Communication-Aware Scheduling Algorithms for Synchronous Dataflow Graphs on Multicore Systems

Mingze Ma and Rizos Sakellariou
School of Computer Science
The University of Manchester, UK
{mingze.ma,rizos}@manchester.ac.uk

ABSTRACT
Synchronous dataflow graphs are widely used to model digital signal processing and multimedia applications. Self-timed execution is an efficient methodology for the analysis and scheduling of synchronous dataflow graphs. In this paper, we propose a communication-aware self-timed execution approach to solve the problem of scheduling synchronous dataflow graphs on multicore systems with communication delays. Based on this communication-aware self-timed execution approach, four communication-aware scheduling algorithms are proposed using different allocation rules. The proposed algorithms are experimentally evaluated in terms of throughput and runtime using realistic applications.

CCS CONCEPTS
• Theory of computation → Scheduling algorithms; • Computer systems organization → Embedded software; • Software and its engineering → Data flow architectures;

KEYWORDS
Synchronous dataflow graphs, scheduling, self-timed execution, multicore systems

ACM Reference Format:

1 INTRODUCTION
Synchronous Dataflow Graphs (SDFGs) [8] are proposed to represent streaming applications that are widely used in the digital signal processing and multimedia fields. Streaming applications process input data streams iteratively and continuously. As a result, the criterion for the performance of a streaming application is not the finishing (execution) time of one iteration of the application, but throughput, which represents the data processing rate of the application. A streaming application consists of functional modules

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SAMOS XVIII, July 15–19, 2018, Pythagorion, Samos Island, Greece © 2018 Association for Computing Machinery. ACM ISBN 978-1-4503-6494-2/18/07 ... $15.00 https://doi.org/10.1145/3229631.3229643

Figure 1: An example SDFG.

which may have different data consuming and producing rates; thus, execution rates of different modules within one iteration may be different. SDFGs are able to describe all these unique features of streaming applications.

An example SDFG is given in Figure 1. The nodes of the graph are called actors; they represent the functional modules of an application. The directed edges between actors are called channels; they represent data dependencies between actors. Communication data is modelled as tokens with a different token size. The token producing and consuming rates of actors are labeled next to the two ends of each channel. The channel with initial tokens (i.e., the black dots on the channel $C \rightarrow A$) represents an inter-iteration data dependency. The initial tokens also enable the existence of cycles in SDFGs and prevent deadlocks.

Compile-time scheduling strategies can be used for the estimation of the performance of an application or the generation of deterministic schedules for an application [1]. A periodic schedule is necessary for the iterative execution of an SDFG. There are two major challenges when scheduling SDFGs. Firstly, the data producing and consuming rates of actors in an SDFG can be different, which means the execution rates of actors in the SDFG can vary. For example, in Figure 1, actor A, B and C have to execute 1, 2 and 2 times within one iteration of the SDFG, respectively. Secondly, cycles may exist in an SDFG, as it is the case in Figure 1. To deal with these two challenges, a transformation based methodology can be used [8]. This methodology converts an SDFG to a Directed Acyclic Graph (DAG), and then applies DAG scheduling algorithms [4] to deal with the converted DAG. The obtained schedule for the converted DAG is used as a periodic schedule for the original SDFG. However, a graph transformation process (i.e., SDFG to DAG) is required for this kind of scheduling methodology, a process that can be very time-consuming under some circumstances. In addition, the size of the obtained DAG can increase dramatically compared with the original SDFG, making the solving time of the scheduling problem even longer. Another problem is that the converted DAG represents only one iteration of the original SDFG. This means that the potential of utilizing parallelism across different iterations of an SDFG is not exploited.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 INTRODUCTION
Synchronous Dataflow Graphs (SDFGs) [8] are proposed to represent streaming applications that are widely used in the digital signal processing and multimedia fields. Streaming applications process input data streams iteratively and continuously. As a result, the criterion for the performance of a streaming application is not the finishing (execution) time of one iteration of the application, but throughput, which represents the data processing rate of the application. A streaming application consists of functional modules

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SAMOS XVIII, July 15–19, 2018, Pythagorion, Samos Island, Greece © 2018 Association for Computing Machinery. ACM ISBN 978-1-4503-6494-2/18/07 ... $15.00 https://doi.org/10.1145/3229631.3229643

Figure 1: An example SDFG.

which may have different data consuming and producing rates; thus, execution rates of different modules within one iteration may be different. SDFGs are able to describe all these unique features of streaming applications.

An example SDFG is given in Figure 1. The nodes of the graph are called actors; they represent the functional modules of an application. The directed edges between actors are called channels; they represent data dependencies between actors. Communication data is modelled as tokens with a different token size. The token producing and consuming rates of actors are labeled next to the two ends of each channel. The channel with initial tokens (i.e., the black dots on the channel $C \rightarrow A$) represents an inter-iteration data dependency. The initial tokens also enable the existence of cycles in SDFGs and prevent deadlocks.

Compile-time scheduling strategies can be used for the estimation of the performance of an application or the generation of deterministic schedules for an application [1]. A periodic schedule is necessary for the iterative execution of an SDFG. There are two major challenges when scheduling SDFGs. Firstly, the data producing and consuming rates of actors in an SDFG can be different, which means the execution rates of actors in the SDFG can vary. For example, in Figure 1, actor A, B and C have to execute 1, 2 and 2 times within one iteration of the SDFG, respectively. Secondly, cycles may exist in an SDFG, as it is the case in Figure 1. To deal with these two challenges, a transformation based methodology can be used [8]. This methodology converts an SDFG to a Directed Acyclic Graph (DAG), and then applies DAG scheduling algorithms [4] to deal with the converted DAG. The obtained schedule for the converted DAG is used as a periodic schedule for the original SDFG. However, a graph transformation process (i.e., SDFG to DAG) is required for this kind of scheduling methodology, a process that can be very time-consuming under some circumstances. In addition, the size of the obtained DAG can increase dramatically compared with the original SDFG, making the solving time of the scheduling problem even longer. Another problem is that the converted DAG represents only one iteration of the original SDFG. This means that the potential of utilizing parallelism across different iterations of an SDFG is not exploited.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 INTRODUCTION
Synchronous Dataflow Graphs (SDFGs) [8] are proposed to represent streaming applications that are widely used in the digital signal processing and multimedia fields. Streaming applications process input data streams iteratively and continuously. As a result, the criterion for the performance of a streaming application is not the finishing (execution) time of one iteration of the application, but throughput, which represents the data processing rate of the application. A streaming application consists of functional modules

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SAMOS XVIII, July 15–19, 2018, Pythagorion, Samos Island, Greece © 2018 Association for Computing Machinery. ACM ISBN 978-1-4503-6494-2/18/07 ... $15.00 https://doi.org/10.1145/3229631.3229643

Figure 1: An example SDFG.
To avoid the drawbacks of the transformation based method, some work (e.g., [2, 3, 10–12, 18, 23–25]) tries to handle SDFGs directly without any conversion. A simple way to do this is to model the problem of scheduling an SDFG as a problem that can be solved with existing solvers. For example, the scheduling problem can be modelled as a linear programming (LP), an integer linear programming (ILP), a constraint programming (CP) or a model-checking problem; representative work that follows each of these approaches to solve the problem can be found in [2], [10], [3] or [13], respectively. Although the LP problem can be solved in polynomial time, the work in [2] does not consider as a constraint the number of processing cores. The approaches based on ILP, CP and model-checking can model an NP-complete problem. However, the runtime of these approaches increases exponentially with the scale of the problem, which limits their applicability when large SDFGs have to be scheduled.

On the other hand, Self-Timed Execution (STE) stemming from [5] appears to be both efficient and effective in solving SDFG scheduling problems as shown in a number of papers [11, 12, 18, 23–25]. When STE is used as a scheduling method, it simulates the execution of an SDFG by using a state transformation based expression. The state of a system is a series of variables which represent its execution state. Changes during the execution of a system are represented by appropriate state transformation. The transforming states are recorded until a periodic execution pattern is found, which leads to a periodic schedule. A static schedule can be obtained by using the starting time of actors within the periodic execution pattern. The non-periodic execution phase before the periodic pattern is called a transient phase. A transient phase of STE actually forms a retiming process [23]. A retiming technique [9] enables the execution of some actors before the beginning of the periodic execution phase, something that redistributes the initial tokens in the original SDFG. The periodic pattern may contain multiple iterations of an SDFG, a feature that unfolds the original SDFG. The retiming and unfolding features embedded in STE are always helpful in improving the performance of STE scheduling algorithms [11, 23–25]. In [5], STE shows the ability to analyze the Maximum Cycle Mean (MCM) of an SDFG efficiently; this determines the maximum throughput of the graph. STE based scheduling is used in [18] to explore the trade-offs in buffer requirements and throughput for SDFGs. For SDFG scheduling, STE is adopted in [11, 18, 24, 25] to decrease the buffer use of the scheduling results and in [23, 25] to improve the processor utility of the scheduling results. However, in all this work communication between cores is not taken into account.

To the best of our knowledge, STE has not been used in prior work to deal with the problem of scheduling SDFGs on multicore systems with communication delays between the cores. As shown in [22], effective retiming can improve the throughput of an SDFG by overlapping the inter-actor communication time with the computing time of actors of the application. Considering that the transient phase in STE can lead to effective retiming of the original SDFG [23–25], we argue that STE is potentially suitable for scheduling problems considering communication delays as it allows us to overlap computation with communication. Thus, in this paper, we use STE based algorithms to propose communication-aware scheduling solutions for SDFGs. The throughput of a system is expected to be enhanced by the retiming in the STE process while no graph transformation process is required.

To provide a solution to the SDFG communication-aware scheduling problem, two questions need to be investigated further.

Q1: Does STE still work well when inter-core communication time has been taken into consideration? Compared with a multicore system that does not consider communication time between cores, an extra set of variables is required to describe the communication states of a multicore system with communication delays. An extra set of variables will imply an increase in the number of states; then, the periodic execution pattern becomes harder to find. Furthermore, for STE, the execution rule for actors is to execute an actor as-soon-as-possible (ASAP). This rule works well when communication delays are ignored. However, no previous work shows or proves that STE scheduling can still find easily periodic schedules when communication delays are taken into consideration. Regardless of whether the rule of executing an actor ASAP can work, this raises a second interesting question.

Q2: Can execution rules other than the ASAP rule be applied to the STE process and give better results? The scheduling process of STE is similar to list scheduling for DAGs. For list scheduling of DAGs, executing an actor ASAP is called Earliest Ready Task (ERT) scheduling [7]. Besides the ASAP rule used in ERT, the execution rules used in some of the most successful list scheduling algorithms like Dynamic Level Scheduling (DLS) [15] or Heterogeneous Earliest Finish Time (HEFT) [21] may be potentially better alternatives of the ASAP rule for STE and may deserve further investigation.

In this paper, both questions Q1 and Q2 are answered. The contributions of this paper are:

- A generic communication-aware STE scheduling approach.
- A total of four execution rules (including ASAP) are applied to the generic communication-aware STE approach to propose communication-aware STE scheduling algorithms.
- A thorough experimental investigation assesses the proposed algorithms providing answers to Q1 and Q2 and showing that the proposed STE scheduling algorithms generally outperform two representative list scheduling DAG scheduling algorithms.

In the remainder of the paper, the basic concepts and a problem description are given in Section 2. Then, a motivational example to illustrate the advantages of an STE based approach over the transformation based approach is given. A generic scheduling approach leading to the proposed four STE scheduling algorithms is described in Section 4, followed by an experimental evaluation in Section 5. Finally, Section 6 concludes the paper and gives suggestions for future work.

2 BACKGROUND

2.1 Synchronous Dataflow Graph

An SDFG is a directed cyclic graph which can be described as a tuple $G = (V, E)$, where $V$ is the set of vertices representing actors and $E$ is the set of directed edges representing channels. The SDFG related notations are listed in the first section of Table 1. Most notations are self-explanatory; we only focus on $T(a)$ and $JT(ch)$. Since the token producing rate and consuming rate of actors can...
Within one iteration is called tokens. Inappropriate settings of either the number of initial tokens input for actors on the cycles. The black nodes in Figure 1 are initial execution of actors across different iterations and provide initial SDFG, initial tokens are used to represent data dependencies of the deadlock-free for the SDFG to be sample-rate consistent [8].

Another condition for the existence of a periodic schedule is another positive integer solution exists for all the actors in an SDFG. The minimum positive integer solution for actor  then this SDFG is periodic. A periodic schedule of an SDFG can have a periodic hardware related notations are given in the second section of Table 1. The processing cores are homogeneous. All the processing cores within a multicore processor are connected to a shared bus. Any core can communicate with all other cores through the bus. The bus can only support the communication of one pair of cores at a time, which means once the bus is occupied by some communication, all other inter-core communications are blocked.

Data transfer on the bus is managed by a bus arbiter. We assume that the communication operations from cores do not interrupt the computation on the cores. This enables the overlapping of computation and communication. The communication data is modelled as tokens. The duration of communication is calculated by Eq. 2.

\[
T_{com}(\alpha, n_t) = \frac{TS(\alpha) \times n_t}{BW}
\]  
(2)

where \(T_{com}(\alpha, n_t)\) is the duration of communication to transfer \(n_t\) tokens of size \(TS(\alpha)\) between two different cores and \(BW\) is the bandwidth of the bus. The duration of communication between two actors on the same core is assumed to be zero.

The processing cores are represented by a vector \(C\). Other hardware related notations are given in the second section of Table 1. As the tokens produced by the source actor of a channel may not be consumed immediately, the tokens produced by multiple execution of the source actors may accumulate on the channel. To guarantee the correctness of the execution of an SDFG, the destination actor of the channel has to consume the earliest produced tokens first. To guarantee the correctness of the execution of an SDFG, the destination actor of the channel has to consume the earliest produced tokens first. Therefore, token transfer on a channel is modelled using a first-in-first-out (FIFO) buffer on the channel. The basic elements on buffers are token blocks. A token block on a channel is produced by one executing iteration. The minimum positive integer solution for actor  then this SDFG is periodic. A periodic schedule of an SDFG can have a periodic hardware related notations are given in the second section of Table 1. The processing cores are homogeneous. All the processing cores within a multicore processor are connected to a shared bus. Any core can communicate with all other cores through the bus. The bus can only support the communication of one pair of cores at a time, which means once the bus is occupied by some communication, all other inter-core communications are blocked.

Data transfer on the bus is managed by a bus arbiter. We assume that the communication operations from cores do not interrupt the computation on the cores. This enables the overlapping of computation and communication. The communication data is modelled as tokens. The duration of communication is calculated by Eq. 2.

\[
T_{com}(\alpha, n_t) = \frac{TS(\alpha) \times n_t}{BW}
\]  
(2)

where \(T_{com}(\alpha, n_t)\) is the duration of communication to transfer \(n_t\) tokens of size \(TS(\alpha)\) between two different cores and \(BW\) is the bandwidth of the bus. The duration of communication between two actors on the same core is assumed to be zero.

The processing cores are represented by a vector \(C\). Other hardware related notations are given in the second section of Table 1. As the tokens produced by the source actor of a channel may not be consumed immediately, the tokens produced by multiple execution of the source actors may accumulate on the channel. To guarantee the correctness of the execution of an SDFG, the destination actor of the channel has to consume the earliest produced tokens first. Therefore, token transfer on a channel is modelled using a first-in-first-out (FIFO) buffer on the channel. The basic elements on buffers are token blocks. A token block on a channel is produced by one executing iteration of the source actors on the channel.
2.3 Objective and Criteria

The notations related to the objectives of the scheduling problem are listed in the third section of Table 1. The objective of a communication-aware SDFG scheduling algorithm is to obtain a periodic schedule \( S(G) \) of an SDFG \( G \). If a retiming process exists, the retiming scheme \( RT(G) \) should also be output as part of the result. The criteria to assess the obtained schedules are their throughput, \( TH \), and the runtime of the scheduling algorithms. Note that \( S(G) \) obtained by STE may contain the execution of more than one iteration of an SDFG, which is equivalent with unfolding the original SDFG multiple times. The number of iterations in \( S(G) \) is called unfolding factor, which is denoted by \( f(S) \). Although unfolding an SDFG \( G \) increases the size of the SDFG by \( f(S) \) times, the average period of one iteration can always be decreased. In this paper, the throughput of a schedule \( S(G) \) is calculated by Eq. 3.

\[
TH(S) = \frac{f(S)}{|S(G)|}
\]  

(3)

3 MOTIVATIONAL EXAMPLE

In this section, we use the SDFG in Figure 1 to show the advantages of the STE based approach compared to the transformation based approach. The SDFG in Figure 1 is assumed to be scheduled onto a two-core system with a bus bandwidth of 1 (token size per time unit). The execution time of actor A, B and C is 20, 10 and 10 time units, respectively. The token size on all channels is 5.

For the transformation based approach, an SDFG has to be converted to a DAG, and then a scheduling algorithm for DAGs may be used to schedule the converted DAG. The converted DAG for the SDFG in Figure 1 is shown in Figure 3(a), where actors B and C both have two copies B1, B2 and C1, C2, respectively. An optimal schedule for the DAG in Figure 3 is shown in Figure 3(b). The finishing time of this schedule is 45 time units. When this schedule is used for the periodic execution of the original SDFG, the throughput of this schedule is \( 1/45 \approx 0.022 \).

In contrast to the transformation based approach, the STE based approach directly uses the original SDFG to carry out scheduling. A schedule for the SDFG in Figure 1 obtained by \( \text{ERAS} \), one of the proposed STE based scheduling algorithms in the next section, is shown in Figure 4. A transient phase and a periodic phase can be found in this schedule. The transient phase is only performed once at the beginning of the schedule, while the periodic phase is then iteratively used for the following execution of the SDFG in Figure 1. The throughput of the schedule is decided by the period of the periodic phase. The period of the periodic phase is 60 time units, and there are two iterations of the SDFG in Figure 1 in one periodic phase. Thus, the throughput of the schedule in Figure 4 is \( 2/60 = 0.033 \), which is 50% higher than the throughput of the optimal transformation based scheduling approach.

The throughput improvement is gained from two techniques: retiming and unfolding. The transient phase of the schedule in Figure 4 is a retiming process, which enables some actors to be executed before the periodic phase, turning some intra-iteration
data dependencies into inter-iteration data dependencies. The inter-
iteration data dependencies do not matter when scheduling one iter-
atron of an SDFG. Furthermore, inter-iteration data communication
is much easier to overlap with actor execution in a single iteration as
opposed to finding overlaps for intra-iteration data communication.
On the other hand, there are two iterations in the periodic phase of
the schedule in Figure 4, which unfolds the original SDFG with an
unfolding factor of 2. The unfolding technique makes the utilization
of parallelism across multiple iterations of an SDFG possible, which
is always helpful to improve throughput.

As a result of the existence of the retiming and unfolding process
in STE based scheduling, the throughput obtained is expected to
be higher than the throughput obtained by transformation based
approaches. In addition, STE based approaches do not need to use
the SDFG-to-DAG conversion process, which is time-consuming
and may enlarge exponentially the size of the original SDFG [17]. In
what follows, we use STE based approaches to propose algorithms
for the communication-aware scheduling problem for SDFGs on
multicore systems.

4 ALGORITHMS

4.1 Overview

STE based scheduling is a greedy strategy essentially similar to ERT
scheduling for DAGs [7], which always schedules the actor with
the earliest starting time in each scheduling step. The difference
between ERT scheduling and STE based scheduling is that the
latter does not remove an actor from the scheduling list even after
all invocations of this actor in one iteration have been scheduled.
This allows the overlapping of multiple iterations of an SDFG’s
execution, something that enables the consideration of retiming
and unfolding in STE based scheduling. The STE process stops
when a periodic execution pattern is found.

In this section, we first extend the original STE to address the
problem of communication-aware scheduling for SDFGs. After that,
four scheduling algorithms are proposed based on four actor-to-
core allocation rules.

4.2 Communication-Aware Self-Timed Execution

Firstly, we give the description of the generic communication-aware
STE scheduling approach proposed in this paper. The proposed
scheduling algorithms are based on this generic communication-
aware STE scheduling approach. The STE process is a simulation
of the execution of an SDFG on a multicore system. The execution
process is modelled as a transformation of STATES of the system.
Different from previous work, which does not consider communication
delays and ignores the interconnection architecture, for our
communication-aware scheduling the state of communication is a
significant part of the STATE of the whole system.

For the bus-based-platform adopted in this paper, a STATE is
described as a tuple \( STATE = (IT, R, B, BUS) \), \( IT \) and \( B \) are two
vectors of channels as shown in Table 1. As explained in Section 2.2,
each element \( B(ch) \) in \( B \) is a FIFO buffer. A token block \( b \) in the buffer
\( B(ch) \) is a tuple \( (num, et, cid) \), where \( b(num) \), \( b(et) \) and \( b(cid) \) are all
explained in Table 1. \( R \) is a vector of cores in which each element
\( R(c) \) is a list of tuples \( (aid, rt) \), where \( aid \) is an actor currently

Algorithm 1 Generic communication-aware STE scheduling

**Input:**
An SDFG \( G = (V, E) \), a vector of cores \( C \), and the bandwidth of
the bus \( BW \);

**Output:**
A retiming scheme \( RT(G) \) and a periodic schedule \( S(G) \);

**Iteration:**
1. Define \( SL \) as an empty list of \( STATE \) and \( CS \) as the initial \( STATE \)
of the system;
2. while \( CS \notin SL \) do
3. Insert \( CS \) to the end of the list \( SL \);
4. Get all the free actors and insert them into a list \( FA \);
5. Use an actor-to-core allocation rule in Section 4.3 to decide the
allocation of the actors in \( FA \) to the processing cores;
6. Use Algorithm 2 to perform the actor-to-core allocation
operation;
7. Clock proceeds;
8. end while
9. return \( RT(G) \) as the sequence in \( SL \) from the start of \( SL \) to the
element which equals to \( CS \), and \( S(G) \) as the sequence in \( SL \)
from the element which equals to \( CS \) to the end of \( SL \);

allocated on core \( c \) and \( rt \) is the remaining time of the execution
of the actor \( aid \) on core \( c \). The list \( R(c) \) is sorted in ascending order
of \( rt \). \( BUS \) is a list of unoccupied time slots on the shared bus. We
use \( slot = (st, le) \) to represent one time slot in \( BUS \), where \( st \) is the
starting time of \( slot \) and \( le \) is the duration of \( slot \). The list \( BUS \) is
sorted in ascending order of \( slot(st) \).

We are now in a position to describe the generic communication-
aware STE scheduling algorithm, shown in Algorithm 1. Line 1 of
the algorithm simply initializes a \( STATE \) list \( SL \) and a current
\( STATE \) \( CS \). One thing to notice is the initial value of \( BUS \) is a list
with only one \( slot \), where \( slot(st) = -\infty \) and \( slot(le) = +\infty \). The
loop body in the algorithm keeps simulating the execution of an
SDFG on a given hardware platform under a certain actor-to-core
allocation rule. The current \( STATE \) \( CS \) is changed in lines 4-7. The past
\( STATEs \) of the system are recorded in the list \( SL \) in line 3. If the
current \( STATE \) \( CS \) equals to a \( STATE \) in \( SL \), this means the
system will keep repeating the \( STATE \) sequence from the element
equal to \( CS \) to the end of the list \( SL \) in the execution that follows.
Therefore, this sequence can be used as a periodic schedule, while
the sequence before the periodic sequence is the retiming scheme.
The equivalent condition of two \( STATEs \) is given in Definition 4.1.

**Definition 4.1.** Two \( STATEs \) are equivalent if and only if all the
elements in the tuples \( (IT, R, B, BUS) \) of the two \( STATEs \) are equal.

In line 4 of Algorithm 1, \textit{free actors} are the actors which have
enough input tokens to execute at least once. We give the definition of
\textit{free actors} in Definition 4.2.

**Definition 4.2.** An actor \( \alpha \) is a free actor if and only if
\[ \forall ch \in IN(\alpha), \sum_{b \in B(ch) | b(et) \geq 0} b(num) \geq Q(ch) + IT(ch). \]

The actor-to-core allocation rule in line 5 of Algorithm 1 is the key for an STE scheduling algorithm. Different rules are
Algorithm 2 Actor-to-core allocation

Input:
The current STATE CS = (IT, R, B, BUS), an actor α and a core c;

Output:
The earliest ready time $t_εr$ of α on c;

Iteration:
1: Define $t_εr$ as the earliest starting time of actor α on core c and initialize $t_εr$ as 0;
2: for all $ch \in IN(α)$ do
3: Define $tcnt$ as the number of tokens that has been counted and initialize $tcnt$ as 0;
4: while $tcnt < Q(ch)$ do
5: Set first element of $B(ch)$ as $b$, and define the number of tokens to transfer as $tnum$;
6: if $IT(ch) > 0$ then
7: $tcnt \leftarrow \min(Q(ch), IT(ch))$;
8: $IT(ch) \leftarrow IT(ch) - tcnt$;
9: end if
10: $tnum \leftarrow \min(b(num), Q(ch) - tcnt)$;
11: $b(num) \leftarrow b(num) - tnum; tcnt = tcnt + tnum$;
12: if $b(num) = 0$, pop the first element $b$ from $B(ch)$;
13: Get the earliest arriving time of the tokens to be transferred as $t_r$ using Algorithm 3;
14: $t_εr \leftarrow \max(t_εr, t_r)$;
15: end while
16: end for
17: Get the longest remaining time in $R(c)$ as $rt$; if $R(c) = \emptyset$, set $rt = 0$;
18: $t_εr \leftarrow \max(t_εr, rt)$;
19: Set $rt' = rt + ET(α)$ and insert $(α, rt')$ to the end of the list $R(c)$, where $rt'$ is the remaining execution time of $α$ on core $c$;
20: for all $ch \in OUT(α)$ do
21: Insert $b = (tnum, et, cid)$ to the end of the list $B(ch)$, where $b(num) = P(ch), b(et) = -rt'$, and $b(cid) = c$;
22: end for
23: return $t_εr$;

Algorithm 3 Get the earliest arriving time

Input:
The token block of the tokens transferred from $b$; the number of the tokens to be transferred $tnum$; and the core to transfer to $c$;

Output:
The earliest arriving time, $t_a$;

Iteration:
1: if $b(cid) = c$ then
2: $t_a \leftarrow -b(et)$;
3: else
4: Get communication duration $cd \leftarrow [tnum * TS(ch)/BW]$;
5: for all slot ∈ BUS do
6: Set $sed \leftarrow \text{slot}(st) + \text{slot}(le)$ and $ted \leftarrow -b(et) + cd$;
7: if $(\text{slot}(st) \leq -b(et)$ and $sed \geq (ted)$) or $(\text{slot}(st) > -b(ett) \text{ and } slot(le) \geq cd)$ then
8: $t_a \leftarrow \max(slot(st), -b(et) + cd)$;
9: Remove the time period in slot occupied by the communication; if the communication occupied a middle period in slot, then slot should be split into two separate slots in BUS;
10: break;
11: end if
12: end for
13: end if
14: return $t_a$

introduced in Section 4.3. In the current section, we only introduce the operation of allocating an actor to a core, which is shown in Algorithm 2. The first for-loop in Algorithm 2 (lines 2-16) is used to get the earliest arriving time of all the input tokens for one execution of an actor $α$ on a core $c$. In the meantime, the input tokens are also consumed from their buffers. Then, in line 18, the earliest available time of core $c$ is compared with the earliest arriving time of the input tokens to get the earliest starting time for the actor. After that, the remaining time of the execution of the actor on $c$ is inserted to the end of the list $R(c)$. In the second for-loop (lines 22-23), the token blocks to be generated are appended at the end of the buffers of the output channels of actor $α$ with a negative existing time, which indicates the remaining time of the token blocks to be generated.

In line 13 of Algorithm 2, the earliest arriving time is hardware-dependent. For the bus-based interconnection, the earliest arriving time can be obtained by Algorithm 3. As shown in lines 1 and 2 of Algorithm 3, if the destination core $c$ is the same as the core where the token block $b$ exists, the communication duration is 0, and then the earliest arriving time $t_a$ is $-b(et)$, where $-b(et)$ is the earliest possible starting time of the communication. Otherwise, the communication duration is obtained by Eq. 2 assigning a value to $cd$ in line 4. The for-loop from line 5 to line 12 in Algorithm 3 is used to find a suitable time slot on the bus for the communication. The communication period must be contained in the slot. Then, the earliest arriving time is $t_a = \max(slot(st), -b(et) + cd)$.

In line 6 of Algorithm 1, all end-ready actors are removed from $R$ of CS. The definition of end-ready actors is given in Definition 4.3.

Definition 4.3. For a tuple $(α, rt) \in R(c)$, if $rt = 0$, the actor $α$ is end-ready on core $c$.

The last step in the while-loop in Algorithm 1 is clock increase (‘clock proceeds’). The clock step is the minimum time step that can change the STATE of the system. For the actors on cores, the minimum clock step is

$$\text{Step}_α = \min_{\forall r \in R[|r| > 0]} \left( r_0(rt) \right),$$

where $r_0$ is the first element in the list $r$. While for tokens on channels, the minimum step is

$$\text{Step}_{ch} = \min_{\forall c \in C \forall r \in R(c), r(rt) = r(rt) - clk} \left( \min_{\forall b \in B(ch)} \min((b(et) < 0) - b(et)) \right).$$

Then, the minimum clock step of the whole system is obtained as $clk = \min(\text{Step}_α, \text{Step}_{ch})$. All the time-related variables in CS change as the clock proceeds, including

- $\forall c \in C \forall r \in R(c), r(rt) = r(rt) - clk$;  
- $\forall ch \in E \forall b \in B(ch), b(et) = b(et) + clk$;  
- $\forall slot \in BUS, slot(st) = slot(st) - clk$.  

For the bus-based communication, another important operation in the clock increase step is to remove unnecessary time slots in BUS. Let \( b(\text{et})_{\text{max}} \) be the longest existing time of all token blocks on buffers of the channels. Then, time slots earlier than \( b(\text{et})_{\text{max}} \) will never be used during the subsequent execution process and therefore can be removed. The removing operation is \( V\text{slot} \in \text{BUS}, \)

- if \( \text{slot}(\text{st}) + \text{slot}(\text{le}) \leq -b(\text{et})_{\text{max}} \), remove \( \text{slot} \) from \( \text{BUS} \);
- else if \( \text{slot}(\text{st}) < -b(\text{et})_{\text{max}} \), set \( \text{slot}(\text{le}) = \text{slot}(\text{le}) - b(\text{et})_{\text{max}} - \text{slot}(\text{st}) \) and \( \text{slot}(\text{st}) = -b(\text{et})_{\text{max}} \), in this order.

If the condition \( \text{CS} \not\in \text{SL} \) of the while-loop in Algorithm 1 is not met, then a periodic \( \text{STATE} \) list is found. As shown in line 9 of Algorithm 1, a periodic schedule \( S(G) \) and a retiming scheme \( R(T(G)) \) are obtained at last.

### 4.3 Scheduling Algorithms

Based on the generic communication-aware STE scheduling algorithm, shown in Algorithm 1, we construct four scheduling algorithms by using four actor-to-core allocation rules in line 5 of Algorithm 1.

**Earliest ready actor scheduling** (ERAS) uses the ASAP rule of the ERT algorithm for DAG scheduling [7]. The earliest ready time of an actor \( \alpha \) on a core \( c \) can be obtained by Algorithm 2 (when Algorithm 2 is only used to get the earliest ready time for an actor, the current \( \text{STATE} \) \( \text{CS} = (\text{IT}, R, B, \text{BUS}) \) should not be varied in the algorithm). Communication delays have been taken into consideration to get the earliest ready time of actors. However, only unoccupied cores can be used during the actor allocation process. We refer to this kind of allocation strategy as **multiple allocation enabled scheduling (MADS)**. Here, we define a core \( c \) as unoccupied if \( R(c) = \emptyset \), otherwise it is occupied.

The allocation process of ERAS is given as follows.

- **Step 1**: Select a pair of a free actor \( \alpha \) and an unoccupied core \( c \) with the earliest ready time among all possible combinations of free actors and unoccupied cores;
- **Step 2**: Allocate \( \alpha \) on \( c \), remove \( \alpha \) from \( FA \) and label \( c \) as an occupied core;
- **Step 3**: Repeat the former two steps until no free actor or unoccupied core exists.

As an example, the SDFG in Figure 1 is used to illustrate the scheduling process of ERAS in Figure 4. In Figure 4, \( CTA_B \) indicates the communication time between actor A and B on the bus. The \( 0 \rightarrow 1 \) under \( CTA_B \) means that the communication is from core 0 to core 1. The SDFG executes according to the execution rule of ERAS as shown in Figure 4. A periodic phase is obtained after a transient phase. The transient phase forms the retiming process of the SDFG; and the periodic phase can then be used as a periodic schedule.

**Earliest finishing actor scheduling** (EFAS) adopts a rule similar to ERAS except that instead of the earliest ready time, the earliest finishing time of actors on cores is used as the scheduling priority. The earliest finishing time \( \text{ef}(\alpha) \) of an actor \( \alpha \) on core \( c \) can be obtained by \( \text{ef}(\alpha) = \text{ef}(\alpha) + \text{ET}(\alpha) \). By replacing the earliest ready time with the earliest finishing time in Step 1 of the allocation process of ERAS, the allocation process of EFAS is obtained. EFAS also belongs to the MADS family.

For DAGs, a scheduling algorithm called DLS, proposed in [15], enables the multiple allocation of actors on a core. This kind of allocation strategy is referred to as **multiple allocation enabled scheduling (MAES)** in this paper. MAES allows the free actors to be allocated on all cores in \( C \) including the occupied cores (by allocating them after the execution of existing actors on the core). As shown in [15], this rule can improve the performance of list-based scheduling for DAGs as the communication time is better handled compared with the MADS rule. Based on this fact, we can expect that applying MAES rules in STE based approaches for SDFGs can also result in better performance than the traditional MADS rule ASAP. In this paper, two MAES algorithms for SDFGs are proposed: **multiple allocation earliest ready actor scheduling** (MERAS) and **multiple allocation earliest finishing actor scheduling** (MEFAS). MERAS and MEFAS are the MAES versions of ERAS and EFAS, respectively. The allocation process of MERAS/MEFAS is given below.

- **Step 1**: Select a pair of a free actor \( \alpha \) and a core \( c \) with the earliest ready time (for MERAS) or earliest finishing time (for MEFAS) among all possible combinations of free actors and all cores;
- **Step 2**: Allocate \( \alpha \) on \( c \), remove \( \alpha \) from \( FA \) and label \( c \) as an occupied core;
- **Step 3**: Repeat the former two steps until no free actor exists.

The time complexity of the STE scheduling algorithms is dependent on the SDFG. As shown in Algorithm 1, the end situation of the while-loop is that a \( \text{STATE} \) in list \( SL \) is found to be equal with \( CS \). As mentioned in [5], two equivalent \( \text{STATEs} \) are guaranteed to be found since the number of \( \text{STATEs} \) for a system is finite. Fortunately, in practice, for STE scheduling without consideration of communication delays, the runtime is always much faster than the worst-case of traversing all possible \( \text{STATEs} \) as shown in [5]. When communication time is taken into consideration, the number of \( \text{STATEs} \) of a system increases by some orders of magnitude, although the number of \( \text{STATEs} \) is still finite. If the runtime of STE scheduling becomes unacceptably long as the number of \( \text{STATEs} \) explodes, the answer to \( Q_1 \) is ‘no’ and STE-based scheduling methods are not suitable for communication-aware scheduling of SDFGs. In Section 5, we use detailed experimental evaluation to answer this question.

### 5 EXPERIMENTAL EVALUATION

#### 5.1 Settings

In this section, we evaluate the performance and runtime of the four proposed STE scheduling algorithms by comparing them with two representative communication-aware DAG scheduling algorithms DLS and EFT [4]. EFT is a homogeneous version of the HEFT algorithm proposed in [21]. The runtime of all algorithms is obtained on a workstation with an Intel Xeon E5-2667V3 processor and 32 GB RAM. All the algorithms in the experiment are implemented and evaluated within SDF3 [14, 19]. A time limit of 10 minutes is applied to all the algorithms in the experiment to stop the execution of algorithms that run for too long. In practice, this applies only to DLS and EFT, which have difficulty to build a schedule in some cases that will be explored next. In most other cases, the runtime of each algorithm is less than 1 second.
Table 2: The number of actors \( N \) and the sum of the minimum repetition number of actors \( SR \) for each of the 12 realistic SDFGs used in the experiments.

<table>
<thead>
<tr>
<th>Application</th>
<th>( N )</th>
<th>( SR )</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeamFormer</td>
<td>56</td>
<td>106</td>
</tr>
<tr>
<td>BitonicSort</td>
<td>40</td>
<td>64</td>
</tr>
<tr>
<td>ChannelVocoder</td>
<td>55</td>
<td>1835</td>
</tr>
<tr>
<td>DCT</td>
<td>8</td>
<td>1058</td>
</tr>
<tr>
<td>DES</td>
<td>53</td>
<td>991</td>
</tr>
<tr>
<td>FFT</td>
<td>17</td>
<td>1406</td>
</tr>
<tr>
<td>FilterBank</td>
<td>85</td>
<td>512</td>
</tr>
<tr>
<td>FMRadio</td>
<td>43</td>
<td>47</td>
</tr>
<tr>
<td>MPEGDecoder</td>
<td>23</td>
<td>2177</td>
</tr>
<tr>
<td>Serpent</td>
<td>120</td>
<td>6355</td>
</tr>
<tr>
<td>TDE</td>
<td>29</td>
<td>5027</td>
</tr>
<tr>
<td>Vocoder</td>
<td>114</td>
<td>404</td>
</tr>
</tbody>
</table>

All SDFGs in the experimental evaluation are assumed to run on a bus-based homogeneous multicore processor. The processor runs at 500MHz and the bandwidth of the bus in the processor is 8GB/s. The parameters of the processor are similar to the multicore DSP processor TMS320C6472 made by TI. The same data set used in [6] is used in the experimental evaluation, which consists of 12 realistic applications. The 12 realistic applications are all obtained from the folder streamit-rc-2.1/apps/benchmarks/asplos06 of StreamIt 2.1.1 (Older release) [16]. They are denoted by BeamFormer, BitonicSort, ChannelVocoder, DCT, DES, FFT, FilterBank, FMRadio, MPEG2Decoder, Serpent, TDE and Vocoder. For each one of the 12 SDFGs, the number of actors, \( N \), and the sum of the minimum repetition number of actors, \( SR \), are shown in Table 2. These applications are written in StreamIt language [16, 20], where a filter can be viewed as an actor. Thus, an application can be extracted as an SDFG through the functions provided in StreamIt, including the estimation for the execution time of actors in clock cycles and for the token size on channels in bytes. The push and pop rates of filters are used as the producing and consuming rates of actors, respectively. The peak operation is not considered in our experiments, focusing on producing and consuming rates only. All actors are assumed to be stateless, which means all actors are duplicable to multiple cores.

If an SDFG is not a strongly connected graph, which means some actors in the graph have only input channels or only output channels, the graph should be converted to a strongly connected graph before the STE scheduling process, as mentioned in [24].

5.2 Results

Two criteria are used for the evaluation of the algorithms in the experiment. The first criterion is throughput speedup of an algorithm: this is the throughput obtained using multiple cores divided by the throughput of a single core baseline. The second criterion is the runtime of an algorithm. The throughput of an SDFG \( G = (V, E) \) on a single core is calculated by

\[
TH_{\text{single}}(G) = \frac{1}{\sum_{a \in V} ET(a) \times Y(a)}.
\]

Then, if the throughput obtained by a schedule \( S \) for an SDFG \( G \) using \( M \) cores is \( TH_M(S) \), the speedup is \( \frac{TH_M(S)}{TH_{\text{single}}(G)} \). For the runtime, to smooth out minor deviations between runs, we run each algorithm ten times for each application and take the average runtime of these executions.

The speedup and runtime obtained by all the algorithms on 2, 4, 8, and 16 cores for the 12 realistic SDFGs are given in Table 3. Since eft and dls cannot finish within the time limit for DES, MPEG2Decoder, Serpent and TDE, the speedup and runtime for these cases are labeled as “—” in Table 3. The runtime of an SDFG-to-HSDFG conversion is also included in the runtime of eft and dls. The function transformSDFtoHSDF(SDFgraph* G) in SDF3 [14, 19] is used to do the conversion.

As shown in Table 3, the proposed STE based algorithms get equal or better speedup than the transformation based algorithms eft and dls in most cases. When the number of cores is small, such as 2 or 4, all the algorithms used in the experiment can get near-optimal speedup in most cases and the advantage of the proposed algorithms is not obvious. When the number of cores is large, such as 8 or 16, the speedup obtained by the proposed algorithms appears to become much higher than the speedup obtained by eft and dls (cf. BitonicSort, FMRadio, and to some extent BeamFormer). We also note that, as expected from the discussion in Section 4.3, the performance of the proposed MAES algorithms meras and mefas is generally better than the performance of the proposed MADS algorithms eras and efas.

For the eight realistic applications which can be completed by all algorithms within the time limit, the average speedup improvement of each algorithm normalized by the speedup of dls, for each different number of cores, is shown in Figure 5. It can be seen that, on average, the proposed algorithms achieve better speedup than dls and eft. It appears that the average speedup improvement of the proposed algorithms increases with the number of cores. The two best MAES algorithms, meras and mefas, achieve on average about 50% higher speedup than dls when the number of cores is 16.
Table 3: Speedup and runtime for all algorithms and applications using 2, 4, 8 and 16 cores.

<table>
<thead>
<tr>
<th>Application</th>
<th>#Core</th>
<th>Speedup</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>ERAS</td>
<td>EFAS</td>
</tr>
<tr>
<td>BeamFormer</td>
<td>2</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.00</td>
<td>3.97</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7.60</td>
<td>7.62</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.30</td>
<td>2.29</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.98</td>
<td>4.97</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>12.38</td>
<td>9.45</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>11.56</td>
<td>11.56</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.30</td>
<td>2.29</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.98</td>
<td>4.97</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>12.38</td>
<td>9.45</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>11.56</td>
<td>11.56</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.30</td>
<td>2.29</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.98</td>
<td>4.97</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>12.38</td>
<td>9.45</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>11.56</td>
<td>11.56</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.30</td>
<td>2.29</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.98</td>
<td>4.97</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>12.38</td>
<td>9.45</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>11.56</td>
<td>11.56</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.30</td>
<td>2.29</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.98</td>
<td>4.97</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>12.38</td>
<td>9.45</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>11.56</td>
<td>11.56</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.30</td>
<td>2.29</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.98</td>
<td>4.97</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>12.38</td>
<td>9.45</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>11.56</td>
<td>11.56</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.30</td>
<td>2.29</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.98</td>
<td>4.97</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>12.38</td>
<td>9.45</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>11.56</td>
<td>11.56</td>
</tr>
</tbody>
</table>

In terms of the runtime, all the proposed algorithms can finish within the time limit. However, the transformation based algorithms EFT and DLS cannot finish the schedule for four applications within the time limit. In total, as shown in Table 3, the runtime of BitonicSort is 2177, the runtime of TDE is 5027, and the runtime of MPEGdecoder is 404. Out of these 8 applications, 7 applications have their runtime smaller when using EFT and DLS compared to the proposed STE based algorithms. The SR of BeamFormer is 106, the SR of BitonicSort is 64 and the SR of MPEGdecoder is 47, values which are around or less than 100, and are close to the values of N for these applications. This suggests that the runtime of EFT and DLS is significantly affected by the value of SR of an SDFG. This is because the SDFG-to-DAG conversion increases the number of actors from N in the original SDFG to SR in the converted DAG. Therefore, if the value of SR is much higher than N, the runtime of DAG scheduling algorithms becomes much longer than the runtime larger than the values of their number of actors, N.
of the proposed algorithms which directly use the original SDFGs for the scheduling.

For the four applications having a low EFT or DLS runtime (less than 500ms), the runtime of the SDFG-to-DAG conversion constitutes a significant part of the overall runtime. For example, when the number of cores is 2, for BeamFormer, FilterBank and FMRadio, the SDFG-to-DAG conversion occupies up to 15%, 67% and 58% of the total runtime of EFT, respectively. A special case is BitonicSort, which also has a short runtime, but the SDFG-to-DAG conversion takes less than 10% of the total runtime. This is because the values of N and SR of BitonicSort are close to each other, which means the SDFG-to-DAG conversion for BitonicSort is relatively simple, hence the runtime of the conversion is low. For the remaining eight applications considered in the experiments, the percentage of the runtime consumed by the SDFG-to-DAG conversion is less than 10% of the total runtime of EFT and DLS.

Finally, we note that the runtime of the MAES algorithms is longer than the runtime of the MADS algorithms in most cases when the number of cores is 16. This is because MAES algorithms have to check the earliest ready or finishing time of free actors on all cores while MAES algorithms only need to check the unoccupied cores.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose a communication-aware STE approach to schedule SDFGs on a multicore system with communication delays. Four different actor-to-core allocation rules are used to form four scheduling algorithms. Through the experimental evaluation, we can answer the two questions raised in the introduction. With respect to Q1, STE based scheduling algorithms are still applicable to communication-aware scheduling problems for SDFGs as, in all cases, STE scheduling algorithms can finish within the time limit in the experiment. For Q2, two MAES scheduling rules have been successfully embedded to the generic STE scheduling approach and were tested with higher performance than the traditional MADS rule ASAP. On average, the four STE scheduling algorithms always outperform DLS and EFT in the experiment in terms of throughput.

The runtime of the proposed algorithms is much lower than DLS and EFT in 8 of the 12 realistic applications in the experiment. Overall, our analysis suggests that the algorithms proposed in this paper have characteristics that outperform transformation-based approaches, which may not be a good strategy to use particularly when the value of the sum of the minimum repetition number of actors is relatively high.

A future direction for this work is to extend the current STE based scheduling algorithms further by considering heterogeneous multicore systems. In this case, as the number of STATES in the STE process of an SDG will increase, the runtime of STE algorithms will also increase. Therefore, another problem to be solved is to develop efficient methods that restrict/reduce the number of STATES and guarantee some high efficiency for communication-aware STE based scheduling algorithms.

REFERENCES


