Mapping of Synchronous Dataflow Graphs on MPSoCs Based on Parallelism Enhancement[☆]

Qi Tang[a,*], Twan Basten[b,c], Marc Geilen[b], Sander Stuijk[b], Ji-Bo Wei[a]

[a]*Department of Electronic Science and Engineering, National University of Defense Technology, Changsha, China.*
[b]*Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, Netherlands*
[c]*Embedded Systems Innovation, TNO, Eindhoven, Netherlands*

Abstract

Multi-processor systems-on-chips are widely adopted in implementing modern streaming applications to satisfy the ever increasing computation requirements. To take advantage of this kind of platform, it is necessary to map tasks of the application properly to different processors, so as to fully exploit the inherent task-level parallelism and satisfy the stringent timing requirements. We propose the Parallelism Graph to capture the task-level parallelism of the application and transform the mapping problem to a graph partitioning problem. The graph partitioning problem is formulated as an Integer Linear Programming problem, which is solved optimally using the ILP solver. To reduce the complexity, a two-step local search algorithm, i.e., the greedy partition and refinement algorithm, is proposed. Since one-shot heuristics cannot guarantee the solution quality, evolutionary algorithms are widely used to search the solution space such that better results can be found. We also integrate the idea of parallelism enhancement into the genetic algorithm and propose a hybrid genetic algorithm to improve the performance. Sets of synthesized Synchronous Data Flow Graphs and some practical applications are used to evaluate the performance of the proposed algorithms. Experiment results demonstrate that the proposed algorithms outperform available algorithms.

Keywords: Synchronous Dataflow Graph, multiprocessor, mapping, graph partition, genetic algorithm

[☆]This work was partially supported by China Scholarship Council (CSC).
[*]Corresponding author
Email addresses: `q.tang.andy@qq.com` (Qi Tang), `a.a.basten@tue.nl` (Twan Basten), `m.c.w.geilen@tue.nl` (Marc Geilen), `s.stuijk@tue.nl` (Sander Stuijk), `wjbhw@nudt.edu.cn` (Ji-Bo Wei)

1. Introduction

Synchronous Data Flow Graphs (SDFGs) are widely used in modeling modern streaming applications, including video/audio en/decoding, software defined radio, etc. To satisfy the quality requirements of the consumer, these applications are becoming more and more complex and computationally intensive, which imposes great challenges on hardware design, especially when it comes to consumer electronics that are generally battery-powered [1]. Many applications have stringent timing requirements, e.g., in terms of throughput. To meet the timing requirements, a straightforward method is to increase the clock speed of the processor. However, as the clock frequency increases, the energy consumption grows rapidly, making this method less attractive. What's more, as the clock frequency approaches physical limits, we can no longer depend on faster processors to provide more computation capacity. Therefore, Multi-Processor Systems-on-Chips (MPSoCs) rather than single-processors with high clock frequency provide another solution. MPSoCs can make a better tradeoff between computation capacity and power consumption, therefore, they are pervasively used in practice. Though MPSoCs can provide powerful computation capacity, it does not necessarily mean that the application deployed on it can take full advantage of it. In fact, few benefits can be obtained by using MPSoCs if the application offers little parallelism. As demonstrated by Amdahl's law [2], the improvements gained by using multiprocessors are limited by the application parallelism, i.e., the extent to which the application can be parallelized. Though many applications, e.g., streaming applications modeled by SDFGs, do expose a lot of parallelism, it remains a problem as to how to exploit it, which is the so-called parallel scheduling problem.

Scheduling consists of mapping, ordering and timing [3]. Mapping determines the task-to-processor assignment, ordering determines the task execution order on each processor, and timing determines the time when each task starts execution. For an optimal schedule, there always exists an optimal mapping, which makes it necessary to study how to obtain the optimal mapping. This paper studies how to map an SDFG onto the MPSoC, specifically, we focus on duplication-free mapping [4, 5, 6] that does not duplicate a task to multiple processors for execution and do not consider data-level parallelism [7, 8, 9]. The throughput is the main concern for many streaming applications. For example, for the video decoder, it is quite important that the frame can be delivered in a specific rate, so as to make the video smooth enough. Therefore, the objective of this paper is to find the task-to-processor mapping that delivers the optimal long-term throughput. To achieve this goal, while constructing the mapping, tasks that are able to execute concurrently should be mapped to different processors so that the final schedule can exploit more parallelism of the application. Since the amount of resources, i.e., the number of processors, is limited, it is not possible to exploit all the parallelism; thus a tradeoff should be made. Therefore, tasks that contribute more parallelism should be mapped to different processors with higher priority. Based on the above ideas, we propose the Parallelism Graph (PG) to quantify and model the task-level parallelism of the SDFG, and transform the mapping problem to a graph partitioning problem. We formulate the graph partitioning problem as a pure 0-1 Integer Linear Programming (ILP) problem and use available ILP solver to solve it. The flaw of ILP is that it is hard to solve, especially for large-scale problems. Therefore, a two-step heuristic called Greedy Partition and Refinement Algorithm (GPRA) is proposed to solve this problem. Both the ILP-based algorithm and GPRA are one-shot methods, which are generally incapable at producing the global optimal solution. Differently, the population-based meta-heuristic algorithms, e.g., the genetic algorithm (GA), are powerful in searching the solution space to find better solutions, however, they face problems in convergence speed and finding local optima [10]. While GA does not integrate any local search strategy, we combine GA with the idea of parallelism enhancement and propose a hybrid genetic algorithm (HGA) for the mapping problem. The proposed algorithms are implemented in SDF3 [11], and are evaluated by sets of randomly generated SDFGs and some real applications.

In the remainder of this paper, we use the following notations. $\mathbb{Z}$, $\mathbb{Z}^+$ and $\mathbb{Z}_0^+$ denote the set of integers, positive integers and non-negative integers respectively. We use boldface capitals to denote vectors/sets and corresponding italic lowercase letters to denote elements in them. For a vector or set, we use $|\cdot|$ to denote the number of its elements.

The remainder of the paper is organized as follows. In Section 2, we discuss the related work. The models and definitions are described in Section 3. In Section 4 we formalize the problem to be solved. In Section 5 the methods for constructing the schedule for any given mapping and analyzing the throughput are introduced. The Parallelism Graph and its construction are presented in Section 6 and the pure 0-1 ILP model of the graph partitioning problem is formulated in Section 7. In Section 8 and 9 we introduce the greedy partition and refinement algorithm and the hybrid genetic algorithm respectively. In Section 10 the experiment results are presented. Finally, we conclude the paper in Section 11.

2. Related Work

Exploiting the inherent parallelism of streaming applications is critical in improving schedule performance. [7, 8] take advantage of data, pipeline and task parallelism to improve the schedule throughput. [7] proposes an ILP formulation to exploit the data parallelism of the application by splitting the work load of a task to multiple copies and distributing copies of a task to different processors, thus balancing the loads on different processors to a deeper extent and improving the throughput of the pipeline schedule. While [7] does not consider task parallelism, [8] combines data, pipeline and task parallelism in the schedule to improve the performance. Both [7, 8] focus on acyclic dataflow graphs and try to exploit data-level parallelism of the application, while this paper focuses on task-level parallelism of cyclic SDFGs.

For the cyclic dataflow graph, e.g., SDFG, [12] proposes a load balancing algorithm to map SDFGs to multiprocessors by balancing the computation load, communication bandwidth and memory consumption. Tasks are bound to processors in non-increasing order of the task priority defined as the estimated maximum cycle mean (MCM) [12] of any execution cycle containing that task. The key idea behind the load balancing method is balancing the load on each resource. However, inter-task precedences in the SDFG are not fully exploited in the algorithm, though the application structure is partly taken into account while computing the task priority [12]. Even though the strategy of distributing computations evenly on different processors works well for applications with little or no inter-task precedences, for applications with more complicate structures, more strategies should be adopted. Load balancing is also utilized in [13] to map SDFGs to multiprocessor. However, the authors focused on reducing the energy consumption while meeting the timing requirements. The proposed method enumerates a lot of mapping solutions to make a good tradeoff between energy consumption and the timing requirements. Rather than balancing the load, this paper quantifies task-level parallelism of the application, and optimizes the mapping indirectly by optimizing the exploited parallelism.

Another method for scheduling the SDFG consists of transforming the SDFG to an equivalent Directed Acyclic Graph (DAG) [3, 9] that can be derived from the equivalent homogeneous SDFG (HSDFG) by simply removing all edges with delay. After the transformation, other DAG scheduling algorithms [14, 15] can be utilized to solve the scheduling problem. Among these DAG scheduling algorithms, list-based algorithms [14, 15] that divide the scheduling to priority assignment and task scheduling are the most popular. The priority can be computed before or during mapping by taking into account the application structure statically or dynamically. The critical path based method is utilized the most in computing priority, since tasks on the critical path intuitively are more important and thus deserve to be scheduled earlier. However, this scheduling strategy primarily applies to systems consider task duplication and the performance for the duplication-free problem is not very good. Differently, this paper optimize the mapping for the duplication-free problem, and use list scheduling and self-timed scheduling to construct ordering and timing. Since by forcing the task to be scheduled onto the same processor, the DAG scheduling algorithm can be used for the problem in this paper, the famous Heterogeneous Earliest-Finish-Time (HEFT) scheduling algorithm [15] is used in this paper for comparison.

ILP-based accurate scheduling algorithms are studied a lot [16]. Since the scheduling problem is a combinatorial optimization problem, using ILP is straightforward. Modeling with the ILP formulation, three aspects of the scheduling, i.e., mapping, ordering and timing, can be solved as a whole. However, such kind of ILP model is difficult to solve even for small-scale problems. Differently, the ILP model proposed in this paper is simpler, since the problem is partly transformed to the graph partitioning problem.

Population-based meta-heuristic algorithms, such as genetic algorithm [17, 18], ant colony optimization [19], particle swarm optimization [20] and artificial bee colony algorithm [21], have been applied to many different scheduling problems. These algorithms are exploration algorithms, and are proven to outperform one-shot heuristic algorithms at the expense of extra computation time.

[17, 18] use the genetic algorithm to model different kinds of scheduling problems. [17] uses the genetic algorithm to solve the scheduling problem that considers code/data overlay on systems with limited scratchpad memory. However, it does not optimize the task allocation while solving the task-to-processor mapping as our methods do. [18] proposes a hybrid genetic algorithm for scheduling DAGs, which integrates the Critical Path Genetic Algorithm into a common genetic algorithm to improve the performance. Our work is inspired by these works. Differently, we use parallelism enhancement to improve the performance.

3. Application Model

A lot of literature uses DAGs to model applications. Recently, data flow graphs like SDFGs and scenario aware dataflow graphs [22, 23] gained a lot of research attention due to their powerful combination of expressivity and analyzability. We also use SDFGs to model streaming applications, e.g., software defined radio and multimedia applications. SDFGs can well capture the execution features of such kind of applications, e.g., multi-rate execution, and also provide some useful analytical properties, e.g., consistency, deadlock-free, repetition vector, memory requirements and throughput, making it an attractive computation model.

Definition 1. ***(SDFG)*** *A synchronous data flow graph is a directed graph and is denoted by $G = (\boldsymbol{V}, \boldsymbol{E})$, where $\boldsymbol{V}$ is a finite set of nodes or vertices representing tasks or actors of an application, and $\boldsymbol{E}$ is a finite set of directed edges denoting the communications between tasks. Each node $v \in \boldsymbol{V}$ is associated with a cost $c(v)$ representing the number of clock cycles needed to complete an execution of the task. Each edge $e \in \boldsymbol{E}$ is defined as a tuple (src, p, dst, q, d), where src is the source task, p is the production rate, dst is the destination task, q is the consumption rate, and d is the initial token count on the edge. For a given edge e, we use the notions $src(e)$, $p(e)$ etc., to denote its elements. A task can only fire when there are sufficient tokens on the edges where it consumes. When the source task $src(e)$ finishes its execution, it produces $p(e)$ tokens on the edge and the destination task $dst(e)$ consumes $q(e)$ tokens from the edge when it is invoked. We also refer to edge e as an output edge of task $src(e)$ and an input edge of task $dst(e)$.*

Since the rate at which tokens are produced on an edge may differ from the rate at which tokens are consumed from the edge in an SDFG, i.e., the SDFG is multi-rate, tasks in the SDFG may execute with different frequencies and thus appear different numbers of times in the schedule. We use the notions of SDFG iteration and repetition vector to capture these features of SDFGs.

Definition 2. ***(SDFG iteration)*** *An SDFG iteration is defined as the process of executing each task the minimum positive number of times so that the token count on each edge returns to the initial value.*

Definition 3. ***(Repetition vector)*** *The repetition vector* $\boldsymbol{R}$ *of an SDFG with* n *tasks numbered from* 0 *to* $n-1$ *is a column vector of length* n*, and the* k*-th element of* $\boldsymbol{R}$*, i.e.,* $\boldsymbol{R}(v_k)$*, represents the number of execution times of task* v_k *in an iteration.*

The repetition vector can be calculated by solving the balance equations [24]. For an SDFG, if the balance equations of the graph have non-trial solutions [24], then $\mathbf{R}$ exists and the SDFG is said to be consistent [24]. An incorrectly constructed SDFG may be dead-lock while executing. This paper only considers SDFGs that are consistent and deadlock-free. A consistent SDFG can always be converted to an equivalent HSDFG [3] in which all rates equal to one. However, this conversion can result in exponential increase in graph size. In the HSDFG, each task v in the SDFG is duplicated $\mathbf{R}(v)$ times, i.e., there are $\mathbf{R}(v)$ copies or instances of v in the HSDFG. So, the task number of the HSDFG is the addition of each element of the repetition vector of the corresponding SDFG. The repetition vector provides information as to how many times each task has to be executed in an iteration. Every time the task is started, we say one task instance has fired. In this paper, we also call $\mathbf{R}(v)$ the instance count of task v in an iteration.

This paper only considers strongly connected SDFGs, in which every task is connected with any other one directly or indirectly. For practical applications, this is a reasonable assumption. Since for practical systems, the maximum token count on each edge should be finite while executing, thus the system can be implemented with a finite buffer. The above constraint can be modeled by adding extra constraining edges to the SDFG. For each edge in the SDFG, an additional reverse edge can be added to limit the buffer size. Therefore, there is a path between each pair of tasks in the connected SDFG, which makes the SDFG strongly connected [5]. Since we consider duplication-free scheduling, we add self-edges to stateless tasks of the SDFG. Then, buffer size constraining edges are added to the SDFG while keeping it rate-optimal [3], i.e., the throughput is not changed.

Fig. 1 shows an SDFG with five tasks, i.e. v_0, v_1, v_2, v_3 and v_4. The production and consumption rates are indicated at the ends of each edge. For simplicity, if the rate is equal to one then it is omitted. The text near the edge is the edge label and the number in the parentheses beside the edge label represents the initial token count of this edge. If the initial token count is zero, then it is not shown in the figure for simplicity. We assume that each task v_0, v_1, v_2, v_3 and v_4 in Fig. 1 has an execution time of $5, 7, 4, 10$, and 14 respectively. The repetition vector of the SDFG in Fig. 1 is $[2, 2, 3, 1, 1]^T$, meaning that in one iteration tasks v_0, v_1, v_2, v_3 and v_4 have to execute $2, 2, 3, 1$ and 1 time respectively.

4. Problem Statement

For concurrent real-time streaming applications modeled by SDFGs, the throughput is among the most important performance metrics. The ideal through-

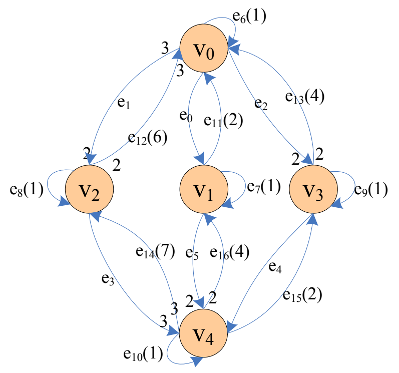


Figure 1: The structure of the example SDFG.

put without resource constraints can be obtained by the self-timed schedule [3]. However, practical systems are limited in resource, thus resource allocation or scheduling is necessary. Scheduling contains three aspects, i.e., mapping tasks onto the platform, ordering the executions of tasks bound to the each processor and determining the start time of each task [3]. In this paper, we investigate the first aspect of the scheduling problem, i.e., mapping the streaming application modeled by an SDFG onto the processors of an MPSoC with the objective of maximizing the long-term throughput. To evaluate the quality of the task-to-processor mapping, we use available techniques to obtain the ordering and timing given the mapping and analyze the throughput, as illustrated in the following section.

As introduced earlier, an SDFG can be converted to an equivalent HSDFG, and the nodes in the HSDFG correspond to task instances in the original graph. Therefore, the SDFG mapping problem can be classified as two categories based on what kind of nodes are mapped: tasks of the original SDFG or task instances of the resulting HSDFG. The latter case is referred as task duplication, which means that different instances of an SDFG task are mapped to different processors. We consider the mapping problem without task duplication, which implies that no more than one processor is allocated to each SDFG task in the mapping. For duplication-based scheduling, the reader can refer to StreamIT [25]; however, it is out of the scope of this paper. Duplication-free mapping provides many advantages, e.g., simplifying data management, reducing memory consumption and avoiding graph transformation [4, 5].

The solved problem in this paper can be summarized as follows. Given an application modeled by an SDFG and a multiprocessor platform, find the task-to-processor mapping without task duplication and construct the task order and task timing on each processor, so as to optimize the long-term throughput.

5. Periodic Static-Order Schedule and Throughput Analysis

Since the aim is to find the mapping that delivers the optimal throughput, it is important to analyze the throughput of the mapping given by the mapping algorithm such that the algorithm performance can be evaluated. Given a mapping, it is still hard to find the optimal schedule in the view of throughput. It should be noted that the throughput differs with the execution style. One execution style is the so-called blocked schedule [3], in which a block is composed of one or several iterations. The number of iteration number in a block is called the blocking factor. In the blocked schedule, different blocks cannot overlap, and the execution of one block should start after the former one has finished. In a block, the execution order of multiple iterations is not forced to be sequential, rather, they can execute in any order if the inherent data precedences are satisfied, hence improving the performance. For the blocked schedule, finding a good blocking factor is critical for improving the performance. However, as the blocking factor increases, the overhead for controlling the system increases. As an alternative, we assume that the system executes according to a periodic static-order schedule [23] that forces tasks to fire one by one in a given order. Different iterations may overlap, proceeding in pipelining style. We construct the periodic static-order schedule for each processor by the use of HEFT [15]. We first convert the SDFG to DAG, then the HEFT algorithm is applied to the DAG based on the given task-to-processor mapping. Having obtained the periodic static-order schedule for each processor, the timing of tasks and the throughput can be determined using available methods.

The state space based throughput analysis method of SDFGs without resource constraints is proposed in [26]. In this method, the token distribution and task remaining execution time specify the state. The execution time and the token count are both discrete, so the state space transition system is a discrete system. Since the SDFG is strongly connected, and range of each element of the state is finite, so the state space is finite. Since no resource contention occurs while executing the SDFG, the transition system is deterministic. After a finite number of transitions, some states will be revisited, hence forming a cycle. Having obtained the cycle in the state space, the throughput can be calculated directly. Another method for computing the throughput is to model the self-timed execution of the SDFG in max-plus algebra [22]. The eigenvalue of the max-plus matrix equals the reciprocal of the throughput. The above methods do not take the mapping and schedule into account, making the result inaccurate. It's possible to model the mapping and the valid periodic static-order schedule in the SDFG [27, 23]. Since the periodic static-order schedule extracted from the schedule generated by HEFT is valid in terms of data precedence, it is always possible to model it into the SDFG. After the above step, the throughput of the schedule can be analyzed by applying methods introduced in [26, 22] on the SDFG modeled with resource constraints. In this paper, we model the periodic static-order schedule into the SDFG using the method proposed in [23] and use the state space based technique [26] to compute the throughput of the schedule.

It should be noted that the throughput analysis method does not guarantee the optimality of the throughput. It remains for further research as to how to obtain the optimal schedule with resource contention, while providing acceptable time complexity.

6. Parallelism Graph and Its Construction

In this paper, we try to optimize the exploited parallelism while mapping the application so as to improve the throughput. Intuitively, if more tasks can execute in parallel, then its performance in terms of throughput is better. However, it is an issue as to how to model the inter-task parallelism of the application. This section introduces the concept of Parallelism Graph (PG) to quantify and model the inter-task parallelism of the SDFG; besides, a method based on the self-timed schedule is proposed to construct the PG. Based on the PG, the mapping problem is converted to a graph partitioning problem. The PG is defined as follows.

Definition 4. ***(Parallelism Graph)*** *The Parallelism Graph of an SDFG $G = (\boldsymbol{V}, \boldsymbol{E})$ is a weighted undirected graph that is denoted as a pair $PG = (\boldsymbol{V}_{pg}, \boldsymbol{E}_{pg})$, in which each vertex corresponds to a task in the SDFG and each edge represents that the connected tasks can execute in parallel. Each edge is associated with a positive number, i.e., the edge weight, representing a heuristic metric of the amount of parallelism between the associated tasks.*

The PG can be constructed in various ways. However, PGs constructed by different methods differ from each other, and the obtained mappings have various performances. This paper computes the inter-task parallelism by the use of self-timed schedule [3] of the SDFG. In the self-timed schedule, tasks are fired as soon as data is available at all their input edges. The self-timed schedule is defined as follows.

Definition 5. ***(self-timed schedule)*** *The self-timed schedule of an SDFG $G = (\boldsymbol{V}, \boldsymbol{E})$ is a mapping $s : \boldsymbol{V} \times \mathbb{Z}^+ \to \mathbb{Z}_0^+$. $s(v_i, k)$ denotes the start time of the k-th instance of task v_i, where $v_i \in \boldsymbol{V}$, $k \in \mathbb{Z}^+$. The values satisfy Equation 1,*

$$s(v_i, k) = \begin{cases} 0, \ \neg \ \exists \ e_{j,i} \in \boldsymbol{E} \ or \ dep_{ij}(k) \leq 0 \\ \max\limits_{v_j \in \boldsymbol{V}, e_{j,i} \in \boldsymbol{E}} \{s(v_j, dep_{ij}(k)) + c(v_j)\}, \ else \end{cases} \tag{1}$$

where $e_{j,i}$ denotes the edge from task v_j to task v_i, and $dep_{ij}(k)$ is defined as Equation 2, denoting that the k-th instance of task v_i depends on the $dep_{ij}(k)$-th instance of task v_j.

$$dep_{ij}(k) = \left\lceil \frac{k * q(e_{j,i}) - d(e_{j,i})}{p(e_{j,i})} \right\rceil \tag{2}$$

As shown in [26], the self-timed schedule can be represented as a state transition system. At each time, the remaining execution time of each task and the token count on each edge comprise a state. Since we consider duplication-free scheduling in which at most one copy of a task can be active at a time, the state can be represented as a vector $\mathbf{S}$. $\mathbf{S}$ has $|\mathbf{V}| + |\mathbf{E}|$ elements, with the first $|\mathbf{V}|$-th elements representing the remaining execution time of each task and the remaining elements representing the token count on each edge. For a specific time t, we use $\mathbf{S}_t$ to represent the state at this time, use $s_t(v)$ to denote the remaining execution time of task $v \in \mathbf{V}$, and use $s_e(t)$ to denote the token count of edge $e \in \mathbf{E}$. In a state $\mathbf{S}_t$, if $s_t(v_i) > 0$, it means that task v_i is active at time t, i.e., the task is executing. Since the state transition system is discrete and deterministic, and the value of each element of the state is limited, the state transition system would reencounter a state that has been encountered as time elapses, i.e., it is composed of the transient state and the following periodic state. We execute the SDFG in a specific way, i.e., at each time instance, all tasks that are on their due time are finished; then, all tasks that are ready are fired. For simplicity, not all states are recorded while executing the SDFG; rather, only the states recording the execution state after starting all the enabled tasks are recorded. We say the state transition system enters the periodic state at time t if the system reencounters the state of time t at $t+T$, with T being the period. We define the period of the state transition system in a time-driven style as follows, in which each time is associated with a state and the state changes as time advances.

Definition 6. ***(Period)*** *The period of the state transition system of SDFG $G = (\boldsymbol{V}, \boldsymbol{E})$ is a minimal subsequence of the system execution trace, and it can be represented as $\{\boldsymbol{S}_t | t \in [t_1, t_2]\}$, where $\boldsymbol{S}_{t_1} = \boldsymbol{S}_{t_2}$.*

In the self-timed schedule, the periodic state determines the value of the throughput. Since the periodic state of the state transition system plays a critical role in determining the throughput, we use it to compute the inter-task parallelism. Whereas each period in the periodic state may cover more than one iteration, this feature does not affect the result no matter whether the computed value of concurrency is divided by this number or not. Because of the periodic feature of the periodic state, there is no need to execute the application indefinitely. To the contrary, the execution can be stopped when a complete period has been found, i.e., if $\mathbf{S}_{t_1} = \mathbf{S}_{t_2}$, $t_1 < t_2$, then the execution can stop at t_2.

Algorithm 1 outlines the process of constructing the PG based on the period of the state transition system of the SDFG. The key idea is to extract the parallel execution time of each pair of tasks in the period, and use the extracted parallel execution time to quantify the parallelism of the task pair. As stated earlier, we consider duplication-free mapping. Since each task is bound to only one processor, instances of a task cannot fire in parallel. To take it into account, the SDFG is extended by adding self-edges to forbid auto-concurrency while constructing the PG. Then, the PG is initialized with the tasks. Each task in the PG is associated with one task in the SDFG. Subsequently, the SDFG is executed according to the self-timed schedule in state transition style. While executing the SDFG, one period of the state transition system is extracted. Based on the period, the edges of the PG are added with the edge weight being computed by lines 8-12. As shown in lines 5-18, the edge weight between each pair of tasks is computed by adding the overlap execution time of the corresponding tasks in the period. As lines 9-11 show, if two tasks are both active at a specific time in the period, i.e., their remaining execution times, as recorded in the state, are positive, then these two tasks overlap in execution time and the weight is renewed by adding one. If the weight is non-zero, then an edge connecting the corresponding tasks is added to the PG, with the edge weight being set as this value.

Algorithm 1 Construct PG

Input: application model $G(\mathbf{V}, \mathbf{E})$.
Output: Parallelism Graph $PG(\mathbf{V}_{pg}, \mathbf{E}_{pg})$.

```
 1: for each task $v$ in the SDFG, add a self-edge $(v, 1, v, 1, 1)$.
 2: add vertices $v'_0, v'_1, \cdots, v'_{|\mathbf{V}|-1}$ to $\mathbf{V}_{pg}$, with $v'_i$ corresponding to $v_i \in \mathbf{V}$.
 3: execute the SDFG using self-timed schedule in state transition style.
 4: find the period $\{\mathbf{S}_t | t \in [t_1, t_2]\}$ of the state transition system.
 5: for $i = 0$ to $|\mathbf{V}| - 2$ do
 6:   for $j = i + 1$ to $|\mathbf{V}| - 1$ do
 7:     initialize $w(i, j) = 0$.
 8:     for $t = t_1$ to $t_2 - 1$ do
 9:       if $\mathbf{S}_t(v_i) > 0$ and $\mathbf{S}_t(v_j) > 0$ then
10:         $w(i, j) = w(i, j) + 1$.
11:       end if
12:     end for
13:     if $w(i, j) > 0$ then
14:       add edge $e_{ij}$ to $\mathbf{E}_{pg}$.
15:       set the weight of edge $e_{ij}$ as $w(i, j)$.
16:     end if
17:   end for
18: end for
```

We use the time-driven style, i.e., associating each time with a state, to describe the state transition system above for clarity and simplicity, however, the event-driven style can be used in implementation. The event-driven transition system only records specific states when a specific event happens, i.e., some tasks finish and/or start. Using this method, the time between the adjacent states can be more than one, hence reducing the state number. However, such an algorithm can be obtained straightforward from Algorithm 1, thus it is not elaborated in the paper.

Fig. 6 shows the self-timed schedule in terms of event-driven state transition system of the example SDFG in Fig. 1. In the figure, each dot denotes a state, and each edge in the figure denotes a transition, with the number on the edge representing the time elapsed between the connected states. Beside each state, the event, i.e., the tasks that are finished and started rightly before entering the state, is depicted. The cycle in Fig. 6 represents the period of the self-timed schedule. The state annotated by red numbers is the state when the execution enters the periodic state, and the numbers represent the remaining execution time of each task at this state. In the figure, if a task is active at a state, then it is also active in the time between this state and the following state. For example, at time 20, the remaining execution time is $4, 6, 3, 0, 14$, showing that all tasks except task v_3 are active. Since the elapsed time from $\mathbf{S}_{20}$ to $\mathbf{S}_{23}$ is 3, as shown on the edge between these two states, all tasks except task v_3 are active in the time interval $[20, 23)$. So, the period time of the schedule in Fig. 6 can be computed by adding the time on all edges of the cycle, with the value being 43. The period consists three iterations, since each task is started three times the number of the corresponding element in the repetition vector, e.g., task v_0 is fired six times that is thrice of $\mathbf{R}(v_0) = 2$.

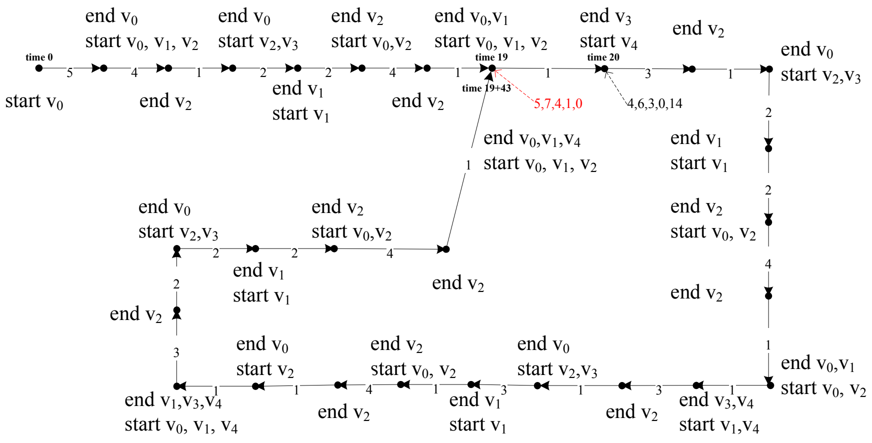
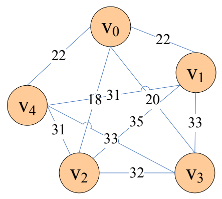


Figure 2: The self-timed schedule in terms of state transition of the example SDFG.

Fig. 3 shows the PG of the SDFG in Fig. 1 and the corresponding adjacency matrix of the PG based on the period in Fig. 6. To compute the overlap execution time, each state in the period $[19, 19 + 43]$ should be enumerated for each task pair. For example, the edge weight between tasks v_0, v_1 is initialized as 0. Since they are active between the time interval $[19, 20]$, $[20, 23]$, so, the intermediate edge weight of these two tasks is four at time 23. By enumerating remaining times in the period using similar method, the final edge weight can be obtained.

In the PG, the edge dictates that the associated tasks can execute in parallel, and the edge weight represents to what extent the tasks can execute simultaneously, i.e., the value of the parallelism. If more tasks can execute in parallel, then an iteration can finish in less time, thus improving the throughput. Hence, the mapping problem with the objective of maximizing the throughput is, to some extent, equivalent to maximizing the concurrency. Using the PG, the inter-task parallelism is quantified, so the mapping problem matches well with partitioning the PG such that the cut is maximized, with the cut defined as the sum of the weights of the edges crossing different partitions. By such a method, the mapping problem is transformed to a graph partitioning problem. In the



(a)

$$\begin{matrix} 0 & 22 & 18 & 20 & 22 \\ 22 & 0 & 31 & 33 & 35 \\ 18 & 31 & 0 & 32 & 31 \\ 20 & 33 & 32 & 0 & 33 \\ 22 & 35 & 31 & 33 & 0 \end{matrix}$$

(b)

Figure 3: The PG (a) of the example SDFG and its adjacency matrix (b).

following, we would show how to use the PG to solve the mapping problem.

7. Pure 0-1 Integer Linear Programming Model

In the above section, the PG is introduced to capture the inter-task parallelism of the application, and the mapping problem is transformed to partitioning the PG such that the cut between partitions is maximized. The above problem is a graph partitioning problem that is NP-hard [28]. We formulate it as a pure 0-1 Integer Linear Programming (ILP) model and use LINGO solver [29] to find the optimal solution. By comparing with other methods, it is shown that optimizing the parallelism is an alternative way to optimize the throughput.

7.1. Preliminaries

Before introducing the pure 0-1 ILP model, the relevant notations and a constraint transformation are presented first. We use the following notations in our model.

N: Task number in the PG.

M: Processor or partition number.

adj: Adjacency matrix of the PG.

$x_{i,k}$: A mapping binary variable. Equals 1 if task i is mapped to processor k and 0 otherwise.

We use the following proposition to transform *If-Then* constraints to linear constraints. A similar transformation can be found in [30].

Proposition 1 Let $x_1, x_2, \cdots, x_n$ be a set of variables. $f(x_1, x_2, \cdots, x_n)$ and $g(x_1, x_2, \cdots, x_n)$ are two functions on them. The non-linear constraint "if $f(x_1, x_2, \cdots, x_n) > 0$ then $g(x_1, x_2, \cdots, x_n) = 0$" can be modeled by the following linear constraints:

$$f(x_1, x_2, \cdots, x_n) \leq L * (1 - au) \tag{3}$$

$$g(x_1, x_2, \cdots, x_n) \geq -L * au \tag{4}$$

$$g(x_1, x_2, \cdots, x_n) \leq L * au \tag{5}$$

$$au = 0 \text{ } or \text{ } 1 \tag{6}$$

where L is a large positive number, chosen large enough so that $f \leq L$ and $-L \leq g \leq L$ hold for all values of $x_1, x_2, \cdots, x_n$ that satisfy other constraints in Equations 3-6, and au is a binary auxiliary variable.

Proof: If $f > 0$, then Equation 3 holds only if $au = 0$. Then Equations 4 and 5 imply $g \geq 0$ and $g \leq 0$, so $g = 0$. Thus, if $f > 0$, then Equations 3-6 ensure that $g = 0$. Also, if $f > 0$ is not satisfied, then Equation 3 allows $au = 0$ or $au = 1$. If $au = 1$, Equations 4 and 5 are satisfied according to the condition $-L \leq g \leq L$. Thus, if $f > 0$ is not satisfied, then the values of $x_1, x_2, \cdots, x_n$ are unrestricted and $g \neq 0$ is possible.

7.2. The Pure 0-1 ILP Model

In our problem, the objective is to find the partition that maximizes the cut. We use binary variable $y_{i,j}$ to denote if task i and j are mapped to different processors. $y_{i,j}$ equals 1 if task i and j are mapped to different processors and equals 0 if they are mapped to the same processor. So the optimization objective can be expressed as follows.

$$max: \quad cut = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} adj_{i,j} * y_{i,j} \tag{7}$$

Since each task is mapped to only one processor, so the task mapping constraint as shown by Equation 8 must be satisfied.

$$\sum_{k=0}^{M-1} x_{i,k} = 1 \ \forall \ i \tag{8}$$

where $x_{i,k}$ indicates whether or not task i is mapped to processor k. $x_{i,k}$ equals 1 if it is mapped to processor k, so the addition of $x_{i,k}$ over all k must equal 1.

In Equation 7, the value of $y_{i,j}$ is unknown and should be modeled by mapping variables $x_{i,k}, x_{j,k}$. According to the definition, if there is any k that makes $x_{i,k} = 1$ and $x_{j,k} = 1$, then $y_{i,j} = 0$; otherwise $y_{i,j} = 1$. Since $x_{i,k}, x_{j,k}$ are binary variables, the statement can be reformulated as: for each k, if $x_{i,k} + x_{j,k} = 1$, then $y_{i,j} = 1$; if $x_{i,k} + x_{j,k} = 2$, then $y_{i,j} = 0$; otherwise the value of $y_{i,j}$ is uncertain. The above constraint can be further reformulated as: for each k, if $x_{i,k} + x_{j,k} > 0$, then $x_{i,k} + x_{j,k} + y_{i,j} - 2 = 0$. This is a *If-Then* constraint and can be linearized using Proposition 1. Applying Proposition 1 to this constraint and set L as 2, we obtain the following constraints:

$$x_{i,k} + x_{j,k} + 2 * au \leq 2 \tag{9}$$

$$x_{i,k} + x_{j,k} + y_{i,j} + 2 * au \geq 2 \tag{10}$$

$$x_{i,k} + x_{j,k} + y_{i,j} - 2 * au \leq 2 \tag{11}$$

$$x_{i,k}, x_{j,k}, y_{i,j}, au = 0 \ or \ 1 \tag{12}$$

where $i, j, k \in \mathbb{Z}_0^+$, $i, j \in [0, N-1]$, $k \in [0, M-1]$. Equations 9-12 should hold for all $i \neq j$, k.

Fig. 4(a) shows the mapping produced by applying the ILP to the PG in Fig. 3(a) onto a 2-processor platform. By this method, v_1 and v_3 are mapped to one processor and the others to another. Using the graph partitioning method, the cut and the potentially exploited concurrency is maximized, with the value being 173. While using the load balancing method, the cut values is 171, as shown in Fig. 4(b). The throughput of the ILP solution is 0.0233, and that of the load balancing solution is 0.02222.

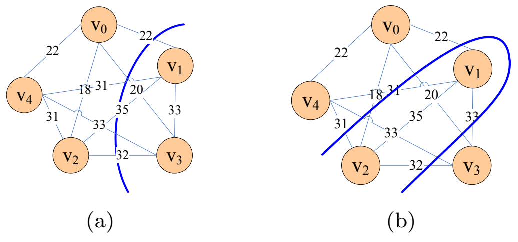


Figure 4: Mappings produced by the ILP-based method (a), the load balancing method (b).

There are also other methods that are more effective for partitioning the graph, such as the spectral partition [31]. However, this method is dedicated to large-scale problems. For small-scale problems, the partitions are much worse than the optimal solutions, making it unsuitable for our problem. Since the ILP-based method is not effective for large-scale problems, as an alternative, we introduce a two-step heuristic algorithm to solve this problem in the following section.

8. Greedy Partition and Refinement Algorithm

The ILP-based method introduced in the previous section is only efficient for small-scale problems; the variable number in the ILP model increases sharply as the task number or processor number increases, making the problem hard to solve. To reduce the time complexity and make the proposed method more efficient in practice, we propose in this section a heuristic algorithm. Our algorithm is a local search algorithm, consisting of two steps. At the first step, an initial partition is produced by a greedy partition method; at the second step, the initial partition is refined by recursively migrating tasks to a better partition that can improve the cut until no improvement can be gained. The initial experiment show that, to obtain a good solution in terms of throughput, it is not necessary to find the accurate partition that maximizes the cut as the ILP-based algorithm does; rather, a heuristic still performs quite good for most cases.

Algorithm 2 shows procedures of the heuristic algorithm. Given the Parallelism Graph constructed by algorithm 1, the loop in lines 3-13 maps tasks of the PG one by one using the greedy strategy, thus producing the initial partition; and the loop in lines 15-22 gradually improves the quality of the initial partition. In every iteration in the first loop, each task v in the set of unmapped tasks

Algorithm 2 Greedy Partition and Refinement Algorithm (GPRA)

Input: the SDFG $G(\mathbf{V}, \mathbf{E})$, the parallelism graph $PG(\mathbf{V}_{pg}, \mathbf{E}_{pg})$ and the set of processors $\mathbf{P}$.

Output: mapping: $\mathbf{V} \rightarrow \mathbf{P}$.

```
1:  let $\mathbf{U} = \mathbf{V}$, denoting the set of unmapped tasks.
2:  let $m_{ini} = \varnothing$ be the partial mapping.
3:  while $\mathbf{U} \neq \varnothing$ do
4:      for each task $v$ in $\mathbf{U}$ do
5:          for each processor $p$ in $\mathbf{P}$ do
6:              $m_{next} = m_{ini} + (v, p)$, representing the new partial mapping.
7:              $impr(v, p) \leftarrow cut(m_{next}) - cut(m_{ini})$,representing the cut improvement of mapping $v$ to processor $p$.
8:          end for
9:      end for
10:     $(v', p') \leftarrow \arg \max_{v \in \mathbf{U}, p \in \mathbf{P}} \{impr(v, p)\}$.
11:     map $v'$ to partition $p'$, and $m_{ini} = m_{ini} + (v', p')$.
12:     remove $v'$ from $\mathbf{U}$.
13: end while
14: $migrate \leftarrow true$.
15: while $migrate$ do
16:     $migrate \leftarrow false$.
17:     $v', p', impr(v', p') \leftarrow Migration(m_{ini})$.
18:     if $impr(v', p') > 0$ then
19:         migrate $v'$ to $p'$ in $m_{ini}$.
20:         $migrate \leftarrow true$.
21:     end if
22: end while
23: return $m_{ini}$.
```

Algorithm 3 Migration

```
Input: the SDFG $G(\mathbf{V},\mathbf{E})$, the parallelism graph $PG(\mathbf{V}_{pg},\mathbf{E}_{pg})$, the set of
    processors $\mathbf{P}$, and the mapping $m_{ini}:\mathbf{V}\rightarrow\mathbf{P}$.
Output: $v'$, $p'$ and $impr(v',p')$.
1: for each task $v$ in $\mathbf{V}$ do
2:     for each processor $p$ in $\mathbf{P}$ do
3:         let $m_{next}=m_{ini}$, and remove the mapping of task $v$ from $m_{next}$.
4:         $m_{next}=m_{next}+(v,p)$, representing the new mapping.
5:         $impr(v,p)\leftarrow cut(m_{next})-cut(m_{ini})$,representing the cut improvement
           of migrating $v$ to $p$.
6:     end for
7: end for
8: $(v',p')\leftarrow \arg\max_{v\in\mathbf{V},p\in\mathbf{P}}\{impr(v,p)\}$.
9: return $v'$, $p'$ and $impr(v',p')$.
```

$\mathbf{U}$ is tentatively allocated to each partition p, and the improvement of the cut value by mapping v to p, i.e., $impr(v,p)$, is computed. As line 6 shows, a new partial mapping m_{next} is obtained by mapping v to p, and $impr(v,p)$ equals the cut value of m_{next} minus that of m_{ini}, denoting the cut improvement. The computation of the cut of the partial mapping is similar with that of the full mapping by only considering the tasks that have already been mapped in the partial mapping and the edges that join them. Then, the task and partition that can improve the cut value most are picked out, and the associated mapping is carried out, as lines 10-11 show. This process is repeated until all tasks have been mapped and an initial partition has been obtained. Then, this partition is refined using a migrating strategy, as the loop in lines 15-22 shows. In the loop, Algorithm 3 is used to compute to what extent the cut value can be improved by migrating a task to another processor, as line 17 shows. The task and processor that make the improvement the largest are found out and the associated migration is carried out, as line 19 shows. If no improvement can be gained any more, the loop terminates. In the loop, the migration is carried out only when the cut improvement is positive, so, using the refinement, the cut value would grow gradually.

Algorithm 3 illustrates the process to improve the cut of a mapping. It is a single-migration algorithm. In this algorithm, only one task is chosen to be migrated to another processor. As shown in the algorithm, lines 1-7 compute the cut improvement by migrating each task to another processor. Then the best migration strategy is selected and returned, as lines 8-9 show.

It should be noted that the proposed method may end into local optima. To jump out of local optima and further improve the quality of the partition, some other strategies such as swapping can be used. However, according to the experiment, the proposed algorithm performs well enough without such a technique.

9. Hybrid Genetic Algorithm Based on Parallelism Enhancement

Evolutionary algorithms (EA) are inspired by natural evolution and they are powerful in solving large-scale combinatorial optimization problems that are generally NP-hard. Among the evolutionary algorithms, genetic algorithms (GAs), which generate solutions to optimization problems using techniques inspired by natural selection, such as inheritance, mutation, selection, and crossover, are widely used in task scheduling problems. While one-shot heuristic algorithms are generally incapable of producing the optimal solution, GAs provide a mechanism to search the solution space, thus providing better performance. However, GAs are not good at finding the local optima though they do very well in identifying the regions where the optima lie. Therefore, integrating a local search heuristic in a genetic algorithm is a good approach to improve the performance. In this section, we propose a hybrid genetic algorithm (HGA) by integrating the idea of parallelism enhancement into the genetic algorithm, with the framework being depicted in Fig. 5(b). We call the GA without parallelism enhancement the common GA (CGA), as shown in Fig. 5(a). Comparing Fig. 5(a) and Fig. 5(b), we can find that HGA consists some common steps as CGA, i.e., generating initial solutions, computing fitness value, crossover and mutation. However, HGA uses an additional step to enhance the parallelism of initial solutions and offsprings produced by the operation of mutation. In the following, we illustrate how each step of HGA works.

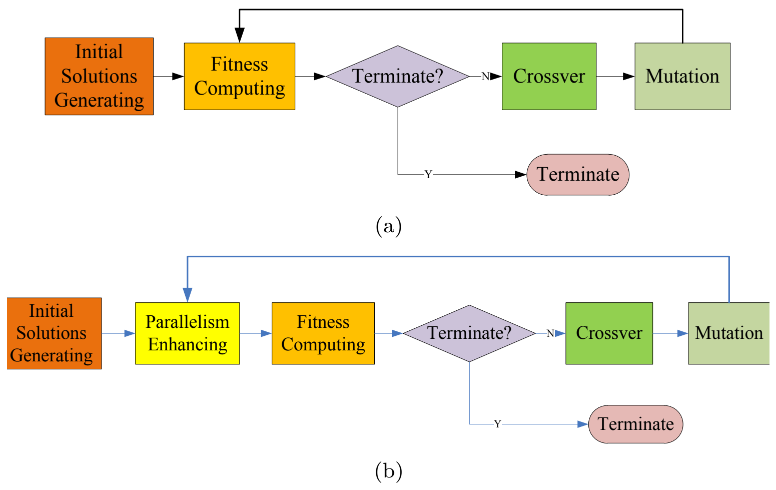


Figure 5: The framework of the CGA (a) and HGA (b).

9.1. Encoding and Initial Population Generation

Each task-to-processor mapping is encoded by an integer (named gene), representing the index of the processor where the task is allocated to. All the genes are cascaded, thus forming a string of integers. This string is called the chromosome and it models a complete mapping.

As most genetic algorithms do, the chromosomes in the initial population are randomly generated. The size of the initial population is determined by the user to find a good compromise between the simulation time and solution quality. For example, set the population size according to the size of the solution space so that a fixed percentage of the solution space will be searched.

9.2. Fitness Function

The fitness function is used to measure the quality of the solutions. While generating successive populations based on the parent population, the fitness of each solution in the parent population is used to determine the probability that the genes of this solution are propagated to the offspring population. In our problem, what we are interested is the throughput of the solution. So, we use the throughput of the mapping modeled by the chromosome as the fitness value. For each solution, we use techniques introduced in Section 5 to compute the throughput of the mapping.

9.3. Selection and Crossover Operator

In the genetic algorithm, the crossover operation is one of the most important steps. Before crossing two chromosomes, they have to be selected from the parent population. We use roulette-wheel selection in this paper. For the parent solution set $\mathbf{C} = \{C_0, C_1, \ldots, C_{|\mathbf{C}|-1}\}$, we use f_i to denote the fitness value or throughput of the i-th solution C_i. The roulette-wheel selection probability of solution C_i is defined by the following equation,

$$rw_i = \sum_{j=0}^{i} f_j \Bigg/ \sum_{k=0}^{|\mathbf{C}|-1} f_k \tag{13}$$

To select a solution, a uniformly distributed random value $rand \in (0, 1]$ is randomly generated. If $rw_{i-1} < rand <= rw_i$, then C_i is selected (supposing $rw_{-1} = 0$).

Having selected two solutions in this way, the crossover operator is applied to these two solutions with the crossover probability $\mu_c \in [0, 1]$, which is generally set between $[0.25, 0.75]$. If the uniformly distributed random value $rand \in [0, 1]$ is less than μ_c, the two selected solutions are crossed according to the crossover operator, else they are put directly into the child population. The standard one-point crossover operator is used in this paper, which first randomly selects a gene position in the chromosome and then, for the two selected chromosomes, swaps the genes before or after this position. The process of selection and crossover is repeated until the size of the child population is the same as the size of the parent population.

9.4. Mutation Operator

To avoid plunging into the local optima, the mutation operator is applied in the genetic algorithm. For each solution in the child population generated by the crossover operator, the mutation operator is applied with a mutation probability $\mu_m \in [0, 1]$, which is generally set between $[0.01, 0.2]$. If the uniformly distributed random value $rand < \mu_m$, then the mutation operator is applied to the solution. The mutation operator operates as follows: firstly a gene is randomly selected, and then the value is changed randomly to another one. Since each gene is an integer that encodes the processor index, the mutation operator reallocates the task associated with the selected gene to another processor randomly.

9.5. Parallelism Enhancement

We augment the genetic algorithm by enhancing the parallelism of the mappings encoded by the chromosomes in the child population and initial solutions. Since more tasks can execute in parallel and the performance of the mapping would increase by improving the parallelism exploited by the mapping, augmenting the genetic algorithm with parallelism enhancement improves the performance in both convergence performance and quality of the final solution. For each tentative mapping solution produced during executing the HGA, the parallelism of the mapping is enhanced. Algorithm 4 shows how to enhance the parallelism of the mappings in the initial and offspring population. This is a single-migration algorithm that migrates only one task, so that the HGA would not trap into local optima quickly. As shown in line 2 of Algorithm 4, for each solution in the offspring population, the best migration strategy is found using Algorithm 3. If the best migration does improve the cut, then it is carried out, as lines 3-5 show.

Algorithm 4 Parallelism Enhancement

```
Input: the SDFG $G(\mathbf{V}, \mathbf{E})$, the parallelism graph $PG(\mathbf{V}_{pg}, \mathbf{E}_{pg})$, the processor
    set $\mathbf{P}$ and the child population $\mathbf{C} = \{C_0, C_1, \ldots, C_{|\mathbf{C}|-1}\}$.
Output: parallelism-optimized $\mathbf{C}$.
1: for each chromosome $C_i$ in $\mathbf{C}$ do
2:     $v', p', impr(v', p') \leftarrow Migration(C_i)$.
3:     if $impr(v', p') > 0$ then
4:         migrate $v'$ to $p'$.
5:     end if
6: end for
```

10. Experiments and Results

In this section, we evaluate the proposed algorithms experimentally. All algorithms are implemented in SDF3 [11]. Since the load balancing method [12] matches the best with our work, we compare our work with it in this section. Besides, the HEFT algorithm [15] is shown to outperform other list scheduling algorithms and it can be adapted to the problem in this paper by forcing each task to be allocated to only one processor, so we also make a comparison with it. Finally, the common genetic algorithm [18] that does not use parallelism enhancement is used to evaluate the performance of HGA to show the effectiveness of parallelism enhancement.

We use throughput and cut as performance metrics, and the runtime used by each algorithm is also compared. The throughput denotes the long-term average number of iterations completed per cycle, and it is computed by techniques introduced in Section 5. The cut of the mapping is formalized by Equation 14,

$$Cut = \sum_{e_{ij} \in \mathbf{E}_{pg}, mapping(v_i) \neq mapping(v_j)} w(i,j) \tag{14}$$

where $\mathbf{E}_{pg}$ is the set of edges in the parallelism graph, $w(i,j)$ is the edge weight of edge e_{ij}, and $mapping(v_i)$ represents the processor where task v_i is mapped to. Equation 14 means that the mapping divides the PG into multiple partitions, and the sum of the weights of the edges crossing different partitions defines the cut. In the experiment, we also extract the mapping from the schedule produced by HEFT and compute the cut of the mapping to further reveal the relation between cut and throughput.

A set of practical applications are used for performance evaluation, including H.263 encoder [32], samplerate conversion [33], MPEG-4 SP decoder [22], MP3 decoder[22], modem [33], TD-SCDMA [6] and WLAN 802.11a receiver [6]. For simplicity, we denote these applications as #1-7 in the table when reporting the experiment result. According to the task number of the application, these applications are categorized as two subsets, i.e., the small applications and the large applications. The former three are classified as small applications, with the task number being 5, 6 and 5 respectively, and the edge number being 7, 11 and 8. The last four are large applications, with the task number being 14, 16, 16 and 24 respectively, and the edge number being 21, 35, 25 and 32. Limited by the variability, these real applications are not enough to cover applications that may appear in the future. For this reason, we have also used sets of synthesized applications generated by the open source tool SDF3 [11] to evaluate the effectiveness of the proposed methods. While generating SDFGs using SDF3, the sum of all entries of the repetition vector is set to be five times of the task number. Other parameters of the application are set partly according to that of real applications, e.g., the in/out-degree and rate. The in-degree and out-degree are generated randomly with the average value and variation of 2, the minimum value of 0 and the maximum value of 4. The production and consumption rates are generated randomly with the average and variation of 5 and 7, the minimum value of 1 and the maximum value of 9. For each graph size, 200 SDFGs are generated. For both real and synthesized applications, the task execution time is uniformly distributed between 400 and 1000 in the experiment. All experiments are carried out on an Intel Core i5 processor (2.60 GHz and 4 GB RAM) running 64 bits Windows 7.

10.1. Results of Synthesized Applications

In this subsection, we evaluate the proposed algorithms using sets of synthesized applications. Three kinds of performance metrics are compared, namely, the throughput ("Thr"), the cut value ("Cut") of the mapping, and the runtime ("Time") of the algorithm. In the following, all metrics of ILP, GPRA and HEFT are normalized by that of LB, so metrics of LB all equal one.

In the first experiment, we evaluate these algorithms on sets of small problems, with the application size ranging from 5 to 15 and the processor number ranging from 2 to 4. The results are shown in Table 1, where "tN" and "pN" denote the task number and processor number respectively.

The last row "Avg" of Table 1 summarizes the average performance. The average throughput of ILP and GPRA are 1.174 and 1.141, being 18.23% to 14.10% higher than HEFT and LB. The average cut of HEFT and LB are 0.902 and 1, which are 9.2%-21.95% less than that of ILP and GPRA. These results demonstrate that optimizing the cut of the mapping performs better than load balancing and HEFT in obtaining schedules with higher throughput.

The throughput of ILP and GPRA are similar, and their cut value are also similar; however, the runtime of ILP is about 100 times larger than GPRA, proving the effectiveness of GPRA and its advantages over ILP. Besides, using the cut to optimize the throughput is not an exact method, meaning that smaller cut can still produce good result. As shown in the table, for the experiment with 5-task and 4-processor, ILP and GPRA have the same average cut, but the average throughput is different. However, as demonstrated by the experiment, optimizing the cut provides better performance compared with available methods averagely.

The runtime of ILP and GPRA are higher than HEFT and LB. However, the runtime of GPRA, LB and HEFT are less than several seconds in our experiment, showing that GPRA is of practical usage. Differently, the runtime of ILP is quite large for large-size problems, e.g., on the 4-processor platform, the runtime of ILP is about half minute for 15-task applications, and one hour for 20-task applications.

We have also recorded the probability when ILP and GPRA outperform LB and HEFT in throughput. Experiment results show that for 73.2% and 66.4% of the problems, ILP performs better than LB and HEFT, and for 61.3% and 50.4% of the problems, GPRA performs better than LB and HEFT, which shows that ILP and GPRA are more effective than LB and HEFT. Besides, it's shown that for 67.6% and 56.4% of the problems, ILP outperforms LB and HEFT by more than 5%; and for 54.3% and 43.5% of the problems, GPRA outperforms LB and HEFT by more than 5%;

In the second experiment, we evaluate GPRA, LB and HEFT using sets of large problems, with the application size ranging from 20 to 30 and the processor number ranging from 6 to 8. Since ILP is time-consuming for large problems and GPRA is as effective as ILP in terms of throughput, as demonstrated in the former experiment, we do not test ILP in this experiment. Table 2 depicts the experiment results. As shown in the table, GPRA is also effective for large

Table 1: Normalized performance of the ILP-based algorithm, GPRA, HEFT and the load balancing method on small problems.

tN	pN	Thr				Cut				Time			
		ILP	GPRA	HEFT	LB	ILP	GPRA	HEFT	LB	ILP	GPRA	HEFT	LB
5	2	1.066	1.048	0.987	1	1.041	1.028	0.849	1	5.902	4.784	0.905	1
	4	1.198	1.208	0.719	1	1.024	1.024	0.641	1	4.057	3.727	0.876	1
10	2	1.120	1.070	1.077	1	1.143	1.132	1.016	1	9.765	8.904	1.109	1
	4	1.305	1.218	0.949	1	1.095	1.088	0.845	1	129.6	9.357	0.812	1
15	2	1.092	1.038	1.098	1	1.164	1.156	1.080	1	10.304	6.853	0.738	1
	4	1.263	1.262	1.128	1	1.132	1.126	0.978	1	3006	2.000	0.604	1
Avg		1.174	1.141	0.993	1	1.100	1.092	0.902	1	527.6	5.938	0.841	1

problems. By using GPRA, the solution quality in terms of throughput is increased a lot compared with LB and HEFT. As shown in the last row of the table, the average normalized throughput of GPRA is 1.161, being 15.06% and 16.10% higher than HEFT and LB. The cut value of GPRA is larger than both HEFT and LB, with the improvement being about 8%. It shows that optimizing the cut value of the mapping leads to better mapping in throughput for large problems. The runtime of GPRA is averagely 2.088 times of LB and 4.89 times of HEFT. However, in all these experiments, GPRA only consumes about ten seconds, and GPRA is even less time-consuming than LB for some problems, showing that it is useful for practical usage. Besides, experiment results show that for 71.3% and 73.8% of the problems, GPRA performs better than LB and HEFT; and for 63.0% and 61.6% of the problems, GPRA performs better than LB and HEFT by more than 5%, showing that GPRA is more effective than LB and HEFT.

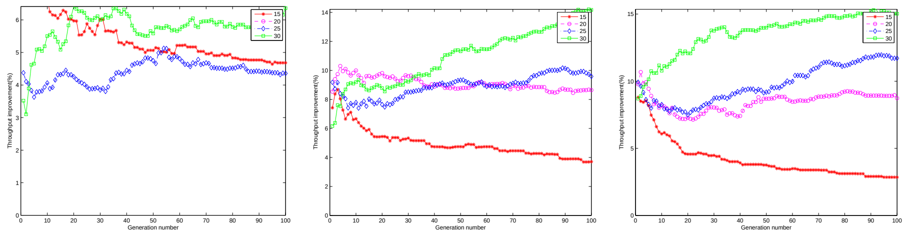
Table 2: Normalized performance of the ILP-based algorithm, GPRA, the list-based algorithm and the load balancing method on large problems.

tN	pN	Thr			Cut			Time		
		GPRA	HEFT	LB	GPRA	HEFT	LB	GPRA	HEFT	LB
20	6	1.189	1.011	1	1.092	0.936	1	3.463	0.558	1
	8	1.235	0.912	1	1.071	0.890	1	3.460	0.557	1
25	6	1.117	1.045	1	1.082	0.981	1	1.965	0.443	1
	8	1.183	0.983	1	1.071	0.943	1	1.974	0.430	1
30	6	1.099	1.099	1	1.089	0.998	1	0.835	0.292	1
	8	1.140	1.003	1	1.074	0.959	1	0.830	0.280	1
Avg		1.161	1.009	1	1.080	0.951	1	2.088	0.427	1

Finally, we test the performance of HGA by comparing it with GPRA and CGA on sets of large-size problems, with the processor number ranging from 4 to 8 and the application size ranging from 15 to 30. Parameters of both HGA and CGA are configured the same, with the crossover probability and mutation probability being 0.7 and 0.1 respectively.

HGA outperforms the one-shot heuristic, i.e., GPRA, with the throughput being increased by 11-50%, showing that despite the fact that the proposed heuristic outperforms some available one-shot algorithms, it still can be improved a lot, however, it still needs further research as to how to improve the performance of the one-shot algorithm. It should be noted that such a comparison is not so fair, since the GA-based algorithm is time-consuming compared with the one-shot algorithm. The genetic algorithm searches the solution space and thus needs more computation time. For the problem solved in this paper, the time spent on evaluating the solution constitutes the biggest part of the total runtime, so the runtime of the genetic algorithm is roughly $popuSize * iterNum$ times of the GPRA, where $popuSize$ is the population size and $iterNum$ is the iteration number, e.g., for the problem with 20 offsprings and 100 iterations, the runtime of the HGA is about 2000 times of the one-shot heuristic.

Fig. 6(a)-6(c) show the average performance improvement in terms of throughput of HGA over CGA in each generation on platforms with different processor numbers. It should be noted that the runtime of HGA is only about 0.74% higher than that of CGA even though it uses parallelism enhancement, since the runtime of this extra operation is ignorable compared with other operations, which makes the comparison fair enough. The horizontal axis and the vertical axis of Fig. 6(a)-6(c) represent the generation number and the average throughput improvement respectively, and the number in the legend denotes the application size in the corresponding experiment. For each problem instance, both HGA and CGA generate 100 generations and the best solution in each generation is recorded. Both HGA and CGA are executed three times for each problem instance and the best solution is used to evaluate the performance. The results in Fig. 6(a)-6(c) show that HGA can generate better solutions compared to CGA in terms of throughput, though the performance improvement varies with application size and processor number. According to these results, a throughput improvement of 3%-15% can be gained by using HGA. From these figures we can also see that the throughput improvement by using HGA has a positive relation with the problem size, i.e., the application size and the processor number. As shown in Fig. 6(b), the improvement for the 15-task applications is about 4% at the last generation, while that of the 30-task applications is about 14%. Similar relation can also be found in Fig. 6(c). The only exception exists when the processor number is 4, as shown in Fig. 6(a), the performance improvements for the 15-task and 20-task applications are higher than the other two sets of applications. As the processor number increases, the performance improvement also has an increasing trend except for small applications, i.e., the 15-task applications. For example, for the 25-task applications, the performance improvement is about 4% in the last generation on the 4-processor platform, while that of platforms with 6 and 8 processors are 10% and 12% respectively. Similar trend can be found for 20-task and 30-task applications, with the smallest application being the exception. For platforms with more processors, the parallelism is highly exploited for small-size applications, so the potential improvement is limited. It shows that HGA is good at finding better solutions compared to CGA, especially for large problems that have a larger solution space. Besides, for 95.4% of the problems, HGA performs better than CGA; and for 51.5% of the problems, HGA outperforms CGA by more than 5%.



(a) (b) (c)

Figure 6: Performance improvement of HGA over CGA on a 4-processor platform (a), 6-processor platform (b) and 8-processor platform (c).

10.2. Results of Practical Applications

This subsection evaluates the performance of the proposed methods using real-life applications. Table 3 reports the experiment result of both small and large applications using ILP, GPRA, LB and HEFT for the mapping problem. It is demonstrated that ILP and GPRA outperform other two one-shot heuristics in terms of throughput for real-life applications as they do for synthesized SDFGs. As summarized in the last row of Table 3, the average normalized throughput of ILP and GPRA are 1.219 and 1.202 respectively, both of which are higher than that of HEFT and LB, i.e., 0.964 and 1, showing that a throughput improvement of over 20% can be gained using the proposed mapping strategy. Since both ILP and GPRA optimize directly the cut while mapping the application, the cut value of ILP and GPRA are higher than that of LB and HEFT, with an improvement of over 40%. The runtime of ILP is roughly the same as LB for small applications, however, for large applications, it is quite high. Differently, the runtime of GPRA is approximately the same as HEFT and smaller than LB.

It should be noted that the average cut value of GPRA is 1.439, which is roughly the same as that of ILP, i.e., 1.439. Besides, the throughput of GPRA and ILP are 1.202 and 1.219 respectively, showing that GPRA only slightly deteriorates the throughput of ILP. However, the runtime of ILP is much higher than GPRA, while the average runtime of ILP is 31.936, GPRA only takes 0.684. The above results show that GPRA is a good alternative of ILP.

Noting that for some cases, smaller cut can lead to larger throughput, it is because the proposed mapping strategy is inherently a heuristic, and maximizing the cut/parallelism of the mapping does not necessarily lead to optimal throughput. In fact, enumerating all the solution space of a problem, it can be found that there are solutions with less cut value and higher throughput compared with the solution produced my our methods. However, statistically, maximizing the cut leads to better solutions, as demonstrated by the results.

Table 3: Normalized performance of GPRA, HEFT and the load balancing method on real applications.

App	pN	Thr				Cut				Time			
		ILP	GPRA	HEFT	LB	ILP	GPRA	HEFT	LB	ILP	GPRA	HEFT	LB
#1	2	1.347	1.347	1.190	1	1.742	1.742	1.256	1	0.956	0.969	0.974	1
	4	1.135	1.135	1.009	1	1.340	1.340	0.981	1	0.974	0.977	0.977	1
#2	2	1.105	1.069	1.023	1	1.154	1.138	1.055	1	1.017	1.033	1.093	1
	4	1.271	1.260	0.951	1	1.105	1.105	0.926	1	1.156	1.142	1.049	1
#3	2	1.229	1.199	0.887	1	2.669	2.669	1.598	1	0.998	0.977	0.874	1
	4	1.351	1.351	0.690	1	1.185	1.185	0.532	1	1.166	1.039	0.870	1
#4	2	1.047	1.133	0.968	1	1.152	1.143	0.915	1	1.022	1.034	1.082	1
	4	1.323	1.065	1.320	1	1.097	1.093	1.024	1	13.635	1.015	1.210	1
#5	2	1.310	1.213	1.168	1	1.665	1.664	1.263	1	1.860	0.529	0.503	1
	4	1.285	1.351	1.106	1	1.296	1.296	0.884	1	21.580	0.633	0.521	1
#6	2	1.152	1.085	1.012	1	1.423	1.422	1.080	1	1.142	0.046	0.034	1
	4	1.217	1.215	1.007	1	1.241	1.240	0.983	1	390.066	0.071	0.037	1
#7	2	1.062	1.069	1.107	1	1.587	1.583	1.548	1	0.706	0.045	0.032	1
	4	1.238	1.335	1.032	1	1.536	1.536	1.087	1	10.832	0.067	0.035	1
Avg		1.219	1.202	1.034	1	1.442	1.439	1.081	1	31.936	0.684	0.663	1

In the next experiment, we compare HGA with ILP/GPRA and CGA by mapping large-size real applications onto a four-processor platform. Even though HGA uses parallelism enhancement to improve the parallelism of each chromosome, the runtime of HGA in the experiment is almost the same as CGA, with a difference of only 1.53%, therefore, we treat the comparison of HGA and CGA a fair one and ignore this difference.

In our experiment, using HGA can improve the throughput by 32%-39% compared with ILP and GPRA for real applications. Similar with synthesized applications, this improvement is at the cost of a great deal of runtime. It also shows that there is great potential to further improve the performance of the one-shot heuristic; however, it remains further research.

Fig. 7 reports the performance improvement of HGA over CGA of large real applications at each iteration. In the experiment, both HGA and CGA gradually improve the solution and finally converge as iteration number increases. As shown in the figure, HGA outperforms CGA in terms of throughput. For MP3 decoder, TD-SCDMA and WLAN 802.11a receiver, the throughput improvement achieves 15.5% , 20.1% and 1.5% respectively. However, HGA cannot further improve the performance compared with CGA for modem, since they converge to the same point. It should be noted that HGA converges quickly than CGA for modem, while HGA converges at the six iteration, CGA converges at the twenty iteration in the experiment. According to Fig. 7, the performance improvement of HGA over CGA is application-dependent. For some applications, the improvement is remarkable; however, it is minimal for other applications. Similar phenomenon also happens for synthesized applications.

The reason should depend on the different convergence speed of CGA and HGA for a specific application. However, HGA outperforms CGA statistically, showing that by searching the solutions with large cut value, better solutions can be found and the convergence speed can be improved.

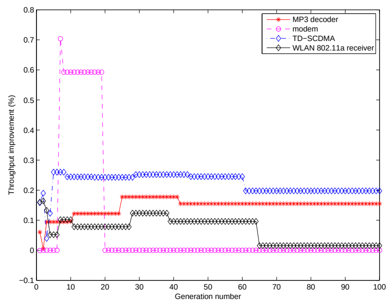


Figure 7: Performance improvement of HGA over CGA with respect to real applications.

11. Conclusions

This paper studies the duplication-free mapping of streaming applications modeled by SDFGs onto MPSoCs with the objective of maximizing the long-term throughput. The idea of maximizing the exploited parallelism in the mapping is utilized to solve this problem. A new model called Parallelism Graph, together with its construction algorithm, is proposed to model the inter-task parallelism of the application. Based on the Parallelism Graph, the mapping problem is transformed to partitioning the Parallelism Graph so as to maximize the cut. An ILP formulation is introduced for this partitioning problem and the ILP solver is utilized to solve it optimally. To reduce the complexity, a two-step heuristic GPRA is proposed to substitute the ILP-based method. Finally, we also proposed a hybrid genetic algorithm that integrates the idea of augmenting the parallelism of the solution. Extensive experiments are carried out on sets of random applications and some real applications. Experiment results demonstrate that the ILP-based algorithm and GPRA outperform the load balancing method and HEFT in terms of throughput; furthermore, the advantage of optimizing the cut in the mapping problem is also demonstrated. Besides, the hybrid genetic algorithm is also shown to have better performance compared to the common genetic algorithm that does not use parallelism enhancement.

References

[1] A. K. Singh, M. Shafique, A. Kumar, J. Henkel, Mapping on multi/many-core systems: survey of current and emerging trends, in: Proceedings of the 50th Annual Design Automation Conference, ACM, 2013, pp. 1–10.

[2] J. L. Hennessy, D. A. Patterson, Computer architecture: a quantitative approach, Elsevier, 2012.

[3] S. Sriram, S. S. Bhattacharyya, Embedded multiprocessors: Scheduling and synchronization, CRC press, 2012.

[4] G. Bilsen, M. Engels, R. Lauwereins, J. Peperstraete, Static scheduling of multi-rate and cyclo-static dsp-applications, in: Proceedings of the International Workshop on VLSI Signal Processing, IEEE, 1994, pp. 137–146.

[5] Q. Tang, T. Basten, M. Geilen, S. Stuijk, J.-B. Wei, Task-fifo co-scheduling of streaming applications on mpsocs with predictable memory hierarchy, in: Fifteenth International Conference on Application of Concurrency to System Design, IEEE, 2015, pp. 90–99.

[6] O. Moreira, H. Corporaal, Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor, Springer, 2014.

[7] M. Kudlur, S. Mahlke, Orchestrating the execution of stream programs on multicore platforms, in: Proceedings of the ACM conference on programming language design and implementation, 2008, pp. 114–124.

[8] M. I. Gordon, W. Thies, S. Amarasinghe, Exploiting coarse-grained task, data, and pipeline parallelism in stream programs, in: Symposium on architectural support for programming languages and operating systems, 2006, pp. 151–162.

[9] G. F. Zaki, W. Plishker, S. S. Bhattacharyya, F. Fruth, Implementation, scheduling, and adaptation of partial expansion graphs on multicore platforms, Journal of Signal Processing Systems (2016) 1–19 DOI: 10.1007/s11265-016-1107-8.

[10] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, J. Weglarz, Handbook on scheduling: from theory to applications, Springer Science & Business Media, 2007.

[11] S. Stuijk, M. Geilen, T. Basten, Sdf3: Sdf for free, in: Sixth International Conference on Application of Concurrency to System Design, Vol. 6, 2006, pp. 276–278.

[12] S. Stuijk, T. Basten, M. Geilen, H. Corporaal, Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs, in: Proceedings of the 44th annual Design Automation Conference, ACM, 2007, pp. 777–782.

[13] J. A. Ambrose, I. Nawinne, S. Parameswaran, Latency-constrained binding of data flow graphs to energy conscious gals-based mpsocs, in: International Symposium on Circuits and Systems, IEEE, 2013, pp. 1212–1215.

[14] O. Sinnen, Task scheduling for parallel systems, Vol. 60, John Wiley & Sons, 2007.

[15] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Transactions on Parallel and Distributed Systems 13 (3) (2002) 260–274.

[16] S. Venugopalan, O. Sinnen, Ilp formulations for optimal task scheduling with communication delays on parallel systems, IEEE Transactions on Parallel and Distributed Systems 26 (1) (2015) 142–151.

[17] J. Choi, H. Oh, S. Kim, S. Ha, Executing synchronous dataflow graphs on a spm-based multicore architecture, in: Proceedings of the 49th Annual Design Automation Conference, ACM, 2012, pp. 664–671.

[18] F. A. Omara, M. M. Arafa, Genetic algorithms for task scheduling problem, Journal of Parallel and Distributed Computing 70 (1) (2010) 13–22.

[19] F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, A. Tumeo, Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 29 (6) (2010) 911–924.

[20] D. Sha, H.-H. Lin, A multi-objective pso for job-shop scheduling problems, Expert Systems with Applications 37 (2) (2010) 1065–1070.

[21] L. Wang, G. Zhou, Y. Xu, S. Wang, M. Liu, An effective artificial bee colony algorithm for the flexible job-shop scheduling problem, The International Journal of Advanced Manufacturing Technology 60 (1-4) (2012) 303–315.

[22] M. Geilen, Synchronous dataflow scenarios, ACM Transactions on Embedded Computing Systems 10 (2) (2010) 16.

[23] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, H. Corporaal, Schedule-extended synchronous dataflow graphs, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 32 (10) (2013) 1495–1508.

[24] E. A. Lee, D. G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Transactions on Computers 100 (1) (1987) 24–35.

[25] W. Thies, M. Karczmarek, S. Amarasinghe, Streamit: A language for streaming applications, in: 11th International Symposium on Compiler Construction, Springer, 2002, pp. 179–196.

[26] A. H. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. J. Bekooij, B. D. Theelen, M. Mousavi, Throughput analysis of synchronous data flow graphs, in: Sixth International Conference on Application of Concurrency to System Design, IEEE, 2006, pp. 25–36.

[27] N. Bambha, V. Kianzad, M. Khandelia, S. S. Bhattacharyya, Intermediate representations for design automation of multiprocessor dsp systems, Design Automation for Embedded Systems 7 (4) (2002) 307–323.

[28] M. R. Garey, D. S. Johnson, Computers and intractability: a guide to the theory of NP-completeness. 1979, San Francisco, LA: Freeman, 1979.

[29] Lindo api version 9.0, lindo systems inc.
URL `http://www.lindo.com/`

[30] W. L. Winston, J. B. Goldberg, Operations research: applications and algorithms, Vol. 3, Duxbury press Boston, 2004.

[31] J. P. Hespanha, An efficient matlab algorithm for graph partitioning, University of California. (2004) 1–8.

[32] H. Oh, S. Ha, Fractional rate dataflow model for efficient code synthesis, Journal of VLSI signal processing systems for signal, image and video technology 37 (1) (2004) 41–51.

[33] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, Synthesis of embedded software from synchronous dataflow specifications, Journal of VLSI signal processing systems for signal, image and video technology 21 (2) (1999) 151–166.