Exploiting Resource-constrained Parallelism in Hard Real-Time Streaming Applications

Jelena Spasic, Di Liu and Todor Stefanov
Leiden Institute of Advanced Computer Science
Leiden University, Leiden, The Netherlands
Email: {j.spasic, d.liu, t.p.stefanov}@liacs.leidenuniv.nl

***Abstract*—In this paper, we study the problem of exploiting parallelism when a hard real-time streaming application modeled as a Synchronous Data Flow (SDF) graph is mapped onto a Multi-Processor System-on-Chip (MPSoC) platform. We propose a new unfolding graph transformation and an algorithm that adapts the parallelism in the application according to the resources in an MPSoC by using the unfolding transformation. We evaluate the efficiency of our unfolding graph transformation and the performance and time complexity of our algorithm in comparison to the existing approaches. Experiments on a set of real-life streaming applications demonstrate that: 1) our unfolding transformation gives shorter latency and smaller buffer sizes when compared to the related approaches; and 2) our algorithm finds a solution with smaller code size, smaller buffer sizes and shorter latency in 98% of the experiments, while meeting the same performance and timing requirements when compared to an existing approach.**

I. Introduction

Modern Multi-Processor System-on-Chip (MPSoC) platforms offer high processing power through many processing elements available on a chip. At the same time, modern streaming applications have high computational requirements and hard real-time constraints. To meet the computational demands and timing requirements of these modern streaming applications, the parallel processing power of MPSoC platforms has to be exploited efficiently. Exploiting the available parallelism in an MPSoC platform to guarantee performance and timing constraints is a challenging task. This is because it requires the designer to expose the right amount of parallelism available in the application and to decide how to allocate and schedule the tasks of the application on the available processing elements such that the platform is utilized efficiently and the timing constraints are met. Several parallel Models-of-Computation (MoCs), e.g. Synchronous Data Flow (SDF) [1] and Cyclo-Static Dataflow (CSDF) [2], have been adopted as the parallel application specification. Within a parallel MoC, an application is represented as a task graph with concurrently executing and communicating tasks. Thus, the parallelism is explicitly specified in the model.

However, the given initial parallel application specification often is not the most suitable one for the given MPSoC platform. This is because application developers mainly focus on realizing certain application behavior while the computational capacity of the MPSoC platform is often not fully taken into account. To better utilize the underlying MPSoC platform, the initial specification of an application, i.e., the initial task graph, should be transformed to an alternative one that exposes more parallelism while preserving the same application behavior. This can be achieved through an unfolding transformation where the tasks from the initial graph are replicated in the equivalent graph a certain number of times, determined by *unfolding factors*. In such a way, replicas corresponding to a task in the initial graph process different data, thereby exploiting data-level parallelism. The unfolding graph transformations proposed so far: 1) introduce additional tasks for managing data among replicas [3], [4], which introduces communication and scheduling overhead; 2) do data reordering or increase rates of data production/consumption on channels [5], [6], which causes an increase of buffer sizes of data communication channels between the tasks and an increase of the application latency. Thus, special care should be taken during the unfolding transformation to avoid all the unnecessary overheads. Moreover, having more tasks' replicas than necessary results in an inefficient system due to overheads in code and data memory, scheduling and inter-tasks communication [4], [5]. Thus, the right amount of parallelism (tasks' replicas), i.e., the proper values of unfolding factors, depending on the underlying MPSoC platform, should be determined in a parallel application specification to achieve maximum performance and timing guarantees.

Therefore, in this paper, we address the following problems: (1) How to efficiently unfold a given initial graph of an application to avoid unnecessary communication/scheduling overheads and unnecessary increases in buffer sizes and the application latency?, and (2) How to find a proper unfolding factor of each task in the initial graph, such that the obtained alternative graph exposes the right amount of parallelism that maximizes the utilization of the available processors in an MPSoC platform under hard real-time scheduling? The specific contributions of this paper are the following:

- We propose a new unfolding graph transformation for SDF graphs which results in graphs with shorter application latency and smaller buffer sizes compared to the related approaches [3], [4], [6], [5], as shown in Sec. VII.
- We propose a new algorithm for finding a proper value for the unfolding factor of each task in a graph when mapping the graph on a platform such that the platform is utilized as much as possible under hard real-time scheduling.
- We show, on a set of real-life streaming applications, that in more than 98% of the experiments, our unfolding graph transformation and algorithm result in a solution with a shorter latency, smaller buffer sizes and smaller values for unfolding factors compared to the solution obtained from [5] while the same performance and timing requirements are satisfied.

Scope of work. In this work, we assume that a given SDF graph is acyclic. This limitation comes from the hard real-time scheduling framework [7] we use to schedule an SDF graph. However, even with this limitation our approach is still applicable to many real-life streaming applications because a recent work [8] has shown that around 90% of streaming applications can be modeled as acyclic SDF graphs. In addition, our approach does not unfold *stateful* tasks and input/output tasks. A *stateful* task is a task which current execution depends on its previous execution, thus those executions cannot be run in parallel. Input and output tasks are the tasks connected to the environment, hence they are not unfolded. We consider systems with distributed program and data memory to ensure predictability of the execution at runtime and scalability. We assume that the communication infrastructure used for inter-processor communication is predictable, i.e., it provides guaranteed communication latency. We use the worst-case communication latency to compute the worst-case execution time of a task, which in our approach includes the worst-case time needed for the task's computation and the worst-case time needed to perform inter-task data communication on the considered platform.

II. Related Work

[3] proposes an Integer Linear Programming (ILP) based approach for maximizing the throughput of an application modeled as an SDF graph by exploiting data parallelism when mapping the application on a platform with fixed number of processors. However, an ILP-based approach suffers from an exponential worst-case time complexity. To overcome the time complexity issue of the approach in [3], [4] separates the task replication and the allocation of replicas. However, decomposing the problem into two strongly related

problems and solving them separately has a negative impact on the solution quality. In addition, the maximum data-level parallelism is revealed in the application without considering the platform constraints. In contrast, in our approach, we solve the problem of task replication and the mapping of replicas simultaneously while taking into account the platform constraints. Both approaches [3] and [4] use *splitter* (S) and *joiner* (J) tasks to distribute and merge data streams processed by replicas, see Fig. 2(a). Those tasks introduce additional communication overhead as data streams have to be sent to them and to the replicas. Moreover, the splitter/joiner tasks have to be considered in the process of mapping and scheduling of tasks. In contrast, in our approach, we do not introduce additional tasks for data management, but we propose a new transformation on SDF in Sec. V where the data is sent by replicas of the original tasks only to replicas which need the data for computation. Thus, we avoid the overhead of scheduling splitter/joiner tasks and duplicated data transfers, as shown in Sec. VII.

[6] proposes a throughput driven transformation of an application modeled as an SDF graph for mapping the application on a platform. The graph transformation method in [6] increases the rates of data production/consumption and hence increases the buffer capacities needed to store the data, see Fig. 2(b). In addition, to enable unfolding of tasks, multiple firings of a certain task in the initial graph are combined into one firing of the corresponding task in the transformed graph, see the increased execution times of tasks in Fig. 2(b), which leads to an increase in latency. In contrast, our transformation technique does not increase the rates of data production/consumption on communication channels and does not combine multiple task firings into one firing which in turn leads to shorter application latency and smaller buffer sizes of the communication channels, as shown in Sec. VII.

The closest to our work, in terms of scope and methods proposed to efficiently utilize the parallelism of an application mapped onto resource-constrained platform is the work in [5]. The authors in [5] propose an approach for exploiting *just-enough* parallelism when mapping a streaming application modeled as an SDF graph on a platform with fixed number of processing elements. The graph transformation method in [5] transforms an initial SDF graph to functionally equivalent CSDF graph while keeping the same rates of data production/consumption on communication channels, see Fig. 2(c). However, the transformation approach in [5] is not efficient in terms of application latency and buffer sizes of the communication channels, as shown in Sec. VII. Moreover, the proposed algorithm in [5] for finding the values of unfolding factors and the mapping of task replicas does not reveal the right amount of parallelism, but it reveals more parallelism than needed and hence the platform is unnecessarily overloaded, as shown in Sec. VII. In contrast, the approach we propose unfolds a graph by doing more aggressive token-flow analysis leading to shorter application latency and smaller buffer sizes. In addition, our approach finds smaller unfolding factors for tasks which leads to less memory needed to store the code of replicas and less memory to implement communication channels between the replicas.

III. Background

In this section, we first introduce the application model, i.e., the SDF MoC. After that we review the scheduling framework proposed in [7], which we use to schedule tasks in the SDF graph.

A. Synchronous Data Flow (SDF)

An application modeled as an SDF [1] is a directed graph $G = (V, E)$ that consists of a set of actors V which communicate with each other through a set of communication channels E. Actors represent a certain functionality of the application, while communication channels are FIFOs representing data dependencies and transferring data tokens between the actors. Every output channel e_u of an actor τ_i has a predefined integer token *production* rate x_i^u. Analogously, a token *consumption* rate on every input channel e_u of an actor τ_i is a predefined integer y_i^u. In addition, for each actor τ_i, we associate a Worst-Case Execution Time (WCET) C_i, where C_i contains the worst-case computation time and the worst-case data-communication time needed to read/write data on the input/output channels of actor τ_i. An important property of the SDF model is the ability to derive at design-time a schedule for the actors. In order to derive a valid static schedule for an SDF graph at design-time, it has

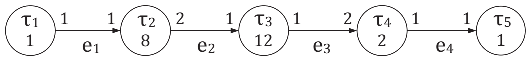


Fig. 1. An SDF graph G.

to be consistent and live. An SDF graph G is said to be consistent if a non-trivial *repetition vector* $\vec{q} = [q_1, q_2, \cdots, q_N]^T \in \mathbb{N}^N$ exists. The smallest non-trivial repetition vector is called *basic repetition vector*. An entry $q_i \in \vec{q}$ represents the number of invocations of actor τ_i in a graph iteration of G. If a deadlock-free schedule can be found, G is said to be live.

Fig. 1 shows an example of an SDF graph. For instance, actor τ_2 has WCET $C_2 = 8$ and its token production rate x_2^2 on channel e_2 is 2. The repetition vector of G in Fig. 1 is $\vec{q} = [1, 1, 2, 1, 1]^T$. Throughout this paper, all SDF graphs are assumed to be consistent and live.

B. Hard Real-Time Scheduling of (C)SDF

In [7], a real-time strictly periodic scheduling (SPS) framework for acyclic CSDF graphs is proposed. The CSDF MoC [2] is the superset of the SDF MoC where each CSDF actor produces/consumes a variable but predefined number of data tokens within a production/consumption sequence. Similar to the SDF MoC, the necessary condition for the existence of a valid periodic schedule for a given CSDF graph is to have a non-trivial repetition vector $\vec{q}$. Two examples of a CSDF graph are given in Fig. 2(c) and in Fig. 2(d).

In the framework in [7], every actor τ_i in a CSDF graph G is converted to an *implicit-deadline periodic* task $\tau_i = (S_i, C_i, D_i, T_i)$ by computing the task parameters. Parameter S_i is the start time of τ_i in absolute time units, C_i is the WCET, D_i is the deadline of τ_i in relative time units and $D_i = T_i$, and T_i is the task period (where $T_i \geq C_i$) in relative time units. To execute graph G in strictly periodic fashion, period T_i for each actor τ_i is computed as in [7]:

$$T_i = \frac{\mathrm{lcm}(\vec{q})}{q_i} \cdot s, \forall \tau_i \in V, \tag{1}$$

$$s = \left\lceil \frac{\max_{\tau_j \in V}\{C_j \cdot q_j\}}{\mathrm{lcm}(\vec{q})} \right\rceil, \tag{2}$$

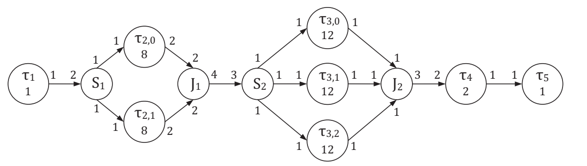
where $\mathrm{lcm}(\vec{q})$ is the least common multiple of all repetition entries in $\vec{q}$. These derived actor periods ensure that each actor τ_i executes q_i times in every graph *iteration period*, also called *hyperperiod*. Note that periods computed by Eq. (1) are the minimum periods for actors scheduled by SPS and that there exist other larger valid periods for actors by taking any integer $s > \left\lceil \frac{\max_{\tau_j \in V}\{C_j \cdot q_j\}}{\mathrm{lcm}(\vec{q})} \right\rceil$. Once the actor periods are computed, we can compute the utilization of actor τ_i, denoted as u_i, $u_i = C_i/T_i$, where $u_i \in (0, 1]$. For a graph G, u_G is the total utilization of G given by:

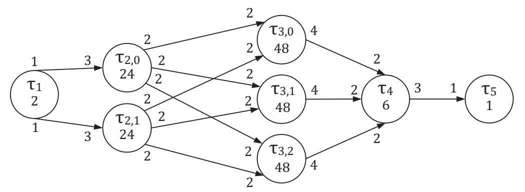
$$u_G = \sum_{\tau_i \in V} u_i = \sum_{\tau_i \in V} \frac{C_i}{T_i}. \tag{3}$$

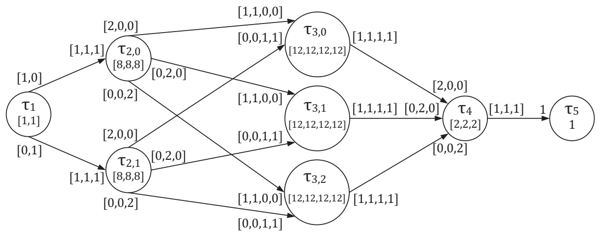
The total utilization of a graph directly determines the minimum number of processors needed to schedule the graph.

The throughput of each actor τ_i can be computed as $1/T_i$. The throughput of a graph G when its actors are scheduled as strictly periodic tasks is determined by the period of the output actor and is equal to $1/T_{out}$. The authors also provide in [7] a method for calculating the latency of a CSDF graph scheduled in a strictly periodic fashion. In addition, the framework computes the minimum buffer size for each channel in a graph such that actors, i.e., tasks, can be executed in strictly periodic fashion.

Converting the actors to periodic tasks (as described above) enables the application of the well-developed hard real-time scheduling theory [9], and hence, fast analytical calculation of the minimum number of processors needed to schedule the tasks in a CSDF. In real-time systems, tasks can be scheduled on processors by using global, hybrid, or partitioned scheduling algorithms [9]. However, global and hybrid scheduling algorithms require task migration, and thus introduce additional run-time overheads and memory overhead on distributed memory systems. The other class


(a) Equivalent of G in Fig. 1 after the transformation in [3], [4]


(b) Equivalent of G in Fig. 1 after the transformation in [6]


(c) Equivalent of G in Fig. 1 after the transformation in [5]

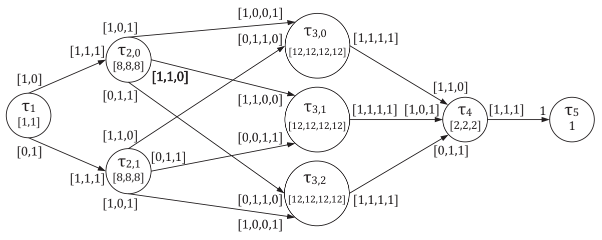

(d) Equivalent of G in Fig. 1 after our transformation

Fig. 2. Equivalent graphs of the SDF graph in Fig. 1 by unfolding actor τ_2 by factor 2 and τ_3 by factor 3.

of scheduling algorithms are partitioned algorithms which do not require task migration, hence they have low run-time overheads. Therefore, in this paper, we consider the partitioned scheduling algorithms. With partitioned scheduling, tasks are first allocated to processors. Then, the tasks on each processor are scheduled using a uniprocessor (hard) real-time scheduling algorithm.

IV. Motivational Example

In the first part of this section, we motivate the need for our new unfolding graph transformation. The throughput of graph G given in Fig. 1 when scheduled under SPS [7] is the same as the throughput obtained under self-timed scheduling [10] and it is equal to $\frac{1}{24}$. Note that an unfolding graph transformation is used to increase the application throughput if it is allowed by the hardware platform on which the application is executed. Let us assume that actors τ_2 and τ_3 of graph G in Fig. 1 are unfolded by factors 2 and 3, respectively, in order to increase the throughput of G. Fig. 2 shows four functionally equivalent graphs obtained after applying the unfolding transformations proposed by [3], [4] – see Fig. 2(a), by [6] – see Fig. 2(b), and by the transformation in [5] – see Fig. 2(c), while the graph given in Fig. 2(d) is obtained by applying our transformation described in Sec. V. Our transformation method unfolds an SDF graph by doing more aggressive data token flow analysis with the aim to spread equally the workload of an actor during the hyperperiod and run in parallel as much replicas of the actor as possible. Table I gives for all four equivalent graphs of G the throughput $\mathcal{R}_{\text{out}}$ of the output actor, actor τ_5, the maximum latency $\mathcal{L}_{\text{in}\to\text{out}}$ on an input-output path, the total size M, of the communication buffers, the total code size CS, and the total number of processors m needed to schedule the graphs under SPS and the self-timed scheduling while achieving the same

TABLE I
Results for G transformed by different transformation approaches.

Approach	SPS [7]					[10]				
	$\mathcal{R}_{\text{out}}[\frac{1}{\mu s}]$	$\mathcal{L}_{\text{in}\to\text{out}}[\mu s]$	$M[B]$	$CS[kB]$	m	$\mathcal{R}_{\text{out}}[\frac{1}{\mu s}]$	$\mathcal{L}_{\text{in}\to\text{out}}[\mu s]$	$M[B]$	$CS[kB]$	m
[3], [4]	1/8	128	50	40	5	1/8	67	31	40	12
[6]	1/8	176	66	36	5	1/8	93	57	36	8
[5]	1/8	80	32	36	5	1/8	76	24	36	8
our	1/8	56	24	36	5	1/8	62	21	36	8

TABLE II
Results for G transformed and mapped on 2 processors by different approaches.

Approach	$\mathcal{R}_{\text{out}}[\frac{1}{\mu s}]$	$\mathcal{L}_{\text{in}\to\text{out}}[\mu s]$	$M[B]$	$CS[kB]$	m
[5]	1/18	180	31	44	2
our	1/18	108	16	32	2

throughput $\mathcal{R}_{\text{out}}$. We can see from the table that by applying our unfolding transformation we can obtain, under SPS, 2.29, 3.14, and 1.43 times shorter latency and 2.08, 2.75, and 1.33 times smaller buffers than the unfolding methods in [3] and [4], [6], and [5], respectively. The number of processors needed to schedule the graph obtained after the transformation under SPS is equal for all the transformation methods. Under self-timed scheduling [10] we obtain 1.08, 1.5, and 1.23 times shorter latency, while buffers are smaller 1.47, 2.71, and 1.14 times compared to the related approaches. Assuming one-to-one mapping for the self-timed scheduling, we need the same number of processors to schedule the unfolded graph obtained by the methods in [6] and [5], and 1.5 times less processors than the unfolding methods in [3] and [4]. For both scheduling algorithms we obtain equal code size as the unfolding methods in [6] and [5], and 1.11 times smaller code size than the methods in [3] and [4]. From Table I, we see that our unfolding transformation approach presented in Sec. V is more efficient than the approaches in [3], [4], [6], and [5].

So far, we considered only the unfolding transformation. Now, we would like to focus on the algorithm for finding the proper unfolding factors for actors when a graph is mapped onto resource-constrained platform and scheduled by a hard real-time scheduler such that the throughput of the graph is maximized. Here, we want to compare our algorithm in Sec. VI with the approach in [5], because only that approach, among the related approaches, exploits the parallelism in an application under hard real-time scheduling. For example, in order to schedule graph G in Fig. 1 on a platform with 2 processors while maximizing the throughput under hard real-time scheduling, the approach in [5] finds a vector of unfolding factors $\vec{f} = [1, 2, 4, 1, 1]$. However, there exists a smaller vector of unfolding factors, i.e., $\vec{f} = [1, 1, 3, 1, 1]$, such that G is schedulable on 2 processors and the throughput is maximized. This smaller vector $\vec{f}$ is found by our algorithm in Sec. VI. Table II gives the throughput $\mathcal{R}_{\text{out}}$, latency $\mathcal{L}_{\text{in}\to\text{out}}$, buffer sizes M and code size CS when G is unfolded and mapped on $m = 2$ processors by applying the approach in [5] and by applying our algorithm presented in Sec. VI. We can see from the table that by applying our algorithm we obtain under SPS 1.67 times shorter latency, 1.94 times smaller buffers, and 1.38 smaller code size than the approach in [5]. From these results and the results given in Table I, we clearly show the necessity and usefulness of the graph unfolding transformation presented in Sec. V, and the algorithm for finding proper values for the unfolding factors presented in Sec. VI.

V. New Unfolding Transformation of SDF Graphs

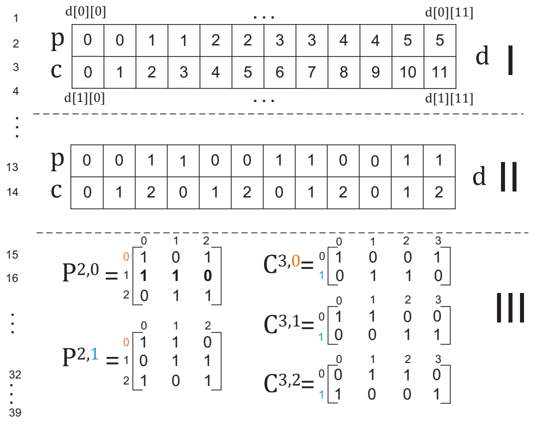
Our new unfolding transformation method is given in Algorithm 1. The algorithm takes an SDF graph G and a vector of unfolding factors $\vec{f}$ and produces an unfolded graph G', which is a CSDF graph. The initial SDF graph and its unfolded version given in the form of a CSDF graph are functionally equivalent, i.e., both of them generate the same sequence of output data tokens for a given sequence of input data tokens. The algorithm consists of three phases. The first phase is given in lines 1 to 4 in Algorithm 1. Given that the execution semantics of the SDF model allows any integer multiple of the basic repetition vector also as a valid repetition vector, in line 1 of Algorithm 1 the basic repetition vector $\vec{q}$ of G is replaced by $\vec{q}^f = \text{lcm}(\vec{f}) \cdot \vec{q}$, where $\text{lcm}(\vec{f})$ is the least common multiple of all elements in $\vec{f}$. Then in lines 2 to 4, for each channel e_u in G, a matrix d is constructed containing as many columns as the number of tokens produced/consumed on the channel during one iteration of G with repetition vector $\vec{q}^f$. Each column in d

Algorithm 1: Procedure to unfold an SDF graph.

```
Input: An SDF graph $G = (V, E)$, a vector of unfolding factors $\vec{f}$.
Output: The equivalent CSDF graph $G' = (V', E')$.
1  Take $\vec{q}^f = [\mathrm{lcm}(\vec{f}) \cdot q_1, \cdots, \mathrm{lcm}(\vec{f}) \cdot q_N]$ as a repetition vector of $G$;
2  for communication channel $e_u = (\tau_i, \tau_j) \in E$ do
3      Get production rate prd and consumption rate cns on $e_u$;
4      Construct a matrix $d$, $d[0][t] = p$, $d[1][t] = c$, $t \in [0, prd \cdot q_i^f - 1]$,
       $p$ is the index of $\tau_i$ firing which produces $t^{th}$ token, $c$ is the index
       of $\tau_j$ firing which consumes $t^{th}$ token on $e_u$;
5  $V' \leftarrow \emptyset$, $E' \leftarrow \emptyset$;
6  for actor $\tau_i \in V$ do
7      for $k = 1$ to $f_i$ do
8          Add replica $\tau_{i,k}$ to $V'$;
9  for communication channel $e_u = (\tau_i, \tau_j) \in E$ do
10     for replica $\tau_{i,k}$ of $\tau_i$ do
11         for replica $\tau_{j,l}$ of $\tau_j$ do
12             Add $e'_u = (\tau_{i,k}, \tau_{j,l})$ to $E'$;
13     for $t = 0$ to $prd \cdot q_i^f - 1$ do
14         $d[0][t] = d[0][t] \bmod f_i$, $d[1][t] = d[1][t] \bmod f_j$;
15 for communication channel $e_u = (\tau_i, \tau_j) \in E$ do
16     Get production rate prd and consumption rate cns on $e_u$;
17     Create empty/zero matrices $P^{i,k}$ with size $f_j \times q_{i,k}$, $k \in [0, f_i - 1]$;
18     Create empty/zero matrices $C^{j,l}$ with size $f_i \times q_{j,l}$, $l \in [0, f_j - 1]$;
19     for $h = 0$ to $q_{i,0} - 1$ do
20         for $k = 0$ to $f_i - 1$ do
21             Initialize a prod. counter seq. $cnt_{\mathrm{prod}}$ of length $f_j$ to 0;
22             for $o = 0$ to $prd - 1$ do
23                 $cnt_{\mathrm{prod}}[d[1][h \cdot k \cdot prd + o]] =$
                   $cnt_{\mathrm{prod}}[d[1][h \cdot k \cdot prd + o]] + 1$;
24             for $l = 0$ to $f_j - 1$ do
25                 $P^{i,k}[l][h] = cnt_{\mathrm{prod}}[l]$;
26     for $h = 0$ to $q_{j,0} - 1$ do
27         for $l = 0$ to $f_j - 1$ do
28             Initialize a cons. counter seq. $cnt_{\mathrm{cons}}$ of length $f_i$ to 0;
29             for $o = 0$ to $cns - 1$ do
30                 $cnt_{\mathrm{cons}}[d[0][h \cdot l \cdot cns + o]] = cnt_{\mathrm{cons}}[d[0][h \cdot l \cdot cns + o]] + 1$;
31             for $k = 0$ to $f_i - 1$ do
32                 $C^{j,l}[k][h] = cnt_{\mathrm{cons}}[k]$;
33     for $k = 0$ to $f_i - 1$ do
34         for $l = 0$ to $f_j - 1$ do
35             if all entries in row $P^{i,k}[l][]$ are 0 then
36                 Delete a channel $e'_u$ connecting replicas $\tau_{i,k}$ and $\tau_{j,l}$;
37             else
38                 Associate production sequence $P^{i,k}[l][]$ and
                   consumption sequence $C^{j,l}[k][]$ with $e'_u = (\tau_{i,k}, \tau_{j,l})$;
39 return $G'$;
```

contains in row 0 an index p, $d[0][t] = p$, which is the index of the firing of the producer actor, $p \geq 0$, which produces the t^{th} token, and an index c in row 1, $d[1][t] = c$, representing the index of the firing of the consumer actor, $c \geq 0$, which consumes the t^{th} token on e_u. Constructing matrix d for channel e_2 of graph G in Fig. 1 when $\vec{f} = [1, 2, 3, 1, 1]$ is given in Fig. 3, lines 2 to 4.

In the second phase the topology of the equivalent CSDF graph G' is created, which is given in lines 5 to 14 in Algorithm 1. In the equivalent CSDF graph G', every actor is replicated a certain number of times, as determined by the unfolding vector, lines 6 to 8. Then each channel e_u in the initial graph is replicated a certain number of times in the equivalent graph such that each replica of the producer on e_u is connected to each replica of the consumer on e_u, as given in lines 9 to 12 of Algorithm 1. The motivation behind unfolding is to equally distribute the workload of an actor in the initial graph by running in parallel replicas corresponding to that actor. The workload of an actor within one graph iteration is determined by the corresponding repetition value of the actor. Thus, each replica $\tau_{i,k} \in G'$ of an actor $\tau_i \in G$ will have the repetition $q_{i,k}$:



1
2
3
4
13
14
15
16
32
39
d[0][0]
...
d[0][11]
p
c
d[1][0]
...
d[1][11]
d I
d II
III
$P^{2,0} = \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 0 & 1 \\ 1 & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 2 & 0 & 1 & 1 \end{array}$
$P^{2,1} = \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 2 & 1 & 0 & 1 \end{array}$
$C^{3,0} = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{array}$
$C^{3,1} = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{array}$
$C^{3,2} = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{array}$

p	0	0	1	1	2	2	3	3	4	4	5	5
c	0	1	2	3	4	5	6	7	8	9	10	11

p	0	0	1	1	0	0	1	1	0	0	1	1
c	0	1	2	0	1	2	0	1	2	0	1	2

Fig. 3. Unfolding channel e_2 from the graph in Fig. 1 by using Algorithm 1 when $\vec{f} = [1, 2, 3, 1, 1]$.

$$q_{i,k} = \frac{q_i^f}{f_i} = \frac{q_i \cdot \mathrm{lcm}(\vec{f})}{f_i}. \tag{4}$$

For example, after the unfolding of the SDF graph in Fig. 1 with the unfolding vector $\vec{f} = [1, 2, 3, 1, 1]$ we obtain the graph shown in Fig. 2(d) with the repetition vector $\vec{q'} = [6, 3, 3, 4, 4, 4, 6, 6]$, where $q_{2,0} = q_{2,1} = \frac{1 \cdot lcm(1,2,3,1,1)}{2} = 3$. Lines 13 to 14 convert the firing production/consumption indexes in $d[0][t]/d[1][t]$ for each token t produced/consumed on channel e_u into the indexes corresponding to the index k of the replica which produces/consumes t. This is illustrated for channel e_2 in Fig. 3, lines 13 and 14.

Lines 15 to 39 represent the third phase of Algorithm 1 and they derive the production and consumption sequences for new channels and perform final placement of the new channels between the corresponding actor replicas. More specifically, for each source replica $\tau_{i,k}$ and destination replica $\tau_{j,l}$ of a channel, a production matrix $P^{i,k}$ and consumption matrix $C^{j,l}$ is created from matrix d in lines 15 to 32. The index of each row in a production matrix $P^{i,k}$ corresponds to the index l of a destination replica $\tau_{j,l}$. The index of each column in a production matrix $P^{i,k}$ corresponds to the firing index of source replica $\tau_{i,k}$. Elements in a production matrix of a source replica contain the number of tokens produced by a certain firing of that replica. Similar holds for elements in a consumption matrix. The created matrices $P^{2,0}$, $P^{2,1}$, $C^{3,0}$, $C^{3,1}$, $C^{3,2}$ for the source replicas $\tau_{2,0}$, $\tau_{2,1}$ and destination replicas $\tau_{3,0}$, $\tau_{3,1}$, $\tau_{3,2}$ on channel e_2 are given in Fig. 3. For example, the value 0 in element $P^{2,0}[\mathbf{2}][\mathbf{0}]$ says that 0 tokens are produced by the **0th** firing of source replica $\tau_{2,0}$ for the destination replica $\tau_{3,\mathbf{2}}$. Once these matrices are constructed, the production and consumption sequences on channel replicas are extracted from the corresponding rows in matrices, as given in lines 33 to 38 in Algorithm 1. For example, the production sequence on the channel between $\tau_{2,0}$ and $\tau_{3,\mathbf{1}}$ in Fig. 2(d) is extracted from row $P^{2,0}[\mathbf{1}][]$ in matrix $P^{2,0}$ and is equal to $[1, 1, 0]$. The extracted production/consumption sequences on replicas of channel e_2 can be seen in Fig. 2(d). The unfolded graph G' is returned in line 39 of Algorithm 1.

VI. The Algorithm for Finding Proper Unfolding Factors

In order to efficiently utilize the parallelism available in an application when mapping the application on a resource-constrained platform under hard real-time scheduling, proper unfolding factors for actors of the application have to be determined. Therefore, in this section, we present an algorithm which derives the proper unfolding factors which maximize the utilization of the platform, i.e., maximize the application throughput.

The algorithm is given in Algorithm 2. It takes an SDF graph G, where the actors are scheduled by SPS [7], a platform with m processors, a scheduling algorithm A [11], an allocation heuristic H [12] and a quality factor ρ. A quality factor $\rho \in (0, 1]$ determines how much of the platform processing resources we want to utilize, with $\rho = 1$ corresponding to full utilization. The algorithm returns the best solution vector of unfolding factors $\vec{f}^{best}$.

Algorithm 2: Finding proper unfolding factors for an SDF graph mapped onto resource-constrained platform.

```
Input: An SDF graph $G$, the number of processors in a platform $m$,
       quality factor $\rho$, a scheduling algorithm $A$, an allocation
       heuristic $H$.
Output: Vector of unfolding factors $\vec{f}^{best}$.
1  $\vec{f} = [1, 1, \cdots, 1]$; $G' = G$;
2  Compute the upper bound $\hat{\vec{f}}$ of $\vec{f}$ by Eq. (3) in [5];
3  Get $u_{G'}$ of $G'$ by Algorithm 3 when scheduled by $A$ and $H$ on $m$;
4  $u_{G^{best}} = u_{G'}$; $\vec{f}^{best} = \vec{f}$;
5  Find the bottleneck actor $\tau_{b,k}$ in $G'$;
6  while $u_{G'} < \rho \cdot m$ and $f_b < \hat{f}_b$ do
7      $f_b = f_b + 1$;
8      Get $G'$ by unfolding $G$ by Algorithm 1;
9      Get $u_{G'}$ of $G'$ by Algorithm 3 when scheduled by $A$ and $H$ on $m$;
10     if $u_{G'} > u_{G^{best}}$ then
11         $u_{G^{best}} = u_{G'}$; $\vec{f}^{best} = \vec{f}$;
12     Find the bottleneck actor $\tau_{b,k}$ in $G'$;
13 return $\vec{f}^{best}$.
```

Algorithm 3: Procedure to find the utilization of a CSDF graph mapped onto resource-constrained platform.

```
Input: A CSDF graph $G'$, the number of processors in a platform $m$, a
       scheduling algorithm $A$, an allocation heuristic $H$.
Output: Graph utilization $u_{G'}$.
1  Calculate $s$ by Eq. (2); calculate $T_i$ by using $s$ in Eq. (1);
2  Calculate $u_{G'}$ by Eq. (3);
3  while $G'$ is not schedulable on $m$ by $A$ and $H$ do
4      $s = s + 1$;
5      Calculate $T_i$ by using $s$ in Eq. (1); calculate $u_{G'}$ by Eq. (3);
6  return $u_{G'}$.
```

Line 1 in Algorithm 2 initializes each unfolding factor of an actor in G to 1 and G' to G. Then, the upper bound $\hat{f}_i$ of unfolding factor f_i for each actor τ_i in G is computed in line 2 in Algorithm 2 by using Eq. (3) in [5]. Line 3 finds the utilization of graph G' when G' is scheduled on m processors by invoking Algorithm 3. The best utilization of G' is initialized in line 4 to be the first schedulable solution on m processors found by Algorithm 3 in line 3. Line 5 finds the bottleneck actor in G'. The bottleneck actor $\tau_{b,k}$ is the actor with the heaviest workload during one hyperperiod, i.e., $C_{b,k} \cdot q_{b,k} = \max_{\tau_{i,k} \in V'}\{C_{i,k} \cdot q_{i,k}\}$. If multiple actors have the same maximum workload, then the one with the smallest code size is selected to be the bottleneck. If the current utilization $u_{G'}$ does not meet the quality requirement checked in line 6, the unfolding factor f_b of the bottleneck actor $\tau_{b,k}$ is increased in line 7 and the graph is unfolded by using Algorithm 1 in line 8. Note that stateful actors and input and output actors are not unfolded, i.e., the upper bound on their unfolding factors is 1. The utilization $u_{G'}$ of the unfolded graph G' mapped on m processors is calculated in line 9 by Algorithm 3. If the current utilization $u_{G'}$ is higher than the best utilization in line 10, then in line 11 the best utilization becomes the one found in line 9 and the best solution vector of unfolding factors becomes the current vector of unfolding factors. Line 12 finds the bottleneck actor in the unfolded graph G'. Lines 6 to 12 are repeated and the algorithm terminates when either a pre-specified quality factor ρ is satisfied (i.e., $u_{G'} \geq \rho \cdot m$) or the unfolding factor of a bottleneck actor exceeds its upper bound $\hat{f}_b$ (i.e., $f_b \geq \hat{f}_b$).

We see that Algorithm 2 uses Algorithm 3 for finding the utilization of the unfolded graph G' when mapped on a platform with m processors. Algorithm 3 takes the unfolded CSDF graph G', a platform with m processors, a scheduling algorithm A [11] and an allocation heuristic H [12] as inputs. Line 1 calculates periods of actors in G' scheduled by SPS [7] by using Eq. (1) and Eq. (2) in Sec. III-B. Then, the utilization $u_{G'}$ of G' is calculated in line 2 by using Eq. (3). As we mentioned in Sec. III-B, actor periods computed by Eq. (1) and Eq. (2) represent the minimum periods when the actors are scheduled under SPS on a platform with unlimited number of processors. It may happen that these minimum periods lead to a graph which is not schedulable on a platform with only m processors. Hence, in line 3 by using the utilization $u_{G'}$ calculated in line 2 we check if G' can be scheduled on m processors by using the corresponding schedulability test for A and H [9]. If G' is not schedulable on the platform, we decrease $u_{G'}$ until G' becomes schedulable by increasing the actor periods T_i. This is done in lines 4 and 5 in Algorithm 3. Once the graph G' becomes schedulable on m processors by A and H, Algorithm 3 returns the utilization of the unfolded graph G' in line 6.

TABLE III
BENCHMARKS USED FOR EVALUATION.

Application	$\|V\|$	$\|E\|$	Application	$\|V\|$	$\|E\|$
Discrete cosine transform (DCT)	8	7	Filterbank	85	99
Fast Fourier transform (FFT)	17	16	Serpent	120	128
Time delay equalization (TDE)	29	28	MPEG2	23	26
Data encryption standard (DES)	53	60	Vocoder	114	147
Bitonic Sorting	40	46	FMRadio	43	53
Channel Vocoder	55	70			

VII. EVALUATION

We present two experiments to evaluate the techniques proposed in Sec. V and Sec. VI. In the first experiment, we evaluate the efficiency of our unfolding transformation in comparison to the unfolding transformation methods in [3], [4], [6], and [5]. In the second experiment, we evaluate the efficiency of Algorithm 2 presented in Sec. VI in terms of performance and time complexity by comparing our approach to the related approach in [5]. The experiments were performed on the real-life applications from the StreamIt benchmarks suit [8], given in Table III. The results of the evaluations are shown in Fig. 4, Fig. 5, and Fig. 6. In all these figures, each vertical line shows the variations in the corresponding results among all the applications. The upper and lower ends of a vertical line represent the maximum and minimum values of the corresponding result while the marker at the middle of each vertical line represents the geometric mean of the result. Note that the Y axis in Fig. 4 to Fig. 6 has a logarithmic scale. We run all the experiments on an Intel Core i7-2620M CPU running at 2.70 GHz with Linux Ubuntu 12.4.

A. Efficiency of the Proposed Unfolding Transformation

In this section, we evaluate the performance of our unfolding transformation method proposed in Sec. V by comparison to the related unfolding transformation methods in [3], [4], [6], and [5]. In this experiment, first we use Algorithm 2 to find a vector of unfolding factors for each application in Table III mapped on a platform with 64 processors with partitioned First-Fit Decreasing Earliest Deadline First (FFD-EDF) scheduler and quality factor $\rho = 0.9$. Then, for each application, we use the found vector of unfolding factors to unfold the application graph by applying our transformation method and the related transformation methods [3], [4], [6], [5]. Finally, we use the SPS framework in [7] to calculate the latency, buffer sizes and code size when the unfolded graphs are scheduled by FFD-EDF on 64-processor platform. The ratios between the results obtained by *related* transformation methods and *our* transformation in terms of application latency ($\mathcal{L}$), buffer sizes (M) and code size (CS) are given in Fig. 4. We can see that our method outperforms all the related methods, and delivers on average 2.82, 3.95, and 1.43 times shorter latency and 1.98, 2.5, and 1.08 times smaller buffers than the method in [3] and [4], [6], and [5], respectively. Although the methods in [3] and [4] introduce additional actors for data management, the average increase in the total code size is only 1%. The other two transformation methods, [6] and [5], have the same code size as our method. Note that all the methods achieve the same application throughput.

B. Performance of Algorithm 2

We evaluate the performance of Algorithm 2 by comparison to the related approach in [5]. For each application *app* in Table III, we construct 28 system configurations (app, m, ρ) with number of processors $m \in \{2, 4, 8, 16, 32, 64, 128\}$, and utilization quality $\rho \in \{0.8, 0.85, 0.9, 0.95\}$. We run Algorithm 2 with FFD-EDF scheduler for each (app, m, ρ) configuration to obtain a vector of unfolding factors $\vec{f}^{best}$. Then, for each configuration, we unfold the corresponding application graph by the obtained vector $\vec{f}^{best}$ by using Algorithm 1. Finally, we use the SPS framework in [7] to calculate the latency of an application, buffer sizes and code size when the unfolded graphs are scheduled by FFD-EDF on

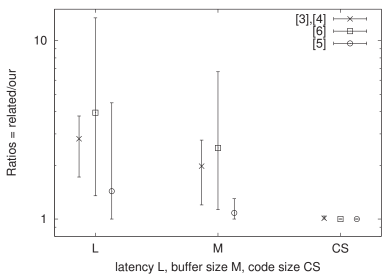
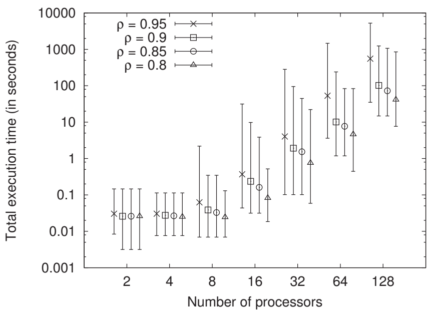
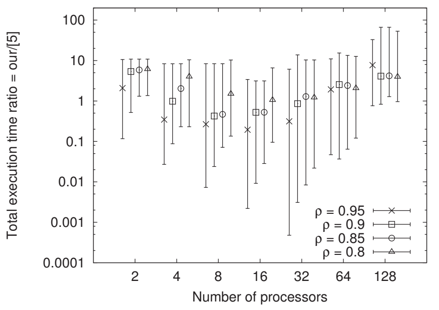


Fig. 4. Comparison of our unfolding transformation to the approaches in [3], [4], [6], [5].

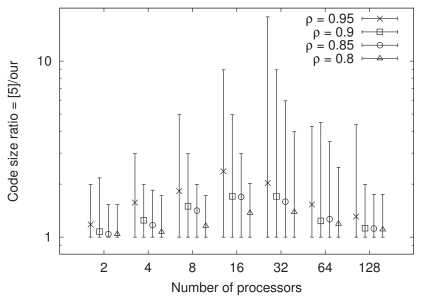


(a) Running time (in seconds) of Algorithm 2

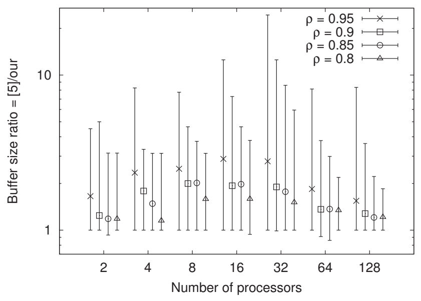


(b) Execution time ratio (lower is better)

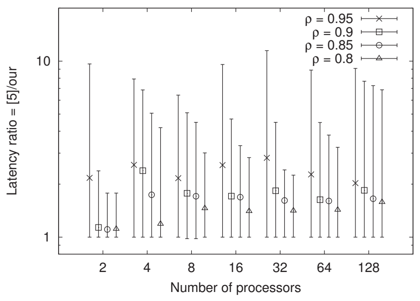
Fig. 5. Results of time evaluation of our proposed approach in comparison to the approach in [5]



(a) Code size ratio (higher is better)



(b) Buffer size ratio (higher is better)



(c) Latency ratio (higher is better)

Fig. 6. Results of performance evaluation of our proposed approach in comparison to the approach in [5].

the corresponding platform. We perform the same experiment by running the related algorithm proposed in [5] and using the SPS framework in [7] for each (*app*, *m*, ρ). The obtained ratios for the total code size, total buffer sizes, and latency between the approach in [5] and our approach are given in Fig. 6(a), Fig. 6(b), and Fig. 6(c), respectively. We can see that by using Algorithm 2 we can achieve up to 17.85 times smaller code size (see Fig. 6(a), ρ=0.95, *m*=32), up to 24.4 times smaller buffers (see Fig. 6(b), ρ=0.95, *m*=32) and up to 11.47 times shorter latency (see Fig. 6(c), ρ=0.95, *m*=32) than the approach in [5]. Note that both approaches meet the same throughput requirements. Regarding the buffer sizes, we obtain in 5 experiments out of 308 experiments larger buffer sizes by up to 1.16 times than the approach in [5] (see for example Fig. 6(b), ρ=0.85, *m*=64). However, for all these experiments we do less unfolding, so we obtain smaller code size. In the case of latency, in 2 experiments out of 308 we get latency which is by 2% larger than the corresponding latency when the approach in [5] is applied (see Fig. 6(c), *m*=8). However, in these two cases, we obtain smaller code size and smaller buffer sizes than the approach in [5].

C. Time Complexity of Algorithm 2

We evaluate the efficiency of our algorithm for finding proper values of unfolding factors in terms of the execution time of our Algorithm 2 to find a solution. The execution times for different quality factors and different number of processors in a platform are given in Fig. 5(a). We compare these execution times with the corresponding execution times of the related approach in [5]. The comparison is given in Fig. 5(b).

As can be seen from Fig. 5(a), for platforms containing up to 16 processors, our Algorithm 2 takes in the worst case 32 seconds to find a solution, and less than 1 second on average for all values of quality factor ρ. For a platform with 32 processors, the execution time of our algorithm is 5 minutes in the worst case, and up to 4 seconds on average. In the case of a 64-processor platform our algorithm needs 25 minutes in the worst case to find a solution, and up to 53 seconds on average. Finally, for a platform with 128 processors Algorithm 2 takes 88 minutes in the worst case and up to 9 minutes on average to find a solution. In addition, it can be seen in Fig. 5(b) that our approach is on average up to 8 times slower than the approach in [5] which is acceptable given that our approach delivers solutions of better quality, as shown in Sec. VII-B, within a matter of minutes.

VIII. Conclusions

In this paper, we presented a new unfolding graph transformation and an algorithm which uses the transformation while exploiting the right amount of parallelism when mapping a streaming application modeled by an SDF graph on a resource-constrained platform under hard real-time scheduling. The experiments on a set of real-life applications showed that our proposed techniques deliver, in a matter of minutes, solutions with smaller code size, smaller buffer sizes and shorter application latency while meeting the same performance and timing requirements as the related approaches.

References

[1] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.
[2] G. Bilsen *et al.*, "Cyclo-static dataflow," *IEEE Trans. Signal Process.*, vol. 44, no. 2, pp. 397–408, 1996.
[3] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *PLDI*, 2008.
[4] S. M. Farhad *et al.*, "Orchestration by approximation: mapping stream programs onto multicore architectures," in *ASPLOS*, 2011.
[5] J. T. Zhai, M. Bamakhrama, and T. Stefanov, "Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems," in *DAC*, 2013.
[6] A. Stulova, R. Leupers, and G. Ascheid, "Throughput driven transformations of synchronous data flows for mapping to heterogeneous MPSoCs." in *ICSAMOS*, 2012.
[7] M. Bamakhrama and T. Stefanov, "On the hard-real-time scheduling of embedded streaming applications," *DAES*, vol. 17, no. 2, pp. 221–249, 2013.
[8] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *PACT*, 2010.
[9] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, p. 35, 2011.
[10] S. Stuijk, M. Geilen, and T. Basten, "Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs," *IEEE Trans. on Computers*, vol. 57, no. 10, pp. 1331–1345, 2008.
[11] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.
[12] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin packing: A survey," in *Approximation algorithms for NP-hard problems*, 1996, pp. 46–93.