



**HAL**  
open science

## Throughput optimization for pipeline workflow scheduling with setup times

Anne Benoit, Mathias Coqblin, Jean-Marc Nicod, Laurent Philippe, Veronika Rehn-Sonigo

► **To cite this version:**

Anne Benoit, Mathias Coqblin, Jean-Marc Nicod, Laurent Philippe, Veronika Rehn-Sonigo. Throughput optimization for pipeline workflow scheduling with setup times. [Research Report] RR-7886, 2012, pp.29. hal-00674057v1

**HAL Id: hal-00674057**

**<https://inria.hal.science/hal-00674057v1>**

Submitted on 24 Feb 2012 (v1), last revised 19 Jun 2012 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



# Throughput optimization for pipeline workflow scheduling with setup times

Anne Benoit, Mathias Coqblin, Jean-Marc Nicod, Laurent Philippe,  
Veronika Rehn-Sonigo

**RESEARCH  
REPORT**

**N° 7886**

February 2012

Project-Team ROMA





## Throughput optimization for pipeline workflow scheduling with setup times

Anne Benoit, Mathias Coqblin, Jean-Marc Nicod, Laurent Philippe, Veronika Rehn-Sonigo

Project-Team ROMA

Research Report n° 7886 — February 2012 — 29 pages

**Abstract:** In this paper, we tackle pipeline workflow applications that are executed on a distributed heterogeneous platform with setup times. In such applications, several computation stages are interconnected as a linear application graph. The considered stage/processor mapping strategy is based on interval mappings, where an interval of consecutive stages is performed by the same processor. Each stage holds a buffer of limited size where intermediate results are stored and a setup time occurs when passing from one stage to another. Typical examples for this kind of applications are streaming applications such as audio and video coding or decoding, image processing using co-processing devices as FPGA. In this paper, we focus on throughput optimization. This problem is known to be NP-hard as soon as heterogeneous processors are used, even without setup times and with homogeneous communication links. We provide an optimal algorithm for the inner-processor scheduling of stages on homogeneous platforms with identical buffer capacities. We also provide an interval mapping that, when running the former algorithm including all processors, maximizes the throughput of the application. Last, we deal with the problem of allocating the buffers for each stage in shared memory. We provide an optimal algorithm for buffer allocation in the case where the memory may be split evenly, and we propose heuristics to handle any remainder in the memory allocation.

**Key-words:** setup times; buffer; coarse-grain workflow application; throughput; complexity results.

**RESEARCH CENTRE  
GRENOBLE – RHÔNE-ALPES**

Inovallée  
655 avenue de l'Europe Montbonnot  
38334 Saint Ismier Cedex

## Optimisation du débit dans l'ordonnancement des flux en pipeline avec temps de reconfiguration

**Résumé :** Dans cet article, nous traitons les applications de type pipeline exécutées sur des plateformes distribuées hétérogènes avec des temps de reconfiguration. Dans ce type d'applications, les différentes tâches qui composent un même calcul sont interconnectées selon un graphe linéaire. La stratégie d'affectation des tâches sur les processeurs est basée sur le principe d'une allocation par intervalles, dans laquelle un intervalle de tâches consécutives est affecté au même processeur. À chaque tâche est assigné un buffer de taille limitée, dans lequel les résultats intermédiaires sont stockés. Un temps de reconfiguration est nécessaire à chaque processeur pour passer de l'exécution d'une tâche à une autre. Un exemple d'applications regroupe les applications gérant des flux de données (*streaming applications*), comme l'encodage/décodage audio et vidéo, ou le traitement d'un flux d'images à l'aide de coprocesseurs tels que le FPGA. Dans cet article, nous cherchons à optimiser le débit de telles applications. Le problème est NP-complet dès lors que les processeurs sont hétérogènes, même avec des communications homogènes, sans temps de reconfiguration. Nous proposons un algorithme optimal pour l'ordonnancement des tâches au sein d'un même processeur disposant de buffers de tailles identiques. Nous proposons également une allocation par intervalles qui permet d'obtenir un débit maximum en exécutant notre algorithme d'ordonnancement d'une plateforme de processeurs homogènes. Enfin, nous nous intéressons au problème de l'attribution des tailles des buffers des tâches allouées à un processeur en mémoire partagée. Nous proposons un algorithme optimal pour l'allocation des buffers dans le cas où la mémoire est découparable en parts égales, et nous proposons des heuristiques pour exploiter au mieux l'éventuel résidu de mémoire après allocation.

**Mots-clés :** temps de reconfiguration; buffer; flux de travaux; débit; résultats de complexité.

## 1 Introduction

In this paper, we consider pipeline workflow applications mapped on a heterogeneous distributed platform such as a grid. This kind of applications is used to process large data sets or data that are continuously produced by some source and produce some final result. The first stage of the pipeline is applied to an initial data to produce an intermediate result that is then sent to the next stage of the pipeline and so on until the final result is computed. Examples of such applications include image set processing where the different stages may be filters, encoders, image comparison or merging and video capture processing and distribution where codecs must be applied on the video flow before being delivered to some device. In this context, a first scheduling problem is to map the pipeline stages on the processors. Subhlok and Vondran [12, 13] show that there exists an optimal interval mapping for a given pipeline and a given platform when communications and processors are homogeneous. An interval mapping is defined as a mapping where only consecutive pipeline stages are mapped on the same processor. However, the cost of switching between stages of the application on one processor is not taken into account. When a new data set arrives on the processor, the first local stage starts to process it as soon as the previous data is output. Then this data set moves from stage to stage until the last local stage, and it is sent to the processor in charge of the following stage. So, at each step of the execution, we switch from one stage to the next one. As a result, if the cost of switching cannot be neglected, several setup times must be added to the processing cost.

Benoit and Robert [3] prove that the basic interval mapping problem without setup times is NP-hard as soon as communications or computations are heterogeneous, even without setup times. For this reason, we restrict this work to homogeneous platforms, where all processors have the same speed and all communication links have the same bandwidth.

The problem of reconfiguration that requires a setup time has been widely studied, and covers a lot of domains (see the survey by Allahverdi et al. [2]). For instance, in semiconductor factories, [14] addressed the problem of wafer-handling robots calibration. They propose a low-cost solution to reduce the robot end effector tolerance requirements, and thus the calibration times, down to 20 times. A solution based on ant colony optimization is proposed to reduce the setup costs in batch processing of different recipes of semiconductors [8, 9]. In the scope of micro-factories, due to the cost of conception and production of micro-assembly cells, micro-assembly cells are being designed with a modular architecture that can perform various tasks, at the cost of a reconfiguration time between them [7]. In the domain of pure computing, setup times may appear when there is a need to swap resources, or to load a different program in memory, e.g., to change the compiler in use [1]. Some authors have also shown interest in using buffers to stock temporary results after each stage of the pipeline, in order to reduce the amount of setups performed. Bryan and Norman [6] consider a flowshop wherein a job consists in  $m$  stages mapped on  $m$  processors, and a processor must be reconfigured after each job to process the next one (in their example, the clean-out of a reactor in a chemical processing facility). They acknowledge that the problem of sequence-dependent setup times, in which a setup time depends on the previous stage and the next one, is NP-hard, and they propose several heuristics. Luh et al. [10] study a scheduling problem in

the manufacturing of gas insulated switchgears. The problems involve significant setup times, strict local buffer capacities, and few possible processing routes.

However, most of those researches focus on the ability of processors to process batches of information (or pieces) from a specific *type*, or *family*, and then to be reconfigured to process batches from another family. The common assumption is that the number of processors is high enough to cover all stages, i.e., each stage is mapped on a distinct processor (one-to-one mapping). In other words, a single processor or a series of processors follows a predefined set of instructions before being reconfigured to process the next batch. Thus, those works mainly focus on merely reducing setup times. When abstracting from setup times, the one-to-one mapping problem can be solved in polynomial time via a binary search algorithm, provided that communications are homogeneous [3]. In our approach, we consider that the number of stages is greater than the number of available processors. We therefore focus on interval mappings, where several consecutive stages are mapped onto the same processor.

In a first step, we tackle the inner-processor scheduling problem, where a single processor has to process several consecutive and dependent pipeline stages. Continuously switching between the stages may lead to a drop in performance, whereas buffering the data and defining a schedule for the processing of stages may limit the number of setups. Hence buffers are introduced to store intermediate results. This makes it possible to perform one stage several times, before switching to the next one. Usually the buffers are limited by the available memory of the system and the buffer size hence influences the possible schedules as it limits the number of repetitions. Several other parameters are also taken into account as the duration of each stage's setup, the homogeneity or heterogeneity of buffers, and the available memory. Eventually, once the inner-scheduling problem has been dealt with, we have to prove the optimality of the overall execution of the pipeline (in terms of throughput).

Starting from the interval mapping results, we tackle in this paper the problem of optimizing the cost of switching between stages mapped on the same processor, depending on the buffer sizes. We formally define the optimization problem in Section 2. The main contributions follow: (i) we provide optimal algorithms when buffers are of fixed (and identical) size within a processor (Section 3); and (ii) we discuss how to allocate memory to buffers on a single processor in Section 4, both from a theoretical perspective (optimal algorithm in some cases), and from a practical point of view (polynomial time heuristics). We formally prove that the heuristics are good approximation algorithms (2-approximation in the worst case), and we evaluate them through a set of simulations. Finally, we conclude and give trails for future works in Section 5.

## 2 Framework

In this section, we formally define the context of our study.

The application is a linear workflow application, or pipeline (see Figure 1). It continuously processes a huge amount of consecutive data sets. Formally, a pipeline is expressed as a set  $S$  of  $n$  stages:  $S = \{S_1, \dots, S_n\}$ . Each data set is fed into the pipeline and traverses the pipeline from one stage to another until the entire pipeline is passed. A stage  $S_i$  receives a task of size  $\delta_i$  from the previous stage, treats the data set which takes a number of  $w_i$  computations,

and outputs data of size  $\delta_{i+1}$ . The output data of stage  $S_i$  is the input data of the next stage  $S_{i+1}$ .

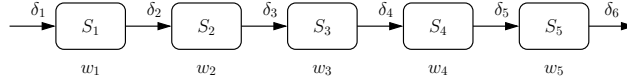


Figure 1: Example of pipeline application.

The target platform is a set  $P$  of  $p$  homogeneous processors  $P = \{P_1, \dots, P_p\}$  fully interconnected as a clique. Each processor  $P_u$  has a processing speed (or velocity)  $v$ , expressed in instructions per time unit, and a memory of size  $M$ . It takes  $X/v$  time units for  $P_u$  to execute  $X$  floating point operations. Each processor  $P_u$  is interconnected with a processor  $P_v$  via a bidirectional communication link  $l_{u,v}$  of bandwidth  $\beta$  (expressed in input size units per time unit). We work with a linear cost model for communications, so it takes  $X/\beta$  time units to send or receive a message of size  $X$  from processor  $P_u$  to processor  $P_v$ . Furthermore communications are based on the one-port model [4, 5], where a given processor can either send or receive in a given time step, but not both. Distinct processor pairs can however communicate in parallel. Communications are non-blocking, i.e., a sender does not have to wait for its message to be received as it is stored in a buffer, and the communications can be covered by the processing times provided that a processor has enough data to process.

Each processor can process data sets from any stage. However, to switch from the execution from a stage  $S_i$  to a stage  $S_j$ , the processor  $P_u$  has to be reconfigured for the next execution. This induces setup times, denoted as  $st$ . Several models are considered: *uniform setup times* ( $st$ ), where all setup times are fixed to the same value, *sequence-independent setup times* ( $st_i$ ), where the setup time only depends on the next stage  $S_i$  to which the processor will reconfigure, and *sequence-dependent setup times* ( $st_{i,j}$ ) that depend on both the current stage  $S_i$  and the next stage  $S_j$ .

The problem with sequence-dependent setup times requires to look for the best setup order in a schedule to minimize the impact of setup times. This has already been proved to be NP-hard, and can be modeled as a Traveling Salesman Problem (TSP) [11]. Hence we will not study this problem in this paper, and we focus on  $st$  and  $st_i$  instead.

To execute a pipeline on a given platform, each processor is assigned an interval of consecutive stages. Hence, we search for a partition of  $[1..n]$  into  $m \leq p$  intervals  $K_k = [I_k, J_k]$  such that  $I_k \leq J_k$  for  $1 \leq k \leq m$ ,  $I_1 = 1$ ,  $I_{k+1} = J_k + 1$  for  $1 \leq k \leq m-1$  and  $J_m = n$ . Interval  $K_k$  is mapped onto a processor  $P_u$ . The allocation function  $a$  makes the correspondence between stages, intervals and processors. For a stage  $S_i$ ,  $a(i) = u$  if it is mapped on  $P_u$ . For a processor  $P_u$ ,  $a'(u) = k$  if  $P_u$  is processing interval  $K_k$ . Once the mapping is fixed, the processor internal schedule has to be decided, since it influences the global execution time. Indeed, each processor is able to perform sequentially its allocated stages. However, setup times are added each time a processor switches from one stage to another. To reduce setup times, a processor may process several consecutive data sets for a same stage. The intermediate results are stored in buffers, and each stage  $S_i$  mapped on  $P_u$  has an input buffer  $B_i$  of size  $m_{i,u}$ .



The sizes of these input buffers depend on the memory size  $M$  available on  $P_u$  and on the number of allocated stages, as well as on the input data sizes. The capacity  $b_{i,u}$  of buffer  $B_i$  is the number of input data sets that the buffer is able to store within the allocated memory  $m_{i,u}$ . Hence, a processor is able to process data sets for a stage  $S_i$  as long as  $B_i$  is not empty, and  $B_{i+1}$  is not full. Actually, if  $S_i$  is the last stage of the interval mapped on  $B_u$ , we allocate an output buffer  $BO_u$  of size  $mo_u$ , with a capacity  $bo_u$ , and this output buffer should not be full, as illustrated in Figure 2.

The current number of input data sets in the buffer  $B_i$  is  $\hat{b}_i$ , while the current number of data sets in the output buffer  $BO_u$  is  $\hat{bo}_u$ . In this paper, we mainly focus on the case where all buffers that are allocated on the same processor  $P_u$  have the same capacity  $b_u$ .

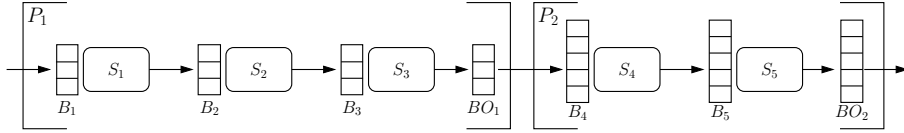


Figure 2: Example of interval mapping with buffers  $b_u$  – Stages  $S_1$  to  $S_3$  are mapped on  $P_1$ :  $b_{1,1} = b_{2,1} = b_{3,1} = bo_1 = b_1 = 3$ , while stages  $S_4$  and  $S_5$  are mapped on  $P_2$ :  $b_{4,2} = b_{5,2} = bo_2 = b_2 = 5$ .

The objective function is to maximize the throughput  $\rho$  of the application,  $\rho = \frac{1}{\mathcal{P}}$ , where  $\mathcal{P}$  is the average period of time between the output of two consecutive data sets. Therefore, we aim at minimizing the period of the application. The period is formally defined in Section 3.1 for single processor scheduling and in Section 3.2 for multi processor scheduling.

### 3 Fixed buffer sizes

In this section, we deal with the problem of both single and multiple processor scheduling with fixed buffer sizes: we consider that buffers are homogeneous within a processor (i.e., they have the same capacity). We first describe a scheduling algorithm for a single processor on which all buffers are identical. Then, we tackle the problem of interval mappings on multiple processors.

#### 3.1 Single processor scheduling

With a single processor, the mapping is known, since stages  $S_1$  to  $S_n$  form a single interval. The problem of inner-scheduling on the processor can be solved in polynomial time by a greedy algorithm. The idea is to maximize the number of data sets that are processed for a stage between each setup. This is done by selecting a stage for which the input buffer is full and the output buffer is empty, so that we can compute exactly  $b$  data sets, where  $b$  is the number of data sets that fits in each buffer. Therefore, we first compute  $b$  data sets for stage  $S_1$ , hence filling the input buffer of  $S_2$ , and then perform a setup so that we can compute  $b$  data sets for stage  $S_2$ , and so on, until these  $b$  data sets have been processed for stage  $S_n$  and exit the pipeline. Then we start with stage  $S_1$  again. This algorithm is called GREEDY-B in the following.

To prove the optimality of GREEDY-B, we introduce a few definitions:

**Definition 1.** During the whole execution, for  $1 \leq i \leq n$ ,

- $nbout$  is the total number of data sets that are output;
- $nbst_i$  is the number of setups performed on stage  $S_i$ ;
- $nbst = \sum_{i=1}^n nbst_i$  the total number of setups;
- $nbcomp_i$  is the average number of data sets processed between two setups on stage  $S_i$ .

We have for  $1 \leq i \leq n$ :

$$nbcomp_i = \frac{nbout}{nbst_i}, \quad nbst_i = \frac{nbout}{nbcomp_i}, \quad \text{and} \quad nbst = \sum_{i=1}^n \frac{nbout}{nbcomp_i}.$$

**Proposition 1.** For each stage  $S_i$  ( $1 \leq i \leq n$ ),  $nbcomp_i \leq b$ .

**Proof.** For each stage  $S_i$ , the number of data sets that can be processed after a setup is limited by its surrounding buffers. Once a setup is done to any stage  $S_i$ , it is not possible to perform more computations than there are data sets or than there is room for result sets. Since all buffers can contain exactly  $b$  data sets, we have  $nbcomp_i \leq b$ .  $\square$

**Proposition 2.** On a single processor with homogeneous buffers, the period can be expressed as:

$$\mathcal{P} = \sum_{i=1}^n \frac{w_i}{v} + \sum_{i=1}^n \frac{st_i}{nbcomp_i}.$$

**Proof.** The period is the total execution time divided by the total number of processed data sets  $nbout$ . The execution time is the sum of the time spent computing, and the time to perform the setups. The computation time is the time to compute each stage once ( $w_i/v$  for stage  $S_i$ ), multiplied by the number of data sets  $nbout$ . The reconfiguration time is the sum of the times required to perform each setup:  $nbst_i \times st_i$ . Therefore, the period can be expressed as:

$$\mathcal{P} = \frac{1}{nbout} \left( \sum_{i=1}^n \frac{w_i}{v} \times nbout + \sum_{i=1}^n st_i \times nbst_i \right),$$

and we conclude the proof by stating that  $nbst_i = \frac{nbout}{nbcomp_i}$ .  $\square$

**Lemma 1.** On a pipeline with homogeneous buffers, the lower bound of the period on a processor is:

$$\mathcal{P}_{min} = \sum_{i=1}^n \left( \frac{w_i}{v} + \frac{st_i}{b} \right).$$

**Proof.** The result comes directly from Propositions 1 and 2:

$$\mathcal{P} = \sum_{i=1}^n \frac{w_i}{v} + \sum_{i=1}^n \frac{st_i}{nbcomp_i} \geq \sum_{i=1}^n \frac{w_i}{v} + \sum_{i=1}^n \frac{st_i}{b} = \mathcal{P}_{min}.$$

$\square$

**Theorem 1.** The scheduling problem on a single processor can be solved in polynomial time, using the GREEDY-B algorithm.

**Proof.** It is easy to see that GREEDY-B is always performing  $b$  computations between two setups, and therefore  $nbcomp_i = b$  for  $1 \leq i \leq n$ . Therefore, the period obtained with this algorithm is exactly  $\mathcal{P}_{min}$ , which is a lower bound on the period and hence it is optimal.  $\square$

### 3.2 Multi processor scheduling

The interval mapping problem on fully homogeneous platforms without setup times can be solved in polynomial time using dynamic programming [12, 13]. However, the problem becomes NP-complete as soon as the platform is heterogeneous [3]. We propose a dynamic programming algorithm for homogeneous platforms that takes into account the setup times in the calculation of a processor's period. To be precise, the calculation of the period is the one obtained by the GREEDY-B algorithm.

Let  $c(j, k)$  be the optimal period achieved by any interval mapping, mapping stages  $S_1$  to  $S_j$  and using  $k$  processors:

$$c(j, k) = \min_{1 \leq l \leq j-1} (\max(c(l, k-1), per(l+1, j))),$$

with the initial condition  $c(j, k) = +\infty$  if  $k > j$ , and where  $per(i, j)$  is the period of the processor on which stages  $S_i$  to  $S_j$  are mapped. Given the memory  $M$ , we can compute the corresponding buffer capacity  $b(i, j) = \lfloor \frac{M}{\sum_{k=i}^{j+1} \delta_k} \rfloor$ , since we restrict in this section to the case with identical buffer capacities. Therefore,

$$per(i, j) = \max \left( \frac{\delta_i \times b(i, j)}{\beta}, \sum_{k=i}^j \left( \frac{w_k}{v} + \frac{st_k}{b(i, j)} \right), \frac{\delta_{j+1} \times b(i, j)}{\beta} \right).$$

Note that  $c(n, p)$  returns the optimal period if and only if the period is actually dictated by the period of the slowest processor, i.e., the slowest processor cannot be in starvation because of buffer sizes. The following theorem ensures that this is true:

**Theorem 2.** *On a pipeline with inner-processor homogeneous buffer capacities  $b_u$ , the period  $\mathcal{P}$  is dictated by the period of the slowest processor.*

**Proof.** We prove the theorem by induction on the number of processors.

Let us consider first a pipeline mapped onto two processors. We aim at proving that the slowest of the two processors is never slowed down either by a lack of input data sets or by a saturation of its output buffer. Let  $P_1$  and  $P_2$  be the two processors;  $BO_1$  is the output buffer of  $P_1$ , and  $BI_{a'(2),2}$  is the input buffer of  $P_2$ . Let  $\mathcal{P}_1$ ,  $\mathcal{P}_2$  and  $b_1$ ,  $b_2$  their respective period and output/input buffer sizes ( $bo_1 = b_1$  and  $BI_{a'(u),2} = b_2$ ). Let  $CT_1$  be the time needed by  $P_1$  to process  $b_1$  data sets, and  $CT_2$  the time needed by  $P_2$  to process  $b_2$  data sets.

We assume that the bandwidth of the communication links is large enough so that communication times are covered by computation times. If not, the period of the system would be dictated by the communication times, whatever the period of the processors.

We consider the two following cases, each of them being split into two sub-cases: (1)  $P_2$  is the slowest processor ( $\mathcal{P}_1 \leq \mathcal{P}_2$ ) with  $b_1 \leq b_2$  (1.a) or  $b_2 \leq b_1$  (1.b) and (2)  $P_1$  is the slowest processor ( $\mathcal{P}_2 \leq \mathcal{P}_1$ ) with  $b_1 \leq b_2$  (2.a) or  $b_2 \leq b_1$  (2.b).

1.  $\mathcal{P}_1 \leq \mathcal{P}_2$  thus  $\frac{CT_1}{b_1} \leq \frac{CT_2}{b_2}$ : in this case the processor  $P_2$  has not to be in starvation. So  $\hat{b}_2 = b_2$  when  $P_2$  is starting a new cycle for a duration of

$CT_2$  unit of time ( $ut$ ). Moreover we assume that at the beginning both the output and the input buffers respectively of  $P_1$  and  $P_2$  are full. If not, this situation occurs after an initialization phase.

(a)  $b_1 \leq b_2$

We have  $\frac{b_2}{b_1} \leq \frac{CT_2}{CT_1}$ . We distinguish three cases:

- If the rational part of  $\frac{CT_2}{CT_1}$  is zero ( $\{\frac{CT_2}{CT_1}\} = 0$ ) then  $CT_2 = qCT_1$  ( $q \in \mathbb{N}$ ). That means  $P_1$  gives exactly enough  $b_1$  output data sets to allow  $P_2$  to enter a new cycle and to perform  $b_2$  input data sets each time  $P_2$  finishes the previous cycle.
- We observe the same conclusion when  $CT_2 \geq \lceil \frac{b_2}{b_1} \rceil CT_1$  because each cycle time  $CT_2$  the processor  $P_1$  produces  $b_2 + x$  output data sets while  $P_2$  only performs  $b_2$  input data sets. From time to time the  $BO_1$  and the input buffer  $B_{I_{a'(2)},2}$  are saturated.
- In the last case we have:

$$CT_2 = qCT_1 + r$$

$$\text{with } q = \left\lfloor \frac{CT_2}{CT_1} \right\rfloor \text{ and } r = CT_2 \bmod CT_1$$

$$\text{and } \frac{b_2}{b_1} CT_1 \leq CT_2 < \left\lceil \frac{b_2}{b_1} \right\rceil CT_1$$

In this platform configuration, the worst case occurs when the processor  $P_2$  is as fast as possible and  $P_1$  is as slow as possible ( $\mathcal{P}_1 = \mathcal{P}_2$ ) and when  $P_1$  and  $P_2$  respectively produces  $b_1$  output data sets or consumes  $b_2$  input data sets at once. So we have also  $b_2 = qb_1 + r$ .

In the following, we propose to prove by induction that  $P_2$  can always start a new cycle without delay. That means that the amount of data sets in  $BO_1$  and  $B_{I_{a'(2)},2}$  is almost equal to  $b_2$  before  $P_2$  enters a new computation cycle.

Let  $C_j$  be the  $j^{\text{th}}$  computation cycle of  $P_2$ . Since  $CT_2 = qCT_1 + r$ , during a cycle  $C_j$  ( $CT_2 ut$ )  $P_1$  is able to output  $qb_1$  data sets and starts a  $(q+1)^{\text{th}}$  before the ending of  $C_j$ . So the cumulative advance of  $P_1$  allows periodically  $P_1$  to finish  $(q+1)$  cycles instead of  $q$ . Let  $i = \lfloor \frac{jr}{b_1} \rfloor$  be the number of times  $P_1$  has finished an extra cycle since the start of  $C_1$ .

We define an induction hypothesis which gives, after the  $j^{\text{th}}$  computation cycle ( $C_j$ ) of  $P_2$ , that the amount of data sets within  $BO_1$  and  $B_{I_{a'(2)},2}$  is larger than  $b_2$  with:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = (jq + i + 1)b_1 - (j - 1)b_2$$

for all  $1 \leq j < CT_1$ .

In any case,  $P_1$  started a new computation cycle exactly  $(jr - iCT_1) ut$  before the end of  $C_j$ . This is because every cycle  $P_1$  starts its last cycle  $r ut$  sooner ( $CT_2 = qCT_1 + r$ ), and each time

an extra cycle of  $P_1$  occurs, the advance of  $P_1$  is decreased by the value of  $CT_1$  (we later refer to this extra time  $P_1$  has to start a cycle as its *advance* over  $C_j$ ). If this hypothesis is verified then  $P_2$  can start  $C_{j+1}$  without delay.

We show in the following that these conditions allow  $P_2$  to enter a new computation cycle without delay for  $j = 1, 2$  ((ii)(iii)). Then considering that these conditions are true for  $j$ , we show that  $P_2$  can enter  $C_{j+1}$  without delay (iv):

- i. After the initialization stage, we have:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = b_1 + b_2$$

$P_2$  can start its first processing cycle ( $C_1$ ) because  $b_2$  input data sets are available for  $P_2$ .

- ii. During  $C_1$  and before the beginning of the next cycle  $C_2$  of  $P_2$ ,  $P_1$  has enough place to produce at least  $qb_1$  output data sets (there is room for  $b_2 = qb_1 + r$  actually). Now:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = (q+1)b_1$$

Since  $b_1 > r$  and thus  $qb_1 + b_1 > qb_1 + r$ ,  $P_2$  is able to enter the processing cycle  $C_2$  because  $(q+1)b_1 > b_2$ . At this time  $P_1$  has already started a computing cycle for  $r$  *ut*. Its advance is too short to be able to finish an extra cycle ( $i = 0$ ).

- iii. After  $C_2$   $P_1$  produced at least  $qb_1$  other new output data and  $P_2$  consumed  $b_2$  input data. At this time:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = (2q+1)b_1 - b_2$$

$P_2$  can enter  $C_3$  only if  $b_2$  is full, that is only if  $(2q+1)b_1 - b_2 \geq b_2$ . This implies that  $2r \leq b_1$ . If this condition is true, then  $P_2$  can indeed start  $C_3$  without delay (and the induction is true), and the advance of  $P_1$  is now  $2r$  *ut*. Otherwise, we have  $2r > b_1$ , and since we are at  $j = 2$  then  $i = \lfloor \frac{2r}{b_1} \rfloor = \lfloor \frac{2r}{b_1} \rfloor = 1$ . In other words the cumulative advance of  $P_1$  was large enough to have performed an extra output of  $b_1$  data sets during  $C_2$ . Thus we can add  $b_1$  to the previous expression of  $\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2}$ . We obtain:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = (2q+1)b_1 - b_2 + b_1 = (2q+1+1)b_1 - b_2$$

The advance of  $P_1$  on the ending of  $C_2$  is now  $(2r - CT_1)ut$ . Since we have indeed  $(2q+1+1)b_1 > 2b_2$  ( $(q+1)b_1 > b_2$ ), and thus  $(2q+1+1)b_1 - b_2 > b_2$ ,  $P_2$  can also start  $C_3$  without delay.

The induction hypothesis is true for  $j = 1, 2$ , and the advance of  $P_1$  is  $(2r - iCT_1)$  in any case because of the value of  $i$ .

- iv. Let us consider that the  $j$  first cycles of  $P_2$  have been started without delay. So the following expression is true at the end of  $C_{j-1}$ , that allows  $P_2$  to start  $C_j$  without delay. Moreover the cumulative advance of  $P_1$  is  $((j-1)r - iCT_1)ut$ . We have:

$$\widehat{b}o_1 + \widehat{b}_{I_{a'(2)},2} = ((j-1)q + i + 1)b_1 - ((j-1) - 1)b_2 \geq b_2$$

The question is if this expression true at the end of  $C_j$ . During  $C_j$ ,  $P_2$  performs  $b_2$  input data sets and  $P_1$  outputs at least  $qb_1$  data sets and its advance on  $P_2$  is  $((j-1)r - iCT_1 + jr)ut = (jr - iCT_1)ut$ . The number of data sets in  $BO_1$  and  $B_{I_{a'(2)},2}$  becomes:

$$\widehat{b}o_1 + \widehat{b}_{I_{a'(2)},2} = ((j-1)q + i + 1)b_1 - ((j-1) - 1)b_2 - b_2 + qb_1$$

$$\widehat{b}o_1 + \widehat{b}_{I_{a'(2)},2} = (jq + i + 1)b_1 - (j-1)b_2$$

Then we obtain that  $\widehat{b}o_1 + \widehat{b}_{I_{a'(2)},2} \geq b_2$  if:

$$(jq + i + 1)b_1 \geq jb_2$$

$$\text{and } i \geq \frac{jr}{b_1} - 1 \quad \text{with } b_2 = qb_1 + r$$

This expression is always true, as  $i = \lfloor \frac{jr}{b_1} \rfloor$  and  $\lfloor x \rfloor > x - 1$ . We can conclude that the induction hypothesis is always true for  $j < CT_1$ .

- v. We can also conclude that the induction hypothesis is always true for any positive  $j$ , as there is a periodic cycle for which the states of buffers and machines are the same. Indeed, we know that in  $CT_1$  units of time  $P_1$  outputs exactly  $b_1$  data sets and loops back: if we start to count time as when  $P_1$  outputted  $x < b_1$  data sets, after exactly  $CT_1 ut$  it will have outputted the rest, and have started to output  $x$  other data sets. Likewise in  $CT_2$  units of times,  $P_2$  will output exactly  $b_2$  data sets. For both, in  $CT_1 \times CT_2$  units of time, they will output  $b_1 \times CT_2$  (respectively  $b_2 \times CT_1$ ) data sets. As the worst case studied is  $\mathcal{P}_1 = \mathcal{P}_2$  and thus  $\frac{CT_1}{b_1} = \frac{CT_2}{b_2}$ , we can conclude that  $b_1 \times CT_2 = b_2 \times CT_1$ , and thus by the time  $j = (k+1)CT_1$ , and thus  $CT_1 \times CT_2$  units of times have elapsed since  $j = kCT_1$ , both processors will be in the exact same state as before (assuming the induction hypothesis is true within that time).

This concludes the case where  $\mathcal{P}_1 \leq \mathcal{P}_2$  and  $b_1 \leq b_2$ .

(b)  $b_2 \leq b_1$

We have  $\frac{CT_1}{CT_2} \leq \frac{b_1}{b_2}$ . We distinguish three cases:

- By using the same arguments as before, if the rational part of  $\frac{CT_1}{CT_2}$  is zero ( $\{\frac{CT_1}{CT_2}\} = 0$ ) then  $CT_1 = qCT_2$  ( $q \in \mathbb{N}$ ). That means  $P_1$  outputs exactly enough  $b_1$  output data sets to allow  $P_2$  to enter  $q$  cycle for a duration of  $CT_2 ut$  each. After a computation cycle of  $P_1$ , the same scenario is repeating for ever. In this case,  $P_2$  has no delay.

- We observe the same conclusion when  $CT_1 \leq \lfloor \frac{b_1}{b_2} \rfloor CT_2$  because the time  $CT_1$  needed by the processor  $P_1$  to output  $b_1$  is shorter than the time for  $P_2$  to consume  $\lfloor \frac{b_1}{b_2} \rfloor b_2$  input data sets. From time to time the  $BO_1$  and the input buffer  $B_{I_{a'(2)},2}$  are saturated.
- In the last case we have:

$$CT_1 = qCT_2 + r$$

$$\text{with } q = \left\lfloor \frac{CT_1}{CT_2} \right\rfloor \text{ and } r = CT_1 \bmod CT_2$$

$$\text{and } \left\lfloor \frac{b_1}{b_2} \right\rfloor CT_2 < CT_1 \leq \frac{b_1}{b_2} CT_2$$

As we assumed in the previous case ( $b_1 \leq b_2$ ), in this platform configuration, the worst case occurs when the processor  $P_2$  is as fast as possible and  $P_1$  is as slow as possible ( $P_1 = \mathcal{P}_2$ ) and when  $P_1$  and  $P_2$  respectively produces  $b_1$  output data sets or consumes  $b_2$  input data sets at once. So we have also  $b_1 = qb_2 + r$ .

In the following, we propose to prove by induction that  $P_2$  can always start  $q$  or  $q + 1$  cycles without delay between two consecutive outputs of  $P_1$ . That means that the amount of data sets in  $BO_1$  and  $B_{I_{a'(2)},2}$  is almost equal to  $(q + 1)b_2$  or  $qb_2$  before  $P_1$  enters a new computation cycle.

Let  $C_j$  be the  $j^{th}$  computation cycle of  $P_1$ . Since  $CT_1 = qCT_2 + r$ , during a cycle  $C_j$  ( $CT_1$  ut)  $P_2$  is able to consume  $qb_2$  data sets and starts a  $(q + 1)^{th}$  after the ending of  $C_j$ . So the cumulative advance of  $P_2$  allows periodically  $P_2$  to perform  $(q + 1)$  cycles instead of  $q$ . Let  $i = \lfloor \frac{qr}{b_2} \rfloor$  be the number of times  $P_2$  has performed an extra cycle since the beginning of  $C_1$ .

We define an induction hypothesis which gives, after the  $j^{th}$  computation cycle ( $C_j$ ) of  $P_1$ , that the amount of data sets within  $BO_1$  and  $B_{I_{a'(2)},2}$  is larger than  $(q + 1)b_2$  or  $qb_2$  with:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = (j + 1)b_1 - (jq + i)b_2$$

for all  $1 \leq j < CT_2$ .

In any case,  $P_2$  started a new computation cycle exactly  $(jr - iCT_2)$  ut before the end of  $C_j$ .

If this hypothesis is verified then  $P_2$  has no delay during  $C_{j+1}$ .

We show in the following that these conditions allow  $P_2$  to repeat its computation cycle without delay for  $j = 1$  (ii). Then considering that these conditions are true for  $j$ , we show that  $P_2$  has no delay for its  $q$  or  $q + 1$  next cycle it has to perform before the ending of  $C_{j+1}$  (iii):

- After the initialization stage, we have:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = b_1 + b_2$$

$P_2$  can start its  $q$  first processing cycles during  $C_1$ , as  $P_1$  and  $P_2$  start at the same time.

- ii. After  $C_1$  and before the beginning of the next cycle  $C_2$  of  $P_1$ ,  $P_1$  outputs  $b_1$  output data sets and  $P_2$  consumed  $qb_2$  data sets and has start the  $(q+1)^{th}$  for  $r$  ut. Now:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = b_1 + b_2 + b_1 - (q+1)b_2$$

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = 2b_1 - qb_2$$

This expression corroborates the general expression of  $\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2}$  for  $j = 1$ . Moreover since  $b_1 = qb_2 + r$ , if  $r \geq \frac{b_2}{2}$  then:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} \geq (q+1)b_2$$

Indeed, the starting of the last cycle of  $P_2$  has begun since  $2r$  ut. So during the next cycle  $C_2$ ,  $P_2$  must start  $(q+1)$  cycles. The previous expression show that  $P_2$  will be able to start all its cycles without delay during  $C_2$ . In this case  $P_2$  will start its last cycle for  $(2r - CT_2)ut$  before the end of  $C_2$  and  $i = 1$ .

If  $r < \frac{b_2}{2}$ , then  $2b_1 - qb_2 = qb_2 + 2r$ . Because of the value of  $r$ , there is only  $q$  new cycle starts in  $C_2$  and the storage of data sets is enough large to make there starts possible without delay. In this case  $P_2$  will start its last cycle for  $(2r)ut$  before the end of  $C_2$  and  $i = 0$ .

- iii. Let us consider that the  $j$  first cycles of  $P_1$  have been started without  $P_2$  delay, that the amount of data sets stored into  $BO_1$  and  $B_{I_{a'(2)},2}$  is the following:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = (j+1)b_1 - (jq+i)b_2$$

Moreover the last cycle of  $P_2$  started  $jr - iCT_2$  ut before the end of  $C_j$ , with  $i = \lfloor \frac{jr}{b_2} \rfloor$  the number of time  $P_2$  starts  $(q+1)$  cycles instead of  $q$  (except during  $C_1$ ).

The condition to make possible the beginning of  $(q+1)$   $P_2$  cycles during  $C_{j+1}$  is

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} \geq (q+1)b_2$$

This means that

$$\frac{(j+1)r}{b_2} - 1 \geq i = \lfloor \frac{jr}{b_2} \rfloor$$

Because  $r < b_2$ , the condition is true only if

$$\lfloor \frac{(j+1)r}{b_2} \rfloor = \lfloor \frac{jr}{b_2} \rfloor + 1$$

So, during  $C_{j+1}$   $P_2$  can perform an extra cycle without delay. The value of  $r$  makes the value of  $i$  to be incremented to  $(i+1)$  at the end of  $C_{j+1}$ . It starts this extra cycle  $((j+1)r - (i+1)CT_2)ut$  before the end of  $C_{j+1}$ .



If the previous condition is wrong,  $\frac{(j+1)r}{b_2} - 1 < i$  or  $r < \frac{i+1}{j+1}b_2$ . Because  $CT_1 = qCT_2 + r$  we have also  $r < \frac{i+1}{j+1}CT_2$ . This implies that  $P_2$  performs only  $q$  cycle between  $C_j$  and  $C_{j+1}$ . But the first cycle of  $P_2$  that ends in this period has started before the beginning of  $C_{j+1}$ . The considered period of time is  $CT_1 + jr - iCT_2$  in which there are only  $q$  cycles  $CT_2$ :

$$CT_1 + jr - iCT_2 = (j+1)r + qCT_2 - iCT_2$$

Since  $r < \frac{i+1}{j+1}CT_2$ :

$$(j+1)r + qCT_2 - iCT_2 < (i+1)CT_2 - iCT_2 + qCT_2$$

$$(j+1)r + qCT_2 - iCT_2 < (q+1)CT_2$$

Therefore:

$$CT_1 + jr - iCT_2 = (q+1)CT_2$$

So there are only  $q$  new cycle starts in this case and as  $P_1$  outputs  $b_2 = qb_1 + r$  new output data sets at the beginning of  $C_{j+1}$ ,  $P_2$  performs its input data without delay and it starts this last cycle  $((j+1)r - iCT_2)$  before the end of  $C_{j+1}$ .

- iv. As with the first studied case ( $b_1 \leq b_2$ ), we can observe there is a periodic cycle on both processors of  $CT_1 \times CT_2$ . This allows to say the induction hypothesis is true for all  $j$ .

This concludes the case where  $\mathcal{P}_1 \leq \mathcal{P}_2$  and  $b_2 \leq b_1$ .

2.  $\mathcal{P}_2 \leq \mathcal{P}_1$  thus  $\frac{CT_2}{b_2} \leq \frac{CT_1}{b_1}$ . The idea here is to prove that the output buffer  $BO_1$  of the first processor  $P_1$  is never saturated. We consider system in its worst configuration regarding this constraint: the output buffer of  $P_1$  ( $BO_1$ ) and the input buffer of  $P_2$  ( $B_{I_{a'(i)},i}$ ) are full when  $P_1$  and  $P_2$  start their respective computation cycle.

- (a)  $b_1 \leq b_2$

We have  $\frac{b_2}{b_1} \geq \frac{CT_2}{CT_1}$ . We distinguish three cases:

- If the rational part of  $\frac{CT_2}{CT_1}$  is zero ( $\{\frac{CT_2}{CT_1}\} = 0$ ) then  $CT_2 = qCT_1$  ( $q \in \mathbb{N}$ ). That means that  $P_2$  consumes exactly  $b_2$  data sets when  $P_1$  outputs  $qb_1 = b_2$  output data sets. Each time  $P_2$  consumes  $b_1$  input data sets,  $P_1$  outputs less than  $b_1$  output data sets because of their respective period. In the case where  $P_1$  is the faster as possible, it fills at most  $qb_1 = b_2$  output data sets. As  $BO_1$  and  $B_{I_{a'(2)},2}$  have a global storage capacity of  $b_1 + b_2$ , these buffers are never saturated in this case.
- We observe the same conclusion when  $CT_2 \leq \lfloor \frac{b_2}{b_1} \rfloor CT_1$  because each cycle time  $CT_2$  processor  $P_2$  is able to consume more than  $\lfloor \frac{b_2}{b_1} \rfloor b_2$  output data sets while  $P_1$  only outputs  $\lfloor \frac{b_2}{b_1} \rfloor b_1$  or  $(\lfloor \frac{b_2}{b_1} \rfloor + 1)b_1$  output data sets. From time to time the  $BO_1$  and the input buffer  $B_{I_{a'(2)},2}$  can be empty.
- In the last case we have:

$$CT_2 = qCT_1 + r$$

$$\text{with } q = \left\lfloor \frac{CT_2}{CT_1} \right\rfloor \text{ and } r = CT_2 \bmod CT_1$$

$$\text{and } \frac{b_2}{b_1}CT_1 < CT_2 \leq \left\lfloor \frac{b_2}{b_1} \right\rfloor CT_1$$

As defined before, the worst case occurs also when  $\mathcal{P}_1 = \mathcal{P}_2$ , when  $BO_1$  and  $B_{I_{a'(i)},i}$  are full at the starting and when  $P_1$  outputs  $b_1$  output data sets at once. The difference is that  $P_2$  consumes one input data set each period  $\mathcal{P}_2$  so as to free the input buffer as slow as possible. So we also have  $b_2 = qb_1 + r$ .

In the following we re-use the same notation as defined in (1.a)  $(j, i)$  and the definition of the cycle  $C_j$  as the  $j^{\text{th}}$  computation cycle of processor  $P_2$ .

Due to the previous constraints introduced to handle this case, we can also re-use the already proved inductive formula of the total number of data sets within  $BO_1$  and  $B_{I_{a'(2)},2}$ :

$$\forall j < CT_1 \quad \widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = (jq + 1 + i)b_1 - (j - 1)b_2$$

In the following we have to prove by induction that  $\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2}$  never exceeds  $b_1 + b_2$ :

- i. After the initialization stage, we have:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = b_1 + b_2$$

$P_2$  can start its first processing cycle ( $C_1$ ) because  $b_2$  input data sets are available for  $P_2$ , and  $P_1$  has enough place to output  $qb_1$  output data sets because of the place freed by  $P_2$  step by step ( $qb_1 < b_2$ ).  $P_1$  has started its last cycle for  $r$  ut. So, during  $C_1$   $BO_1$  and  $B_{I_{a'(2)},2}$  are not saturated.

- ii. By using the previous expression of  $\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2}$ , after  $C_1$  and before the starting of  $C_2$  the total amount of data sets insides the buffer  $BO_1$  and  $B_{I_{a'(2)},2}$  is:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = (q + 1)b_1$$

During the next cycle of  $P_2$  ( $C_2$ ),  $P_2$  is consuming  $b_2$  data sets and  $P_1$  adds at least  $qb_1$  outputs. But  $P_1$  has started cycles that can potentially finish during  $C_2$  sooner, and before the beginning of  $C_3$ . The period of time that we have to consider is  $(r + CT_2)ut$  with  $CT_2 = qCT_1 + r$ . So if  $CT_1 \leq 2r$ ,  $P_1$  is finishing  $(q + 1)$  cycles before the ending of  $C_2$  and  $q$  if not. Since  $P_2$  is consuming  $b_2$  data sets in the same time, we have to verify the following constraint in the worst case ( $i = 1$ ):

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = (2q + 2)b_1 - b_2 \leq b_1 + b_2$$

This constraint is verified only if  $b_1 \leq 2r$  ( $b_2 = qb_1 + r$ ). And this is also the condition to allows  $P_1$  to output  $(q + 1)b_1$

output data set during  $C_2$ . If  $CT_1 > 2r$  ( $i = 0$ ) we have to verify the next constraint with  $i = 0$  and  $qb_1$  outputs instead of  $(q + 1)b_1$ :

$$\widehat{b}o_1 + \widehat{b}_{I_{a'(2)},2} = (2q + 1)b_1 - b_2 \leq b_1 + b_2$$

This constraint is  $qb_1 \leq b_2$  that is always true by definition of  $b_1$  and  $b_2$ .

Thus, by the end of  $C_2$  and before  $C_3$ ,  $P_1$  has been working on its current computation cycle for  $(2r - iCT_1)ut$ , and  $BO_1$  and  $B_{I_{a'(2)},2}$  are never saturated during  $C_2$ .

- iii. Let us consider that during the  $j$  first cycles of  $P_2$ . After  $C_j$ ,  $BO_1$  and  $B_{I_{a'(2)},2}$  are never saturated. We have to prove no saturation occurs during the next cycle  $C_{j+1}$ .

At the end of  $C_j$ , the number of stored data sets within  $BO_1$  and  $B_{I_{a'(2)},2}$  is given by the formula:

$$\widehat{b}o_1 + \widehat{b}_{I_{a'(2)},2} = (jq + 1 + i)b_1 - (j - 1)b_2$$

During the next cycle  $C_{j+1}$ ,  $P_2$  consumes  $b_2$  input data sets while  $P_1$  outputs  $(q + i)b_1$  data sets. As  $P_2$  consumes as fast as  $P_1$  outputs in average, each time  $P_1$  outputs  $b_1$  data sets  $P_2$  has freed  $b_1$  places within the input buffer of  $P_2$ . Since  $b_2 = qb_1 + r$  the  $q$  first computation cycle of  $P_1$  do not increase the number data sets within the buffers. But  $P_1$  has started its first output before the start of  $C_{j+1}$ . If the extra period of time in which  $P_1$  computes data sets is high enough, that extra amount of data sets can potentially be outputted before the ending of  $C_{j+1}$ . This whole period of time is  $(jr - iCT_1 + CT_2)ut$ . So we have to verify that an extra cycle of  $P_1$  does not saturate the buffers.

If this period is larger than  $(q + 1)CT_1$  then an extra cycle is finishing before the ending of  $C_{j+1}$ . This extra cycle occurs when:

$$r \geq \frac{i+1}{j+1}b_1 \quad \text{or} \quad (i+1) \leq \frac{j+1}{b_1}$$

This last constraint is true for instance when:

$$(i+1) = \lfloor \frac{j+1}{b_1} \rfloor$$

This equality means that  $i$  has to be incremented by one at the end of  $C_{j+1}$  to represent the number of extra cycles.

Now we have to verify if an extra cycle does not saturate the buffers. So the next constraints must be true:

$$\widehat{b}o_1 + \widehat{b}_{I_{a'(2)},2} = ((j+1)q + 1 + i + 1)b_1 - (j - 1)b_2 \leq b_1 + b_2$$

We obtain  $(i + 1) \leq \frac{(j+1)r}{b_1}$  what is true because we have  $(i + 1)$  extra cycles during the  $(j + 1)^{th}$  cycles of  $P_2$ . This

constraint remains true without an extra cycle of  $P_1$  during  $C_{j+1}$ .

- iv. As with the two other studied cases, we can observe there is a periodic cycle on both processors of  $CT_1 \times CT_2$ . This allows to say the induction hypothesis is true for all  $j$ .

This concludes the case where  $\mathcal{P}_2 \leq \mathcal{P}_1$  and  $b_1 \leq b_2$ .

(b)  $b_2 \leq b_1$

We have  $\frac{b_1}{b_2} \leq \frac{CT_1}{CT_2}$ . We distinguish three cases:

- By using the same arguments as before, if the rational part of  $\frac{CT_1}{CT_2}$  is zero ( $\{\frac{CT_1}{CT_2}\} = 0$ ) then  $CT_1 = qCT_2$  ( $q \in \mathbb{N}$ ). That means  $P_2$  consumes exactly enough  $qb_2$  input data sets to allow  $P_1$  to output  $qb_1$  data sets for a duration of  $CT_1$  *ut* each. After a computation cycle of  $P_1$ , the same scenario is repeating for ever. In this case,  $P_1$  has always enough place to output its data sets. There is no starvation.
- We observe the same conclusion when  $CT_1 \geq \lceil \frac{b_1}{b_2} \rceil CT_2$  because the time  $CT_1$  needed by processor  $P_1$  to output  $b_1$  is longer than the time for  $P_2$  to consume  $\lceil \frac{b_1}{b_2} \rceil b_2$  input data sets. From time to time the output buffer  $BO_1$  and the input buffer  $B_{I_{a'(2)},2}$  can be empty.
- In the last case we have:

$$CT_1 = qCT_2 + r$$

$$\text{with } q = \left\lfloor \frac{CT_1}{CT_2} \right\rfloor \text{ and } r = CT_1 \bmod CT_2$$

$$\text{and } \frac{b_1}{b_2} CT_2 \leq CT_1 < \left\lceil \frac{b_1}{b_2} \right\rceil CT_2$$

As assumed before, the worst case occurs when the processor  $P_1$  is as fast as possible and  $P_2$  is as slow as possible ( $\mathcal{P}_1 = \mathcal{P}_2$ ), when  $BO_1$  and  $B_{I_{a'(i)},i}$  are full at the starting, when  $P_1$  outputs  $b_1$  output data sets at once and when  $P_2$  consumes one input data set each period  $\mathcal{P}_2$  so as to free the input buffer as slow as possible. So we also have  $b_2 = qb_1 + r$ .

In the following we re-use the same notation as defined in (1.b) ( $j, i$ ) and the definition of cycle  $C_j$  as the  $j^{\text{th}}$  computation cycle of processor  $P_1$ .

Due to the previous constraints introduced to handle this case, we can also re-use the already proved inductive formula of the total number of data sets within  $BO_1$  and  $B_{I_{a'(2)},2}$ :

$$\widehat{b}o_1 + \widehat{b}_{I_{a'(2)},2} = (j+1)b_1 - (jq+i)b_2$$

for all  $1 \leq j < CT_2$ .

In any case, after  $C_j$ ,  $P_2$  has started a new computation cycle for  $(jr - iCT_2)$  *ut* for the same reason explained in the case (1.a).

In the following we prove by induction that  $\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2}$  never exceeds  $b_1 + b_2$ :

- i. After the initialization stage, we have:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = b_1 + b_2$$

$P_2$  can start its  $q+1$  first processing cycles during  $C_1$  because  $P_1$  and  $P_2$  start at the same time and because  $BO_1$  and  $B_{I_{a'(2)},2}$  contain  $b_1 + b_2 > (q+1)b_2$  data sets. So before the ending of  $C_1$  the buffers contain only  $r$  input data sets.  $P_1$  has started its last cycle for  $r$  *ut*. During  $C_1$  the buffers are not saturated.

- ii. After  $C_1$  and before the beginning of the next cycle  $C_2$  of  $P_1$ ,  $P_1$  outputs  $b_1$  output data sets and  $P_2$  consumed  $(q+1)b_2$  data sets. Now:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = b_1 + r = 2b_1 - qb_2$$

Since the next data sets is only adding at once at the end of the new cycle  $C_2$ , and since  $P_2$  is consuming at least  $qb_2$  data sets, the buffer are only decreasing during  $C_2$ . So the condition that the buffers are not saturated during  $C_2$  is:

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} \leq b_1 + b_2$$

With  $\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = b_1 + r$ , the condition is verified because  $r < b_2$ .

During  $C_2$ ,  $P_2$  consumed at least  $qb_2$  input data sets or  $(q+1)b_2$  when  $b_2 \leq 2r$  ( $i=1$ ) as explained several times before. We recall that  $i = \lfloor \frac{jr}{b_2} \rfloor$ . Thus,  $P_1$  started a new computation cycle exactly  $(2r - iCT_2)$  *ut* before the end of  $C_2$ , and the amount of data sets within the buffer is  $2r - ib_2$  at the end of  $C_2$ .

During  $C_2$   $BO_1$  and  $B_{I_{a'(2)},2}$  are not saturated and  $P_1$  has no delay.

- iii. Let us consider now that during the  $j$  first cycles of  $P_1$ , there is no delay on processor  $P_1$  and the amount of data sets within  $BO_1$  and  $B_{I_{a'(2)},2}$ :

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = (j+1)b_1 - (jq+i)b_2$$

As introduced before,  $\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2}$  decreases during a cycle of  $P_1$ . The consequence is that  $\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} \leq b_1 + b_2$  is a sufficient condition to avoid the saturation of the buffer and thus to avoid  $P_1$  to be delayed. After the cycle  $C_j$  we have a total of data sets between  $BO_1$  and  $B_{I_{a'(2)},2}$ :

$$\widehat{bo}_1 + \widehat{b}_{I_{a'(2)},2} = (j+1)b_1 - (jq+i)b_2$$

using  $b_1 = qb_1 + r$  we obtain:

$$jr \leq (i+1)b_2$$

and thus

$$\frac{jr}{b_1} - 1 \leq i = \left\lfloor \frac{jr}{b_1} \right\rfloor$$

that is always true by definition of the floor function. It means that  $P_1$  will never be delayed and the period of both processors  $P_1$  and  $P_2$  is  $\mathcal{P}_1$ .

- iv. As with the other studied cases, we can observe there is a periodic cycle on both processors of  $CT_1 \times CT_2$ . This allows to say the induction hypothesis is true for all  $j$ .

This concludes the case where  $\mathcal{P}_2 \leq \mathcal{P}_1$  and  $b_2 \leq b_1$ .

Now we consider  $j$  processors on which some stages of a pipeline application is mapped using an interval mapping. We assume that the theorem is true for these  $j$  processors, i.e., the period is dictated by the period of the slowest of these processors. We have now to prove that the theorem is still true when we add a  $j + 1^{th}$  processor that processes some more stages.

Since the period of the  $j$  first processors is dictated by the slowest one, we can group these processors as one single processor whose period (the mean duration between two consecutive outputs) is the period of the slowest processor. This period is given by the inner-processor scheduling algorithm. Moreover, the size of the output buffer  $BO_j$  of this processor is  $bo_j$ , the size of the last buffer in this set of processors.

Using this observation, all the previously studied cases can be applied, where  $P_1$  is the single processor replacing the  $j$  first processors, and  $P_2$  is the  $j + 1^{th}$  processor. This concludes the proof.  $\square$

## 4 Variable buffer sizes

In this section, we tackle the problem of allocating the buffers for all stages on a single processor  $P$  from an available memory  $M$ . We focus on platforms with homogeneous data input sizes ( $\delta_i = \delta$ ) and setup times ( $st_i = st$ ). First we propose an allocation algorithm, ALL-B, which returns buffers of identical capacities, and we discuss its optimality in Section 4.1. In Section 4.2, we design polynomial time heuristics for the case when the algorithm is not optimal, and we evaluate them through simulations in Section 4.3. Finally, we briefly discuss in Section 4.4 the cases with  $\delta_i$  or  $st_i$ .

### 4.1 Allocation algorithm

Given the memory  $M$  and the size of the data  $\delta$ , since there are  $n + 1$  buffers if the processor is processing  $n$  stages, if we want all buffers to contain the same number of data sets, then the maximum number of data sets that can fit in each buffer can be computed as:

$$b = \left\lfloor \frac{M}{(n+1)\delta} \right\rfloor.$$

The actual memory allocated for each buffer is  $m_i = m = b\delta = \left\lfloor \frac{M}{n+1} \right\rfloor$ . The memory used by this allocation is then  $(n+1)\delta \times b \leq M$ , and we call

$\mathcal{R} = M - (n + 1)\delta \times b$  the *remainder* of memory after the allocation, i.e., the unused part of the memory. This algorithm is called ALL-B.

We prove that this allocation algorithm is optimal if there is no remainder in the memory after the application of the algorithm, i.e., if it returns  $\mathcal{R} = 0$ .

**Theorem 3.** *The algorithm ALL-B is optimal on a single processor (i.e., the period is minimized with this allocation) when  $\mathcal{R} = M - (n + 1)\delta \times \lfloor \frac{M}{(n+1)\delta} \rfloor < \delta$ .*

**Proof.** First note that since all data sets have the same size, the maximum number of data sets that can fit in memory is  $\lfloor M/\delta \rfloor$ , and the remainder of the memory cannot be used. Let  $M' = (n + 1)\delta \times \lfloor \frac{M}{(n+1)\delta} \rfloor$ . We assume in this theorem that  $M - M' < \delta$ , i.e., even if  $M' \leq M$ , both memories can contain exactly the same number of data sets. Moreover,  $b = \frac{M'}{(n+1)\delta}$  is an integer number of data sets. Therefore, we assume in the following that the memory is  $M = M'$ , so that we do not need to consider integer parts anymore.

Next, we need to express the period of a solution in which buffers may have different sizes, i.e., the  $i$ -th buffer can contain  $b_i$  data sets, for  $1 \leq i \leq n + 1$ . We can reuse the result of Lemma 1, and the only difference comes from the fact that we need to amend Proposition 1 as  $nbcomp_i \leq \min(b_i, b_{i+1})$  (for  $1 \leq i \leq n$ ), since the input (resp. output) buffer of stage  $S_i$  can contain  $b_i$  (resp.  $b_{i+1}$ ) data sets, and once a setup is done for a stage  $S_i$ , it is not possible to perform more computations than there are data sets or than there is room for result sets. Finally, since we consider that all setup times are identical, we have:

$$\mathcal{P}_{min}(b_1, \dots, b_{n+1}) = \sum_{i=1}^n \frac{w_i}{v} + \sum_{i=1}^n \frac{st}{\min(b_i, b_{i+1})}.$$

We want to prove by induction on  $n$  that the minimum of this function is reached for  $b_1 = \dots = b_{n+1} = b = \frac{M}{(n+1)\delta}$ , under the constraint that  $\sum_{i=1}^{n+1} b_i \delta = M$ . Note that we do not need in the following to assume that the  $b$  and  $b_i$ 's are integer, but the condition on  $\mathcal{R}$  ensures that the value of  $b$  is an integer when considering a pipeline of  $n$  stages.

- For  $n = 1$ , we have

$$\mathcal{P}_{min}(b_1, b_2) = \sum_{i=1}^1 \left( \frac{w_i}{v} + \frac{st}{\min(b_i, b_{i+1})} \right) = \frac{w_1}{v} + \frac{st}{\min(b_1, b_2)}.$$

Knowing that  $b_1 + b_2 = M/\delta = 2b$ , we can express  $b_1$  and  $b_2$  as  $b_1 = b + \varepsilon$  and  $b_2 = b - \varepsilon$ . Thus:

$$\mathcal{P}_{min}(b_1, b_2) = \frac{w_1}{v} + \frac{st}{\min(b + \varepsilon, b - \varepsilon)},$$

and this function is clearly minimized when  $\varepsilon = 0$ , i.e.,  $b_1 = b_2 = b$ .

- Let us assume now that the result holds true for  $n - 1$ , and let  $b_{n+1}$  be the size of the buffer that is added when considering  $n$  stages instead of  $n - 1$ . The memory available for the  $n - 1$  first stages is therefore  $M - b_{n+1}\delta$ . Recall that values of  $b_i$ 's may not be integer at this point. The period can be expressed as follows, reusing the period for the  $n - 1$  first stages  $\mathcal{P}_{n-1}$ :

$$\mathcal{P}_{min}(b_1, \dots, b_{n+1}) = \mathcal{P}_{n-1} + \frac{w_n}{v} + \frac{st}{\min(b_n, b_{n+1})}.$$

By induction, the minimum value for  $\mathcal{P}_{n-1}$  is obtained when  $b_1 = \dots = b_n = \frac{M - b_{n+1}\delta}{n\delta}$ . We then have:

$$\mathcal{P}_{n-1} = \sum_{i=1}^{n-1} \frac{w_i}{v} + n \times \frac{st}{M/n\delta - b_{n+1}/n} = \sum_{i=1}^{n-1} \frac{w_i}{v} + \frac{n^2 \times st}{M/\delta - b_{n+1}};$$

$$\mathcal{P}_{\min}(b_1, \dots, b_{n+1}) = \sum_{i=1}^n \frac{w_i}{v} + \frac{n^2 \times st}{M/\delta - b_{n+1}} + \frac{st}{\min(b_n, b_{n+1})}.$$

Let us assume first that  $b_{n+1} \leq b_n$ . We then have  $\min(b_n, b_{n+1}) = b_{n+1}$ , and the goal is to minimize  $f(x) = \frac{n^2}{M/\delta - x} + \frac{1}{x}$ , where  $x$  corresponds to  $b_{n+1}$ . The first derivative of this function is  $f'(x) = \frac{n^2}{(M/\delta - x)^2} - \frac{1}{x^2}$ , and its only positive root is  $x = \frac{M}{(n+1)\delta}$ , which corresponds to a minimum of the function  $f(x)$ . For this value of  $b_{n+1}$ , we have  $b_1 = \dots = b_n = \frac{M}{(n+1)\delta} = b_{n+1}$ , and hence the hypothesis  $b_{n+1} \leq b_n$  is verified, and the solution that minimizes the period is such that all  $b_i$ 's are equal.

However, if  $b_{n+1} \geq b_n$ , we need to minimize a function of  $b_n$ , and  $\frac{n^2}{M/\delta - b_{n+1}} + \frac{st}{b_n} \geq \frac{n^2}{M/\delta - b_n} + \frac{st}{b_n}$ . We obtain the same function  $f(x)$  as above, and hence the lower bound on the period is minimized for  $b_n = \frac{M}{(n+1)\delta}$ . Moreover, the smaller  $b_{n+1}$ , the smaller the expression of the period, since the only term in  $b_{n+1}$  is  $\frac{n^2}{M/\delta - b_{n+1}}$ . Therefore, the period is minimized for  $b_{n+1} = b_n$ , hence obtaining the same solution as in the first case  $b_{n+1} \leq b_n$ .

Finally, we conclude by saying that for the pipeline with  $n$  stages, the value of  $b$  is an integer thanks to the condition on the memory, and the lower bound on the period is reached according to Theorem 1.  $\square$

## 4.2 Memory remainder

If there is a remainder in the memory after the allocation of buffers ALL-B, it is under certain conditions possible to use this remainder to increase the size of some buffers. It may also be possible to have another allocation, not based on ALL-B, that would make better or full use of the memory. In both cases, the period achieved by some scheduling algorithm may be lower than the one we have.

The first problem we have to deal with is the fact that distributing the remainder amongst buffers will inevitably lead to heterogeneous buffer capacities, while we focused on buffers of identical size within a same processor so far. We extend the GREEDY-B algorithm of Section 3.1 for buffers of different sizes. Instead of choosing a stage where the input buffer is full and the output buffer is empty, we choose a stage where either the input buffer is full and we have enough space to fully empty it, or the output buffer is empty and we have enough data sets to compute to fully fill it. That way, we still maximize the amount of data sets processed after each setup: we are limited by the lowest capacity buffer, which is either a fully emptied input buffer, or a fully filled output buffer. This algorithm is called GREEDY-BI, and it is formalized as Algorithm 1.

This algorithm may not return an optimal schedule in the general case, but we can prove its optimality in the case of *multiple buffers*, i.e., each buffer



```

 $S_c \leftarrow S_1;$ 
while true do
   $S_i \leftarrow \text{null};$ 
  repeat
    Wait for data in buffers;
     $S_i \leftarrow$  latest stage that observes Condition 1 or 2;
  until  $S_i \neq \text{null};$ 
  Setup from  $S_c$  to  $S_i$  ( $S_c \leftarrow S_i$ );
  Perform  $\min(b_i, b_{i+1})$  computations (Condition 1 or 2 guarantees that
  exact amount);
end
Condition 1:  $\hat{b}_i \geq b_{i+1}$  &  $\hat{b}_{i+1} = 0$  (ensures we can fully fill an empty
output buffer);
Condition 2:  $\hat{b}_i = b_i$  &  $(b_{i+1} - \hat{b}_{i+1}) \geq b_i$  (ensures we can fully empty a
full input buffer);

```

**Algorithm 1:** GREEDY-BI.

capacity is a multiple of the capacities of both its predecessor and its successor: for  $1 \leq i \leq n$ ,  $\min(b_i, b_{i+1}) \mid \max(b_i, b_{i+1})$ .

Note that if GREEDY-BI is applied with multiple buffers, the deterministic behavior of the algorithm is guaranteed, since a series of  $x$  batches of data sets will always perfectly fit any other  $x$  times larger buffer, and any large buffer can always be split into  $y$  equal parts. We can easily adapt Proposition 1 for the heterogeneous case, with  $nbcomp_i \leq \min(b_i, b_{i+1})$  (the amount of computations possible after each setup on  $S_i$  is limited by the smallest of both input and output buffers). Finally, the optimal period, which can be reached using GREEDY-BI (see Theorem 4), is  $\mathcal{P}_{min} = \sum_{i=1}^n \left( \frac{w_i}{v} + \frac{st}{\min(b_i, b_{i+1})} \right)$ .

**Theorem 4.** *The scheduling problem with multiple buffers on a single processor can be solved in polynomial time, using the GREEDY-BI algorithm.*

**Proof.** We first amend Proposition 1 as  $nbcomp_i \leq \min(b_i, b_{i+1})$  (for  $1 \leq i \leq n$ ), and thus, according to this and Proposition 2, the lower bound of the period, as showed in Lemma 1, is  $\mathcal{P}_{min} = \sum_{i=1}^n \left( \frac{w_i}{v} + \frac{st_i}{\min(b_i, b_{i+1})} \right)$ .

With the GREEDY-BI algorithm, we setup on a stage if and only if Condition 1 or 2 is observed (see Algorithm 1):

- Condition 1 is reached if and only if  $b_i \geq b_{i+1}$  and we can compute enough data sets to fill an empty  $b_{i+1}$ . Therefore,  $b_i \geq b_{i+1}$  and  $nbcomp_i = b_{i+1}$  in this case.
- Condition 2 is reached if and only if  $b_i \leq b_{i+1}$  and we can compute enough data sets to empty a full  $b_i$ . Therefore,  $b_i \leq b_{i+1}$  and  $nbcomp_i = b_i$ .

We always have:  $Setup_{S_i} \iff \text{Condition 1 or Condition 2}$ , which corresponds to  $Setup_{S_i} \iff (b_i \geq b_{i+1} \text{ and } nbcomp_i = b_{i+1}) \text{ or } (b_i \leq b_{i+1} \text{ and } nbcomp_i = b_i)$ . The logical outcome is that  $Setup_{S_i} \iff nbcomp_i = \min(b_i, b_{i+1})$ .

This means that using GREEDY-BI, the number of computations per setup for one stage is constant and is always  $\min(b_i, b_{i+1})$ . Since the value is con-

stant, it is also the value of the average number of computations per setup:  $\forall i, nbcomp_i = \min(b_i, b_{i+1})$ .

According to Proposition 2, for any scheduling algorithm the period is  $\mathcal{P} = \sum_{i=1}^n \left(\frac{w_i}{v}\right) + \sum_{i=1}^n \left(\frac{st_i}{nbcomp_i}\right)$ , and therefore the period obtained with GREEDY-BI is  $\sum_{i=1}^n \left(\frac{w_i}{v}\right) + \sum_{i=1}^n \left(\frac{st_i}{\min(b_i, b_{i+1})}\right)$ , which corresponds exactly to  $\mathcal{P}_{min}$ , hence concluding the proof.  $\square$

We now give a counter example to illustrate why ALL-B may not be optimal, since it is leaving some memory unused:

**Proposition 3.** *Given an application with homogeneous setup times  $st$  and input sizes  $\delta$ , the buffer allocation ALL-B may not give an optimal solution if  $\mathcal{R} > \delta$ .*

**Proof.** Let us consider a single processor, with a memory  $M = 20$ , and a speed  $v = 1$ . A total of  $n = 6$  stages are mapped on this processor, and we have  $\delta = w = st = 1$ .

There are seven buffers, and therefore ALL-B returns buffers of size  $b = 2$ , and the remainder is  $\mathcal{R} = 20 - 2 \times 7 = 6$ . The optimal period is obtained by scheduling the stages with the GREEDY-B algorithm (see Theorem 1), and therefore:

$$\mathcal{P} = \sum_{i=1}^6 \left(\frac{w_i}{v}\right) + \sum_{i=1}^6 \left(\frac{st}{b}\right) = 6 + \left(\frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2}\right) = 9.$$

However, let us consider the following allocation:  $b_1 = b_2 = b_3 = b_4 = 2$  and  $b_5 = b_6 = b_7 = 4$ . This allocation uses all the memory, and it corresponds to the definition of *multiple buffers*. Therefore, the optimal period is obtained by scheduling the stages with the GREEDY-BI algorithm, and:

$$\mathcal{P} = \sum_{i=1}^6 \left(\frac{w_i}{v}\right) + \sum_{i=1}^6 \left(\frac{st}{\min(b_i, b_{i+1})}\right) = 6 + \left(\frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4}\right) = 8.5.$$

This allocation leads to a smaller period than ALL-B, which concludes the proof.  $\square$

### Heuristics for ALL-B allocation with a remainder.

We developed several heuristics to deal with the memory remainder created by ALL-B. Note that after allocating buffers with ALL-B,  $\forall 1 \leq i \leq n+1, b_i = b$ , and there is not enough memory left to have  $\forall 1 \leq i \leq n+1, b_i = b+1$  (since  $\mathcal{R} < (n+1)\delta$ ). In some cases however, it is still possible to use  $\mathcal{R}$  to increase the size of several (but not all) buffers. According to Proposition 3, the use of this remainder may lead to a decrease of the period. We restrict to the construction of *multiple buffers* as defined above, so that we are able to find optimally the period thanks to the GREEDY-BI algorithm. Hence, if there is enough memory to increase the size of buffers by steps of  $b$ , and if there is at least  $2b\delta$  memory left, then the size of two consecutive buffers can be doubled, resulting in halving the number of setups for the corresponding stage.

**H1** (see Algorithm 2) – The first algorithm allocates batches of  $2b\delta$  to increase the size of the two last buffers by  $b$  each, and continues to increase them

as long as  $2b\delta$  memory units are available. According to the expression of the period, increasing the size of these two consecutive buffers reduces the setup times for  $S_n$ , while keeping the same values everywhere else.

**H2** (see Algorithm 3) – The second algorithm starts off by doubling the size of the two last buffers if there are  $2b\delta$  memory units left, then will continue to increase the capacity of the adjacent buffers by  $b$  as long as  $b\delta$  memory units are still available. Note that since  $\mathcal{R} < (n+1)\delta$ , the algorithm is guaranteed to end before having doubled the size of all buffers.

**Data:**  $M$  – the memory available  
**Output:**  $m_i$  – the memory allocated for each buffer  $B_i$ , and  $\mathcal{R}$  – the remainder.

Apply ALL-B: current buffer sizes are  $m_i = b\delta$ , capacity is  $b_i = b$ , the remainder is  $\mathcal{R} = M - (n+1)b\delta$ ;

**while**  $\mathcal{R} \geq 2b\delta$  **do**  
      $m_{n+1} \leftarrow m_{n+1} + b\delta$ ;  $b_{n+1} \leftarrow b_{n+1} + b$ ;  
      $m_n \leftarrow m_n + b\delta$ ;  $b_n \leftarrow b_n + b$ ;  
      $\mathcal{R} \leftarrow \mathcal{R} - 2b\delta$ ;  
**end**

**Algorithm 2:** Heuristic H1 for memory remainder reallocation.

**Data:**  $M$  – the memory available  
**Output:**  $m_i$  – the memory allocated for each buffer  $B_i$ , and  $\mathcal{R}$  – the remainder.

Apply ALL-B: current buffer sizes are  $m_i = b\delta$ , capacity is  $b_i = b$ , the remainder is  $\mathcal{R} = M - (n+1)b\delta$ ;

**if**  $\mathcal{R} \geq 2b\delta$  **then**  
      $k \leftarrow n+1$ ;  
     **while**  $\mathcal{R} \geq b\delta$  **do**  
          $m_k \leftarrow m_k + b\delta$ ;  $b_k \leftarrow b_k + b$ ;  
          $\mathcal{R} \leftarrow \mathcal{R} - b\delta$ ;  
          $k \leftarrow k - 1$ ;  
     **end**  
**end**

**Algorithm 3:** Heuristic H2 for memory remainder reallocation.

### Performance of the heuristics.

Given the available memory  $M$ ,

- $\mathcal{P}_b(M)$  is the period obtained if  $\forall i \in [1, n+1], b_i = b$ ;
- $\mathcal{P}_{algo}(M)$  is the period obtained by one of the heuristics (it may be specified as  $\mathcal{P}_{H1}$ , or  $\mathcal{P}_{H2}$ );
- $\mathcal{P}_{opt}(M)$  is the optimal (minimal) period that can be achieved with memory  $M$ .

We compute the value of  $b$  obtained by ALL-B algorithm, and therefore  $M = b(n+1)\delta + \mathcal{R}$ , with  $\mathcal{R} < (n+1)\delta$ . It has already been proved (see

Theorem 3) that if there is no remainder after ALL-B,  $\mathcal{P}_b(M)$  is optimal. More formally:

$$M = b(n+1)\delta \iff \mathcal{P}_b(M) = \mathcal{P}_{opt}(M) .$$

We define  $M^* = (b+1)(n+1)\delta = M + (n+1)\delta - \mathcal{R}$ . With a memory  $M^*$ , there is also no remainder and  $\mathcal{P}_{b+1}(M^*) = \mathcal{P}_{opt}(M^*)$ . We first prove that both  $\mathcal{P}_{algo}(M)$  and  $\mathcal{P}_{opt}(M)$  can be bounded by  $\mathcal{P}_b(M)$  and  $\mathcal{P}_{b+1}(M^*)$ :

**Lemma 2.** *We have  $\mathcal{P}_b(M) \geq \mathcal{P}_{algo}(M) \geq \mathcal{P}_{opt}(M) \geq \mathcal{P}_{b+1}(M^*)$ .*

**Proof.** By definition, we have  $\mathcal{P}_{algo}(M) \geq \mathcal{P}_{opt}(M)$ . For the upper bound, both algorithms (H1 and H2) are potentially improving  $\mathcal{P}_b(M)$  by exploiting the remainder, and the period cannot be increased by the allocation of the remainder of the memory.

For the lower bound, note that  $\mathcal{P}_{b+1}(M^*)$  is the optimal period with memory  $M^* > M$ , and therefore  $\mathcal{P}_{opt}(M)$  cannot be better, otherwise we would have a better solution with  $M^*$  that would not use all memory.  $\square$

**Theorem 5.** *The three algorithms ALL-B, H1 and H2 are  $\frac{b+1}{b}$ -approximation algorithms.*

**Proof.** Let  $W = \sum_{i=1}^{n+1} \left(\frac{w_i}{v}\right)$ . We have  $\mathcal{P}_b(M) = W + \frac{(n+1)st}{b}$ , and  $\mathcal{P}_{b+1}(M^*) = W + \frac{(n+1)st}{b+1}$ . Therefore,

$$\frac{\mathcal{P}_b(M)}{\mathcal{P}_{b+1}(M^*)} = \frac{W + \frac{(n+1)st}{b}}{W + \frac{(n+1)st}{b+1}} \leq \frac{\frac{(n+1)st}{b}}{\frac{(n+1)st}{b+1}} = \frac{b+1}{b} ,$$

since  $W > 0$  and  $\frac{(n+1)st}{b+1} \leq \frac{(n+1)st}{b}$ .

Finally, thanks to Lemma 2, we have:

$$\mathcal{P}_{algo}(M) \leq \mathcal{P}_b(M) \leq \frac{b+1}{b} \mathcal{P}_{b+1}(M^*) \leq \frac{b+1}{b} \mathcal{P}_{opt}(M) ,$$

which concludes the proof (recall that  $\mathcal{P}_b(M)$  is the period obtained by algorithm ALL-B). Note that the worst approximation ratio is achieved for  $b = 1$ , and then we have 2-approximation algorithms. However, when  $b$  increases, the period achieved by the algorithms tend to the optimal solution.  $\square$

### 4.3 Simulation results

We evaluate the quality of the heuristics for buffer allocation with some simulations. We vary both the pipeline size (i.e., the number of buffers to allocate), and the memory  $M$  of the processor. Both parameters have an impact on the value of  $b$  and the remainder, and hence on the efficiency of the heuristics.

Therefore, we conduct two simulations. In the first one, the available memory is fixed to  $M = 150$  and the size  $n$  of the pipeline increases:  $3 \leq n \leq 50$ . Thus, if there are more stages, less memory is available for each buffer. In the second simulation, the size of the pipeline is fixed to  $n = 5$ , but the available memory  $M$  increases:  $6 \leq M \leq 50$ . Without loss of generality, we assume that  $\delta = 1$ , and therefore with  $n = 5$  and  $M = 6$ , there is just enough memory to allocate buffers of capacity one to all stages. When  $M$  increases, the capacity  $b$  obtained by ALL-B increases.

We run the two heuristics and compute the period for each configuration ( $\mathcal{P}_{H1}$  and  $\mathcal{P}_{H2}$ ), as well as the period  $\mathcal{P}_b$  obtained by  $\mathcal{P}_b(M)$  (algorithm ALL-B without using the remainder), and the lower bound on the period  $\mathcal{P}_{b+1}$  achieved by  $\mathcal{P}_{b+1}(M^*)$  (algorithm ALL-B with capacities  $b + 1$ , assuming that we have enough memory). As we can observe in Figures 3 and 4, both  $\mathcal{P}_{H1}$  and  $\mathcal{P}_{H2}$  are included in  $[\mathcal{P}_b, \dots, \mathcal{P}_{b+1}]$ . We can also observe that  $\mathcal{P}_{H2}$  is always at least equal to  $\mathcal{P}_{H1}$ , and often better (lower period).

In Figure 3,  $\mathcal{P}_{H1}$  tends to always be very close to  $\mathcal{P}_b$ , while  $\mathcal{P}_{H2}$  only gets close when the remainder is small. This is due to the more restrictive nature of H1, and the way it uses the remainder. At each iteration of H1, it needs exactly  $2b\delta$  supplementary memory, while only  $b\delta$  are required for H2. Also, note that after doubling the size of the two last buffers, for every new  $2b\delta$  available, H2 is doubling the size of two more buffers, thus halving the value of two more  $\frac{1}{b}$  terms in the setup times part of the period. H1 keeps on halving the same last fraction, which has less impact on the overall value of the period.

In Figure 4, while  $\mathcal{P}_{H2}$  is still better than  $\mathcal{P}_{H1}$ , we can see that both have the same value as soon as  $M \geq 18$ , which corresponds to  $b \geq 3$  for ALL-B. In fact, for any value of  $n$ , there is a breaking point in the values of  $M$  for which both heuristics do not improve ALL-B anymore, which corresponds to  $2b \geq n + 1$ . Since the value of the remainder is always lower than  $(n + 1)\delta$  by definition, and the required remainder for the heuristics to have an effect is  $2b\delta$ , whenever the value of  $M$  is high enough so that  $2b \geq n + 1$ , both heuristics return the same result as ALL-B. Note however that the worst case is  $b = 1$ , since that the approximation ratio is then  $b + 1/b = 2$ , while the ratio tends to one when  $b$  increases. Therefore, the heuristics do not improve the solution when there is a lot of memory, but ALL-B becomes very close to the optimal solution, as can be seen in the figure.

#### 4.4 With heterogeneous data input sizes or setup times

The case of heterogeneous setup times ( $st_i$ ) is kept for future work, since it turns out to be much more complex. Indeed, allocating buffers while taking setup times into account requires to prioritize higher setup times by allocating larger buffer capacities. However, this requires both the input and output buffers of the corresponding stage to be larger, and it will inevitably lead to side effects on surrounding stages.

For heterogeneous data input sizes ( $\delta_i$ ), we can use a variant of the ALL-B algorithm to allocate buffers of identical capacities, in terms of data sets. Indeed, the amount of memory distributed to a given buffer should be proportional to the relative size of its data sets in comparison to the total need:  $m_i = \left\lfloor \frac{\delta_i}{\sum_{k=1}^{n+1} \delta_k} \times M \right\rfloor$ . Then, the number of data sets that can fit in buffer  $i$  is  $b_i = \left\lfloor \frac{m_i}{\delta_i} \right\rfloor = \left\lfloor \frac{M}{\sum_{k=1}^{n+1} \delta_k} \right\rfloor = b$ . This allocation returns buffers of same size, we still call it the ALL-B allocation.

In this case, the memory used is  $\sum_{i=1}^{n+1} b \times \delta_i \leq M$ , and the remainder is  $\mathcal{R} = M - \sum_{i=1}^{n+1} b \times \delta_i$ . However, even if there is no remainder, the allocation may not be optimal:

**Proposition 4.** *Given an application with homogeneous setup times  $st$  and heterogeneous input sizes  $\delta_i$ , the buffer allocation ALL-B may not give an optimal*

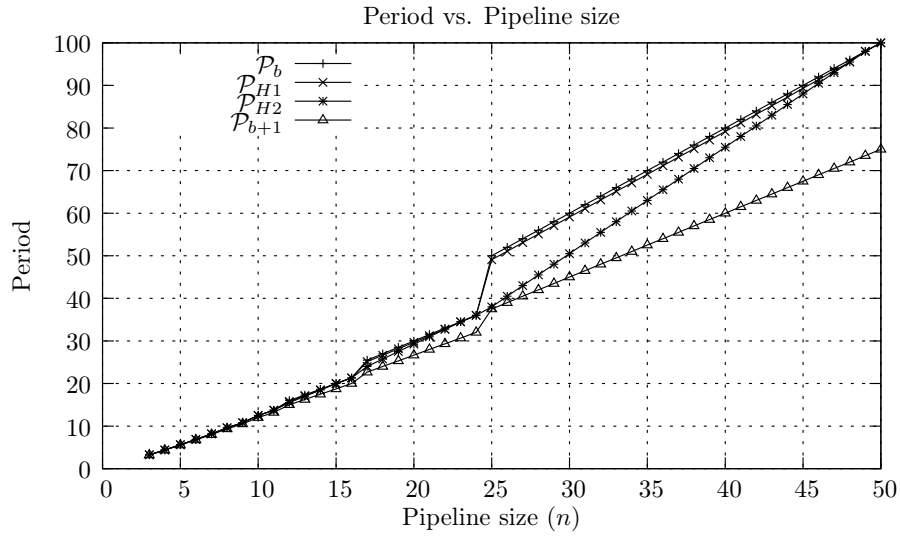


Figure 3: Simulation 1:  $3 \leq n \leq 50$ .

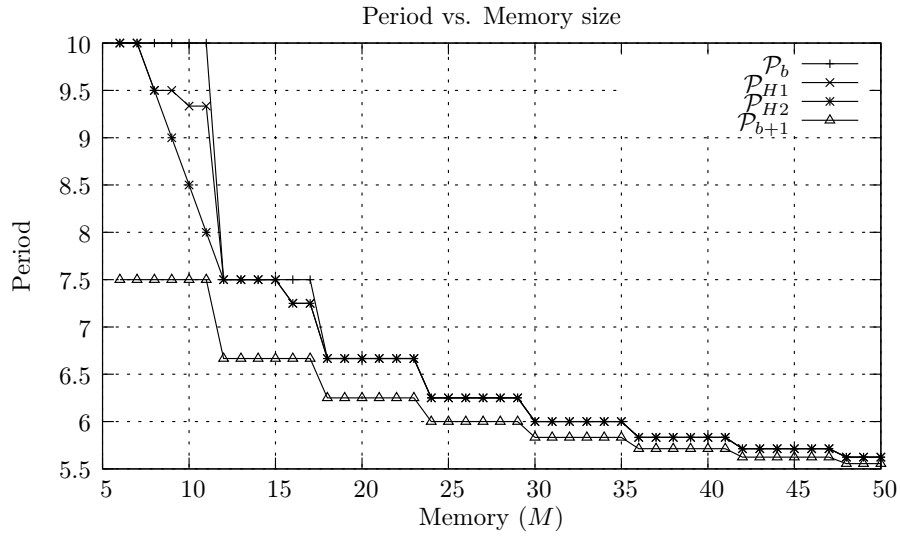


Figure 4: Simulation 2:  $6 \leq M \leq 50$ .

solution, even if the remainder  $\mathcal{R} = 0$ .

**Proof.** Let us consider a single processor, with a memory  $M = 301$ , speed  $v = 1$ . There are  $n = 4$  stages with  $w = st = 1$ . The different input sizes are:  $\delta_1 = 20, \delta_2 = 20, \delta_3 = 1, \delta_4 = 1, \delta_5 = 1$  ( $\delta_5$  is the output size of  $S_4$ ).

ALL-B returns buffers of size  $b = 7$ , and the remainder is  $\mathcal{R} = 301 - (20 \times 7 + 20 \times 7 + 1 \times 7 + 1 \times 7 + 1 \times 7) = 0$ . The optimal period is obtained by scheduling the stages with the GREEDY-B algorithm (see Theorem 1), and therefore:

$$\mathcal{P} = \sum_{i=1}^4 \left( \frac{w_i}{v} \right) + \sum_{i=1}^4 \left( \frac{st}{b} \right) = 4 + \left( \frac{1}{7} + \frac{1}{7} + \frac{1}{7} + \frac{1}{7} \right) = 4.571 .$$

However, let us consider the following allocation:  $b_1 = b_2 = 6$  and  $b_3 = b_4 = b_5 = 18$ . This allocation uses less memory, yet has way higher capacity buffers for  $b_3$  to  $b_5$ , with the only trade-off being the reduction of the capacity of  $b_1$  and  $b_2$  by one. This allocation corresponds to the definition of multiple buffers. Therefore, the optimal period is obtained by scheduling the stages with the GREEDY-BI algorithm, and:

$$\mathcal{P} = \sum_{i=1}^4 \left( \frac{w_i}{v} \right) + \sum_{i=1}^4 \left( \frac{st}{\min(b_i, b_{i+1})} \right) = 4 + \left( \frac{1}{6} + \frac{1}{6} + \frac{1}{18} + \frac{1}{18} \right) = 4.444 .$$

This allocation leads to a smaller period than ALL-B, which concludes the proof.  $\square$

## 5 Conclusion

In this paper, we present solutions to the problem of optimizing setup times and buffer use for pipeline workflow applications. For the problem of fixed buffer sizes, of identical size within a same processor, we provide an optimal greedy algorithm for a single processor, and a dynamic programming algorithm for multiple processors. In the latter case, the application period is equal to the period of the slowest processor. In the case of variable buffer sizes, we tackle the problem of distributing the available processor memory into buffers such that the period is minimized. When the memory allocation results in no remainder (all the memory is used), the algorithm turns out to be optimal, and we propose some approximation algorithms for the other cases.

In future work, we plan to consider sequence-dependent setup times ( $st_{i,j}$ ), a problem that is already known to be NP-complete. We envisage the design of competitive heuristics, whose performance will be assessed through simulation. Furthermore, for the  $st_i$  case, we plan to investigate the memory allocation problem on a single processor. On the long term, we will consider the case of heterogeneous buffer capacities  $b_i$ . This case is particularly challenging, as the buffer allocation heuristics lead to heterogeneous buffer sizes, which have not yet been proved optimal with our scheduling solutions for multiple processors.

## References

- [1] A. Allahverdi and H. Soroush. The significance of reducing setup times/setup costs. *European Journal of Operational Research*, 187(3):978 – 984, 2008. ISSN 0377-2217. doi: 10.1016/j.ejor.2006.09.010.
- [2] A. Allahverdi, C. Ng, T. Cheng, and M. Kovalyov. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, 187(3):985–1032, 2008.
- [3] A. Benoit and Y. Robert. Mapping pipeline skeletons onto heterogeneous platforms. *J. Parallel and Distributed Computing*, 68(6):790–808, 2008.
- [4] P. Bhat, C. Raghavendra, and V. Prasanna. Efficient collective communication in distributed heterogeneous systems. In *ICDCS'99 19th Int. Conf. on Distributed Computing Systems*, pages 15–24, 1999.
- [5] P. Bhat, C. Raghavendra, and V. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63:251–263, 2003.
- [6] A. Bryan and Norman. Scheduling flowshops with finite buffers and sequence-dependent setup times. *Computers & Industrial Engineering*, 36(1):163 – 177, 1999. ISSN 0360-8352. doi: 10.1016/S0360-8352(99)00007-8.
- [7] D. Gendreau, M. Gauthier, D. Hériban, and P. Lutz. Modular architecture of the microfactories for automatic micro-assembly. *Journal of Robotics and Computer Integrated Manufacturing*, 26(4):354–360, 2010.
- [8] L. Li and F. Qiao. Aco-based scheduling for a single batch processing machine in semiconductor manufacturing. In *IEEE Int. Conf. on Automation Science and Engineering (CASE)*, pages 85–90, 2008.
- [9] L. Li, F. Qiao, and Q. Wu. Aco-based scheduling of parallel batch processing machines to minimize the total weighted tardiness. In *IEEE Int. Conf. on Automation Science and Engineering (CASE)*, pages 280–285, 2009.
- [10] P. B. Luh, L. Gou, Y. Zhang, T. Nagahora, M. Tsuji, K. Yoneda, T. Hasegawa, Y. Kyoya, and T. Kano. Job shop scheduling with group-dependent setups, finite buffers, and long time horizon. *Annals of Operations Research*, 76:233–259, 1998. ISSN 0254-5330.
- [11] B. Srikar and S. Ghosh. A milp model for the n-job, m-stage flowshop with sequence dependent set-up times. *International Journal of Production Research*, 24(6):1459–1474, 1986.
- [12] J. Subhlok and G. Vondran. Optimal mapping of sequences of data parallel tasks. In *ACM SIGPLAN Notices*, volume 30(8), pages 134–143, 1995.
- [13] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, page 71. ACM, 1996.
- [14] M. Zhang and K. Goldberg. Calibration of wafer handling robots: A fixturing approach. In *IEEE Int. Conf. on Automation Science and Engineering (CASE)*, pages 255–260, 2007.





**RESEARCH CENTRE  
GRENOBLE – RHÔNE-ALPES**

Inovallée  
655 avenue de l'Europe Montbonnot  
38334 Saint Ismier Cedex

Publisher  
Inria  
Domaine de Voluceau - Rocquencourt  
BP 105 - 78153 Le Chesnay Cedex  
[inria.fr](http://inria.fr)

ISSN 0249-6399