Symbolic computation of latency for dataflow graphs

(abstract)

Adnan Bouakaz Pascal Fradet Alain Girault

INRIA; Univ. Grenoble Alpes

first.last@inria.fr

I. Motivation

We present the symbolic computation of data-flow graphs latency, with two variants: the multi-iteration latency and the input-output latency. These are important timing constraints that are usually used in the design of real-time control systems. The input-output latency is particularly useful for real-time control systems since it is the maximum delay between sampling data from sensors and sending control commands to the actuators.

Latency analysis can either be performed at compile time, for design space exploration, or at run-time, for resource management and reconfigurable systems. However, this analysis has an exponential time complexity, which may cause a huge run-time overhead or make design space exploration unacceptably slow. We propose to compute the latency *symbolically*, i.e., as a function of parameters of the given data-flow graph. By parameters, we mean the input and output rates of the data-flow actors, as well as their execution times. Such functions can be quickly evaluated for each different configuration or checked w.r.t. different quality-of-service requirements.

II. Definitions and basic notions

We are given a data-flow graph G made of edges of the form $A \xrightarrow{p \quad q} B$ with two *actors* A and B such that A produces p tokens each time it *fires* while B consumes p tokens each time it fires. Besides, we are given the execution time t_A of A and t_B of B. Essentially, G is a parameterized version of an SDF graph [7] where rates and execution times can be formal parameters. As with any SDF graph, we can solve the system of balance equations to find the iteration of the graph [7]. *E.g.,* for the simple graph $A \xrightarrow{p \quad q} B$, there is a single balance equation $z_A p = z_B q$, where z_X denotes the number of firings of actor X in the iteration.

The repetition vector of this simple graph is:

$$[z_A = q/\gcd(p,q), z_B = p/\gcd(p,q)]$$

Finally, the *load* imposed by actor X is the product $z_X t_X$.

In this work, we focus on as soon as possible (ASAP) scheduling of consistent graphs without auto-concurrency. In such self-timed executions, an actor fires as soon as it becomes idle (no auto-concurrency) and has enough tokens on its input channels. We assume that there are sufficient processing units, e.g., there are as many processors as actors or all actors are implemented in hardware. ASAP scheduling allows the graph to reach its maximal throughput. Such schedules are naturally pipelined and composed of a prologue followed by a steady state that repeats infinitely.

The *multi-iteration latency* $\mathcal{L}_G(n)$ of the first n iterations of a graph G is equal to the finish time of the last firing of its first n iterations (time is counted from the very first firing).

The *period* $\mathcal{P}_G$ of the execution of a graph G is the average length of an iteration, formally defined as:

$$\mathcal{P}_G = \lim_{n \to \infty} \frac{\mathcal{L}_G(n)}{n} \quad (1)$$

The *input-output latency* $\ell_G(n)$ of the n^{th} iteration of a graph G is equal to the time between the start time of the first firing and the finish time of the last firing of the n^{th} iteration. The definition given in [6] is slightly different but in our context (graphs with initially empty channels) the two definitions are equivalent. The input-output latency of the complete execution ℓ_G is defined as the maximal latency over all iterations:

$$\ell_G = \max_{n=1..\infty} \ell_G(n) \quad (2)$$

III. Symbolic computation

We first derive analytic formulas for the multi-iteration latency of the first n iterations (i.e., $\mathcal{L}_G(n)$) of graph $G = A \xrightarrow{p \quad q} B$. Since we are interested in the (approximation of) the minimum achievable latency, we assume that buffers are unbounded. There are two cases depending on whether A or B imposes the highest load.

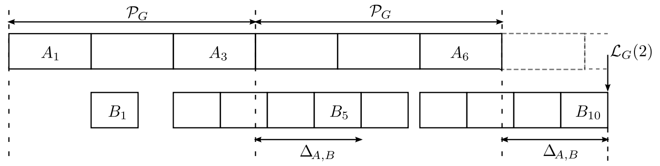

Fig. 1: ASAP schedule and multi-iteration latency $\mathcal{L}_G(2)$ of graph $G = A \xrightarrow{p \quad q} B$ in the case $z_A t_A \geq z_B t_B$ ($p = 5$, $q = 3$, $t_A = 14$, $t_B = 8$). Each box represents the firing and execution time of one actor

• **Case** $z_A t_A \geq z_B t_B$, *i.e., A* imposes a higher load than B. As illustrated in Fig. 1, actor A never gets idle and $\mathcal{P}_G = z_A t_A$. Therefore, we have:

$$\mathcal{L}_G(n) = n\mathcal{P}_G + \Delta_{A,B} \quad (3)$$

such that $\Delta_{A,B}$ is the remaining execution time for actor B after actor A has finished its firings of the n^{th} iteration ($\Delta_{A,B}$ is constant over all iterations). The formulas for $\Delta_{A,B}$ can be found in [2].

• **Case** $z_A t_A < z_B t_B$: see [2].

For a chain $A \xrightarrow{p_1 \quad q_1} B \xrightarrow{p_2 \quad q_2} C \to \cdots \to Z$ of actors, we compute an *upper bound* of the multi-iteration latency of the first n iterations, denoted $\hat{\mathcal{L}}_{A \to Z}$ (we omit n for the sake of conciseness). We first compute exactly $\mathcal{L}_{A \to B}$. However, since this computation assumes that the producer can run consecutively, it cannot be applied between B and C.

We compute an upper bound linearization of the firings of B such that they are consecutive and $\forall j \leq nz_B.\ f_{B^u}(j) \geq f_B(j)$ (details are in [2]). The intuition is to transform each actor for which the execution has gaps (*e.g.,* B in Fig. 1) into a virtual actor having the same load but without any gap, so that we can compute the latency for each edge of the chain.

We actually use two upper bound linearization methods to make the firings of B consecutive: **(i)** The *Push* method pushes *all* firings of B to the right end to get rid of all the gaps; **(ii)** The *Stretch* method increases the execution time of B in order to fill the gaps over an infinite execution. They are incomparable and there are graphs for which either *Push* or *Stretch* is better. Since the two methods are not costly to try, we apply both and take the minimum.

For a general acyclic SDF graph G, we represent it as a set of *maximal chains* $\mathcal{G}(G)$, that is, chains from a source actor to a sink actor. We then compute the multi-iteration latency of each such chain g, and we finally have: $\mathcal{L}_G(n) = \max_{g \in \mathcal{G}(G)} \{\mathcal{L}_g(n)\}$.

Regarding the input-output latency, we can compute the maximum input-output latency of the n^{th} iteration of a chain G, denoted $\ell_G(n)$, from its multi-iteration latency: $\ell_G(n) = \mathcal{L}_G(n)$ minus the start time of the first firing of the source actor in the n^{th} iteration. If the source actor A imposes the highest load among all actors of the graph or if all the channels are unbounded, then the source actor never gets idle and achieves the maximal throughput (otherwise, buffer sizes must be taken into account, see [2]). It follows that:

$$\ell_G(n) = \mathcal{L}_G(n) - (n-1) z_A t_A \tag{4}$$

For an arbitrary chain G, we also make use of a backward linearization technique to compute a safe upper bound of ℓ_G.

IV. Results

We have evaluated our approach for computing the multi-iteration latency using millions of randomly generated chains. The experiments show that the average over-approximation is negligible when the number of firings per iteration of the graph is small. Indeed, if there are many harmonious rates (recall that, when p divides q or q divides p, the computed latency for $A \xrightarrow{p \quad q} B$ is exact), then the computed latency remains close to the exact value. Then, the average over-approximation increases to reach its peak (approximately 2.5%) at around fifty firings per iteration. This is because the exact values of latency at these points are small and hence the over-approximation is more noticeable. Then, the average over-approximation decreases and converges to zero for graphs with large latencies.

Table I presents our results for five real applications: the H.263 decoder, the data modem and sample rate converter from the SDF^3 benchmarks [8], the Fast Fourier Transform (FFT), and the time delay equalizer (TDE) from the StreamIt benchmarks [9]. All these graphs have a chain structure. Table I shows that our approach gives exact results for most of these benchmarks. Production and consumption rates of channels of these graphs are quite harmonious (p divides q or q divides p), for which our approach performs very well.

Finally, we evaluate our approach for computing the input-output latency using 10^5 randomly generated chains. The experiment shows that our analysis over-approximates the exact computation, on average, by at most 13%. The over-approximation is less noticeable for graphs with large input-output latencies.

TABLE I: Multi-iteration latency computation for real benchmarks.

graph	$\mathcal{P}_G$	$\mathcal{L}_G(1)$	$\hat{\mathcal{L}}_G(1)/\mathcal{L}_G(1)$	$\hat{\mathcal{L}}_G(2)/\mathcal{L}_G(2)$
(a) modem	32	62	1	1
(b) sample con.	960	1000	1.022	1.011
(c) H.263 dec.	332046	369508	1	1
(d) FFT	78844	94229	1	1
(e) TDE	17740800	19314069	1	1

V. Related work

Few symbolic results about SDF graphs can be found in the literature. The results reported here and in [2], [3] on symbolic latency are the first of their kind.

For the symbolic throughput computation, [4] consider the *token timestamp vector* $\vec{s}_i$, where each entry corresponds to the production time of tokens in the i^{th} iteration of the graph. Then, the authors use the max-plus algebra to express the evolution of the token timestamp vector: $\vec{s}_i = M\vec{s}_{i-1}$. They have proved that the eigenvalue of matrix M is equal to the period of the graph.

[5] presents a parametric throughput analysis for SDF graphs with *bounded* parametric execution times of actors but constant rates. Since rates and delays are non-parametric, the SDF-to-HSDF transformation is possible and the throughput analysis is based on the MCM of the resulting HSDF graph.

A different analytic approach to estimate lower bounds of the maximum throughput is to compute strictly periodic schedules instead of ASAP schedules (*e.g.,* [1]). This approach is similar to our *Stretch* linearization method to compute the latency of the graph. We have however found that using both *Push* and *Stretch* methods usually gives better results.

References

[1] B. Bodin, A. Munier-Kordon, and B. de Dinechin. Periodic schedules for cyclo-static dataflow. In *Symposium on Embedded Systems for Real-time Multimedia*, pages 105–114, 2013.

[2] A. Bouakaz, P. Fradet, and A. Girault. Symbolic analysis of dataflow graphs (extended version). Technical Report 8742, INRIA, 2016.

[3] A. Bouakaz, P. Fradet, and A. Girault. Symbolic buffer sizing for throughput-optimal scheduling of dataflow graphs. In *Proceedings of the 2016 IEEE 22nd Real-Time and Embedded Technology and Applications Symposium*, 2016.

[4] M. Geilen. Synchronous dataflow scenarios. *ACM Trans. Embed. Comput. Syst.*, 10(2):16:1–16:31, 2011.

[5] A. H. Ghamarian, M. C. W. Geilen, T. Basten, and S. Stuikj. Parametric throughput analysis of synchronous data flow graphs. In *Conf. on Design, Automation and Test in Europe*, pages 116–121, 2008.

[6] A. H. Ghamarian, S. Stuijk, T. Basten, M. C. W. Geilen, and B. D. Theelen. Latency minimization for synchronous data flow graphs. In *Euromicro Conf. on Digital System Design Architectures, Methods and Tools*, pages 189–196, 2007.

[7] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, pages 1235–1245, 1987.

[8] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF for free. In *Int. Conf. on Application of Concurrency to System Design*, pages 276–278, 2006.

[9] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for languages and compiler design. In *Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 365–376, 2010.