Implementation-Aware Buffer-Throughput Tradeoff in Embedded Stream Applications

Kamyar Mirzazad*, Matin Hashemi*, Volodymyr Khibin†, and Soheil Ghiasi†

*Sharif University of Technology
{kammirzazad,matin}@sharif.edu

†University of California, Davis
{vykhibin,ghiasi}@ucdavis.edu

Abstract. We study the tradeoff between throughput and interprocessor buffer size of streaming applications specified as SDF graphs that are to be implemented on MPSoC platforms. We demonstrate the inaccuracy of the analysis when implementation constraints are not taken into consideration, and propose a rigorous SDF graph transformation which brings implementation awareness into the state of the art tradeoff analysis technique. Cycle accurate simulations show that our approach results in significantly more accurate estimates.

1 Introduction

A streaming application modeled in SDF graph is represented as a set of concurrent actors that communicate by sending and receiving messages (a.k.a. tokens) via point-to-point FIFO buffers [6, 12]. According to SDF operational semantics, every actor consumes (produces) its input (output) tokens upon start (completion) of its execution. An implementation-oblivious analysis would have to honor model execution according to the operational semantics. In real implementations, however, not all of actor's input (output) tokens are consumed (produced) at exactly the same time. For example, if actors are implemented as software modules running on embedded single-issue processor cores, a sequential order is implicitly imposed on the consumption (production) of input (output) tokens. We utilize the state of the art implementation-oblivious buffer-throughput tradeoff analysis developed by Stuijk et al [11], and demonstrate that its tradeoff characterization is conservative.

To address the resulting inaccuracy, we propose to take target implementation into account during analysis. The implementation information that we expose to the analysis engine is quite limited in nature. Thus, the analysis is not tied to any specific architecture. We propose SDF graph transformations to capture the sequential nature of software execution in MPSoC implementations, and to rigorously embed implementation awareness into the model. Experiments with a number of streaming applications show that the implementation-awareness yields significantly more accurate and smaller buffer sizes (9X on average) for the same throughput.

Compared to cycle accurate simulation of a specific MPSoC implementation, the error of the proposed method in throughput estimation is 19%, while it runs over 100X faster. Thus, the proposed implementation-aware analysis offers a favorable tradeoff between analysis solely based on SDF operational semantics (implementation-oblivious), and cycle-accurate simulation. The high degree of throughput estimation accuracy and substantial savings in runtime are due to the fact that the proposed approach considers only key pieces of information from target implementation, as opposed to over-emphasizing or ignoring implementation-specific information.

2 Preliminaries

2.1 Synchronous Dataflow (SDF) Model

SDF applications are modeled as a directed graph $G(V, E)$, where vertex $v \in V$ represents an actor, and edge $uv \in E$ represents a point-to-point FIFO *channel* from actor u to v. Actors communicate by sending/receiving data items, called *tokens*, via the channels. Actor v is a tuple (In, Out, ε) and channel uv is a tuple (u, v, r_p, r_c). $In(v) \subset E$ and $Out(v) \subset E$ are input and output channels of v, and $\varepsilon(v)$ is its execution time, i.e., the average time actor v takes to perform its computation in an implied implementation (Figure 1.A). For a channel $uv \in E$, the number of tokens produced by u for channel uv, on every firing of u, is called the production rate of uv and is denoted by $r_p(uv)$. Consumption rate $r_c(uv)$ is defined similarly. Data rates are constant and actor execution is meant to continue infinitely [6, 11].

Fig. 1. A) Example SDF graph (actors and channels are annotated with execution times and data rates, respectively.) B) An implied implementation of self-timed execution.

SDF Operational Semantics: Upon firing of actor $v \in V$, it simultaneously consumes $r_c(uv)$ tokens from all of its input channels $uv \in In(v)$, then carries out its computation in $\varepsilon(v)$ time units, and finally it simultaneously produces $r_p(vw)$ tokens on all of its output channels $vw \in Out(v)$.

Firing Condition: Actor v can fire at time t, if and only if (I) previous firing of v is completed , and (II) enough tokens are available on all its input channels, that is $\forall uv \in In(v) : \gamma(uv, t) \geq r_c(uv)$, where $\gamma(uv, t)$ quantifies the number of tokens stored in uv. In Figure 1.A we have $\varepsilon(b) = 300$, $r_c(ab) = 50$ and $r_p(bc) = 10$. Thus, upon availability of at least 50 tokens on ab, actor b can fire. In every firing of b, 50 tokens are simultaneously consumed from ab, then the computation of actor b is carried out in 300 time units, and finally 10 tokens are simultaneously produced on channel bc.

2.2 Target Platform Model

We target MPSoC platforms whose abstract model for SDF execution can be viewed as a distributed-memory message-passing system with point-to-point interprocessor FIFO buffers (Figure 1.B). This abstract view is directly implemented in some platforms such as AsAP [13] and TILE64 static network [3]. Some other platforms are programmed to implement virtual buffers with software assistance (e.g., in shared-memory systems, via shared arrays in memory along with proper locking).

We focus on self-timed execution which implicitly assumes allocation of dedicated core to every actor (Figure 1.B). Under self-timed execution, an actor fires as soon as its firing conditions are satisfied [11]. In many cases, an embedded application is developed on an MPSoC target by splitting the application into many actors and placing each actor on one core (e.g., 1080p H.264 encoder on AsAP [14]). Otherwise, the collection of actors allocated to the same processor under static schedule can be viewed as a coarse-grain actor in an up-scaled graph that conforms to our model.

2.3 Buffer-Throughput Tradeoff

Throughput is one of the most important quality metrics in streaming applications. A number of factors, such as actor execution times, interprocessor buffer capacities and SDF graph cycles, impact steady-state throughput[1] [4]. In practice, the FIFO channels are implemented with limited buffering capacity, which can limit the throughput [11]. Characterizing the tradeoff between interprocessor buffer sizes and application throughput is quite important, as typical design scenarios require the implementation to meet performance requirement at minimum buffer size allocation.

Throughput: Throughput of an actor v is defined as the average number of v firings per unit time [4], i.e., $\tau(v) = \lim_{T\to\infty} \frac{1}{T} \times \big(\#\text{ of } v \text{ firings from } t = 0 \text{ to } t = T\big)$. Since SDF data rates are constant, in the steady state, the number of times different actors fire are a constant factor of one another. Hence, normalized throughput, which decouples the choice of actor from SDF throughput, is defined as $\tau = \tau(v) \div q(v)$ for an arbitrary actor $v \in V$, where, $q(v)$ is the number of times v fires in one iteration of the simplest periodic schedule [4, 6]. In our example, $q(a, b, c) = (5, 2, 1)$.

Buffer size: Buffer size $\beta(uv)$ is defined as capacity of the interprocessor FIFO buffer which implements channel $uv \in E$. In other words, $\beta(uv)$ is the maximum number of tokens that channel uv can hold at any time during execution. Formally, $\gamma(uv, t) \leq \beta(uv)$. Total buffer size is defined as $|\beta| = \sum_{\forall uv \in E} \beta(uv)$.

2.4 Tradeoff Analysis Based on SDF Operational Semantics

According to SDF operational semantics, after actor u fires and completes its computation, at least $r_p(uv)$ empty spaces are required on every output channel $uv \in Out(u)$ in order to write tokens produced by u. Otherwise, since sufficient space is not available, the actor is stalled at the end of its firing. The actor will resume execution to complete its previously stalled firing only after enough space becomes available.

[1] We use the terms "throughput" and "steady-state throughput" interchangeably.

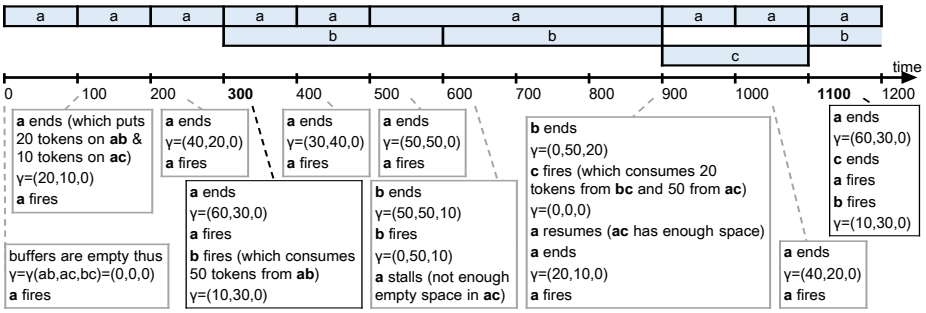

Fig. 2. Throughput Analysis based on SDF operational semantics when $\beta(ab, ac, bc) = (60, 50, 20)$. At $t = 1100$, states of actors and channels are the same as $t = 300$.

Stall and Resume Conditions: Under self-timed execution assumption, a running actor $u \in V$ fired at time t_1 stalls at time $t_2 > t_1$ if and only if $\exists uv \in Out(u)$: $\beta(uv) - \gamma(uv, t_2) < r_p(uv)$ and resumes operation at a time $t_3 > t_2$ if and only if $\forall uv \in Out(u) : \beta(uv) - \gamma(uv, t_3) \geq r_p(uv)$.

Throughput is degraded if actors stall due to unavailable space. For a given set of buffer sizes β, throughput can be obtained by considering the firing, stall and resume conditions. Stuijk et al. developed a Pareto point exploration algorithm to find throughput vs. total buffer size of an SDF graph [11]. The algorithm works by executing the SDF graph, for a judiciously selected subset of buffer size allocations, while maintaining the state of actors and channels. Each step of application execution is modeled as a transition in the augmented state space of actors and channels. When a state is revisited for the first time, the execution arrives its steady-state, as a cycle in the state space is formed. Subsequently, throughput $\tau(v)$ is calculated as the number of v firings during the cycle, divided by the amount of time lapsed in the cycle. The above procedure is repeated for different buffer sizes in order to evaluate all Pareto points [11]. We later utilize this algorithm in our experimentation in Section 4.

Figure 2 demonstrates throughput calculation for our running example of Figure 1 when $\beta(ab, ac, bc) = (60, 50, 20)$. At time $t = 1100$, the progress and capacities of all actors and channels are equal to those of time $t = 300$. Thus, the steady state is reached, and $\tau(b) = \frac{2}{1100-300}$ and $\tau = \frac{\tau(b)}{q(b)} = \frac{1}{800}$. If the buffer size of channel ac is increased from 50 to 70, throughput improves from 1/800 to 1/600, because channel ac becomes full 200 time units later, and actor a stalls for 200 fewer time units.

Deadlock: When at least one stalled actor never resumes operation, then deadlock happens, in which case, overall throughput τ becomes zero. By analyzing SDF operational semantics, Ade et al. [1] proved the following theorem regarding deadlocks.

Theorem 1. If there exists a channel $uv \in E$ with buffer size $\beta(uv)$ less than
$$r_p(uv) + r_c(uv) - gcd(r_p(uv), r_c(uv))$$
then deadlock happens between actors u and v [2]. We denote the above equation with $\beta_{min}(uv)$. In our example, $\beta_{min}(ab, ac, bc) = (60, 50, 20)$. Note that the theorem does not state if deadlock happens when "for all" channels $uv \in E$, $\beta(uv) \geq \beta_{min}(uv)$. In such a case, more thorough deadlock analysis is required [15].

3 Implementation-Aware Buffer-Throughput Analysis

We propose improving the analysis by taking into account very few pieces of information on the target MPSoC implementation in form of the following abstract view.

3.1 Abstract View of Implementation

Figure 3.A demonstrates our abstract view of embedded software that implements the SDF application on a MPSoC. First, the required tokens are read from input FIFO buffers, next the actor's transformation computation is executed, and finally, the generated data is written to output buffers. This sequence is repeated infinitely. Let's define "task" as "implementation of actor" according to this abstract view.

Figure 3.B shows the typical implementation of communication API calls. The SDF model allows tokens of arbitrary size, hence, one may define a large block of

[2] Proofs of all theorems are omitted due to space limitation.

(A)

```
// task 'a' on P1
token ab[20];
token ac[10];

while(){
 a(ab,ac);
 write(ab,20,P2);
 write(ac,10,P3);
}
```

```
// task 'b' on P2
token ab[50];
token bc[10];

while(){
 read(ab,50,P1);
 b(ab,bc);
 write(bc,10,P3);
}
```

```
// task 'c' on P3
token bc[20];
token ac[50];

while(){
 read(bc,20,P2);
 read(ac,50,P1);
 c(bc,ac);
}
```

(B)

```
void write (token* x, int n, int dst){
 for i=[0,n)
  for j=[0,s)
   writePacket(x[i],j,dst);
}
```

```
void read (token* x, int n, int src){
 for i=[0,n)
  for j=[0,s)
   readPacket(x[i],j,src);
}
```

Fig. 3. Abstract view of A) software implementation, and B) communication APIs.

data (e.g., a video frame) as a single token. However, interconnect networks have limited bandwidth and not necessarily capable of transferring one token at a time (e.g., one video frame takes multiple clock cycles). In practice, each token may need to be split into $s = \lceil \frac{sizeof(token)}{sizeof(packet)} \rceil$ packets and be transferred sequentially as shown in the inner loop in Figure 3.B. The outer loop repeats this process for every token in the array. For brevity, we assume $s = 1$ in the rest of this paper. Our approach, however, is readily extensible to other packet sizes.

Note that this abstract view refers to very general implementation guidelines, rather than a specific platform or software coding style. A number of different concrete implementations conform to the abstract view, albeit with different parameters. For example, many interprocessor API calls, which appear atomic to the programmers, are implemented by splitting large data into smaller pieces and transferring them sequentially. As another example, in software implementations conceptually-concurrent token transfer would have to be implemented in some order.

3.2 Implications of Implementation-Awareness

The sequential nature of instruction execution on single-issue processor cores implies that a task can write (read) only one token to (from) only one channel at a time. This additional information on implementation leads to an operation that is quite different from the pure SDF model in which actors write to (read from) all channels simultaneously at arbitrary rates.

As shown in Figure 2, analysis based on SDF model concluded that throughput for buffer size $\beta(ab, ac, bc) = (60, 50, 20)$ is $\tau = \frac{1}{800}$. Actor c waits for data from actor b and upon availability of sufficient number of tokens produced by b, actor c fires and immediately consumes all of them.

The implementation, however, behaves differently by allowing tasks to only read and write one token at a time (Figure 3). Task c (processor P3) stalls when it tries to read for the first time, since there is no token available on channel bc. Once task b (processor P2) places the first token on this channel, the stalled `readPacket` function in c resumes execution and reads that token. In this setting, therefore, $\beta(bc) = 1$ would be sufficient to achieve the same throughput as shown in Figure 2. This amounts to a substantial 20X reduction in buffer size of bc without any throughput degradation. The example underscores the inaccuracy of implementation-oblivious analysis, and motivates us to consider the implementation in buffer-throughput analysis.

3.3 Implementation-Aware SDF Graph Transformation

Our implementation-aware approach to characterization of buffer-throughput tradeoff works in two steps. First, we transform the original SDF graph G by embedding limited information of the nature of target implementation into the graph. Subsequently, the transformed SDF graph G' is analyzed by leveraging an implementation-oblivious analysis technique, described in Section 2.4, to obtain its buffer-throughput tradeoff points (Figure 6.B).

Based on the abstract view of implementation, tasks can read (write) only one token at a time (property I), and from (to) only one channel at a time (property II). Property I is modeled by adding virtual *reader* and *writer* actors, and property II is captured by adding virtual *sync* actors to the SDF graph.

Reader and Writer Actors: For every channel $uv \in E$, a virtual writer actor W is added at the output of actor u, and a virtual reader actor R is added at the input of actor v, such that the output of actor W feeds data into the input of actor R (Figure 4.A). All reader and writer actors have identity data transformation functionality and thus, do not alter the data.

Reader and writer actors have data rates of 1. For every firing of u, W fires $r_p(uv)$ times sequentially (auto-concurrency is not allowed), and consumes the tokens produced by u one at a time. Similarly, for every $r_c(uv)$ firings of R, actor v fires once. Buffer sizes for virtual channels uW and Rv are set to $r_p(uv)$ and $r_c(uv)$, respectively. Buffer size of channel uv in the original graph determines the buffer size of channel WR in the transformed graph (Figure 4.A).

Writer actor W models behavior of the `writePacket` function call (Figure 3.B). $r_p(uv)$ firings of W, which produce $r_p(uv)$ tokens, model the loop and iterative calls to `writePacket` function in the `write` API call in execution of task u. Intuitively, virtual channel uW models the local processor memory that temporarily stores the output tokens of u (e.g., `token ab[20]` in task a in Figure 3.A). Similarly, actor R models the `readPacket` call, and channel Rv models the local memory that temporarily stores the input tokens of a task v (e.g., `token ab[50]` in task b in Figure 3.A).

Theorem 2. *Addition of reader/writer actors preserves SDF functionality.*

As a result of the above transformation, every actor $v \in V$ is transformed into a subgraph G_v (Figure 4.B). Let $|In(v)|$ and $|Out(v)|$ denote the number of input and output channels of v. Let c_i for $i \in [\ 1, |In(v)|\ ]$ denote the consumption rates for input channels of v, and let p_j for $j \in [\ 1, |Out(v)|\ ]$ denote the production rates for output channels of v. Subgraph G_v has $|In(v)|$ reader actors R_1, R_2, ... $R_{|In(v)|}$, and $|Out(v)|$ writer actors W_1, W_2, ... $W_{|Out(v)|}$. Production and consumption rates and buffer sizes of virtual channels in G_v are set as:

virtual channel	r_p	r_c	β
R_iv	1	c_i	c_i
vW_j	p_j	1	p_j

A firing of actor v in G corresponds to the following sequence of events in subgraph G_v in the transformed graph. Reader actor R_i fires c_i times. As a result, it reads c_i tokens from the corresponding input channel of G_v and writes them to virtual channel R_iv. At this point, actor v fires once and consumes these tokens and produces p_j tokens on virtual channels vW_j. Next, virtual actor W_j fires p_j times, and copies the tokens to the corresponding output channel of G_v. That is, subgraph G_v models the execution of task v based on the implementation view discussed in Section 3.1.

Sync Actors: In subgraph G_v developed above, reader actors, writer actors and actor v can potentially fire simultaneously. In order to correctly model the sequential nature of task execution based on the abstract implementation view, we need to eliminate the simultaneity. We add a number of virtual sync actors to every subgraph G_v in order to enforce the following sequential ordering on the execution of actors.

$$R_1, R_2, \ldots R_{|In(v)|}, v, W_1, W_2, \ldots W_{|Out(v)|}$$

This sequential ordering conforms to the implementation of task v, where first the `read` API calls, next the computation of actor v, and finally the `write` API calls are executed on the processing core (Figure 3.A).

To enforce the above ordering in G_v, we add virtual sync actors $S^R_{i,i+1}$ between R_i and R_{i+1}, and virtual sync actors $S^W_{j,j+1}$ between W_j and W_{j+1} (e.g., S_1, S_2 and S_3 in Figure 5.A), and set data rates and buffer size of the newly added virtual channels (marked blue in the figure) as follows:

virtual channel	r_p	r_c	β	virtual channel	r_p	r_c	β
$R_iS^R_{i,i+1}$	1	c_i	c_i	$W_jS^W_{j,j+1}$	1	p_j	p_j
$S^R_{i,i+1}R_{i+1}$	c_{i+1}	1	c_{i+1}	$S^W_{j,j+1}W_{j+1}$	p_{j+1}	1	p_{j+1}

The parameters are carefully selected such that upon c_i firings of R_i, $S^R_{i,i+1}$ fires once, and then R_{i+1} can fire c_{i+1} times. Similarly, upon p_j firings of W_j, $S^W_{j,j+1}$ fires once, and then W_{j+1} can fire p_{j+1} times. The construction ensures that the desired ordering is enforced by creating appropriate dependencies.

Lastly, we add a sync actor between $W_{|Out(v)|}$ and R_1 (e.g., S_4 in Figure 5.B). This creates a cycle in G_v and prohibits concurrent execution of actors (e.g., a reader actor and a writer actor). Specifically, it stops R_1 from firing until $W_{|Out(v)|}$ fires $p_{|Out(v)|}$ times. Note that c_1 initial tokens are required on this cycle in order to avoid deadlock, since R_1 fires c_1 times for every firing of v.

Sync actors has no effect on the transfer function of reader/writer actors. In particular, the reader and writer actors continue to copy application data (black and green channels in Figure 5.B), and do not mix up the data with dependency channels of the sync actors (blue channels in Figure 5.B).

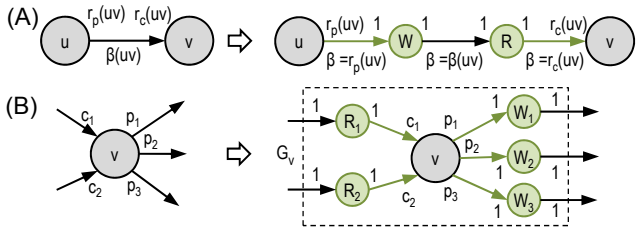

Fig. 4. A) Writer and reader actors for channel $uv \in E$. Virtual actors and channels are shown in green. B) The transformed subgraph G_v for an actor v with 2 incoming and 3 outgoing channels.

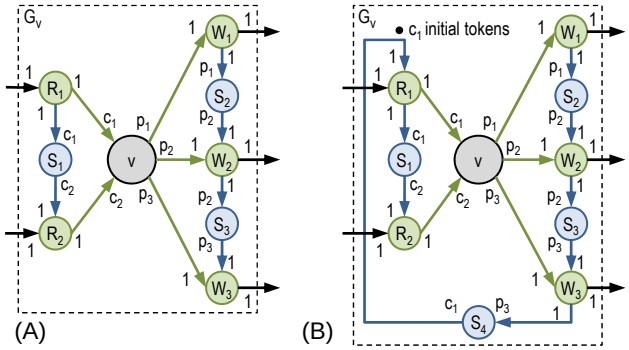

Fig. 5. A) Sync actors S_1, S_2 and S_3 enforce the sequential order $R_1, R_2, v, W_1, W_2, W_3$ in subgraph G_v of Figure 4.B. The newly added virtual actors & channels are shown in blue. B) Sync actor S_4 prohibits auto-concurrency.

Theorem 3. *Addition of sync actors preserves the original SDF functionality.*

It follows that the transformed subgraph G_v in G' correctly models the execution of task v according to the abstract view discussed in Section 3.1.

3.4 Throughput Analysis

Since sync actors are added to only enforce a sequential order among read and write operations, they must not have any impact on the total execution time of G_v. We conservatively assume that the information regarding platform-dependent latency of read and write operations are unavailable. Hence, the execution times of `read` and `write` API calls and the data transformation computation of a task are viewed to be inseparable. To capture this in G_v, we set the execution times of reader and writer actors to zero ($\varepsilon = 0$), and assign the entire execution time of the original actor to v. In case of access to specific parameters of the target architecture, one could improve the model fidelity by separating the latency of read and write operations from data transformation computation, and assigning more accurate execution times to actors in G_v.

Theorem 4. *Given an SDF graph G and a set of buffer size choices β for channels in G, throughput of transformed graph G' is not less than G.*

Theorem 5. *The maximum throughput of G and G', which is obtained when all channels of G have infinite buffer size, are equal.*

3.5 Related Work

Many previous analysis algorithms are solely based on SDF operational semantics [2, 4, 8, 11]. To increase accuracy in throughput analysis, Moonen et al. [7] proposed to construct a cyclo-static dataflow (CSDF) graph from the given SDF graph by splitting the computation of a SDF task into multiple phases (white box actor model). Our proposed technique, however, focuses on accurate modeling of token production and consumption order (black box actor model), and does not require manual decomposition of actor computation.

Oh and Ha [9] proposed a fractional rate model to reduce the buffer size requirement. For applications that work on large blocks of data, e.g., video frames, the dataflow graph is manually transformed into another graph in which, actors operate on smaller pieces of data, e.g., one row of a video frame. As a result the buffer requirement is reduced (white box actor model). Our proposed technique does not require modification of tasks' functional behavior, and treats them as unknown black boxes.

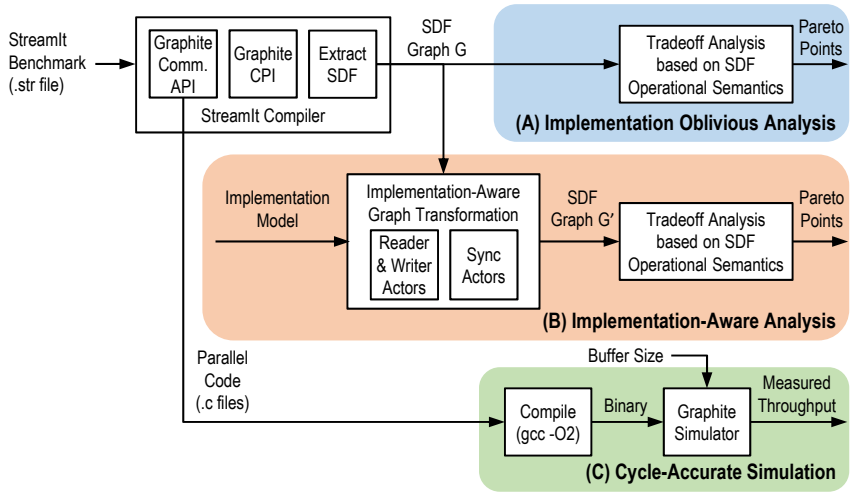

Fig. 6. Experimentation flow: A) Baseline implementation-oblivious buffer-throughput tradeoff analysis based on SDF operational semantics. B) Proposed implementation-aware analysis. C) Cycle-accurate simulation of the compiled binary code.

4 Experiments

To evaluate the proposed technique we employ StreamIt benchmarks. StreamIt is a programming language and compiler for stream programs [12]. For every benchmark application, we execute StreamIt compiler (Figure 6, top left) and extract SDF graph topology, data rates (r_p and r_c) and estimates of actor execution time (ε). Actor execution times are estimated by the StreamIt compiler based on rough mapping between high-level StreamIt language constructs and typical processor instruction sets. Original cycle per instruction (CPI) estimates of StreamIt compiler are based on the RAW processor. We have modified StreamIt such that its CPI estimates match Graphite processor model [5]. Graphite is a cycle-accurate MPSoC simulator, and is used as the target platform in our experimentation.

The proposed implementation-aware tradeoff analysis involves two steps (Figure 6.B). First, we apply the proposed transformation discussed in Section 3 and transform the SDF graph G into G'. The transformation is based on our abstract view of target implementation as discussed in Section 3.1, which includes very limited information on the target implementation (sequentially-ordered read and write operations) into graph G'.

Next, we perform buffer-throughput tradeoff analysis on G' based on SDF operational semantics, as discussed in Section 2.4. In this part, we utilize SDF3 [10, 11], which implements the tradeoff analysis algorithm explained in Section 2.4. We have modified SDF3 to force it to ignore the virtual channels introduced by the transformation, while exploring the search space. Buffer size of the virtual channels are also omitted from the reported total buffer size. The analysis yields a set of pareto optimal points between the total interprocessor buffer size, $|\beta|$, and the corresponding overall throughput, τ. To compare the proposed approach against an established standard, we also perform the implementation-oblivious tradeoff analysis directly on graph G (Figure 6.A).

Figure 7 shows the result of tradeoff analysis for both the proposed implementation aware and the baseline implementation oblivious techniques. The experimental results show that for all benchmarks the implementation-aware tradeoff analysis yields much smaller buffer sizes than the implementation-oblivious analysis for the same level of throughput. This confirms our claim that the analysis solely based on SDF operational semantics is overly conservative and yields far larger buffer sizes than required. In addition, it empirically confirms Theorem 5, since both approaches always result in the same maximum throughput. In case of `mpeg` application, for example, the implementation oblivious technique reports that a total buffer size of $|\beta| = 15243$ is required to achieve the maximum throughput, while the implementation aware analysis reduces this to $|\beta| = 326$, which is a 46X smaller.

Figure 8 highlights the substantial reduction in total buffer size requirement, using the data of Figure 7. The horizontal axis is in logarithmic scale (base 3) and compares the implementation oblivious vs. implementation aware ratio of total buffer size, $|\beta|$, required to achieve the maximum throughput, 80% of the maximum throughput, 50%

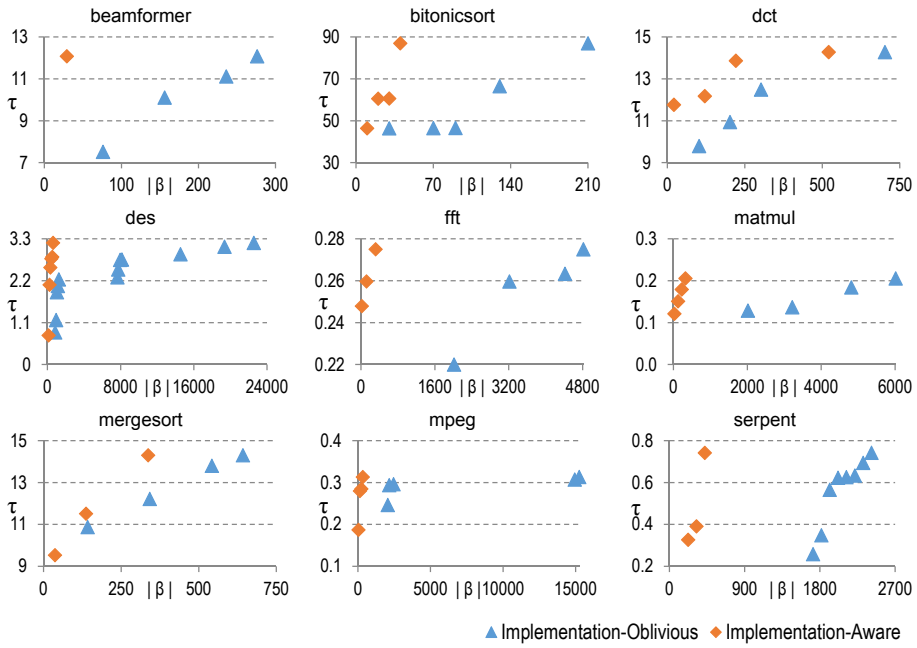

Fig. 7. Pareto points between total interprocessor buffer size, $|\beta|$, and the corresponding throughput, τ, for both the baseline implementation-oblivious and the proposed implementation-aware tradeoff analysis techniques. The proposed method yields substantially improved buffer size estimates under identical throughput constraints.

of the maximum throughput, and to avoid deadlock, respectively. On average (geometric mean), using the proposed implementation aware technique, total buffer size $|\beta|$ required to achieve the maximum throughput, 80% of the maximum throughput, 50% of the maximum throughput, and to avoid deadlock is reduced by a factor of 8.5X, 9.0X, 8.5X and 9.3X, respectively.

Figure 9 shows the ratio of the time it takes to run the proposed implementation aware tradeoff analysis technique over the time it takes to run the baseline implementation oblivious technique. The ratio heavily depends on the application, e.g., 98X for `mpeg` and 0.11X for `fft` benchmark. On average (geometric mean), the ratio is 7.3X. The workstation employed in our experiments has 8 GB of memory and 3.4 GHz Core i7 processor with 8 MB of cache.

To quantify the accuracy of estimates produced by the baseline and proposed techniques, we set out to generate executable binaries and simulate their performance under different buffer sizes using the Graphite cycle-accurate simulator [5] (Figure 6.C). Specifically, we utilize StreamIt compiler (RAW processor backend) and generate parallel software code in form of multiple C files from StreamIt SDF applications[3]. We parse the C files and replace generated RAW interprocessor communications with Graphite interprocessor communication API calls. Next, we compile the generated code into binary using `gcc -O2` command, and pass the binaries to Graphite for cycle-accurate simulation (Figure 6.C).

For every benchmark, we adjust the buffer size distribution ($\beta(uv)$ for all channels uv) to match buffers that result in the maximum throughput according to implementation-aware model analysis. That is, we select buffer size distribution of the orange diamond-shaped point with the highest throughput in every pareto chart in Figure 7. We have slightly modified Graphite to simulate interprocessor channels with limited buffer size. Since the simulated number of cycles can vary from one application iteration to the next (due to control flow variations, cache effects, etc), we measure throughput by examining its steady-state long term average. That is, we continue the simulation until no significant change (no more than 1%) in long term throughput is observed.

Figure 10 compares the throughput esimated by implementation aware and implementation oblivious analysis techniques for the selected buffer size distribution,

[3] We also experimented with SDF3 benchmarks. However SDF3 benchmarks merely include graph parameters and not task implementations. Thus, we could only perform the experiments shown in Figure 6.A and 6.B and not 6.C. Detailed results are omitted due space limitations. For SDF3 benchmarks, on average, buffer size reduction using impl.-aware analysis is 6X, and runtime ratio of impl.-aware over impl.-oblivious analysis is 5X.

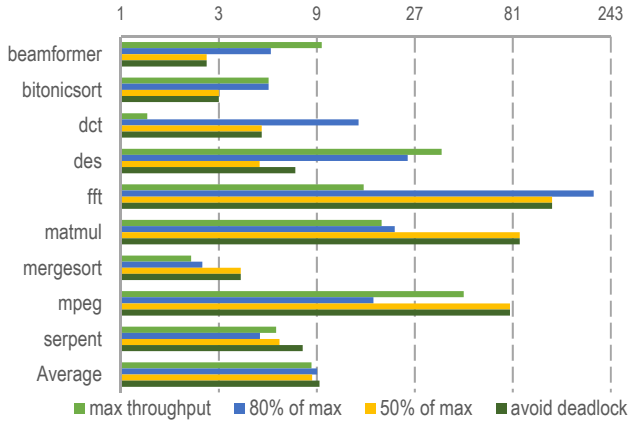

Fig. 8. Reduction in total buffer size estimates using implementation aware analysis. X-axis is in base 3 logarithmic scale.

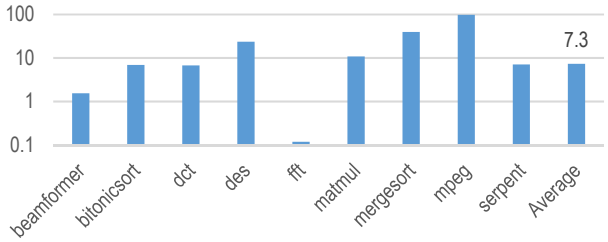

Fig. 9. Runtime of implementation aware over implementation oblivious analysis.

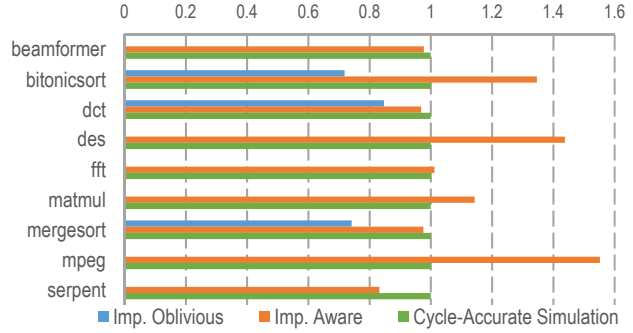

Fig. 10. Comparison of (normalized) throughput estimated by implementation aware and implementation oblivious techniques against cycle-accurate simulation. Implementation oblivious technique inaccurately predicts deadlock in most cases, and is less accurate in the remaining cases.

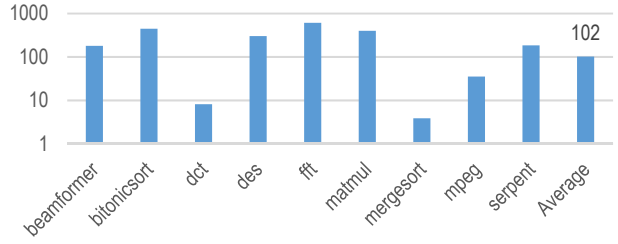

Fig. 11. Runtime of cycle-accurate simulation over the proposed implementation aware analysis technique.

against cycle-accurate simulated throughput. The numbers are normalized with respect to the throughput given by Graphite. Hence, a value of 1.0 means zero error in estimation of throughput, in comparison with cycle-accurate simulation. The implementation oblivious analysis falsely reports deadlock ($\tau = 0$) in six out of nine benchmarks. This occurs because the selected buffer sizes are smaller than what implementation oblivious analysis believes to be required for avoiding deadlock. In the other three benchmarks (`bitonicsort`, `dct` and `mergesort`), the average error is 23%. The overall average error across all the nine benchmarks using the implementation oblivious analysis is 74%. The implementation aware analysis, however, estimates the throughput very closely. Compare the orange and green bars in Figure 10. The error in estimation of throughput is less than 5% in `beamformer`, `dct`, `fft` and `mergesort` benchmarks. On average, the error of implementation aware analysis in estimation of throughput is 19%, compared to cycle-accurate simulation.

Figure 11 shows runtime of cycle-accurate simulation over runtime of implementation aware analysis for all benchmarks. The runtime ratio is higher than $100X$ in six out of nine benchmarks. In the `fft` benchmark the ratio is $606X$. On average (geometric mean), it takes about 102X longer to run cycle-accurate simulations than to run the proposed implementation aware analysis.

Let us highlight the key benefits offered by the proposed approach. In comparison with implementation oblivious analysis (analysis solely based on SDF operational semantics), it offers substantially more accurate (9X smaller) buffer size esimates for the same level of throughput. This is achieved by taking into account very limited information on target implementation. In comparison with cycle-accurate simulation, the implementation aware analysis offers 102X speedup in runtime and relatively low error (19%) in estimation of throughput. As such, our porposed technique offers a very favorable tradeoff point for early design space exploration. Note that the proposed method is performed at a high-level on SDF graphs, while the cycle-accurate simulation is performed on compiled binary codes and thus, has access to all relevant implementation details, such as processors' instruction set, cache, program control flow, among others.

5 Conclusion

We investigated the tradeoff between buffer size and throughput of streaming applications modeled as SDF graphs. We demonstrated that the quality of model-based tradeoff exploration algorithms can be considerably improved if one incorporates very mild assumptions about the target implementation into analysis.

References

1. M. Ade, R. Lauwereins, and J. Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. *Design Automation Conference*, 1997.
2. M. A. Bamakhrama and T. P. Stefanov. On the hard-real-time scheduling of embedded streaming applications. *Design Automation for Embedded Systems*, 2012.
3. S. Bell et al. Tile64 - processor: A 64-core soc with mesh interconnect. *International Solid-State Circuits Conference*, 2008.
4. A. H. Ghamarian et al. Throughput analysis of synchronous data flow graphs. *International Conference on Application of Concurrency to System Design*, 2006.
5. Graphite. http://graphite.csail.mit.edu.
6. E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 1987.
7. A. Moonen et al. Practical and accurate throughput analysis with the cyclo static dataflow model. *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2007.
8. O. M. Moreira and M. J. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007.
9. H. Oh and S. Ha. Fractional rate dataflow model for efficient code synthesis. *Journal of VLSI signal processing systems for signal, image and video technology*, 2004.
10. SDF3. http://www.es.ele.tue.nl/sdf3.
11. S. Stuijk et al. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. *Design Automation Conference*, 2006.
12. W. Thies et al. Streamit: A language for streaming applications. *International Conference on Compiler Construction*, 2002.
13. D. Truong et al. A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. *IEEE Symposium on VLSI Circuits*, 2008.
14. Z. Xiao and B. Baas. 1080p h.264/avc baseline residual encoder for a fine-grained many-core system. *IEEE Transactions on Circuits and Systems for Video Tech.*, 2011.
15. Y. Zhou and E. A. Lee. A causality interface for deadlock analysis in dataflow. *International Conference on Embedded Software*, pages 44–52, 2006.