An Efficient Technique for Analysis of Minimal Buffer Requirements of Synchronous Dataflow Graphs with Model Checking

Weichen Liu
Hong Kong University of Science and Technology
Hong Kong, China
weichen@cse.ust.hk

Zonghua Gu
Zhejiang University
Hangzhou, China
zonghua@gmail.com

Jiang Xu
Hong Kong University of Science and Technology
Hong Kong, China
jiang.xu@ust.hk

Yu Wang
Tsinghua University
Beijing, China
yu-
wang@tsinghua.edu.cn

Mingxuan Yuan
Hong Kong University of Science and Technology
Hong Kong, China
csyuan@cse.ust.hk

ABSTRACT

Synchronous Dataflow (SDF) is a widely-used model of computation for digital signal processing and multimedia applications, which are typically implemented on memory constrained hardware platforms. SDF can be statically analyzed and scheduled, and the memory requirement for correct execution can be predicted at compile time. In this paper, we present an efficient technique based on model-checking for exact analysis of minimal buffer requirement of an SDF graph to guarantee deadlock-free execution. Performance evaluation shows that our approach can achieve significant performance improvements compared to related work.

Categories and Subject Descriptors

C.4 [**Performance of systems**]: Modeling techniques

General Terms

Algorithms, Performance

Keywords

Scheduling, memory management, model checking, synchronous dataflow, optimization

1. INTRODUCTION

Synchronous Dataflow (SDF) [14] is a widely-used model of computation for signal processing and multimedia applications. An SDF graph consists of actors that produce/consume a constant number of tokens on its output/input edges at every firing. SDF can be statically analyzed and scheduled, and can be used to generate efficient implementations in terms of (data and code) memory size and runtime overhead. Since embedded systems running DSP applications typically have limited memory resource, it is desirable to minimize the memory size requirement of the software implementation. The memory requirement of an application consists of two parts: code memory and data buffer memory. Code memory can be minimized by constructing a *Single-Appearance Schedule (SAS)* [15], in which each actor invocation appears exactly once in the program body, but this may lead to a significant increase in data memory. Some authors [16] have developed efficient quasi-static scheduling techniques to minimize data memory with (slightly) increased runtime overhead. In this paper, we assume that this increased runtime overhead is acceptable, so that we do not require the schedule to be a SAS. We focus on minimizing the data memory, and present an approach based on the model-checker SPIN to find the minimum buffer size requirement of an SDF graph that guarantees its deadlock-free execution.

The rest of the paper is structured as follows: we discuss background and related work in Section 2; present the Bounded Greedy Algorithm (BGA) for the schedulability test of SDF graphs in Section 3. We present the SPIN model for minimal buffer analysis in Section 4; performance evaluation results in Section 5, and conclusions in Section 6.

2. BACKGROUND AND RELATED WORK

We first introduce some notations used in the paper. For a given SDF graph $G(V, E)$, the *Buffer Size Distribution* $BSD = \{b(e) \in N \mid e \in E\}$ is a vector of size $|E|$, where $b(e)$ is the buffer size of edge e. For a directed edge e, its source (producer) actor $src(e)$ produces $p(e)$ tokens per firing; its sink (consumer) actor $sink(e)$ consumes $c(e)$ tokens per firing. $p(e)$ and $c(e)$ are called the token production and consumption rate of edge e, respectively. The number of initial tokens, also called *initial delays*, on edge e is denoted $d(e)$. For an actor $v \in V$, $in(v)$ denotes the set of incoming edges to v; $out(v)$ denotes the set of outgoing edges from v.

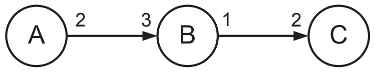

Figure 1: A simple SDF graph.

During execution, the current remaining number of tokens on edge e is denoted $r(e)$.

As an example, Fig. 1 shows a simple SDF graph with no initial delays. Each invocation of actor A produces 2 tokens on edge AB; each invocation of actor B consumes 3 tokens on edge AB and produces 1 token on edge BC; each invocation of actor C consumes 2 tokens on edge BC. For edge AB, $src(AB) = A$, $sink(AB) = B$, $p(AB) = 2$, $c(AB) = 3$, $d(AB) = 0$.

The *balance equation* for each edge e encodes the constraint that the total number of tokens produced is equal to the total number of tokens consumed in a single iteration of the periodic schedule:

$$\begin{cases} r_A * 2 = r_B * 3 \\ r_B * 1 = r_C * 2 \end{cases}$$

with the solution $r_A = 3; r_B = 2; r_C = 1$, where r_i is the number of invocations of actor i in a periodic schedule. Given the actor ordering ABC, the *repetition vector* is (3,2,1). The *repetition counts* of actors A, B and C are their respective entries in the repetition vector: 3, 2, 1. The *Expected Schedule Length* (ESL) is the sum of repetition counts of all actors (3+2+1=6). One feasible static schedule is $AAABBC$. Starting from a given initial state, if the actors fire in the sequence $AAABBC$, then the SDF graph goes back to the initial state, where there are no tokens on either edges for the example. Therefore, we can execute this sequence of actor firings repeatedly without deadlock caused by buffer overflow or underflow. In this paper, we assume that each edge has its separate buffer space without sharing. The schedule $AAABBC$ has total buffer size requirement of 8, with buffer size distribution $\{b(AB) = 6, b(BC) = 2\}$. Another feasible schedule is $AABABC$, with a smaller total buffer size requirement of 6, with buffer size distribution $\{b(AB) = 4, b(BC) = 2\}$. In fact, it turns out that 6 is the minimum buffer size requirement for this SDF graph, i.e., there does not exist any feasible schedule if total buffer size is less than 6.

An SDF graph is *schedulable* if it has a correct infinite sequential execution that executes in bounded memory. The *SDF buffer minimization* problem is the problem of finding the buffer size distribution with minimum total buffer size while still making the SDF graph schedulable. Ade et al [3] presented formulas on the upper bounds on the minimum buffer memory requirement for some restricted subclasses of delayless, acyclic graphs, including arbitrary-length chain-structured graphs. But for general SDF graphs, the buffer minimization problem is NP-complete. Model-checking is an effective technique for solving NP-complete combinatorial optimization and scheduling problems. Geilen [7] and Gu [9] used the model-checkers SPIN [10] and NuSMV [5], respectively, to solve the SDF buffer minimization problem, but the scalability of these techniques is limited due to state space explosion. In this paper, we use the model-checker SPIN to solve the buffer minimization problem, but with a modeling technique that is quite different from those in [7] and [9]. While it does not change the NP-complete nature of the problem, performance evaluation shows that our approach can achieve impressive improvements on scalability and efficiency, compared to [7] and [9].

Parks [17] presented an algorithm for scheduling a Kahn Process Network (KPN) in bounded memory if it is possible: start with a arbitrary bound on the capacity of all buffers; use any scheduling algorithm that avoids buffer overflow or underflow; if system deadlocks because of buffer overflow, increase size of smallest buffer and continue. Since SDF is a special case of KPN, Parks' algorithm can also be applied to an SDF graph to find a buffer size distribution so that the SDF graph does not deadlock. However, it is not guaranteed to find the *minimum* buffer size requierment. To do this, we add backtracking search to the algorithm, i.e., whenever deadlock occurs, we determine the set of *constraining edges*[1], and increase buffer size of each of the constraining edges in order to resolve the deadlock. When a feasible buffer size distribution is found, we backtrack and explore other branches in the search tree in the search for the *minimum* feasible buffer size distribution until the entire search tree has been explored.

3. THE BOUNDED GREEDY ALGORITHM

Let $h(v)$ denote the accumulated number of firings of actor v since the beginning of execution. The *loop count* $lc(v)$ of actor v is the maximum number of times that actor v can fire consecutively without interleaved firings of other actors, which is constrained by three factors: number of tokens in its input buffer(s), number of empty spaces in its output buffer(s), and its repetition count (entry in the repetition vector $q(v)$) minus its accumulated number of firings $h(v)$:

$$lc(v) = \min\left\{q(v) - h(v), k(v), l(v)\right\} \tag{1}$$

where

$$k(v) = \min_{e_i \in in(v)} \left\lfloor \frac{r(e_i)}{c(e_i)} \right\rfloor \tag{2}$$

$$l(v) = \min_{e_j \in out(v)} \left\lfloor \frac{b(e_j) - r(e_j)}{p(e_j)} \right\rfloor. \tag{3}$$

The term *Bounded Greedy Algorithm* is named due to the fact that each actor fires for the maximum number of times (*greedy*) consecutively while respecting the Expected Schedule Length constraint (*bounded*). After actor v fires for $lc(v)$ number of times, the number of tokens on its input and/or output edge buffer(s) are updated as follows (Of course, actor v does not fire if $lc(v) = 0$):

$$\forall e_i \in in(v), r(e_i) = r(e_i) - lc(v) * c(e_i) \tag{4}$$

$$\forall e_j \in out(v), r(e_j) = r(e_j) + lc(v) * p(e_j). \tag{5}$$

Alg. 1 shows the pseudo-code for BGA. The input to BGA is an SDF graph G and a fixed buffer size distribution BSD. BGA is a deterministic firing rule: it tries to fire each actor one-by-one in a fixed order called the <u>*Actor Appearance Order*</u> (AAO) (Lines 4-10)[2]. In each iter-

[1]Edge e is a constraining edge in a deadlocked state if its producer actor cannot fire due to insufficient buffer space on edge e.

[2]The AAO can be chosen arbitrarily without affecting correctness of the algorithm, but it does affect its efficiency. If the SDF graph is a DAG (Directed Acyclic Graph), a good heuristic is to use one of the topological sorts.

Algorithm 1 Bounded Greedy Algorithm (BGA)

```
Require: SDF graph $G$ with a given buffer size distribution
 1: $sl = 0$
 2: while $true$ do
 3:   $prev\_sl = sl$;
 4:   for each actor $v$ in Actor Appearance Order do
 5:     $lc(v) = \text{ComputeLoopCount}(v, G)$
 6:     if $lc(v) > 0$ then
 7:       $\text{FireActor}(v, lc(v), G)$
 8:       $sl += lc(v)$
 9:     end if
10:   end for
11:   if $sl == ESL(G)$ then
12:     $schedulable = true$
13:     break
14:   else if $prev\_sl == sl$ then
15:     $schedulable = false$
16:     break
17:   end if
18: end while
19: return $schedulable$
```

ation, each actor v fires for $lc(v)$ number of times as determined by Equation (1) (Lines 5 to 9). The actual actor firing is performed in $FireActor()$, which consumes (produces) tokens from its input (output) edges (Line 7) according to Equations (4) and (5). This process iterates until the schedule length reaches the ESL, so a feasible schedule has been found, and we declare $schedulable == true$; or none of the actors could fire in the previous iteration ($prev\_sl == sl$), which means that deadlock has occurred, so we declare $schedulable == false$. Even though not shown in Alg. 1, the set of constraining edges can be easily determined upon detection of a deadlock.

As an example, consider the SDF graph in Fig. 1 with Expected Schedule Length $ESL = 6$. Suppose we adopt the Actor Appearance Order of ABC. Consider different buffer size distributions:

- If the buffer size distribution is $\{b(AB) = 6, b(BC) = 2\}$, then in the first iteration, we try to fire actors A, B and C in turn. Actor A fires for $lc(A) = 3$ times; followed by actor B firing for $lc(B) = 2$ times; followed by actor C firing for $lc(C) = 1$ time. Now the schedule length sl has reached the ESL 6, so $schedulable = true$, and the periodic schedule $AAABBC$ has been found after the first iteration.

- If the buffer size distribution is $\{b(AB) = 4, b(BC) = 2\}$, then actor A fires for $lc(A) = 2$ times; followed by actor B firing for $lc(B) = 1$ time; Since $lc(C) = 0$, actor C cannot fire. Now $sl = 2 + 1 = 3$, which is less than the ESL of 6. Since 2 out of 3 actors have fired, there is no deadlock. So we continue with the next iteration to fire actor A for 1 time, actor B for 1 time, and actor C for 1 time. Now $sl = 6$, so $schedulable = true$, and the periodic schedule $AABABC$ has been found after two iterations.

- If the buffer size distribution is $\{b(AB) = 3, b(BC) = 2\}$, then actor A fires for $lc(A) = 1$ time. Now there are not enough tokens on edge AB for actor B to fire, $lc(B) = 0$, so we move on to actor C to find that $lc(C) = 0$. Since 1 out of 3 actors has fired, there is no deadlock yet. So we continue with the next iteration to try to fire A again. But we find that $lc(A) = 0$ due to insufficient output buffer space on edge AB. So we move on to actors B and C in turn, and find $lc(B) = 0$ and $lc(C) = 0$. Since none of the actors are able to fire in this iteration, before the ESL of 6 is reached, we conclude that a deadlock has occurred, and $schedulable = false$.

BGA is similar to the firing rule used to construct the Dynamic Loop-Count Single-Appearance Schedule (dlcSAS) [16]. The difference is that for the firing rule of dlcSAS, the loop count is computed without the constraint that it must not exceed its repetition count, i.e, instead of Equation (1), we have:

$$lc(v) = \min\{k(v), l(v)\}$$

The firing rule for dlcSAS is designed for generating efficient code from an SDF graph with a given fixed buffer size distribution. The code size is minimum, i.e., each actor firing appears only once in the loop, but each actor may have different firing counts in different iterations of loop execution. (In fact, BGA can also be used as the firing rule in dlcSAS to generate code.) The buffer size distribution can be set with any heuristic or optimal scheduling algorithm that guarantees deadlock-free execution of the SDF graph. Therefore, there is no need for deadlock detection for dlcSAS like in BGA.

We first present some definitions and theorems excerpted from [14].

Definition 1 (Topology Matrix). *[14] The topology matrix of an SDF graph is a matrix with a column for each actor and a row for each edge. The entry at position (i, j) is the number of tokens produced by actor j on edge i at each actor j's firing. If actor j consumes tokens from edge i, then this number is negative; and if it is not connected to edge i, then this number is 0.*

Definition 2 (SSSA). *[14] Given an SDF graph with topology matrix Γ and repetition vector q, s.t. $\Gamma q = 0$, and an initial state for the edge buffers, the ith actor is* runnable *at a given time if it has not been run q_i times and running it will not cause buffer underflow, i.e., the number of tokens goes negative. An SDF Sequential Scheduling Algorithm (SSSA) is any algorithm that schedules an actor if it is runnable, updates edge buffers and stops (terminates) only when no more actors are runnable. If an SSSA terminates before each actor has been fired the number of times specified in the q vector, then it is said to be* deadlocked.

Theorem 1. *[14] Given an SDF graph with topology matrix Γ and repetition vector q s.t. $\Gamma q = 0$, if a PASS (periodic admissible sequential schedule) with period $p = 1^T q$ exists, where 1^T is a row vector full of ones, any SSSA algorithm will find such a PASS.*

According to Theorem 1, whenever multiple actors are runnable during the simulation, an SSSA can make an arbitrary choice as to which one to fire, and it will always terminate and find a periodic admissible sequential schedule (PASS) with length equal to the Expected Schedule Length,

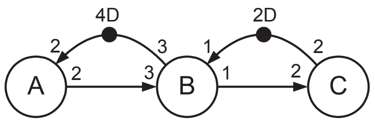

Figure 2: The SDF graph in Fig. 1 with backedges encoding buffer size constraints $b(AB) = 4$ and $b(BC) = 2$.

or it will run into a deadlock before such a PASS is found[3]. On the other hand, if an SDF graph causes an SSSA to run into a deadlock before a PASS is found, then it will cause *any other* SSSA to also run into a deadlock. Hence Theorem 1 is equivalent to the statement that *an SDF graph is schedulable if and only if it is schedulable with an SSSA*. Since BGA is obviously an SSSA, we can conclude that *an SDF graph is schedulable if and only if it is schedulable with BGA*.

In the above discussions, there are no buffer size constraints for the SDF graph. Consider the definition of *runnable*: "actor i is runnable if it has not been run q_i times and running it will not cause buffer *underflow*. " If each edge has a limited buffer size, then we need to add one additional constraint to the definition of runnable to be "actor i is runnable if it has not been run q_i times and running it will not cause buffer *underflow* or *overflow*." Since buffer size constraints can be encoded by adding additional *back-edges* to the original SDF graph without buffer size constraints to generate a new SDF graph without explicit buffer size constraints [18], the schedulability of an SDF graph with buffer size constraints can be checked by testing the new SDF graph without buffer size constraints. Since BGA can be applied on the SDF graph with back-edges encoding buffer size constraints, which is still an SDF graph in nature, the discussions above still hold true for SDF graphs with buffer size constraints. As an example, the SDF graph in Fig. 1 with buffer size constraints $b(AB) = 4$ and $b(BC) = 2$ can be encoded as the SDF graph in Fig. 2, where two additional back-edges with initial tokens are added to model the buffer size constraints. (If the original SDF graph already contains backedges, some edges may become redundant after adding the backedges, but this procedure is still valid.)

We can now conclude that *an SDF graph* ***with a given buffer size distribution*** *is schedulable if and only if it is schedulable with BGA*. This statement serves as the theoretical foundation for this paper.

We adopt BGA in this paper, but any other SSSA can also be used in its place without affecting correctness. If running BGA on an SDF graph leads to a deadlock due to insufficient buffer sizes on a set of constraining edges, we can increase the buffer sizes on one of these edges to resolve the deadlock and run BGA again. If this run leads to a deadlock again due to insufficient buffer sizes on another set of constraining edges, we can increase the buffer sizes on these edges and run BGA again. This process can be repeated until deadlock is resolved permanently, and a feasible buffer size distribution is found. However, it is not sufficient to arbitrarily choose one of the constraining edges to increase its buffer size, and stop when deadlock is resolved permanently,

[3]Of course, different choices among multiple enabled actors may result in schedules with different quality attributes, i.e., buffer size, runtime overhead, etc.

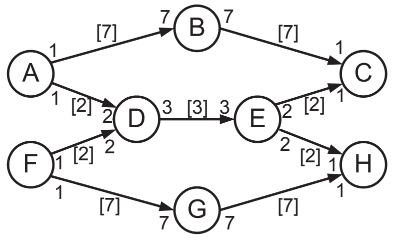

Figure 3: An example SDF graph. Numbers in square brackets denote initial edge buffer sizes.

since it may not lead to the optimal solution with minimum buffer size requirement. Instead, we need to exhaustively try to increase the buffer size on each of the constraining edges in turn, each choice forming one branch of the search tree. Following different branches along the search tree can lead to different total buffer size requirements. We call this algorithm *BGA with Buffer Increase*. We use an example to illustrate this point.

Figure 3 shows an SDF graph from [7] (Ex.9 in Table 3). The repetition vector is ($14a$, $2b$, $14c$, $7d$, $7e$, $14f$, $2g$, $14h$) (with slight abuse of notation). All edges are initially empty. Suppose we assign buffer size on each edge as $(7, 7, 2, 2, 3, 2, 2, 7, 7)$ in the edge order of (ab, bc, ad, ec, de, fd, eh, fg, gh), based on the minimum buffer size Equation (7) in Section 4.1. Fig. 4 shows the partial search tree. After the initial schedule $(2a, 2f,\ d, e,\ 2a, 2f,\ d, 2a, 2f)$, a deadlock state is encountered with token distribution $(6, 0, 2,\ 2, 3, 2,\ 2, 6, 0)$. The constraining edges are $(ad, fd,\ de,\ ec, eh)$. We can choose to increase buffer size on either one of these 5 edges to resolve the deadlock, either temporarily or permanently:

- Increase buffer size on edge ad by 1 to follow the leftmost search tree branch. Then after schedule $(a, b, 2c)$, another deadlock state is encountered with token distribution $(0, 5, 3, 0, 3, 2, 2, 6, 0)$, with constraining edges (ad, fd, de, eh). We then continue to go down the search tree to increase buffer size on one of these edges, but subsequent steps are omitted in the figure.

- Increase buffer size on edge fd by 1. This is similar to the previous case.

- Increase buffer size on edge de by 3. The deadlock is resolved permanently, and we have obtained a feasible buffer size distribution of $(7, 7, 2, 2, 6, 2, 2, 7, 7)$ with total buffer size 42.

- Increase buffer size on edge ec by 2. The deadlock is not resolved, with constraining edges (ad, fd, de, eh). We can continue the search tree to increase buffer size on one of these edges. Suppose we choose to increase buffer size on edge eh by 2. Now the deadlock is resolved permanently, and we have obtained a feasible buffer size distribution of $(7, 7, 2, 2, 3, 4, 4, 7, 7)$ with total buffer size 43.

- Increase buffer size on edge eh by 2. This is similar to the previous case.

This example shows that increasing buffer size on the constraining edge de by 3 to resolve the first deadlock leads to a

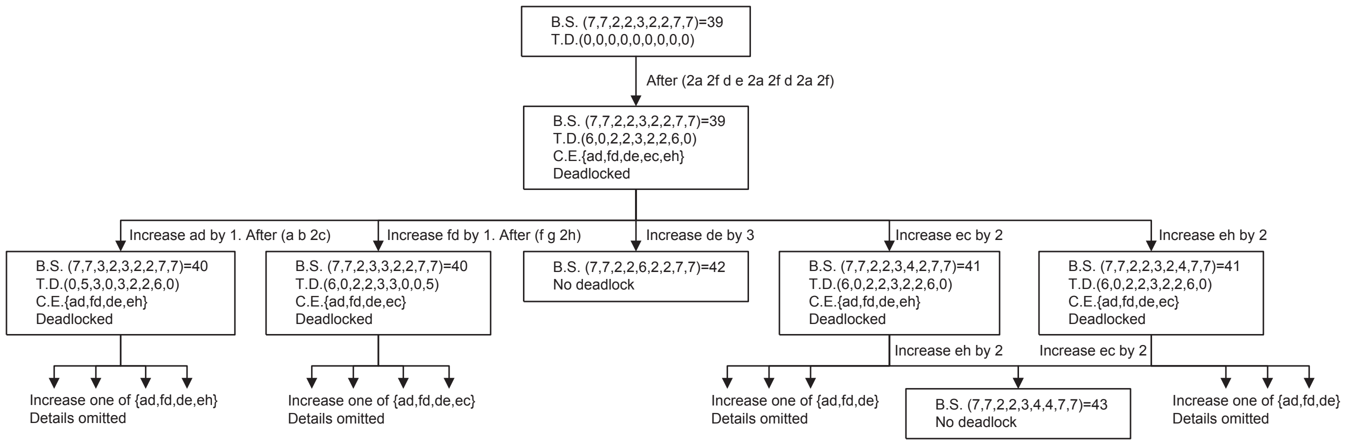

Figure 4: Partial search tree for the minimum buffer size distribution for the example in Fig. 3. B.S: Buffer Size; T.D.: Token Distribution; C.E.: Constraining Edges. B.S. and T.D. are in the edge order of (*ab*, *bc*, *ad*, *ec*, *de*, *fd*, *eh*, *fg*, *gh*).

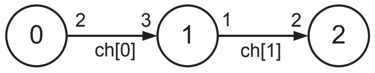

Figure 5: The same SDF graph as in Fig. 1, with different actor and edge labels to be consistent with the SPIN model.

feasible buffer size requirement of 42, while increasing buffer size on the constraining edge *ec* or *eh* by 2 to resolve the first deadlock leads to a feasible buffer size requirement of 43. Therefore, we need to explore both branches of the search tree to find the optimal solution of 42.

4. THE SPIN MODEL

The SPIN model implements *BGA with Buffer Increase* as discussed in the previous section. Table 1 shows the SPIN model for the SDF graph in Fig. 5, which is the same SDF graph as Fig. 1, but with different actor and edge labels to be consistent with the SPIN model. Table 2 explains the key notations in the model. Lines 27-30, lines 31-34, lines 35-38 describe firing rules of actors 0, 1 and 2, respectively. Take actor 1 for example:

1. Line 31: calculate `np[1]` (`nc[1]`), the maximum allowed number of firings for actor 1 without buffer overflow on its output edge 1 (buffer underflow on its input edge 0).
2. Line 32: obtain the loop count as the minimum value between `nf[1]`, `np[1]` and `nc[0]`. It is the maximum allowed number of firings for actor 1 without exceeding its repetition count, or causing any buffer overflow or underflow.
3. Lines 33-34: fire actor 1 and update its input/output edge buffers (`ch[0]` and `ch[1]`), remaining allowed number of firings (`nf[1]`), and the current schedule length (`sl`).

After one iteration of BGA (Lines 27-38), if there is no progress, i.e., the schedule length before this iteration `prev_sl` is the same as the schedule length after it `sl` (Line 40), this

Table 2: Key notations used in the SPIN model.

`sz[i]`	Buffer size of edge *i*
`ch[i]`	Current number of tokens on edge *i*
`nf[j]`	Maximum allowed number of firings of actor *j* before its repetition count is reached
`np[i]`	Maximum allowed number of firings of the producer actor of edge *i* without buffer overflow on edge *i*
`nc[i]`	Maximum allowed number of firings of the consumer actor of edge *i* without buffer underflow on edge *i*
`sl`	Current schedule length (sum of all actor firing counts so far)
`prev_sl`	Schedule length (in the previous iteration of the BGA)
`ESL`	Expected schedule length (sum of repetition counts of all actors)
`lc`	Loop count of an actor
`ACTOR`	Number of actors
`EDGE`	Number of edges

implies that none of the actors could fire during the BGA, and the system has run into a deadlock. Recall that Edge *i* is a constraining edge in a deadlocked state if `np[i] == 0`, i.e., its producer actor cannot fire due to insufficient buffer space on edge *i*, so the buffer size on edge *i* is increased to the size that allows *exactly one* additional firing of its producer actor. The new buffer size is obtained by summing up the current number of tokens on the edge and the production rate of its producer actor (See Line 6 for the definition of the macro `INCREASE`). In case there are multiple constraining edges, one is chosen nondeterministically to increase its buffer size, each choice forming one branch of the search tree. For clarification, the general form of buffer size increase is as follows:

```
if ::IsConstraining(ch[0]) -> IncreaseBufferSize(ch[0],p[0]);
   ::IsConstraining(ch[1]) -> IncreaseBufferSize(ch[1],p[1]);
   :: ...
   ::IsConstraining(ch[EDGE-1]) ->
            IncreaseBufferSize(ch[EDGE-1],p[EDGE-1]);
fi;
```

During model-checking, SPIN will exhaustively explore all candidate edges to increase each one's buffer size due to the

Table 1: SPIN model for the example in Fig. 5 (Line 42 is optional, as explained in Section 4.1).

```
01 #define sum sz[0]+sz[1]

02 #define MAXPROD(i,p)  np[i] = (sz[i]-ch[i])/p
03 #define MAXCONS(i,c)  nc[i] = ch[i]/c
04 #define PRODUCE(i,p)  ch[i] = ch[i]+p*lc
05 #define CONSUME(i,c)  ch[i] = ch[i]-c*lc
06 #define INCREASE(i,p) sz[i] = ch[i]+p
07 #define COUNT(j)      nf[j] = nf[j]-lc; sl = sl+lc
08 #define MIN2(x,y)     ((x<y) -> x:y)
09 #define MIN3(x,y,z)   ((x<y) -> MIN2(x,z):MIN2(y,z))
10 #define ACTOR         3
11 #define EDGE          2
12 #define ESL           6

13 int sl, lc, nf[ACTOR];
14 int ch[EDGE], sz[EDGE], np[EDGE], nc[EDGE];

15 init {
16   atomic {
17     sz[0] = 4; sz[1] = 2;
18     ch[0] = 0; ch[1] = 0; sl = 0;
19     nf[0] = 3; nf[1] = 2; nf[2] = 1;
20     run BGA();
21   }
22 }
```

```
23 proctype BGA() {
24   int prev_sl;
25   do :: atomic {
26     prev_sl = sl;

27     MAXPROD(0,2);
28     lc = MIN2(nf[0], np[0]);
29     if :: lc>0 -> PRODUCE(0,2); COUNT(0);
30        :: else fi;
31     MAXPROD(1,1); MAXCONS(0,3);
32     lc = MIN3(nf[1], np[1], nc[0]);
33     if :: lc>0 -> PRODUCE(1,1); CONSUME(0,3); COUNT(1);
34        :: else fi;
35     MAXCONS(1,2);
36     lc = MIN2(nf[2], nc[1]);
37     if :: lc>0 -> CONSUME(1,2); COUNT(2);
38        :: else fi;

39    if :: sl == ESL -> break :: else fi;
40    if :: prev_sl == sl ->
41         if :: np[0] == 0 -> INCREASE(0,2);
42            /*:: np[1] == 0 -> INCREASE(1,1);*/
43         fi;
44       :: else fi;
45  } od;}
```

nondeterminstic choice construct above, instead of stopping at a feasible but suboptimal solution. This guarantees optimality of the final solution. (The "atomic" keyword in the SPIN model is used to preserve atomicity of the operations, which prevents expansions of useless intermediate states and improves the search efficiency.)

The following LTL formula is used to check the minimal buffer requirement:

```
<> (sum > BOUND)
```

This means that we challenge the model-checker SPIN to prove that "every schedule will eventually require total buffer size strictly larger than a user-specified `BOUND`." If this is proven true, then `BOUND` is a safe lower bound on the buffer size requirement, but it may not be a tight bound, so we increment the value of `BOUND` and invoke SPIN again; If this is proven to be false, then SPIN has found a schedule with total buffer size `<=BOUND` as a counter example, so we decrement the value of `BOUND` and invoke SPIN again. The minimum buffer size requirement is the value `BOUND` such that the LTL formula is true for `BOUND-1` but false for `BOUND`. Binary search can be used to narrow down the range of the minimum buffer size requirement.

The model encoding in [7, 9] does not assume any knowledge of edge buffer sizes, and does not impose any deterministic firing rule. It keeps track of the maximum number of tokens on each edge, and all possible interleavings among actor firings are explored to check if the specified `BOUND` is feasible. In contrast, we assume a known initial buffer size distribution, and use the deterministic firing rule BGA to detect deadlocks: actors fire in a fixed order (the order is actors (0, 1, 2) in Table 1); each actor fires for its loop count obtained with Equation (1) in each iteration of BGA until either the Expected Schedule Length (ESL) is reached, so the current given buffer size distribution is feasible; or the system runs into deadlock and one of the constraining edges is nondeterministically chosen to increase its buffer size (BGA). The edge buffers are increased until either a feasible solution with total buffer size `sum <= BOUND` is found (the LTL formula `<> (sum > BOUND)` is proven false); or the total buffer size has exceeded `BOUND` without finding a feasible solution. In our approach, the search space consists of all possible ways of choosing one among multiple *constraining edges* to increase its buffer size to allow one additional firing of the producer actor; in [7, 9], the search space consists of all possible schedules (interleavings of actor firings), and the job of the model-checker is to find one with total buffer size `sum <= BOUND`.

While different encodings of the SDF buffer minimization problem do not change the NP-completeness nature of the problem, we offer the following explanation for why our approach is much more efficient than [7, 9]: It is a NP-complete problem to check if the SDF graph is schedulable if only the *total buffer size bound* is known: the deterministic firing rule BGA cannot be used without the knowledge of the buffer size distribution, but there are an exponential number of buffer size distributions with the same total buffer size, as in [7, 9]. On the other hand, it is very easy to check if the SDF graph is schedulable if the exact *buffer size distribution* is known: BGA or any other SSSA can be used to do this with number of steps not exceeding the Expected Schedule Length, as in our approach. However, this does not help us much if it is necessary to exhaustively explore all possible buffer size distributions with the same total buffer size. Instead of exhaustive exploration, we only explore a small subset of possible buffer size distributions by selectively increasing buffer sizes only on the *constraining edges* at a deadlock state to allow one additional firing of the producer actor. This reduces the search space drastically. It is somewhat similar to the approach of *Directed Model-Checking* [6], where domain-specific knowledge is used to guide the model-checker towards promising regions of the state space by pruning away many uninteresting branches of the search tree. (In fact, we can also exploit domain-specific knowledge to optimize the model encoding in [7, 9] by limiting the schedule length to not exceed the Expected Schedule Length.)

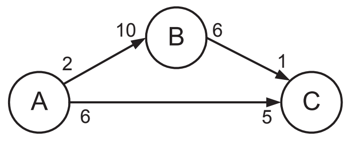

Figure 6: An SDF graph example to show that $bslb(e)$ may not be a tight bound.

4.1 Setting Initial Edge Buffer Sizes

Now we answer the question of how to set the initial edge buffer sizes (`sz[i]` in the `init` process in the SPIN model). The initial buffer size on each edge in the SPIN model should be safe, i.e., it is guaranteed to be less than or equal to the edge buffer size in the optimal solution. At the same time, it should be as large as possible to reduce the search space.

For any edge e with token production (consumption) rate of $p(e)$ ($c(e)$), and $d(e)$ number of initial tokens, the Buffer Size Upper Bound $bsub(e)$ is set to be large enough to contain the number of tokens produced by its producer actor $src(e)$ if it fires consecutively for its repetition count plus the number of initial delays:

$$bsub(e) = p(e) * q(src(e)) + d(e), \tag{6}$$

where $q(v)$ is the repetition count of actor v.

The Buffer Size Lower Bound on edge e for all possible valid schedules is given as [4]:

$$bslb(e) = \begin{cases} p(e) + c(e) - g(e) + d(e) \bmod g(e), \\ \qquad\qquad \text{if } d(e) < p(e) + c(e) - g(e) \\ d(e), \quad \text{otherwise,} \end{cases} \tag{7}$$

where $g(e) = gcd(p(e), c(e))$.This bound is determined solely based on the token production/consumption rates and initial buffer size on the edge e, and independent of the rest of the SDF graph. Hence it may not be a very tight bound. As an example, consider the SDF graph in Fig. 6. The minimum buffer size of edge AC obtained from Equation (7) is $bslb(AC) = 6 + 5 - 1 = 10$. However, we can derive a larger minimum buffer size of 30 as follows: Actor C can fire only after both actors A and B have fired at least once to produce enough tokens on edges AC and BC. Actor B can fire only after actor A has fired 5 times to produce 10 tokens on edge AB. This implies that buffer size on edge AC must be at least 30 to accommodate the 30 tokens produced by 5 firings of A. Obviously, using the larger buffer size of 30 as the initial buffer size helps to reduce search tree depth and search space compared to the smaller buffer size of 10.

If the buffer size lower and upper bounds of an edge are equal, then its buffer size is fixed and never needs to be changed. We can omit it in the nondeterministic choice for buffer size increase in the SPIN model in order to improve model-checking efficiency. For example, Line 42 in Table 1 can be removed, since buffer size lower and upper bounds for edge 1 are both 2. It turns out that this situation is quite common for many practical applications.

Next, we present the TLB (Tight Lower Bound) Algorithm for obtaining a tighter (larger) initial buffer size on an edge e than using the lower bound Equation (7) as in [7, 9]. The initial buffer size on edge e is set to be the lower bound value obtained with Equation (7), and buffer sizes on all other edges are set to infinity, i.e., there is no buffer size constraints on these edges. Run BGA in Alg. 1 to detect potential deadlocks. If deadlock is detected, then increase the buffer size on edge e to allow one more firing of its producer actor. Keep doing this until there is no more deadlock. Alternatively, we can use binary search between the lower and upper bounds on the buffer size to narrow down the range and eventually find the correct buffer size bound. This algorithm is run for every edge, then the obtained buffer size bounds are used as the initial buffer sizes in the SPIN model. The TLB algorithm is orthogonal to our SPIN model, and can also be incorporated into the models in [7, 9] to improve their efficiency.

Algorithm 2 Tight Lower Bound Algorithm

```
Require: SDF graph $G$
 1: for each edge $e$ in $G$ do
 2:    $tlb(e) = \infty$, $sz(e) = \infty$
 3: end for
 4: for each edge $e$ in $G$ do
 5:    for each candidate $b(e) \in CAND(e)$ by binary search
       do
 6:       $schedulable = BGA(G)$
 7:       if $schedulable = true$ and $b(e) < tlb(e)$ then
 8:          $tlb(e) = b(e)$
 9:       end if
10:    end for
11:    $sz(e) = \infty$
12: end for
13: return $tlb(e)$ for every edge $e$
```

The set of candidate buffer sizes $CAND(e)$ include the series of buffer sizes between the lower bound $bslb(e)$ and upper bound $bsub(e)$ with increments of step size equal to the greatest common divisor of its token production and consumption rates $gcd(p(e), c(e))$.

$$\begin{aligned} CAND(e) = \{&bslb(e) + i * gcd(p(e), c(e)) \mid \\ &0 \le i \le (bsub(e) - bslb(e))/gcd(p(e), c(e))\}. \end{aligned} \tag{8}$$

It can be easily shown that limiting the search space to the set of candidate edges in $CAND(e)$ will not lead to loss of any optimal solution: the number of tokens on edge e after its producer actor fires for x times and its consumer actor fires for y times is $x*p(e)-y*c(e)+d(e)$, which can be written as $i * gcd(p(e), c(e)) + d(e)$. Taking into account the lower and upper bound constraints, we can derive that the set of buffer sizes $CAND(e)$ as defined in Equation (8) is identical to the set of all possible legal numbers of tokens on edge e during execution of any schedule with length $\le ESL$, hence we can safely limit the search space to the set $CAND(e)$.

Alg. 2 shows the pseudo-code for the TLB algorithm. All edge buffer sizes are initially set to ∞. At Line 5, a candidate buffer size for edge e is chosen from the set $CAND(e)$, using binary search between $bslb(e)$ and $bsub(e)$ with step size $gcd(p(e), c(e))$ until the minimum feasible buffer size is found. At Line 6, the BGA is invoked to check if the SDF graph is schedulable for the given buffer size distribution. It is a polynomial-time algorithm that runs very fast in practice.

5. PERFORMANCE EVALUATION

We use both real-world DSP and multimedia applications and synthetic SDF graphs for performance comparison with

Table 3: SDF examples used in the experiments. *Exp.S.L.* stands for Expected Schedule Length; *B.S.* stands for Buffer Size.

Ex	Model	No. Actors	No. Edges	Exp. S.L.	Opt. B.S.
1	Sample rate ([7])	6	5	612	32
2	Modem ([7])	16	19	48	38
3	Carrier-sync ([11])	8	8	17	20
4	H.263 ([12])	7	7	203	595
5	H.264 ([13])	11	12	403	991
6	Mp3 ([2])	9	11	102	4645
7	Inmarsat ([8])	22	26	4515	1542
8	Fig.7 in [7]	6	8	45	83
9	Fig.8 in [7]	8	9	74	42
10	Synthetic1 [1]	11	16	705	166
11	Synthetic2 [1]	15	22	127	516

the techniques in [7] and [9], using the model-checker SPIN and NuSMV, respectively. Table 3 shows the details of the examples used. The complexity of model-checking for an SDF graph is not only dependent on the number of actors and edges in the graph, but also highly dependent on the topology and token production/consumption rates. The examples include Sample rate converter, Modem, Carrier-synchronizer, H.263/H.264 decoder, MP3 decoder, Inmarsat satellite receiver, Fig.7 and Fig.8 in [7]. We also manually constructed two larger synthetic examples (*Ex.10* & *Ex.11*, available online at [1]) to demonstrate the efficiency of our approach. *Ex.10* is obtained by concatenating two copies of *Ex.8*, and *Ex.11* is an artificial one. Since all three approaches produce optimal results, we focus on the comparison of time and memory usage of the model-checker. The experiments are conducted on a workstation running Fedora Linux Core with 4 $\times$ AMD Opteron 844 (1.8GHz) CPU and 8GB RAM.

In Table 4, "BGA" stands for our technique presented in this paper; "Geilen" stands for the technique in [7][4]; and "Gu" stands for the technique proposed in [9]. Rows labeled "feas." contain results for model checking sessions when the correct `BOUND` is tested in the LTL formula `<> (sum > BOUND)`, which is proven false, and a feasible schedule is returned as the counter example. Rows labeled "infeas." contain results when `BOUND-1` is tested in the LTL formula `<> (sum > BOUND)`, which is proven true. "OOM" and "OOT" stand for "Out of Memory (system memory usage exceeds 8GB) and "Out of Time" (algorithm running time exceeds a pre-defined threshold of 5 hours), respectively. We can see that the BGA technique consistently outperforms the other two techniques. Our BGA technique finished model-checking sessions for all the test cases within a maximum of 7.75 seconds and 64.1 MB memory usage, while Geilen's technique experienced OOM for 2 out of the 11 examples, and Gu's technique experienced OOT for 4 examples, including H.264 decoder, Inmarsat satellite receiver and the two synthetic examples[5]. Except these cases of OOM and OOT, Geilen's technique uses up to 80.9 seconds and 774.2 MB memory, and Gu's technique uses up to 9608.7 seconds and 325.4 MB memory.

Table 4: Performance comparison of our approach with those by Geilen [7] and Gu [9].

Ex	Cond	Time Usage (s)			Memory Usage (MB)		
		BGA	Geilen	Gu	BGA	Geilen	Gu
1	feas.	0.01	0.01	0.28	3.6	8.7	26.0
	infeas.	0	1.38	0.01	3.6	23.5	2.5
2	feas.	0	10.1	1.34	3.6	115.9	27.5
	infeas.	0.1	80.9	0.7	3.6	744.2	24.0
3	feas.	0.1	0.06	0.01	3.6	9.3	0.3
	infeas.	0	0.04	0	3.6	3.6	11.3
4	feas.	0	3.32	0.2	3.6	44.3	19.4
	infeas.	0.1	7.83	0.04	3.6	82.1	10.2
5	feas.	0.02	0.03	OOT	3.6	9.3	OOT
	infeas.	0.99	25.7	216.0	10.4	252.0	325.4
6	feas.	0	0	6.21	3.6	8.5	81.9
	infeas.	0.01	15.7	5.7	3.6	149.5	65.4
7	feas.	0.03	0.39	9608.7	3.6	29.8	321.8
	infeas.	0.01	6.42	OOT	3.6	72.1	OOT
8	feas.	0	0.01	6.65	3.6	3.6	29.2
	infeas.	0.01	0.07	6.33	3.6	3.6	28.7
9	feas.	0.1	2.16	0.16	3.6	34.5	23.2
	infeas.	0	1.7	0.06	3.6	28.7	2.5
10	feas.	6.0	OOM	OOT	50.2	OOM	OOT
	infeas.	7.75	OOM	OOT	64.1	OOM	OOT
11	feas.	0.01	OOM	OOT	3.6	OOM	OOT
	infeas.	0	OOM	OOT	3.6	OOM	OOT

6. CONCLUSIONS

In this paper, we have proposed an exact technique for buffer memory minimization of SDF graphs based on model-checking. Our model encoding exploits domain-specific knowledge of SDF graphs for effective pruning and reduction of the search space. Performance evaluation shows significant improvements of efficiency and scalability over existing techniques based on model-checking.

7. REFERENCES

[1] `http://www.cse.ust.hk/~weichen/sdf_ex.html`.

[2] `http://www.es.ele.tue.nl/sadf/`.

[3] M. Ade, R. Lauwereins, and J. A. Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. In *Design Automation Conference*, pages 64–69, 1997.

[4] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *International Conference on Computer-Aided Verification (CAV)*, 2002.

[4]The initial buffer size on each edge e is set to be $bslb(e)$ as computed with Equation (7), which is shown to help improve performance compared to setting it to 0 [7].

[5]Since NuSMV uses symbolic encoding of the state space, its memory usage does not necessarily grow with state space size, hence it may experience OOT before OOM; Since SPIN uses explicit-state encoding of the state space, its memory usage grows with state space size, hence it may experience OOM before OOT.

[6] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with hsf-spin. In M. B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 57–79. Springer, 2001.

[7] M. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 819–824, New York, NY, USA, 2005. ACM Press.

[8] S. Goddard and K. Jeffay. Managing memory requirements in the synthesis of real-time systems from processing graphs. In *Real-Time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE*, pages 59–70, Jun 1998.

[9] Z. Gu, M. Yuan, N. Guan, M. Lv, X. He, Q. Deng, and G. Yu. Static scheduling and software synthesis for dataflow graphs with symbolic model-checking. In *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 353–364, Washington, DC, USA, 2007. IEEE Computer Society.

[10] G. J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[11] J. Horstmannshoff and H. Meyr. Optimized system synthesis of complex rt level building blocks from multirate dataflow graphs. In *ISSS*, pages 38–43, 1999.

[12] D. Kim, M. Kim, and S. Ha. A case study of system level specification and software synthesis of multimode multimedia terminal. In *ESTImedia*, pages 57–64, 2003.

[13] S. Kwon, H. Jung, and S. Ha. H.264 decoder algorithm specification and simulation in simulink and peace. In *International SoC Design Conference*, pages 9–12, Oct 2004.

[14] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[15] P. K. Murthy and S. S. Bhattacharyya. *Memory Management for Synthesis of DSP Software*. CRC Press, 2006.

[16] H. Oh, N. Dutt, and S. Ha. Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 497–502, New York, NY, USA, 2006. ACM Press.

[17] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, 1994.

[18] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 899–904, 24-28 July 2006.