# Data Memory Minimisation for Synchronous Data Flow Graphs Emulated on DSP-FPGA Targets

Marleen Adé, Rudy Lauwereins, J.A. Peperstraete
Katholieke Universiteit Leuven, ESAT Department, ACCA Laboratory
Kard. Mercierlaan 94, B-3001 Heverlee, Belgium
email: Marleen.Ade@esat.kuleuven.ac.be

## Abstract

*The paper presents an algorithm to determine the close-to-smallest possible data buffer sizes for arbitrary synchronous data flow (SDF) applications, such that we can guarantee the existence of a deadlock free schedule. The presented algorithm fits in the design flow of GRAPE, an environment for the emulation and implementation of digital signal processing (DSP) systems on arbitrary target architectures, consisting of programmable DSP processors and FPGAs. Reducing the size of data buffers is of high importance when the application will be mapped on Field Programmable Gate Arrays (FPGA), since register resources are rather scarce.*

## 1. Introduction and motivation

GRAPE (Graphical RApid Prototyping Environment) is an environment, developed at our laboratory, which facilitates the real-time emulation and implementation of synchronous DSP applications on heterogeneous target platforms consisting of DSPs and FPGAs [1]. Many aspects of GRAPE resemble the environments Ptolemy of UC Berkeley [2] and COSSAP of RWTH Aachen [3], currently further developed by Synopsys; the main distinction is that GRAPE is targeted at real-time execution whereas the other environments mainly target simulation.

GRAPE's design flow consists of four phases. In the specification phase, the application is described using an extended data flow model, called cyclo-static data flow (CSDF) [4], which is an extension of Lee's Synchronous Data Flow [5]. In short, the application is represented as a directed graph G=(N,E), where the nodes N represent computation tasks, and the edges E the communication of the results (called *tokens*) from a producing to a consuming task. The functionality of the nodes is specified in a conventional high level language like C or VHDL. The number of tokens a task produces respectively consumes during an execution phase of a task is known at compile time, allowing for a compile time analysis of the graph in the next phases of GRAPE's design flow and leading to highly efficient run-time code. Still in GRAPE's specification phase, the target architecture is specified as a connectivity graph, with an indication of the amount and type of resources each processing device possesses [6]. In the second phase, the amount of resources required by each of the tasks when executed on each of the processing devices, is estimated. Next, the application is mapped onto the target hardware. In this phase, each task is assigned to a specific processing device, a communication path is established for each edge in the application's graph and a compile time schedule order is determined per device that minimises the total makespan. In GRAPE's fourth and last design phase, code in C or VHDL is generated for each of the processing devices, consisting of a main program and communication primitives. Note that a single design flow is used for software targets (DSPs) as well as for hardware targets (FPGAs) [7].

The number of tokens produced by a task on an edge may be different than the number of tokens consumed by a task from that edge in SDF as well as in CSDF. Buffers are hence required on the edges to temporarily store the tokens that are produced but not yet consumed. The size of those buffers is left undetermined at specification time and hence cannot be taken into account in the resource estimation phase. When buffer sizes are still not fixed during scheduling, as is currently done in GRAPE, the scheduler has the highest possible freedom in ordering the task to obtain a small input-to-output latency, but the size of the buffers needed to implement the resulting schedule can be quite large and may exceed the available resources on the target. Resource limitations are of different nature for software targets (DSPs) than for hardware targets (FPGAs).

For DSPs, data buffers as well as program code and data variables are all stored in memory. The sum of these three memory requirements hence has to fit the available memory resources on the DSP device. Bhattacharyya [8] presented a method for a single processor target, which first reduces the amount of program and data memory by employing a single appearance schedule, and by then ordering the tasks in the single appearance schedule such that buffer memory is minimised.

For FPGAs, the implementation of a task requires combinatorial logic and routing resources, and to a lesser extent registers to store intermediate data variables. Data buffers on the other hand only require registers, a type of resource that is very scarce on FPGAs. To let a schedule fit within the severe resource limitations of an FPGA, it is important to limit the buffer sizes, already before scheduling, to the smallest values for which no deadlock will occur, even when this reduces the scheduling freedom. The increased latency caused by this limited scheduling freedom is more than compensated for by the fact that each task on an FPGA runs on a separate execution thread, i.e. concurrently with all other tasks assigned to the FPGA, where all tasks on a DSP run sequentially on a single thread.

This paper presents the algorithm to determine the close-to-smallest possible data buffer sizes for SDF applications for which we can guarantee that a deadlock free schedule exists. Due to space limitations, no proofs will be given and the algorithm will be explained for acyclic graphs only; the proofs and the extension to cyclic graphs may be found in [9]. Preliminary results, with proofs, for only a few of the simplest cases have been published in [10] and [11]. The algorithms do not consider the multiplexing of registers between buffers, since this is too expensive on FPGAs.



DAC 97, Anaheim, California

In the next section, the equations are presented to determine the minimum buffer sizes for basic graph entities consisting of single chains or single cycles. In the fourth section, it is shown how arbitrary graphs can be broken down in basic graph entities and how their mutual influence can be taken into account.

## 2. Simple case: basic graph entities

Any arbitrary graph can be decomposed into *chains* and *cycles*. We define a chain as an open path of edges and nodes independent of the orientation of the edges, for which only one path exists between any pair of nodes in the chain. A cycle is defined as a closed path of edges and nodes in which each node occurs only once, with the last node equal to the first node. When all edges originate from a node, we call the node a source of the cycle. When all arrive at the node, we call it a sink of the cycle. A *cluster* is a graph in which every pair of nodes is interconnected by at least two distinct paths. As a consequence, a chain can never be part of a cluster, nor partly nor completely. A cluster hence is composed of cycles solely. Figure 1 shows an example of an arbitrary graph. The dotted lines highlight its chains, the solid lines highlight the cycles and clusters.

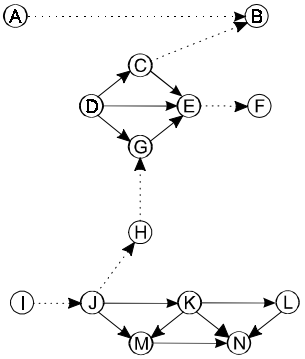


**Figure 1. 3 chains (*{ABC}, {EF}, {GHJI}*) and 9 cycles in 2 clusters (*{DCE}, {DEG}, {DCEG}, {JKM}, {KLN}, {KMN}, {JKNM}, {KLNM}, {JKLNM}*) in an arbitrary graph.**

### 2.1. Single chain

The simplest chain we can construct, has one edge and two nodes, as in Figure 2. Node $n_1$ produces $p_k$ and node $n_2$ consumes $c_k$ tokens at each firing of the node. A FIFO data buffer $b_k$ of depth $l_k$ stores the tokens on the edge that have been produced but not yet consumed (called the *live* tokens); $t_k^i$ denotes the number of initial tokens present in the buffer before the first node in the graph is fired. Initial tokens implement delays in the graph and as such can make cyclic graphs *live*, i.e. they can run forever.

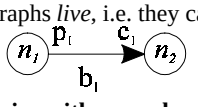


**Figure 2. Simplest chain, with $p_k$ and $c_k$ production and consumption numbers respectively.**

Implicitly, we will assume a uniform token size over the whole graph as well as a uniform buffer cost. As a consequence the buffer cost is only influenced by its length. Therefore all theory concentrates on minimising the buffer lengths for all edges of the graph, such that the total buffer length of the application is minimised, and still a deadlock free schedule can be found. Different token sizes and costs can be incorporated very easily into the algorithms: it is sufficient to multiply the production $p_k$, the consumption $c_k$ and the number of initial tokens $t_k^i$ of the buffer $b_k$ by the size of the token on edge $k$ and by the relative cost to implement $b_k$ (which will be cheaper for buffers implemented on DSP processors than for buffers realised on FPGAs, as explained above). By substituting all production and consumption numbers and all numbers of initial tokens in the equation that follow by the newly computed ones, the presented algorithms are still capable of minimising total buffer cost.

The minimum length $l_k^{\min}$ of buffer $b_k$ for which no buffer overflow will occur, may be computed using Equation ①. Note again that no proofs are given due to space limitations, but that they can be found in [9].

$$\begin{aligned} &\text{if } t_k^i \geq p_k + c_k - d_k \text{ then } l_k^{\min} = t_k^i \\ &\text{if } t_k^i < p_k + c_k - d_k \text{ then } l_k^{\min} = p_k + c_k - d_k + \delta_k \\ &\text{with } d_k = \gcd(p_k, c_k) \\ &\text{and } \delta_k = t_k^i \bmod d_k \end{aligned} \qquad ①$$

As an example, let us take the simple chain of Figure 2 with $p$=3 and $c$=5 and with 4 initial tokens. Intuitively, one would expect that the minimum required buffer length equals the least common multiple of $p$ and $c$ (i.e. $l^{min}$=15) possibly increased by the number of initial tokens (i.e. $l^{min}$=19). According to Equation ①, a buffer of length $l^{min}$=7 is sufficient.

In a graph with $N$ nodes in series (Figure 3), the minimum buffer length $l_k^{\min}$ of buffer $b_k$ is still given by Equation ①.

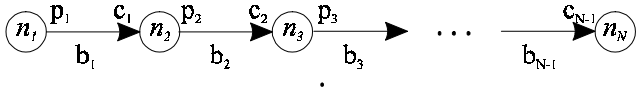


**Figure 3. A feedforward chain of N nodes.**

### 2.2. Single cycle

Consider the 4-node cycle of Figure 4, without initial tokens. When a cycle is traversed counterclockwise, some (directed) edges follow the traversed path and others are opposite to the traversed path. We call the set of edges pointing in the traversed direction the *up-edges* (marked with *u*) and the set of edges pointing in the opposite direction the *down-edges* (marked with *d*). By using Equation ① on each of the buffers, we would obtain $l_1^{\min(d)} = 2$, $l_2^{\min(d)} = 6$, $l_1^{\min(u)} = 3$ and $l_2^{\min(u)} = 4$. We will however show that these sizes for the data buffers would lead to a deadlock, and hence that Equation ① may not be used to compute the minimum buffer lengths in cycles.

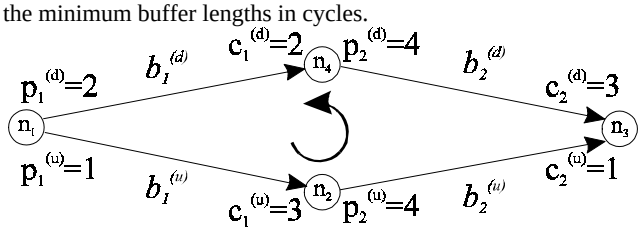


**Figure 4. Example of a 4-node cycle.**

To reduce the sum of all buffer sizes in the graph, it is best to have as much tokens as possible leaving the sink before the source is executed again. Figure 5 shows on each row the actual

number of tokens $t_k$ in the buffer $b_k$ when the nodes indicated at the top of the figure are fired. At the position surrounded by the rectangle, neither $n_4$, neither $n_2$ or $n_3$ can be executed, although $n_3$ has sufficient tokens on one of its inputs but not on the second one. The only possibility to continue the schedule is to fire $n_1$, but this would cause an overflow of buffer $b_1^{(d)}$ since $t_1^{(d)} > l_1^{\min(d)}$, as is seen in the last column.

| | | $\mathbf{n_1}$ | $\mathbf{n_4}$ | $\mathbf{n_1}$ | $\mathbf{n_1}$ |
|---|---|---|---|---|---|
| $\mathbf{t_1^{(d)}}$ | 0 | 2 | 0 | 2 | ***4*** |
| $\mathbf{t_2^{(d)}}$ | 0 | 0 | 4 | 4 | 4 |
| $\mathbf{t_1^{(u)}}$ | 0 | 1 | 1 | 2 | 3 |
| $\mathbf{t_2^{(u)}}$ | 0 | 0 | 0 | 0 | 0 |

**Figure 5. Evolution of the number of tokens in the buffers when the indicated nodes are fired.**

We will now indicate how the minimum buffer lengths for a cycle should be computed. The presented method is valid for every cycle which contains at least one source and one sink. Cycles without sources and sinks, also called *loops*, have only down-edges or only up-edges; they have to be treated in a slightly different way, as proven in [9]. We assume that the cycle is consistent [12], i.e. that it can be implemented with buffers of finite length. This consistency implies the relationship between the actual number of tokens present in the buffers as expressed in Equation ②; the left hand side covers the down-edges and vice versa. $N^{(d)}$ and $N^{(u)}$ are the number of down-edges and up-edges respectively.

$$\sum_{k=1}^{N^{(d)}}\left[\frac{\left(t_k^{(d)}-t_k^{i(d)}\right)}{p_k^{(d)}}\prod_{j=1}^{k-1}\frac{c_j^{(d)}}{p_j^{(d)}}\right]=\sum_{k=1}^{N^{(u)}}\left[\frac{\left(t_k^{(u)}-t_k^{i(u)}\right)}{p_k^{(u)}}\prod_{j=1}^{k-1}\frac{c_j^{(u)}}{p_j^{(u)}}\right] \quad ②$$

We call Equation ② the *cycle equation*.

$$\begin{cases}\text{one of the down}-\text{edges needs larger buffers iff}\\ \sum_{k=1}^{N^{(d)}}\left[\frac{\left(l_k^{\min(d)}-p_k^{(d)}+d_k^{(d)}-t_k^{i(d)}\right)}{p_k^{(d)}}\cdot\prod_{j=1}^{k-1}\frac{c_j^{(d)}}{p_j^{(d)}}\right]\le\\ \sum_{k=1}^{N^{(u)}}\left[\frac{\left(c_k^{(u)}-d_k^{(u)}+\delta_k^{(u)}-t_k^{i(u)}\right)}{p_k^{(u)}}\cdot\prod_{j=1}^{k-1}\frac{c_j^{(u)}}{p_j^{(u)}}\right]\\ \text{one of the up}-\text{edges needs larger buffers iff}\\ \sum_{k=1}^{N^{(d)}}\left[\frac{\left(c_k^{\min(d)}-d_k^{(d)}+\delta_k^{(d)}-t_k^{i(d)}\right)}{p_k^{(d)}}\cdot\prod_{j=1}^{k-1}\frac{c_j^{(d)}}{p_j^{(d)}}\right]\ge\\ \sum_{k=1}^{N^{(u)}}\left[\frac{\left(l_k^{\min(u)}-p_k^{(u)}+d_k^{(u)}-t_k^{i(u)}\right)}{p_k^{(u)}}\cdot\prod_{j=1}^{k-1}\frac{c_j^{(u)}}{p_j^{(u)}}\right]\\ \text{otherwise, no buffers need to be expanded}\end{cases} \quad ③$$

From Equation ②, we can deduct a criterion to determine whether one of the down-edges or up-edges of the cycle needs buffers larger than computed using Equation ①. We need larger buffers in a down-edge if an extra firing of the source would cause an overflow, i.e. when every buffer $b_k^{(d)}$ already contains at least $t_k^{(d)} = l_k^{\min(d)} - p_k^{(d)} + d_k^{(d)}$ tokens. The reason why we have to fire the source in the described situation is to be found in the up-edges where each buffer $b_k^{(u)}$ contains insufficient tokens to make the sink fireable. This means each buffer $b_k^{(u)}$ contains at most $t_k^{(u)} = c_k^{(u)} - d_k^{(u)} + \delta_k^{(u)}$ tokens. Filling out these conditions in Equation ②, gives us the algorithm of Equation ③.

Assuming that we know from Equation ③ that one of the directions (say, the down-direction, called the expansion direction) needs a buffer $b_k^{(d)}$ larger than computed using Equation ①, we need to determine which buffer to increase the size of (we will call this an expansion buffer, indicated with subscript *e*), and how large this expansion $e_e^{(d)}$ needs to be. We consider each buffer $b_k^{(d)}$ in the expansion direction in turn as the possible expansion buffer and compute its necessary buffer length $l_e^{(d)} = l_e^{\min(d)} + e_e^{(d)}$ using Equation ④. The buffer with the smallest expansion $e_e^{(d)}$ is selected as expansion buffer. Equation ④ is derived using a similar worst case reasoning as Equation ③ [9]. Note that the worst case conditions sometimes do not occur in practice and that hence the buffers may be overestimated[1].

$$e_e^{(d)} = d_e^{(d)} \cdot \left\{1 + \left|\frac{p_e^{(d)}}{d_e^{(d)}} \cdot \prod_{m=1}^{e-1}\frac{p_m^{(d)}}{c_m^{(d)}} \cdot \left(\sum_{k=1}^{N^{(u)}}\left(\frac{c_k^{(u)} - d_k^{(u)} + \delta_k^{(u)} - t_k^{i(u)}}{p_k^{(u)}} \cdot \prod_{j=1}^{k-1}\frac{c_j^{(u)}}{p_j^{(u)}}\right)\right.\right.\right. \quad ④$$
$$\left.\left.\left. - \sum_{k=1}^{N^{(d)}}\left(\frac{l_k^{\min(d)} - p_k^{(d)} + d_k^{(d)} - t_k^{i(d)}}{p_k^{(d)}} \cdot \prod_{j=1}^{k-1}\frac{c_j^{(d)}}{p_j^{(d)}}\right)\right)\right|\right\}$$

Applying this algorithm to the example of Figure 4, without initial tokens (i.e. $t_k^{i(x)} = 0, \delta_k^{(x)} = 0$ ), Equation ③ for the down edges becomes:

$$\frac{2-2+2-0}{2} + \frac{6-4+1-0}{4}\cdot\frac{2}{2} < \frac{3-1+0-0}{1} + \frac{1-1+0-0}{4}\cdot\frac{3}{1}$$

or 1.75<2; the down-direction is hence an expansion direction, as we have seen already from the schedule of Figure 5. Using Equation ④, we compute the possible extension for each buffer of the two down-edges:

$$e_1^{(d)} = 2\left\{1 + \left|\frac{2}{2}\cdot 1\cdot\left[\left(\frac{3-1+0-0}{1} + \frac{1-1+0-0}{4}\cdot\frac{3}{1}\right) - \left(\frac{2-2+2-0}{2} + \frac{6-4+1-0}{4}\cdot\frac{2}{2}\right)\right]\right|\right\},$$

$$e_2^{(d)} = 1\left\{1 + \left|\frac{4}{1}\cdot\frac{2}{2}\cdot\left[\left(\frac{3-1+0-0}{1} + \frac{1-1+0-0}{4}\cdot\frac{3}{1}\right) - \left(\frac{2-2+2-0}{2} + \frac{6-4+1-0}{4}\cdot\frac{2}{2}\right)\right]\right|\right\}$$

[1] This happens when the combination of production and consumption numbers is such that not all buffers are filled with the worst case number of tokens used to derive equations ③ and ④.

or $e_1^{(d)} = 2$ and $e_2^{(d)} = 2$. We may hence arbitrarily chose to increase the length of either buffer $b_1^{(d)}$ or buffer $b_2^{(d)}$ with 2 compared to its length computed using Equation ①.

A special type of cycle is depicted in Figure 6. It consists of a single chain with a bridge from source to sink. For this type of cycles, we can prove that the buffers in the chain may always be computed using Equation ①, and that an expansion, if needed, will always be in the bridging buffer.

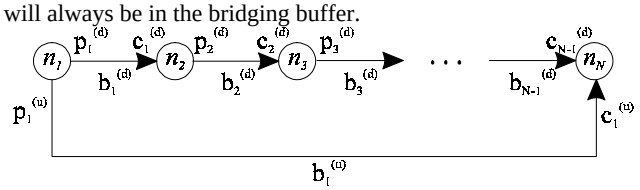


**Figure 6. Special type of cycle: single chain with a bridge from source to sink.**

## 3. General case: arbitrary acyclic SDF graphs

In previous section, we indicated how the minimum buffer sizes, for which a deadlock free schedule can be found, can be computed for single chains and cycles. In this section, we will indicate how an arbitrary acyclic graph can be broken down in such basic graph entities and how the mutual dependencies between the basic graph entities can be taken into account. At the end of section 2.1, we already indicated that the buffers in chains do not influence neither are influenced themselves by any other buffer in the graph, since each such buffer only belongs to one chain. Cycles however can share buffers though which their buffer lengths are coupled. By definition, two cycles belonging to a different cluster do not share and edge. They hence do not influence each others minimal buffer lengths. These intuitively introduced observations lead to following algorithm for computing the minimal buffer lengths in any acyclic SDF graph; again, formal proofs are given in [9].

Algorithm:

1. classify edges in chains and clusters
2. compute the chains : for each edge equation ① applies. This computation can be done irrespectively of all other buffers, since buffers in a chain are not influenced by and do not influence themselves any other buffer. Remove the chains from the graph.
3. compute each cluster. The buffer lengths in each cycle of the same cluster may be influenced by the production and consumption behavior of other cycles in the same cluster, but not by cycles belonging to a different cluster.
   1. for each edge in the cluster, compute an initial buffer length, according to equation ①.
   2. detect all cycles
   3. mark every cycle as uncomputed.
   4. while uncomputed cycles remain
      1. determine for each cycle the set of expansion candidate buffers according to Equation ③. If none of the inequalities holds, the set of expansion candidate buffers is empty. If the first inequality is true, the set of expansion candidate buffers consists of all down-edges of the cycle. The set contains all up-edges if the second inequality holds.
      2. mark the cycles with an empty set of expansion candidate buffers as computed
      3. order the uncomputed cycles, according to the principle that a cycle whose set of expansion candidate buffers is completely contained by the set of another cycle is to be computed before the other cycle. If several of such cycles exist, the one with the smallest set is selected first. If cycles have identical sets, then their ordering is random. For the remaining cycles the ordering is random. The goal of this heuristic rule for ordering the cycles is to ensure that an expansion in one of the cycles is preferably done in a buffer which is also an expansion candidate in another cycle and hence also helps in reducing the expansion needs in the other cycle; this reduces the total buffer requirements.
      4. for the first cycle in the ordered list :
         1. compute the expansion on every expansion candidate buffer using Equation ④
         2. if only one buffer is present with smallest expansion, update its length by adding the expansion. Mark the cycle as computed
         3. if more than one buffer is present with smallest expansion, select the buffer that has the highest frequency as expansion candidate in the uncomputed cycles. Update its length and mark the cycle as computed. Again, this heuristic rule aims at the maximum reuse of this expansion in other cycles.

The computational complexity of this algorithm is $O\left(e_{c(\max)}^3\right)$, where $e_{c(\max)}$ is the maximum number of edges over all clusters.

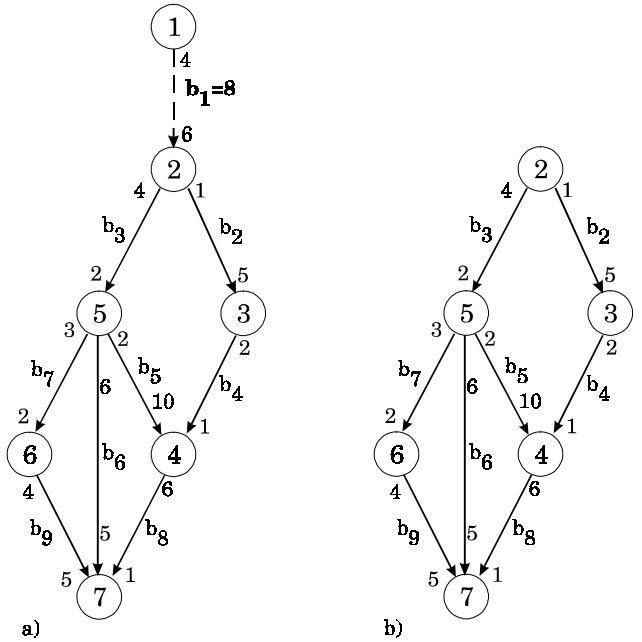


**Figure 7. Applying Steps 1 and 2 to the graph depicted in a) yields one chain: $b_1$. Its buffer length, computed using Equation ①, is indicated next to the edge. Picture b) shows the graph after removing the chain-edge and the not-connected node. It is the input for Step 3.**

We will now apply the above algorithm to the example of Figure 7.

1. classify edges in chains and clusters. Figure 7.a shows the edge belonging to a chain as a dashed line and the ones belonging to clusters as solid lines.
2. compute the chains : use equation ① for each edge. For example, the length of buffer $b_1$ equals $l_1^{\min} = p_1 + c_1 - \gcd(p_1, c_1) = 4 + 6 - 2 = 8$. The resulting buffer length is indicated in bold next to the chain edge. Removing the chain from the graph, we obtain Figure 7.b.
3. compute each cluster. As depicted in Figure 8, there is one clusters $C_1$: $C_1 = \{b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9\}$.

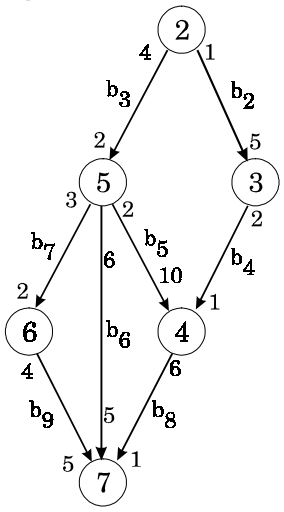


**Figure 8. Results of applying Step 3 on the graph of Figure 7.b.**

1. For each edge in the cluster, we compute an initial buffer length, according to equation ①. This results in $l_2$=5, $l_3$=4, $l_4$=2, $l_5$=10, $l_6$=10, $l_7$=4, $l_8$=6, $l_9$=8.
2. We determine all cycles. In the following list of cycles, the down-edges are marked with a bar over the buffer name: $r_1 = \{\overline{b_2}, b_3, \overline{b_4}, b_5\}$, $r_2 = \{\overline{b_5}, b_6, \overline{b_8}\}$, $r_3 = \{\overline{b_6}, b_7, b_9\}$, $r_4 = \{\overline{b_5}, b_7, \overline{b_8}, b_9\}$, $r_5 = \{\overline{b_2}, b_3, \overline{b_4}, b_6, \overline{b_8}\}$, $r_6 = \{\overline{b_2}, b_3, \overline{b_4}, b_7, b_9, \overline{b_8}\}$.
3. We mark every cycle as uncomputed .
4. As long as there are uncomputed cycles, we continue our computations.
   1. We determine for each cycle the set of expansion candidate buffers according to Equation ③. For cycle $r_1$, the second check of Equation ③ is true, meaning that an expansion of an up-edge is needed. The set of expansion candidate buffers for cycle $r_1$ is hence $E_1 = \{b_3, b_5\}$. Analogously, the sets of expansion candidate buffers for the other cycles are determined, yielding: $E_2 = \{b_6\}$, $E_3 = \{\overline{b_6}\}$, $E_4 = \{b_7, b_9\}$, $E_5 = \{b_3, b_6\}$, $E_6 = \{b_3, b_7, b_9\}$.
   2. None of the sets of expansion candidate buffers is empty. Hence, none of the cycles can be marked as computed.
   3. Since $E_2 = E_3 \subset E_5$, $r_2$ and $r_3$ have to be computed before $r_5$, since a possible expansion on b6 required in $r_2$ or $r_3$ can be used to reduce the needed expansion in $r_5$. Similarly, since $E_4 \subset E_6$, $r_4$ needs to be computed before $r_6$. The ordering rules yield no other restrictions. A possible ordering of the set of uncomputed cycles is $U = \{r_4, r_6, r_2, r_3, r_5, r_1\}$.
   4. For the first cycle in the ordered list, $r_4$, we start computing the expansion:
      1. Using Equation ④, we compute the expansion on each of the buffers in the set of expansion candidate buffers $E_4$. This yields $e_7$=8 and $e_9$=10.
      2. Only one buffer has the smallest expansion, namely $b_7$. The expanded length for $b_7$ becomes $l_7$=4+8=12. Cycle $r_4$ is marked as computed.
4. Since there are still uncomputed cycles, we restart our computations with step 1.
   1. We re-compute the set of expansion candidate buffers for all uncomputed cycles containing the previously expanded buffer $b_7$, i.e. for cycle $r_6$. Equation ③ yields possible expansion in the up-edges: $E_6 = \{b_3, b_7, b_9\}$. Note that the updated buffer length $l_7$=12 in all equations!
   2. Since $E_6$ is not empty, $r_6$ cannot be removed from the list of uncomputed cycles.
   3. The only remaining ordering rule requires that $r_2$ and $r_3$ have to be computed before $r_5$, since $E_2 = E_3 \subset E_5$. A possible ordering of the set of uncomputed cycles is $U = \{r_6, r_2, r_3, r_5, r_1\}$.
   4. For the first cycle in the ordered list, $r_6$, we start computing the expansion:
      1. Using Equation ④, we compute the expansion on each of the buffers in the set of expansion candidate buffers $E_6$. This yields e3=6, $e_7$=9 and $e_9$=18.
      2. Only one buffer has the smallest expansion, namely $b_3$. The expanded length for $b_3$ becomes $l_3$=4+6=10. Cycle $r_6$ is marked as computed.
4. Again, since there are uncomputed cycles, we restart our computations with step 1.
   1. We re-compute the set of expansion candidate buffers for all uncomputed cycles containing the previously expanded buffer $b_3$, i.e. for cycles $r_1$ and $r_5$. Equation ③ indicates that no expansion is needed anymore in cycle $r_1$, i.e. the expansion of $b_3$ needed in cycle $r_6$ is also sufficient to remove the deadlock problem of cycle $r_1$. The set of expansion candidate buffers $E_1$ is hence empty. Equation ③ applied to cycle r5 yields $E_5 = \{b_3, b_6\}$.
   2. Since $E_1$ is empty, $r_1$ is removed from the list of uncomputed cycles.
   3. The only remaining ordering rule requires that $r_2$ and $r_3$ have to be computed before $r_5$, since $E_2 = E_3 \subset E_5$. A possible ordering of the set of uncomputed cycles is $U = \{r_2, r_3, r_5\}$.
   4. For the first cycle in the ordered list, $r_2$, we start computing the expansion:
      1. Using Equation ④, we obtain $e_6$=20.

2. Only one buffer has the smallest expansion, namely $b_6$. The expanded length for $b_6$ becomes $l_6$=10+20=30. Cycle $r_2$ is marked as computed.

4. Again, since there are uncomputed cycles, we restart our computations with step 1.
   1. We re-compute the set of expansion candidate buffers for all uncomputed cycles containing the previously expanded buffer $b_6$, i.e. for cycles $r_3$ and $r_5$. Equation ③ indicates that no expansion is needed anymore in cycle $r_3$ as well as in cycle $r_5$. The sets of expansion candidate buffers $E_3$ are hence empty.
   2. Since $E_3$ and $E_5$ are empty, $r_3$ and $r_5$ are removed from the list of uncomputed cycles.
4. No uncomputed cycles remain for cluster $C_1$.

Figure 9 shows the finally required buffers lengths.

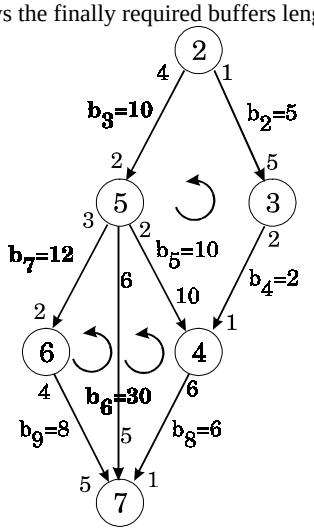


**Figure 9. Results of applying Step 3 on the cluster of Figure 8.**

The obtained buffer lengths are the absolute minimum for which a deadlock-free schedule exists. The sum of the length of all buffers equals 91. When each buffer length $l_k$ would have been computed as the least common multiple of the production and consumption numbers $p_k$ and $c_k$, the total length would also have been 95, but *NO* deadlock-free schedule exists in this case. To the best of the authors' knowledge, the only known method to determine the buffer length (without actually scheduling the application) such that it is guaranteed that a deadlock-free schedule can be found, computes the buffer length on an edge as the product of the production number $p_k$ and the repetition rate of the producing node, i.e. the number of times the node is executed before the schedule is repeated. Applying this method to the example above yields a total buffer length of 478, which is an order of magnitude larger than the one obtained with the algorithm we present.

## 4. Conclusion

The paper indicated that it is important to determine the minimal possible data buffer sizes for which a deadlock free schedule can be found, when an SDF application is implemented on today's FPGAs that have severe register limitations. It first presented the equations to compute minimum buffer sizes for single chains and single cycles. It then showed how arbitrary acyclic graphs can be broken down into single chains and clusters and how their mutual influence may be computed. These algorithms where developed to fit into the design flow of GRAPE, a programming environment facilitating the real-time emulation and implementation of DSP applications on heterogeneous target architectures consisting of DSPs and FPGAs.

## 5. Acknowledgements

Marleen Adé is a post-doctoral researcher of the K.U.Leuven. Rudy Lauwereins is a senior research associate of the FWO Vlaanderen. This project is partly sponsored by IUAP-50, 4/20, 4/24, by various Esprit, ESA, IWT, FWO and Texas Instruments projects. K.U.Leuven-ESAT is a member of the DSP Valley™ network.

## 6. References

1 Rudy Lauwereins, Marc Engels, Marleen Adé, J.A. Peperstraete, "Grape-II: A System-Level Prototyping Environment for DSP Applications", *IEEE Computer*, Feb. 1995, pp. 35-43.

2 J. Buck, S. Ha, E.A. Lee, D.G. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems", *Int. Journal of Computer Simulation*, Vol. 4, April 1994, pp. 155-182.

3 Synopsys Inc., 700 E. Middlefield Rd., Mountain View, CA 94043, USA, COSSAP User's Manual.

4 Greet Bilsen, Marc Engels, Rudy Lauwereins, J.A. Peperstraete, "Cyclo-Static Dataflow", *IEEE Transactions on Signal Processing*, Vol. 44, No. 2, Feb. 1996, pp. 397-408.

5 E.A. Lee, D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Transactions on Computers*, Vol. C-36, No. 1, Jan. 1987, pp. 24-35.

6 Greet Bilsen, Marc Engels, Rudy Lauwereins, J.A. Peperstraete, "Compile-time Makespan-optimal Multi-resource Mapping for Hardware/Software Co-design", *KULeuven Technical Report ESAT-ACCA 95-02*.

7 Marleen Adé, Rudy Lauwereins, J.A. Peperstraete, "Hardware-Software Co-design with GRAPE", *Proceedings of the 6th Int. Workshop on Rapid System Prototyping*, IEEE Computer Society Press, Ed. R. Lauwereins, Chapel Hill, North-Carolina, USA, June 1995, pp. 40-47.

8 S.S. Bhattacharyya, P.K. Murthy, E.A. Lee, "Converting Graphical DSP Programs into Memory Constrained Software Prototypes", Proc. of the 6th IEEE Int. Workshop on Rapid System Prototyping, IEEE Computer Society Press, Ed. R. Lauwereins, Chapel Hill, NC, June 1995, pp. 194-200.

9 Marleen Adé, "Data Memory Minimisation for Synchronous Data Flow Graphs Emulated on DSP-FPGA Targets" , *Ph.D. Diss.*, KULeuven-ESAT, Oct 1996 (downloadable from http://www.esat.kuleuven.ac.be/acca/reports/phd_marleen.html).

10 Marleen Adé, Rudy Lauwereins, J.A. Peperstraete, "Minimum memory buffers in DSP applications", *Electronics Letters*, March 17, 1994, Vol. 30, No. 6, pp.469-471.

11 Marleen Adé, Rudy Lauwereins, J. A. Peperstraete, "Buffer Memory Requirements in DSP Applications", *Proceedings of the 5th IEEE International Workshop on Rapid System Prototyping*, Ed. N. Kanopoulos, IEEE Computer Society Press, Grenoble, France, June 1994, pp. 108-123.

12 E.A. Lee, "Consistency in Data Flow Graphs", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 2, No. 2, April 1991.