Implementation-Aware Model Analysis: The Case of Buffer-Throughput Tradeoff in Streaming Applications

Kamyar Mirzazad Barijough* Matin Hashemi* Volodymyr Khibin† Soheil Ghiasi†
* Sharif University of Technology
†University of California, Davis
kammirzazad@ee.sharif.edu, matin@sharif.edu [1], vykhibin@ucdavis.edu, ghiasi@ucdavis.edu

Abstract

Models of computation abstract away a number of implementation details in favor of well-defined semantics. While this has unquestionable benefits, we argue that analysis of models solely based on operational semantics (implementation-oblivious analysis) is unfit to drive implementation design space exploration. Specifically, we study the tradeoff between buffer size and streaming throughput in applications modeled as synchronous data flow (SDF) graphs. We demonstrate the inherent inaccuracy of implementation-oblivious approach, which only considers SDF operational semantic. We propose a rigorous transformation, which equips the state of the art buffer-throughput tradeoff analysis technique with implementation awareness. Extensive empirical evaluation show that our approach results in significantly more accurate estimates in streaming throughput at the model level, while running two orders of magnitude faster than cycle-accurate simulation of implementations.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Modeling Techniques

Keywords Synchronous DataFlow (SDF), Buffer-Throughput Tradeoff Analysis, Embedded Multi-Processor

1. Introduction

The model-based design methodology advocates separation of application specification from target implementation, and representation of application behavior using formal models of computation [18, 20]. Such models enable one to develop or to utilize various analysis, optimization and synthesis techniques for either exploration of implementation space or generation of efficient implementations. While this approach has unquestionable benefits, we argue that in certain situations *complete* separation of specification from target platform obscures key pieces of information that are essential for accurate characterization of the design space.

We study this rather general idea in the context of tradeoff analysis between buffer size and throughput of streaming applications specified as synchronous dataflow (SDF) graphs that are to be implemented on multiprocessor system on chips (MPSoC). In the SDF model, the application is represented as a set of concurrent tasks that communicate by sending and receiving messages (tokens) via point-to-point FIFO buffers [11, 23]. The rates at which tasks produce and consume messages are constant and known at compile time. This property enables utilization of a number of analysis techniques at compile time, including scheduling of tasks [10, 12] and characterization of the tradeoff between buffer size requirement and streaming throughput [22].

SDF operational semantics specifies consumption of all input messages to a task upon start of its execution, and production of all its output messages upon completion of its execution. An implementation-oblivious analysis technique would have to follow model execution according to the operational semantics. In actual implementations, however, not all messages of a task are consumed or produced at exactly the same time. Presence of limited information or mild assumptions about the nature of target implementation would increase the timing resolution during model execution. For example, if one assumes that tasks are going to be implemented as software modules running on parallel processors, a sequential order would have to be imposed on the production and consumption of messages. This breaks the pessimistic simultaneous message production and consumption that is dictated by the operational semantics, and potentially leads to more accurate analysis.

Specifically, we utilize the state of the art implementation-oblivious buffer-throughput trade-off analysis technique developed by Stuijk et al. [22], and argue that in presence of very mild MPSoC target platform assumptions, its character-

LCTES'15, June 18 - 19, 2015, Portland, OR, USA
© ACM. ISBN 978-1-4503-3257-6// $15.00
http://dx.doi.org/10.1145/2670529.2754968

[1] Contact author: Matin Hashemi, matin@sharif.edu

ization of buffer-throughput trade-off is overly pessimistic. We propose transformations to the application SDF model to capture the sequential nature of message production and consumption by software, and to rigorously embed implementation awareness into the model. Subsequently, we leverage the method of Stuijk et al. to characterize the implementation space of the transformed model. The additional information that we expose to our analysis algorithm are quite limited in nature: merely sequential order between production and consumption of messages, which is implied by the assumption of implementation as software. As such, the analysis is not tied to the *details* of target execution platform, and would complement, rather than contradict with, the model-based design paradigm.

Experiments with a number of streaming applications show that implementation-awareness yields substantially more accurate buffer sizes (9X smaller on average) for the same throughput, compared to the conservative implementation-oblivious analysis. Moreover, the proposed implementation-aware trade-off analysis yields sufficiently accurate estimations in significantly shorter time, compared to precise implementation simulations. In comparison to cycle accurate simulation of a specific MPSoC implementation, the error of the proposed model-analysis method in throughput estimation is merely 19%, while it runs over 100X faster. The high degree of throughput estimation accuracy and substantial savings in runtime are due to the fact that the proposed approach considers only a relevant piece of information from target implementations, 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 logical point-to-point FIFO *channel* from actor u to v. Actors communicate by sending and receiving messages, also known as *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 the target 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, with respect to the tokens consumed by v. Data production and consumption rates are specified statically, and application execution is meant to continue indefinitely [12, 22].

Execution (Firing) Condition: Actor v can execute, also known as fire, at time t, if and only if (I) previous firings of v have completed[2], and (II) enough tokens are available on all of 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 at time t.

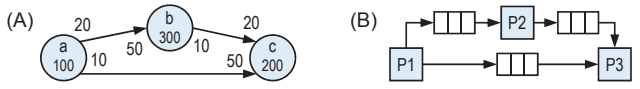

Figure 1. A) Example SDF graph (actors and channels are annotated with execution times and data rates, respectively.) B) An implied implementation of self-timed execution.

SDF Operational Semantics: Upon scheduling of actor v for execution, 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 $vw \in Out(v)$. Figure 1.A shows an example in which, $\varepsilon(b) = 300$, $r_c(ab) = 50$ 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 written to 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 [24] and TILE64 static network [3]. Some other platforms implement the abstract view via circular arrays that are allocated in the shared memory, using proper producer-consumer synchronization schemes. Regardless and for sake of our discussion, the platform can be abstractly viewed as a multiprocessor with a FIFO interconnection network.

We focus on self-timed execution, which implicitly assumes allocation of dedicated execution resources to every actor (Figure 1.B). Under self-timed execution, an actor fires as soon as its firing conditions are satisfied [22]. In many cases, an embedded application is developed on an MPSoC target by splitting the application into many actors, and assigning each actor to its dedicated core (e.g., 1080p H.264 encoder on AsAP [24]). Otherwise, the collection of actors allocated to the same processor under static schedule can be viewed as a coarse-grain actor in an upscaled version of the graph that conforms to our model.

2.3 Buffer-Throughput Tradeoff

Throughput[3] is one of the most important quality metrics in streaming applications. A number of factors, such as actor execution times, actor allocation and scheduling on processor cores, interprocessor buffer capacities, SDF graph structure and SDF graph cycles impact steady-state throughput [6, 8–10, 22].

[2] Auto-concurrency, i.e., multiple concurrent firings of an actor, is not allowed in our discussion.

[3] In this paper, we use the terms "steady-state throughput" and "throughput" interchangeably.

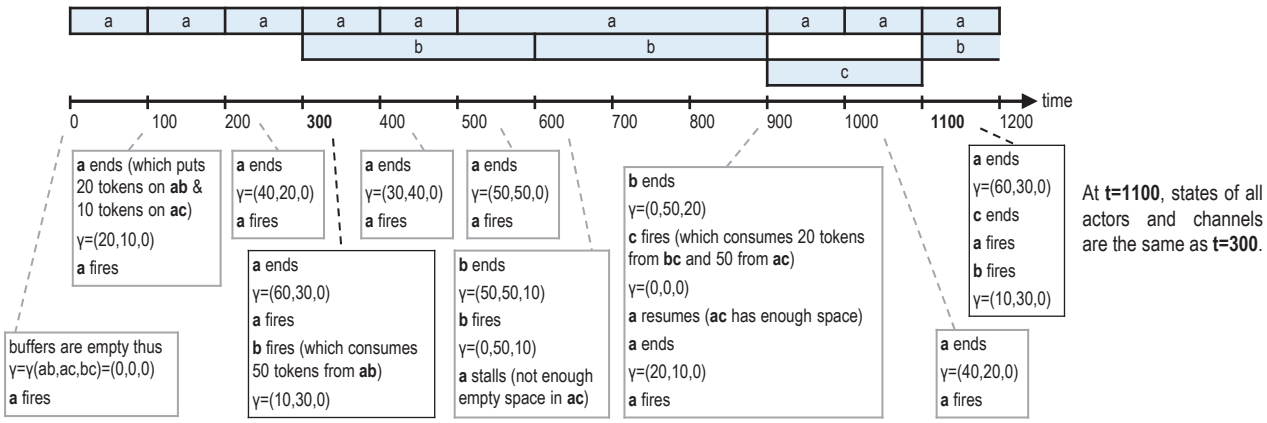

Figure 2. Throughput Analysis based on SDF operational semantics when $\beta(ab, ac, bc) = (60, 50, 20)$.

In practice, the FIFO channels must be implemented with finite buffering capacity, which may limit the throughput [22]. 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 capacities.

Throughput: Throughput of an actor v is defined as the average number of v firings per unit time [6], i.e., $\tau(v) = \lim_{T\to\infty} \frac{1}{T} \times \big(\text{number of } v \text{ firings from } t = 0 \text{ to } t = T\big)$. 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 of the SDF application [6, 12]. 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, $\beta(uv)$ limits 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, u 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.

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 $t_2 - t_1 \leq \varepsilon(u)$ and $\exists uv \in Out(u) : \beta(uv) - \gamma(uv, t_2) < r_p(uv)$. Actor u 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 [22]. The algorithm works by executing the SDF graph while keeping track of 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 a judiciously selected subset of buffer size allocations in order to evaluate all Pareto points [22]. 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 of all actors and number of tokens stored on all 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, deadlock happens and overall throughput τ becomes zero. By analyzing SDF operational semantics, Ade et al. [1] proved the following theorem regarding deadlocks.

Theorem 2.1 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. We denote the above formula with $\beta_{min}(uv)$. In our example, $\beta_{min}(ab, ac, bc) = (60, 50, 20)$.

Note that the theorem does not make any statement if "for all" channels $uv \in E$, $\beta(uv) \geq \beta_{min}(uv)$. In such a case, more thorough deadlock analysis is required [25].

3. Implementation-Aware Buffer-Throughput Analysis

We propose taking into account a key piece of information about target MPSoC implementations by which, buffer-throughput tradeoff analysis would become more accurate. In our discussion, we adopt the following abstract view of implementation for an application modeled as a SDF graph.

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 specific computation is executed, and finally, the generated data is written to output buffers. This sequence is repeated indefinitely. Let us 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 data, such as a video frame, as a single token. However, interconnect networks have limited bandwidth and they are 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, which have to be transferred sequentially as shown in the inner loop of 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 sequential order.

3.2 Implications of Implementation-Awareness

In practice, when SDF graph is implemented in a form that conforms to our abstract implementation, the simultaneity in reading and writing tokens at arbitrary rates is not faithfully implemented. 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 about implementations leads to an operation that is quite different from the pure SDF model, in which actors write to (read from) all channels simultaneously at specified 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 b and upon availability of sufficient number of tokens produced by b, actor c fires and immediately consumes all of them.

(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);
}
```

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

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 bc, 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 implications of software implementation in buffer-throughput analysis.

3.3 Implementation-Aware SDF Graph Transformation

We take a two step approach to bring implementation-awareness into characterization of buffer-throughput tradeoff. First, we transform the original SDF graph G by embedding implementation-dictated sequential data production and consumption into the graph. Clearly, the transformation must preserve the function and other relevant aspects of the original application. Subsequently, the transformed SDF graph G' is analyzed by leveraging an implementation-oblivious technique, described in Section 2.4, to obtain its buffer-throughput Pareto 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). Our proposed SDF graph transformation models these two properties by adding *virtual* actors and channels to the SDF graph. Specifically, 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 W feeds data into the input of 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 production and consumption rates of 1. Thus, for every firing of u, W has to fire $r_p(uv)$ times sequentially to consume the tokens produced by u one at a time. Recall that auto-concurrency is disallowed in our discussion. Similarly, for every $r_c(uv)$ firings of R, actor v fires once. Buffer sizes for 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).

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 $R_1, R_2, \ldots R_{|In(v)|}$, and $|Out(v)|$ writer actors $W_1, W_2, \ldots W_{|Out(v)|}$. Data production ($r_p$) and consumption rates (r_c), and buffer sizes (β) 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 G'. 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 all of the input 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. Note that input channels of G_v have consumption rates of 1 because they are connected to reader actors. Similarly, output channels of G_v have production rates of 1. Thus, subgraph G_v models the execution of task v based on the implementation view discussed in Section 3.1.

Theorem 3.1 *Addition of reader and writer actors preserves SDF functionality.*

Proof: SDF functionality is independent of task execution order (scheduling), and merely depends on the value and order of data tokens in channels [5]. Both reader and writer actors have the identity transfer function and do not alter data. Moreover, they preserve the order of data tokens delivered from the producer to the consumer. Therefore, the end to end functionality of the SDF graph remains intact. □

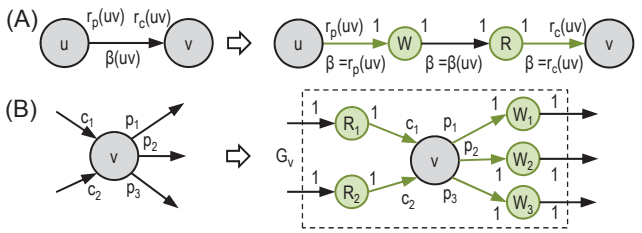

Figure 4. A) Writer & reader actors for channel $uv \in E$. Virtual actors and channels are shown in green. B) The transformed subgraph G_v for an actor v with 2 incoming and 3 outgoing channels.

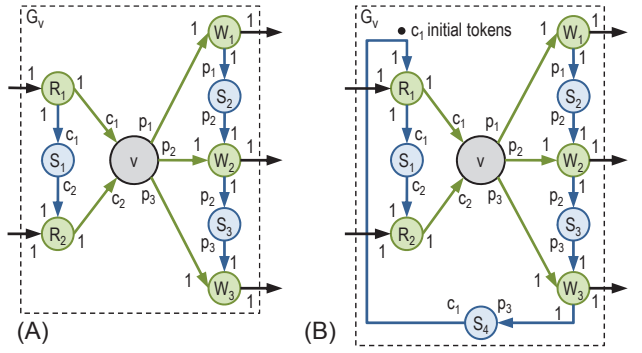

Figure 5. A) Sync actors S_1, S_2 and S_3 enforce the sequential order $R_1, R_2, v, W_1, W_2, W_3$ in subgraph G_v of Figure 4.B. The newly added virtual actors & channels are shown in blue. B) Sync actor S_4 prohibits auto-concurrency.

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 data consumption, computation and data production based on the abstract implementation view, we need to eliminate the simultaneity. Our approach is to 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).

Specifically, 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 production (r_p) and consumption (r_c) rates, and buffer size (β) of the newly added virtual channels (marked blue in the figure) as follows:

virtual channel	r_p	r_c	β
$R_iS^R_{i,i+1}$	1	c_i	c_i
$S^R_{i,i+1}R_{i+1}$	c_{i+1}	1	c_{i+1}
$W_jS^W_{j,j+1}$	1	p_j	p_j
$S^W_{j,j+1}W_{j+1}$	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. By creating appropriate dependencies, the construction ensures that the desired ordering is enforced.

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 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 have 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). 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.

Theorem 3.2 *Addition of sync actors preserves the original SDF functionality.*

Proof: By construction, read and write actors do not mix up application data tokens (green channels) with synchronization tokens (blue channels). The application functionality merely depends on the data values and their ordering on green channels, which is isolated from sync actors. As such, sync actors have no impact on original SDF functionality. □

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 subgraph 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 3.3 *Assume that the set of actors in G_v in the transformed graph and actor v in the graph G start execution at the same time and from the same buffer state in G. In that case, if G_v stalls during execution, the corresponding actor v in the graph G must also stall.*

Proof: Stalling execution occurs because at least one channel does not have enough capacity to receive the produced tokens at that point in time. By construction, stalling G_v implies that at least one of output channels of the write actors does not have sufficient capacity during execution of G_v, as channels internal to G_v are allocated sufficient capacity to execute G_v. Since production rate of write actors is one (the smallest possible rate), at least one of the output channels of G_v must be entirely full. Execution of G_v takes exactly the same time as execution of v, and thus, the corresponding output channel of v in G is also full during execution of v, which would stall the execution of v. □

The following theorem articulates the pessimistic nature of implementation-oblivious analysis.

Theorem 3.4 *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.*

Proof: Assume that the theorem does not hold. Then, there must be a time t at which, *for the first time* the execution of an actor v in G starts earlier than the corresponding execution of the graph G_v in the transformed graph. Execution of v at t implies that both conditions I and II discussed in Section 2.1 are satisfied at t.

Condition I indicates that no other firing of v is stalled. Based on theorem 3.3 there cannot be a stalled version of G_v, since t is the first point in time at which, G supposedly runs ahead of the transformed graph. Hence, condition I is also satisfied for G_v. Condition II indicates that all of the input channels of v have sufficient number of tokens available at time t. Input channels of v form input channels of the read actors in G_v. Moreover, the consumption rate of read actors is one, which is the smallest possible valid rate. Thus, there must be sufficient number of tokens for read actors of G_v at time t, and its execution should not be stalled. The contradiction proves the theorem. □

As one would expect, the two original and transformed graphs should yield the same throughput if buffer capacity constraint is relaxed.

Theorem 3.5 *The maximum throughput of G and G', which is obtained when all channels of G have infinite buffer size, are equal.*

Proof: If buffer sizes are sufficiently large, throughput would be limited by the slowest actor or the iteration bound of the corresponding "homogeneous" SDF graph (HSDF). The iteration bound of an HSDF graph is equal to its maximum cycle mean, which is defined as the cycle latency divided by the number of initial tokens in the cycle [6, 17].

The transformation only adds actors with zero execution time to the graph, and hence, the slowest actor would have the same execution time in both graphs. The transformation creates cycles within the subgraph G_v, however, all such cycles have latency of $\varepsilon(v)$. There is at least one initial token in all cycles inside G_v, as the feedback edge from the last writer actor to the first reader actor must be part of the cycle. Thus, the cycle mean for cycles that are created inside G_v is not more than $\varepsilon(v)$, which would not limit the throughput. Finally, for cycles in the transformed graph

that are not inside G_v, neither the cycle latency nor the number of initial tokens in the cycle are changed under the transformation, and hence, the two graphs will have the same limit throughput. □

3.5 Related Work

Many previous analysis algorithms are solely based on SDF operational semantics [2, 6, 15, 22]. To increase accuracy in throughput analysis, Moonen et al. [14] 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 task computation.

Oh and Ha [16] 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.

Cycle-accurate simulation of MPSoC platforms is accurate but takes very long, even using multicore processors or GPGPU [13, 19] mainly because it considers almost every architectural detail. Our proposed technique, however, incorporates limited information on the target implementation in order to provide high accuracy without the often prohibitive runtime of cycle-accurate simulations or the need to use many computing resources for simulation acceleration.

4. Experiments

4.1 Setup and benchmark applications

To evaluate the proposed technique we employ StreamIt benchmark applications. StreamIt is a programming language and compiler for stream programs [23]. For every benchmark application, we execute StreamIt compiler (Figure 6, top left) and then 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 source code such that its CPI estimates match Graphite processor model [7]. Graphite is a cycle-accurate MPSoC simulator, and is used as the target platform in our experimentation.

The above procedure yields a series of SDF graphs which are used as benchmarks in our experimentation. We have released the generated SDF graphs along with details of the above procedure on the web [4].

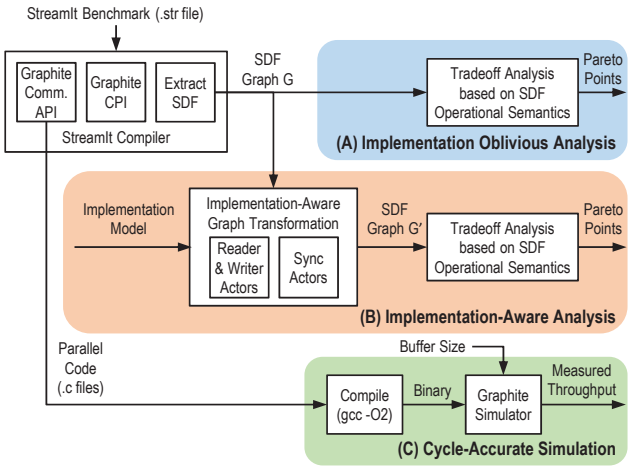

Figure 6. Experimentation flow: A) Baseline implementation-oblivious buffer-throughput tradeoff analysis based on SDF operational semantics. B) Proposed implementation-aware analysis. C) Cycle-accurate simulation of the compiled binary code.

4.2 Implementation-aware vs. implementation-oblivious analysis

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 another SDF graph 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 (sequentially-ordered read/write operations) into SDF 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 [21, 22], which implements the tradeoff analysis algorithm explained in Section 2.4. We 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 total interprocessor buffer size, $|\beta|$, and corresponding overall throughput, τ. To compare the proposed approach against an established standard, we also perform the implementation-oblivious 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 3.5, since both approaches always result in the same maximum throughput.

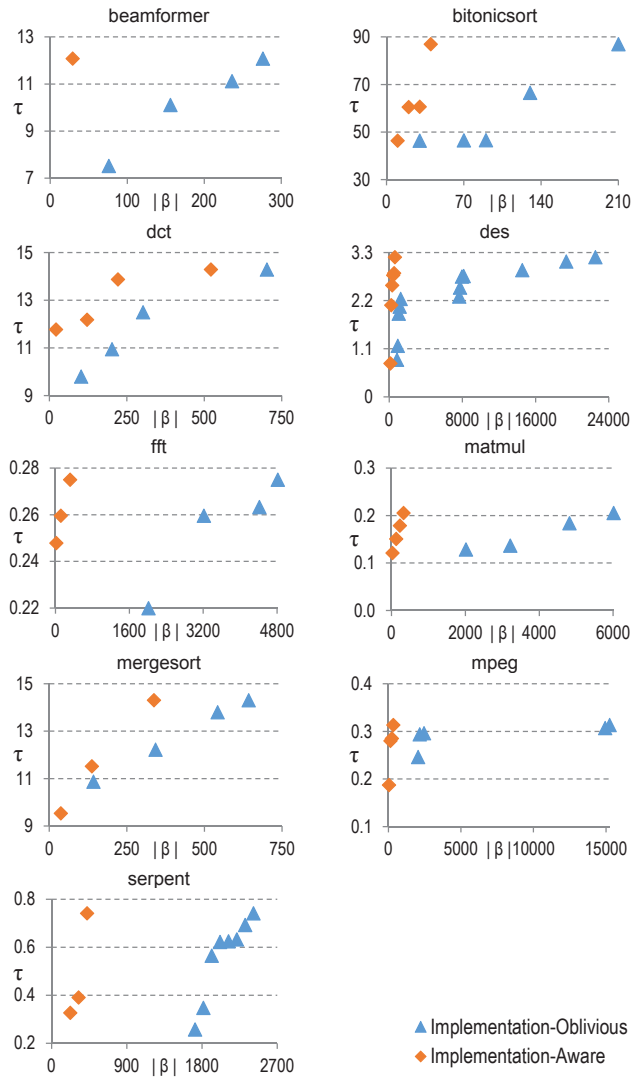

Figure 7. Pareto points between total interprocessor buffer size, $|\beta|$, and the corresponding throughput, τ, for both the baseline implementation-oblivious and the proposed implementation-aware tradeoff analysis techniques. The proposed method yields substantially improved buffer size estimates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

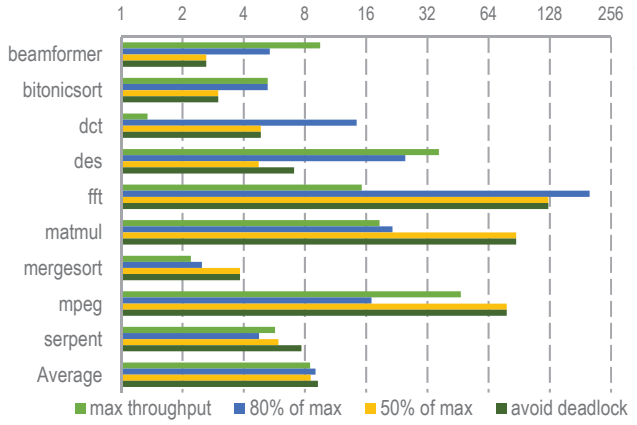

Figure 8. Reduction in total buffer size estimates, i.e., the implementation oblivious over implementation aware ratio of total buffer size, $|\beta|$, required to achieve the maximum throughput, 80% of the maximum throughput, 50% of the maximum throughput, and to avoid deadlock. X-axis shows the ratio in base 2 logarithmic scale.

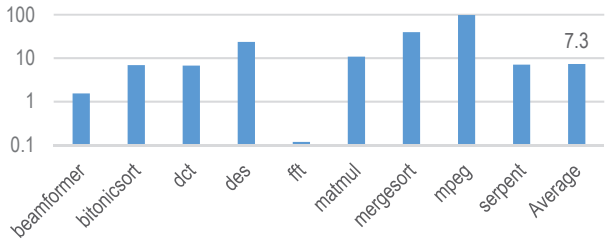

Figure 9. Runtime of implementation aware over implementation oblivious analysis.

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 46X smaller.

Figure 8 highlights the substantial reduction in total buffer size requirement, using the same data of Figure 7. The horizontal axis is in logarithmic scale (base 2) 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% 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 how the increase in complexity of the model translates into an increase in the runtime of the analysis. Specifically, it 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.

4.3 Comparison against cycle accurate simulation

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 [7] cycle-accurate simulator (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[4]. 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 estimated by implementation aware and implementation oblivious analysis techniques for the selected buffer size distribution, against cycle-accurate simulated throughput. The numbers are normalized with respect to the throughput given by Graphite cycle-accurate simulator. 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 technique 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.

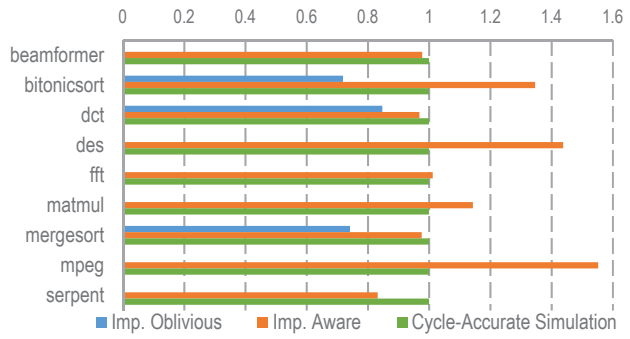

Figure 10. Comparison of (normalized) throughput estimated by implementation aware and implementation oblivious techniques against cycle-accurate simulation. Implementation oblivious technique inaccurately predicts deadlock in most cases, and is less accurate in the remaining cases.

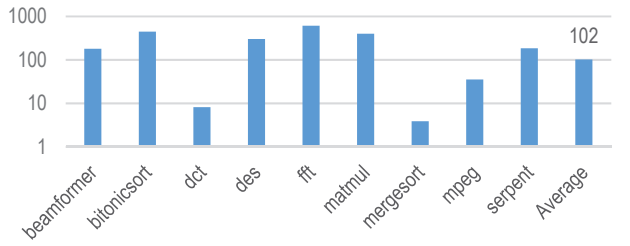

Figure 11. Runtime of cycle-accurate simulation over the proposed implementation aware analysis technique.

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 estimates 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 proposed 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 details, such as processors' instruction set, cache and program control flow.

5. Conclusion

Restricting the analysis to SDF operational semantics and ignoring implications of software implementation are likely to yield inaccurate conclusions. In particular, we investigated the trade-off between buffer size requirement and through-

[4] We also experimented with SDF3 benchmarks in Section 4.2. 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 limits. For SDF3 benchmarks, on average, buffer size reduction using implementation-aware analysis is 6X, and runtime ratio of implementation-aware over implementation-oblivious is 5X.

put 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. Consequently, we proposed an implementation-aware buffer-throughput trade-off analysis algorithm. Experimental results show that model analysis solely based on SDF operational semantics yields much more pessimistic buffer sizes than is actually required to achieve a desired level of throughput. Moreover, implementation-aware SDF model analysis yields sufficiently accurate throughput estimates, in comparison to cycle accurate simulation of target implementations, while running two orders of magnitude faster. Thus, we have made a strong case for identification of key relevant platform information, and integrating them into high-level model-based analysis techniques.

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] Benchmarks. `http://sharif.edu/~matin` and `http://leps.ece.ucdavis.edu`.

[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Springer, 1996. ISBN 1461286018.

[6] A. H. Ghamarian et al. Throughput analysis of synchronous data flow graphs. *International Conference on Application of Concurrency to System Design*, 2006.

[7] Graphite. `http://graphite.csail.mit.edu`.

[8] M. Hashemi and S. Ghiasi. Versatile task assignment for heterogeneous soft dual-processor platforms. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 29(3), 2010.

[9] M. Hashemi, M. H. Foroozannejad, S. Ghiasi, and C. Etzel. Formless: Scalable utilization of embedded manycores in streaming applications. *International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, pages 71–78, 2012.

[10] M. Hashemi, M. H. Foroozannejad, and S. Ghiasi. Throughput-memory footprint trade-off in synthesis of streaming software on embedded multiprocessors. *ACM Transactions on Embedded Computing Systems*, 13(3), 2013.

[11] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[12] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 1987.

[13] J. Miller et al. Graphite: A distributed parallel simulator for multicores. *International Symposium on High-Performance Computer Architecture*, January 2010.

[14] 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.

[15] O. M. Moreira and M. J. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007.

[16] 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.

[17] K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. Wiley-Interscience, 2008. ISBN B000UGR930.

[18] A. Pinto, A. Bonivento, A. L. Sangiovanni-Vincentelli, R. Passerone, and M. Sgroi. System level design paradigms: Platform-based design and communication synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 11 (3):537–563, 2006.

[19] S. Raghav, A. Marongiu, C. Pinto, M. Ruggiero, D. Atienza Alonso, and L. Benini. SIMinG-1k: A thousand-core simulator running on GPGPUs. *Concurrency and Computation: Practice and Experience*, 25(10):1443–1461, 2013.

[20] A. Sangiovanni-Vincentelli and G. Martin. A vision for embedded systems: platform-based design and software methodology. *Design Test of Computers*, 18(6):23 –33, 2001.

[21] SDF3. `http://www.es.ele.tue.nl/sdf3`.

[22] S. Stuijk et al. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. *Design Automation Conference*, 2006.

[23] W. Thies et al. Streamit: A language for streaming applications. *International Conference on Compiler Construction*, 2002.

[24] 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.

[25] Y. Zhou and E. A. Lee. A causality interface for deadlock analysis in dataflow. *International Conference on Embedded Software*, pages 44–52, 2006.