Exploiting Just-enough Parallelism when Mapping Streaming Applications in Hard Real-time Systems

Jiali Teddy Zhai
tzhai@liacs.nl

Mohamed A. Bamakhrama
mohamed@liacs.nl

Todor Stefanov
stefanov@liacs.nl

Leiden Institute of Advanced Computer Science
Leiden University, Leiden, The Netherlands

ABSTRACT

Embedded streaming applications specified using parallel *Models of Computation* (MoC) often contain ample amount of parallelism which can be exploited using Multi-Processor System-on-Chip (MPSoC) platforms. It has been shown that the various forms of parallelism in an application should be explored to achieve the maximum system performance. However, if more parallelism is revealed than needed, it will overload the underlying MPSoC platform. At the same time, the revealed parallelism should be sufficient such that the MPSoC platform is fully utilized. Therefore, the amount of revealed and exploited parallelism has to be just-enough with respect to the platform constraints. In this paper, we study the problem of exploiting just-enough parallelism by application task unfolding, when mapping streaming applications modeled using the Synchronous Data Flow (SDF) MoC onto MPSoC platforms in hard real-time systems. We show that our problem of simultaneously unfolding and allocating tasks under hard real-time scheduling has a bounded solution space and derive its upper bounds. Subsequently, we devise an efficient algorithm to solve the problem, while the obtained solution meets a pre-specified quality. The experiments on a set of real-life streaming applications demonstrate that our algorithm results, within reasonable amount of time, in a system specification with large performance gain. Finally, we show that our proposed algorithm is on average 100 times faster than one of the state-of-the-art meta-heuristics, i.e., NSGA-II genetic algorithm, while achieving the same quality of solutions.

1. INTRODUCTION

Streaming applications are widely used in embedded systems in several application domains, such as image processing, video/audio processing, and digital signal processing. The ample amount of parallelism in streaming applications matches perfectly the processing power of Multi-Processor System-on-Chip (MPSoC) platforms, which contain an increasing number of Processing Elements (PEs). Having many PEs available has imposed huge challenges on both application developers and design tools to identify and exploit the right amount of parallelism which can utilize the PEs efficiently. To tackle the challenges, *Models-of-Computation* (MoC) have been adopted as the parallel application specification in most of the design approaches and tools [7]. For example, Kahn Process Networks [11] and Synchronous Data Flow (SDF) [14] are two prominent parallel MoCs. In such MoCs, a streaming application is modeled as a directed graph, where the graph nodes represent the application tasks and the graph edges represent the data communication FIFO channels. The tasks execute concurrently and communicate data explicitly via the FIFOs. In this case, the task-level parallelism is naturally exposed.

However, in most cases, an initial graph given by application developers to specify application behavior is not the most suitable for a given MPSoC platform. This is because application developers mainly focus on realizing certain application behavior, including the identification of the functionality of tasks and the synchronization/communication between these tasks. The computational capacity of the MPSoC platform is often not fully taken into account. The authors in [12] showed that, for a set of representative streaming benchmarks, the maximum achievable speedup of mapping the initial graphs can only reach up to a limited number. To better utilize the underlying MPSoC platform, the initial graph of an application should be transformed to an alternative one that exposes more parallelism while preserving the same application behavior. To this end, task unfolding is an effective technique to generate such alternative graphs. Basically, task unfolding replicates the functionality of a task by a certain number of times, referred as *unfolding factor*. Then, replicas of tasks concurrently process different data, thereby exploring also data-level parallelism next to the task-level parallelism.

Unfolding individual tasks in an initial graph by different unfolding factors results in a large number of possible alternative graphs. To transform the initial graph to an alternative one by unfolding, the main problem is to determine a proper unfolding factor for each task. This problem is challenging because platform constraints must be considered during unfolding. The platform constraints can be the number of available PEs and temporal scheduling of tasks on the PEs. On the one hand in [22, 5, 20], the authors determine an unfolding factor for each task such that the obtained alternative graph exposes the maximum data-level parallelism, without considering the platform constraints. However, unfolding a task too many times reveals more parallelism than needed. The overwhelming parallelism leads to an inefficient mapping of replicas of tasks. That is, the excessive number of replicas cannot be efficiently allocated and temporally scheduled on the available PEs. Moreover, the excessive number of replicas introduces significant memory overhead for both code and data. On the other hand in [9, 12, 17], the authors assume that the unfolding factor of a task cannot exceed the number of available PEs. This assumption, however, restricts the amount of revealed parallelism because a proper unfolding factor is not necessarily less than or equal to the number of available PEs. As a consequence, the aforementioned assumption might lead to under-utilized PEs. From the discussion above, we can see that exploiting excessive or insufficient parallelism may result in sub-optimal system utilization and performance. Therefore, in this paper, we address the problem of determining a proper unfolding factor of each task in a given initial graph, such that the obtained alternative graph exposes *just-enough* parallelism to fully utilize the available PEs. This is achieved by considering the platform constraints when determining the unfolding factors.

We solve the problem explained above when a streaming application is modeled using the SDF MoC and mapped onto MPSoC platforms with hard real-time constraints. The SDF MoC has been successfully adopted in both industrial and academic tools. We consider the problem in the context of hard real-time systems, because many streaming applications nowadays require hard real-time execution. For instance, collision avoidance algorithms used in the avionics or automotive industry require very strict timing guarantees. At the same time, it has been reported in [1] that these algorithms require approximately 170 million calculations for each frame update, with the expectation of being executed on up to 64 PEs.

1.1 Paper Contributions

We propose an efficient approach to exploit just-enough parallelism in streaming applications modeled using the SDF MoC in hard real-time systems, in order to increase the performance that can be guaranteed on an MPSoC platform. More specifically, our problem is to determine simultaneously which actors (i.e., tasks) to unfold by what factor, and the allocation of unfolded actors onto PEs. We show that the solution space of our problem is bounded and derive its upper bounds. We then propose an efficient

DAC'13, May 29 - June 07 2013, Austin, TX, USA.

algorithm to find a solution to the problem, while the obtained solution meets a pre-specified quality. In addition, we evaluate the efficiency and time complexity of the proposed algorithm on 11 real-life applications. Finally, we show that our algorithm is, on average, 100 times faster than a state-of-the-art meta-heuristic, i.e., NSGA-II genetic algorithm [4], while achieving the same quality of the solution.

1.2 Scope of Work

In this paper, we assume that a given SDF graph is acyclic. Such assumption covers a large set of applications as it has been empirically shown in [18] that around 90% of streaming applications can be modeled as acyclic SDF graphs. Once a cycle exists in an SDF graph, one can always fuse all actors in the cycle into a single stateful actor. A stateful actor is the one whose next execution depends on the current execution. As a consequence, our approach does not unfold stateful actors. Furthermore, the data source and sink actors, which are connected to the external environment, are not unfolded. The target platform assumed in this work is a homogeneous programmable MPSoC with distributed memory. The interconnection structure between PEs must provide guaranteed communication latency, e.g., Æthereal network-on-chip [8].

2. RELATED WORK

The approach in [17] is closely related to our work, although the considered problem is relaxed, i.e., without considering timing constraints, compared to our problem. A genetic algorithm based heuristic is proposed to determine the unfolding factor of an actor and allocation of all replicas. The unfolding factor of an actor cannot exceed the number of PEs, which might result in sub-optimal solutions as we show later in Sec. 5. Moreover, we show in the experiments that our approach outperforms significantly the genetic algorithm based heuristic in terms of running time.

In [12], an Integer Linear Programming (ILP) formulation gives exact solutions to minimize makespan on any PE while simultaneously unfolding actors in an SDF graph and allocating them on PEs. In the ILP formulation, an unfolding factor of an actor cannot exceed the number of available PEs. This assumption might lead to sub-optimal system performance as discussed previously. Moreover, it has been shown in [5] that the ILP formulation is even intractable for benchmarks with medium graph size. For instance, it takes around 70 hours to solve the ILP formulation for the FFT benchmark with 26 actors on 4 PEs (see Table 2 in [5]). In practice, real-life applications have been shown to contain up to 2868 actors [18]. Therefore, it is clear that the ILP-based approach suffers from severe scalability issues. In contrast, our proposed algorithm solves the combined problem within a reasonable amount of time as demonstrated later in Sec. 7.

To address the scalability issue of [12], the authors in [5] propose to decompose the combined problem into two problems and solve them separately. The separation of the two problems often leads to inferior performance, as both problems are strongly related. In contrast, our proposed algorithm is capable of solving the combined problem simultaneously. Moreover, our algorithm takes into account timing constraints, while the work in [5] does not.

In the context of synthesizing an SDF graph using dedicated hardware, the authors in [10] also determine which actors to unfold and by what factor. The addressed problem is easier than ours because there is no need to consider allocation of actors after unfolding in case of hardware synthesis.

In [13], a synchronous programming model is used for the application specification under hard real-time scheduling. The term "synchronous" in this context refers to the fact that a master thread can *fork* a job into several parallel execution segments and they *join* upon completion. These parallel execution segments are, to some extent, similar to unfolded actors in our case. There is also no need to consider allocation of parallel segments at compile-time because migration at run-time is allowed targeting MPSoC platforms with shared memory. In contrast, we solve the problem of allocating actors at compile-time. Recall from Sec. 1.2 that we consider the MPSoC platforms with distributed memory. On such platforms, migration of actors at run-time introduces non-negligible overhead.

3. BACKGROUND

In Sec. 3.1, we first introduce the application specification, i.e., the SDF MoC, and the unfolding operation on SDF graphs. After that in Sec. 3.2, hard real-time scheduling of the SDF MoC is reviewed. These are essential for better understanding our problems formally defined in Sec. 4 and contributions presented in Sec. 5 and 6.

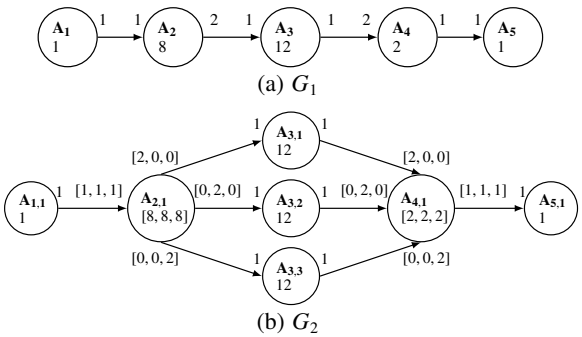


Figure 1: (a) An example of an SDF graph and (b) its equivalent CSDF graph by unfolding actor A_3 by factor 3.

3.1 Unfolding of SDF Graphs

The SDF MoC is defined as a directed graph $G = (\mathcal{A}, \mathcal{E})$, where $\mathcal{A}$ is a set of n actors and $\mathcal{E}$ is a set of communication edges. An actor $A_i \in \mathcal{A}$ executes by first consuming data tokens from all its incoming edges, performing certain computation, and subsequently producing data tokens to all its outgoing edges. The number of tokens consumed from an edge or produced to an edge is known a-priori and given as a constant integer. It has been shown in [14] that, to have a valid periodic schedule, an SDF graph has to be *consistent* with a non-trivial *repetition vector* $\vec{q} \in \mathbb{N}^n$. An entry $\mathsf{r}(A_i) \in \vec{q}$ denotes how many times an actor $A_i \in \mathcal{A}$ has to be executed in every graph iteration of G. Additionally, for each actor A_i, we associate a Worst-Case Execution Time (WCET) C_i and its code size S_i. For a summary of all the notations used in the paper, please refer to Appendix A.

An SDF graph G_1 is shown in Figure 1(a). The actors A_1 and A_5 are the data source and sink actors, respectively. G_1 has five actors and a repetition vector $\vec{q} = [1, 1, 2, 1, 1]^T$. The WCET of each actor is shown below its name, e.g., $C_3 = 12$ for actor A_3.

The unfolding operation on an SDF graph used in this paper is conceptually similar to the one used in [5, 9, 10, 12], in which two special constructs *splitter* and *joiner* are employed for the unfolded actors. Given a vector $\vec{f} \in \mathbb{N}^n$ of unfolding factors, where f_i denotes the unfolding factor for actor A_i, the unfolding operation replaces A_i by f_i replicas of itself. Then, instead of inserting a splitter and joiner before and after the f_i replicas of A_i, we transform the initial SDF graph to a functionally equivalent Cyclo-Static Data Flow (CSDF) [3] graph. The CSDF MoC generalizes the SDF MoC in the sense that each CSDF actor may produce or consume a variable but predefined number of data tokens in consecutive executions, called *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 $\vec{q'}$. To ensure the functional equivalence, the production and consumption rates of an SDF actor are modified accordingly to the production and consumption sequences in the resulting CSDF graph. This modification results in a different repetition vector of the obtained CSDF graph to ensure its consistency.

The algorithm for performing the unfolding of actors in SDF graphs is given in Algorithm 2 in Appendix C. The algorithm accepts as inputs an SDF graph G and a vector $\vec{f}$ of unfolding factors. The algorithm produces as an output a CSDF graph G', where $A_{i,f}$ denotes the fth replica of A_i with repetition $\mathsf{r}(A_{i,f})$ given by:

$$\mathsf{r}(A_{i,f}) = \frac{\mathsf{r}(A_i) \cdot \mathrm{lcm}(\vec{f})}{f_i}, \tag{1}$$

where $\mathsf{r}(A_i)$ is the repetition of actor A_i in the initial SDF graph and $\mathrm{lcm}(\vec{f})$ denotes the least common multiple of $f_i \in \vec{f}$. It follows that the repetition vector of G', denoted by $\vec{q'} \in \mathbb{N}^{n'}$ where $n' = \sum_{A_i \in \mathcal{A}} f_i$, is given by $\vec{q'} = [\mathsf{r}(A_{1,1}), \cdots, \mathsf{r}(A_{1,f_1}), \cdots, \mathsf{r}(A_{n,f_n})]^T$. After obtaining $\vec{q'}$ using Eq. 1, production/consumption sequences of each CSDF actor are generated accordingly.

Suppose that a vector of unfolding factors is given as $\vec{f} = [1, 1, 3, 1, 1]$ for G_1 in Figure 1(a). Algorithm 2 outputs a CSDF graph G_2 shown in Figure 1(b) with three replicas $A_{3,1}$, $A_{3,2}$ and $A_{3,3}$ for actor A_3 in G_1. The unfolding results in a repetition vector of G_2 as $\vec{q'} = [\mathsf{r}(A_{1,1}), \mathsf{r}(A_{2,1}), \mathsf{r}(A_{3,1}), \mathsf{r}(A_{3,2}), \mathsf{r}(A_{3,3}), \mathsf{r}(A_{4,1}), \mathsf{r}(A_{5,1})]^T = [3, 3, 2, 2, 2, 3, 3]^T$. For example, SDF actor A_4 executes only

once ($\mathsf{r}(A_4) = 1$) in G_1 per graph iteration, while executing three times ($\mathsf{r}(A_{4,1}) = 3$) in G_2 per graph iteration. Three consumption sequences of actor $A_{4,1}$ in G_2 behave similar to a joiner, with which $A_{4,1}$ collects data tokens from the three replicas $A_{3,1}$, $A_{3,2}$ and $A_{3,3}$. Analogous to a splitter, actor $A_{2,1}$ with three production sequences distributes tokens to the three replicas.

3.2 Hard Real-time Scheduling of (C)SDF

The authors in [2] showed that the actors in acyclic CSDF graphs can be executed in a strictly periodic fashion. Note that this result applies also to acyclic SDF graphs, since the SDF MoC is a special case of the CSDF MoC. As a result, a variety of hard-real-time scheduling algorithms, such as Earliest Deadline First (EDF, [15]), can be applied to temporally schedule the actors allocated on a PE.

To execute the actors in an acyclic (C)SDF graph in a strictly periodic fashion, the period of each actor needs to be computed first. To do so, we introduce the following definition:

Definition 1. The *workload* of a (C)SDF actor A_i per graph iteration, denoted by W_i, is given by $W_i = \mathsf{r}(A_i)C_i$, where $\mathsf{r}(A_i)$ is the repetition of A_i and C_i is the WCET of A_i.

Accordingly, we define the maximum workload per graph iteration, denoted by $\hat{W}_G$, as $\hat{W}_G = \max_{A_i \in \mathcal{A}}(\mathsf{r}(A_i)C_i)$. The period of an actor A_i, denoted by T_i where $T_i \in \mathbb{N}$, has a lower bound, denoted by $\check{T}_i$. It is given in [2] as follows:

$$\check{T}_i = \frac{\mathrm{lcm}(\vec{q})}{\mathsf{r}(A_i)} \left\lceil \frac{\hat{W}_G}{\mathrm{lcm}(\vec{q})} \right\rceil, \tag{2}$$

where $\mathrm{lcm}(\vec{q})$ is the least common multiple of all repetition entries in $\vec{q}$. The actual period T_i of an actor A_i is given by $T_i = s\check{T}_i$, where $s \in \mathbb{N}$ is the period *scaling factor* of the graph. For a given (C)SDF graph G, s is a constant that is used to scale the periods of all its actors. The deadline of each actor is the start of its next period, i.e., the deadline is equal to the period (often called *implicit deadline*). Once a period T_i of each actor is derived, we can compute the *utilization* $u_i \in (0, 1]$ of actor A_i as $u_i = C_i/T_i$. Based on this, we can also compute the utilization of a CSDF graph G, denoted by U_G, as $U_G = \sum_{A_i \in \mathcal{A}} C_i/T_i$. The throughput of a graph G when its actors are scheduled as strictly periodic actors is determined by the period of the sink actor. In the rest of the paper, we denote the lower bound on the period of the sink actor by $\check{T}_{\mathrm{snk}}$, and the actual period of the sink actor by T_{snk}. Accordingly, the throughput of the graph is $1/T_{\mathrm{snk}}$.

In this work, we consider that the schedule on each PE is built according to the EDF scheduling algorithm, which is known to be optimal on a uniprocessor system [15]. A set of actors allocated on a PE is schedulable using EDF if and only if their total utilization does not exceed 1. The schedule itself can be built either off-line for efficiency, or on-line for flexibility according to the system requirements. The problem of allocating actors onto PEs is similar to the bin-packing problem and can be solved using either *exact* or *approximate* allocation algorithms. An example of an exact allocation algorithm is proposed in [16], which returns an optimal allocation of actors. One disadvantage of using an exact algorithm is its high computational complexity. Therefore, to have a trade-off between optimality of the allocation and computational complexity, we choose in this paper an approximate allocation algorithm, namely the First-Fit Decreasing (FFD) algorithm, which has a proven worst-case approximation ratio $R_{\mathrm{FFD}} = \frac{11}{9}$ [21].

For instance, consider the CSDF graph G_2 in Figure 1(b) with repetition vector $\vec{q} = [3, 3, 2, 2, 2, 3, 3]^T$ as computed in Sec. 3.1. The WCET $C_{i,f}$ of each actor $A_{i,f}$ is shown below its name. The throughput of G_2 is determined by the period of the sink actor $A_{5,1}$. We first compute $\hat{W}_{G_2} = \mathsf{r}(A_{3,1}) \cdot C_{3,1} = 24$ and $\mathrm{lcm}(\vec{q}) = 6$. By solving Eq. 2, we obtain that the minimum period of the sink actor $A_{5,1}$ is $\check{T}_{\mathrm{snk}} = \frac{6}{3} \cdot \lceil \frac{24}{6} \rceil = 8$. This means that the maximum throughput of G_2 is $\frac{1}{8}$. The strictly periodic execution of all actors in G_2 is visualized in Figure 2. The bars indicate the execution of the actors in their periods and the up arrows denote the earliest starting time of each actor. It can be seen that actor $A_{5,1}$ executes for 1 time unit every 8 time units. The total utilization of G_2 is $U_{G_2} = \frac{1}{8} + \frac{8}{8} + \frac{12}{12} + \frac{12}{12} + \frac{12}{12} + \frac{2}{8} + \frac{1}{8} = 4.5$. To achieve the maximum throughput $\frac{1}{8}$ assuming the EDF scheduling algorithm and the FFD allocation algorithm, 5 PEs are required.

Using the scaling factor s defined earlier, we can have a trade-off between processing resources and guaranteed performance as shown in the following lemma:

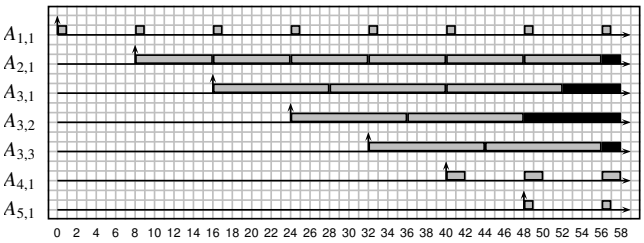


Figure 2: A schedule showing the strictly periodic execution of the actors in G_2. The x-axis represent the time. The last execution of $A_{2,1}$, $A_{3,1}$, $A_{3,2}$, and $A_{3,3}$ in the figure is truncated due to space limits.

LEMMA 1. *Let G be a CSDF graph that is schedulable using a scheduling algorithm SA and an allocation algorithm AA on $\check{m}$ PEs, when the period of each actor A_i is equal to $\check{T}_i$. G is schedulable using the same SA and AA on $\lceil \frac{\check{m}}{s} \rceil$ PEs, when the period of each actor A_i is equal to $s\check{T}_i$.*

The proof of Lemma 1 can be found in Appendix B. Considering G_2 in Figure 1(b), it can be scheduled on $\lceil \frac{5}{2} \rceil = 3$ PEs achieving a period $T_{\mathrm{snk}} = 2 \times \check{T}_{\mathrm{snk}} = 16$, i.e., throughput $\frac{1}{16}$ by scaling all minimum periods by $s = 2$.

Now, suppose that *AA* is an approximate allocation algorithm with an approximation ratio R_{AA}. Then, we can have the following corollary of Lemma 1:

COROLLARY 1. *Let G be a CSDF graph that is schedulable using a scheduling algorithm SA and any exact allocation algorithm on $\check{m}$ PEs, when the period of each actor A_i is equal to $s\check{T}_i$. G is schedulable using SA and any approximate allocation algorithm AA, with approximation ratio R_{AA}, on $\check{m}$ PEs, when the period of each actor A_i is equal to $sR_{AA}\check{T}_i$.*

4. PROBLEM FORMULATION

Now, we formally define our problem introduced in Sec. 1 as follows:

Problem 1. Given an SDF graph G, where the actors are scheduled as strictly periodic actors, and m available PEs. Suppose that each actor A_i in G is to be unfolded by an unfolding factor f_i. Find, for each actor A_i, the minimum value of f_i and the allocation of each replica $A_{i,f}$, where $1 \leq f \leq f_i$, such that the period of the sink actor T_{snk} in the unfolded graph is minimized.

If Problem 1 is considered as *primal*, we can have its equivalent *dual* problem defined as follows:

Problem 2. Given an SDF graph G, where the actors are scheduled as strictly periodic tasks, and m available PEs. Suppose that each actor A_i in G is to be unfolded by an unfolding factor f_i. Find, for each actor A_i, the minimum value of f_i and the allocation of each replica $A_{i,f}$, where $1 \leq f \leq f_i$, such that the total utilization $\sum_{A_{i,f} \in \mathcal{A}'} C_{i,f}/T_{i,f}$ of the unfolded graph on m PEs is maximized.

It can be seen that Problems 1 and 2 are not trivial. In general, for a given SDF graph, the number of possible alternative graphs that can be generated using unfolding grows exponentially as the number of actors increases. Furthermore, for each alternative graph, we have to perform allocation of unfolded actors which is by itself an NP-hard problem.

5. BOUNDING THE SOLUTION SPACE

In order to solve Problems 1 and 2 defined in Sec. 4, we need first to bound the solution space, i.e., to bound the values of the unfolding factors f_i. Bounding the solution space ensures that the algorithm devised in Sec. 6 terminates. We define the *upper bound* on unfolding factors as follows:

Definition 2. Let G be an SDF graph, where the actors in G are scheduled as strictly periodic actors, and assume that the number of PEs is unlimited. Suppose that every actor A_i in G is to be unfolded by a factor f_i resulting in a CSDF graph G', where $\check{T}_{i,f}$ is the minimum period of each replica $A_{i,f}$ and $C_{i,f} = C_i$ is its WCET. The *upper bound* on f_i, denoted by $\hat{f}_i$, is the minimum value which results in utilization $C_{i,f}/\check{T}_{i,f} = 1.0$ for each replica $A_{i,f}$ in G'.

In other words, unfolding an SDF graph G by a vector of unfolding factors $\vec{\hat{f}} = [\hat{f}_1, \cdots, \hat{f}_n]$ results in a graph G' with utilization $U_{G'} = n'$, where n' is the number of actors in the unfolded graph. Hence, unfolding any actor A_i by an unfolding

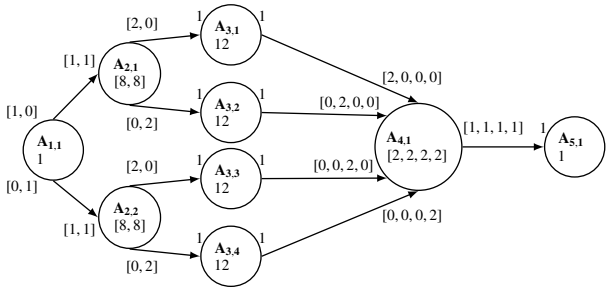


Figure 3: G_3: Optimal alternative graph of G_1 in Figure 1(a) with unfolding factors $f_2 = 2, f_3 = 4$ when scheduled on 2 PEs.

factor $f_i^* > \hat{f}_i$ cannot result in any increase in the total utilization of the unfolded graph. Moreover, the unfolded graph achieves the maximum achievable throughput since the sink actor fully utilizes the PE on which it executes. Therefore, $\vec{\hat{f}}$ defines the solution space that has an impact on the total utilization of the unfolded graph.

Determining the upper bound $\vec{\hat{f}}$ is not trivial. One common assumption, e.g., in [9] and [12], is to set $\vec{\hat{f}} = [m, m, \cdots, m]$, where m is the number of PEs. In this section, we show, using an example, that this assumption sometimes limits the solution space. As a consequence, the limited solution space might not contain the optimal solution to Problems 1 and 2.

Let us consider G_1 in Figure 1(a) and suppose that 2 PEs are available. The optimal alternative graph of G_1 is G_3, shown in Figure 3, when the vector of unfolding factors is $\vec{f} = [1, 2, 4, 1, 1]$. The repetition vector of G_3 can be computed according to Eq. 1 as $\vec{q'} = [\mathsf{r}(A_{1,1}), \mathsf{r}(A_{2,1}), \mathsf{r}(A_{2,2}), \mathsf{r}(A_{3,1}), \mathsf{r}(A_{3,2}), \mathsf{r}(A_{3,3}), \mathsf{r}(A_{3,4}), \mathsf{r}(A_{4,1}), \mathsf{r}(A_{5,1})]^T = [4, 2, 2, 2, 2, 2, 2, 4, 4]^T$. It follows that $\hat{W}_{G_3} = 2 \times 12$ and $\mathrm{lcm}(\vec{q'}) = 4$. Solving Eq. 2 yields the minimum period of the sink actor $A_{5,1}$ as $\check{T}_{\mathrm{snk}} = \frac{4}{4} \cdot \lceil \frac{24}{4} \rceil = 6$. To achieve $\check{T}_{\mathrm{snk}} = 6$, 6 PEs are required. Then, we can scale all periods of the actors in G_3 by $s = 3$, which yields a period $T_{\mathrm{snk}} = 3\check{T}_{\mathrm{snk}} = 18$. According to Lemma 1, the graph G_3 is schedulable on $\lceil \frac{6}{3} \rceil = 2$ PEs. After scaling the periods of all actors, the total utilization U_{G_3} of G_3 on 2 PEs is 2.0, thereby no shorter period can be achieved. Thus, G_3 is the optimal alternative graph of G_1 for 2 PEs with an unfolding factor $f_3 = 4$, which is greater than the number of PEs available. Therefore, this example shows that the optimal solution is beyond $\vec{\hat{f}} = [2, 2, 2, 2, 2]$, which defines the solution space if we set $\vec{\hat{f}} = [m, m, \cdots, m]$. Hence, we conclude that the upper bound on an unfolding factor is not necessarily equal to the number of PEs.

Now, we derive the upper bound on the unfolding factor for each actor in the initial SDF graph by stating the following theorem:

Theorem 1. *Given an SDF graph G, where the actors are scheduled as strictly periodic actors. Suppose that each actor A_i is to be unfolded by a factor f_i. The upper bound on f_i according to Definition 2 can be computed as follows:*

$$\hat{f}_i = \mathrm{lcm}\{x_1, x_2, \cdots, x_n\}/x_i, \tag{3}$$

where $x_i = \mathrm{lcm}\{W_1, W_2, \cdots, W_n\}/W_i$ (W_i is the workload of actor A_i given by Definition 1).

The proof of Theorem 1 is given in Appendix B.

Now, we give an example on how to compute $\vec{\hat{f}}$. For G_1 in Figure 1(a), $\vec{x}$ containing the values of x_i is given by $\vec{x} = [24, 3, 1, 12, 24]$. Then, we obtain $\mathrm{lcm}(\vec{x}) = 24$, and $\vec{\hat{f}} = [1, 8, 24, 2, 1]$.

6. THE ALGORITHM

Considering the upper bounds on unfolding factors $\vec{\hat{f}}$ derived in Sec. 5, we devise, in this section, an efficient algorithm to solve Problems 1 and 2 as defined in Sec. 4.

The algorithm accepts as an input the following: 1) the initial SDF graph G; 2) the number of available PEs m; 3) the vector containing the upper bounds on the unfolding factors $\vec{\hat{f}}$ computed using Eq. 3; and 4) a pre-specified quality factor $\rho \in (0, 1]$, which is used to terminate the algorithm. The outputs of the algorithm are: 1) a vector of unfolding factors that is the solution to Problems 1 and 2; 2) the allocation of the unfolded SDF graph on m PEs; 3) the minimum achievable period of the sink actor in the unfolded SDF graph on m PEs which is the objective of Problem 1; and 4) the maximum utilization of the unfolded SDF graph on m PEs which is the objective of Problem 2.

6.1 Algorithm Description

The algorithm builds, incrementally during its execution, a list of nodes in which each node represents a possible vector of unfolding factors $\vec{f}$. Initially, the list contains only a single node which corresponds to the given initial SDF graph with a vector of unfolding factors $\vec{f} = \vec{1}$. Then, we compute the minimum period of the sink actor T_{snk} in the initial SDF graph G, when G is allocated on m PEs, and its total utilization U_G. Both values initialize a tuple $(T_{\mathrm{best}}, U_{\mathrm{best}})$ which holds the period and total utilization of the current best solution. During the execution of the algorithm, new nodes are created and added to the list, where a node represents an alternative CSDF graph G' of the initial graph G with a vector $\vec{f}$ of unfolding factors. Each entry $f_i \in \vec{f}$ ranges from 1 up to $\hat{f}_i$ derived in Eq. 3.

A newly created node inherits from its previous node a copy of the unfolding factors vector $\vec{f}_{\mathrm{prev}}$ used by the previous node to generate the unfolded graph G'_{prev}. After that, we search in G'_{prev} for the bottleneck actor, denoted by $A_{b,f}$, which is the one with the maximum workload $\hat{W}_G$ as defined in Sec. 3.2. If multiple actors have the same maximum workload, then the one with the smallest code size is selected. Next, we increment by one the entry f_b in the inherited unfolding factors vector $\vec{f}_{\mathrm{prev}}$, thereby, obtaining $\vec{f}_{\mathrm{curr}}$. Then, we unfold the initial graph G by the factors in $\vec{f}_{\mathrm{curr}}$ which results in a CSDF graph G'_{curr}. The next step is to evaluate the unfolded graph G'_{curr} when it is allocated on m PEs. The procedure for evaluating G'_{curr} is explained in details in Sec. 6.2. The result of the evaluation procedure is the minimum period of the sink actor T_{snk} in G'_{curr}, when G'_{curr} is allocated on m PEs, and the total utilization of the graph U_{curr}. If the obtained U_{curr} is higher than U_{best} corresponding to the current best solution (i.e., T_{snk} smaller than T_{best}), then T_{best} and U_{best} are updated with T_{snk} and U_{curr}, respectively. Otherwise, T_{best} and U_{best} remain unchanged.

The creation of new nodes is terminated when one of the following conditions is met:

1. The total utilization $U_{G'}$ of the CSDF graph at the current node satisfies $U_{G'} \geq \rho m$, where $\rho \in (0, 1]$ is the quality factor given as an input to the algorithm. If $\rho = 1$, then this means that each PE is fully utilized, which means that no shorter period can be obtained.
2. The unfolding factor f_i of an actor A_i exceeds either its upper bound $\hat{f}_i$ if A_i is stateless, or 1 if A_i is stateful or a data source/sink actor. Recall that stateful actors together with the data source and sink actors cannot be unfolded.

After the creation of new nodes is terminated, we select the first node in the list that has a minimum sink period and a total graph utilization equal to T_{best} and U_{best}, respectively. The selected node contains the solution to Problems 1 and 2.

6.2 Evaluating the Unfolded Graphs

As explained in Sec. 6.1, at each node, the initial SDF graph G is unfolded to produce a CSDF graph $G' = \{\mathcal{A}', \mathcal{E}'\}$. Then, we compute two values for G': 1) the minimum sink actor period T_{snk} when G' is allocated on m PEs; and 2) its total utilization $U_{G'}$. In this section, we explain in details how these two values are computed. Recall from Sec. 3.2 that T_{snk} is given by $T_{\mathrm{snk}} = s\check{T}_{\mathrm{snk}}$, and $U_{G'}$ can be computed as follows:

$$U_{G'} = \sum_{A_{i,f} \in \mathcal{A}'} \frac{C_{i,f}}{s\check{T}_{i,f}}. \tag{4}$$

Recall also that the objective of Problem 2 is to maximize the utilization. Therefore, we need to find a value of scaling factor s, such that all actors in G' are schedulable on m PEs and $U_{G'}$ is maximized. To do so, we first bound the search range for s by deriving its lower and upper bounds. Using any allocation algorithm, we have from Lemma 1 a lower bound on s, denoted by $\check{s}$, as follows:

$$\check{s} = \left\lceil \frac{1}{m} \sum_{A_{i,f} \in \mathcal{A}'} \frac{C_{i,f}}{\check{T}_{i,f}} \right\rceil. \tag{5}$$

That is, for any AA, the scaling factor s cannot be smaller than $\check{s}$. From Corollary 1 in Sec. 3.2, we compute, using the approximation

Algorithm 1: The procedure for evaluating an unfolded graph.

Input: A CSDF graph G', number of available PEs m, and the period and total utilization corresponding to the current best solution T_{best} and U_{best}.
Result: *alloc* which is an m-partition describing the allocation of the actors in G' onto m PEs

1 Compute $\check{s}$ using Eq. 5 and $\hat{s}$ using Eq. 6 ;
2 **for** $s = \check{s}$ **to** $\hat{s}$ **do**
3 Compute the period $T_{i,f}$ of each actor $A_{i,f}$ as $T_{i,f} = s\check{T}_{i,f}$;
4 **if** $T_{snk} \geq T_{best}$ **then**
5 **return** $\emptyset$;
6 Compute the utilization $U_{G'}$ using Eq. 4;
7 Find an m'-partition of the actors in G', denoted by *alloc*, using the FFD algorithm and assuming the EDF scheduling algorithm;
8 **if** $m' \leq m$ **then**
9 $U_{\text{best}} = U_{G'}$, $T_{\text{best}} = T_{\text{snk}}$;
10 **return** *alloc* ;

ratio of the FFD allocation algorithm $R_{\text{FFD}} = 11/9$ given in Sec. 3.2, the upper bound on the scaling factor s, denoted by $\hat{s}$, as follows:

$$\hat{s} = \left\lceil \frac{11}{9m} \sum_{A_{i,f} \in \mathcal{A}'} \frac{C_{i,f}}{\check{T}_{i,f}} \right\rceil + 1. \quad (6)$$

Once the lower and upper bounds of s are found using Eq. (5) and Eq. (6), respectively, we perform a linear search to seek the smallest s, such that a CSDF graph G' is schedulable on m PEs. Specifically, we check if an m-partition of all actors in G' exists, assuming the EDF scheduling algorithm and the FFD allocation algorithm explained in Sec. 3.2. The complete procedure for evaluating the unfolded graphs is depicted in Algorithm 1. If the period resulting from a given scaling factor s is greater than T_{best}, then Algorithm 1 terminates immediately to speed-up the search (see line 4 in Algorithm 1).

6.3 Example

Now, we illustrate our algorithm using graph G_1 in Figure 1(a) and schedule it on 2 PEs (i.e., $m = 2$). Suppose that $\rho = 0.95$, i.e., the algorithm terminates when $U_{G'} \geq 0.95 \times 2 = 1.9$. The whole list produced by the algorithm is illustrated in Figure 4. The numbers inside the nodes correspond to the sequence in which the nodes are created. The algorithm starts with the initial G_1 in node 0 and computes the scaling factors $\check{s}$ and $\hat{s}$ which result in $U_{G_1} = 1.5$ and period $T_{\text{snk}} = 24$. At this point, U_{best} is initialized to 1.5 and T_{best} to 24. Node 1 inherits from node 0 a vector of unfolding factors equal to $[1, 1, 1, 1, 1]$. After that, we search in $G'_{\text{prev}} = G_1$ for the bottleneck actor which is A_3. Next, we increment f_3 in the inherited vector of unfolding factors at node 1 resulting in $\vec{f} = [1, 1, 2, 1, 1]$. Then, G' is generated and Algorithm 1 is invoked. Since U_{best} cannot be improved (see line 4 in Algorithm 1), the algorithm continues by creating node 2. At node 2, a new bottleneck actor $A_{2,1}$ is introduced. Therefore, at node 3, the unfolding factor f_2 is incremented by 1. Then, the algorithm continues to node 4, at which one termination criterion is met, namely $U_{G'} \geq 1.9$. As a result, $\vec{f} = [1, 2, 4, 1, 1]$ is the solution with $T_{\text{best}} = 18$ and $U_{\text{best}} = 2.0$.

7. EVALUATION

In this section, we present the results of evaluating our algorithm using a set of real-life streaming applications. We evaluate the algorithm by performing two experiments. In the first experiment, we run our algorithm on the applications and report the following: 1) the performance gain resulting from mapping the SDF graph unfolded using the unfolding factors obtained from our algorithm, compared to mapping the initial SDF graph without unfolding; and 2) the total time needed to execute our algorithm.

In the second experiment, we compare our proposed algorithm with one of the state-of-art search meta-heuristics, since problems 1 and 2 in general can be readily formulated and solved by these meta-heuristics, such as genetic algorithms, simulated annealing, etc. However, meta-heuristics normally require parameter tuning to achieve a good solution. In this work, we select a particular meta-heuristic, namely Genetic Algorithms (GA) for two reasons: 1) they have been applied by several researchers to solve similar problems (e.g., [17]), and 2) several researchers have reported the

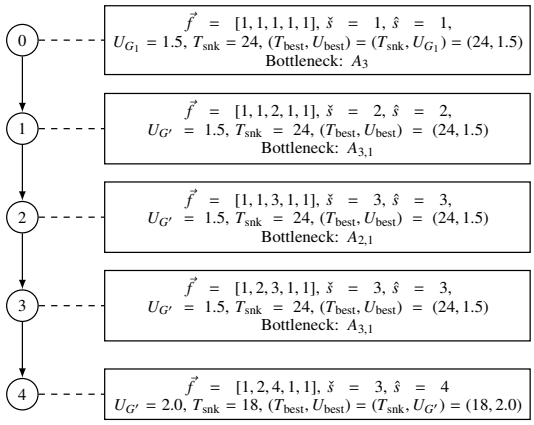


Figure 4: The list produced by the algorithm for G_1 in Figure 1(a) on 2 PEs.

optimal parameter settings for GA in the context of our problem (e.g., [19]). In particular, we compare our proposed algorithm with the NSGA-II genetic algorithm [4]. Specifically, we compare two metrics: 1) the total execution time needed by each algorithm to find a solution; and 2) the total code size of the returned solution.

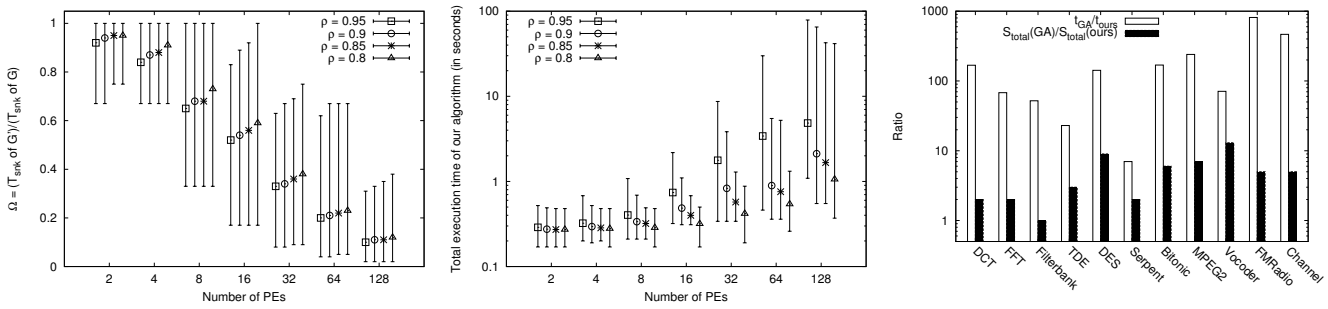
We conducted all experiments on 11 real-life streaming applications from the StreamIt benchmarks suite [9]. The exact characteristics of the benchmarks are outlined in Table 2 in Appendix D. The experiments were performed on an Intel Core 2 Duo T9600 CPU running at 2.80 GHz with Linux Kubuntu 10.4.

7.1 Evaluating the Proposed Algorithm

First, we present the performance gain resulting from mapping the unfolded SDF graph, compared to mapping the initial SDF graph without unfolding. We do so by running the algorithm on the benchmarks and mapping each application on a number of PEs that varies from 2 up to 128 PEs. We evaluate the trade-off between the performance gain and total execution time by setting different quality factors $\rho \in \{0.8, 0.85, 0.9, 0.95\}$. To measure the performance gain, we compute, for each benchmark, the ratio between the sink actor period resulting from mapping the unfolded SDF graph, and the period resulting from mapping the initial SDF. This ratio is denoted by Ω and is given by $\Omega = (T_{\text{snk}} \text{ of } G')/(T_{\text{snk}} \text{ of } G)$, where G' is the unfolded graph, and G is the initial SDF graph. A lower value of Ω indicates a shorter sink actor period in the unfolded graph, and therefore, a higher throughput. In Figure 5(a), each vertical line shows the variations in Ω for all the benchmarks. The marker at the middle of each vertical line represents the Geometric Mean (GM) of Ω, while the upper and lower ends of the line represent the maximum and minimum values of Ω, respectively. It can be seen that mapping the unfolded SDF graphs of the benchmarks achieves significant performance improvement compared to mapping the initial SDF graphs of the benchmarks. As the number of PEs increases, the unfolded SDF graphs utilize the PEs much better than the initial SDF graphs. For example, on 64 and 128 PEs, mapping the unfolded SDF graphs with quality factor $\rho = 0.95$ achieves a GM of Ω equal to 0.2 and 0.1, respectively. The DCT benchmark benefits significantly from the algorithm and achieves a GM of Ω equal to 0.021 and 0.042 on 128 and 64 PEs, respectively. Even when a small number of PEs is available, the unfolded SDF graphs still achieve, with quality factor $\rho = 0.95$, a GM of Ω equal to 0.92 and 0.85 on 2 and 4 PEs, respectively.

During the experiment, we also find that the unfolding factor of an actor, obtained using our algorithm, is not necessarily equal to the number of PEs. For example, the obtained unfolded SDF graph of the Vocoder benchmark, when mapped onto 8 PEs, requires the RectangularToPolar actor in the initial SDF graph to be unfolded by a factor of 20. This confirms our statement in Sec. 5.

We also evaluate the total execution time of our algorithm, denoted by t_{ours}, when it is invoked on the benchmarks. Figure 5(b) shows the total execution time of our algorithm in seconds for all the benchmarks. For all benchmarks, our algorithm takes a GM of 6.07 seconds for 128 PEs with utilization ratio $\rho = 0.95$. The Serpent benchmark (the largest graph size with 120 actors) takes the longest running time (78.90 seconds), while the DCT benchmarks takes the shortest running time (1.09 seconds). As the quality factor ρ is decreased from 0.95 to 0.9, the GM of the running time drops to 2.49 seconds for 128 PEs. These results show clearly that our



(a) Period ratio (lower is better)

(b) Running time of our algorithm

(c) The ratios of total execution time and total code size for the GA and our algorithm

Figure 5: (a and b) Results of evaluating our proposed algorithm and (c) comparing our algorithm vs. GA.

algorithm results, within a reasonable amount of time, in a large performance gain.

7.2 Comparison with Genetic Algorithm

To compare our algorithm with the GA-based heuristic, we perform the following steps. First, we run the GA to map each benchmark onto 64 PEs. It outputs an achievable period T and total utilization U_{GA}. Then, we run our algorithm to map the same benchmark onto 64 PEs with a termination criterion $U_{G'} \geq U_{\text{GA}}$. This criterion ensures a fair comparison since our algorithm runs till it finds the same or better solution in terms of the sink actor period and total utilization compared to the best solution found by the GA-based heuristic. Then, we compare two metrics: 1) the total execution time of each algorithm; and 2) the total code size resulting from the unfolding factors returned by each algorithm. The total code size is computed as $\sum_{A_{i,f} \in \mathcal{A}'} S_{i,f}$, where $S_{i,f}$ is the code size for actor $A_{i,f}$.

In this work, we use the NSGA-II implementation from the DEAP framework [6]. For the GA-based heuristic, each individual (also known as a chromosome) encodes a particular unfolding vector $\vec{f}$ of the initial SDF graph and the allocation of the replicas on m PEs. The structure of an individual is visualized in Figure 6. Basically, in an individual, each SDF actor A_i in the initial graph has $\hat{f}_i$ cells as derived in Eq. 3, indicating that A_i may have up to $\hat{f}_i$ replicas. Each cell may have a value varying from 0 up to m. A value of 0 denotes that the replica does not exist, while a value of 1 up to m denotes the PE on which the replica is allocated. Then, we formulate Problem 1 as a multi-objective optimization problem with two objectives. The first objective is to minimize the sink actor period, and the second one is to minimize the total code size of the unfolded graph. During the search, we use the evaluation function shown in Algorithm 3 in Appendix C. The GA outputs a set of Pareto points, for which we select the one with the shortest achievable period. In order to control the GA, we use the parameters reported in [19], because the target application domain and used platforms are similar to ours. The values of these parameters are given in Table 3 in Appendix D.

Figure 5(c) shows two ratios. The first ratio (shown in white bars) is the total execution time ratio given by $t_{\text{GA}}/t_{\text{ours}}$, where t_{GA} is the total time needed by the GA, and t_{ours} is the total time needed by our algorithm. The second ratio in Figure 5(c) (shown in black bars) is the total code size ratio given by $S_{\text{total}}(\text{GA})/S_{\text{total}}(\text{ours})$, where $S_{\text{total}}(\text{GA})$ is the total code size of the solution obtained using the GA, and $S_{\text{total}}(\text{ours})$ is the total code size of the solution obtained using our algorithm. Our algorithm is on average 104 times faster than the GA-based heuristic. For example, to unfold and map the FMRadio benchmark onto 64 PEs, our algorithm takes only 3 seconds, while the GA-based heuristic takes 2439 seconds. This means that our algorithm, for the FMRadio benchmark, is 813 times faster. We also see from Figure 5(c) that our algorithm results in less total code size compared to the GA-based heuristic. These results show clearly that our algorithm outperforms the GA-based heuristic in terms of: 1) the time needed to obtain the solution; and 2) the total code size of the obtained solution.

8. CONCLUSIONS

In this paper, we addressed the problem of exploiting just-enough parallelism when mapping a streaming application modeled using the SDF MoC in hard real-time systems. Exploiting just-enough parallelism is achieved by simultaneously unfolding and allocating the SDF actors onto an MPSoC platform, while considering the number of available PEs and hard real-time scheduling of actors on the PEs. We showed that the solution space to our problem is bounded and subsequently derived its upper bound. We devised an efficient algorithm to solve the problem and evaluated the algorithm on a set of real-life applications. The experiments showed that our algorithm results a system specification with large performance gain. We also compared our algorithm with one of the state-of-the-art meta-heuristics, i.e., NSGA-II genetic algorithm, and showed that our algorithm is on average 100 times faster than the GA, while achieving the same quality of the solution.

9. ACKNOWLEDGMENT

This work is supported by the Dutch STW NEtherlands STreaming (NEST) project and CATRENE/MEDEA+ TSAR project.

10. REFERENCES

[1] EU FP-7 parMERASA project. http://www.parmerasa.eu/.
[2] M. Bamakhrama and T. Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proc. EMSOFT*, 2011.
[3] G. Bilsen et al. Cyclo-static data flow. *IEEE Trans. Signal Process.*, 44:397–408, 1996.
[4] K. Deb et al. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, 6(2):182 –197, 2002.
[5] S. M. Farhad et al. Orchestration by approximation: mapping stream programs onto multicore architectures. In *Proc. ASPLOS*, 2011.
[6] F.-A. Fortin et al. DEAP: Evolutionary algorithms made easy. *J. Mach. Learn. Res.*, 2171–2175(13), 2012.
[7] A. Gerstlauer et al. Electronic system-level synthesis methodologies. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 28(10):1517–1530, 2009.
[8] K. Goossens et al. Æthereal network on chip: concepts, architectures, and implementations. *IEEE Des. Test. Comput.*, 22(5):414 – 421, 2005.
[9] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. ASPLOS*, 2006.
[10] A. Hagiescu et al. A computing origami: folding streams in FPGAs. In *Proc. DAC*, 2009.
[11] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of IFIP Congress*. 1974.
[12] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. PLDI*, 2008.
[13] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proc. RTSS*, 2010.
[14] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36:24–35, 1987.
[15] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.
[16] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1 edition, 1990.
[17] A. Stulova et al. Throughput driven transformations of synchronous data flows for mapping to heterogeneous MPSoCs. In *ICSAMOS*, 2012.
[18] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT*, 2010.
[19] M. Thompson. *Tools and techniques for efficient system-level design space exploration*. PhD thesis, University of Amsterdam, 2012.
[20] H. Yang and S. Ha. Pipelined data parallel task mapping/scheduling technique for MPSoC. In *Proc. DATE*, 2009.
[21] M. Yue. A simple proof of the inequality FFD (L) ≤11/9 OPT (L) + 1, ∀*L* for the FFD bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, 1991.
[22] J. T. Zhai, H. Nikolov, and T. Stefanov. Mapping of streaming applications considering alternative application specifications. *ACM Trans. Embed. Comput. Syst.*, 12:34:1–34:21, 2013.

APPENDIX

A. NOTATIONS

Table 1: Notations used in the paper.

$\mathbb{N}$	the set of natural numbers excluding zero
$\check{x}$	lower bound (minimum) of a value x
$\hat{x}$	upper bound (maximum) of a value x
lcm	least common multiple
$\lceil x \rceil$	smallest integer that is greater or equal to x
A_i	ith actor, where $1 \leq i \leq n$
C_i	worst-case execution time of the actor A_i (equivalent to μ_i in [2])
f_i	unfolding factor for actor A_i
G	a (C)SDF graph, $G = \{\mathcal{A}, \mathcal{E}\}$
m	number of PEs
n	number of actors in a (C)SDF graph
$\mathsf{r}(A_i)$	repetition of actor A_i in one graph iteration
ρ	quality factor $\rho \in (0, 1]$
s	scaling factor for periods of all actors in a (C)SDF graph
S_i	code size of actor A_i
T_i	period of actor A_i (equivalent to λ_i in [2])
u_i	utilization factor of actor A_i, $u_i = \frac{C_i}{T_i}$
U_G	total utilization of (C)SDF graph G, $U_G = \sum_{A_i \in \mathcal{A}} C_i/T_i$
W_i	workload of actor A_i per graph iteration, $W_i = \mathsf{r}(A_i) \cdot C_i$

B. PROOFS

Proof. (of Lemma 1) Let U_{SA} be the utilization bound of a scheduling algorithm SA. If G is schedulable on $\check{m}$ PEs using SA and any AA, then this means that the total utilization of the actors on each PE j, where $1 \leq j \leq \check{m}$, is $U_{\mathrm{PE}_j} \in (0, U_{SA}]$. If we scale the periods of the actors in G by s, then this means that $U_{\mathrm{PE}_j} \in (0, \frac{U_{SA}}{s}]$. Therefore, it is possible to combine the actors in every s PEs into 1 PE. Hence, the number of PEs needed after scaling the periods is $\lceil \frac{\check{m}}{s} \rceil$. □

Proof. (of Theorem 1) Suppose that G' is the CSDF graph obtained by unfolding each actor A_i in the initial SDF graph G by $\hat{f}_i$. From Definition 2, it follows that every replica $A_{i,f}$ in G' has $\check{T}_{i,f} = C_{i,f} = C_i$. Therefore, we can re-write Eq. 2 as:

$$C_i = \frac{\mathrm{lcm}(\vec{q'})}{\mathsf{r}(A_{i,f})} \left\lceil \frac{\hat{W}_{G'}}{\mathrm{lcm}(\vec{q'})} \right\rceil \tag{7}$$

where $\mathsf{r}(A_{i,f})$ is the repetition of $A_{i,f}$ in G'. Eq. 7 can be re-written as:

$$\mathsf{r}(A_{i,f}) C_i = \mathrm{lcm}(\vec{q'}) \left\lceil \frac{\hat{W}_{G'}}{\mathrm{lcm}(\vec{q'})} \right\rceil \tag{8}$$

Since $\mathrm{lcm}(\vec{q'}) \lceil \hat{W}_{G'} / \mathrm{lcm}(\vec{q'}) \rceil$ is constant, then we re-write Eq. 8 as:

$$\mathsf{r}(A_{1,1})C_1 = \mathsf{r}(A_{1,2})C_1 = ... = \mathsf{r}(A_{1,f_1})C_1 = ... = \mathsf{r}(A_{n,f_n})C_n \tag{9}$$

Now, we can write $\mathsf{r}(A_{i,f}) = x_i \cdot \mathsf{r}(A_i)$, where $\mathsf{r}(A_i)$ is the repetition of A_i in the initial SDF graph and x_i is an integer factor. That is:

$$x_1 \mathsf{r}(A_1) C_1 = x_2 \mathsf{r}(A_2) C_2 = \cdots = x_n \mathsf{r}(A_n) C_n \tag{10}$$

Eq. 10 can be re-written as:

$$x_1 W_1 = x_2 W_2 = \cdots = x_n W_n \tag{11}$$

where W_i is the workload of actor A_i according to Definition 1. The minimum solution to Eq. 11 is:

$$x_i = \mathrm{lcm}\{W_1, W_2, \cdots, W_n\}/W_i \tag{12}$$

Since $\mathsf{r}(A_{i,f}) = x_i \mathsf{r}(A_i)$ and the graph is unfolded by $\vec{\hat{f}}$, we can substitute this in Eq. 1 to get:

$$x_i \mathsf{r}(A_i) = \frac{\mathsf{r}(A_i)\,\mathrm{lcm}(\vec{\hat{f}})}{\hat{f}_i} \tag{13}$$

which can be re-written as:

$$x_i \hat{f}_i = \mathrm{lcm}(\vec{\hat{f}}) \tag{14}$$

Since $\mathrm{lcm}(\vec{\hat{f}})$ is constant, then Eq. 14 can be re-written as:

$$x_1 \hat{f}_1 = x_2 \hat{f}_2 = \cdots = x_n \hat{f}_n \tag{15}$$

The minimum solution to Eq. 15 is:

$$\hat{f}_i = \frac{\mathrm{lcm}\{x_1, x_2, \cdots, x_n\}}{x_i} \tag{16}$$

□

C. ALGORITHMS

Algorithm 2: Unfolding an SDF graph.

Input: An SDF graph $G = \{\mathcal{A}, \mathcal{E}\}$ with a vector $\vec{f}$ of unfolding factors.
Result: The equivalent CSDF graph $G' = \{\mathcal{A}', \mathcal{E}'\}$

$\mathcal{A}' = \emptyset, \mathcal{E}' = \emptyset$;
foreach $A_i \in \mathcal{A}$ **do**
  Add $f_i \in \vec{f}$ replicas of A_i to $\mathcal{A}'$;
  Set repetition entry $\mathsf{r}(A_{i,ii}) = \frac{\mathsf{r}(A_i) \cdot \mathrm{lcm}(\vec{f})}{f_i}, \forall ii \in [1, f_i]$;
foreach $E \in \mathcal{E}$ **do**
  Get source actor A_i and sink actor A_j of edge E ;
  Get production rate $prd(E)$ and consumption rate $cns(E)$;
  $lcm\_pc = \mathrm{lcm}(prd(E), cns(E))$;
  if f_j *is dividable by* f_i **then** $OP = f_j/f_i; IP = 1;$
  else if f_i *is dividable by* f_j **then** $IP = f_i/f_j; OP = 1;$
  else $IP = f_i/f_j; OP = 1;$
  for $ii = 1$ **to** f_i **do**
    Add OP output ports to $A_{i,ii}$;
    for $k = 1$ **to** OP **do**
      Initialize a production sequence $\mathcal{P}_{i,ii}$ of length $\mathsf{r}(A_{i,ii})$ to 0;
      $\mathcal{P}_{i,ii}[p] = prd(E), \forall p \in [(k-1)\frac{lcm\_pc}{prd(E)} + 1, k\frac{lcm\_pc}{prd(E)}]$;
      if f_j *is dividable by* f_i **then** $jj = (ii-1)OP + k$;
      else if f_i *is dividable by* f_j **then** $jj = ii/IP$;
      else $jj = k$;
      Initialize a consumption sequence $C_{j,jj}$ of length $\mathsf{r}(A_{j,jj})$ to 0;
      $C_{j,jj}[c] = cns(E), \forall c \in [(ii-1)\frac{lcm\_pc}{cns(E)} + 1, ii\frac{lcm\_pc}{cns(E)}]$;
      Create a new channel E' connecting replica $A_{i,ii}$ to replica $A_{j,jj}$;
      Add channel E' to $\mathcal{E}'$;
Compact the production and consumption sequences of each actor in $\mathcal{A}'$;

Algorithm 3: Evaluation function in the GA-based meta-heuristic

Input: An individual to be evaluated
Result: An achievable period and total code size.

Check if the given individual is valid ;
if *the individual is invalid* **then return** $(-1, -1)$;
Build the vector of unfolding factors $\vec{f}$ from the individual ;
Generate the CSDF graph G' by unfolding G with $\vec{f}$ using Algorithm 2;
Compute the minimum achievable period $\check{T}_{i,f}$ of each actor $A_{i,f}$ using to Eq. 2 ;
Compute $\check{s}$ according to Eq. 5 ;
$s = \check{s}$;
while *true* **do**
  Compute the period $T_{i,f}$ of each actor $A_{i,f}$ as $T_{i,f} = s\check{T}_{i,f}$;
  if G' *is schedulable on m PEs* **then**
    Compute total code size $S_{total} = \sum_{A_{i,f} \in \mathcal{A}'} S_{i,f}$;
    Get the period T_{snk} of the sink actor in G' ;
    return (T_{snk}, S_{total}) ;
  else
    $s = s + 1$;

D. EXPERIMENTS

Table 2: Benchmark characteristics.

Benchmark	Num. of Actors	Num. of Edges	Has Stateful Actors?
DCT	8	7	No
FFT	17	16	No
Filterbank	85	99	No
TDE	29	28	No
DES	53	60	No
Serpent	120	128	No
Bitonic	40	46	No
MPEG2	23	26	Yes
Vocoder	114	147	Yes
FMRadio	43	53	No
Channel	55	70	No

$A_{1,1}$	...	$A_{1,\hat{f}_1}$	...	$A_{n,1}$	...	$A_{n,\hat{f}_n}$
j	...	0	...	1	...	2

Figure 6: An example of an individual. The first replica of A_1 is allocated on the jth PE and the $\hat{f}_1$th replica of A_1 does not exist.

Table 3: Parameters for the genetic algorithm.

Parameter	Recommended value in [19]
Population size	80
Number of generations	300
Crossover rate	0.9
Mutation rate	0.05
Mating rate	0.1