Buffer Dimensioning for Throughput Improvement of Dynamic Dataflow Signal Processing Applications on Multi-Core Platforms

Małgorzata Michalska, Endri Bezati, Simone Casale-Brunet, Marco Mattavelli
EPFL SCI-STI-MM
École Polytechnique Fédérale de Lausanne, Switzerland

***Abstract*—Executing a dataflow program on a parallel platform requires assigning to each buffer a given size so that correct program executions take place without introducing any deadlock. Furthermore, in the case of dynamic dataflow programs, specific buffer size assignments lead to significant differences in the throughput, hence a more appropriate optimization problem is to specify the buffer sizes so that the throughput is maximized and the used resources are minimized. This paper introduces a new heuristic methodology for the buffer dimensioning of dynamic dataflow programs, which is considered as a stage of a more general design space exploration process.**

I. Introduction

Last generation of massive parallel many/multi-core processing platforms have brought a renewed interest in dataflow programming approaches attempting to find more natural methodologies to efficiently exploit the available parallelism. The evolution of processing platforms towards concurrent systems composed of homogeneous or heterogeneous arrays of processors providing massive parallelism has been essentially triggered by the limitations of the switching frequency and the power dissipation of deep submicron CMOS technology. In the meantime, the common practice of software development is still relying on sequential approaches and ad-hoc transformations in concurrent non-portable SW versions. In principle, a dataflow program is defined as a directed graph in which each node represents a computational kernel, called *actor*, and each edge a first-in first-out *(FIFO)* lossless interconnection channel, called and often implemented by a memory *buffer*. The processing part of the actors is encapsulated in the atomic executions (*firings*) called *actions*. The communication between actors is permitted only by the exchange of atomic data packets, called *tokens*, by means of interconnection channels implemented by buffers with, in the abstract description of the program, infinite size. Figure 1 illustrates the structure of a simple dataflow program.

A dataflow program can be seen as an high-level description or specification of a processing algorithm which abstracts from aspects of the actual execution on a processing platform. The program is the starting point of the stages of a design flow that generates specific hardware and/or software implementations by removing the abstractions and adding design settings according to specific constraints of the platform and optimization objectives of the design. Hence, a precious feature of a dataflow program is essentially the portability of the program providing the abstract implementation of an application and the portability of implementation features such as the parallelism as explicitly exposed in the abstract dataflow program itself [1], [2]. Such properties are very interesting, but require dealing with several challenges in the design flow process of generating efficient implementations for the new many/multi-core processing platforms. The effectiveness of an implementation depends on several settings determined at the different design flow stages, among which the commonly outlined in literature are: (a) the partitioning of the actors to the available processing units (binding, mapping in the space domain), (b) the sequencing of actor executions for each processing unit (scheduling or mapping in the temporal domain) [3]. These design settings might need to comply with several constraints and/or optimization design objectives such as data throughput, energy consumption, memory utilization, latency etc.

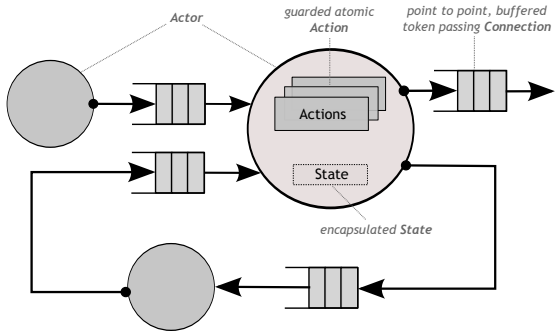


Fig. 1: Illustration of an example of a simple dataflow program.

According to the specific Model of Computation *MoC* used to express the dataflow program, the sizes of the interconnecting channels (buffers) constituting the network might be considered unbounded [4] and this is the case for all dynamic MoC. However, when a program is executed on a given platform, each buffer needs to be specified with a finite size. A necessary constraint at the base of this process is to specify the buffer sizes so that the program can correctly execute without any deadlocks. A possible design optimization objective can also be to minimize the amount of memory used by all buffers in order to meet specific memory constraints of the

ISBN 978-0-9928626-7-1 © EURASIP 2017

platform (i.e., embedded-memory limitations of FPGAs). In the case of dynamic dataflow programs, data dependencies determining action executions and the characteristics of the traffic flowing across each buffer are important factors that determine the overall program data throughput and are subject to the constraints imposed by the finite buffer sizes. The joint optimization problem aiming at the minimization of the total buffer size and the maximization of program throughput constitutes an interesting design objective. The methodology described in this paper provides a new heuristic solution to this problem. The optimization stage can be considered a part of a more general design space exploration and optimization approach capable of dealing with the more general class of dataflow *MoC*, namely dynamic dataflow programs.

The methodology is based on building a model of the dataflow program execution by generating the graph representing the execution trace with the associated dependencies. The model built with a given set of statistically meaningful input stimuli and then completed by the selected design settings, corresponding to a point in the multidimensional design space, provides accurate and rich measures about the performance of the corresponding implementations. The exploration of the multidimensional design space in terms of partitioning, scheduling and buffer size settings proceeds according to provided metrics. It differs from other approaches that disregard the exploration of the buffer size settings and limit it to finding only a deadlock-free configuration (e.g. [5]) or explore the buffer size configurations disregarding the other dimensions in the space (e.g [6]). In this context, to the best of our knowledge, this is the first work that considers the buffer dimensioning problem with regards to throughput improvement and targets dynamic applications, without limiting the approach to a subset of static dataflow programs.

The paper is structured as follows: Section II formulates the design space exploration problem and describes the underlying methodology related to the construction of an execution trace. Then, Section III describes two variants of a throughput constrained buffer minimization heuristic, which is the core contribution of the paper. Experimental results are reported and discussed in Section IV. Finally, conclusions and directions of future works are summarized in Section V.

II. Problem formulation and modeling

The design space exploration problem considered in this work consists of searching for three setting configurations. Each firing j (corresponding to an execution of an action) must be assigned to one of m parallel machines. Each firing j has an associated processing time (referred to as weight) p_j and a group (or actor) g_j. All firings belonging to the same group (actor) must be assigned to the same machine. If two firings belong to different groups, a communication time $w_{jj'}$ occurs between them. Since it is assumed that only one firing can be executed on one machine at a time, for each machine ρ a scheduling policy must be specified to decide the execution order. The scheduling must respect the precedence orders, including the communication time, if relevant. Each interconnecting buffer b must be assigned with a finite size B_b. These sizes are taken into account during the scheduling and a firing j can be chosen for execution ($j_s = j$) only if the sum of tokens currently present in its outgoing buffer(s) (q_b) and the tokens produced by the firing j does not exceed the specified buffer size. The objective of specifying these configuration settings is to maximize the throughput, or, in other words, minimize the termination time of the very last firing. Hence, the objective function assigned to this optimization problem can be defined as $min(T_{end}(j_{last}))$, where the decision variables involve the partitioning of every firing $K_p(j)$, the scheduling inside each machine $K_s = \{j, j', \ldots\}$ and the buffer size specification for each buffer $K_b(b) = B_b$. The following constraints are defined:

- **Group**: $g_j = g_{j'} \Rightarrow K_p(j) = K_p(j')$
- **Precedence**: $j \prec j' \Rightarrow T_{start}(j') \geq T_{end}(j)$
- **Buffer capacity**: $j_s = j \Rightarrow \Sigma_{q_b} + tokens(j) \leq B_b$

The optimization problem to be solved is multidimensional, furthermore, the configurations are closely dependent. For instance, the buffer dimensioning influences the establishment of an execution order by the buffer capacity constraint. Nevertheless, defining each of the configuration, even if disregarding the others, can be demonstrated to be a NP-complete problem. Finding good-quality solutions requires providing a model of a dynamic execution describing the firings and the relationships between then at an appropriate level of details and capable of taking into account the entire dynamic behavior of a program. Such a representation can be built by generating a directed acyclic graph G, called *Execution Trace Graph* (*ETG*), where each node, called *firing*, represents a single action execution and each directed edge, called *dependency*, represents an execution constraint between two firings. The processing times p_j's and communication times $w_{jj'}$'s are the weights assigned to the nodes and arcs, respectively. Such a representation has been already successfully used in order to solve a set of optimization problems related to the design space exploration of dynamic dataflow implementations [5], [7].

III. Throughput constrained buffer minimization

The starting point of the methodology is a deadlock-free buffer size configuration. It constitutes a border between the set of feasible and unfeasible design points. This configuration, considered as close-to-minimal, is evaluated on the basis of *ETG*, as described in [8]. Starting from this configuration, in each iteration the size of only one buffer is increased. The throughput constrained buffer minimization approach can be implemented in two different variants, both are based on the analysis of the critical path (cp) of the design, which is defined as the longest path between the start of the program to its termination. Hence, it contains the firings contributing to the longest serial part of a program execution. If a firing in the cp requires writing to an output buffer, this buffer is considered to be *critical*. If an execution of any firing is delayed because of the insufficient space in the buffer, a *blocking slot* occurs. If such a slot occurs for a firing in the cp it directly adds on to the length of the cp and hence, affects the total execution

ISBN 978-0-9928626-7-1 © EURASIP 2017

time (overall data throughput). The algorithm relies on the following properties:

- b_{cp}: identification of a critical buffer;
- tk_b: measure of the number of tokens blocked in one blocking slot;
- $time_b$: measure of the duration of one blocking slot;
- bi_{cp}: determination of the number of blocking slots along the critical path occurring throughout the execution for a buffer;
- bi: measure of the overall number of blocking slots occurring throughout the execution for a buffer;

These properties are tracked and extracted by means of performance estimation using the software tool described in [9]. The same tool is also used as an evaluation of each configuration.

A. Notion of partitioning and scheduling

As stated earlier, this work considers the problem of determining the buffer dimensions in the context of multidimensional design space exploration, hence with regards to the partitioning and scheduling configurations. Since this approach differs from the discussed state-of-the-art approaches which target only fully parallel executions, an example is this Section illustrates the difference of the approach. In case of a parallel execution which disregards the partitioning and scheduling configurations, increasing the size of any buffer always leads to an improvement of the performance (if a relevant blocking slot is removed) or the execution time remains unchanged (if an increase is not sufficient to remove a relevant blocking slot). The case of an execution, where a given subset of actors is executed sequentially within one processing unit, that is, with given partitioning and scheduling configurations, it is also affected by the presence of the scheduler. Depending on the scheduling policy within each processing unit, actors can be chosen for execution in a different order and the buffer capacity constraint affects the feasibility of different schedules. Hence, it is possible that increasing the size of a buffer might lead to a *decrease* of performance, since the admissible order of execution inside a given processing unit differs and may result less favorable. Such situation takes place quite often, since an increase of a given buffer affects the scheduling eligibility of *all* firings requiring writing to such buffer. In fact, only a fraction of them might be critical and executing a non-critical firing instead of a critical one might lead to the, mentioned earlier, drop of performance.

So as to illustrate this problem, Figure 2 presents a simple network consisting of few actors assigned to two partitions. The scheduling policy assumes that an actor is executed as many times as possible and in *Actor Q* the *action q1* has a priority over *action q2*. Two scenarios are considered: (1) all buffers have an equal size of 1, (2) *buffer b1* has the size of 2, the others of 1. Figure 3 presents the Gantt charts obtained in both cases. It can be noticed that although in the second scenario the buffer size configuration is larger, the execution time has been extended by 2 units. At this stage it must be emphasised that the likelihood of this behavior of a network remains fully dependent on the scheduling policy and its sensitivity to the buffer sizes. For this reason, the moves (i.e., increases of the buffer sizes) cannot be performed *blindly* and after each iteration it is mandatory to evaluate a move and revert it if a performance decrease has occurred. Furthermore, in each iteration instead of picking up one buffer, a *ranking* of buffers must be created and in case of a necessity to revert a move, the next buffer from the ranking is considered for an increase.

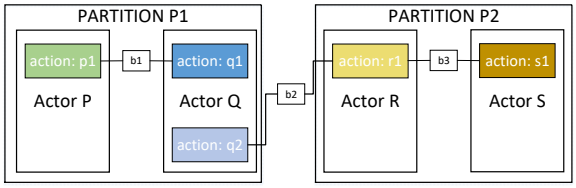


Fig. 2: Simple network with the assigned partitioning, scheduling and buffer dimensioning configuration.

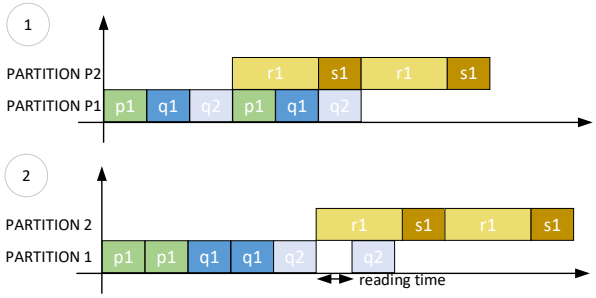


Fig. 3: Gantt charts for the execution of network from Figure 2 for the two buffer size configurations.

B. Heaviest blocking ranking

The first ranking, presented in Algorithm 1, looks for the *heaviest* blocking slot along the critical path. In this context a *heaviness* of a blocking slot is measured by a multiplication of the number of tokens blocked (tk_b) and the time they remained blocked ($time_b$). For each buffer in the critical path (b_{cp}) a maximal *heaviness* throughout the execution is recorded and among different critical buffers the one with the largest corresponding *heaviness* is chosen. This ranking intends to remove the most impacting sources of delay in the execution. Having to revert a move implies considering the next buffer from the map. Symbols p, s, b refer to partitioning, scheduling and buffer dimension configurations, respectively.

C. Criticality ratio ranking

The second ranking, presented in Algorithm 2, calculates the *ratio* between the critical blocking slots of a buffer (bi_{cp}) and all blocking slots of this buffer (bi) throughout the execution. Buffers with the highest ratio bi_{cp}/b_i are first considered for an increase. This ranking intends to minimize the unnecessary increases for the firings which are not in the critical path. It must be emphasized that for the case of both rankings, the cp analysis has to be performed in every iteration, since changing even one buffer size in the network can modify the execution order and, consequently, the location of the cp.

ISBN 978-0-9928626-7-1 © EURASIP 2017

Data: IN: $conf(p,s)$, ETG; OUT: $conf(b)$
$conf(b)$ = minConf(ETG);
while *iteration* $<$ *max* **do**
  map$\{b_{cp}, max(tk_b * time_b)\}$ = cp($conf(p,s,b)$);
  foreach b_{cp} **do**
    $b* = max($map$\{b_{cp}, max(tk_b * time_b)\}$;
    $conf(b*) = conf(b*) * 2$;
    if $time^* < time$ **then**
      break;
    end
    else
      $conf(b*) = conf(b*)/2$;
    end
  end
end

Algorithm 1: Heaviest blocking ranking.

Data: IN: $conf(p,s)$, ETG; OUT: $conf(b)$
$conf(b)$ = minConf(ETG);
while *iteration* $<$ *max* **do**
  map$\{b_{cp}, \langle bi_{cp}, bi\rangle\}$ = cp($conf(p,s,b)$);
  foreach b_{cp} **do**
    $b* = max(bi_{cp}/b_i)$;
    $conf(b*) = conf(b*) * 2$;
    if $time^* < time$ **then**
      break;
    end
    else
      $conf(b*) = conf(b*)/2$;
    end
  end
end

Algorithm 2: Criticality ratio ranking.

IV. Experimental results

This Section reports the experimental results performed for the two variants of the throughput constrained buffer minimization heuristic. Two different video applications written in CAL Actor Language [10] have been tested: MPEG4-SP decoder design [11] (consisting of 34 actors, 80 buffers) and High Efficient Video Coding (HEVC) decoder [12] (consisting of 22 actors, 219 buffers). The sequences used in the experiments are a 30-frame QCIF (176x144 8-bit pixels) *Foreman* bit-stream for MPEG and a 10-frame HD (1280x720 8-bit pixels) *BQ Terrace* bit-stream for HEVC. Basing on the performed experiments it has been observed that good quality configurations found for these sequences remain of good quality also for other sequences. Different partitioning configurations (as a part of the multidimensional exploration process) have been established manually so that good values of throughput are achieved for a given number of processing units (2, 3 and 4 for MPEG4-SP and 7 for HEVC). The target platforms used for the experiments included Intel i7-3770 CPU with 4 cores and Intel i7-5960X CPU with 8 cores.

In order to illustrate the importance of dimensioning the buffers for obtaining the highest throughput (close-to optimal), the first set of results (Tables I and II) reports the throughput obtained for both design in two cases: for the close-to-minimal buffer size configuration (used as a starting point for the heuristic procedure) and the *extreme* configuration, where to each buffer a size equal to $2^{18} = 262144$ is assigned, disregarding its real traffic. The corresponding total buffer sizes (expressed in tokens) are indicated in the headers of the Tables. It can be observed that the differences are remarkable for both designs and for different partitioning configurations. Hence, it shows the effectiveness of searching a joint optimization of throughput and memory objective functions.

Proc.	14k	21m
4	1820	3138
3	1410	2240
2	1156	1613
1	744	1005

TABLE I: MPEG4-SP: performance differences [FPS]

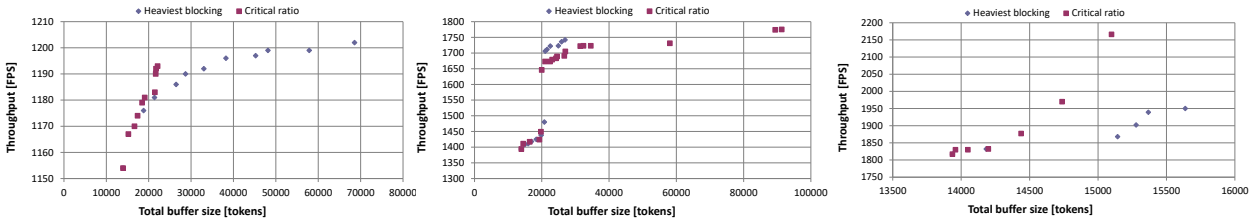
Proc.	93k	57m
7	53	127
6	54	130
5	57	123
4	52	121
3	51	103
2	42	79
1	41	53

TABLE II: HEVC: performance differences [FPS]

Figures 4a-5 present the buffer size configurations generated in different iterations of the throughput constrained buffer minimization heuristic in its two variants. Both variants have been executed with the same upper bound on the number of iterations. The charts summarize the throughput with regards to the total buffer size of each configuration. For the case of MPEG4-SP, 4 processing units and HEVC, 7 processing units it can be concluded that the *criticality ratio* ranking provides better results, because within the same number of iterations it leads to higher throughput values and smaller total buffer sizes. For the other two analyzed cases, it is not possible to make the same observation, because one variant goes more towards higher throughput values, whereas the other one towards smaller total buffer sizes. It can be also observed that the partitioning configuration remains dominant over buffer size configuration, but for each partitioning configuration buffer dimensioning improves the solution in the range of 3-22%.

Another part of the experiments consist of using the throughput constrained buffer minimization heuristic (*heaviest blocking* variant in this case) to find a good quality buffer size configuration using the full parallel execution, hence, exploring the design space in a simpler way only with regards to the buffer size as already attempted in the reported related works. It has been checked if the solutions obtained in this setting can be exploited also when the other two configurations are included. As shown in Figure 6, the throughput of the solutions resulting from a single dimension exploration (parallel-based analysis) are by far providing worse results than the results obtained by the solutions resulting from a multidimensional exploration (partitioned-based analysis). They do not form a monotonic curve and they rapidly converge to a local optimum. The same Figure reports also the entire curve of solutions, that is, when the throughput constrained buffer minimization is

ISBN 978-0-9928626-7-1 © EURASIP 2017



(a) 2 processing units. (b) 3 processing units. (c) 4 processing units.

Fig. 4: MPEG4-SP decoder

executed without any upper bound on the number of iterations and eventually converges to an approximation of an infinite buffer size. The approximation of an infinite buffer size is obtained by continuously doubling the size of all buffers in the network till no more performance improvement is achieved. It can be clearly observed that the solutions obtained with the throughput constrained buffer minimization yield much higher throughput values for much smaller overall buffer sizes. The reference point in the Figure is the initial deadlock-free configuration.

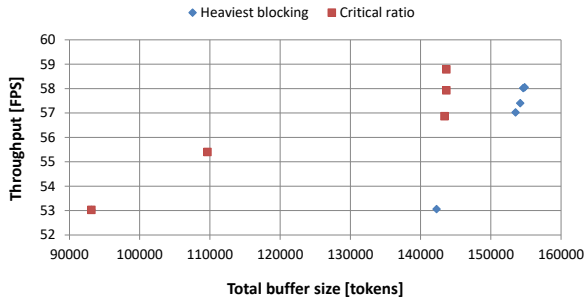


Fig. 5: HEVC decoder: 7 processing units.

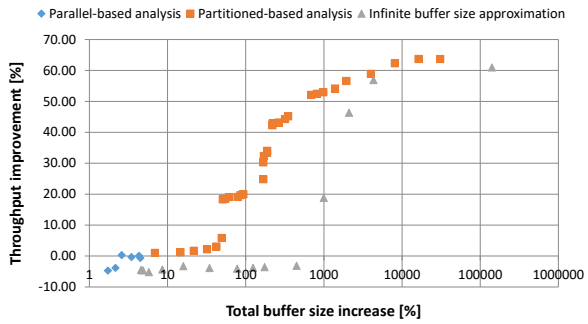


Fig. 6: Throughput constrained buffer minimization vs infinite buffer size approximation and parallel-based analysis: MPEG4-SP decoder, 3 processing units, heaviest blocking ranking.

V. Conclusion

The paper describes two variants of a heuristic methodology for dimensioning buffer sizes based on a multidimensional design space exploration process of dynamic dataflow implementations. The objective of the algorithm is to determine solutions that jointly maximize the data throughput for a minimal memory resources usage. The approach employing two variants has been validated by different dynamic dataflow designs characterized by high levels of complexity and executed on a real processing platform, in which the number of processing units is significantly smaller than the number of actors in the network. The obtained high quality results show the effectiveness of the throughput constrained buffer minimization algorithm based on an accurate model of the dataflow execution and on a multidimensional representation and exploration of the design space. Among possible improvements of the design space exploration methodology, an interesting option is to study the performance decreases occurring in correspondence of some buffer size increases to investigate if they can be avoided by modeling the worst-case execution scenario or predicting the actual choices of the scheduler according to the implemented policy.

References

[1] S. Bhattacharyya, P. Murthy, and E. Lee, *Software synthesis from dataflow graphs*. Springer Science & Business Media, 2012, vol. 360.

[2] E. Lee and T. Parks, "Dataflow process networks," in *Proceedings of the IEEE*, 1995, pp. 773–799.

[3] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," in *Seventh International Conference on Application of Concurrency to System Desig*. IEEE, 2007, pp. 29–40.

[4] G. Kahn, "The semantics of simple language for parallel programming," *IFIP Congress*, 1974.

[5] J. Castrillon, R. Leupers, and G. Ascheid, "MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs," *IEEE Transactions on Industrial Informatics*, pp. 527 – 545, 2013.

[6] S. Casale-Brunet, M. Mattavelli, and J. W. Janneck, "Buffer optimization based on critical path analysis of a dataflow program design," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 1384–1387.

[7] J. Janneck, I. Miller, and D. Parlour, "Profiling dataflow programs," in *Multimedia and Expo, 2008 IEEE International Conference on*, Jun. 2008, pp. 1065–1068.

[8] T. Parks, "Bounded scheduling of process networks." *PhD Thesis at University of California at Berkeley, USA*, 1995.

[9] M. Michalska, S. Casale-Brunet, E. Bezati, and M. Mattavelli, "High-precision performance estimation of dynamic dataflow programs," *10th International Symposium on Embedded Multicore/Many-core Systems on Chip*, 2016.

[10] J. Eker and J. W. Janneck, *CAL Language Report*. UC Berkeley: Tech. Memo UCB/ERL M03/48, 2003.

[11] E. Bezati, R. Thavot, G. Roquier, and M. Mattavelli, "High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms," *Journal of Real-Time Image Processing*, vol. 9, no. 1, pp. 251–262, 2014.

[12] document ITU-T Rec. H.265, I.-T. ISO/IEC 23008-2 (HEVC), and ISO/IEC, "High Efficiency Video Coding," 2013.

ISBN 978-0-9928626-7-1 © EURASIP 2017