefined fixed routing scheme, like xy-routing, and calculate the cost function according to this scheme. On the other hand, the routing scheme can be part of the embedding itself. In the following, we present one algorithm with a fixed routing scheme and one with an adaptive routing scheme.

5.1 Path Load and Best Neighbor

In [3], the *Path Load* (PL) is proposed as a goal function to avoid congestion and to minimize the average link occupation. Given the mapper task t', a successor t, and the communication c, $(t',c),(c,t) \in E$. The cost function for a mapping $m(t)$ with routing $r(t)$ is given as:

$$pathLoad(m(t), t, r(c), c) = \sum_{l \in r(c)} rate_{link}(l) \qquad (15)$$

where the already consumed bandwidth of a link l is derived by considering the bandwidth imposed by all messages being routed over link l:

$$rate_{link}(l) = \sum_{\substack{c \in C \\ \exists i: l = r(c)_i}} bw(c). \qquad (16)$$

The algorithm in [3] works by considering each mapping option $m(t) \in U$ with routes calculated by a fixed routing scheme, e.g., xy-routing. Evaluating every single tile in the NoC results in a search space of $h = (N-1)+(M-1)$ hops. The number of inspected units is $N \times M$ and the number of inspected links is $(N \times M) \times 2 - (M+N)$. It is obvious that for large NoCs, the amount of monitoring overhead would be enormous. If a suitable mapping option is already found, then it is not necessarily required to continue the search on the remaining units. Equation 15 makes clear that the path load will only get worse when adding links to a route. The observation also holds for other network metrics, such as *Mininmal Average Channel Load* (MAC) and *Minimum Maximum Channel Load* (MMC) as proposed in [3]. This lead to our proposal of restricting the search to a locality and applying bounds on the search area.

This restriction limits the units considered to the set U_h, containing all units with a distance of h hops to the current unit u_n, and to the set of links L_h.

$$U_h = \{u \mid d_m(u_n, u) \leq h\} \qquad (17)$$
$$L_h = \{(u_1, u_2) \mid u_1, u_2 \in U_h \wedge d_m(u_1, u_2) = 1\} \qquad (18)$$

If the algorithm ran without success, the search range can be increased by applying Algorithm 1. Independent of the applied goal function, this incarnation can be generalized as *best-neighbor* (BN) algorithm: All units and links in a certain area are inspected and the best unit and routing is chosen according to the cost function. In our path load example, the cost function is communication-centric and the sum of the link occupations. But also a task-oriented goal function, e.g., considering the resource occupation, could be used in the same manner.

5.2 Random Walk

One major disadvantage of the path load algorithm is the fixed routing scheme (here xy-routing). Even though xy-routing establishes routes with minimal hops and is easy to implement, it can't deal with faulty links or routers. The random walk-based approach for task embedding in [12] can help here. The *random walk* (RW) can be performed on each unit. When considering to map a task, a unit either keeps a newly spawned task or sends it to a neighbor randomly chosen with uniform probability. We modify this theoretical investigation to comply with our self-embedding approach. Here, each unit that holds an unmapped task sends a request to a non-faulty neighbor which is randomly selected. This neighbor does the same, however, the unit from which it gets the request is excluded to prevent cycles. This procedure is repeated until the random walk has been performed h times. The found tiles and links are then evaluated with the cost function. This can be implemented by rolling back the route taken by the random walk, evaluating the cost function and keeping the best unit found so far. When returning to the initial unit, the best fitting unit of the random walk is available. Moreover, the route taken by the random walk is used for the communication between the tasks, thus enabling fault-tolerant routing.

A modification of this algorithm is to apply two cost functions: besides the cost function that evaluates a mapping option $m(t)$, we weight the probabilities to chose the next neighbor for the random walk. We call this algorithm *weighted random walk* (RWW). To calculate these weights, a cost function is needed. For example, one goal could be a low

channel load and the other goal a low resource occupation. So, a neighboring cell which is connected via a link with less occupation should be chosen with a higher probability than a cell with a high congestion. Especially if knowledge about neighboring links is available in the routers, this approach is promising. If random walk was run without success and no suitable unit was found, then the algorithm can be started again. Because of the non-deterministic behavior, it can return different units.

5.3 Discussion

The two presented incarnations of embedding algorithms have different behavior. While the best-neighbor algorithm searches every unit in a certain area within it choses the best mapping option. The random walk approach choses a random set of units as search space with only one unit per hop distance. Projecting this into the search domain, BN performs a breadth-first search while the random-walk represents a depth-first search. The BN algorithm with the fixed xy-routing scheme can't provide fault tolerance. In contrast, the random walk delivers an adaptive route and guarantees fault-tolerance. Another aspect are the costs of an hardware support. While BN compares only values of cost functions and maybe sorts them, the RWW algorithm needs at least pseudo-random numbers and arithmetical support to calculate the weighted probabilities.

6. SEED POINT SELECTION

As every task places its succeeding tasks and communication, the initial task or root task has to be placed by a different algorithm. For these tasks, it is necessary to find those units on the NoC on which it is suitable to load the root nodes. We call these units *seed points* (U_s) and several methodologies are available to determine this set. As we don't want to loose the decentralized characteristics of our approach, the seed point determination should not have a global view of the entire NoC system. Nevertheless, some global information, like the size of NoC, previous seed points and cluster information have to be stored centrally. However, these algorithms are only executed once per application and the saved information is linear in the NoC size. So the scalability is kept. An example of the three proposed seed point selections strategies can be seen in Fig. 4.

If the number of applications is know a priori or they are mapped concurrently, a clustering algorithm like *k-means* can be suitable given that the applications and their tasks have roughly the same characteristics. With an effective implementation like in [10], centers of regions with similar sizes can be found. Global knowledge is needed for this procedure, but not the status of every unit and link.

If the applications spawn dynamically, another approach from the clustering area can be used. The Hochbaum-Shmoys or *Farthest-Away* algorithm [6] searches for the farthest point away from the already found seed points (U_s). The next seed point u_1 is calculated as below:

$$u_1 = \underset{u \in U}{argmax}(\underset{u_s \in U_s}{min}(d_m(u, u_s))) \quad (19)$$

If the applications are dynamically spawned and have different requirements for resources the approach of *near convex regions* introduced by Chou and Marculescu [5] offers a viable approach. The regions are calculated with the help of a dispersion factor and a centrifugal factor. The first one describes how many idle neighbors a unit within a region should have, the second how far a unit should be away from the current region border. Both together help to prevent isolated units and to keep the regions compact. To use this algorithm for determining the seed point for an application, the near convex region of the application is calculated and the center is chosen as seed point. This calculation has to be executed by an central manager or agents. In our implementation, this calculation only has to save which units were already used for previous near convex regions. A knowledge about the entire system status is not needed if this algorithm is only used to determine the seed.

7. EXPERIMENTAL RESULTS

In this section, we present the test environment and show the experimental results in comparison to already existing task embedding algorithms.

7.1 Simulation Setup

To evaluate the proposed algorithms, we implemented a Java NoC simulator. There the units and links are modeled to match the definitions in Section 3, see also Fig. 4 for some screenshots. As seed point selection the Farthest Away algorithm is used.

To test our algorithms, we use three different classes of trees.

- *Chains*: These trees can typically be found in signal processing, where a stream of signals is processed sequentially in a filter chain.
- *Binary trees* : These kind of applications are often used to solve decision-problems.
- *Quadtrees*: Quadtrees are often used to decompose a picture in separate regions to de- and encode with different granularity. Such a coder is e.g. the QSDPCM codec [13].

We use TGFF [8] format and the e3s Benchmark-Suite [7] for our test applications. Several of these applications are embedded iteratively to simulated the dynamically occurrence of new applications. 500 runs of each testcase are made.

7.2 Evaluation Metrics

For the remainder of this paper, additional *evaluation metrics* are required to measure the success of an embedding and to compare different algorithms. An important metric is the *average network load*. It directly influences the energy consumption of the system and indirectly reflects the length of found communication paths.

$$avg_{net} = \frac{\sum rate_{link}(l)}{|L|} \; \forall \; l \in L \quad (20)$$

Another important measurement is the *number of unsuccessful runs* of the embedding algorithm. If the algorithm returns not a valid unit or a valid route, it has to be re-run again with adapted parameters. It might even be necessary to execute a different algorithm. The number of fails ($\#Fails$) quantifies the practicability of an embedding algorithm as well as the parameter set.

To measure the overhead which is caused by the embedding, those *monitoring messages* are counted that are exchanged between the PEs to accomplish the mapping. If a unit needs information about the neighboring units and links, then we assume that this information is acquired by

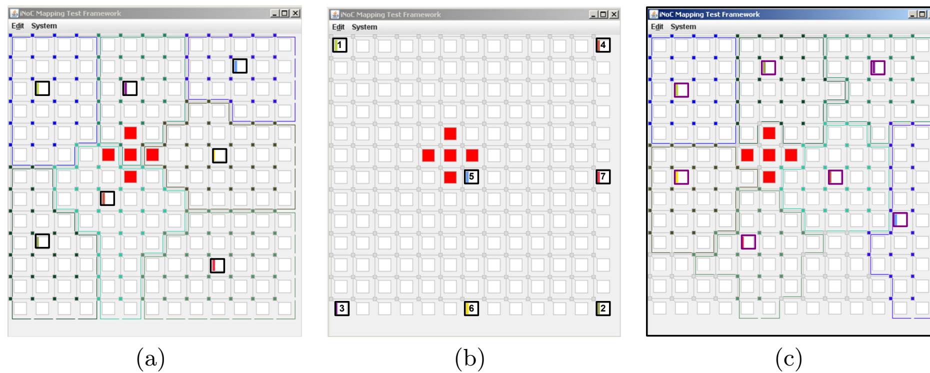


Figure 4: Seed point selection according to (a) k-means, (b) farthest away, (c) near convex region

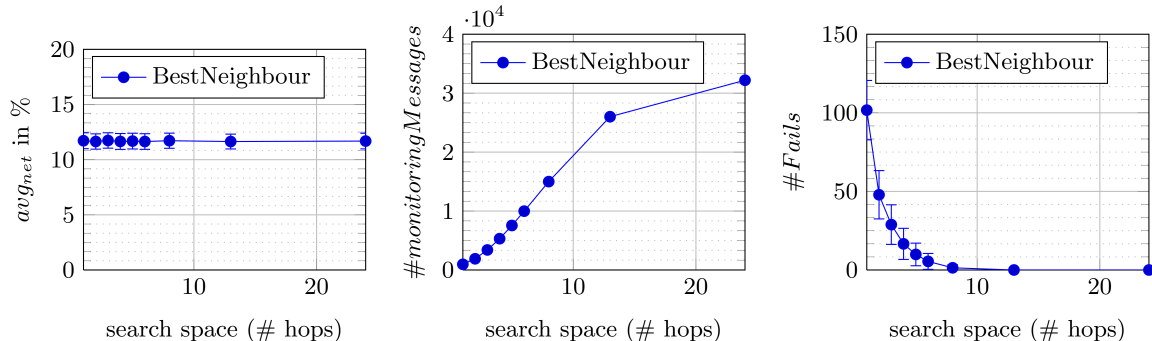


Figure 5: Best Neighbor with different search space

exchanging messages. Also loading new tasks onto processing units requires sending the configuration via messages. Since they don't belong to the application-specific traffic, they need also to be counted as monitoring messages.

7.3 Scalability

In Fig. 5, our best neighbor algorithm is evaluated with different size of the search space. Here the search space ranges from 1 (only the four cardinal neighbors) to 24 (the whole NoC). The algorithm with 24 hops search space is equal to the global path load algorithm from [3]. It points out that avg_{net} doesn't improve with a bigger search space while the monitoring overhead rises rapidly. The number of fails drops with a further search space increase, but converges to zero before the whole search space is reached, for h=13.

In Fig. 6, the experiments is repeated for NoCs of different sizes (here N=M=NoC Width). To keep the resource occupation similar the number of application is also raised (7, 8, 9, 10, 11). It is obvious that the BN Algorithm with global search scales worse as can be seen from the huge amount of $monitoringMessages$ while both produce similar avg_{net}.

7.4 Random-Walk with weighted Probability

In Fig. 7, we compare the average network load between a random-walk (RW) that uses equal probabilities to perform the random walk and a random walk with weighted probabilities (RWW) as we introduced in Section 5.2. We use a 13×13 NoC configuration with 5 faulty units in the center (see Fig. 4). The seed point selection is for both test cases the same It shows that RWW provides a better overall network utilization and also reduces the number of failed runs. It is obvious, that the number of fails increases comparing quadtrees with binary tree and binary trees with chains, because in a chain only one successor has to be embedded and so one feasible unit is needed. On the opposite for a quadtree four units are needed. The probability that four available units are in the neighborhood is less than only one available unit.

8. CONCLUSION

In this paper we have described a novel approach to map tree-structured streaming applications onto NoCs. To prove the power of decentralized self-embedding, we have presented a model and interface for this kind of algorithm and also have introduced two incarnations of these algorithms. The local Best Neighbor algorithm can compete with the global Path Load algorithm by offering a magnitude less monitor overhead. Also we have developed an improved random-walk based algorithm that offers several advantages against a pure random walk algorithm. In our future work, we want to extend our algorithms to different structured applications so that it is possible to map graphs where a task has more than one predecessor. We also want to implement the self-embedding as a part of hardware router into a NoC.

Acknowledgement

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

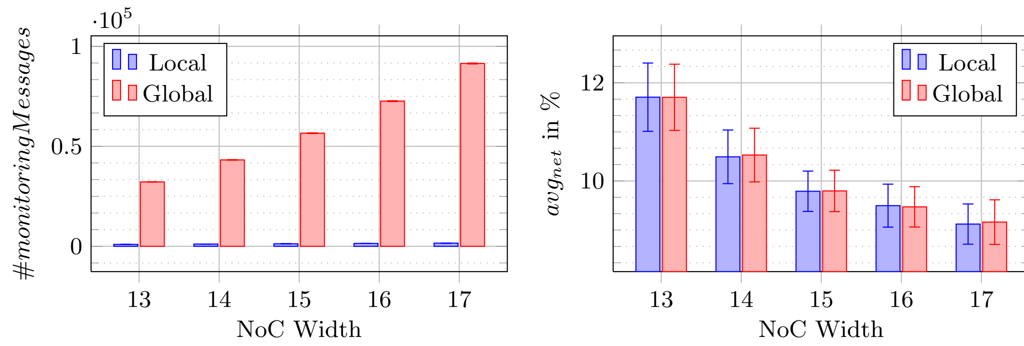


Figure 6: Increasing of the NoC Width

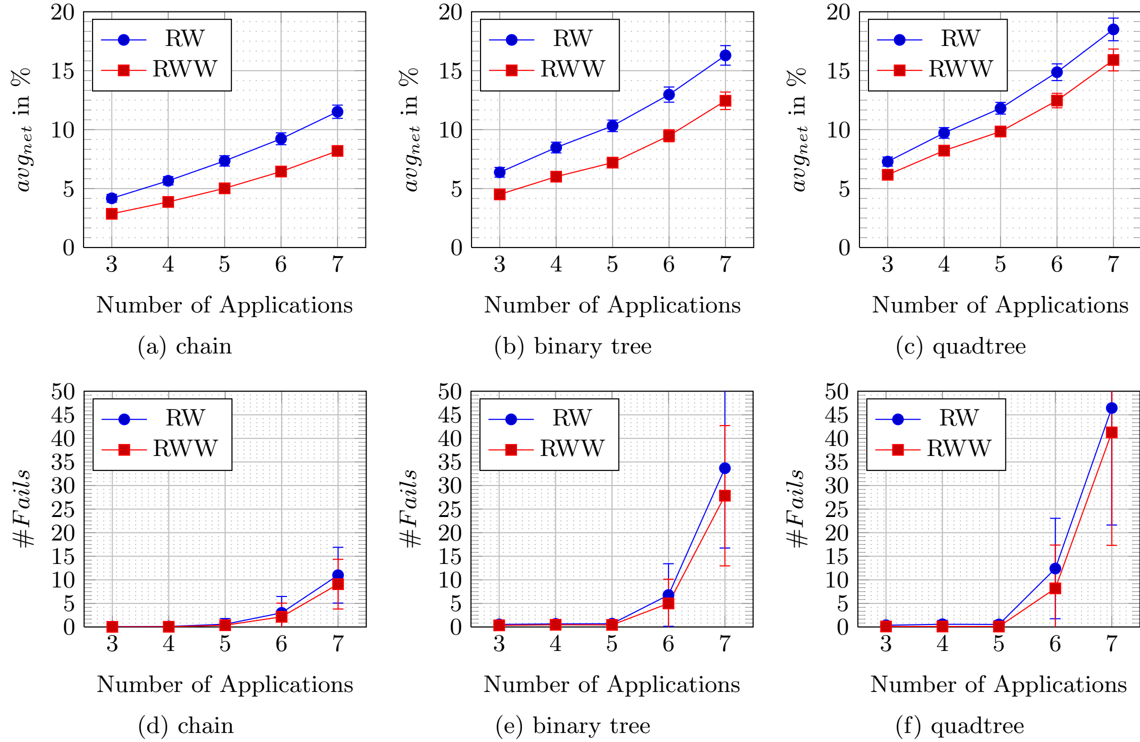


Figure 7: Comparison between random walk and weighted random walk

9. REFERENCES

[1] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. Cost considerations in network on chip. *Integr. VLSI J.*, 38:19–42, Oct. 2004.

[2] E. W. Brião, D. Barcelos, and F. R. Wagner. Dynamic task allocation strategies in MPSoC for soft real-time applications. In *Proc. of DATE '08*, pages 1386–1389, 2008.

[3] E. Carvalho, N. Calazans, and F. Moraes. Heuristics for Dynamic Task Mapping in NoC-based Heterogeneous MPSoCs. In *Proc. of RSP '07*, pages 34 –40, May 2007.

[4] C. Chou and R. Marculescu. Run-time task allocation considering user behavior in embedded multiprocessor networks-on-chip. *IEEE Trans. ofComputer-Aided Design of Integrated Circuits and Systems*, 29(1):78–91, 2009.

[5] C.-L. Chou, U. Ogras, and R. Marculescu. Energy- and Performance-Aware Incremental Mapping for Networks on Chip With Multiple Voltage Levels. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1866–1879, Oct. 2008.

[6] S. Dasgupta. Performance guarantees for hierarchical clustering. In *Computational Learning Theory*, pages 235–254. Springer, 2002.

[7] R. Dick. Embedded System Synthesis Benchmarks Suite. http://ziyang.eecs.umich.edu/ dickrp/e3s/, 2010.

[8] R. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *Proc. of CODES/CASHE '98*, pages 97–101, 1998.

[9] M. A. A. Faruque, R. Krist, and J. Henkel. ADAM: run-time agent-based distributed application mapping for on-chip communication. In *Proc. of DAC '08*, pages 760–765, 2008.

[10] G. Frahling and C. Sohler. A fast k-means implementation using coresets. In *Proc. of SCG '06*, pages 135–143, 2006.

[11] M. Hosseinabady and J. Nunez-Yanez. Run-time resource management in fault-tolerant network on reconfigurable chips. In *Proc. of FPL '09*, pages 574–577, Sept. 2009.

[12] K. Li. A random-walk-based dynamic tree evolution algorithm with exponential speed of convergence to optimality on regular networks. In *Proc. of FCST '09*, pages 80–85. IEEE, 2010.

[13] P. Strobach. Tree-structured scene adaptive coder. *IEEE Trans. on Communications*, 38(4):477 –486, Apr. 1990.