Optimizing memory allocation for multi stage scheduling including setup times

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

Journal of scheduling manuscript No.
(will be inserted by the editor)

Optimizing memory allocation for multi stage scheduling including setup times

Anne Benoit · Mathias Coqblin · Jean-Marc Nicod · Veronika Rehn-Sonigo

the date of receipt and acceptance should be inserted later

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 by 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. Three non scalable brute force algorithms are also provided to compare heuristic approaches to optimal ones for small applications and to evaluate the relevance of our approach.

A preliminary version of this work as been published in APDCM 2014.

Anne Benoit
LIP, École Normale Supérieure de Lyon, CNRS & INRIA, France E-mail: anne.benoit@ens-lyon.fr

Mathias Coqblin and Jean-Marc Nicod · Veronika Rehn-Sonigo
FEMTO-ST Institute, UFC/CNRS/ENSMM/UTBM, Besançon, France E-mail: firstname.surname@femto-st.fr

Keywords Linear workflow · mapping · setup times · buffers · cost minimization

1 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 as proposed by Ramanath et al (2005), Guirado et al (2006) and Hartley et al (2009). Schneider et al (2009) show other examples of such applications are stream-processing applications composed of processing elements, or pipelined query operators with precedence constraints as presented in Burge et al (2005).

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, Subhlok and Vondran (1995, 1996) propose an algorithm to find the optimal interval mapping. Moreover Benoit and Robert (2008) prove that the problem becomes NP-complete as soon as communications or computations are heterogeneous.

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 written by Allahverdi et al (2008). 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 as demonstrated by Allahverdi and Soroush (2008).

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 as proved by Bryan and Norman (1999). 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. Luh et al (1998) have presented other scheduling problems with the same model have been studied, as for instance in the context of the manufacturing of gas insulated switchgears.

In our previous work, Benoit et al (2012) 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 multiples 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 memory size, 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 2. Then we illustrate our reasoning through a simpler case study with only two different setup costs in Section 3. The core of the study is in Section 4. 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. In addition to these heuristics and to help the analyze of their significance, we provide at the end of this section optimal brute force algorithms which are only dedicated to small pipeline sizes because of their exponential complexity. They concern either the computation of two optimal rounding steps based on non-integer ratios or the computation of an *ad hoc* valid solution that minimizes the global setup cost overhead. Extensive simulations are provided in Section 5, 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 6.

2 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: $S = \{S_1, \ldots, S_n\}$. 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 δ_i from the previous stage, treats the data set, which takes a number of w_i computations, and outputs data of size δ_{i+1}. The output data of stage S_i is the input data of the next stage S_{i+1}. Note that δ_1 and δ_{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 ($1 \leqslant i \leqslant n$). 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 ($1 \leqslant i \leqslant n$) 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:

$$\sum_{i=1}^{n+1} \delta_i \times b_i \leqslant M \tag{1}$$

It was shown in Benoit et al (2012) that deciding in which order to execute stages is a difficult problem, even with constant setup times. If all the data sizes (the δ_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 $1 \leqslant i \leqslant n$, 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:

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

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 3, before moving to the general problem in Section 4.

3 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 $1 \leqslant i \leqslant n+1$.

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: $b_i = b_{i+1} = B \geqslant b$. 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 $\delta_i = 1$):

$$M \geqslant (n-1)b + 2B \tag{3}$$

It was shown in Benoit et al (2012) that an efficient schedule can be found only if two consecutive buffers are multiples. Therefore, we assume that $B = \alpha \times b$, where α is an integer (and $\alpha \geqslant 1$). Note that $\alpha \leqslant \left\lfloor \frac{M-(n-1)}{2} \right\rfloor$, 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:

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

The goal is to find the value of α, 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 α:

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

and the derivative is

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

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

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 α that minimizes the cost.

3.1 With several consecutive setup costs

The problem statement is the following with $st << ST$:

- n tasks
- $n+1$ buffers
- M the total memory space which have to be reserved for the $n+1$ buffers to receive the inputs/outputs of each task such that the output buffer of one task is the input buffer of the next one.
- x tasks whose setup is small (st) and whose input and output buffer is small too (b)
- $n-x$ tasks whose setup is big (ST) and whose input and output buffer is large (B)
- y sets (with at least one element) of consecutive tasks whose setup is big

In these conditions, we can deduce that we have:

- $n-x+y$ big buffers
- $x-y+1$ small buffers ($y \leqslant x+1$)

We proved that we obtain the best throughput when two consecutive buffers have their size such that one is a multiple of the other-one. So we have with α an integer:

$$B = \alpha \times b \tag{4}$$

The total memory space constraints the size of the memory use:

$$\begin{aligned} M &\geq (x-y+1)b + (n-x+y)B \\ &\geq (x-y+1)b + \alpha(n-x+y)B \\ &\geq b(x-y+1+\alpha(n-x+y)) \end{aligned}$$

So the size of a small buffer is

$$b = \left\lfloor \frac{M}{(x-y+1)+\alpha(n-x+y)} \right\rfloor \tag{5}$$

The overhead C that we pay because of the setup depends on the value of α. Let $C(\alpha)$ be the function of α defined as follows:

$$C(\alpha) = \frac{x\,st}{b} + \frac{(n-x)ST}{B} \tag{6}$$

Let us consider the rational value of b:

$$\begin{aligned} C(\alpha) \times M = x\,st(x-y+1+\alpha(n-x+y)) \\ + \frac{(n-x)ST((x-y+1)+\alpha(n-y+1))}{\alpha}) \end{aligned} \tag{7}$$

The derivative $C'(\alpha)$ is

$$C'(\alpha) \times M = x\,st(n-x+y) - \frac{(n-x)ST(x-y+1)}{\alpha^2}$$

The function $C'(\alpha)$ is decreasing for α such that:

$$1 \leqslant \alpha \leqslant \sqrt{\frac{(n-x)ST(n-y+1)}{x\,st(n-x+y)}} = \alpha_{opt}$$

α has to be an integer value. The optimal value of α is either $\lfloor \alpha_{opt} \rfloor$ or $\lceil \alpha_{opt} \rceil$, depending on $C(\alpha)$. Moreover, the value of b was considered as a rational value within Equation (7). The choice of α also depends on the available memory M.

4 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 Benoit et al (2012)).

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

For $1 \leqslant i \leqslant n$, we express α_i as a function of the α_k's, with $k > i$. Then, we obtain an expression for α_{n+1}, and we can recursively derive all values of α_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 α_1 thanks to the memory constraint:

$$\alpha_1 = \frac{M}{\delta_1 + \sum_{k=2}^{n+1} P_2^k \delta_k} \tag{8}$$

Next, we express the cost as a function of α_2, replacing α_1 by its optimal value:

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

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:

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

We prove recursively that

$$\alpha_i = \sqrt{\frac{\delta_{i-1}}{st_{i-1}}\frac{\sum_{k=i}^{n} st_k P_{k+1}^n}{P_{i+1}^n \sum_{k=i}^{n+1} P_{i+1}^k \delta_k}} \tag{9}$$

for $i \geqslant 2$. Furthermore, we prove that minimizing the cost $C(\alpha_i)$ for fixed values of α_k, $i < k \leqslant n+1$, 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 α_{i+1}. We express Δ_i as a function of Δ_{i+1}, assuming that the optimal value of α_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 Δ_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, α_i is minimizing Δ_i, and because of the form of Δ_i, we have $\alpha_i = \sqrt{\frac{b}{a}\frac{\delta_{i-1}}{Mst_{i-1}}}$, 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 α_{i+1} appear in Δ_{i+1}. Therefore, in order to minimize the cost expressed as a function of α_{i+1}, we need to minimize $\Delta_{i+1}(\alpha_{i+1})$. This result is obtained directly, similarly to the way we obtained α_2, thanks to the form of the function.

This holds for $i \leqslant n$, 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

$$\alpha_n = \sqrt{\frac{\delta_{n-1}}{st_{n-1}}\frac{st_n}{\delta_n + \delta_{n+1}}} \tag{10}$$

We can therefore compute all optimal rational values of the α_i's. Interestingly, α_i depends only of the δ_k's and of the st_k's for $k \geqslant i-1$. Only α_1 accounts for M, because all other values are ratios.

We will discuss how to choose integer values in Section 4.2. Indeed, the integer value α_i can be either $\lceil \alpha_i \rceil$ or $\lfloor \alpha_i \rfloor$. For all $1 \leqslant i \leqslant n+1$, each choice for α_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 α_i using three heuristics (Up, Down and Closest) detailed in the next section.

4.2 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 4.1. 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 4.1, 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 polynomial heuristics to compute buffer capacities. The first one, described in Section 4.2.1, 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 4.2.2 and 4.2.3), with three variants per heuristic

depending on the rounding strategy used to obtain integer values. These heuristics consider that each buffer size is multiple of smaller buffer sizes because of the non-decreasing model reused within these heuristics. We also add two approaches associated to the two previous categories of heuristics so as to show the influence of the rounding step. Therefore, when the number of stages is not too large, we have computed optimal buffer capacities depending on the integer values of α_i ($1 \leqslant i \leqslant n$) computed previously by comparing all possible rounds. These two brute force rounding algorithms are not scalable but help us to validate the rounding step of α_k values ($1 \leqslant k \leqslant n$). Finally we provide in Section 4.2.4 optimal solutions obtained by a non scalable brute force algorithm for the most appropriate memory distribution into buffers. The optimal approach aims at helping us to compare previous heuristics to the best solution when the platform size is obviously not large because of the algorithm complexity (i.e., about 10 stages). This last approach is no longer based on the non-decreasing model.

4.2.1 SameB

The first heuristic SameB is the naive approach that we mentioned in Section 1 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 $1 \leqslant i \leqslant n+1$, buffer B_i contains data of size δ_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:

$$b = \left\lfloor \frac{M}{\sum_{i=1}^{n+1} \delta_i} \right\rfloor \tag{11}$$

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 $1 \leqslant i, j \leqslant n+1$. 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.

4.2.2 H1-Up, H1-Down, H1-Closest and H1-BF

We propose a set of heuristics, building upon the theoretical results of Section 4.1, 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 π such that $st_{\pi(i)} \leqslant st_{\pi(j)}$ if $\pi(i) < \pi(j)$, for $1 \leqslant i, j \leqslant n$. We then compute the sequence of the α_k's backwards according to Equations (10), (9), (8), 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 ($st_n \leqslant \max_{1 \leqslant i \leqslant n}(st_i)$), 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 α_k's. As mentioned before, in any case $b_n = b_{n+1}$ because b_{n+1} is an output buffer. As we know the position of st_n with the sorted sequence of setups ($\pi(n)$), we will be able to compute $b_{n+1} = b_{\pi(n)} = \prod_{k=1}^{\pi(n)} \alpha_k$ when all α_k will be known. However we can use $\prod_{k=2}^{\pi(n)}$ twice for the computation of α_1 as explained before in the non-decreasing model. Then we obtain the sequence of α_k's, $1 \leqslant k \leqslant n$.
2. We define three rounding policies (resp. Up, Down and Closest) to obtain an integer value for each α_k so as to compute each value of $b_k = \prod_{\ell=1}^{k} \alpha_\ell$ for $1 \leqslant k \leqslant n$ (and $b_n = b_{n+1}$) since $b_k \in \mathbb{N}^*$. We make our best to increase α_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 α_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 α_k. H1-Up, H1-Down and H1-Closest are three different heuristics corresponding to three respective ways to choose an integer value for α_k:
   (a) H1-Up: we consider α_k from $k = 1$ to n 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 α_k to the lower integer part, the size of the available memory is increasing, hence leading more room for further upgrades of the remaining α_k's.
   (b) H1-Down: we proceed as before, but we consider the α_k's in a decreasing order, from n to 1.
   (c) H1-Closest: with this last policy, we do not try to force each time the value of α_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 $1 \leqslant k \leqslant n$ using the integer values $\widehat{\alpha_\ell}$ of α_ℓ with $1 \leqslant \ell \leqslant k$ and $1 \leqslant k \leqslant n$. We recall that $b_{n+1} = b_n$;

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, $1 \leqslant k \leqslant n$.
5. We then consider each stage S_i that is a local maximum 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 reallocation 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.

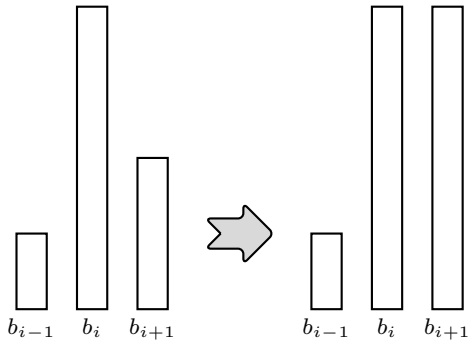


Fig. 1 Adapting the buffer size to b_i when the available memory is large enough.

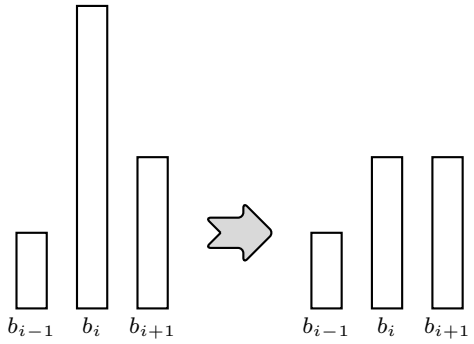


Fig. 2 Adapting the buffer size to b_{i+1} when the available memory is not large enough.

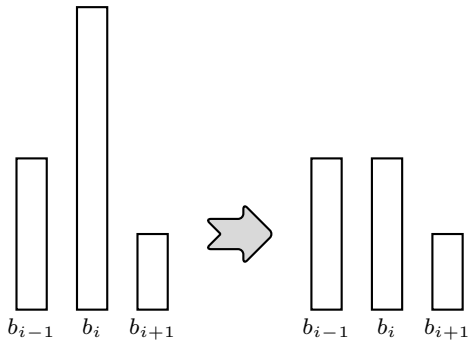


Fig. 3 Adapting the buffer size to b_{i-1} when the available memory is not large enough.

The performance of these heuristics depends not only on the rational α_k computation step based on the non-decreasing model but also on the rounding step described before. To evaluate the significance of this step and to measure the distance to an optimal rounding process we have developed a brute force rounding process named H1-BF. In this approach we consider, when the number of stages n is not too large, the 2^n ways to round the n rational values of α_k into $\widehat{\alpha}_k$ ($1 \leqslant k \leqslant n$). Indeed, the value of $\widehat{\alpha}_k$ is equal to $\lceil \alpha_k \rceil$ or $\lfloor \alpha_k \rfloor$. For each of the 2^n rounding solutions computed by this brute force approach we also process steps (3), (4) and (5). Note that we do not need to compute $\widehat{\alpha}_{n+1}$ because we know that $b_{n+1} = b_n$ if b_n is the input buffer capacity of the last stage when we consider buffers in their right place (after step (4)). Then we keep only the sequence of $\widehat{\alpha}_k$ ($1 \leqslant k \leqslant n$) corresponding to the lowest setup cost configuration for the buffers including in particular steps (5) of the previous description as already made for the three heuristics.

The three heuristics H1-Up, H1-Down and H1-Closest are compared together in Section 5 that shows simulations onto numerous scenarios. These heuristics are also compared to H1-BF.

4.2.3 H2-Up, H2-Down, H2-Closest and H2-BF

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 $1 \leqslant i \leqslant n$. 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} 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 α_1 is designed to take advantage of the memory considering the already computed values of each α_k with $1 < k \leqslant n$, using Equation (8) but by using only integer numbers, it is possible to re-evaluate α_1 considering the same formula by changing P_2^k to $b_k/\widehat{\alpha_1}$ since $b_k = \prod_{\ell=2}^{k} \widehat{\alpha_\ell} \times \widehat{\alpha_1}$. The values of the b_k are considered in the order given by the π function. Let α_1' be the new rational value of α_1:

$$\alpha_1' = \frac{M}{\delta_1 + \sum_{k=2}^{n+1} \frac{b_k}{\widehat{\alpha_1}} \delta_k} \tag{12}$$

4. We round down α_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 $b_i' = b'_{\pi^{-1}(k)}$ 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.

Moreover, as for heuristics H1-* and for the same motivation, we propose a brute force rounding process to H2, named H2-BF. This brute force approach also considers the 2^n rounding configurations (if n is not too large) instead of step (1) and computes for each configuration step (2), step (3) and step (4) and keeps only the configuration that corresponds to the lowest setup cost overhead. The obtained results are compared to H2-*.

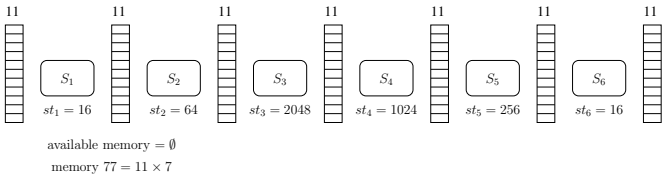


Fig. 4 Memory distribution using SameB strategy with 11 slots for each buffer. There is no available memory left. $C = 311.27$

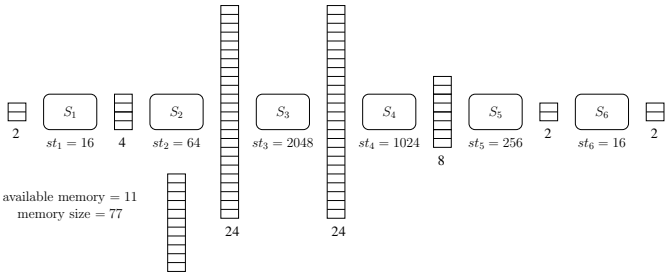


Fig. 5 Memory distribution using H1/H2-Down strategies. There are 11 memory slots left. $C = 373.33$

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.

4.2.4 Brute Force

To evaluate heuristic approaches described before, an optimal brute force algorithm (BruteForce) has been developed. This algorithm consists in exploring every buffer value such that the sum of the memory space split into buffers is less than equal to the total memory space M and such that two consecutive buffers are multiples. We exhaustively provide all valid solutions and we only keep one for which the overhead C is the smallest. This algorithm has obviously an exponential complexity and the number of explored solutions is huge (exponential with the the number of stages n) such that we are not able to compute solutions for platforms larger than $n = 10$ stages. However the considered applications are large enough to make the comparison of heuristic solutions with an optimal one possible. Numerous simulations show that our sub-optimal approaches in the general case based on the non-decreasing model find solutions close to this optimal.

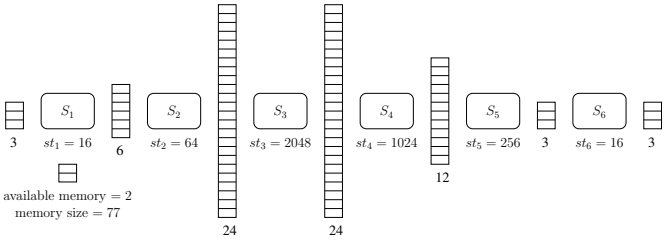


Fig. 6 Memory distribution using H1/H2-UP/Closest strategies. There are 2 memory slots left. $C = 277.33$

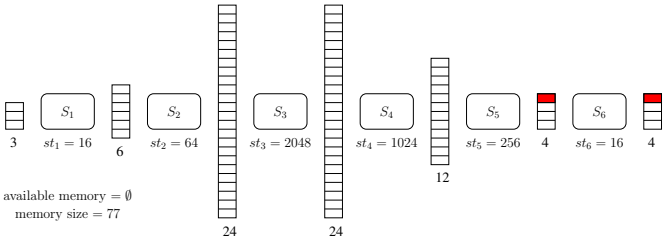


Fig. 7 Memory distribution using proportional buffer sizes within non-decreasing or non increasing decreasing stage subsequences. There is no available memory left. $C = 254.66$

4.3 Counter example and enhancement

The paper makes so far the assumption that all buffer sizes are proportional. Especially when setup cost are in a non-decreasing order, we have proven that the optimal cost follows naturally this assumption. However, in the general case, this assumption is too restrictive even if we have developed efficient strategies based on the optimal non-decreasing strategy. Indeed in the general case we only have to ensure that buffer sizes within a non-decreasing or a non-increasing sub-sequence are proportional to each other. But since two following subsequences share a stage within the sequence, the multiple values of the buffers with each sub-sequence maintains a dependence with its adjacent sub-sequences because of the shared buffers. If the shared stage S_i is a local maximum, every buffer size within the two adjacent sub-sequences are proportional to the input and output buffer sizes of S_i (say b_i and b_{i+1} with $b_i = b_{i+1}$). Otherwise if the shared stage S_i is a local minimum, B_i and B_{i+1} which are respectively the input and the output buffers of S_i have the same size $b_i = b_{i+1}$. Moreover this buffer size is proportional to each buffer size of the two sub-sequences. Figure 7 shows an example where the buffers of the sub-sequence S_1 to S_3 are proportional to 3 and the buffers of the sub-sequence S_3 to S_6 are proportional to 4, although every buffer size of the pipeline is a multiple of $b_3 = 24$ or $b_4 = 24$. Considering the same pipeline the same memory size, we show respectively three different memory allocations, each corresponding respectively to one strategy developed before: Figure 4 shows the sameB strategy, Figure 5 shows the H1/H2-Down strategies and Figure 6 shows the H1/H2-Up/Closest strategies. The obtained values for the overhead C is respectively $C = 311.27$, $C = 373.33$ and $C = 277.33$. However, by allowing to have a different sequence of proportional buffer size values with two adjacent sub-sequences, we show in Figure 7 that it is possible to decrease again the setup cost. In this counter example we obtain $C = 254.66$.

5 Simulation results

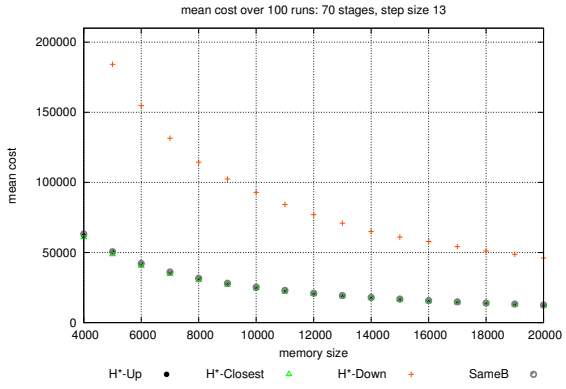
In order to evaluate the different heuristics, we conduct several simulations. Our major goal is to demonstrate the importance of an intelligent buffer allocation and its influence on the setup cost. All task sizes are set to $\delta_i = 1$, hence allowing a better comparison of the different solutions.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

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

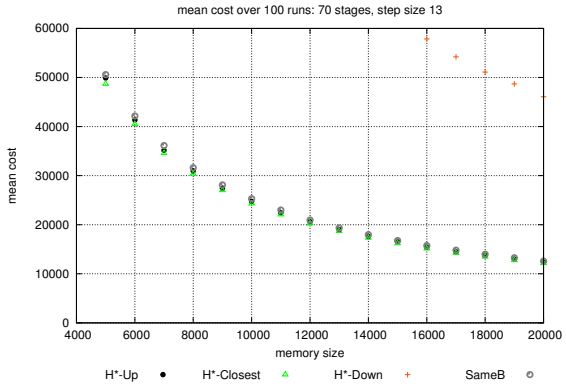
- Rand1 – In the first approach, the setup costs are randomly drawn between 5 and 100001 with a step size fixed in the experiment. This means for a step size of x, the random variable can take its value among the values $5 + k \times x < 100001$. This leads to setups of different costs, but which may 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 $1 \leqslant i \leqslant x$. Then, for each setup type, we randomly fix the number of stages that belong to each setup type, with the constraint that $\#(t_i) \geqslant \#(t_{i+1})$.

5.1 Non-decreasing setup costs

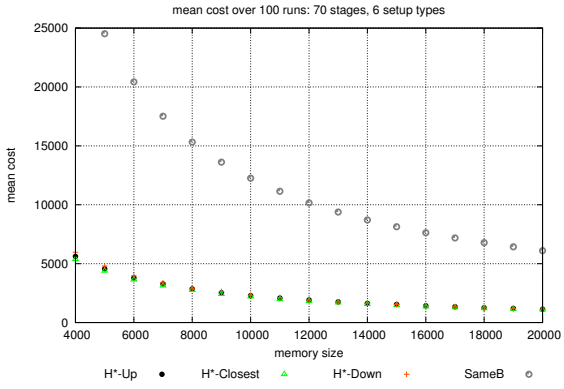
In the first experiment, we evaluate the behavior of the heuristics for applications with non-decreasing setup costs. In this case, we can compute the optimal rational buffer sizes as described in Section 4.1. The final buffer sizes then solely depend on the rounding technique used to convert the rational 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 8 shows the results for applications with 70 stages and



(a) Applications with randomly increasing setup costs.



(b) Detail: Applications with randomly increasing setup costs.



(c) Applications with 6 setup types.

Fig. 8 Mean cost for applications with 70 stages and non-decreasing setup costs.

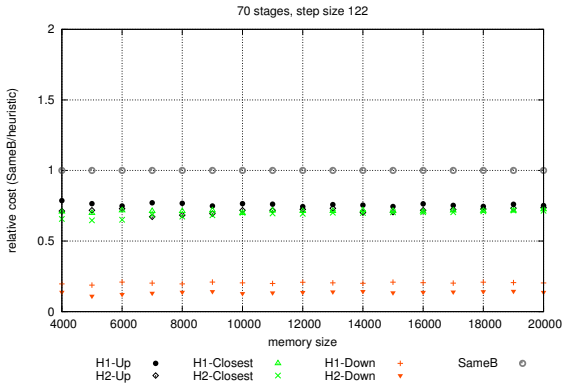
memory varing from 4000 to 20000: Figure 8(a) contains the results obtained with Rand1 (step size 13), whereas Figure 8(c) shows the results of the simulations with Rand2. Figure 8(b) shows the details of Figure 8(a). All graphs plot the mean values of the cost of 100 different applications. As can be seen, the heuristic SameB is outperformed in both cases. Still, in the case of platforms with low variance of setup costs (Rand1, Figure 8(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 α-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 8(c)), all H1 and H2 heuristics outperform SameB. We achieve results up to 5.2 times better than SameB (H*-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.

5.2 General case: setup costs in wave shape

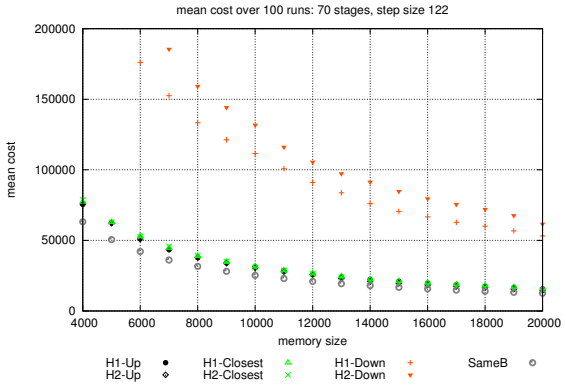
The second series of simulations tests the behavior of our heuristics on general platforms.

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 9. 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 9(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.

SameB outperforms all other heuristics. This good performance is due to the characteristics of Rand1 generated applications and its better performance in comparison to applications with non-decreasing setup costs



(a) Cost for one random application



(b) Mean cost for 100 random applications.

Fig. 9 Platform with 70 stages, setup costs are randomly generated.

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 approached or even outperformed.

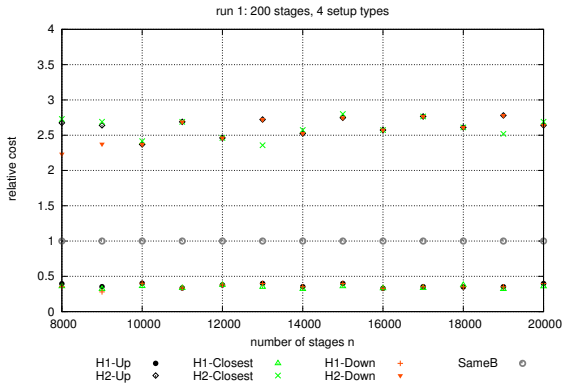
Figure 9(b) shows the mean cost of 100 applications with 70 stages. We observe that the previous ranking holds true and with increasing memory size, the total setup cost decreases as more buffer slots can be allocated for each stage. The relative performance nevertheless stays the same.

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.

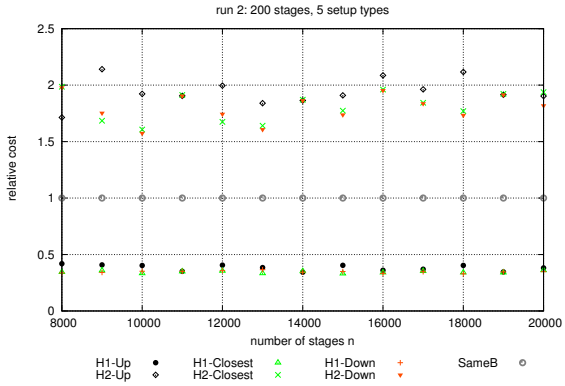
Figure 10 shows the results of an application with 200 stages and memory varying from 4000 to 20000 and we plot the results for 4, 5 and 6 setup types. The results of all heuristics are normalized with the result of SameB. A first observation reveals that the entire series of H1 heuristics performs awfully bad. Indeed, their ratio in general is less than 0.4 (0.4 and 0.2) for 4 (5 and 6) setup types, respectively (Figures 10(a), 10(b) and 10(c)). This means that H1-Up, H1-Closest and H1-Down are more than two times worse than SameB. As already explained in Section 4, 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 α-values does not anticipate this fact, the H1 heuristics adapt the buffer sizes in consequence (Section 4.2.2 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 $M = 15000$ (see Figure 10(a)), H2-Up and H2-Down achieve a ratio of $2,7$ and H2-Closest achieves $2,8$. When the number of setup types increases, the ratio between H2 and SameB further increases. For instance, in the case of 6 setup types (Figure 10(c)), the H2 heuristics are almost three times more as efficient as SameB. In the case of 6 setup types (Figure 10(c)), we make the following observation: when there is only a small memory size ($M \leqslant 10000$ 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 α_i-values ($1 < i \leqslant n + 1$) 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. H2-Up achieves results more than three times better than SameB. 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 $M = 18000$ with a value of 1.7, which means that SameB is still highly outperformed.

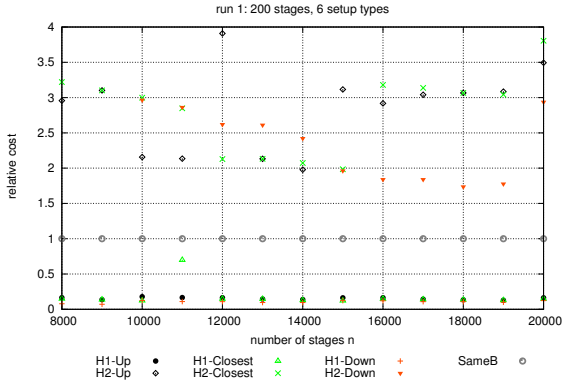
To validate our observations on one application with 200 stages, we plot in Figure 11 the mean result over 100 applications for H2 and SameB. The cost C decreases with increasing memory and the H2 heuristics achieve



(a) 4 setup types.



(b) 5 setup types.



(c) 6 setup types.

Fig. 10 Platform with 200 stages, increasing memory and different number of setup types.

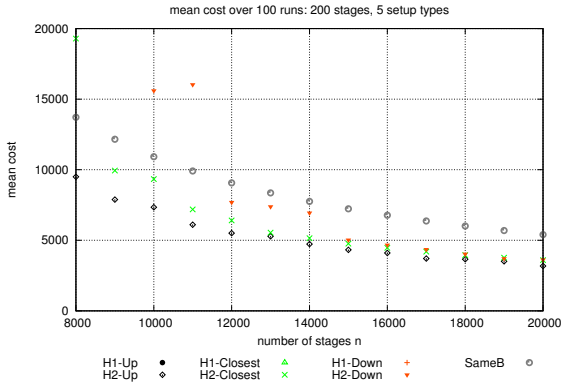


Fig. 11 Mean results over 100 applications with 200 stages and 5 setup types, zoom on H2 and SameB.

results that are in average 2.7 times better than the results found with SameB.

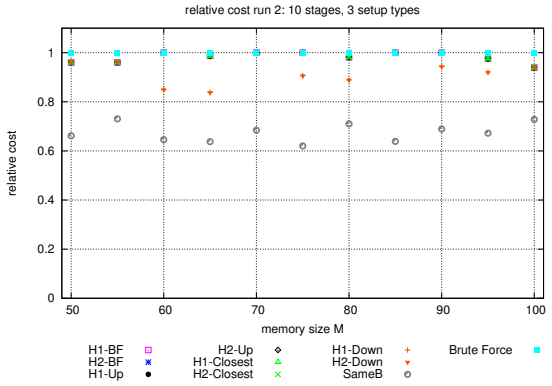
5.3 Comparison to the optimal solution

Finally, we compare our heuristics to the optimal solution BruteForce as well as to the two strategies H1-BF and H2-BF, where the α-values are rounded by brute force. These experiments allow to evaluate the behavior of our heuristics on small application instances. All applications of this section have 10 stages and are generated with Rand2.

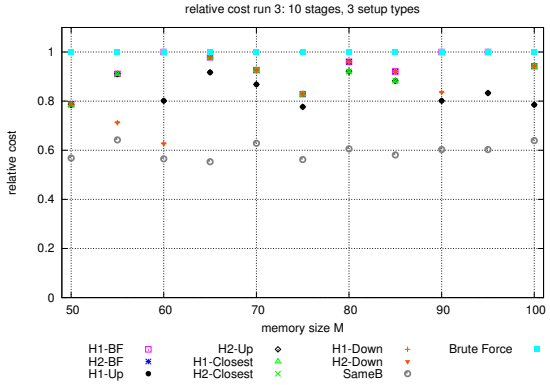
5.3.1 Non-decreasing setup costs

Figure 12 shows the results for two different application with 10 stages and non-decreasing setup costs. The memory size M varies between 50 and 100 and the applications have 3 types of setup costs. Both graphs show the relative cost, where the cost of each strategy is normalized by the cost of the BruteForce solution: $rel\ cost = cost_{BruteForce}/cost_{heuristic}$. As can be seen, SameB achieves the worse results: 60% of the optimal solution in Figure 12(a) and 50% in Figure 12(b). The H1 and H2-versions of the heuristics perform similarly. Closest achieves good results in all cases, but is outperformed by Down in two cases ($M = 80$ and $M = 90$ in Figure 12(a)). Depending on the application either Down or Up performs the poorest of the three strategies. The brute force rounding achieves near optimal results, but is not always able to find the optimal solution.

The mean results over 100 runs in Figure 13 confirm the observations and allow the following rating: SameB finds solutions twice as costly as BruteForce. H*-BF performs better than Closest, which performs better than Up, which outperforms Down. Still the gap is very



(a) Application with 10 stages and 3 setup types.



(b) Application with 10 stages and 3 setup types.

Fig. 12 Relative cost for two different applications with 10 stages and non-decreasing setup costs. 3 setup types.

close and we achieve a performance factor of less than 1.2.

5.3.2 General case

In the general case we once again state the superiority of the H2 approach over H1. Figure 14 shows results of general platforms with 3 setup types. Figures 14(a) and 14(b) show the relative performance of two applications. We observe that SameB achieves different ranking positions depending on the application: it is either situated between H1 and H2 (Figure 14(a)) or it outperforms all rounding heuristics (Figure 14(b)). The mean results of Figure 14(c) show, that SameB outperforms Up, Down and Closest in the general case, but not H*-BF. This result is not surprising as these experiments are conducted on very small application instances (less than 10 stages) with few available memory. These applications are too small to be able to fully exploit the capacity of the heuristics. Still, this result shows, that the usage of SameB even in this case is not recommended as the optimal solution can be computed BruteForce and SameB hardly finds the optimal solution.

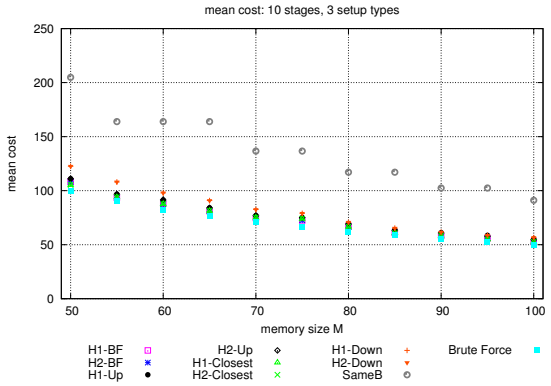


Fig. 13 Mean results over 100 applications with 10 stages and 3 setup types.

We also conducted experiments on applications with exactly one setup mountain, i.e., one peak. The result of one such application is shown in Figure 15.

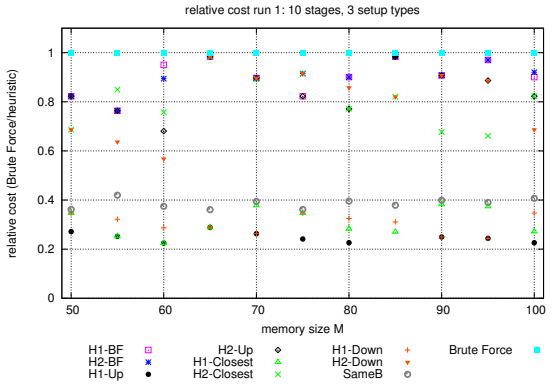
We once again find the ranking H*-BF, Up, Closest, Down and finally SameB and achieve results at 80% of the optimal cost in comparison to SameB, which achieves 55%. We suppose that this ranking holds true for larger applications in a real wave shape as one such application can be considered as a sequence of mountain-shape applications and all our experiments including wave shape assess the importance of a more sophisticated strategy for rounding and repair than SameB.

5.4 Summary

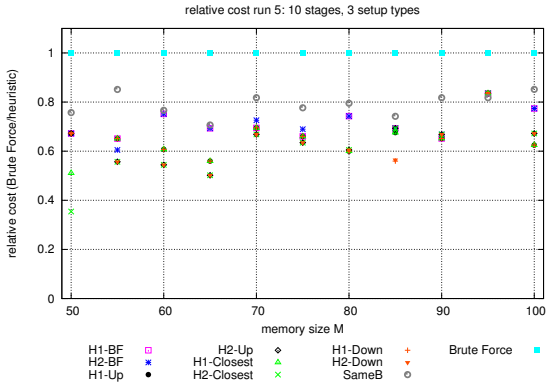
To summarize our results, we were able to show the importance of the different rounding techniques on the final result. Also, the naive heuristic SameB, where memory is distributed equally under the stages is efficient but in general cases a more intelligent way to allocate the memory leads to considerably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

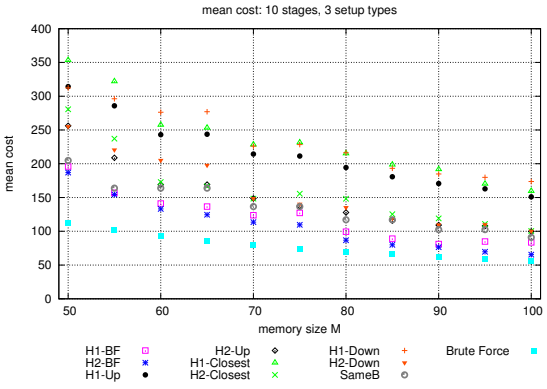
When applications provide at least one peak or have very diverging setup costs (Rand2), the simple SameB



(a) Application with 10 stages and 3 setup types. Run 1



(b) Application with 10 stages and 3 setup types. Run 5



(c) Mean cost of over 50 runs.

Fig. 14 Relative cost for three different applications with 10 stages. 3 setup types.

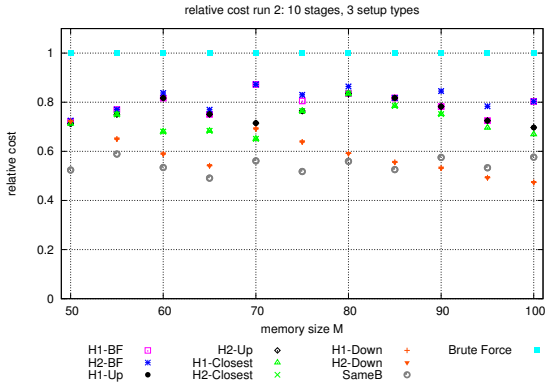


Fig. 15 Relative cost of an application with 10 stages and setup times in mountain shape.

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. When applications have about 10 stages and small memory size, we use the BruteForce. With $n = 10$ stages and $M = 100$, obtaining an optimal buffer allocation takes less than 20 seconds on processor Intel i7 2.0 GHz (mac book pro) but more than 4 minutes with $n = 10$ and $M = 200$.

6 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 straight forward approach is only efficient in cases of applications with homogeneous setup times or in particular cases of small fluctuations or zigzags. 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 then our result to applications with non-decreasing setup costs. We give an optimal rational solution of the problem in the latter case and give 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. We also provide optimal brute force algorithms to compute either optimal *ad hoc* buffer allocation with the lowest setup cost overhead or optimal rounding non-integer ratios provided by the non-decreasing model. Although these algorithms are not scalable and can be used only when both the number of stages and the memory size is small, it helps us to complete our analysis of the heuristics. Numerous simulations based on exhaustive scenarios show the relevance of our approach. The obtained performance 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 setups 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 place onto a peak have to be equal. This novel approach promises to be interesting form a theoritical point of view.

Acknowledgements A. Benoit is with the Institut Universitaire de France.
This work was supported in part by the ANR *RESCUE* project and by the Labex ACTION project (contract ANR-11-LABX-01-01).
Computations have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté. – Besançon – France.

References

Allahverdi A, Soroush H (2008) The significance of reducing setup times/setup costs. European Journal of Operational Research 187(3):978 – 984

Allahverdi A, Ng C, Cheng T, Kovalyov M (2008) A survey of scheduling problems with setup times or costs. European Journal of Operational Research 187(3):985–1032

Benoit A, Robert Y (2008) Mapping pipeline skeletons onto heterogeneous platforms. J Parallel and Distributed Computing 68(6):790–808

Benoit A, Coqblin M, Nicod JM, Philippe L, Rehn-Sonigo V (2012) 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, URL http://graal.ens-lyon.fr/ abenoit/papers/RR-7886.pdf

Bryan A, Norman (1999) Scheduling flowshops with finite buffers and sequence-dependent setup times. Comp & Indus Engineering 36(1):163 – 177

Burge J, Munagala K, Srivastava U (2005) Ordering pipelined query operators with precedence constraints. Technical Report 2005-40, Stanford InfoLab, URL http://ilpubs.stanford.edu:8090/705/

Guirado F, Ripoll A, Roig C, Hernàndez A, Luque E (2006) Exploiting throughput for pipeline execution in streaming image processing applications. In: Proceedings of the 12th international conference on Parallel Processing, Springer-Verlag, Berlin, Heidelberg, Euro-Par'06, pp 1095–1105

Hartley TDR, Fasih A, Berdanier CA, Özgüner F, Çatalyürek ÜV (2009) Investigating the use of gpu-accelerated nodes for sar image formation. In: CLUSTER, pp 1–8

Luh PB, Gou L, Zhang Y, Nagahora T, Tsuji M, Yoneda K, Hasegawa T, Kyoya Y, Kano T (1998) Job shop scheduling with group-dependent setups, finite buffers, and long time horizon. Annals of Operations Research 76:233–259

Ramanath R, Snyder W, Yoo Y, Drew M (2005) Color image processing pipeline. Signal Processing Magazine, IEEE 22(1):34–43

Schneider S, Andrade H, Gedik B, Biem A, Wu KL (2009) Elastic scaling of data parallel operators in stream processing. In: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, IEEE Computer Society, Washington, DC, USA, IPDPS '09, pp 1–12, DOI http://dx.doi.org/10.1109/IPDPS.2009.5161036

Subhlok J, Vondran G (1995) Optimal mapping of sequences of data parallel tasks. In: ACM SIGPLAN Notices, vol 30(8), pp 134–143

Subhlok J, Vondran G (1996) Optimal latency-throughput tradeoffs for data parallel pipelines. In: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures, ACM, p 71