Static Dataflow with Access Patterns: Semantics and Analysis

Arkadeb Ghosal*, Rhishikesh Limaye*, Kaushik Ravindran*, Stavros Tripakis**
Ankita Prasad*, Guoqiang Wang*, Trung N Tran*, Hugo Andrade*
* National Instruments Corp., Berkeley, CA, USA, {firstname.lastname}@ni.com
** University of California, Berkeley, CA, USA, stavros@eecs.berkeley.edu

ABSTRACT

Signal processing and multimedia applications are commonly modeled using Static/Cyclo-Static Dataflow (SDF/CSDF) models. SDF/CSDF explicitly specifies how much data is produced and consumed per firing during computation. This results in strong compile-time analyzability of many useful execution properties such as deadlock absence, channel boundedness, and throughput. However, SDF/CSDF is limited in its ability to capture how data is accessed in time. Hence, using these models often leads to implementations that are sub-optimal (i.e., use more resources than necessary) or even incorrect (i.e., use insufficient resources).

In this work, we advance a new model called Static Dataflow with Access Patterns (SDF-AP) that captures the timing of data accesses (for both production and consumption). This paper formalizes the semantics of SDF-AP, defines key properties governing model execution, and discusses algorithms to check these properties under correctness and resource constraints. Results are presented to evaluate these analysis algorithms on practical applications modeled by SDF-AP.

Categories and Subject Descriptors: C.3 [Special-purpose and Application-based Systems]: Signal processing systems

General Terms: Theory, Algorithms, Experimentation

Keywords: Dataflow, semantics, access patterns

1. INTRODUCTION

Static Dataflow (SDF) is a model of computation to specify, analyze, and implement multi-rate computations that operate on infinite streams of data [13]. An SDF model is represented as a directed graph of computational actors interconnected by FIFO channels. The SDF semantics requires that the number of tokens consumed and produced by an actor per firing is fixed and pre-specified. This guarantees decidability of key model properties: existence of deadlock-free and memory-bounded infinite computation, throughput, latency, and execution schedule [1, 13]. The expressiveness of the SDF model in naturally capturing streaming applications, coupled with its strong compile-time predicitability properties, has made it popular in the domains of multimedia, digital signal processing, and communications.

While the standard SDF model is untimed, it is a common practice to associate worst-case execution time (WCET) models to analyze the timing behavior of applications [7, 12, 14, 15, 20]. These timing annotations enable static analysis of SDF models and mapping solutions to specific platforms under resource and performance constraints. Worst-case timing models have been applied to capture execution behavior of SDF actors for software and hardware implementations.

However, these timing models suffer a key deficiency: they lose information about the precise timing of consumption and production of tokens by an actor during a firing cycle. The problem is particularly evident when SDF models are used to capture hardware implementations. Many hardware IP blocks require that data tokens be delivered to them at precisely specified clock cycles from the start of execution. This loss of timing information in SDF models results in sub-optimal analysis and implementations that conservatively estimate the resources needed.

For example, consider a design connecting a producer P to a consumer C. P produces 1 token per firing and executes in 1 time unit, and C consumes 8 tokens per firing and executes in 8 clock cycles. Suppose that the IP block implementing C requires 8 tokens to be delivered in 8 consecutive cycles. Unfortunately, the SDF timing model is not sufficiently expressive to capture this behavior. The semantics of SDF assumes that an actor cannot start firing until sufficient tokens are present at the inputs. As a result, if the above example is modeled with SDF, C cannot start firing until after C completes eight firings. Therefore, a buffer of size at least 8 must be added between P and C; C may start its execution only after the buffer has collected 8 tokens from P. While this is a valid implementation, it is sub-optimal in terms of allocation of buffer resources. In contrast, a better implementation can exploit knowledge about the behavior of C and determine that a buffer of size one is sufficient.

Cyclo-Static Dataflow (CSDF) [2] is a generalization of SDF that appears to resolve the problem. CSDF “breaks” a firing into finer-grained phases, and specifies consumptions and productions of tokens for each phase. But CSDF still relies on the same basic hypothesis as SDF, i.e., that an actor will wait until sufficient tokens have accumulated at the input channels before beginning a phase. Unfortunately, this hypothesis violates requirements related to the precise timing of token accesses. In the example above, C requires that it receive 8 tokens in 8 consecutive clock cycles once it commences firing. CSDF cannot capture this constraint and as a result can lead to incorrect implementations [19].

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2012, June 3-7 2012, San Francisco, California, USA
Copyright 2012 ACM 978-1-4503-11199-1/12/06 ...$10.00.
For example, consider an alternate producer \( P \) with an execution time of 2. Then a CSDF model would conclude that a buffer of size 1 between \( P \) and \( C \) is sufficient, but this would violate the timing requirement of \( C \).

It may be argued that the requirement of precise timing of tokens is artificial, since actors can be stalled or turned off. Actors implemented in software can easily be disabled or context switched. For hardware IP blocks, there is typically a “clock enable” signal that regulates their execution. Setting this signal to “false” freezes the actor when inputs are unavailable. However, this solution is not satisfactory in practical designs. The area overhead due to the enable logic is undesirable. Also, any logic that regulates the clock contributes additional delay to timing-critical paths. The increased distribution of “clock enable” signals further adversely impacts the achievable frequency. Hence, it is important to capture precise timing of token accesses to generate resource optimal implementations. Both SDF and CSDF models are not equipped for this.

To remedy the expressiveness problems of SDF/CSDF, a new model, called \( SDF \) with Access Patterns (SDF-AP), is introduced informally in [19]. SDF-AP strikes a balance between the analyzability of SDF/CSDF while accurately capturing the interface timing behavior. The latter is achieved by specifying access patterns that capture the precise timing behavior of token productions and consumptions. The original motivation for SDF-AP comes from modeling hardware IP blocks, where access patterns are precisely characterized and presented as timing diagrams. Nevertheless, the timing extensions that access patterns provide are general and applicable to actors implemented in software as well.

The goal of [19] is to justify that choosing the right model is important for generating correct and non-defensive implementations from high level component abstractions. It informally introduces the SDF-AP model and advocates a general methodology based on Finite State Machines to reason about performance and resource trade-offs. However, [19] does not define the semantics of the SDF-AP model. It also does not develop analysis methods to reason about model properties. This paper closes this gap. Our main contributions are: (a) a formal definition of the SDF-AP model with its operational semantics, (b) formal definitions of key model properties, such as executability and throughput, (c) algorithms for efficient static analysis of these properties, and (d) case studies to evaluate these algorithms.

### 2. RELATED WORK

Real-time streaming applications are widely deployed on embedded platforms. Model-based design is a well-tested approach for the implementation of these systems. A comprehensive survey on concurrent models of computation can be found in [11]. Prior research has shown that dataflow and its variants are sufficiently expressive enough to capture the task and data parallelism in streaming applications. SDF and CSDF models enable compile time analysis of key execution properties, e.g., absence of deadlocks and consistency of execution rates, via efficient algorithms [1,12,13]. Recent variants like Heterochronous Dataflow (HDF) [6], Scenario Aware Dataflow (SADF) [18], and Core Functional Dataflow (CFDF) [7] extend SDF/CSDF with specifications for control. Design frameworks like Ptolemy-II [5], SDF.NET [17], and OpenDF [8] deliver hardware and software implementations.

Though SDF/CSDF models have many advantages, they are limited in their ability to capture precise timing information of data production and consumption. This is particularly evident when dataflow models are targeted for hardware implementations. Prior efforts are conservative in their implementation of the glue logic to stitch SDF actors in hardware [4,8–10]. SDF-AP is introduced in [19] to remedy that deficiency. Model properties like consistency, absence of deadlock, bounded execution, and throughput need to be checked before the model can be implemented. There are existing techniques to check the properties for SDF/CSDF models. However, they cannot be directly used for SDF-AP models due to differences in semantics. In this paper, we present the formal semantics of SDF-AP models and algorithms to efficiently check key model properties.

### 3. SDF-AP: SYNTAX AND SEMANTICS

An SDF-AP model consists of actors connected over channels. Actors read tokens from incoming channels and write to outgoing channels. Once an actor has fired, it consumes (resp. produces) a fixed number of tokens from (resp. to) input (resp. output) channels over the execution time. An actor associates each channel with a pattern represented as a binary word of length equal to the execution time of the actor. The pattern denotes whether the actor reads (resp. writes) a token or not from (resp. to) the incoming (resp. outgoing) channel at a particular cycle in the execution. The access pattern can be provided by the user or derived from the timing diagrams accompanying the documentation of the IP block [19]. Given the application domain and hardware implementation, we will restrict reading and writing of at most one token per channel at any clock transition. Nevertheless, the model semantics can be easily generalized to allow multiple tokens to be read or written.

Fig. 1 shows an SDF-AP model with a build stream actor, \( bs \), which takes two input streams and merges them in one. Actor \( bs \) is fed by two source actors \( i1 \) and \( i2 \): \( i1 \) generates 1 token every 2 cycles, and \( i2 \) generates 3 tokens every 4 cycles. At each firing, \( bs \) consumes 2 tokens produced by \( i1 \), and 4 tokens produced by \( i2 \), and places them in a merged stream of 6 tokens at the output (tokens from \( i1 \) preceding those of \( i2 \)). Actor \( bs \) is connected to a sink actor \( o \). The net token count and respective pattern are shown separated by “;”, e.g., “3:1101” on \( c2 \) denotes that \( i2 \) produces 2 tokens with the pattern 1101 on channel \( c2 \). Channels \( c1, c2, c3 \) connect \( i1 \) with \( bs, i2 \) with \( bs \), and \( bs \) with \( o \), respectively.

#### 3.1 Syntax

An SDF-AP model is a pair \( M = (aset, cset) \), where \( aset \) is a set of actors, and \( cset \) is a set of channels. For the example in Fig. 1, \( aset = \{ bs, i1, i2, o \} \), and \( cset = \{ c1, c2, c3 \} \).

![Figure 1: SDF-AP model for build stream actor interacting with two sources and one sink](image-url)
at : oc → N] mapping each input (resp. output) channel to the total number of tokens read from (resp. written to) that channel per firing of a, et ∈ N is the time in clock cycles it takes to complete one firing of a, cp (resp. pp) is a map of input (resp. output) channels to consumption (resp. production) patterns. The pattern cp (resp. pp) maps each input (resp. output) channel to a binary word of length et, i.e., cp : ic → 2^et and pp : oc → B^et. The i-th letter of the word is denoted as c(p,c(i)) (resp. p(p,c(i))).

The sum of the 1's in cp(c) (resp. pp(c)) equals the input (resp. output) token count for the channel when (it(c) (resp. ot(c))). For source actors, ic = ∅, it = cp = B^et; for sink actors, oc = ∅, ot = pp = ∅.

A channel c ∈ cset is a unique id, and must appear exactly once in the input channel set of an actor, and exactly once in the output channel set of an actor. This ensures no dangling channels and no non-determinism in channel access.

3.2 Semantics

The operational semantics of an SDF-AP model M = (aset, cset) is defined as a state transition system. A state of the system tracks the number of tokens on each channel, the set of running instances of each actor, and for each instance, the number of clock cycles it has been executing. Formally, a state s is a pair (γ, υ) where γ : cset → Z is a channel quantity[16] (we allow negative values for token counts, see below for the interpretation), and υ : aset → MS(N_0 × [w, r, l]) maps each actor to a multiset of pairs of the form (η, κ) ∈ N_0 × [w, r, l]. If υ(α) = ∅ then actor a has no active (i.e., running) instances currently. Otherwise, each pair (η, κ) ∈ υ(a) represents an active instance of a: η denotes the number of clock cycles the instance has been executing, and κ is a flag denoting the stage the instance is in the current clock cycle. There are three possible stages: beginning of clock cycle ⊥ (idle stage), reading r, and writing w. The meaning will become clear in what follows. 1 A state s is called stable if ∀a ∈ aset. γ(c,c) ∈ υ(a), κ = ⊥. The initial state s_0 = (γ_0, υ_0) where γ_0 ∈ aset, υ_0(a) = ∅, and γ_0 maps each channel to a given number of initial tokens. The initial state (which gets modified with different set of initial tokens) determines the behavior of the model.

Following[16], we define operations on channel quantities. If γ_1, γ_2 are channel maps from sets of channels cset_1, cset_2, with cset_2 ⊆ cset_1, then γ_2 ≤ γ_1 if ∀c ∈ cset_2, cset_2(c) ≤ cset_1(c). The operation γ_1 + γ_2 is defined as pointwise addition. If γ_2 ≤ γ_1, then the operation γ_1 - γ_2 is defined as pointwise subtraction. We will use γ = 0 to denote that token counts on all channels are 0, γ ≥ 0 to denote that all channels map to N_0, and γ ≤ β where β ∈ N_0 to denote that channel counts are bounded by β. For actor a and i ∈ [1 ... et(a)], we define the following channel quantities: γ^R(a) (resp. γ^W(a)) maps every input (resp. output) channel c of a to cp(c,i) (resp. pp(c,i)). For source actors, γ^R(a) = 0 and for sink actors, γ^W(a) = 0, for all i.

A transition δ = (s, l, s') of M from state s = (γ, υ) to state s' = (γ', υ') labeled with label l, also denoted s ⊮ l s', can be any one of those shown in Table 1. s' is called a successor of s. A transition labeled begin(a) adds a new instance of a to the set of active actor instances. The clock counter of the new instance is initialized to 0 and the instance is idle (i.e., not ready to read or write). A transition labeled end(a) removes an instance of a from the set of active instances, provided the instance has finished its firing, i.e., its clock counter has reached et(a). A transition labeled clock marks the beginning of a clock cycle: all active actor instances increase their clock counter by 1 and move from the idle stage ⊥ to the reading stage r. A transition labeled read(a) (resp. write(a)) corresponds to a reading from (resp. writing to) its input (resp. output) channels. Once it has read, an actor instance moves from reading stage r to writing stage w. Once it has written, it moves back to stage ⊥, until the beginning of the next clock cycle.

Note that read transitions may result in channel capacities becoming negative. This is because no precondition on having enough tokens in the channel is imposed for taking a read transition. Similarly, nothing prevents writing, which means that no a-priori bounds on channel size are imposed. This approach makes the semantics easier to formalize. We identify below situations where a negative token count models non-executable vs. transient behaviors as well as distinguish between bounded and unbounded executability.

Also note that reads and writes occur asynchronously between actor instances (i.e., different instances interleave) while for a given instance, the read always occurs before the write. The latter is done to model causality where a consuming actor needs to wait till a producing actor places a token in the channel. A synchronous semantics is also possible, where all actors read simultaneously, then write simultaneously, to complete a clock cycle. The synchronous semantics results in far fewer transitions than the asynchronous semantics. However, the synchronous semantics does not allow to distinguish between non-executable and certain executable models (see discussion on Figure 2 in Section 4).

An execution trace τ is an infinite sequence of transitions τ = s_0 ⊮ l_1 s_1 ⊮ l_2 s_2 ... where s_0 is the initial state. Any subsequence τ' = s_n ⊮ l_{i+1} s_{i+1} ... s_m with for some n ∈ N_0 and i ≤ n is a sub-trace. The set of traces of M is denoted traces(M). The set of states visited along a trace τ is denoted states(τ). Refer to Supplemental Section S1 for traces from the running example. A state s is called reachable from initial state s_0 if s ∈ states(τ) for some trace τ. Note that our semantics guarantees that any readable state s has a successor state s'. State s is a post clock transition (PCT) state if s' ⊮ clock for some state s'. Given a PCT state s, a stable state s' is a next stable state of s, denoted NSS(s), if there exists a sub-trace τ' = s ⊮ l_1 s_2 ... s' (for some n ∈ N) such that none of the labels l_1, l_2, ..., l_n are of the types begin(a), end(a) or clock for all actors a ∈ aset. Our semantics guarantees that for any PCT state s there is a unique next stable state NSS(s) (refer to Supplemental Section S2 for formal reasoning). A PCT state s corresponds to the beginning of a clock cycle, and NSS(s) corresponds to the end of that cycle. If s = (·, υ) is a PCT state where ∀a ∈ aset. υ(a) = ∅, then s is a stable state, and is a next stable state of itself. This corresponds to a situation when no actor has fired, and hence no read transition is enabled.

Given a trace τ, all(τ) is the set of traces generated by
combining τ with all possible sub-traces between all the PCT states of τ and their corresponding NSS states. all(τ) can be seen as a set of traces, but also as a transition system, which is a part of the transition system of the model. The set of states in all(τ) is denoted as states(all(τ)).

SDF-AP actors are auto-concurrent, i.e., multiple instances of an actor can execute simultaneously. However, this may not be feasible in practice due to restrictions like finite resources, IP block properties etc. Such constraints are captured through initiation interval ii ∈ N_{≥0} which specifies the minimum time between two firings of an actor. If ii ≥ ct, then actor execution cannot be concurrent; otherwise actors can execute in parallel. If ii is specified for an actor, then enabling condition of a begin fire transition should check that the state is stable, and ensure that a minimum of ii clock cycles has passed after the latest firing of the actor.

4. MODEL PROPERTIES

Interesting properties for standard SDF/CSDF models are deadlock/livelock-freedom (can the model execute with some/all actors firing infinitely often?), boundedness (can the model execute forever with finite buffers?), etc. In this section we define properties similar in spirit for SDF-AP.

Definition 4.1. A trace τ is live if both begin(a) ∀a ∈ aset and clock appear infinitely often in τ.

The semantics of SDF-AP allows token counts to be negative. Hence, every model has live traces. In reality buffers cannot have negative token count. However, there is an interesting situation where a trace models an implementable behavior, even though the trace visits states with negative token counts. Consider a channel c whose token count becomes −1 between a PCT state s and NSS(s). This implies a situation c is empty and an actor writes to c while another actor reads from c at the same clock cycle. If the read happens before the write (our asynchronous semantics allows that) c will have a (transient) negative token count of −1. This scenario can however be implemented with a fast buffer that allows writing and reading a token in the same cycle. A model that can be executed without a fast buffer is strongly executable, otherwise, it is weakly executable.

Definition 4.2. An SDF-AP model M is weakly executable if there exists a live trace τ ∈ traces(M) such that ∀s = (γ, r) ∈ states(τ), γ ≥ 0. M is strongly executable if there exists a live trace τ ∈ traces(M) such that ∀s = (γ, r) ∈ states(all(τ)), γ ≥ 0.

We distinguish between executability and bounded executability. The former only captures problems of negative token counts (i.e., deadlocks or livelocks in standard SDF/CSDF parlance). Bounded executability is stronger and requires in addition ability to execute with bounded buffers.

Table 1: State Transitions (transition δ = s \rightarrow s', s = (γ, v), s' = (γ', v'))

<table>
<thead>
<tr>
<th>Type</th>
<th>Label</th>
<th>Precondition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin fire</td>
<td>begin(a)</td>
<td>s is stable</td>
<td>v′(a) = v(a) \text{ if } ((i, \bot) \in v(a)) \text{ otherwise, } v′(a') \neq a = v(a')</td>
</tr>
<tr>
<td>end fire</td>
<td>end(a)</td>
<td>s is stable, and (ct(a), \bot) \in v(a)</td>
<td>v′(a) = v(a) \setminus {(ct(a), \bot)}, v′(a') \neq a = v(a')</td>
</tr>
<tr>
<td>clock</td>
<td>clock</td>
<td>s is stable (\tau(\gamma, v, \text{end}(a)))</td>
<td>∀a ∈ aset, if v(a) = 0, then v′(a) = 0 else each ((i, \bot) \in v(a)) is updated to ((i + 1, r) \in v′(a))</td>
</tr>
<tr>
<td>read</td>
<td>read(a)</td>
<td>(i, r) \in v(a)</td>
<td>γ′ = γ − γ_{a'i}, v′(a) = v(a) \setminus {(i, r)} \cup {(i, w)}, v′(a') \neq a = v(a')</td>
</tr>
<tr>
<td>write</td>
<td>write(a)</td>
<td>(i, r) \in v(a)</td>
<td>γ′ = γ + γ_{ai}, v′(a) = v(a) \setminus {(i, w)} \cup {(i, \bot)}, v′(a') \neq a = v(a')</td>
</tr>
</tbody>
</table>

Model \(M = (aset, cset)\), actors \(a, a' \in aset\), and \(w\) and \(\gamma\) denote multiset union and difference.

DEFINITION 4.3. An SDF-AP model M is bounded weakly (resp. strongly) executable if \(\exists \beta \in \mathbb{N}_{≥0}\) and live trace τ such that (1) M is weakly (resp. strongly) executable with respect to τ, and (2) ∀s ∈ states(τ) (resp. states(all(τ))), \(\gamma(s) ≤ \beta\).

Figure 2: Liveness, Executability and Boundedness

Fig. 2(a) is not executable as any trace generated will always have negative token counts on channels. Note that the synchronous semantics would be unable to detect the deadlock, as the channel counts are 0 at all stable states. With an initial token in one of the channels (Fig. 2(b)) the model is strongly executable. Similarly Fig. 2(c) is not executable. However with a token on channel c65, the model (Fig. 2(d)) is weakly executable for the following reason. Consider a cycle when a5 fires (with the token from c65) and a6 is idle; a5 consumes 1 token and produces 1 token, i.e., token count on c65 and c56 are 0 and 1, respectively. In the next cycle a5 can continue executing as it does not need any token, and a6 starts firing by consuming the token from c56; at the end of the cycle, token count on c65 and c56 are 1 and 0, respectively. In the following cycle, a5 consumes 1 token from c65, and produces 1 token on c56. Actor a6 (in the second cycle of its execution) consumes the token and continues execution. Note that this scenario is possible as the token produced by a5 can be consumed by a6 in the same cycle thus making the model weakly executable. Fig. 2(e) is strongly executable as a7 has execution time of 1, and hence can wait indefinitely between firings until required amount of token has been generated in the incoming channel. Fig. 2(b), (d) and (e) are bounded. Fig. 2(f) is strongly executable but not bounded.

We define throughput for bounded executable traces and models. Throughput \(\Gamma(M, a, \tau)\) of an actor a, for a bounded executable trace τ of a model M, is the average rate of firing of the actor a in τ. In a model, one is typically interested in the throughput of certain actors, e.g., certain sources or sinks. We assume that a is such a fixed actor for given model M, and we denote \(\Gamma(M, a, \tau) = \Gamma(M, a, \tau)\) to be the throughput of the model for the trace τ. The optimal throughput of the model is then \(\Gamma(M) = \max_{\tau \in T(M, \tau)}(\{I(M, \tau)\})\), where T is the set of all bounded executable traces of the model.
5. ALGORITHMS FOR STATIC ANALYSIS

We now show that bounded strong executability is decidable by providing algorithms to check the property. As we are concerned with strong executability, synchronous semantics suffice, with the benefit of making analysis efficient.

5.1 Boundedness

A model is bounded if it can be executed without termination using buffers with finite capacity. In SDF models, bounded execution is verified by proving that the model is sample rate consistent [13]. An SDF model is sample rate consistent if there exists a fixed non-zero number of firings for each actor, called the repetitions vector, such that executing these firings reverts the model to its original state.

The concept of sample rate consistency can be applied to check boundedness of SDF-AP models. If the underlying SDF model is sample rate consistent, then there exists a non-zero repetitions vector \( r : \text{aset} \rightarrow \mathbb{N} \) such that the number of tokens produced and consumed on each channel is balanced, i.e., \( \forall c \in \text{aset}, r(a)oc(c) = r(a)it(c) \), where \( a \) and \( a' \) are the producing and consuming actors of \( c \). The repetitions vector provides a recipe for a non-terminating periodic execution of the SDF-AP model in bounded memory.

5.2 Bounded Executability

The concept of bounded executability can be translated to the model being bounded and deadlock free. An SDF model is deadlock free if it can be executed without interruption for one full iteration (in which each actor fires as many times as specified in the repetitions vector). The algorithmic solution is to compute a self timed schedule for one iteration (in which an actor fires as soon as all its input tokens are available) [1].

However, for SDF-AP models, the underlying SDF being deadlock free is a sufficient but not necessary condition. For SDF-AP models, it may be necessary to fire an actor before all tokens are available. Consider the SDF-AP model in Fig. 2(d). The underlying SDF model is deadlock. But in the SDF-AP model, \( a_5 \), which has a consumption access pattern of \([101]\), can begin firing and consume the initial token. Hence, the SDF-AP model is bounded executable though the underlying SDF model is deadlock.

We formalize the problem of checking bounded executability of SDF-AP models. The objective is to determine start times for actor firings that respect data dependence and access patterns. Let \( b \) be the repetitions vector of a bounded SDF-AP model \( M \). An actor \( a \) must produce \( r(a)oc(c) \) tokens on output channel \( c \in oc \) in one iteration. For each token, we associate a firing index \( fp \) and time offset \( ot \) to characterize when it is produced: \( \forall a \in \text{aset}, c \in oc(a), n \in \mathbb{N}, fp(a,c,n) = [n/ot(c)] \), and \( op(a,c,n) = \theta(pp(c), n \ (	ext{mod} \ ot(c)) + 1) \), where \( \theta : \{pp(c)\} \times [1..ot(c)] \rightarrow \mathbb{N}^+ \) is the offset from the start of a firing when a token is produced. E.g., given pattern \( pp = [11001] \) for an actor that produces 3 tokens in 5 cycles, \( \theta(pp,1) = 0, \theta(pp,2) = 1, \theta(pp,3) = 4 \). We similarly characterize the firing index \( fc(a,c,n) \) and offset \( oc(a,c,n) \) at which a token on an input channel is consumed by substituting \( ot(c) \) by \( it(c) \) and \( pp(c) \) by \( cp(c) \) in the prior equations.

We present a constraint system to determine if an SDF-AP model is bounded executable. The variables are the start times of actor firings: \( x(a,i) \in \mathbb{Z}, \forall a \in \text{aset}, \forall i \in \{1..r(a)\} \). The dependencies in start times are encoded as

\[
\forall c \in \text{aset}, \forall n \in \{\gamma_0(c) + 1, \ldots, r(a)oc(c)\},
\]

\[
x(a,fp(a,c,n) - \gamma_0(c)) + op(a,c,n) + 1 \leq x(a',fc(a',c,n)) + oc(a',c,n)
\]

These constraints are all of the form \( x_1 - x_2 \leq k \), where \( x_1 \) and \( x_2 \) are variables and \( k \) is a constant. Such a system of difference constraints can be solved by encoding it as a problem of finding shortest paths in a weighted directed graph [3]. The Bellman-Ford algorithm is applied to solve the shortest path problem. Two outcomes are possible: (a) Bellman-Ford returns the delay of the shortest path to each vertex, or (b) Bellman-Ford detects a negative cycle proving that the constraint system is infeasible. Outcome (a) corresponds to the SDF-AP model being bounded executable. The shortest path delays correspond to valid start times for all actor firings in one iteration. Outcome (b) proves that the SDF-AP model is not bounded executable. Thus, this translation to a well-known graph theoretic problem provides an effective mechanism to check bounded executability.

The number of constraints is equal to the total number of firings in one iteration, which is exponential in the worst case in the number of actors in the model. However, the problem of checking whether an SDF model deadlocks also incurs the same complexity [12]. As our experiments show, this is a feasible solution method for practical SDF-AP models.

5.3 Throughput and Buffer Sizing

We address the problem of checking if a bounded executable SDF-AP model meets a specified throughput \( \Gamma \). The constraint formulation can be extended to solve this problem. Intuitively, the throughput constraint is an upper bound on the time between successive firings of an actor. This can be expressed as a difference equation of the form \( x(a,i+1) - x(a,i) \leq 1/\Gamma \). The constraint system can still be solved as a shortest path problem. The optimal throughput of the model can be computed by a binary search over the range of feasible values for \( \Gamma \).

Further, the constraint formulation can be repeatedly applied to explore buffer sizes for channels to meet a specified throughput. The buffer size of a channel can be encoded as a back edge with initial tokens corresponding to the size \( \Gamma \). The solution approach is a search algorithm in which an outer loop fixes buffer sizes and the constraint formulation is analyzed to check throughput of each configuration. One direction of future work is to find efficient heuristics to guide exploration of buffer sizes for SDF-AP models.

6. EXPERIMENTAL RESULTS

We evaluate the benefits of the SDF-AP model on several streaming applications (see Table 2). The first seven applications are SDF models of realistic FPGA implementations consisting of streaming hardware IP blocks. We compute the access patterns for the IP blocks from the cycle-level timing information in their datasheets. The other applications are benchmarks from the SDF3 [17] analysis tool. For our experiments, we choose different amounts of buffer space for each application, and determine throughput (an average rate of firing of the actors) and latency (total duration of a single iteration of the model) using: (1) traditional SDF analysis using eager, self-timed symbolic simulation [16], and (2) our SDF-AP analysis using difference constraints.
7. CONCLUSIONS
The SDF-AP model aims to strike a balance between the analyzability of SDF-like models while accurately capturing the interface timing behavior by including access patterns. In this paper, we formalize the SDF-AP model, discuss its operational semantics, and define executability and boundedness properties. We also present algorithms to check these properties. The experimental results validate their perfor-

Table 2 summarizes the results. For each model, we spec-
ify the number of actors, channels, firings per iteration, and optimal throughput (assuming unbounded buffers). The throughput is relative to a sink actor which has a repetition count of one. Then for each model, throughput and latency analysis is done by bounding the buffer space. Rest of the columns compare throughput, latency and CPU run-time for the two models. We observe that in all cases SDF-AP models have higher throughput and lower latency. In two cases, the SDF model is deadlocked (denoted by “-”), though the SDF-AP model is not.

While run-time for a majority of examples is better for the SDF-AP model, there are instances for which the run-

time is worse than the underlying SDF model. The SDF-AP analysis uses a novel algorithmic method based on difference constraints, which can be further optimized for better perfor-

Table 2: Throughput and latency analysis for different applications for given buffer spaces

<table>
<thead>
<tr>
<th>Name</th>
<th>#Actors</th>
<th>#Channels</th>
<th>Firings/ Iteration</th>
<th>Optimal Throughput (Hz)</th>
<th>SDF Latency (µsec)</th>
<th>SDF Run-time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFDM Tx 2msps</td>
<td>11, 14</td>
<td>585</td>
<td>833</td>
<td>14210</td>
<td>496</td>
<td>481</td>
</tr>
<tr>
<td>OFDM Tx 5msps</td>
<td>11, 14</td>
<td>585</td>
<td>2083</td>
<td>14210</td>
<td>784</td>
<td>481</td>
</tr>
<tr>
<td>OFDM Tx 25msps</td>
<td>9, 12</td>
<td>6250</td>
<td>6250</td>
<td>13858</td>
<td>6200</td>
<td>481</td>
</tr>
<tr>
<td>OFDM Rx Full</td>
<td>46, 66</td>
<td>1667</td>
<td>1573</td>
<td>57216</td>
<td>1573</td>
<td>697</td>
</tr>
<tr>
<td>OFDM Rx Full</td>
<td>46, 66</td>
<td>1667</td>
<td>1573</td>
<td>57216</td>
<td>1573</td>
<td>697</td>
</tr>
<tr>
<td>OFDM Rx Full</td>
<td>46, 66</td>
<td>1667</td>
<td>1573</td>
<td>57216</td>
<td>1573</td>
<td>697</td>
</tr>
<tr>
<td>OFDM Rx Full</td>
<td>46, 66</td>
<td>1667</td>
<td>1573</td>
<td>57216</td>
<td>1573</td>
<td>697</td>
</tr>
<tr>
<td>ZeroPad 600</td>
<td>3, 4</td>
<td>2</td>
<td>48804</td>
<td>12820</td>
<td>6642</td>
<td>481</td>
</tr>
<tr>
<td>Van de Beek</td>
<td>19, 28</td>
<td>28164</td>
<td>39032</td>
<td>12059</td>
<td>39032</td>
<td>481</td>
</tr>
<tr>
<td>MP3Decoder</td>
<td>14, 18</td>
<td>911</td>
<td>15</td>
<td>9264</td>
<td>27</td>
<td>481</td>
</tr>
<tr>
<td>H263Decoder</td>
<td>4, 3</td>
<td>1783</td>
<td>15</td>
<td>596</td>
<td>15</td>
<td>481</td>
</tr>
<tr>
<td>H263Encoder</td>
<td>5, 5</td>
<td>201</td>
<td>120</td>
<td>299</td>
<td>120</td>
<td>481</td>
</tr>
</tbody>
</table>

While run-time for a majority of examples is better for the SDF-AP model, there are instances for which the run-time is worse than the underlying SDF model. The SDF-AP model aims to strike a balance between the analyzability of SDF-like models while accurately capturing the interface timing behavior by including access patterns. In this paper, we formalize the SDF-AP model, discuss its operational semantics, and define executability and boundedness properties. We also present algorithms to check these properties. The experimental results validate their perfor-

8. REFERENCES
S1: Supplemental Section 1

The SDF-AP model $\mathcal{M}$ of the example introduced in Section 3 is $\mathcal{M} = (a\text{set}, c\text{set})$ where $a\text{set} = \{1, i, 2, bs, o\}$ and $c\text{set} = \{c_1, c_2, c_3\}$. The actors are defined as follows:

- $i_1 = (\emptyset, \{c_1\}, \emptyset, \{c_1 \rightarrow 1\}, 2, \emptyset, \{c_1 \rightarrow 0\})$
- $i_2 = (\emptyset, \emptyset, \{c_2 \rightarrow 3\}, 4, \emptyset, \{c_2 \rightarrow 1101\})$
- $bs = (\{(c_1, c_2), \{c_1 \rightarrow 2, c_2 \rightarrow 4\}, \{c_3 \rightarrow 6\}, 6, \{c_1 \rightarrow 110000, c_2 \rightarrow 100111\}, \{c_3 \rightarrow 111111\})$
- $o = (\{(c_3)\}, \emptyset, \{c_3 \rightarrow 6\}, \emptyset, \{c_3 \rightarrow 111111\})$

In the running example, there are no initial tokens. So the initial state $s_0 = (\gamma_0, v_0)$ where $\gamma_0 = \{(c_1 \rightarrow 0, c_2 \rightarrow 0, c_3 \rightarrow 0\}$, and $v_0 = \{(i_1 \rightarrow \emptyset, i_2 \rightarrow \emptyset, bs \rightarrow \emptyset, o \rightarrow \emptyset\})$. Figure 3 shows a possible sub-trace with one complete firing of actor $i_1$. The start state remains the same as above. For the rest of the states, only the token count for $c_0$ and the instance information for $i_1$ change; hence only those values have been shown. The trace starts with a begin$(i_1)$ transition which adds one instance of actor $i_1$ to the instance information in state $s_1$. A clock transition updates the instance of $i_1$ to read state. $i_1$ being a source actor, no token is consumed, and the read transition between states $s_2$ and $s_3$ does not change token counts. A write transition for $i_1$ from state $s_3$ does not change token count as the output pattern for $i_1$ is 01, and one clock cycle has passed after the begin fire transition. A clock transition from $s_4$ increases the execution time info for running instance of $i_1$ to 2. A read transition from $s_5$ does not change any token count, but a write transition from $s_6$ produces one token on channel $c_1$, and the token count for the channel is updated to 1. An end transition at state $s_7$ removes the running instance of $i_1$. Figure 4 shows a possible sub-trace where both $i_1$ and $i_2$ fires. The start state $s_0$ remains the same. In the remaining states, only the token counts for $c_0$ and $c_1$, and the instances for $i_1$ and $i_2$ change.

S2: Supplemental Section 2

**Lemma 8.1.** Any reachable state has a successor state.

**Proof.** Consider a reachable state $s$ which does not have a successor state. Given that $s$ is reachable, there must be a transition $s' \xrightarrow{a} s$ for some transition label $a$. If $s$ is stable (i.e., either $l = \text{begin}(a)\text{end}(a)$ for some $a \in \text{a\text{set}}$, or $l = \text{clock}$ but none of the actors are executing, or $l = \text{write}(a)$ with no further write transitions possible), then a begin fire transition, or an end fire transition (if preconditions are met), or a clock transition (if no end transitions are enabled) are possible from state $s$. If $l = \text{read}(a)\text{write}(a)$ and $s$ is unstable, then at least one read or write transition is enabled at $s$; if $s$ is stable, then see above. Hence at least one transition is enabled at $s$, and thus $s$ must have a successor state.

**Theorem 8.2.** For any PCT state $s$, $NSS(s)$ is unique.

**Proof.** Let PCT state $s = (\gamma, v)$ have multiple next stable states. The possible scenarios are discussed below.

**Scenario 1:** If $\forall a \in \text{a\text{set}}, v(a) = \emptyset$, then $s$ is a $NSS(s)$ by definition. Hence $NSS(s)$ is unique for this scenario.

**Scenario 2:** If $v(a) = \{e(v), r(v)\}$, then a possible sub trace from $s$ is $\tau' = s \xrightarrow{\text{read}(a)} s' \xrightarrow{\text{write}(a)} s''$. The first stable state in the trace is $s''$ and is $NSS(s)$. There are no other transitions possible for $s$ and $s''$. Hence $NSS(s)$ is unique for this scenario. Without loss of generality, the same argument can be made for any other actor $a' \in \text{a\text{set}} \setminus \{a\}$.

**Scenario 3:** If $v(a) = \{(e(v), (e', r))\}$ $(e, e' \in N)$, and $\forall a' \in \text{a\text{set}} \setminus \{a\}, v(a') = \emptyset$, there are several possible sub traces starting from $s$. We will first consider the traces where all read transitions precede all the write transitions. Consider the trace $\tau : s \xrightarrow{\text{read}(a)} s_1 \xrightarrow{\text{write}(a)} s_2 \xrightarrow{\text{read}(a)} s_3 \xrightarrow{\text{write}(a)} s_4$. Without loss of generality, the same argument can be made for any other actor $a' \in \text{a\text{set}} \setminus \{a\}$.

**Scenario 4:** If $v(a) = \{(e(v), (e', r'))\}$ $(e, e' \in N)$, and $\forall a' \in \text{a\text{set}} \setminus \{a\}, v(a') = \emptyset$, then the above argument can be extended to show that there are multiple possible sub traces starting from $s$. In each such sub trace, there are three unstable states (excluding $s$) followed by a stable state. The four states are generated by two read transitions and the two corresponding write transitions. The transitions can be interspersed in any order, but the channel quantity and the execution map would be identical for the stable state along all of the sub traces implying that $NSS(s)$ is unique for this scenario.

The arguments discussed for the last three scenarios can be extended for arbitrary number of active actors with arbitrary number of instances.

**Corollary 8.3.** For a trace $\tau$, the set of stable states of $\tau$ is identical to the set of stable states in all $\tau$.

**Proof.** Every PCT state $s$ has an unique $NSS(s)$ (Lemma 8.3). This implies that generating sub traces between $s$ and $NSS(s)$ do not affect any state before $s$ or any state after $NSS(s)$. States on sub traces between $s$ and $NSS(s)$ are all unstable. Hence generating the sub traces neither adds nor removes any stable state.
Figure 3: A sub trace with one complete firing of actor $i1$

Figure 4: A sub trace where actors $i1$ and $i2$ fire