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 which can be exploited using Multi-Processor System-on-Chip (SDF) [14] are two prominent parallel MoCs. In such MoCs, the 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 exploiting 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 the 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
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 PEIs 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 PEIs, 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 the exact solutions to minimize makespan on any PE while simultaneously unfolding actors in an SDF graph and allocating them on PEIs. In the ILP formulation, an unfolding factor of an actor cannot exceed the number of available PEIs. 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 PEIs (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 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 this paper, we assume that a given SDF graph is acyclic. In the context of synthesizing an SDF graph using dedicated hardware, the authors in [10] also determine which actors to unfold empirically shown in [18] that around 90% of streaming applications as 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 \( r(A_i) \in \vec{q} \) denotes how many times an actor \( A_i \) has to be executed in every graph generation of \( G \). Additionally, for each actor \( A_i \), we associate a Worst-Case Execution Time (WCET) \( T_i \) 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 \( A_2 \) 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., \( C_1 = 12 \) for actor \( A_1 \).

The unfolding operation on an SDF graph used in this paper is simple and its code size \( S_i \). It has been shown in [14] that, for a valid periodic schedule for an SDF graph, a non-trivial repetition vector \( \vec{q} \). To ensure the functional equivalence, 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 \( f \)-th replica of \( A_i \) with replication \( r(A_i) \) given by:

\[
\tau(A_i) = \frac{r(A_i) \cdot \text{lcm}(\vec{f})}{\vec{f}},
\]

where \( \tau(A_i) \) is the repetition of actor \( A_i \) in the initial SDF graph and \( \text{lcm}(\vec{f}) \) denotes the least common multiple of \( f_i \) in \( \vec{f} \). It follows that the repetition vector of \( G' \), denoted by \( \vec{q}' \), is \( \vec{q}' = [n, \tau(A_1), \ldots, \tau(A_5)]^T \). After obtaining \( \vec{q}' \) by 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 \( A_1, A_2 \), and \( A_3, A_4, A_5 \) for actor \( A_1 \) in \( G_1 \). The unfolding results in a repetition vector of \( G_2 \) as \( \vec{q}' = [\tau(A_1), \tau(A_2), \tau(A_3), \tau(A_4), \tau(A_5)]^T = [3, 3, 2, 2, 3, 3]^T \). For example, SDF actor \( A_1 \) executes only

\[
\tau(A_1) = \frac{r(A_1) \cdot \text{lcm}(\vec{f})}{\vec{f}},
\]

where \( \tau(A_i) \) is the repetition of actor \( A_i \) in the initial SDF graph and \( \text{lcm}(\vec{f}) \) denotes the least common multiple of \( f_i \) in \( \vec{f} \). It follows that the repetition vector of \( G' \), denoted by \( \vec{q}' \), is \( \vec{q}' = [n, \tau(A_1), \ldots, \tau(A_5)]^T \). After obtaining \( \vec{q}' \) by 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 \( A_1, A_2 \), and \( A_3, A_4, A_5 \) for actor \( A_1 \) in \( G_1 \). The unfolding results in a repetition vector of \( G_2 \) as \( \vec{q}' = [\tau(A_1), \tau(A_2), \tau(A_3), \tau(A_4), \tau(A_5)]^T = [3, 3, 2, 2, 3, 3]^T \). For example, SDF actor \( A_1 \) executes only

\[
\tau(A_1) = \frac{r(A_1) \cdot \text{lcm}(\vec{f})}{\vec{f}},
\]

where \( \tau(A_i) \) is the repetition of actor \( A_i \) in the initial SDF graph and \( \text{lcm}(\vec{f}) \) denotes the least common multiple of \( f_i \) in \( \vec{f} \). It follows that the repetition vector of \( G' \), denoted by \( \vec{q}' \), is \( \vec{q}' = [n, \tau(A_1), \ldots, \tau(A_5)]^T \). After obtaining \( \vec{q}' \) by 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 \( A_1, A_2 \), and \( A_3, A_4, A_5 \) for actor \( A_1 \) in \( G_1 \). The unfolding results in a repetition vector of \( G_2 \) as \( \vec{q}' = [\tau(A_1), \tau(A_2), \tau(A_3), \tau(A_4), \tau(A_5)]^T = [3, 3, 2, 2, 3, 3]^T \). For example, SDF actor \( A_1 \) executes only

\[
\tau(A_1) = \frac{r(A_1) \cdot \text{lcm}(\vec{f})}{\vec{f}},
\]

where \( \tau(A_i) \) is the repetition of actor \( A_i \) in the initial SDF graph and \( \text{lcm}(\vec{f}) \) denotes the least common multiple of \( f_i \) in \( \vec{f} \). It follows that the repetition vector of \( G' \), denoted by \( \vec{q}' \), is \( \vec{q}' = [n, \tau(A_1), \ldots, \tau(A_5)]^T \). After obtaining \( \vec{q}' \) by Eq. 1, production/consumption sequences of each CSDF actor are generated accordingly.
once \( (r(A_1) = 1) \) in \( G_1 \), per graph iteration, while executing three times \( (r(A_{s1}) = 3) \) in \( G_2 \) per graph iteration. Three consumption sequences of actor \( A_{s1} \) in \( G_2 \) behave similar to a joiner, with which \( A_{s1} \) collects data tokens from the three replicas \( A_{s1}, A_{s2} \) and \( A_{s3} \). Analogous to a splitter, actor \( A_{s2} \) 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), 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.** The workload of a (C)SDF actor \( A_i \) per graph iteration, denoted by \( W_i \), is given by \( W_i = r(A_i)C_i \), where \( r(A_i) \) 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 \( \hat{W}_G = \max_{A_i \in A}(r(A_i)C_i) \).

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:

\[
\check{T}_i = \frac{\text{lcm}(\hat{W}_G, \check{r}(A_i))}{\text{lcm}(\hat{q})}, \tag{2}
\]

where \( \text{lcm}(q) \) is the least common multiple of all repetition entries in \( q \).

The actual period \( T_i \) of an actor \( A_i \) is given by \( T_i = \check{T}_i \).

The throughput of each actor \( A_i \), \( \hat{u}_i \), is given in [2] as follows:

\[
\hat{u}_i = \frac{\hat{r}(A_i)}{\hat{C}_i}, \tag{3}
\]

where \( \hat{r}(A_i) \) is the unfolding factor of the actor \( A_i \).

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 approximate allocation algorithm on \( n \) PEs, when the period of each actor \( A_i \) is equal to \( T_i \). \( G \) is schedulable using the same SA and AA on \( \lceil \frac{T_i}{\check{T}_i} \rceil \) PEs, when the period of each actor \( A_i \) is equal to \( \check{T}_i \).

The proof of Lemma 1 can be found in Appendix B. Considering \( G_2 \) in Figure 1(b), it can be scheduled on \( \lceil \frac{T_i}{\check{T}_i} \rceil = \lceil 3 \check{T}_i \rceil \) PEs due to space limits.

**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 \( n \) 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 \( A_i \), where \( 1 \leq f_i \leq f_i \), such that the period of the sink actor \( T_{\text{sink}} \) 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 \( A_i \), where \( 1 \leq f_i \leq f_i \), such that the total utilization \( \sum_{A_i \in A} C_i f_i \) 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 graph 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 \( G'(f_i) \), where \( T_i \) is the minimum period of each replica \( A_i \), and \( C_i \) is its WCET. The upper bound on \( f_i \) denoted by \( \bar{f}_i \), is the minimum value which results in utilization \( C_i T_i / f_i \leq 1.0 \) for each replica \( A_i \) in \( G' \).

In other words, unfolding an SDF graph \( G \) by a vector of unfolding factors \( f = \{ f_1, \ldots, f_n \} \) 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
factor $\vec{f}^* > \vec{f}$ 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, $\vec{f}$ defines the solution space that has an impact on the total utilization of the unfolded graph.

Determining the upper bound $\vec{f}$ is not trivial. One common assumption, e.g., in [9] and [12], is to set $\vec{f} = \{m, m, \ldots, m\}$, 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.

To consider $G_1$ in Figure 1(a) and suppose that 2 PEs are available. The optimal alternative graph of $G_1$ is $G_{2,1}$, shown in Figure 3, when the vector of unfolding factors is $\vec{f} = \{1, 2, 2, 1, 1\}$. The repetition vector of $G_1$ can be computed according to Eq. 1 as $\vec{r} = \{r(A_{1,1}), r(A_{2,1}), r(A_{2,2}), r(A_{3,1}), r(A_{3,2}), r(A_{3,3}), r(A_{4,1}), r(A_{4,2})\} = \{4, 2, 2, 2, 2, 2, 4, 4\}$. It follows that $\vec{T}_{\text{curr}} = 2 \times 12$ and $\vec{lcm}(\vec{r}) = 4$. Solving Eq. 2 yields the minimum period of the sink actor $\vec{T}_{\text{min}} \geq \frac{4}{2} \geq 6$. To achieve $\vec{T}_{\text{best}} = 6$, 6 PEs are required. Then, we can scale all periods of the actors in $G_1$ by $x = 3$, which yields a period $\vec{T}_{\text{sink}} = 3 \times 6 = 18$. According to Lemma 1, the graph $G_3$ is schedulable on $\frac{\vec{T}}{2} = 2$ 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 $\vec{f} = 4$, which is greater than the number of PEs available. Therefore, this example shows that the optimal solution is beyond $\vec{f} = \{2, 2, 2, 2\}$, which defines the solution space if we set $\vec{f} = \{m, m, \ldots, m\}$. 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:

$$\hat{f}_i = \text{lcm}(x_1, x_2, \ldots, x_n)/x_i,$$

where $x_i = \text{lcm}(W_i, W_{i-1}, \ldots, W_1)/W_i$ (when $W_i$ 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 $\hat{f}$. 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 $\vec{lcm}(\vec{x}) = 24$, and $\vec{f} = [1, 8, 24, 2, 1]$.

6. The Algorithm

Considering the upper bounds on unfolding factors $\vec{f}$ 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 $\vec{f}$ computed using Eq. 3; and 4) a pre-specified quality factor $q \in (0, 1]$, 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 $T_{\text{sink}}$ in the initial SDF graph $G$, when $G$ is allocated on $m$ PEs, and its total utilization $U_{G}$. Both values initialize a tuple $(T_{\text{best}}, U_{\text{best}})$ 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 $\vec{f}_{\text{prev}}$ used by the previous node to generate the unfolded graph $G'_{\text{prev}}$. After that, we search in $G'_{\text{prev}}$ for the bottleneck actor, denoted by $A_{\text{sink}}$, which is the one with the maximum workload $W_{\text{sink}}$ 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 $f_i$ in the inherited unfolding factors vector $\vec{f}_{\text{prev}}$, thereby, obtaining $G'_{\text{curr}}$. Then, we unfold the initial graph $G$ by the factors in $\vec{f}_{\text{curr}}$, which results in a CSDF graph $G'_{\text{curr}}$. The next step is to evaluate the unfolded graph $G'_{\text{curr}}$ when it is allocated on $m$ PEs. The procedure for evaluating $G'_{\text{curr}}$ is explained in details in Sec. 6.2. The result of the evaluation procedure is the minimum period of the sink actor $T_{\text{sink}}$ in $G'_{\text{curr}}$, when $G'_{\text{curr}}$ is allocated on $m$ PEs, and the total utilization of the graph $U_{G_{\text{curr}}}$. If the obtained $U_{G_{\text{curr}}}$ is higher than $U_{\text{best}}$ corresponding to the current best solution (i.e. $T_{\text{sink}}$ smaller than $T_{\text{best}}$), then $T_{\text{best}}$ and $U_{\text{best}}$ are updated with $T_{\text{sink}}$ and $U_{G_{\text{curr}}}$, respectively. Otherwise, $T_{\text{best}}$ and $U_{\text{best}}$ 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 $U_G \geq q\, pm$, where $p \in [0, 1]$ is the quality factor given as an input to the algorithm. If $p = 1$, 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/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 $T_{\text{best}}$ and $U_{\text{best}}$, 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 $G' = \{\vec{f}, G'\}$. Then, we compute two values for $G'$: 1) the minimum sink actor period $T_{\text{sink}}$ 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 $T_{\text{sink}}$ is given by $T_{\text{sink}} = \frac{\vec{T}_{\text{sink}}}{\vec{f}_{\text{sink}}}$, and $U_{G'}$ can be computed as follows:

$$U_{G'} = \sum_{A_i \in \vec{G}} \frac{C_{i,j}}{\vec{f}_{\text{sink}}} \tag{4}$$

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 $\bar{s}$, as follows:

$$\bar{s} = \left\lceil \frac{1}{\min_{A_i \in \vec{G}} \frac{C_{i,j}}{\vec{f}_{\text{sink}}}} \right\rceil. \tag{5}$$

That is, for any $A_i$, the scaling factor $s$ cannot be smaller than $\bar{s}$. 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 $\delta$ using Eq. 5 and $\delta$ using Eq. 6;
2. for $s = 3$ to $\delta$ do
   3. Compute the period $T_{f,i}$ of each actor $A_{f,i}$ as $T_{f,i} = sT_{f}$;
   4. if $T_{f,i} \geq T_{best}$ then
      5. return $0$;
   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. $U_{best} = U_{G'}, T_{best} = T_{f,i}$;
   10. return $alloc$;

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

$$\delta = \left\lfloor \frac{11}{9m} \sum_{A_{f,i} \in G'} \frac{C_{f,i}}{T_{f,i}} \right\rfloor + 1. \quad (6)$$

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 $\delta$ and $\delta$ which result in $U_{G_1} = 1.5$ and period $T_{best} = 24$. 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 $\{1, 1, 1, 1\}$. After that, we search $G_{2}^{\text{prev}} = G_1$ 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 $f' = \{1, 1, 2, 1, 1\}$. 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 $A_{2,3}$ 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, $f' = \{1, 2, 4, 1, 1\}$ is the solution with $T_{best} = 18$ and $U_{best} = 2.0$.

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 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 $\Omega$ and is given by $\Omega = (T_{sink}/G')/(T_{sink}/G)$, where $G'$ is the unfolded graph, and $G$ is the initial SDF graph. A lower value of $\Omega$ 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 $\Omega$ for all the benchmarks. The marker at the middle of each vertical line represents the Geometric Mean (GM) of $\Omega$, while the upper and lower ends of the line represent the maximum and minimum values of $\Omega$, 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 $\Omega$ equal to 0.2 and 0.1, respectively. The DCT benchmark benefits significantly from the algorithm and achieves a GM of $\Omega$ 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 $\Omega$ 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 Wavefront benchmark, when mapped onto 8 PEs requires the $\text{RectangularToPolular}$ 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 $t$, when it is applied 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 $\text{Säverät}$ 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 $\rho$ 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
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 $U_C \geq U_{GA}$. 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 $\sum_{i \in \mathcal{C}} S_i \cdot f_i$, where $S_i$ is the code size for actor $A_i$, and $f_i$ is the factor which the GA encodes a particular unfolding vector.

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 $f$ 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 $A$, in the initial graph has $f_i$ cells as derived in Eq. 3, indicating that $A_i$ may have up to $f_i$ 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 $f_{ours}/f_{GA}$, where $f_{ours}$ is the total time needed by the GA, and $f_{GA}$ 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 $S_{ours}/S_{GA}$, where $S_{ours}$ is the total code size of the solution obtained using the GA, and $S_{ours}$ 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

APPENDIX

A. NOTATIONS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{N}$</td>
<td>the set of natural numbers excluding zero</td>
</tr>
<tr>
<td>$\delta$</td>
<td>lower bound (minimum) of a value $x$</td>
</tr>
<tr>
<td>$\tilde{x}$</td>
<td>upper bound (maximum) of a value $x$</td>
</tr>
<tr>
<td>$\text{lcm}$</td>
<td>least common multiple</td>
</tr>
<tr>
<td>$[x]$</td>
<td>smallest integer that is greater or equal to $x$</td>
</tr>
<tr>
<td>$A_i$</td>
<td>$i$th actor, where $1 \leq i \leq n$</td>
</tr>
<tr>
<td>$C_i$</td>
<td>worst-case execution time of the actor $A_i$ (equivalent to $\mu_i$ in [2])</td>
</tr>
<tr>
<td>$f_i$</td>
<td>unfolding factor for actor $A_i$</td>
</tr>
<tr>
<td>$G$</td>
<td>a (C)SDF graph, $G = ([A], E)$</td>
</tr>
<tr>
<td>$m$</td>
<td>number of PEs</td>
</tr>
<tr>
<td>$n$</td>
<td>number of actors in a (C)SDF graph</td>
</tr>
<tr>
<td>$\tau(A_i)$</td>
<td>repetition of actor $A_i$ in one graph iteration</td>
</tr>
<tr>
<td>$\rho$</td>
<td>quality factor $\rho \in (0, 1]$</td>
</tr>
<tr>
<td>$s$</td>
<td>scaling factor for periods of all actors in a (C)SDF graph</td>
</tr>
<tr>
<td>$S_i$</td>
<td>code size of actor $A_i$</td>
</tr>
<tr>
<td>$T_i$</td>
<td>period of actor $A_i$. (equivalent to $A_i$ in [2])</td>
</tr>
<tr>
<td>$u_i$</td>
<td>utilization factor of actor $A_i$, $u_1 = \frac{s}{T_i}$</td>
</tr>
<tr>
<td>$U_G$</td>
<td>total utilization of (C)SDF graph $G$</td>
</tr>
<tr>
<td>$W_i$</td>
<td>workload of actor $A_i$ per graph iteration, $W_i = \tau(A_i) \cdot C_i$</td>
</tr>
</tbody>
</table>

B. PROOFS

Proof (of Lemma 1) Let $U_{R_i}$ be the utilization bound of a scheduling algorithm $A_i$. If $G$ is schedulable on $R$ PEs using $SA$ and any $AA$, then this means that the total utilization of the actors on each PE $j$, where $1 \leq j \leq n$, is $U_{R_j} \in (0, U_{R_i})$. If we scale the periods of the actors in $G$ by $s$, then this means that $U_{R_j} \in (0, \frac{s}{T_i})$. 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 $\left\lceil \frac{s}{T_i} \right\rceil$.

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 $\tilde{s}$. From Definition 2, it follows that every replica $A_{i,j}$ in $G'$ has $T_{i,j} = C_{i,j} = C_i$. Therefore, we can write Eq. 2 as:

$$C_i = \frac{\text{lcm}(q_i)}{r(A_i)} \cdot \frac{W'_{G_i}}{\text{lcm}(q_i)}$$

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

$$r(A_{i,j})C_i = \text{lcm}(q_i) \cdot \frac{W'_{G_i}}{\text{lcm}(q_i)}$$

Since $\text{lcm}(q_i) [W'_{G_i} / \text{lcm}(q_i)]$ is constant, then we re-write Eq. 8 as:

$$r(A_{i,j})C_i = \text{lcm}(q_i) \cdot \frac{W'_{G_i}}{\text{lcm}(q_i)}$$

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

$$x_i \tau(A_i)C_i = x_i \tau(A_2)C_2 = \ldots = x_i \tau(A_n)C_n$$

Eq. 10 can be re-written as:

$$x_i W_1 = x_2 W_2 = \ldots = x_n W_n$$

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

$$x_i = \text{lcm}(W_1, W_2, \ldots, W_n)/W_1$$

Since $r(A_{i,j}) = x_i \tau(A_i)$ and the graph is unfolded by $\tilde{s}$, we can substitute this in Eq. 1 to get:

$$x_i \tilde{s} = \frac{r(A_i) \text{lcm}(\tilde{s})}{\tilde{s}}$$

which can be re-written as:

$$x_i \tilde{s} = \text{lcm}(\tilde{s})$$

Since $\text{lcm}(\tilde{s})$ is constant, then Eq. 14 can be re-written as:

$$x_1 \tilde{s}_1 = x_2 \tilde{s}_2 = \ldots = x_n \tilde{s}_n$$

The minimum solution to Eq. 15 is:

$$\tilde{s}_i = \frac{\text{lcm}(x_1, x_2, \ldots, x_n)}{x_i}$$

C. ALGORITHMS

Algorithm 2: Unfolding an SDF graph.

Input: An SDF graph $G = ([A], E)$ with a vector $\tilde{s}$ of unfolding factors.
Result: The equivalent CSDF graph $G' = ([A'], E')$.

1. $R = 0$, $E' = 0$.
2. foreach $A_i \in A$ do
3.   Add $f_i \in \tilde{s}$ replicas of $A_i$ to $E'$.
4.   Set repetition entry $r(A_i) = \frac{\text{lcm}(s_i)}{\tilde{s}_i}$, $\forall i \in [1, f_i]$.
5. endforeach $E \in E$ do
6.   Get source actor $A_i$ and sink actor $A_j$ of edge $E$.
7.   Get production rate $pr(E)$ and consumption rate $cn(E)$.
8.   $\text{lcm}_{pc} = \text{lcm}(pr(E), cn(E))$.
9.   if $f_i$ is divisible by $f_j$ then $OP = f_j/c_i, IP = 1$;
10. else if $f_i$ is divisible by $f_j$ then $IP = f_j/c_i, OP = 1$;
11. else $IP = f_j/c_i, OP = 1$.
12. for $ii = 1$ to $f_i$ do
13.   Add OP output ports to $A_{i,j}$;
14. for $ii = 1$ to OP do
15.   Initialize a production sequence $P_{ii}$ of length $r(A_{i,j})$ to 0;
16.   $P_{ii}[p] = pr(E), \forall p \in [k-1, \text{lcm}_{pc} + 1, \frac{\text{lcm}_{pc}}{OP}]$;
17.   if $f_i$ is divisible by $f_j$ then $jj = (i-1)OP + k$;
18. else if $f_i$ is divisible by $f_j$ then $jj = ii/IP$;
19. else $jj = k$.
20. Initialize a consumption sequence $C_{i,j}[[j] + 1, \text{lcm}_{pc}]$ to 0;
21. $C_{i,j} = \text{cn}(E), \forall c \in [k-1, \text{lcm}_{pc} + 1, \frac{\text{lcm}_{pc}}{IP}]$;
22. Create a new channel $E'$ for replica $A_{i,j}$. to replica $A_{i,j}$;
23. Add channel $E'$ to $E'$;
24. Compact the production and consumption sequences of each actor in $R$.

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

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

1. Check if the given individual is valid.
2. if the individual is invalid then return $(-1, -1)$.
3. Build the vector of unfolding factors $\tilde{s}$ from the individual.
4. Generate the CSDF graph $G'$ by unfolding $G$ with $\tilde{s}$ using Algorithm 2.
5. Compute the minimum achievable period $T_{i,j}$ of each actor $A_{i,j}$ using to Eq. 2.
6. Compute $f$ according to Eq. 5.
7. $s = \frac{\tilde{s}}{f}$.
8. while true do
9.   Compute the period $T_{i,j}$ of each actor $A_{i,j}$ as $T_{i,j} = sT_{i,j}$.
10. if $G'$ is schedulable on $m$ PEs then
11.   Compute total code size $S_{\text{code}} = \sum_{k=0}^{s} S_{i,j}$.
12.   Get the period $T_{\text{sink}}$ of the sink actor in $G'$.
13.   return $(T_{\text{sink}}, S_{\text{code}})$.
14. else
15.   $s = s + 1$.
## D. EXPERIMENTS

Table 2: Benchmark characteristics.

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

Figure 6: An example of an individual. The first replica of $A_1$ is allocated on the $j$th PE and the $j^f$th replica of $A_1$ does not exist.

Table 3: Parameters for the genetic algorithm.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Recommended value in [19]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>80</td>
</tr>
<tr>
<td>Number of generations</td>
<td>300</td>
</tr>
<tr>
<td>Crossover rate</td>
<td>0.9</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>0.05</td>
</tr>
<tr>
<td>Mating rate</td>
<td>0.1</td>
</tr>
</tbody>
</table>