Reducing Code Size in Scheduling 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
A Synchronous Dataflow Graph (SDFG) is a widely used abstraction to capture the characteristics of a number of applications often running on embedded systems. When scheduling/mapping an SDFG on a multicore embedded system, the code of tasks may have to be duplicated onto multiple cores to fully utilize the parallelism of the multicore system. However, such an approach may increase the overall code size in the system, which may not be desirable. This paper proposes a code-size-aware scheduling heuristic, which decreases code duplication of SDFGs on multicore systems, hence minimizing overall code size while not affecting throughput. In experiments, the proposed heuristic achieves significant code size reduction for all the tested SDFGs compared with a state-of-art recently proposed scheduling algorithm.

CCS CONCEPTS
• Computer systems organization → Real-time operating systems; Embedded software; • Hardware → Operations scheduling;

1 INTRODUCTION
Traditionally, code size reduction has been an important issue with embedded system design. Although the capacity of storage devices is continuously increasing and the unit costs are decreasing accordingly, the demand for storage space increases even faster [1]. In embedded systems, the cost of instruction memory, directly related to code size, is often comparable to the cost of the processor. In general, accessing on-chip memory is much faster than off-chip memory but at the same time also more expensive. Thus, as cost restrictions typically limit the size of on-chip memory, there is strong interest in the development of code size reduction technologies [1]. Besides cost design considerations, embedded systems are often required to satisfy hard real-time constraints. This means that runtime predictability also becomes an issue. Overall, smaller code size can lead to fewer accesses of off-chip memory or even help avoid off-chip memory access altogether, which can improve the performance and power consumption of a system accordingly [2].

Many common applications in embedded systems are digital signal processing and multimedia applications. These applications are typically used for processing continuous data streams like frames of a video, and are known as streaming applications [11]. Synchronous Dataflow Graphs (SDFGs) [3] are widely used to model streaming applications. An SDFG consists of actors and channels, as shown in the example SDFG in Fig. 1. The nodes in the graph are referred to as actors, which are used to model the computational modules of an application. The directed edges are referred to as channels, which represent data communication between actors. The execution of actors is called as firing. The communication data is represented by tokens which are transferred through first-in-first-out (FIFO) buffers on channels. The firing of an actor consumes tokens on its input channels and produces tokens on its output channels. The numbers annotating the two ends of a channel are token producing and consuming rates of the corresponding actors. The token producing and consuming rates of actors can be different, making the firing rates of the actors also different. Streaming applications are executed iteratively. Data communication between actors in different iterations can also be expressed through an SDFG by setting initial tokens on channels. The average computation time per iteration is referred to as iteration period (IP). The speed of a streaming system is represented by throughput, which is the reciprocal of IP.

In recent years, there has been lots of research related to scheduling and executing SDFGs on multicore [8, 10]. In brief, scheduling an SDFG on a multicore system consists of mapping actors onto different cores and deciding the execution of sequences of the actors on different cores. Mapping can be actor-to-core binding based, which means an actor can only be mapped on one core, or actor duplication-enabled, which means an actor can be mapped onto one or more cores [10]. With binding based mapping, an actor is restricted to fire on a fixed processing core. In contrast, duplication-enabled mapping enables an actor to be fired on more than one processing core, making the parallelism within an application to be better utilized. Especially with SDFGs, an actor in an SDFG always fires multiple times within an iteration. For example, actors A, B, and C in Fig. 1 have to fire at least 3, 2 and 2 times within an iteration, respectively. Therefore, duplication-enabled mapping is preferable in terms of improving throughput. However, the code
size of the application is very likely to increase by actor duplication as an actor has to be mapped onto multiple cores. This raises an important problem, which is the motivation for our paper: code size reduction approaches for duplication-enabled mapping.

This paper is an extended version of an idea that first appeared as a 2-page ‘work-in-progress’ note in [5]. The main contribution of this paper is a duplication-enabled mapping heuristic for Code-Size-Aware Scheduling (denoted by csas), which aims at reducing the extra code size introduced by the duplicated actors while maintaining the throughput obtained by the original scheduling algorithm. Experimental results suggest some significant code size reduction obtained by csas.

The remainder of the paper is organized as follows. Firstly, some basic concepts and a problem definition are given in Section 2. In Section 3, we use a motivational example to illustrate the basic idea of the paper. After that, the proposed heuristic is elaborated in Section 4, following by an experimental evaluation in Section 5. Finally, Section 6 concludes this paper.

2 BACKGROUND

2.1 Synchronous Dataflow Graphs

Firstly, a graph theory based definition for an SDFG is given.

*Definition 2.1.* An SDFG $G$ is a tuple $(A, E)$, where $A$ is a finite set of actors and $E$ is a set of directed channels which connect the actors in set $A$. An actor $\alpha$ in $A$ is a tuple $(input, output, ft, cs)$, where $input$ is a set of input channels of the actor $\alpha$, $output$ is a set of output channels of $\alpha$, $ft$ is the firing time of $\alpha$, and $cs$ is the code size of $\alpha$. A channel $\epsilon \in E$ is a tuple $(src, snk, p, q, itn, tn)$, where $src$ is the source actor of the channel, $snk$ is the sink actor of the channel, $p$ is the token producing rate of the actor $src$, $q$ is the token consuming rate of the actor $snk$, $itn$ is the number of initial tokens on the channel, and $tn$ is the number of tokens on the channel.

Actors in an SDFG may have to be fired multiple times within one iteration of the graph. The smallest firing times of actors to perform one iteration are specified by a *repetition vector*, which is denoted by $R$, where $\forall q \in R, q \in N^+$. The repetition vector can be obtained by solving a series of balance equations [3]:

$$R(e(src))e(p) = R(e(snk))e(q), \forall e \in E.$$  
(1)

The calculation method of the repetition vector is elaborated in [7]. If an SDFG does not have a legal repetition vector $R$, it is not *sample rate consistent* [3], which means no legal schedule exists for this SDFG. The speed of SDFGs is revealed by its *iteration period (IP)* [3], which means no legal schedule exists for this unfolded graph. Unfolding unfolds the original graph, making two or more iterations to be expressed in one iteration of the unfolded graph. These techniques are adopted in the original scheduling algorithm used this paper to achieve better input graphs for the proposed heuristic. In particular, the unfolding of an SDFG also brings about the actor duplication, which can lead to the increase of code size. This kind of code increase can also be reduced by the proposed code-size-aware scheduling heuristic.

2.2 Self-Timed Scheduling

If an SDFG is verified to be sample rate consistent and deadlock-free, there must be a periodic schedule for the SDFG. A duplication-enabled scheduling algorithm $Sch_p(G, P)$ proposed in [12] is adopted in this paper as the algorithm to be enhanced; in the rest of the paper, we denote this algorithm as des. A scheduling strategy called self-timed scheduling (STS) is used in des. During the STS process, actors in an SDFG fire as soon as possible until a periodic execution pattern is found. The periodic phase of the execution forms a periodic schedule of the SDFG. The execution phase before the periodic phase forms a retiming for the SDFG. The retimed SDFG $G_r$ is used as input of the Code-Size-Aware-Scheduling algorithm csas, to be discussed in Section 4.

2.3 Problem Definition

The problem addressed in this paper is to propose a code-size-aware duplication-enabled scheduling strategy for an SDFG on a multicore system. The objective of the scheduling strategy is to reduce the extra code size caused by the duplication of actors without affecting the throughput of the original schedule. The target multicore platform is assumed to have a set of homogeneous processing cores, which are connected by an ideal interconnection network: this means that communication overheads between cores are assumed to be zero. For the computation of the code size of a system, we make the following assumption. If an actor is mapped onto a core, the code of the actor is stored in the private memory of the core and cannot be shared with other cores. This means that if an actor is mapped on multiple cores, then multiple copies of the code of the actor should be stored separately on the corresponding cores. All code is stored in on-chip memory, and therefore no off-chip accessing is considered. Based on this assumption, code size in this paper is calculated by the following equation:

$$CodeSize = \sum_{c \in C} \sum_{\alpha \in M(c)} \alpha(cs),$$  
(2)

where $C$ is a set which consists of all cores in the given multicore platform, and $M(c)$ is a set of actors which are mapped on core $c$.

3 MOTIVATIONAL EXAMPLE

The SDFG in Fig. 1 is adopted as a motivational example. The numbers on the two ends of a channel are data producing and consuming rates of the connected actors. The firing time and code size of actors are all assumed to be 1. The SDFG is assumed to be mapped on a two-core system. For simplicity, retiming
As shown in the table, for the duplication-enabled mapping, its size is much larger than the other two methods since code size reduction is not considered during its mapping process. For the code size-aware scheduling (csas), makes use of the STS strategy to determine the firing of actors. This guarantees that the throughput obtained by csas is the same as des. The scheduling process is shown in Algorithm 1. The input of the algorithm is the retimed SDFG $G_C = (A, E)$ obtained by des and the repetition vector $R$ of $G_C$. All elements in the repetition vector $R$ should be multiplied by the unfolding factor $uf$ obtained by des.

Algorithm 1 Code-size-aware scheduling (csas)

Input:
- The retimed SDFG $G_C = (A, E)$ and its repetition vector $R$
- Output:
The periodic schedule for $G_C$ with code-size-aware mapping

Transformation:
1. Set variable $\epsilon$ as the maximum number of cores $\epsilon$.
2. Set variable $\epsilon$ as the maximum number of cores $\epsilon$.
3. Set variable $\epsilon$ as the maximum number of cores $\epsilon$.
4. Set variable $\epsilon$ as the maximum number of cores $\epsilon$.
5. Set variable $\epsilon$ as the maximum number of cores $\epsilon$.
6. Set variable $\epsilon$ as the maximum number of cores $\epsilon$.
7. Set variable $\epsilon$ as the maximum number of cores $\epsilon$.
8. Set variable $\epsilon$ as the maximum number of cores $\epsilon$.
9. Set variable $\epsilon$ as the maximum number of cores $\epsilon$.

Algorithm 1 is initialized in line 1. The following while-loop (line 2 to line 9) in Algorithm 1 is the scheduling process. According to the concept of the repetition vector, the loop stops when a periodic schedule is produced for the given SDFG. Therefore, the algorithm decrements $sr$ by 1 after an actor has been fired and stops when $sr$ equals to 0. The obtained schedule can be applied periodically for the infinite iterative execution of the graph. During the scheduling process, firstly, the firings of actors on cores with the smallest ending time $ET(c)$ in $OC$ are finished. Multiple actors can finish at the same time. When the firing of an actor $\alpha$ is finished, the following firing finishing operations should be performed:

- Produce tokens to the output channels of the actor $\alpha$ by setting $\forall x \in \alpha(output), \epsilon(x) = \epsilon(x) + \epsilon(p)$.
- Set variable $CLK$ to $CLK \leftarrow ET(c)$, where $c$ is the core with the smallest ending time.
- Release core $c$ which is currently occupied by actor $\alpha$ by inserting $c$ into $AC$ and removing $c$ from $OC$.

Then, the algorithm looks for the actors which are ready to fire and inserts them into $RA$. An actor $\alpha$ is ready to fire as long as the ready condition in Equation 3 is satisfied.

$$ [\forall x \in \alpha(input), \epsilon(x) \geq \epsilon(q)] \land [RF(\alpha) > 0] = \text{true} \quad (3) $$

The condition $[\forall x \in \alpha(input), \epsilon(x) \geq \epsilon(q)]$ ensures that there are enough tokens on the input channels to be consumed to fire once for an actor $\alpha$. The other condition $[RF(\alpha) > 0]$ makes sure

<table>
<thead>
<tr>
<th>Table 1: Code size and IP got by three mapping methods.</th>
</tr>
</thead>
<tbody>
<tr>
<td>core0</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Duplication-enabled</td>
</tr>
<tr>
<td>Code-size-aware</td>
</tr>
<tr>
<td>Binding based</td>
</tr>
</tbody>
</table>

Figure 2: Three schedules of the example SDFG based on three mapping strategies respectively.
Algorithm 2: Select the actor with the highest priority \( \alpha_p \)

**Input:**
The set of actors which are ready to fire \( RA \)

**Output:**
The actor with the highest priority \( \alpha_p \)

**Iteration:**
1. Set \( \alpha_p \) to be the first actor in \( RA \).
2. for all \( \alpha \in RA \) do
   3. if \( \alpha_p(n\epsilon) > \alpha(n\epsilon) \) then
      4. \( \alpha_p(n\epsilon) = \alpha(n\epsilon); \)
      5. else if \( \alpha_p(n\epsilon) = \alpha(n\epsilon) \) then
         6. if \( \alpha_p(f\epsilon) > \alpha(f\epsilon) \) then
            7. \( \alpha_p(f\epsilon) = \alpha(f\epsilon); \)
         8. else if \( \alpha_p(f\epsilon) = \alpha(f\epsilon) \) then
            9. if \( \alpha_p(f\epsilon) > \alpha(f\epsilon) \) then
               10. \( \alpha_p(f\epsilon) = \alpha(f\epsilon); \)
      11. end if
   12. end if
   13. end if
3. end for
4. return \( \alpha_p \);

that the number of firings of actor \( \alpha \) within one iteration is exactly the value of \( R(\alpha) \). If an actor satisfies the ready condition, we call this actor as a *ready actor*.

The inner while-loop (line 5 to line 8) of Algorithm 1 maps ready actors onto processing cores and starts the firing of the actors. The loop keeps running until no ready actors exist in \( RA \) or no unoccupied cores are available for the ready actors in \( AC \). Line 6 of the algorithm decides the allocation of an actor to a core implementing the code-size-aware strategy proposed in this paper. The strategy consists of two steps. Firstly, if two or more actors fire at the same time, the strategy chooses an actor with the highest priority to do the mapping; this is further explained by Algorithm 2 presented in Section 4.2. Next, the strategy decides which core should the chosen actor be allocated to using Algorithm 3, further discussed in Section 4.3. After the chosen actor has been mapped, the firing of the actor starts as shown in line 7 of the algorithm. For an actor \( \alpha \), the firing start operation consists of the following behavior:

1. Consume tokens from the input channels of the actor \( \alpha \) by setting \( V \epsilon \in \alpha(input) \).
2. Update \( ET(\alpha) \leftarrow CLK + \alpha(f\epsilon) \).
3. Decrement \( RF(\alpha) \) by one.
4. Remove \( \alpha \) from \( RA \) if the ready condition in Equation 3 for \( \alpha \) is no longer met after the token consuming operation.

Finally, the value of \( sr \) is decremented by one to ensure that the scheduling process gives a schedule for exactly one iteration of the given SDFG.

### 4.2 Select an Actor to be mapped

An algorithm to select the actor with the highest mapping priority, \( \alpha_p \), is shown in Algorithm 2. If an actor \( \alpha \) is in the set \( AL(c) \), then core \( c \) is a *prior core* for the actor \( \alpha \), which means that at least one prior instantiation of actor \( \alpha \) was allocated to core \( c \). During the mapping process, two extra elements \( nc \) and \( rc \) are introduced to the tuple of an actor \( \alpha \). For an actor \( \alpha \), \( \alpha(nc) \) is the number of its prior cores in \( AC \), and element \( \alpha(rc) \) is the number of repetitions of \( \alpha \) in one iteration of the graph, which equals to \( R(\alpha) \). In particular, if no prior core exists for an actor \( \alpha \), then there is no difference to map the actor to any of the available cores. Therefore, \( \alpha(nc) \) should be set as the number of available cores \( |AC| \).

The four elements \( nc, rc, cs \) and \( ft \) within each actor are used as criteria to judge the priority of actors. The input of the algorithm is \( RA \), and the output is the actor with the highest priority \( \alpha_p \). The actor \( \alpha_p \) is initialized as the first actor in \( RA \) in line 1 of Algorithm 2. The for-loop between lines 2 and 14 is used to find out the actor with the highest priority. As shown in line 3 and line 4 in the algorithm, an actor with a smaller \( \alpha(nc) \) value has a higher priority. This is because an actor with a small \( nc \) number has only limited options to do the mapping if we want to map the actor to one of its prior cores. If its prior cores are occupied by other actors at the current point in time, the actor will have to be allocated onto a non-prior core, which will lead to an increase of the code size. Therefore, it is reasonable to map the actor with the smallest \( nc \) first.

If the \( nc \) of two actors is equal, then the quotients of the repetition number \( rc \) and code size \( cs \) of actors are used as a second criterion, as shown in line 6 and line 7 in the algorithm. An actor with a larger repetition number has a lower priority because this actor is more likely to be distributed on multiple cores. This means the duplication of the code of this actor on various cores is less avoidable. On the other hand, since the duplication of an actor with a larger code size leads to a greater increase of code size than the duplication of an actor with a smaller code size, the actor with a larger code size should have a higher priority during the mapping process. Therefore, we jointly use the two factors \( rc \) and \( cs \) to decide the priority within two actors.

Then, if the quotients of \( rc \) and \( cs \) of two actors are also the same, the firing time \( ft \) is used as the final tie-breaker, as shown in line 9 to line 11 in the algorithm. An actor with a larger firing time has a lower priority. Since an actor with a larger \( ft \) value can occupy a core for a longer period, it is better to fire an actor with a smaller \( ft \) value first in order to regain the availability of the core sooner.

### 4.3 Map an Actor onto a Core

After \( \alpha_p \) has been obtained, Algorithm 3 is used to map \( \alpha_p \) to the core with the highest priority \( cp \). The input of the algorithm consists of \( \alpha_p, RA, AC, OC \) and \( AL \). \( cp \) is initialized to the first core in the set of unoccupied cores \( AC \). The for-loop is used to get the core with the highest priority \( cp \) for \( \alpha_p \). Mapping actor \( \alpha_p \) onto one of its prior cores can reduce the overall code size by preventing the code duplication of the actor on a non-prior core. Therefore, the algorithm gives priority to prior cores of actor \( \alpha_p \), as shown in line 7. If \( \alpha_p \) has more than one prior core or it does not have any prior core, which means the condition in line 3 of the algorithm is met, then the value of \( |AL(\alpha)| \) is used to break the tie. As shown in line 4 to line 6 of the algorithm, a core with a larger \( |AL(\alpha)| \) has lower priority. A core with a larger \( |AL(\alpha)| \) value means the core is the prior core for more actors. Therefore, the algorithm gives lower priority for this core to reserve the core for more actors.

After \( cp \) has been obtained, the actor \( \alpha_p \) should be inserted into \( AL(cp) \) as shown in line 11 to line 13 of the algorithm. Then, the core \( cp \) should be removed from \( AC \) and inserted to \( OC \) as shown in line 14 to line 17 in the algorithm. The firing time \( \alpha_p(ft) \) is 0 only if the actor \( \alpha_p \) is a dummy actor generated by some SDFG.
transformation techniques, which means the firing of the actor \(a_p\) does not actually occupy a processing core. In this case only, the core \(c_p\) does not need to be removed from \(AC\) and moved to \(OC\).

5 EVALUATION

5.1 Experimental Setup

Two datasets are used in our experiment. The first dataset consists of 16 realistic applications which are all obtained from the benchmark folder of Streamit 2.1.1 (Older release) [11]. Specifically, the 16 applications include BeamFormer and SerializedBeamFormer under the subfolder beamformer; FFT2, FFT3 and FF4 under the subfolder fft; FilterBankNew under the subfolder filterbank; FMRadio under the subfolder fm; MatrixMultBlock under the subfolder matmul-block; and BeamFormer, BitonicSort, ChannelVocoder, FFT5, FMRadio5, MPEGdecoder, tde_pp and VocoderTopLevel under the subfolder asplos06.

A second dataset is also used to enhance the evaluation and understanding of csas, which consists of 5 groups of random SDFGs generated by a tool named SDF3 [9]. The 5 SDFG groups are denoted as Random10, Random20, Random30, Random40 and Random50. Each of these groups includes 100 different SDFGs with 10, 20, 30, 40 or 50 actors, respectively. In the five groups, the sum of the elements in the repetition vector of an SDFG is set to be 30, 60, 90, 120, 150, respectively. Other generation parameters for the five groups are the same. The number of the channels connected to an actor is generated randomly with an average of 3, variation of 1, minimum of 1 and maximum of 5. The data producing and consuming rates of actors are generated randomly with an average of 3, variation of 1, minimum of 1 and maximum of 5. The data producing and consuming rates of actors are generated randomly with an average of 3, variation of 1, minimum of 1 and maximum of 5. The data producing and consuming rates of actors are generated randomly with an average of 3, variation of 1, minimum of 1 and maximum of 5. The data producing and consuming rates of actors are generated randomly with an average of 3, variation of 1, minimum of 1 and maximum of 5. The data producing and consuming rates of actors are generated randomly with an average of 3, variation of 1, minimum of 1 and maximum of 5. The data producing and consuming rates of actors are generated randomly with an average of 3, variation of 1, minimum of 1 and maximum of 5. The data producing and consuming rates of actors are generated randomly with an average of 3, variation of 1, minimum of 1 and maximum of 5. The data producing and consuming rates of actors are generated randomly with an average of 3, variation of 1, minimum of 1 and maximum of 5. The data producing and consuming rates of actors are generated randomly with an average of 3, variation of 1, minimum of 1 and maximum of 5.

In the experiments we compare csas with the duplication-enabled scheduling algorithm, des, from [12]. Since the actor-to-core allocation of des is not explicitly given in [12], we assume that an actor is allocated on the first available core in the core list. The auto-concurrency degree of all the SDFGs used in the experiment is restricted to 1, which means multiple simultaneous firings of an actor are not allowed. The restriction can be applied to an SDFG by adding a self-edge with one initial token to every actor in the SDFG.

5.2 Realistic Applications

Compared with des, the code size reduction of csas for different SDFGs on platforms with different number of cores is shown in Fig. 3, where \(M\) is the number of cores. The reduction is calculated by \((\text{CodeSize}_d - \text{CodeSize}_c)/\text{CodeSize}_d\), where \(\text{CodeSize}_d\) is the code size obtained by des and \(\text{CodeSize}_c\) is the code size obtained by csas. The throughput obtained by csas for all the realistic SDFGs is the same as des. From Fig. 3, we can see that csas achieves up to 75% code reduction compared with des. The average code size reduction of all the tested SDFGs is also given in Fig. 3, labeled average (right-end of the figure). The average code size reduction on a two-core system is about 4%, while the average code size reduction on the platforms with 16 cores is about 30%. The average code size reduction of SDFGs increases with the number of cores. A similar trend can also be observed with most of the SDFGs in Fig. 3. This means csas performs better when the computing resources are abundant.

The reason for this can be explained easily. When the number of cores in a multicore system is very small, the parallelism in an SDFG cannot be sufficiently utilized by the system. In this case, the duplication of actors is not necessary and therefore the overall code size can be small anyway. As the number of the cores in the system increases, the duplication of actors starts to improve the throughput of the system, but the probability that an actor is not mapped onto a prior core also increases thereby increasing overall code size. At the same time, this gives more scope and potential for csas, which takes into account core locality for actors, to achieve a high code size reduction.

5.3 Randomly Generated SDFGs

For the second dataset, consisting of five groups of randomly generated SDFGs, the average code size obtained by csas and des for each of the five SDFG groups is shown in Table 2. The top row of the table shows the number of cores considered in each case. The leftmost column of the table gives the names of the five SDFG groups. The throughput obtained by csas and des is not given in the table since they both get the same throughput for all the SDFGs tested in the experiment.

The code sizes in Table 2 are all normalized by the minimum code sizes of the corresponding SDFGs. The minimum code size of an SDFG is the sum of the code size of each actor. Thus, if the obtained code size for an SDFG is 1, the obtained code size equals to the minimum code size of the SDFG. The code sizes obtained by csas on a 64-core system for all the five groups are all 1, which suggests that csas achieves minimum code size for each SDFG in all the five groups when the number of cores is larger than the number of actors.

---

**Algorithm 3 Map \(a_p\) onto the core with the highest priority**

**Input:**
- The actor with the highest priority \(a_p\), the set of actors which are ready to fire (RA), the set of available cores (AC), the set of occupied cores (OC), and the vector of current actor allocation on these cores (AL)

**Output:**
- Updated AL, AC and OC

**Iteration:**
1. Set \(c_p\) to be the first core in AC;
2. for all \(c \in AC\) do
3. if \((a_p, c) \in AL(c) \&\& a_p \notin AL(c_p)) \&\& (\exists a_p \in AL(c), a_p \notin AL(c_p))\) then
4. if \(|AL(c)| > |AL(c)|\) then
5. \(c_p = c;\)
6. end if
7. else if \(a_p \in AL(c) \&\& a_p \notin AL(c_p)\) then
8. \(c_p = c;\)
9. end if
10. end for
11. if \(a_p \notin AL(c_p)\) then
12. Insert \(a_p\) into \(AL(c_p)\);
13. end if
14. if \(a_p(f) > 0\) then
15. Remove \(c_p\) from AC;
16. Insert \(c_p\) to OC;
17. end if
Figure 3: Code size reduction of CSAS for realistic SDFGs on platforms with 2, 4, 8 and 16 cores compared with DES.

Table 2: The code size obtained by CSAS and DES for random SDFGs.

<table>
<thead>
<tr>
<th>#name</th>
<th>#cores</th>
<th>CSAS</th>
<th>DES</th>
<th>Red.</th>
<th>CSAS</th>
<th>DES</th>
<th>Red.</th>
<th>CSAS</th>
<th>DES</th>
<th>Red.</th>
<th>CSAS</th>
<th>DES</th>
<th>Red.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random10</td>
<td>2</td>
<td>1.12</td>
<td>1.21</td>
<td>7%</td>
<td>1.05</td>
<td>1.32</td>
<td>18%</td>
<td>1.03</td>
<td>1.31</td>
<td>26%</td>
<td>1.00</td>
<td>1.31</td>
<td>22%</td>
</tr>
<tr>
<td>Random20</td>
<td>4</td>
<td>1.09</td>
<td>1.12</td>
<td>2%</td>
<td>1.06</td>
<td>1.19</td>
<td>10%</td>
<td>1.02</td>
<td>1.20</td>
<td>14%</td>
<td>1.01</td>
<td>1.20</td>
<td>15%</td>
</tr>
<tr>
<td>Random30</td>
<td>8</td>
<td>1.07</td>
<td>1.09</td>
<td>2%</td>
<td>1.09</td>
<td>1.21</td>
<td>9%</td>
<td>1.02</td>
<td>1.18</td>
<td>15%</td>
<td>1.01</td>
<td>1.18</td>
<td>14%</td>
</tr>
<tr>
<td>Random40</td>
<td>16</td>
<td>1.09</td>
<td>1.10</td>
<td>1%</td>
<td>1.07</td>
<td>1.17</td>
<td>8%</td>
<td>1.03</td>
<td>1.18</td>
<td>12%</td>
<td>1.02</td>
<td>1.18</td>
<td>13%</td>
</tr>
<tr>
<td>Random50</td>
<td>32</td>
<td>1.06</td>
<td>1.09</td>
<td>2%</td>
<td>1.06</td>
<td>1.13</td>
<td>5%</td>
<td>1.02</td>
<td>1.14</td>
<td>10%</td>
<td>1.01</td>
<td>1.14</td>
<td>11%</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Red. column in Table 2 is the code size reduction of CSAS relative to DES. For all the five groups, the normalized code sizes obtained by CSAS are less than DES. The normalized code size obtained by CSAS decreases as the number of cores increases, while the normalized code size obtained by DES generally shows an opposite tendency. When the number of actors is much smaller than the number of cores, the code size reduction of CSAS compared with DES is not remarkable (as low as 1% for Random40 on a 2-core system as shown in Table 2). The code size reduction increases to 12%-22% for the five groups when the number of cores is large enough. In conclusion, the advantage of CSAS relative to DES observed in Section 5.2 is observed again in this section when using a large number of random graphs.

6 CONCLUSION AND FUTURE WORK

In this paper, a code-size-aware mapping heuristic is proposed to reduce the code size of SDFGs on multicore systems during scheduling. Firstly, the heuristic obtains the actor with the highest priority when multiple actors fire concurrently. Then, the obtained actor is mapped onto the core with the highest priority for the actor. The proposed mapping heuristic is combined with a self-timed scheduling to form a code-size-aware scheduling (CSAS) strategy. The experimental results show that compared with the original scheduling algorithm, CSAS can always get lower code size for all the SDFGs in the experiment. The code reduction increases with the number of cores. It appears that the minimum code size can always be achieved by CSAS if a multicore system has enough processing cores (more than the number of actors). Future work will attempt to: (i) find exact solutions for the code size reduction problem; (ii) consider more metrics than just code size and throughput; (iii) use more accurate communication models instead of the ideal model; (iv) extend the work onto heterogeneous multicore platforms, etc.

REFERENCES