2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems

An approximate method for throughput evaluation of cyclo-static dataflow programs

Pascal Aubry, Mohamed Benazouz, Renaud Sirdey
CEA, LIST,
Embedded Real Time Systems Laboratory,
Point Courrier 94, 91191 Gif-sur-Yvette Cedex, France
Email : {p.aubry, mohamed.benazouz, renaud.sirdey}@cea.fr

***Index Terms*—many-core, dataflow, throughput evaluation, buffer sizing**

***Abstract*—Because of the multiplication of multi-core architecture, dataflow programming languages regained interest during the last years. In the case of massively multi-core embedded system architectures, the computation of throughput is used for buffer sizing under time constraints. This paper introduces an approximate method for the throughput evaluation for cyclo-static dataflow graphs.**

I. Introduction

With the development of massively multi-core architectures, the dataflow paradigm regains popularity. This programming model represents concurrent processes communicating through buffered-channels. In the case of embedded applications, the management of different resources is primordial. The reason is that these applications have a number of real-time constraints that must be satisfied. One of these is the throughput depending on the sizes of channel buffers and on the execution times.

The most known dataflow model is the Synchronous DataFlow Graph (SDFG) [9]. The processes are modeled by nodes (tasks) which communicate via FIFO-channels (First-In-First-Out). At each firing, a task produces (respectively consumes) the same amount of data for each input (respectively output) channel.

This model has been extended by the Cyclo Static DataFlow Graph (CSDFG) [4]. In this model, each task is executed repeatedly a finite number of occurrences. A task fires if the data in input is sufficient. At each occurrence a different quantity is produced/consumed on each input/output channel.

In this paper we propose an approximate method to compute throughput for CSDFGs. Several measures for the throughput can be defined. The most used one is the maximum throughput, which can be deduced from an “as soon as possible” (ASAP) execution. This method consists in searching for a periodic pattern in the execution of the CSDF that can be repeated an infinitely large number of times after a first transient phase [7]. The maximum throughput can be computed using the number of executions of different tasks in this periodical phase. The size of the periodic pattern depends on consumption/production values and can be exponential. For large CSDF systems this method cannot provide a result in a reasonable time. That is why it is important to find a good approximation for the maximum throughput. Another method for finding the maximum throughput is to transform the CSDFG into a Homogeneous Synchronous DataFlow Graph (HSDFG) [4] and apply the Maximum Cycle Mean algorithm (MCM) [5]. The main problem of this solution is the exponential size of the generated graph and its impact on the running time of the MCM algorithm. Even for small CSDFGs, the size of the HSDFG can be prohibitively large.

In this paper, we introduce a new method, based on self timed execution, for finding an approximate value of the throughput. This method provides a good alternative in the case when the research of the pattern is complex and time consuming. The purpose of throughput evaluation is to compute a sizing for buffers.

The paper is organized as follows: the CSDFG model and the employed notations are presented in Section 2. The model for finding the self timed execution is presented in Section 3. We introduce the evaluation of the throughput in Section 4. In Section 5 we present the experimental results we have obtained and, finally, Section 6 contains our conclusions and future works.

II. Cyclo-static Dataflow Graph

Let us define a Cyclo-Static DataFlow Graph as a directed graph $\mathcal{G} = (T, A)$ where T is the set of tasks and A is the set of communication links between different tasks. Every task $t \in T$ is defined by a succession of transitions that are denoted by $\tau^k(t)$, $k \in \{0, \ldots, n_\tau(t) - 1\}$, where $n_\tau(t) \in \mathbb{N}^*$. Each transition of a task is executed cyclically every $n_\tau(t)$ occurrences of the task. A task is said to be executed when it finishes a cycle of transitions. We denote by $\gamma \in \mathbb{N}^{|T|}$ the repetition vector of the application. It assures that the system comes back to its initial state (in terms of token number on links) if each task is executed exactly γ_t times. The set of executions for all tasks is called a cycle of execution of the system.

A task can produce or consume on different links. For each task, we define $P_t \in A$ (respectively $C_t \in A$) the set of links for which t is a producer (respectively consumer), with $P_t \cap C_t = \emptyset$. For each link $l \in P_t$ (respectively $l \in C_t$) and for each transition $\tau^k(t)$ we associate a quantity $qp_l(k)$ (respectively $qc_l(k)$) of data tokens produced (respectively consumed) by this transition.

For each link $l \in A$ we define $qp^i(l)$ (respectively $qc^i(l)$) the total quantity produced (respectively consumed) by the link l at the end of the i^{th} ($i \in \mathbb{N}$) occurrence of the producer (respectively consumer). This production (respectively consumption) is done by the task $p(l) \in T$ (respectively by $c(l) \in T$). It exists exactly one producer and one consumer by link. If $p(l)$ occurred i times, i.e. $p(l)$ has been executed $n(p(l)) = \left\lfloor \frac{i}{n_\tau(p(l))} \right\rfloor$ times and the last executed transition is $\tau^k(t)$ such as $k = i\%n_\tau(p(l))$ then $qp^i(l) = n(p(l)) \times \sum_{j=0}^{n_\tau(p(l))-1} qp_l(j) + \sum_{j=0}^{k} qp_l(j)$. In a similar way, we can deduce the equation for the data consumed by $c(l)$: $qc^i(l) = n(c(l)) \times \sum_{j=0}^{n_\tau(c(l))-1} qc_l(j) + \sum_{j=0}^{k} qc_l(j)$.

In the context of embedded massively multi-core architecture, we need to work within bounded memory. For each $l \in A$ we define the size of the buffer as $d_l \in \mathbb{N}$. The vector d is called the storage distribution of the CSDFG. Sometimes several buffers can contain data in the initial state of the system. This initial data is called preload. Let $q_0(l) \in \mathbb{N}$ be the preload of link l.

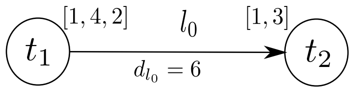


Figure 1. An example of CSDFG.

Figure 1 is a simple example for a CSDFG, composed of two tasks, t_1 and t_2, that share data via the link l_0. Task t_1(resp. t_2) has 3 transitions (resp. 2). The repetition vector for this application is $\gamma = [4, 7]$. The productions and consumptions for l_0 are $qp_{l_0} = [1, 4, 2]$ and $qc_{l_0} = [1, 3]$. The size for the buffer of l_0 is equal to the minimum size possible, i.e. $d_{l_0} = 6$. No data is present in the buffer at the beginning of the execution.

III. Computation of the self timed execution

The method introduced in this paper is based on an ASAP execution of multiple CSDFG execution cycles, which is called the self timed execution. For a given storage distribution, we aim to compute the throughput of the tasks and check if the target throughput is reached. Different constraints describing the execution of the system must be introduced in order to compute tasks throughputs. The set of these constraints defines a scheduling graph and gives an execution order.

For each task t, let us define $\alpha^i(t)$ (respectively $\beta^i(t)$) the start time (respectively end time) of the *i*th occurrence of t. For each task t, let $\varphi_t \in \mathbb{R}_+^{n_\tau(t)}$ be a vector with the execution times of transitions for task t.

Time consistency constraints

For each task t, the end time execution of an occurrence i is equal to the start time of this occurrence plus its execution time:

$$\beta^i(t) = \alpha^i(t) + \varphi_t(i\%n_\tau(t)), \forall t \in T, \forall i \in \mathbb{N}.$$

For each task t, its start time must be greater than the end time of the previous occurrence:

$$\beta^i(t) \leq \alpha^{i+1}(t), \forall t \in T, \forall i \in \mathbb{N}.$$

Production consistency constraints

For a link, the total quantity of stored tokens in the associated buffer must always be larger than the total quantity of consumed tokens. Thus, for each link l, if for the ith occurrence of the producer $p(l)$ and for the jth occurrence of the consumer $c(l)$ we have $qp^{i-1}(l) + q_0(l) < qc^j(l)$ and $qp^i(l) + q_0(l) \geq qc^j(l)$ then :

$$\beta^i(p(l)) \leq \alpha^j(c(l)), \forall i \in \mathbb{N}, \forall j \in \mathbb{N}.$$

Capacity constraints

For each link l, the quantity of data contained in its buffer must never exceed the buffer size, i.e. the difference between the total produced quantity and the total consumed quantity, at any time, must not be larger than the difference between the size of the buffer and its preload. Thus for each link l, if for the ith occurrence of the producer $p(l)$ and for any jth occurrence of the consumer $c(l)$ we have $qp^i(l) - qc^{j-1}(l) > d_l - q_0(l)$ and $qp^i(l) - qc^j(l) \leq d_l - q_0(l)$ then :

$$\beta^j(c(l)) \leq \alpha^i(p(l)), \forall i \in \mathbb{N}, \forall j \in \mathbb{N}.$$

Function to minimize

We aim to find the smallest date when an occurrence of a task can start. Thus, we must minimize the end date (which is equivalent to minimize begin date) of all the transitions:

$$\min \sum_{t \in T} \sum_{i \in \mathbb{N}} \alpha^i(t)$$

These constraints can be modeled as a dependency graph, where the nodes represent the executions of task occurrences and the arcs are the constraints between theses occurrences. The weight of an arc between two task occurrences is equal to the execution time of the tail occurrence.

In Figure 2 is illustrated the dependency graph of application from Figure 1. We remind that the repetition vector for (t_1, t_2) is equal to $\gamma = [4, 7]$. In one execution cycle task t_1 occurred 12 times ($n_\tau(t_1).\gamma_{t_1}$) and task t_2 14 times ($n_\tau(t_2).\gamma_{t_2}$). The size of the buffer is equal to 6. Black arcs correspond to consistency constraints (time and production) and red arcs correspond to capacity constraints. Thanks to this graph, we are able to compute efficiently the self timed execution for the application.

IV. Computation of the throughput for a given storage distribution

Different methods have been proposed for computing the throughput, either exact or approximate. The self timed execution provides the maximal throughput. The throughput of a task corresponds to the number of occurrences per unit time. It is denoted by $Th(t)$. We can define the throughput of the CSDF $\mathcal{G} = (T, A)$ as $Th(G) = \frac{Th(t)}{n_\tau(t)\gamma_t}$ for an arbitrary $t \in T$ [7].

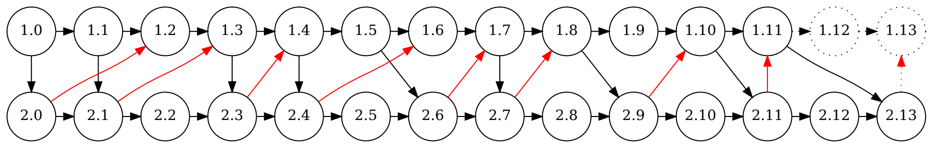


Figure 2. Dependency graph for one execution cycle of the application illustrated in Figure 1.

The self timed execution has 2 sequential phases: a transient phase composed of a finite sequence of occurrences and a periodic phase. The transient phase is bounded while the periodic phase repeats indefinitely. [12] describes an exact method for finding the value of the maximal throughput. This method extracts the periodic phase and computes the throughput for this phase. As the system is supposed to be executed indefinitely, the throughput of the system tends to be equal to the throughput of the periodic phase. Although this method finds the maximal throughput, it is efficient only for simple CSDFs with a moderate size for the cycle execution. The main difficulty is the extraction of the periodic phase which is exponential. Thus, for complex CSDFs the time required for the computation of throughput and the number of states to store is prohibitively large. This is the main motivation to develop a method that can give a good lower bound to the maximum throughput, with a lower computing time and smaller storage needs.

Hereafter we introduce a method that computes a lower bound to the throughput. Because the research of the periodic phase can take a huge amount of time and/or storage memory, it is more efficient to compute an approximate value for the throughput. Our idea is based on an experimental observation that even if a limited number of states is examined it is possible to find a sufficiently close bound to the exact value. The purpose of our method is to generate a sequence of a schedule using the self timed execution that we will repeat indefinitely to get a complete schedule. We compute the throughput for this sequence. We call it the root sequence of our schedule. After the computation of the self timed execution for a significant number of occurrences, it is possible to consider a number of execution cycles as the root sequence. For the global schedule, the exact copy of the root sequence may, for example, begin at the end of previous occurrence (i.e. the last occurrence of the previous root sequence).

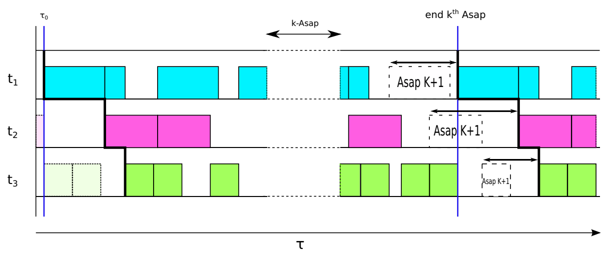


Figure 3. Representation of the scheduling for a 3 tasks application with a root sequence starting at the end of previous root sequence.

Figure 3 gives an example for the generation of such a root sequence. After the computation of the self timed execution for n execution cycles, let us define τ_0 as the beginning of the $(n+1)^{th}$ execution cycle, i.e. the start time of the first occurrence of task t_1. The first n cycles are ignored because these states are more likely to be in the transient phase. Indeed, especially the beginning transient phase induced underestimation of the lower bound. Let us compute k execution cycles from τ_0 and define these k cycles as the root sequence. We can define the beginning of the next sequence, i.e. the begin date of the first occurrence of t_1 in our example, at the end time of the last occurrence of k^{th} cycle from τ_0, i.e. at the end of execution of the last occurrence of t_3. The delay between the self timed execution (in dotted lines) and the periodic execution is also shown in the Figure 3.

For finding the application throughput we need to compute the inverse of the duration between the beginning of the two sequences multiplied by the number of execution cycles k.

Another solution, which provides a better bound, is to define the beginning of the following occurrence as soon as it is possible to get the periodicity. The shape induced by the start time of first occurrences of tasks at τ_0 must be repeated in the following execution cycle. With this method, at least one task starts at the same time as the self timed execution of the following cycle.

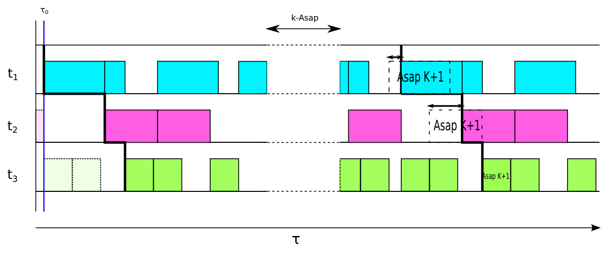


Figure 4. Representation of the scheduling of the same 3 tasks application as in Figure 3 with an improved periodic phase.

Figure 4 shows a scheduling with the same root sequence as in Figure 3, with the same self timed execution. After k cycles of execution from τ_0, the new phase begins as soon as it is possible to get the same shape as the one from τ_0. Tasks t_1 and t_2 are delayed from the $(k+1)^{th}$ ASAP. The task t_3 occurred at the same time as the ASAP time. The throughput is computed like in the previous method.

It is easy to prove that the obtained throughput is a lower bound to the maximal throughput. Indeed, the self timed execution gives the maximum possible value for the throughput, so in the best case the proposed approximate method will give this value as throughput.

If the computation of the throughput uses an approximate method without extracting the periodic phase, the number of states to store will be limited. The number of execution cycles to store for the computation of the self timed execution is determined by

$$\max_{l\in A}\left(\left\lceil \frac{q_0(l)}{\gamma_{p(l)}\sum_{j=0}^{n_\tau(p(l))-1} qp_l(j)} \right\rceil, \left\lceil \frac{d_l}{\gamma_{p(l)}\sum_{j=0}^{n_\tau(p(l))-1} qp_l(j)} \right\rceil\right).$$

This expression represents the number of cycles required for expressing all production consistency constraints and all capacity constraints. The start times for the first execution cycle of the root sequence are required for the evaluation of its duration, i.e. $|T|$ start times. And lastly, during the computation of the self timed execution it is only required to store the start dates for the current states, i.e. only for the $|T|$ current states.

The obtained value of the throughput converges in a non-monotonic way to the value of the maximal throughput in function of the number of the execution cycles. The throughput depends on various parameters : number of states for cycles of execution, storage distribution and preload values. The choice of the number of cycles to schedule an approximate throughput depends on these parameters.

Let us introduce a trivial application allowing us to understand the advantage of our approximate method. This application is composed of one transmitter T connected to two receivers: R_1 and R_2. Transmitter and receiver tasks are both SDF and they produce (or consume) one data at each firing. The repetition vector associated to the SDFG is $\gamma_{TR} = \left[\begin{array}{ccc} 1 & 1 & 1 \end{array}\right]^T$. The two communication links are defined by l_0 and l_1.

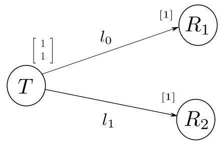


Figure 5. Simple application with 1 transmitter and 2 receivers.

Let us define the size of buffers $d_{l_0} = 3$ and $d_{l_1} = 3$. These buffers are empty at initial state. The execution time for the transmitter T is set to 10 ms. The execution time of R_1 is set to 99 ms and the execution time of R_2 is set to 100 ms. The execution pattern search algorithm needs to proceed 191 to execution cycles before finding the periodic phase. Even if it is a simple application, the number of execution cycles to proceed is high because the execution times of the receivers are similar. At each end of an execution cycle, the end of R_2 is shifted with 1ms from the end of R_1. The periodic phase is reached when T can't fire anymore because the buffer of l_1 is full. The more the size of the buffer of $l_1$is is large, higher is the number of execution cycle to get. In this example, each time the size buffer of l_1 is increased by 1, 100 more execution cycles are needed to find the periodic phase. The lower the gap between execution times of R_1 and R_2 is, the higher the number of execution cycles needed to find the periodic phase will be.

It is obvious that the throughput evaluation of this application is easy to solve because of its specifications. But such a scheme can be nested several times in a greater application with non-trivial repetition vector. The cost of finding the periodic phase can increase with the complexity of the application. In the case of buffer sizes evaluation, the throughput evaluations must be repeated a number of times growing exponentially with the number of buffers. The number of execution cycles in the periodic phase can also be very high. Thus it is necessary to have an approximate method.

V. Experimental results

Buffer sizing under throughput constraints is used in the context of many-core embedded architectures. The method we propose is used in an end-to-end compilation chain for the ΣC dataflow programming language [8][1]. The method we developed for throughput evaluation uses both the exact and the approximate computation of the throughput presented in Section 4. The exact method is used during the scheduling of states. When a periodic pattern is found, the exact throughput is returned. Otherwise, if a sufficient number of states have been scheduled, we define a root schedule and we compute a lower bound for the throughput. The method used for exploring the storage distributions space is the same as the one presented in [12]. For each storage distribution explored, an evaluation for throughput is realized. The applications used for the tests have been implemented in ΣC language. For all these applications we consider the shared data are homogeneous or have been homogenized previously in the compilation chain.

First, we tested some applications presented in different papers. The MP3 playback is a simple benchmark used for evaluating computation of minimal buffer sizes under throughput constraints methods[13][2]. This application is a sequence of 4 tasks, but even if there are only 4 tasks, the number of occurrences in one execution cycle is 10791. For this example the exact method provides results from less than 4 execution cycles for every storage space. By comparison with the exact method, our approximate method gives a similar throughput in the same number of execution cycles.

[2] presents a version of an improved MP3 playback with two independent input streams. When testing this application we obtain some interesting results: for certain distribution storages, the exact method is able to find a periodic pattern only after proceeding an important number of scheduling.

[10] provides a CSDF of a channel equalizer application. This application is used to compensate the distortion of a FM signal. Here dependency arcs are added in order to model the accesses on the memory shared of the tasks on the same processors. The methods proposed in [2] and [13] do not manage to provide the throughput of this application. [3] gives an underestimate value of the maximal throughput for the optimal distribution storage. Our approximate method gives

the optimal results of maximal throughput evaluation for this distribution storage.

We tested these applications, without searching the periodic pattern for various applications. Our aim is to test the accuracy of our approximate method. In all applications tested, the approximate method gives a satisfying result within the 5 first execution cycles. It provides almost the same throughput (differences are insignificant). For the example of the MP3 application with 2 inputs, the exact method finds the throughput for one distribution after scheduling 370 execution cycles. As each cycle is composed of 10998 occurrences, the number of states to explore is equal to 4157244. This number is huge for an application with only 6 tasks. The approximate method can provide the same throughput value in only 4 execution cycles.

We have also tested our method on a motion detection application which performs the tracking of a target in a sequence of video frames. This application is interesting for illustrating the advantage of the approximate method for throughput estimation. In this application, two successive video frames are analyzed and the movement of targets between these frames is outputted. The method is based on the Block Matching Algorithm. This algorithm aims to discover the temporal redundancy between two successive images and find matching blocks in these frames. In this method we compare the absolute difference of measure between a pixel and the corresponding pixel in previous frames. Each frame is divided in horizontals strips. The standard deviation is computed for all the macro-blocks of size 8x8 of a strip and the minimum is selected. The minimum standard deviation of all the strips is selected as a threshold (or noise level). We apply a filter to generate a binary image which reveals pixels for which the deviation of the absolute difference is over the threshold. Then, all connected components are computed for each strip there is a difference in pixels. The application ends with merging all the connected components which are overlapping on different strips. The output image shows the moving targets with bounding boxes.

Figure 6 [11] is an overview of the ΣC dataflow graph of the motion detection application. Two video frames readers (*io*), for current and previous frames, are in input. 8 different strips of both images are distributed via split tasks (*s*) to the Δ agents. The Δ agents process the difference between the 2 frames. The outputs of Δ agents are duplicated (empty vertexes are the duplicate tasks). The first output of the duplicate task sends data to a σ agent which processes the variance of the absolute difference. All the σ agents merge to the agent m which select the minimal variance. The square root of this variance defines the threshold for the construction of the binary image. The threshold agent (*t*) compares the deviation of the result of Δ agents and the value of the threshold. Agent t builds a binary image by strip which is used by agent c for the extraction of connected components. The second agent *m* makes the fusion between bounding boxes of each strip. The video frame writer(*io*) gives the output image identifying the moving targets.

Each agent in Figure 6 is composed of multiple tasks. The application has 67 tasks and 95 links. Each execution cycle contains 95 occurrences. The application can be considered as a relatively simple application. The simulation, using ISS (Instruction Set Simulator), gives the number of processor cycles for each agent execution. The execution time is deduced from the mean of these executions. This application is mapped on the architecture provided by Kalray and presented in [6]. The frequency of the chip is 400MHz. The duplicate and the split tasks have no execution times because they are system tasks which are compiled. Theoretically, they do not exist and are not mapped to the chip. The test have been made with execution times for these agents equal to 0, 1 and 500 processor cycles. The results obtained have been almost the sames for each one of the three configurations.

	number of execution cycles	pattern found	throughput	time elapsed
MD	100000	no	24.47Hz	>30 min
MD	4	-	24.47Hz	1ms
MD revised	20000	no	24.53Hz	>5min
MD revised	4	-	24.52Hz	1ms

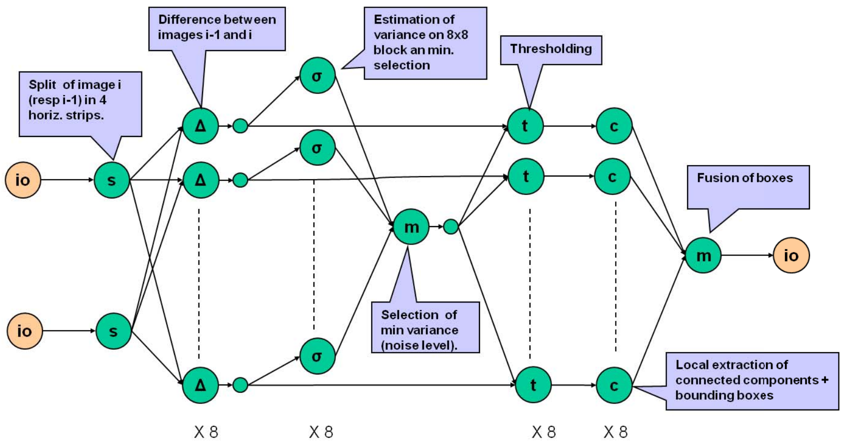
Table I
RESULTS FOR MOTION DETECTOR APPLICATION.

First, we used the mean execution time for all single tasks of the motion detection application. The execution times for tasks of the same types are different but close in value. With these execution times, the exact method does not manage to find the periodic pattern after the computation of more than 100000 execution cycles for all the distribution storages the target throughput is satisfied. The observed behavior is similar to the application presented in Figure 5. The end of the transient phase is not reached after a large number of execution cycles. Our approximate method provides a satisfying throughput, for all explored distributions, after only 4 execution cycles.

We performed the test for the motion detection application with the same execution times for all tasks of the same type (MD revised). We affected the mean times to all tasks of the same type. Even in this case there are distributions for which we do not manage to find the periodic pattern after 20000 execution cycles.

Table I shows the results obtained for the motion detection application. We compare the difference between the approach using exact method with approximate method after the processing of a certain number of execution cycles and the approximate method limited to 4 cycles. We can see that the results obtained with the approximate method in 4 cycles are similar whereas the time needed for computing buffer sizes is in the order of the millisecond.

The approximate method seems to be valid with such results. Nevertheless, there must be applications where the exact method is more efficient and gives a better accuracy for a small number of execution cycles: applications for which the approximate method converges in a non-monotonic way to the exact value of maximal throughput. However, we can say the approximate method we propose is an accurate complement to the exact method. Indeed, the more the periodic pattern of execution is searched, the more the root sequence is large and



Difference between images i-1 and i
Estimation of variance on 8x8 block an min. selection
Thresholding
Split of image i (resp i-1) in 4 horiz. strips.
Fusion of boxes
Selection of min variance (noise level).
Local extraction of connected components + bounding boxes
io
s
Δ
σ
m
t
c
X 8

Figure 6. Dataflow scheme for motion detection application.

the approximate throughput will be close to the exact value.

VI. Conclusion

In this paper we presented a new method for the computation of throughput in order to evaluate the buffer size. Existing methods have several disadvantages. Exact methods can be efficient for simple cases, but the throughput evaluation and the storage space to explore have an exponential complexity. Existing approximate methods are efficient but the underestimation of throughput and/or buffer sizes are important for many applications. Moreover these methods can't provide results for a certain type of applications.

Our method combines both exact and approximate methods for the evaluation of the throughput. This technique avoids a too long exploration of the state space in order to find the periodic phase and minimize the underestimation of the throughput.

The next step is to improve the algorithm of the research in the distribution storage space. Indeed, even with the approximation presented in [12] the number of storage space to explore remains exponential. We want to develop an accurate method which is polynomial with the number of buffers to dimension in the application.

Another problem to investigate is to introduce uncertainty in the execution times of the tasks. Indeed, in function of the input, the time needed for the execution of a task can vary significantly. We may be able to take into account these variations in the throughput evaluation. The main issue is that taking into account the uncertainty will complexify the problem of the throughput evaluation and we will have to perform some approximations. We should see if the results of this research path are interesting compared to those given by a worst case execution time approach.

References

[1] Pascal Aubry, Pierre-Edouard Beaucamps, Frédéric Blanc, Bruno Bodin, Sergiu Carpov, Loïc Cudennec, Vincent David, Philippe Dore, Paul Dubrulle, Benoît Dupont de Dinechin, et al. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. *Procedia Computer Science*, 18:1624–1633, 2013.

[2] M. Benazouz, O. Marchetti, A. Munier-Kordon, and T. Michel. A new method for minimizing buffer sizes for cyclo-static dataflow graphs. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, pages 11–20. IEEE, 2010.

[3] M. Benazouz, A. Munier-Kordon, et al. Cyclo-static dataflow phases scheduling optimization for the throughput constrained buffer sizes minimization problem. 2011.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258 vol.5. IEEE, May 1995.

[5] Ali Dasdan, Sandy S. Irani, and Rajesh K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 37–42, New York, NY, USA, 1999. ACM Press.

[6] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the kalray MPPA®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[7] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij. Throughput analysis of synchronous data flow graphs. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 25–36. IEEE, 2006.

[8] Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. ΣC: A Programming Model and Language for Embedded Manycores. In *ICA3PP (1)*, pages 385–394, 2011.

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

[10] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen. Evaluation of the throughput computed with a dataflow model-a case study. *Eindhoven University of Technology, Department of Electrical Engineering, Electronic Systems. ISSN*, pages 1574–9517, 2007.

[11] Renaud Sirdey. *Contributions à l'optimisation combinatoire pour l'embarqué: des autocommutateurs cellulaires aux microprocesseurs massivement parallèles*. Habilitation, Université de Technologie de Compiègne, 2011.

[12] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *Computers, IEEE Transactions on*, 57(10):1331–1345, 2008.

[13] Maarten H. Wiggers, Marco J. G. Bekooij, and Gerard J. M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 658–663, New York, NY, USA, 2007. ACM.