

# Optimizing buffer sizes for pipeline workflow scheduling with setup times

Anne Benoit<sup>1</sup>, Jean-Marc Nicod<sup>2</sup>, and Veronika Rehn-Sonigo<sup>2</sup>

1. LIP, École Normale Supérieure de Lyon, CNRS & INRIA, France

2. FEMTO-ST Institute, UFC/ENSMM, Besançon, France

**Abstract**—Mapping linear workflow applications onto a set of homogeneous processors can be optimally solved in polynomial time for the throughput objective with fewer processors than stages. This result even holds true, when setup times occur in the execution and homogeneous buffers are available for the storage of intermediate results. In this kind of applications, several computation stages are interconnected as a linear application graph, and each stage holds a buffer of limited size where intermediate results are stored and a processor setup time occurs when passing from one stage to another. In this paper, we tackle the problem where the buffer sizes are not given beforehand and have to be fixed before the execution to maximize the throughput within each processor. The goal of this work is to minimize the cost induced by the setup times allocating buffers with proportional sizes of each other. We present a closed formula to compute the optimal buffer allocation in the case of non-decreasing setup costs in the linear application. For the case of unsorted setup times, we provide competitive heuristics that are validated via extensive simulation.

## I. INTRODUCTION AND RELATED WORK

Several real-life applications have a linear pipelined structure, where each data set must go through all application stages in a sequential manner. For instance, in image processing applications, a flow of images (data sets) enters the pipeline and must go through several stages such as filters, encoders, and so on [1], [2], [3]. Other examples of such applications are stream-processing applications composed of processing elements [4], or pipelined query operators with precedence constraints [5].

Such applications have been widely studied in the last years, and in particular, when a large-scale platform consisting of many processors is available, one difficulty is to decide how to efficiently decompose the application into intervals of stages, and then map each interval of stages onto a distinct processor, so that work can be parallelized and the throughput of the application can be improved. The throughput is defined as the number of data sets that can be processed per time unit. In the case of a homogeneous platform, Subhlock and Vondran [6], [7] propose an algorithm to find the optimal interval mapping. The problem becomes NP-complete as soon as communications or computations are heterogeneous [8].

However, in the previous studies, the cost of switching between stages of the application on one processor (if it was assigned more than one stage) is completely neglected: the first data set goes sequentially through all stages assigned to this processor, then the processor starts processing the second data set, and so on. Therefore, once a data set has been processed

for a given stage, the processor must get ready to execute a new stage (either on the same data set, or on the next data set). The cost of switching between stages is called a *setup time*, and these cannot be neglected in several applications covering many domains, see the survey [9]. For instance, 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 [10].

A traditional solution to reduce setup times consists in using buffers between each stage, so that several data sets can be processed consecutively on the same stage before moving to the next stage. It was already proved that if the setup time depends both on the previous stage and on the next stage, then the problem is NP-complete [11]. They are considering a different context of a flowshop application where each data set takes a different execution time on each machine, but they have a similar model with finite buffers and setup times. Other scheduling problems with the same model have been studied, as for instance in the context of the manufacturing of gas insulated switchgears [12].

In our previous work [13], we have introduced the inner-processor scheduling problem: consider a single processor that is in charge of a linear chain of stages, and a set of buffers that can hold in memory some data sets between two consecutive stages. We have shown that in the general case, it is difficult to decide in which order each data set and each stage should be executed, so that the throughput is maximized. Note that on a single processor, maximizing the throughput is done by minimizing the sum of all setup times, which are slowing down the whole application. However, if buffers are of proportional size, i.e., the two buffers before and after each stage are always multiple of each other in terms of number of data sets that they can handle, then we proposed some optimal inner-scheduling algorithms to decide in which order to execute stages (and data sets).

In this paper, we go one step further and we tackle the difficult problem of deciding how to allocate memory to buffers so that the throughput can be maximized. Indeed, with a fixed size memory, the naive approach is to split memory between buffers so that each buffer can hold the same number of data sets. However, this approach is unduly restrictive in the case of heterogeneous setup times: one would like to favor large buffers surrounding a costly stage so that less corresponding setup times are paid. One difficulty is that we will gain setup costs only if both of these surrounding buffers are large, hence using more memory. To keep the problem tractable, we focus on sequence-independent setup times, where the setup time

only depends on the next stage to which the processor will reconfigure.

The paper is organized as follows. We first detail the framework in Section II. Then we illustrate our reasoning through a simpler case study with only two different setup costs in Section III. The core of the study is in Section IV. We first prove how to find the optimal ratios between consecutive buffers, in the case of non-decreasing setup costs. Then we introduce several heuristics for the general case, building upon the ratios obtained before. One difficulty is to round the non-integer ratios into integer values without exceeding the total memory capacity (or leaving too much memory unused), and also to decide when it is worth having two large buffers surrounding a stage with a costly setup time. Extensive simulations are provided in Section V, demonstrating the efficiency of the heuristics that carefully choose the buffer sizes, compared to the naive solution with same-size buffers. Finally, we conclude and give future research directions in Section VI.

## II. FRAMEWORK

The application is a linear workflow application, or pipeline. 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  $S_{i+1}$ . Note that  $\delta_1$  and  $\delta_{n+1}$  are respectively the size of the input and output data of the application.

To switch from the execution from a stage  $S_i$  to a stage  $S_j$ , the processor has to be reconfigured for the next execution. This induces a setup time, denoted as  $st_i$  for stage  $S_i$  ( $1 \leq i \leq n$ ). Typically, the setup time only depends on the next stage  $S_i$  to which the processor will reconfigure.

If setup times can be neglected, the easiest way to proceed is to deal with the first data set, processing it through stages  $S_1$  to  $S_n$ , and then continue with the next data set. This implies a reconfiguration cost for each stage and each data set.

Rather, in order to avoid too many setup times, intermediate results can be stored in buffers. Therefore, each stage  $S_i$  ( $1 \leq i \leq n$ ) has an input buffer  $B_i$  that can store a number  $b_i$  of data sets. The output buffer for stage  $S_i$  is  $B_{i+1}$  (hence a total of  $n+1$  buffers). The number of consecutive computations of a same stage that can be done is bounded by the input buffer and output buffer capacities: the 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, and it is bounded by  $\min(b_i, b_{i+1})$  consecutive executions. The sizes of these buffers are limited by the total memory size  $M$ ; the memory constraint writes:

$$\sum_{i=1}^{n+1} \delta_i \times b_i \leq M. \quad (1)$$

It was shown in [13] that deciding in which order to execute stages is a difficult problem, even with constant setup times. If

all the data sizes (the  $\delta_i$ 's) are equal, then it seems natural to have same-size buffers, and to process, say,  $b$  data sets through the first stage, then the same data sets through the second stage, and so on, and then start over with the next  $b$  data sets. However, if buffers are of different sizes, it is extremely difficult to figure out the best way of scheduling the stage execution, unless the buffers are of proportional sizes, i.e., for  $1 \leq i \leq n$ , either  $b_i/b_{i+1}$  or  $b_{i+1}/b_i$  is integer. In this case, we can design a scheduling of stages such that the cost is:

$$C = \sum_{i=1}^n \frac{st_i}{\min(b_i, b_{i+1})}. \quad (2)$$

Indeed, each data set is going through all stages, but it may pay a setup cost only for some of the stages. In average, because stage  $S_i$  can be executed  $\min(b_i, b_{i+1})$  times without paying a setup cost, the cost incurred by a data set in stage  $S_i$  is  $\frac{st_i}{\min(b_i, b_{i+1})}$ , hence the result.

Note that minimizing the cost  $C$  is equivalent to maximizing the application throughput, defined as the inverse of the period  $\sum_{i=1}^n \frac{w_i}{s} + C$ , where  $s$  is the processor speed (hence it takes a time  $\frac{w_i}{s}$  to compute one data set for stage  $S_i$ ).

Finally, the goal is to minimize  $C$ , given the memory constraint stated in Equation (1).

We start with a simpler case study with only two different setup costs in Section III, before moving to the general problem in Section IV.

## III. WITH TWO DIFFERENT SETUP COSTS

We first consider that only one task has a different setup cost than the others, that is larger. For simplicity, we consider identical data sizes in this section, i.e.,  $\delta_i = 1$  for  $1 \leq i \leq n+1$ .

If all tasks have identical setup time  $st$  except one (say  $S_i$ ) with setup time  $ST > st$ , it is natural to have identical buffer sizes,  $b$ , except for the input and output buffers of  $S_i$ :  $b_i = b_{i+1} = B \geq b$ . Therefore, less setup cost must be paid for task  $S_i$ . The cost can then be expressed as:

$$C = \frac{st}{b} \times (n-1) + \frac{ST}{B}.$$

There are two buffers of size  $B$  and  $n-1$  buffers of size  $b$ , and therefore the memory constraint writes (remember that  $\delta_i = 1$ ):

$$M \geq (n-1)b + 2B. \quad (3)$$

It was shown in [13] that an efficient schedule can be found only if two consecutive buffers are multiples. Therefore, we assume that  $B = \alpha \times b$ , where  $\alpha$  is an integer (and  $\alpha \geq 1$ ). Note that  $\alpha \leq \left\lfloor \frac{M-(n-1)b}{2} \right\rfloor$ , the largest value being possibly achieved in the case  $b = 1$ .

Now we replace  $B$  by  $\alpha \times b$  in Equation (3) and we obtain:

$$b \leq \frac{M}{(n-1) + 2\alpha}.$$

The goal is to find the value of  $\alpha$ , and therefore the values of  $b$  and  $B$ , so that the cost  $C$  is minimized. We first consider that  $b$  can be rational, and then we will explain how to choose integer values. Therefore, we set  $b = \frac{M}{(n-1)+2\alpha}$ . The cost can then be expressed as a function of  $\alpha$ :

$$C(\alpha) = \frac{1}{M} \left( \frac{ST(n-1+2\alpha)}{\alpha} + st(n-1+2\alpha)(n-1) \right),$$

and the derivative is

$$C'(\alpha) = \frac{n-1}{M} \left( 2st - \frac{ST}{\alpha^2} \right).$$

The function  $C'(\alpha)$  is decreasing for  $1 \leq \alpha \leq \sqrt{\frac{ST}{2st}} = \alpha_{opt}$ , and increasing for  $\alpha \geq \alpha_{opt}$ . If  $\alpha_{opt} > \lfloor \frac{M-(n-1)}{2} \rfloor$ , then we let  $\alpha_{opt} = \lfloor \frac{M-(n-1)}{2} \rfloor$ .

Finally, we compute the optimal integer values of  $b$  and  $B$  for  $\alpha = \lfloor \alpha_{opt} \rfloor$  and  $\alpha = \lceil \alpha_{opt} \rceil$ , and we keep the choice of  $\alpha$  that minimizes the cost.

#### IV. OPTIMIZING THE BUFFER SIZES WITH DIFFERENT SETUP COSTS

We are now back to the general problem with  $n$  different setup costs. We first focus on the case where all setup costs are non-decreasing. Therefore, the buffers get larger and larger when we move towards the end of the pipeline. We aim at keeping buffer sizes that are multiples two by two, so that we can easily derive a scheduling algorithm that achieves the minimum period, without additional cost (see [13]).

##### A. All setup costs are non-decreasing

With non-decreasing setup costs, we always have  $\min(b_i, b_{i+1}) = b_i$ , and therefore the cost function of Equation (2) becomes

$$C = \sum_{i=1}^n \frac{st_i}{b_i},$$

with  $b_{i+1} \geq b_i$ . Because all  $b_i$ 's should be multiple, we set  $b_i = \prod_{k=1}^i \alpha_k$ , for  $1 \leq i \leq n+1$ . We therefore have  $n+1$  unknowns  $\alpha_k$ , for  $1 \leq k \leq n+1$ , and  $b_i = \alpha_i b_{i-1}$ , for  $1 \leq i \leq n+1$ , assuming that  $b_0 = 1$ .

For  $1 \leq i \leq n$ , we express  $\alpha_i$  as a function of the  $\alpha_k$ 's, with  $k > i$ . Then, we obtain an expression for  $\alpha_{n+1}$ , and we can recursively derive all values of  $\alpha_i$ . All these values that optimize the cost function  $C$  are rational, and we will round them to integer values in a later step.

For the ease of notations, let  $P_a^b = \prod_{\ell=a}^b \alpha_\ell$ , and  $P_a^b = 1$  for  $a > b$ .

First, we obtain  $\alpha_1$  thanks to the memory constraint:

$$\alpha_1 = \frac{M}{\delta_1 + \sum_{k=2}^{n+1} P_2^k \delta_k}. \quad (4)$$

Next, we express the cost as a function of  $\alpha_2$ , replacing  $\alpha_1$  by its optimal value:

$$\begin{aligned} C(\alpha_2) &= \sum_{k=1}^n \frac{st_k}{P_1^k} = \frac{1}{\alpha_1} \left( st_1 + \sum_{k=2}^n \frac{st_k}{P_2^k} \right) \\ &= \left( \frac{\delta_1}{M} + \frac{\alpha_2}{M} \sum_{k=2}^{n+1} P_3^k \delta_k \right) \left( st_1 + \frac{1}{\alpha_2} \sum_{k=2}^n \frac{st_k}{P_3^k} \right). \end{aligned}$$

This function is of the form  $a\alpha_2 + \frac{b}{\alpha_2} + c$ , and therefore the minimum is achieved for  $\alpha_2 = \sqrt{\frac{b}{a}}$ , which gives us:

$$\alpha_2 = \sqrt{\frac{\frac{\delta_1}{M} \sum_{k=2}^n \frac{st_k}{P_3^k}}{\frac{st_1}{M} \sum_{k=2}^{n+1} P_3^k \delta_k}} = \sqrt{\frac{\delta_1 \sum_{k=2}^n st_k P_{k+1}^n}{st_1 P_3^n \sum_{k=2}^{n+1} P_3^k \delta_k}}.$$

We prove recursively that

$$\alpha_i = \sqrt{\frac{\delta_{i-1} \sum_{k=i}^n st_k P_{k+1}^n}{st_{i-1} P_{i+1}^n \sum_{k=i}^{n+1} P_{i+1}^k \delta_k}} \quad (5)$$

for  $i \geq 2$ . Furthermore, we prove that minimizing the cost  $C(\alpha_i)$  for fixed values of  $\alpha_k$ ,  $i < k \leq n+1$ , is equivalent to minimizing the function

$$\Delta_i(\alpha_i) = \left( \frac{\delta_{i-1}}{M} + \frac{\alpha_i}{M} \sum_{k=i}^{n+1} P_{i+1}^k \delta_k \right) \left( st_{i-1} + \frac{1}{\alpha_i} \sum_{k=i}^n \frac{st_k}{P_{i+1}^k} \right).$$

We have already shown these results for  $i = 2$ , because we have exactly  $C(\alpha_2) = \Delta_2(\alpha_2)$ .

Let us assume that the results are true for all values smaller than or equal to  $i$ , and let us establish the result for  $\alpha_{i+1}$ . We express  $\Delta_i$  as a function of  $\Delta_{i+1}$ , assuming that the optimal value of  $\alpha_i$  is used in the expression of the cost. Let  $a = \frac{\alpha_i}{M} \sum_{k=i}^{n+1} P_{i+1}^k \delta_k$  and  $b = \sum_{k=i}^n \frac{st_k}{P_{i+1}^k}$ . We can rewrite  $\Delta_i$  as:

$$\Delta_i = \frac{st_{i-1} \delta_{i-1}}{M} + st_{i-1} a \alpha_i + \frac{\delta_{i-1}}{M \alpha_i} b + ab.$$

Because of the hypothesis,  $\alpha_i$  is minimizing  $\Delta_i$ , and because of the form of  $\Delta_i$ , we have  $\alpha_i = \sqrt{\frac{b}{a} \frac{\delta_{i-1}}{M st_{i-1}}}$ , and

$$\Delta_i = \frac{st_{i-1} \delta_{i-1}}{M} + 2 \sqrt{\frac{st_{i-1} \delta_{i-1}}{M}} ab + ab.$$

It is then easy to check that  $ab = \Delta_{i+1}$  by developing the product, and the only terms in  $\alpha_{i+1}$  appear in  $\Delta_{i+1}$ . Therefore, in order to minimize the cost expressed as a function of  $\alpha_{i+1}$ , we need to minimize  $\Delta_{i+1}(\alpha_{i+1})$ . This result is obtained directly, similarly to the way we obtained  $\alpha_2$ , thanks to the form of the function.

This holds for  $i \leq n$ , and we note that  $\alpha_{n+1} = 1$  (no gain can be achieved by having a larger last buffer). Therefore, we derive the value of

$$\alpha_n = \sqrt{\frac{\delta_{n-1} st_n}{st_{n-1} \delta_n + \delta_{n+1}}}. \quad (6)$$

We can therefore compute all optimal rational values of the  $\alpha_i$ 's. Interestingly,  $\alpha_i$  depends only of the  $\delta_k$ 's and of the  $st_k$ 's for  $k \geq i-1$ . Only  $\alpha_1$  accounts for  $M$ , because all other values are ratios.

We will discuss how to choose integer values in Section IV-B. Indeed, the integer value  $\alpha_i$  can be either  $\lceil \alpha_i \rceil$  or  $\lfloor \alpha_i \rfloor$ . For all  $1 \leq i \leq n+1$ , each choice for  $\alpha_i$  influences the value of the cost. Consequently, we have potentially to consider  $2^{n+1}$  different configurations. Moreover, the limited memory size  $M$  is an additional constraint to these choices. It is why we choose the integer value of each  $\alpha_i$  using three heuristics (Up, Down and Closest) detailed in the next section.

### B. General case

In the general case, it is no longer possible to foresee if the value of  $\min(b_i, b_{i+1})$  is  $b_i$  or  $b_{i+1}$  when computing the overhead  $C$  given by Equation (2). However, intuitively, if stage  $S_i$  has a larger setup time than  $S_{i-1}$  and  $S_{i+1}$  (for  $1 < i < n$ ), the capacity of its input and output buffers should be larger than or equal to the capacity of the input buffer of  $S_{i-1}$  and of the output buffer of  $S_{i+1}$ , and both of these buffers should have the same size, so that we can run stage  $S_i$  exactly  $b_i = b_{i+1}$  times before paying a setup time.

Our goal is to reuse the theoretical results derived in Section IV-A. One possibility would have been to compute the optimal values of the ratios between buffers for each sequence of stages whose setup costs are monotonic, but then it is very difficult to decide how to share memory between each of these sequences of stages. Rather, we decide to sort all setup costs and compute the ratios as in Section IV-A, and then we heuristically decide how to choose integer values of buffer size capacities, while not exceeding the total memory capacity.

We now describe seven heuristics to compute buffer capacities. The first one, described in Section IV-B1, uses identical buffer capacities. There are obviously some cases where this naive approach will be optimal, but we then introduce heuristics that target applications with heterogeneous setup times, for which it may be better to derive buffer capacities of different sizes. There are two categories of heuristics (Sections IV-B2 and IV-B3), with three variants per heuristic depending on the rounding strategy used to obtain integer values.

1) *SameB*: The first heuristic SameB is the naive approach that we mentioned in Section I that consists in sharing the memory  $M$  into  $n+1$  buffers with identical capacity, i.e., they can hold the same number of data sets. For  $1 \leq i \leq n+1$ , buffer  $B_i$  contains data of size  $\delta_i$ , and for a capacity  $b$ , it will therefore use an amount of memory  $b \times \delta_i$ . Therefore, the buffer capacity  $b$  for the SameB heuristic is defined as follows:

$$b = \left\lfloor \frac{M}{\sum_{i=1}^{n+1} \delta_i} \right\rfloor. \quad (7)$$

As said before, since the  $b_i$ 's are proportional two by two, a solution with different buffer capacities imposes that we have at least  $b_i = 2 \times b_j$  for  $1 \leq i, j \leq n+1$ . Therefore, because the memory size is limited and has to be shared between all buffers, choosing the same buffer capacities could be the best compromise.

2) *H1-Up, H1-Down and H1-Closest*: We propose a set of heuristics, building upon the theoretical results of Section IV-A, as explained above. For stages with a local maximum setup times, we plan to adapt the buffer size as shown in Figures 1, 2 and 3, depending on the available memory space. For simplification, we assume that the last buffer is of identical size as the input buffer for stage  $S_n$ , i.e.,  $b_n = b_{n+1}$ . The implemented heuristics are described as follows:

- 1) We first sort the setup values into a non-decreasing order, using a permutation function  $\pi$  such that  $st_{\pi(i)} \leq st_{\pi(j)}$  if  $\pi(i) < \pi(j)$ , for  $1 \leq i, j \leq n$ . We then compute the sequence of the  $\alpha_k$ 's backwards according to Equations (6), (5), (4), so that  $b_{\pi(i)} = \prod_{k=1}^{\pi(i)} \alpha_k$ . But, because the stage  $S_n$  has not necessarily the biggest setup time ( $st_n \leq \max_{1 \leq i \leq n} (st_i)$ ), we have to foresee a room for the buffer  $b_{n+1}$  in order to anticipate the memory usage to be able to end the computation of the  $\alpha_k$ 's. As mentioned before,  $\alpha_{n+1} = 1$  allows us to obtain  $b_n = b_{n+1}$  in the non-decreasing case described in the previous section. Now we have to know the position of  $st_n$  within the sorted sequence of setups ( $\pi(n)$ ) to be able to insert  $\alpha_{\pi(n)+1} = 1$  into the sequence by shifting the end of the sequence. This can also be done by artificially adding an extra stage  $S_{n+1}$  with  $st_{n+1} = st_n$  before the sorting process. Then we obtain the sequence of  $\alpha_k$ 's,  $1 \leq k \leq n+1$ .
- 2) We define three rounding policies (resp. Up, Down and Closest) to obtain an integer value for each  $\alpha_k$  so as to compute each value of  $b_k = \prod_{\ell=1}^k \alpha_\ell$  for  $1 \leq k \leq n+1$  since  $b_k \in \mathbb{N}^*$ . We make our best to increase  $\alpha_k$  to  $\lceil \alpha_k \rceil$  to increase the buffer capacity of  $B_{\pi^{-1}(k)}$  and to reduce as far as possible the total setup cost. However, each  $\alpha_k$  is an optimal theoretical rational value that minimizes the cost  $C$ , and increasing its value is not always possible because of the limited memory  $M$  available on the machine. Let  $\widehat{\alpha}_k$  be the integer value chosen for  $\alpha_k$ . H1-Up, H1-Down and H1-Closest are three different heuristics corresponding to three respective ways to choose an integer value for  $\alpha_k$ :
  - a) H1-Up: we consider  $\alpha_k$  from  $k = 1$  to  $n+1$  and for each value of  $k$ , we set  $\widehat{\alpha}_k = \lceil \alpha_k \rceil$  if the memory use is less than or equal to the memory size  $M$ , and  $\widehat{\alpha}_k = \lfloor \alpha_k \rfloor$  otherwise. Note that when we set an  $\alpha_k$  to the lower integer part, the size of the available memory is increasing, hence leading more room for further upgrades of the remaining  $\alpha_k$ 's.
  - b) H1-Down: we proceed as before, but we consider the  $\alpha_k$ 's in a decreasing order, from  $n+1$  to 1.
  - c) H1-Closest: with this last policy, we do not try to force each time the value of  $\alpha_k$  to  $\lceil \alpha_k \rceil$  but to  $\lfloor \alpha_k \rfloor$  if  $\alpha_k - \lfloor \alpha_k \rfloor < 0.5$  and to  $\lceil \alpha_k \rceil$  otherwise. The memory size is also taken into account to validate or not the choice of  $\lceil \alpha_k \rceil$  for  $\widehat{\alpha}_k$ , considering that the choice of  $\lfloor \alpha_k \rfloor$  is always possible.
- 3) We compute each  $b_k$  with  $1 \leq k \leq n+1$  using the integer values  $\widehat{\alpha}_\ell$  of  $\alpha_\ell$  with  $1 \leq \ell \leq k$  and  $1 \leq k \leq n+1$ ;
- 4) We replace each value of  $b_k$  to its original position to become the size of  $B_{\pi^{-1}(k)}$  within the pipeline using the function  $\pi^{-1}(k)$  for each  $k$ ,  $1 \leq k \leq n+1$ .
- 5) We then consider each stage  $S_i$  that is a local maximum

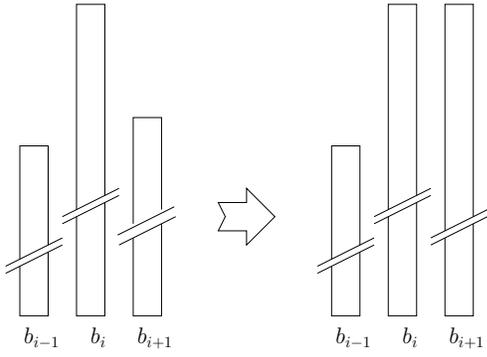


Figure 1. Adapting the buffer size to  $b_i$  when the available memory is large enough.

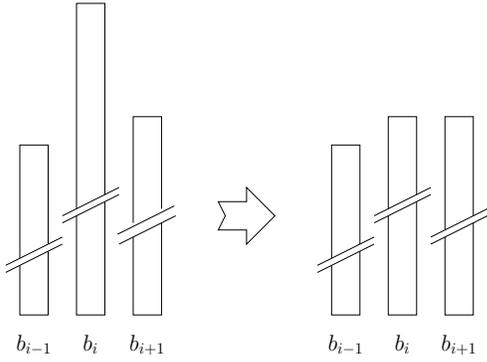


Figure 2. Adapting the buffer size to  $b_{i+1}$  when the available memory is not large enough.

in terms of setup costs, i.e.,  $st_i = \max(st_{i-1}, st_i, st_{i+1})$  (for  $1 < i < n$ ). As we said before,  $b_i$  should be equal to  $b_{i+1}$  even if we plan to take advantage of the whole  $b_i$  inputs of  $S_i$ , in order to reduce the overhead caused by the  $st_i$  setup cost. Therefore, we want ideally to increase the allocated memory of the output buffer of  $S_i$ , as shown in Figure 1. As  $b_k = \prod_{\ell=1}^k \widehat{\alpha}_\ell$ , if  $b_{i+1} < b_i$  then there is at least a factor of 2 between  $b_i$  and  $b_{i+1}$ . This re-allocation phase is a very memory consuming process. In some cases, increasing the capacity of output buffer  $B_{i+1}$  of the stage  $S_i$  from  $b_{i+1}$  to  $b_i$  may not be possible. When the available memory is not large enough to increase the value of  $b_{i+1}$ , we re-allocate the memory as follows. We reduce the value of  $b_i$  to  $\max(b_{i-1}, b_{i+1})$  as shown in Figure 2 when  $b_{i+1} = \max(b_{i-1}, b_{i+1})$  and in Figure 3 when  $b_{i+1} = \max(b_{i-1}, b_{i+1})$ . In this way, the available memory size is increasing step by step and may make one or more re-allocations as presented in Figure 1 possible.

The three heuristics H1-Up, H1-Down and H1-Closest are compared together in Section V that shows simulations onto numerous scenarios.

3) *H2-Up, H2-Down and H2-Closest*: Because of the limited memory space, giving a larger number of inputs as far as possible to stages that are bigger than the others could appear as a good deal to reduce the value of  $C$  compared to sharing this extra memory space to each buffer  $b_i$  with  $1 \leq i \leq n+1$ . We introduce three new heuristics H2-Up, H2-Down and H2-Closest that aim at changing the number of inputs of each  $S_{i+1}$

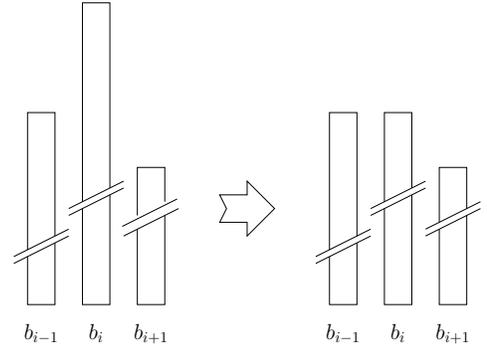


Figure 3. Adapting the buffer size to  $b_{i-1}$  when the available memory is not large enough.

when  $st_i = \max(st_{i-1}, st_i, st_{i+1})$  for all  $1 < i < n$ , even if the memory space is not large enough. H2-Up, H2-Down and H2-Closest are designed as follows:

- 1) We follow the first four steps for each of the three heuristics H1-Up, H1-Down and H1-Closest to begin respectively the three new heuristics H2-Up, H2-Down and H2-Closest. For instance, each  $b_i$  computed using H2-Up has the same value as with H1-Up after the step (4) of H1-Up description (before dealing with local maximum setup costs).
- 2) For each stage  $S_i$  with  $st_i = \max(st_{i-1}, st_i, st_{i+1})$  ( $1 < i < n$ ), we propose to force  $b_{i+1}$  to take the value of  $b_i$  to decrease as far as possible the impact of the large values of  $st_i$  in the expression of  $C$  (see equation (2)), even if there is not enough available memory. Figure 1 illustrates how the re-allocation phase works. As explained in the previous section in the step (5) of H1-\*, this re-allocation phase is a very memory consuming process. To make this operation possible, we repair the memory loss by changing the proportionality between each buffer as explained in the next step.
- 3) We re-compute the value of  $\widehat{\alpha}_1$  as follows: as  $\alpha_1$  is designed to take advantage of the memory considering the already computed values of each  $\alpha_k$  with  $1 < k \leq n+1$ , using Equation (4) but by using only integer numbers, it is possible to re-evaluate  $\alpha_1$  considering the same formula by changing  $P_2^k$  to  $b_k / \widehat{\alpha}_1$  since  $b_k = \prod_{\ell=2}^k \alpha_\ell \times \alpha_1$ . The values of the  $b_k$  are considered in the order given by the  $\pi$  function. Let  $\alpha'_1$  be the new rational value of  $\alpha_1$ :

$$\alpha'_1 = \frac{M}{\delta_1 + \sum_{k=2}^{n+1} \frac{b_k}{\alpha_1} \delta_k}. \quad (8)$$

- 4) We round down  $\alpha'_1$  to give an integer value to  $\widehat{\alpha}'_1$ . Because of the limited memory space, the only possible value for  $\widehat{\alpha}'_1$  is  $\lfloor \alpha'_1 \rfloor$ . We are then able to compute each  $b'_k$  value and we replace each value of  $b_i$  by  $b'_k$  where  $\pi(i) = k$  or  $\pi^{-1}(k) = i$ . Let  $b'_i = b'_{\pi^{-1}(i)}$  be the new number of input data sets of the buffer  $B_i$ . The relation between  $b'_i$  and  $b'_{i+1}$  is maintained in spite of the computation of  $\widehat{\alpha}'_1$ . The re-allocation process guarantees that the available memory size is greater than zero.

The previous description leads to define H2-Up, H2-Down and H2-Closest that will also be compared with the previous

heuristics.

Simulations in the next section aim at highlighting that it can be very interesting to find the appropriate buffer capacity, so that the impact of the setup costs on the performance of the system can be significantly decreased.

## V. SIMULATION RESULTS

In order to evaluate the different heuristics, we conduct several simulations. Our major goal is to demonstrate the importance of an intelligent buffer allocation and its influence on the setup cost. All task sizes are set to  $\delta_i = 1$ , hence allowing a better comparison of the different solutions. Therefore, the memory size is expressed in terms of total buffer capacities, and we consider in all experiments that  $M$  varies between 4000 and 14000.

In all the simulations, we use two mechanisms to create the values of the setup costs.

- **Rand1** – In the first approach, the setup costs are randomly drawn between 5 and 100001 with a step size fixed in the experiment. This means for a step size of  $x$ , the random variable can take its value among the values  $5 + k \times x < 100001$ . This leads to setups of different costs, but which may vary only slightly. The setup costs may even zigzag with alternate small and high values.
- **Rand2** – The second approach aims at generating setup costs that differ highly and secondly promise to achieve more of a wave shape than a zigzag. For this purpose, the setup costs belong to one of  $x$  setup types, fixed for each simulation. Each setup type  $t_i$  is drawn randomly a value  $x_i$  between 1 and 9, with the constraint that type  $t_i = x_i \times 10^{i-1}$ , for  $1 \leq i \leq x$ . Then, for each setup type, we randomly fix the number of stages that belong to each setup type, with the constraint that  $\#(t_i) \geq \#(t_{i+1})$ .

### A. Non-decreasing setup costs

In the first experiment, we evaluate the behavior of the heuristics for applications with non-decreasing setup costs. In this case, we can compute the optimal rational buffer sizes as described in Section IV-A. The final buffer sizes then solely depend on the rounding technique used to convert the rational solution into an integer solution. Note that in this application type, both the H1 and H2 versions of Up, Down and Closest behave exactly the same way as no buffer adaption has to be done. Figure 4 shows the results for applications with 70 stages and memory varying from 4000 to 14000: Figure 4(a) contains the results obtained with Rand1 (step size 122), whereas Figure 4(c) shows the results of the simulations with Rand2. Figure 4(b) shows the details of Figure 4(a). All graphs plot the mean values of the cost of 100 different applications. As can be seen, the heuristic SameB is outperformed in both cases. Still, in the case of platforms with low variance of setup costs (Rand1, Figure 4(a)), SameB performs almost as good as the best heuristics of H1 and H2, namely H1-Up, H2-Up, H1-Closest, and H2-Closest. Both H1-Down and H2-Down perform poorly. Here we can state the importance of the rounding policy: the Down heuristics tend to increase the

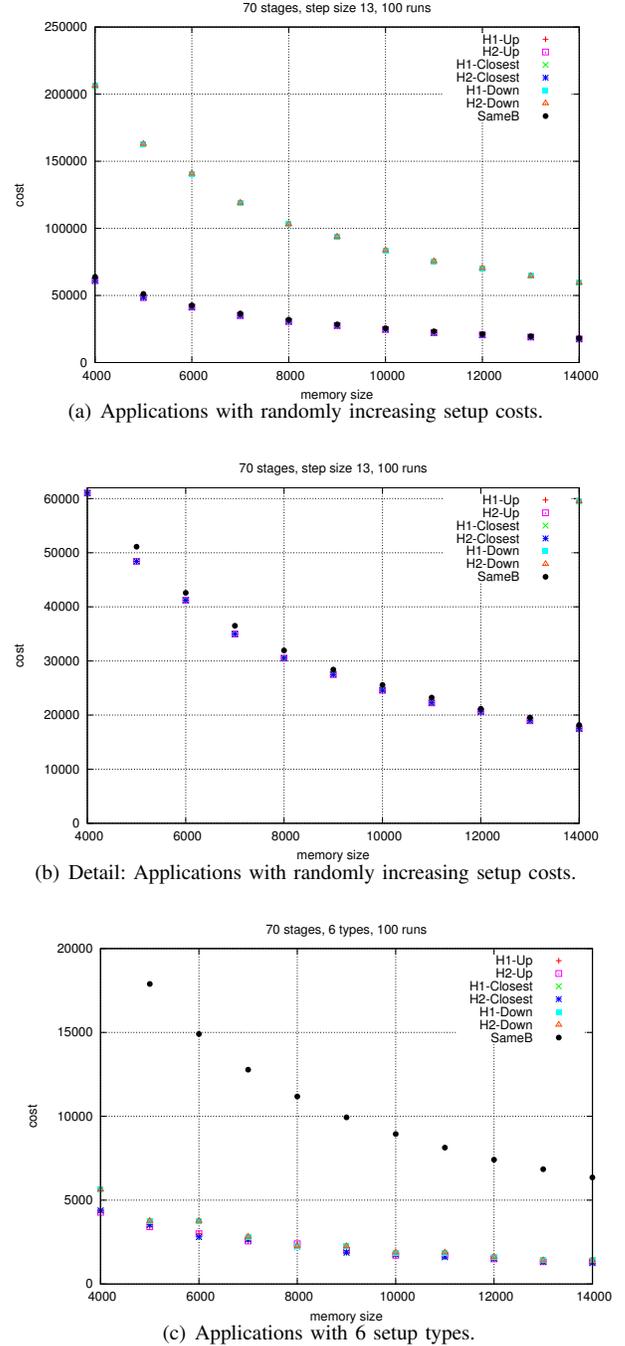


Figure 4. Mean cost for applications with 70 stages and non-decreasing setup costs.

$\alpha$ -factor of big buffers, whereas the other strategies tend to increase globally the buffer sizes, which leads to smaller buffer size differences. Hence the strategies Closest and Up allow us to better reflect the setup behavior. In the case of platforms with very heterogeneous setup costs (Figure 4(c)), all H1 and H2 heuristics outperform SameB. We achieve results up to 5.5 times better than SameB (H2-Closest for  $M = 6000$ ). The Down heuristics are still slightly outperformed by Closest and Up. The defeat of SameB is due to additional setups that have to be performed for stages with high setup costs, as SameB allocates less for these expensive stages than the

other strategies.

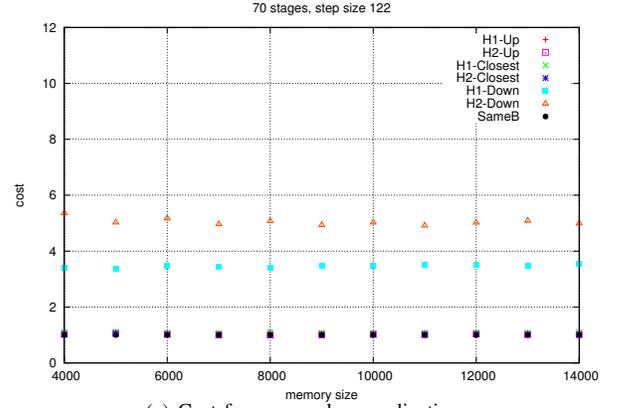
### B. General case: setup costs in wave shape

The second series of simulations tests the behavior of our heuristics on general platforms.

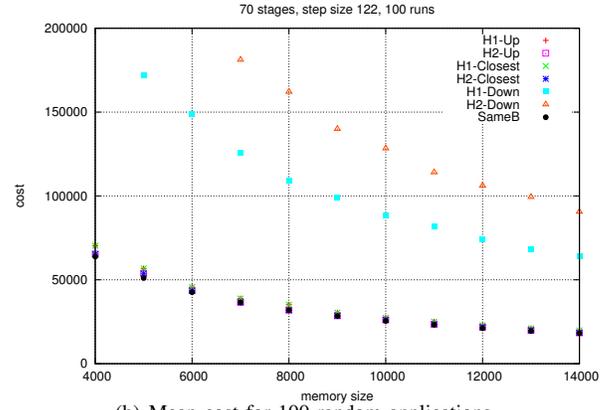
Rand1 creates general applications with the following properties: first, setup costs are heterogeneous, but often within the same order of magnitude; second, as already mentioned, setup costs tend to zigzag, which imposes to allocate larger buffers to stages with small setup costs in order to be able to benefit of the larger buffer as output buffer for a stage with large setup cost. Results for such applications are shown in Figure 5. The behavior of the heuristics is similar to the case of non-decreasing applications, with one important difference: SameB slightly outperforms the other heuristics and we find the previous ranking within the rounding heuristics: H2-Closest and H2-Up are better than H1-Up and H1-Closest. H1-Down is remarkably worse than the others and H2-Down performs the worst. Figure 5(a) shows the result of one application with 70 stages and step size 122. The cost of each heuristic is normalized by the cost of SameB. Hence SameB has a ratio of 1. When the memory size is fixed to 8000, H2-Up and H2-Closest also achieve 1, and H1-Up and H1-Closest obtain a ratio of 1.1. H1-Down achieves 3.4 and H2-Down finally has a ratio of 5.1. The good performance of SameB is due to the characteristics of Rand1 generated applications and its better performance in comparison to applications with non-decreasing setup costs can be explained as follows. The zigzag of setup values asks for same buffer sizes, because we need at least two consecutive large buffers to save setup costs, hence the good performance of SameB. For non-decreasing applications, it can be necessary to increase the buffer sizes at the end of the application to save some setups for high setup costs, hence SameB is outperformed. Figure 5(b) shows the mean cost of 100 applications with 70 stages. We observe that the previous ranking holds true and with increasing memory size, the total setup cost decreases as more buffer slots can be allocated for each stage. The relative performance nevertheless stays the same.

We then test the behavior of the heuristics on applications generated with Rand2. These applications have the following properties: first, successive buffers are either the same or they differ at least of one order of magnitude; second, short applications have at least one peak, whereas large applications have several peaks. We hence expect our heuristics to better cope with the peaks than SameB.

Figure 6 shows the results of an application with 200 stages and memory varying from 4000 to 20000 and we plot the results for 4, 5 and 6 setup types. The results of all heuristics are normalized with the result of SameB. A first observation reveals that the entire series of H1 heuristics performs awfully bad. Indeed, their ratio in general exceeds 2 (3 and 4) for 4 (5 and 6) setup types, respectively (Figures 6(a), 6(b) and 6(c)). This means that H1-Up, H1-Closest and H1-Down are at least two times worse than SameB. As already explained in Section IV, to be able to benefit from a large buffer, we need at least two of them to be able to gain in setups as we can perform stage  $S_i$   $\min(b_i, b_{i+1})$  times without paying a setup cost. As the computation of the  $\alpha$ -values does not



(a) Cost for one random application.



(b) Mean cost for 100 random applications.

Figure 5. Platform with 70 stages, setup costs are randomly generated.

anticipate this fact, the H1 heuristics adapt the buffer sizes in consequence (Section IV-B2 step 5). As sometimes there will not be enough remaining available memory, the big buffers have to be reduced and H1 loses its advantage over SameB.

A further observation then shows that SameB is outperformed by all H2 heuristics. For example, in the case of 4 setup types and  $M = 8000$  (see Figure 6(a)), H2-Up achieves a ratio of 0.69, H2-Down a ratio of 0.79 and H2-Closest achieves 0.86. This ranking holds true for the case of 4 setup types: H2-Up outperforms H2-Down, which outperforms H2-Closest, which outperforms SameB. When the number of setup types increases, the ratio between H2 and SameB further increases. For instance, in the case of 5 setup types (Figure 6(b)), the H2 heuristics are almost twice as efficient as SameB. In the case of 6 setup types (Figure 6(c)), we make the following observation: when there is only a small memory size ( $M \leq 5000$  for setup costs up to 900000), then not all of the H2 heuristics are able to provide a solution. Indeed, H2-Closest and H2-Down lack in finding a valid buffer allocation. More precisely, they are not capable to adjust the buffers within the remaining available memory, once the  $\alpha_i$ -values ( $1 < i \leq n + 1$ ) are fixed. They provide an  $\alpha_1 < 1$  and hence there is not sufficient memory left to round to 1. When the initial memory however is reasonably high, they once again provide competitive solutions, see Figure 6(d) for a zoom on H2 and SameB. H2-Up achieves results more than three times better than SameB (0.31 for  $M = 15000$  and  $M = 18000$ ). However, there is no ranking within the H2

heuristics observable. Each strategy outperforms the others on different memory sizes. The poorest ratio is given by H2-Down for  $M = 12000$  with a value of 0.59, which means that its performance is still 1.69 times better than SameB.

To validate our observations on one application with 200 stages, we plot in Figure 7 the mean result over 100 applications for H2 and SameB. The cost  $C$  decreases with increasing memory and the H2 heuristics achieve results that are in average 2.7 times better than the results found with SameB.

### C. Summary

To summarize our results, we were able to show the importance of the different rounding techniques on the final result. Also, the naive heuristic SameB, where memory is distributed equally under the stages is efficient but in general cases a more intelligent way to distribute the memory leads to considerably better solutions.

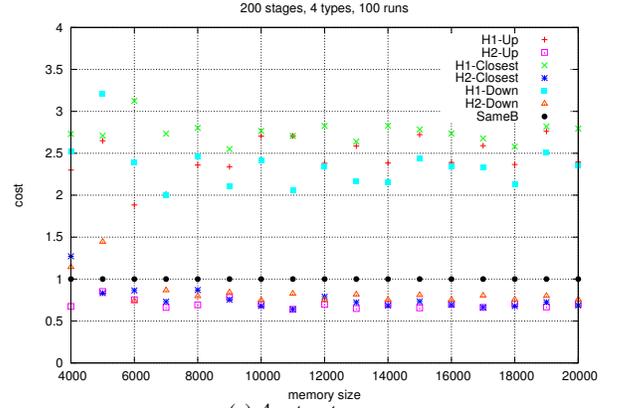
We can state that in applications with little variance in setup sizes or zigzag of setup sizes from one stage to another (Rand1), the simple SameB heuristic achieves comparable results to the Up and Closest (resp. H1 and H2) heuristics: For non-decreasing setup cost applications, SameB is slightly outperformed and in the general case, SameB is almost always better than H1 and H2, but the gap is negligibly small. Both Down heuristics however are to be avoided and are not competitive.

When applications provide at least one peak or have very diverging setup costs (Rand2), the simple SameB approach fails completely in performance. For non-decreasing setup cost applications, all H1 and H2 heuristics achieve at least 2 times better costs than SameB. For general applications, we state that H1 performs very poorly, but the H2 versions perform better than SameB.

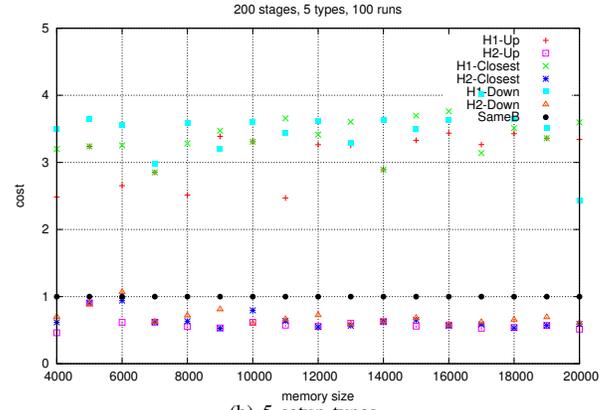
To conclude, we propose to use H2 when peaks of large setup costs occur in the application, then test the three H2 versions and take the best.

## VI. CONCLUSION

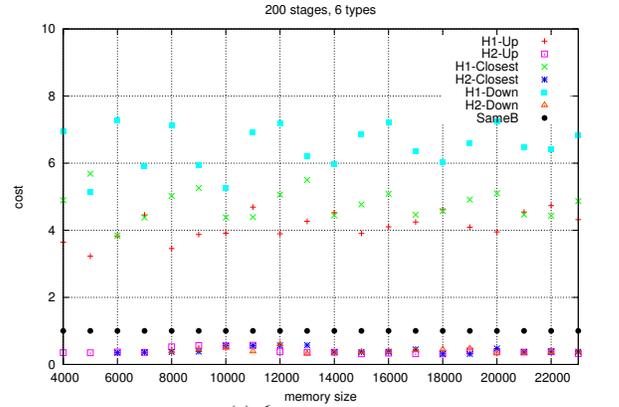
In this paper, we investigate the problem of allocating memory to buffers of a linear pipelined multi stage application in order to maximize the throughput when a setup cost has to be paid to switch from one stage to another. A naive approach consists in allocating the same buffer capacity to all stages, which allows each stage to save the same number of setups. We demonstrate in our work that this straightforward approach is only efficient in cases of applications with homogeneous setup times or in particular cases of small fluctuations or zigzags in setup times. In the case where setup times are heterogeneous, this strategy is unduly restrictive. We provide a theoretical study of applications where only one setup time differs from the others and extend this result to applications with non-decreasing setup costs. We give an optimal rational optimal solution of the problem in the latter case and provide a solution to obtain an efficient integer solution. For general applications, we adapt our solution and propose seven heuristics, the naive solution and six more complex ones based on the theoretical study. Numerous simulations based on exhaustive scenarios



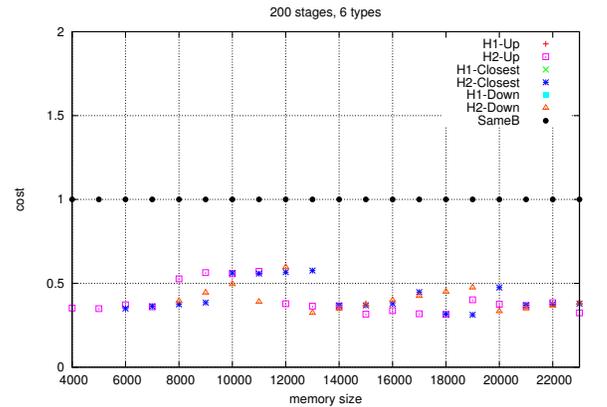
(a) 4 setup types.



(b) 5 setup types.



(c) 6 setup types.



(d) Details for 6 setup types.

Figure 6. Platform with 200 stages, increasing memory and different number of setup types.

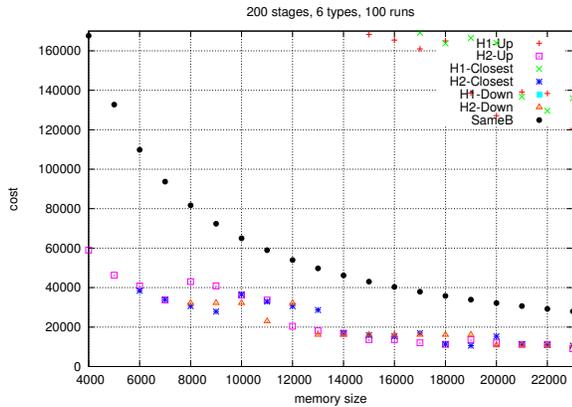


Figure 7. Mean results over 100 applications with 200 stages and 6 setup types, zoom on H2 and SameB.

show the relevance of our approach. The results show that our heuristics outperform the naive solution in all cases. Indeed, we achieve, depending on the application type, solutions that are up to 3.3 times better than the naive approach.

In future work, we plan to consider applications where setup times obey to wave shaped costs. In this case, from one subsequence of stages with monotonic setup costs to another, the proportionality of the buffer sizes is no longer mandatory even if both the input and the output buffer sizes of the stage whose setup is placed onto a peak have to be equal. This novel approach promises to be interesting from a theoretical point of view.

*Acknowledgements:* This work was supported in part by the ANR *RESCUE* project and by the Labex *ACTION* project (contract “ANR-11-LABX-01-01”). A. Benoit is with the Institut Universitaire de France.

## REFERENCES

- [1] R. Ramanath, W. Snyder, Y. Yoo, and M. Drew, “Color image processing pipeline,” *Signal Processing Magazine, IEEE*, vol. 22, no. 1, pp. 34–43, Jan 2005.
- [2] F. Guirado, A. Ripoll, C. Roig, A. Hernández, and E. Luque, “Exploiting throughput for pipeline execution in streaming image processing applications,” in *Proceedings of the 12th international conference on Parallel Processing*, ser. Euro-Par’06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 1095–1105. [Online]. Available: [http://dx.doi.org/10.1007/11823285\\_115](http://dx.doi.org/10.1007/11823285_115)
- [3] T. D. R. Hartley, A. Fasih, C. A. Berdanier, F. Özgüner, and Ü. V. Çatalyürek, “Investigating the use of gpu-accelerated nodes for sar image formation,” in *CLUSTER*, 2009, pp. 1–8.
- [4] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, “Elastic scaling of data parallel operators in stream processing,” in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12. [Online]. Available: <http://dx.doi.org/10.1109/IPDPS.2009.5161036>
- [5] J. Burge, K. Munagala, and U. Srivastava, “Ordering pipelined query operators with precedence constraints,” Stanford InfoLab, Technical Report 2005-40, 2005. [Online]. Available: <http://ilpubs.stanford.edu:8090/705/>
- [6] J. Subhlok and G. Vondran, “Optimal mapping of sequences of data parallel tasks,” in *ACM SIGPLAN Notices*, vol. 30(8), 1995, pp. 134–143.
- [7] —, “Optimal latency-throughput tradeoffs for data parallel pipelines,” in *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1996, p. 71.

- [8] A. Benoit and Y. Robert, “Mapping pipeline skeletons onto heterogeneous platforms,” *J. Parallel and Distributed Computing*, vol. 68, no. 6, pp. 790–808, 2008.
- [9] A. Allahverdi, C. Ng, T. Cheng, and M. Kovalyov, “A survey of scheduling problems with setup times or costs,” *European Journal of Operational Research*, vol. 187, no. 3, pp. 985–1032, 2008.
- [10] A. Allahverdi and H. Soroush, “The significance of reducing setup times/setup costs,” *European Journal of Operational Research*, vol. 187, no. 3, pp. 978 – 984, 2008.
- [11] A. Bryan and Norman, “Scheduling flowshops with finite buffers and sequence-dependent setup times,” *Comp. & Indus. Engineering*, vol. 36, no. 1, pp. 163 – 177, 1999.
- [12] 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*, vol. 76, pp. 233–259, 1998.
- [13] A. Benoit, M. Coqblin, J.-M. Nicod, L. Philippe, and V. Rehn-Sonigo, “Throughput optimization for pipeline workflow scheduling with setup times,” in *Proceedings of CGWS 2012, the CoreGRID/ERCIM Workshop on Grids, Clouds and P2P Computing, in conjunction with EuroPar 2012*, Aug. 2012. [Online]. Available: <http://graal.ens-lyon.fr/~abenoit/papers/RR-7886.pdf>