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].

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.
PARMA-DITAM '18, January 23, 2018, Manchester, United Kingdom
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6444-7/18/01…$15.00
https://doi.org/10.1145/3183767.3183781

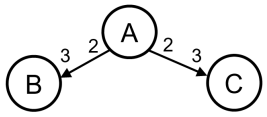

Figure 1: An example SDFG.

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 multicores [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 α in A is a tuple $(input, output, ft, cs)$, where $input$ is a set of input channels of the actor α, $output$ is a set of output channels of α, ft is the firing time of α, and cs is the code size of α. A channel $\varepsilon \in E$ 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 $\boldsymbol{R}$, where $\forall q \in \boldsymbol{R}$, $q \in \mathbb{N}^+$. The repetition vector can be obtained by solving a series of balance equations [3]:

$$\boldsymbol{R}(\varepsilon(src))\varepsilon(p) = \boldsymbol{R}(\varepsilon(snk))\varepsilon(q), \forall \varepsilon \in E. \quad (1)$$

The calculation method of the repetition vector is elaborated in [7]. If an SDFG does not have a legal repetition vector $\boldsymbol{R}$, 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) or throughput. IP is the time consumed by firing all actors $\alpha \in A$ by $\boldsymbol{R}(\alpha)$ times. Throughput is the reciprocal of IP.

Since channels with initial tokens are introduced into SDFGs, cycles are allowed in the graphs. However, a cycle can be deadlocked if the number of the initial tokens is not set properly. An SDFG is *deadlock-free* if and only if the infinite execution of the graph does not stop for the insufficiency of the input tokens for any actor, otherwise it is *deadlocked* [3]. If an SDFG is deadlocked, it also means the graph does not have a legal schedule. Therefore, a deadlock test should also be conducted before scheduling.

Some graph transformation techniques can be applied to an SDFG, like *retiming* [4] and *unfolding* [6], transforming the original graph to its equivalent graph which has the potential to get better scheduling results. Retiming redistributes the initial tokens on channels of the original graph. Unfolding unfolds the original graph, making two or more iterations to be expressed in one iteration of the unfolded graph. These two 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:

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

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

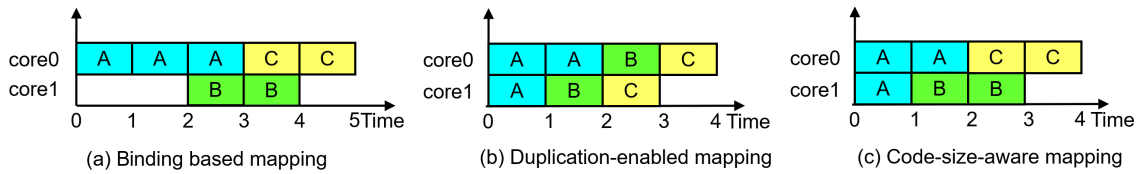

Figure 2: Three schedules of the example SDFG based on three mapping strategies respectively.

Table 1: Code size and IP got by three mapping methods.

	*core*0	*core*1	code size	IP
Duplication-enabled	{A, B, C}	{A, B, C}	6	4
Code-size-aware	{A, C}	{A, B}	4	4
Binding based	{A, C}	{B}	3	5

and unfolding are not considered during the scheduling of the example graph. Three different mapping strategies are applied to this example SDFG to illustrate the motivation of this paper.

The outcomes of the three schedules in Fig. 2 are shown in Table 1. The columns *core0* and *core1* give the actor distribution on the two processing cores. The column *code size* gives the total code size of actors on all cores. The IPs are shown in the column *IP*. As shown in the table, for the duplication-enabled mapping, its IP is the minimum among the three methods. However, the code size is much larger than the other two methods since code size reduction is not considered during its mapping process. For the binding based method, the code size is the minimum in these three methods since this method binds an actor onto one and only one processing core. However, throughput is accordingly restricted by this feature. One of the fastest schedules for the binding based mapping is shown in Fig. 2(a). The IP of the schedule is 5, which is longer than the schedules for the other two duplication-enabled mapping methods shown in Fig. 2(b) and 2(c). The reason is that the parallelism in the example graph is not fully utilized, as actor A is bound to core0 by the binding based mapping and the parallel execution of two entities of actor A on the two cores in Fig. 2(a) and (b) is not allowed. The code-size-aware mapping method tries to decrease the code size of duplication-enabled scheduling under the constraint of maintaining the IP of the original schedule. As shown in Table 1, compared with the duplication-enabled method, the code-size-aware mapping gets same throughput and 33% less code size.

In this example, we can see that compared with binding based mapping, actor duplication can decrease the IP of the execution of an SDFG, i.e. increase the throughput of the system. The code size increase brought by the actor duplication can be effectively reduced if code-size-aware mapping is applied.

4 CODE-SIZE AWARE SCHEDULING

4.1 Scheduling Process

Code-Size-Aware-Scheduling (CSAS) makes use of the STS strategy (see Section 2.2) 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 1 Code-size-aware scheduling (CSAS)

Input:
The retimed SDFG $G_r = (A, E)$ and its repetition vector $\boldsymbol{R}$
Output:
A periodic schedule for G_r with code-size-aware mapping
Iteration:
1: Get the total actor firings sr by summing the elements in $\boldsymbol{R}$; $\forall \varepsilon \in E$, set $\varepsilon(tn) = \varepsilon(itn)$; set CLK (current time) to 0; set a vector of remaining firings of actors $\boldsymbol{RF} = \boldsymbol{R}$; set a vector of ending time of cores $\boldsymbol{ET} = \boldsymbol{0}$; set a set of actors which are ready-to-fire $RA =$; set a set of unoccupied cores AC to a set of all cores; set a set of occupied cores $OC =$; and set each element $\boldsymbol{AL}(c)$ in a vector of current actor allocation on cores $\boldsymbol{AL}$ to be $\boldsymbol{AL}(c) =$, where the size of the vector $|\boldsymbol{AL}|$ equals to the number of cores;
2: **while** $sr > 0$ **do**
3: For core(s) c in OC with the smallest ending time $\boldsymbol{ET}(c)$, finish the firing(s) of the actor(s) on the core(s), including updating $\varepsilon(tn)$, AC, OC and CLK;
4: Find actors which are ready to fire and insert these actors into RA;
5: **while** $(RA \neq)\&\&(AC \neq)$ **do**
6: Choose an actor from RA using Algorithm 2. Map the actor on a core c in AC (and update $\boldsymbol{AL}$, AC and OC) using Algorithm 3;
7: Start the firing of the mapped actor, including updating $\varepsilon(tn)$, RA, $\boldsymbol{RF}$ and $ET(c)$; decrement sr by 1;
8: **end while**
9: **end while**

algorithm is the retimed SDFG $G_r = (A, E)$ obtained by DES and the repetition vector $\boldsymbol{R}$ of G_r. All elements in the repetition vector $\boldsymbol{R}$ should be multiplied by the unfolding factor uf obtained by DES.

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 $\boldsymbol{ET}$ in OC are finished. Multiple actors can finish at the same time. When the firing of an actor α is finished, the following firing finishing operations should be performed:

(1) Produce tokens to the output channels of the actor α by setting $\forall \varepsilon \in \alpha(output), \varepsilon(tn) = \varepsilon(tn) + \varepsilon(p)$.
(2) Set variable CLK to $CLK \leftarrow ET(c)$, where c is the core with the smallest ending time.
(3) Release core c which is currently occupied by actor α 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 α is ready to fire as long as the *ready condition* in Equation 3 is satisfied.

$$[\forall \varepsilon \in \alpha(input), \varepsilon(tn) \geq \varepsilon(q)] \wedge [\boldsymbol{RF}(\alpha) > 0] = true \qquad (3)$$

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

Algorithm 2 Select the actor with the highest priority α_p

Input:
The set of actors which are ready to fire (RA)
Output:
The actor with the highest priority α_p
Iteration:
1: Set α_p to be the first actor in RA;
2: **for all** $\alpha \in RA$ **do**
3: **if** $\alpha_p(nc) > \alpha(nc)$ **then**
4: $\alpha_p(nc) = \alpha(nc)$;
5: **else if** $\alpha_p(nc) \equiv \alpha(nc)$ **then**
6: **if** $\alpha_p(rc)/\alpha_p(cs) > \alpha(rc)/\alpha(cs)$ **then**
7: $\alpha_p(rc) = \alpha(rc)$;
8: **else if** $\alpha_p(rc)/\alpha_p(cs) \equiv \alpha(rc)/\alpha(cs)$ **then**
9: **if** $\alpha_p(ft) > \alpha(ft)$ **then**
10: $\alpha_p(ft) = \alpha(ft)$;
11: **end if**
12: **end if**
13: **end if**
14: **end for**
15: **return** α_p;

that the number of firings of actor α within one iteration is exactly the value of $\boldsymbol{R}(\alpha)$. 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 α, the firing start operation consists of the following behavior:

(1) Consume tokens from the input channels of the actor α by setting $\forall \varepsilon \in \alpha(input), \varepsilon(tn) = \varepsilon(tn) - \varepsilon(q)$.
(2) Update $\boldsymbol{ET}(c) \leftarrow CLK + \alpha(ft)$, where c is the core to which actor α has been mapped on.
(3) Decrement $\boldsymbol{RF}(\alpha)$ by one.
(4) Remove α from RA if the ready condition in Equation 3 for α 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, α_p, is shown in Algorithm 2. If an actor α is in the set $\boldsymbol{AL}(c)$, then core c is a *prior core* for the actor α, which means that at least one prior instantiation of actor α was allocated to core c. During the mapping process, two extra elements nc and rc are introduced to the tuple of an actor α. For an actor α, $\alpha(nc)$ is the number of its prior cores in AC, and element $\alpha(rc)$ is the number of repetitions of α in one iteration of the graph, which equals to $\boldsymbol{R}(\alpha)$. In particular, if no prior core exists for an actor α, 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 α_p. The actor α_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 α_p has been obtained, Algorithm 3 is used to map α_p to the core with the highest priority c_p. The input of the algorithm consists of α_p, RA, AC, OC and $\boldsymbol{AL}$. c_p 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 c_p for α_p. Mapping actor α_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 α_p as shown in line 7. If α_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 $|\boldsymbol{AL}(c)|$ is used to break the tie. As shown in line 4 to line 6 of the algorithm, a core with a larger $|\boldsymbol{AL}(c)|$ has lower priority. A core with a larger $|\boldsymbol{AL}(c)|$ 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 c_p has been obtained, the actor α_p should be inserted into $\boldsymbol{AL}(c_p)$ as shown in line 11 to line 13 of the algorithm. Then, the core c_p 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 α_p is a dummy actor generated by some SDFG

transformation techniques, which means the firing of the actor α_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 9, minimum of 1 and maximum of 30. The execution time of actors is generated randomly with an average of 50, variation of 250, minimum of 10 and maximum of 100. The probability that initial tokens exist on a channel is set to 0. The code size of each actor in these randomly generated SDFGs is taken from a uniformly distributed random distribution with values between 10 and 100.

Algorithm 3 Map α_p onto the core with the highest priority

Input:
The actor with the highest priority α_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 ($\boldsymbol{AL}$)
Output:
Updated $\boldsymbol{AL}$, AC and OC
Iteration:
1: Set c_p to be the first core in AC;
2: **for all** $c \in AC$ **do**
3: **if** $(\alpha_p \in \boldsymbol{AL}(c)$ && $\alpha_p \in \boldsymbol{AL}(c_p))$ || $(\alpha_p \notin \boldsymbol{AL}(c)$ && $\alpha_p \notin \boldsymbol{AL}(c_p))$ **then**
4: **if** $|\boldsymbol{AL}(c_p)| > |\boldsymbol{AL}(c)|$ **then**
5: $c_p = c$;
6: **end if**
7: **else if** $\alpha_p \in \boldsymbol{AL}(c)$ && $\alpha_p \notin \boldsymbol{AL}(c_p)$ **then**
8: $c_p = c$;
9: **end if**
10: **end for**
11: **if** $\alpha_p \notin \boldsymbol{AL}(c_p)$ **then**
12: Insert α_p into $\boldsymbol{AL}(c_p)$;
13: **end if**
14: **if** $\alpha_p(ft) > 0$ **then**
15: Remove c_p from AC;
16: Insert c_p to OC;
17: **end if**

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 $(CodeSize_d - CodeSize_c)/CodeSize_d$, where $CodeSize_d$ is the code size obtained by DES and $CodeSize_c$ 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.

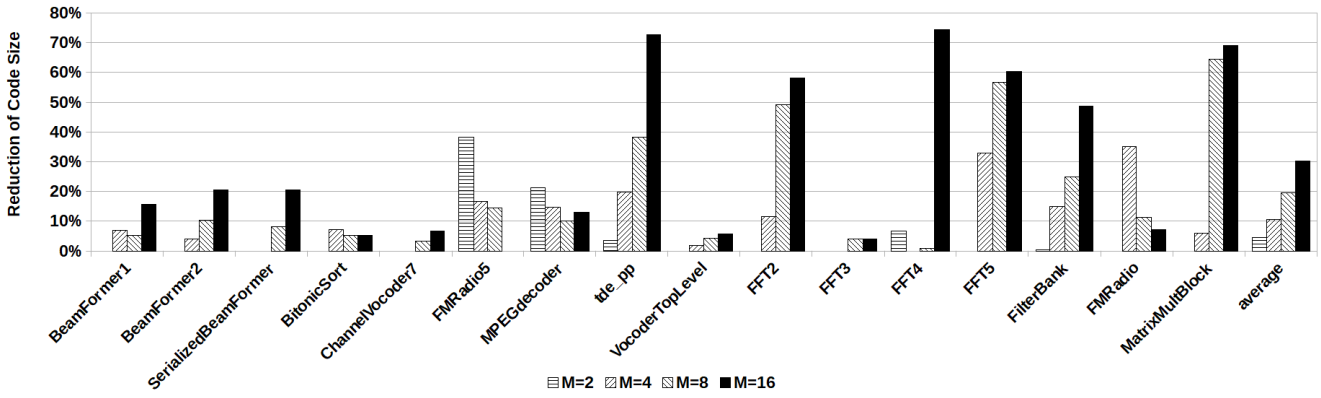

Figure 3: Code size reduction of CSAS for realistic SDFGs on platforms with 2, 4, 8 and 16 cores compared with DES.

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

#name \ #cores	2			4			8			16			32			64		
	CSAS	DES	Red.	CSAS	DES	Red.	CSAS	DES	Red.	CSAS	DES	Red.	CSAS	DES	Red.	CSAS	DES	Red.
Random10	1.12	1.21	7%	1.08	1.32	18%	1.03	1.31	20%	1.00	1.31	22%	1.00	1.31	22%	1.00	1.31	22%
Random20	1.09	1.12	2%	1.06	1.19	10%	1.02	1.20	14%	1.01	1.20	15%	1.00	1.20	16%	1.00	1.20	16%
Random30	1.07	1.09	2%	1.09	1.21	9%	1.02	1.18	13%	1.01	1.18	14%	1.00	1.18	15%	1.00	1.18	15%
Random40	1.09	1.10	1%	1.07	1.17	8%	1.03	1.18	12%	1.02	1.18	13%	1.01	1.18	13%	1.00	1.18	14%
Random50	1.06	1.09	2%	1.06	1.13	5%	1.02	1.14	10%	1.01	1.14	11%	1.00	1.14	12%	1.00	1.14	12%

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

[1] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. 2003. Survey of code-size reduction methods. *Comput. Surveys* 35, 3 (2003), 223–267.

[2] Koen Danckaert, Kostas Masselos, F Cathoor, Hugo J De Man, and Costas Goutis. 1999. Strategy for power-efficient design of parallel systems. *IEEE Trans. VLSI Syst* 7, 2 (1999), 258–265.

[3] Edward Ashford Lee and David G Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.* 100, 1 (1987), 24–35.

[4] Charles E Leiserson and James B Saxe. 1991. Retiming synchronous circuitry. *Algorithmica* 6, 1 (1991), 5–35.

[5] Mingze Ma and Rizos Sakellariou. Code-size-aware mapping for synchronous dataflow graphs on multicore systems. In *CASES*. (work in progress).

[6] Keshab K Parhi and David G Messerschmitt. 1991. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Trans. Comput.* 40, 2 (1991), 178–195.

[7] Sundararajan Sriram and Shuvra S Bhattacharyya. 2009. *Embedded multiprocessors: scheduling and synchronization.* CRC press.

[8] Sander Stuijk, Marc Geilen, and Twan Basten. 2006. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *DAC*.

[9] S. Stuijk, M.C.W. Geilen, and T. Basten. 2006. SDF^3: SDF For Free. In *ACSD*. http://www.es.ele.tue.nl/sdf3

[10] Qi Tang, Twan Basten, Marc Geilen, Sander Stuijk, and Ji-Bo Wei. 2017. Mapping of synchronous dataflow graphs on MPSoCs based on parallelism enhancement. *J. Parallel and Distrib. Comput.* 101 (2017), 79–91.

[11] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*.

[12] Xue-Yang Zhu, Marc Geilen, Twan Basten, and Sander Stuijk. 2016. Multi-constraint static scheduling of synchronous dataflow graphs via retiming and unfolding. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 35, 6 (2016), 905–918.