Towards Memory-Optimal Schedules for SDF

Mitchell Jones[1], Julián Mestre[2,3], and Bernhard Scholz[2(✉)]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, USA
[2] School of Information Technologies, University of Sydney, Sydney, Australia
Bernhard.Scholz@sydney.edu.au
[3] Facebook, Menlo Park, USA

Abstract. The Synchronous Data Flow (SDF) programming model is an established programming paradigm for stream processing applications. SDF programs are expressed by actors and streams that establish communication among actors. Streams are implemented as FIFO buffers, and the size of the FIFO buffers depends on the steady-state schedule. Finding a steady-state schedule that minimizes the sizes of FIFO buffers, is of great importance to minimize the memory consumption. The state-of-the-art provides ad-hoc heuristics only, so finding memory-optimal steady-state schedules is still an open challenge.

In this work, we study three objective functions capturing the memory utilization of three different implementations of the FIFO buffers. We show that one objective is NP-hard to optimize, while the other two can be solved optimally in polynomial time. The algorithm for computing these optimal schedules is implementable as an online algorithm. We show the effectiveness of our new algorithm comparing it with the state-of-the-art heuristics. Our experiments show that for large synthetic instances, our algorithm generates schedules that use up to 8 times less memory.

Keywords: Synchronous Data Flow (SDF) · Scheduling
Optimality · FIFO-buffer

1 Introduction

Stream programming paradigm has its origins in the Kahn's processing model [5] and data-flow computing [3]. Stream programs are a natural fit for applications that process large unbounded regular sequences of data. There are many examples for established stream programming applications including digital signal processing, audio, video, graphics, networking and for big data.

Stream programs are expressed by a set of *actors* and a set of *data channels* between actors. Conceptually, actors are independent processing units with their own memory and program counters. An actor exchanges information with another actor via a data channel using tokens. The channels fully expose the dependencies between actors, and are directed: the *producer* is the actor at the

D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 94–105, 2018.
https://doi.org/10.1007/978-3-319-75178-8_8

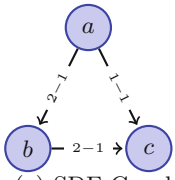
source of a data channel, and the *consumer* is the actor at the destination of the data channel. The data channels are commonly implemented as FIFO buffers, and the size of the FIFO buffers depend on the point in time when actors are executed (also known as *fired*).

If the firing of actors is not coordinated, actors may starve or the memory of FIFO buffers may deplete. To overcome this problem, Synchronous Data Flow (SDF) Model was introduced [2] to bound the size of FIFO buffers and make computations of infinite streams of data deterministic and controllable. In the SDF model, the actor are constrained such that for each actor firing, only a fixed number of tokens are consumed and produced, respectively. For a well-formed SDF program, a finite periodic schedule can be constructed [2] that consists of a finite sequence of actor firings. The schedule can be computed a priori and invokes each actor of the stream graph at least once, and produces no net change in the system state after executing the schedule. I.e., the number of tokens in each data channel is the same before and after executing the schedule. Hence, a periodic schedule can be executed again and again for unbounded regular streams without starving actors and without exhausting memory. The state before and after the execution of a periodic schedule is known as a *steady-state*. Hence, the SDF model is a popular model for stream programming because the memory consumption of the data channels is known a prior at compile time. There are many different steady-state schedules for an SDF program, and the sizes of the FIFO buffers for channels depend on the chosen steady-state schedule. Finding a steady-state schedule that minimizes the sizes of FIFO buffers, is still an open research problem. The current state-of-the-art algorithms for finding steady-state schedules are ad-hoc heuristics only [2] that do not optimize for minimal memory. Hence, stream programs may not fully utilize caches and/or modern massively parallel architectures (e.g. GPGPUs) may need to utilize slower memory rather than fast memory. Hence, finding memory-optimal steady-state schedules is of importance for the SDF model.

Contributions: This work is of theoretic nature and explores the problem of finding memory optimal steady-state schedules in an algorithmic fashion. We anticipate large instances of SDF programs in near future that necessitates new algorithmic contributions for memory-optimal steady-state schedules. We provide three notions of memory optimality based on how FIFO buffers utilize memory. We show for each notion of optimality, algorithmic and complexity theoretic results. We also provide a synthetic set of experiments to show the effectiveness of our new algorithmic approach in comparison with the state-of-the-art algorithm for large instances.

2 Motivating Example

The data-flow model [3] represents a program as a *stream graph* $G = (V, E)$ whose vertices V are called *actors* and whose edges $E \subseteq V \times V$ are called *channels*. A channel $(u, v) \in E$ buffers data elements called *tokens*, which are passed from the output of actor u to the input of actor v. In Fig. 1(a) a stream



(a) SDF Graph

(u,v)	$p(u,v)$	$c(u,v)$
(a,b)	1	2
(b,c)	2	1
(a,c)	1	1

(b) Data Rates

(u,v)	$f(u,v)$
(a,b)	1
(b,c)	0
(a,c)	0

(c) Fill-State

u	$r(u)$
a	2
b	1
c	2

(d) Rep.

$$2 \cdot r(a) = 1 \cdot r(b)$$
$$1 \cdot r(b) = 2 \cdot r(c)$$
$$1 \cdot r(a) = 1 \cdot r(c)$$

(e) Balance Eq.

Fig. 1. Example: stream graph consists of actors a, b, and c; channels are augmented with numbers of produced and consumed tokens for its adjacent actor when its fires. Fill-state, repetition for steady-state, and balance equation are given.

graph is depicted whose actors are a, b and c. The directed edges of the example graph represent channels that transport streams of tokens from the producing actor to the consuming actor. In the following we denote by n and m the number of actors and channels, respectively.

Synchronous dataflow [7] restricts the semantics of the dataflow model by fixing the number of consumed and produced tokens for a single firing of an actor. The number of consumed tokens for a single firing of actor v from an incoming channel $(u, v) \in E$ is given by function $c : E \to \mathbb{N}$. Function $p : E \to \mathbb{N}$ denotes the number of produced tokens for an outgoing channel of an actor. We also refer to the functions p and c as *data rates*. The data rates of our motivating example are shown in Fig. 1(b) and are also depicted as edge annotation in the graph in Fig. 1(a). A *schedule* $s = \langle u_1, \ldots \rangle \in V^*$ is a sequence of actors, where a given actor may occur several times. Each occurrence $u_i \in V$ in the schedule is called an *firing* of actor u_i. A firing of an actor modifies the state of the system by producing and consuming tokens from the channels adjacent to the actor. The *fill-state* of the system is the numbers of tokens on the channels between actor invocations that have been queued but have not been consumed yet. We will specify the fill-state of the system at a given point in time with a function $f : E \to \mathbb{N}$. The fill-state is an abstraction of the actual tokens that are stored on the channels. Let us assume that we have an initial fill-state as given in Fig. 1(c), which implies that there is a single token in channel (a, b) and there are no tokens on channel (b, c), and channel (a, c).

A *periodic schedule* has finite length, includes every actor of the stream graph at least once, and its execution produces no net change in the fill-state after executing the schedule. A periodic schedule may be computed a-priori [7], and executed ad-infinitum without exhausting memory[1]. We refer to the fill-state before and after the execution of a periodic schedule as *steady-state*. A periodic schedule s has a *repetition vector* $r : V \to \mathbb{N}$ that counts the occurrences of each actor in s. The *length* of s is given by $\sum_{u \in V} r(u)$. We denote with $\mathcal{S}$ the set of periodic schedules for a given stream graph instance. Periodic schedules are constrained by two factors. First, recall that every actor needs to be fired at

[1] Under the assumption that the memory consumption for a single actor invocation is bounded.

least once. For our example, actor a, b, and c must occur in schedule s. Second, in order to conserve the fill-state of the FIFO-buffers after the execution of the schedule, for each buffer the number of tokens put into the buffer must equal the number of tokens consumed from the buffer. These constraints give rise to the so-called *balance equations*:

$$p(u,v) \cdot r(u) = c(u,v) \cdot r(v) \qquad \forall\,(u,v) \in E \tag{1}$$

$$r(u) > 0 \qquad \forall\, u \in V \tag{2}$$

The balance equations of the example in Fig. 1 are given in the Fig. 1(e) with the additional constraint that $r(a) > 0$, $r(b) > 0$, and $r(c) > 0$ where $r(u)$ are the *repetitions* for actor u, i.e., there must be $r(u)$ occurrences of actor u in the schedule s. Finding the smallest integral repetitions for actors can be expressed as a problem of finding the smallest integral vector in the null-space of the *topological matrix* [7]. It is known that for connected stream graphs, an integral repetition vector satisfying Eqs. (1) and (2) exists if and only if the *topological matrix* of the stream graph has rank $n - 1$. The repetitions of the motivating example in Fig. 1(a) is shown in Fig. 1(d).

Algorithm 1. GREEDY$((V, E, p, c), t)$

```
1.  L ← Σ_{u∈V} r(u)
2.  let F be the set of fireable actors in V using fill-state t
3.  let D be the set of deferrable actors in F
4.  for i = 1 to L do
5.    if F \ D ≠ ∅ then
6.      u ← an actor from F \ D
7.    else
8.      u ← an actor in F that increases total number of tokens the least
9.    add u to the schedule s
10.   r(u) ← r(u) − 1
11.   invoke actor u
12.   update F and D                    // An actor u is not fireable if r(u) < 1.
13. return s
```

For the example, actor a is invoked twice since it produces only one token on the edge (a, b), but to fire b at least once it needs to consume two tokens on (a, b). Actor c is invoked twice because a is invoked twice, and c only consumes a single token for every token produced by a along the channel (a, c). No smaller repetition vector can be found. To find a schedule s, a greedy heuristic was devised by Battacharyya *et al.* [2] (cf. Sect. 3.3.2). The heuristic is outlined in Algorithm 1. The goal of GREEDY is to minimize the sum of the maximum number of tokens required for each channel over a periodic schedule. Given a graph G, and the initial delay t as part of the input, it returns a schedule s. Note that we say an actor v is *fireable*, if for every incoming edge (u, v), $f(u, v) \geq$

$c(u, v)$. An actor v is *deferrable* if it is fireable, and for at least one of its outgoing edges (v, u) (that is not a transitive edge[2]) it holds that $f(v, u) \geq c(v, u)$.

For our motivating example in Fig. 1(a), the greedy algorithm will fail to find the optimal periodic schedule. For the example we further assume, that we have an initial delay of one token on edge (a, b). To minimize the memory consumption the optimal periodic schedule to use is $s = \langle a, b, c, a, c \rangle$.

During the firing of these actors, we can keep track of the maximum number of tokens needed for each channel. For the edge (a, b), the initial fill-state $f(a, b)$ on the edge is 1. We first fire actor a, and thus the fill on the edge (a, b) is increased to 2. Similarly, the edge (a, c) is now storing a single token. Next, actor b is invoked, consuming two tokens from the channel (a, b) and producing two tokens on the channel (b, c). Actor c fires, which consumes one token from edge (b, c) and (a, c). At this point, the edge (a, b) has a fill of zero. In order to ensure this schedule is periodic, we must fire a again, producing a single token on both (a, b) and (a, c). We return back to our initial fill-state by firing c once more. Over this execution, we can see that the maximum number of tokens on the channels (a, b) and (b, c) is two, while the maximum number of tokens on the channel (a, c) is one. Summing this together, this schedule requires memory to store at most five tokens during this execution.

However, it is entirely possible that the greedy algorithm could choose the schedule $s' = \langle a, b, a, c, c \rangle$. By executing each actor in s' one-by-one, we can observe that the maximum number of tokens used by the channels (a, b), (b, c) and (a, c) is two. Thus enough memory will be needed to store six tokens, which is clearly suboptimal compared to the schedule s.

To see why this is the case, note that the difference between s and s' is the order of the 3rd and 4th actors a and c. After firing a and then b, both a and c are fireable but not deferrable. Notice that a is not deferrable, even though there is an outgoing edge that meets the consumption requirements ($f(a, c) \geq c(a, c)$), the edge (a, c) is a transitive edge. Since $f(a, b) \not\geq c(a, b)$ at this point in time, a does not meet the criteria to be marked as a deferrable actor. Thus, the greedy algorithm can choose to fire either actor a or c. The greedy algorithm could make the suboptimal choice to fire a again (Line 6 of Algorithm 1), instead of firing c, which would consume the single token on the channel (a, c). By choosing to fire a, the greedy algorithm produces a second token on the channel (a, c). This leads to the schedule s'.

3 Problem Statement

Stream programs admit an exponential number of periodic schedules. In fact, given a repetition vector r, any sequence of length $L = \sum_{u \in V} r(u)$ where actor u occurs $r(u)$ times is a periodic schedule. Therefore, the number of periodic schedules is given by $|\mathcal{S}| = \frac{L!}{\prod_{u \in V} r(u)!}$. Among these schedules some consume less

[2] We say that a directed edge (v, u) is a transitive edge in a graph $G = (V, E)$ if there exists a directed path from v to u in G using only the edges $E \setminus \{(v, u)\}$.

memory on their FIFO-buffers than others. How much memory a given schedule consumes will depend on the implementation details of the FIFO-buffers and on the evolution of the fill-state over the execution of the schedule.

Recall that the fill-state function keeps track of the number of tokens stored on each channel waiting to be consumed. Given the current fill-state of the system, it is possible to determine the fill-state after the execution of the particular actor. Therefore, given the initial fill-state, we can easily compute the fill-state after the i-th step of the schedule execution. The fill-state function $f_s^i : E \to \mathbb{N}$ defines the fill-state of channel (u, v) after the i-th execution step of schedule s and may be defined as $f_s^0(u, v) = t(u, v)$ for the first step, and

$$f_s^{i+1}(u,v) = \begin{cases} f_s^i(u,v) + p(u,v), & \text{if } u = s(i+1), \\ f_s^i(u,v) - c(u,v), & \text{if } v = s(i+1), \\ f_s^i(u,v), & \text{otherwise.} \end{cases}$$

where $t(u, v)$ is initial fill-state of (u, v) at the beginning of the execution of s.

A periodic schedule and an initial fill-state are said to be *admissible* if the schedule can be executed without ever running out of tokens on any channel.

Definition 1. *Finite periodic schedule s with initial fill-state t is admissible, if*

$$\begin{aligned} &f_s^i(u,v) \geq 0, && \forall (u,v) \in E,\ i \in \{0,\ldots,L\} \\ &f_s^0(u,v) = f_s^L(u,v), && \forall (u,v) \in E \end{aligned}$$

It is worth noting that for each periodic schedule $s \in \mathcal{S}$ there exists an initial fill-state that makes it admissible. Therefore, all we need is a method for deciding which periodic schedule to use.

We study three objective functions that capture the memory utilization of the system under different implementations of the FIFO buffers. In each case the goal is to compute an admissible schedule (s, t), the only difference is the objective being optimized:

(P1) The Min-Max-Max Problem:

$$\min_{(s,t)} \max_{0 \leq i \leq L} \max_{(u,v) \in E} f_s^i(u,v)$$

(P2) The Min-Sum-Max Problem:

$$\min_{(s,t)} \sum_{(u,v) \in E} \max_{0 \leq i \leq L} f_s^i(u,v)$$

(P3) The Min-Max-Sum Problem:

$$\min_{(s,t)} \max_{0 \leq i \leq L} \sum_{(u,v) \in E} f_s^i(u,v)$$

The objective **(P1)** minimizes the maximum buffer requirement across all buffers. This objective captures a simplistic implementation of FIFO buffers where space is allocated ahead of time and buffers have uniform length. The objective **(P2)** minimizes the sum of the maximum requirements. This objective captures a simple implementation of FIFO buffers where space is allocated ahead of time, but different buffers can differ in size. The objective **(P3)** minimizes the maximum combined size of all buffers at any point in time. This objective capture a more sophisticated implementation where buffer space can be acquired and released dynamically.

4 Scheduling to Minimize Memory Usage

In this section we consider the objectives defined in Sect. 3 under the assumption that the initial fill-state of each buffer can be set arbitrarily. In other words, given an instance (V, E, c, p), the goal is to compute a schedule s and an initial fill-state $t : E \to \mathbb{N}$ so that the schedule is admissible and one of the three objectives **(P1–P3)** is minimized.

Our algorithm for Min-Max-Max and Min-Sum-Max assumes the Balance Eqs. (1) and (2) for the instance are feasible and that we are given the smallest integral repetition vector r that the instance admits. In addition to the instance (V, E, p, c) the algorithm take as a parameter a permutation of the actors, we use $\pi : V \to [1, n]$ to the denote the position of each actor within this permutation.

First, the algorithm computes for each channel the appropriate initial fill-state that will ultimately make the schedule admissible. Second, each actor u is added to a priority queue with priority 0. The algorithm then enters an infinite loop where in each iteration we remove from the priority queue the actor u with the smallest key x (if there are several actors with the same key, we break ties using the permutation order), we invoke u, and re-insert u with priority $x + \frac{1}{r(u)}$. The pseudo-code of the procedure is given in Algorithm 2.

Notice that for each actor u, its priority becomes 1 after $r(u)$ invocations. Therefore, after $L = \sum_{u \in V} r(u)$ executions of the while loop every actor has priority 1. At this point in time, the schedule executed thus far is periodic. We call this periodic schedule, the canonical schedule induced by π and denote it by (s_π, t_π). Notice, however, that Algorithm 2 itself never ends. Indeed, after the L-th iteration the while loop goes on to repeat this periodic schedule ad-infinitum.

The proof of correctness hinges on the following observation on the minimum buffer size of a channel based on the data rates of its endpoints.

Lemma 1 ([2, Theorem 3.3]**).** *Let (u, v) be a channel. In any admissible schedule, the buffer for channel (u, v) has size at least $p(u, v) + c(u, v) - \gcd(p(u, v), c(u, v))$ at some point in time during the execution of the schedule.*

Proof. For sake of brevity, let us denote $p(u, v)$ with a, $c(u, v)$ with b, and $a + b - \gcd(a, b)$ with $\textsc{lb}(a, b)$. Let (s, t) be an admissible schedule. Since we are interested in deriving a lower bound on the buffer size for channel (u, v), we assume without loss of generality that this is the only channel in the graph.

If $a \mid b$ then the buffer size has to be at least $b = \text{LB}(a, b)$, so the lemma follows. Let us then assume from now on that $1 < a < b$ and that b is not a multiple of a; a symmetric argument can be used to handle the cases $b \mid a$ and $1 < b < a$.

Consider two executions of v in s. If we have two consecutive executions of v, or the schedule begins and ends with v, then the buffer size is at least $2b > \text{LB}(a, b)$. So let us assume this does not happen. We apply the following transformation to the schedule: If the schedule starts by executing v, then we fuse every execution of v with the execution of u that immediately follows it into a new actor v'; if the schedule starts by scheduling u, then we fuse every execution of v with the execution of u that immediately precedes it into a new actor v'. The result is a schedule for a new channel (u, v') with production rate a and consumption rate $b - a$. It is easy to check that if $\text{LB}(a, b - a)$ is a lower bound on the buffer size for the new channel, then $\text{LB}(a, b - a) + a$ is a lower bound on the buffer size for the original channel.

A simple proof by induction finishes the argument if $a \mid b$ then we are at the base of the inductive proof. Otherwise, the size of the buffer for the channel (u, v) must be at least

$$\begin{aligned}\text{LB}(a, b - a) + a &= a + b - a - \gcd(a, b - a) + a \\ &= a + b - \gcd(a, b - a) \\ &= a + b - \gcd(a, b),\end{aligned}$$

where the last equality follows by Euclid's algorithm.

Now that we have a lower bound on the size of each buffer, we will prove that these bounds are attained simultaneously by our algorithm.

Lemma 2. *For any permutation π, the schedule (s_π, t_π) is admissible and for each channel (u, v) the maximum size of the buffer during the execution of the schedule is $p(u, v) + c(u, v) - \gcd(p(u, v), c(u, v))$.*

Algorithm 2. CANONICAL$((V, E, p, c), \pi)$

1. **for** $(u, v) \in E$ **do**
2. $\quad t_\pi(u, v) \leftarrow \begin{cases} c(u, v) - \gcd(p(u, v), c(u, v)) & \text{if } \pi(u) < \pi(v) \\ c(u, v) & \text{if } \pi(v) < \pi(u) \end{cases}$
3. let Q be an empty priority queue
4. **for** $u \in V$ **do**
5. $\quad$ `insert` u with priority 0 into Q
6. **while** true **do**
7. $\quad (u, x) \leftarrow$ `delete-min`(Q) // break ties using the π order
8. $\quad$ execute actor u
9. $\quad$ `insert` u with priority $x + \frac{1}{r(u)}$ into Q

Proof. We prove the bounds on the size of the buffer for a fixed, but arbitrary, channel (u, v). For sake of brevity, let us denote $p(u, v)$ with a and $c(u, v)$ with b. Recall that Balance Eq. (1) for channel (u, v) implies $\frac{r(u)}{r(v)} = \frac{b}{a}$.

First, consider the case $\pi(u) < \pi(v)$. Notice that the 1st execution of v is preceded by an execution of u. In general, the k+1st execution of v is preceded by

$$\left\lfloor \frac{k\frac{1}{r(v)}}{\frac{1}{r(u)}} \right\rfloor + 1 = \left\lfloor k\frac{r(u)}{r(v)} \right\rfloor = \left\lfloor \frac{kb}{a} \right\rfloor + 1$$

executions of u. Therefore, the fill-state of the channel after the $k+1$st execution of v is precisely

$$t_\pi(u, v) + a\left(\left\lfloor \frac{kb}{a} \right\rfloor + 1\right) - (k+1)b.$$

Using the fact that $t_\pi(u, v) = b - \gcd(a, b)$ when $\pi(u) < \pi(v)$, we can show that the fill-state of the channel is always non-negative:

$$\begin{aligned} t_\pi(u, v) + a\left(\left\lfloor \frac{kb}{a} \right\rfloor + 1\right) - (k+1)b = \\ = a\left(\left\lfloor \frac{kb}{a} \right\rfloor + 1\right) - kb - \gcd(a, b) \geq \gcd(a, b) - \gcd(a, b) = 0, \end{aligned}$$

where the inequality follows from the fact that $a\left(\left\lfloor \frac{kb}{a} \right\rfloor + 1\right) - kb > 0$ and Bézout's Lemma [9].

On the other hand, just before before the $k+1$st execution of v, the fill-state of the buffer is

$$t_\pi(u, v) + a\left(\left\lfloor \frac{kb}{a} \right\rfloor + 1\right) - kb.$$

Again, using the fact that $t_\pi(u, v) = b - \gcd(a, b)$ we get

$$t_\pi(u, v) + a\left(\left\lfloor \frac{kb}{a} \right\rfloor + 1\right) - kb = a + b - \gcd(a, b) + a\left\lfloor \frac{kb}{a} \right\rfloor - kb \leq a + b - \gcd(a, b),$$

so the buffer size never exceeds $a + b - \gcd(a, b)$.

Now consider the case $\pi(v) < \pi(u)$. In this case the $i+1$st execution of v is preceded by $\left\lceil \frac{ib}{a} \right\rceil$ executions of u. A similar argument (but using the fact that $t_\pi(u, v) = b$ when $\pi(u) > \pi(v)$) shows that the schedule is admissible and that the maximum buffer size is $a + b - \gcd(a, b)$.

Combining the lower bound from Lemma 1 and the upper bound from Lemma 2 we get that every canonical schedule is an optimal solution for **(P1)** and **(P2)**. The following theorem summarizes the results in this section.

Theorem 1. *There is a polynomial time algorithm for computing an optimal periodic schedule for the objectives **(P1)** and **(P2)** with flexible initialization. Furthermore, the schedule can be computed online using $\Theta(n)$ space and $O(\log n)$ time per actor invocation.*

Proof. The optimality of the objectives **(P1)** and **(P2)** follows immediately from Lemmas 1 and 2. The complexity claims follow from using a priority queue implementation that uses $\Theta(n)$ space and performs `insert` and `delete-min` operations in $O(\log n)$ time.

We contrast our positive results from the previous section by showing that it is NP-hard to optimize the Min-Max-Sum (cf. **(P3)**).

Theorem 2. *It is NP-hard to optimize **(P3)** with flexible initialization.*

The theorem can be shown by reducing the Minimum Feedback Arc Set (MFAS) problem to our problem.

5 Experiments

With the advent of stream programming we anticipate large instances of stream programs. In the absence of large stream programs, we have generated complete, directed, acyclic graphs as a synthetic benchmark suite. We generated the graphs as follows: We start with a directed graph $G = (V, E)$ of n nodes, and number the vertices $v_1, v_2, \ldots v_n$. For each vertex $v_i \in V$, we select a random repetition value $r(v_i)$ uniformly at random from the range $\{1, \ldots, n\}$. We then iterate through every pair of vertices $v_i, v_j \in V$. If $i \neq j$ and $i < j$, we add the directed edge (v_i, v_j) to E, with $p(v_i, v_j) = \frac{r(v_j)}{\gcd(r(v_i), r(v_j))}$ and $c(v_i, v_j) = \frac{r(v_i)}{\gcd(r(v_i), r(v_j))}$. This generation template guarantees that a repetition vector exists, and the topological matrix of this directed, acyclic graph has rank $n - 1$.

Using this approach, we generated graphs of size $n = 10, 15, \ldots, 50$ and ran both algorithms. As before, we timed the execution of each algorithm, taking the average over twenty runs. The numerical results of these experiments are shown in Table 1 and are visualized in Fig. 2.

Table 1. Performance comparison on randomly generated instances.

Instance	CANONICAL			GREEDY	
$\|V\|$	**(P1)**	**(P2)**	Time (s)	**(P2)**	Time (s)
10	20	586	0.0030	1226	0.0097
15	28	1048	0.0047	2937	0.0362
20	32	2483	0.0081	7818	0.1124
25	48	6131	0.0143	30306	0.5421
30	60	9486	0.0188	47979	1.6658
35	70	16782	0.0291	68126	5.0272
40	80	22927	0.0352	149469	8.3609
45	84	29781	0.0454	244380	17.6809
50	100	46203	0.0567	347676	39.5296

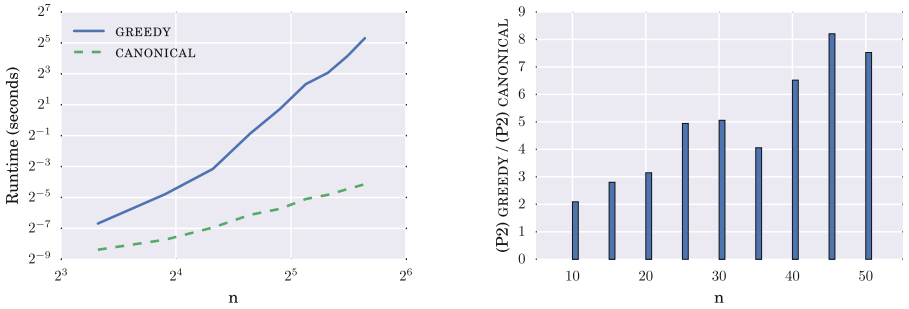


Fig. 2. Visualizing the performance on randomly generated instances.

Two observations stand out from Fig. 2. First, CANONICAL seems to be asymptotically faster GREEDY. We suspect that this is due to the fact that as the graph becomes denser, there will be many "fireable" actors in each iteration, leading GREEDY to spend nearly quadratic time per iteration, whereas CANONICAL, is guaranteed to spend at most logarithmic time. Second, GREEDY was never able to find an optimal schedule and the quality of the solutions it produced deteriorated as n grew.

These experiments strongly support the hypothesis that GREEDY is slower and produces worse schedules with respect to memory consumption when more actors become "fireable" at each iteration.

6 Related Work

Our work is closely connected to the greedy algorithm proposed in [2]. Their approach is based on a heuristic which keeps track of the set of "fireable" actors. Our algorithm is based on optimality theorems which produces both the optimal memory schedule and the required initial delay to achieve optimality. With a given initial delay, the NP-hard proof of the problem is given in [2]. The approach in [4] uses a model-checking method to find optimal schedules which requires a machinery which is outside of the complexity class P. There is a stream of literature on scheduling of SDF programs with model checking. Other approaches use time automata to solve the scheduling problem for SDF [1], and variations of the problem definition taking other metrics such as throughput into account [8]. There is also related work concerning to eliminate buffers via unrolling the finite periodic schedules [6].

7 Conclusion

In this work, we have studied three mathematical definitions of memory optimality based on how FIFO buffers utilize memory. We started by showing that two of these objectives can be solved in logarithmic worst-case time per actor

invoked, and that the last objective is NP-hard. Experiments showed that our new algorithm drastically outperformed existing heuristics in both speed and the memory usage of schedules produced on dense instances.

Acknowledgements. This project has been partially supported by the Australian Research Council Discovery Project DP1096445. We would like to thank our reviewers for the insightful feedback, and Yousun Ko for her help with the experiments.

References

1. Ahmad, W., de Groote, R., Hölzenspies, P.K.F., Stoelinga, M., van de Pol, J.: Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. In: Proceedings of the 2014 14th International Conference on Application of Concurrency to System Design, ACSD 2014, pp. 72–81. IEEE Computer Society, Washington, DC (2014). https://doi.org/10.1109/ACSD.2014.13
2. Battacharya, S.S., Murthy, P.K., Lee, E.A.: Software Synthesis from Dataflow Graphs. Kluwer Academic Publishers, Norwell (1996)
3. Dennis, J.B.: First version of a data flow procedure language. In: Robinet, B. (ed.) Programming Symposium. LNCS, vol. 19, pp. 362–376. Springer, Heidelberg (1974). https://doi.org/10.1007/3-540-06859-7_145
4. Geilen, M., Basten, T., Stuijk, S.: Minimising buffer requirements of synchronous dataflow graphs with model checking. In: Proceedings of the 42nd Annual Design Automation Conference, DAC 2005, pp. 819–824. ACM, New York (2005). https://doi.org/10.1145/1065579.1065796
5. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress, pp. 471–475 (1974)
6. Ko, Y., Burgstaller, B., Scholz, B.: LaminarIR: compile-time queues for structured streams. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 121–130. ACM, New York (2015). https://doi.org/10.1145/2737924.2737994
7. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proc. IEEE **75**(9), 1235–1245 (1987)
8. Wiggers, M.H., Bekooij, M.J.G., Smit, G.J.M.: Buffer capacity computation for throughput-constrained modal task graphs. ACM Trans. Embed. Comput. Syst. **10**(2), 17:1–17:59 (2011). https://doi.org/10.1145/1880050.1880053
9. Wikipedia: Bézout's identity (2016). https://en.wikipedia.org/wiki/Bzout%27s_identity. Accessed 12 Aug 2016