

Implementing DSP Applications on Heterogeneous Targets Using Minimal Size Data Buffers

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 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 targetted 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 and VHDL. The number of tokens a task produces resp.

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 minimizes 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 makespan, 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 minimized.

For FPGAs, the implementation of a task requires combinatorial logic and routing resources, and to a lesser extend also 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 makespan 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 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].

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 for which only one path exists between any pair of nodes in the chain, irrespective of the direction of the edges. 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. Figure 1 shows an example of an arbitrary graph. The dotted lines in part *a* highlight its chains, and in part *b* they highlight the cycles.

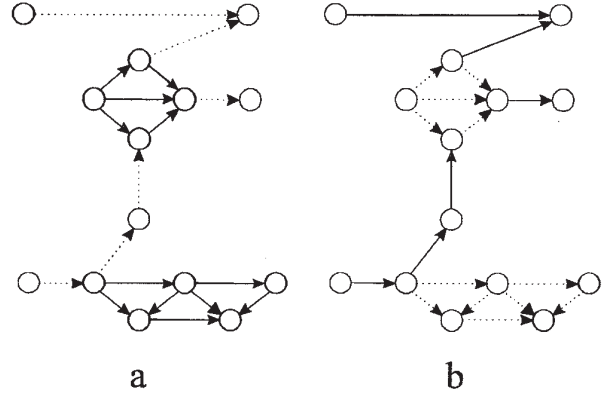


Figure 1. 3 chains (a) and 6 cycles (b) in an arbitrary graph.

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.

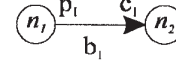


Figure 2. Simplest chain, with p_k and c_k production and consumption numbers respectively.

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].

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

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.

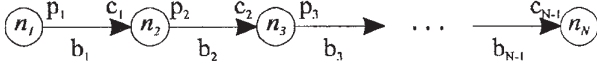


Figure 3. A feedforward chain of N nodes.

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.

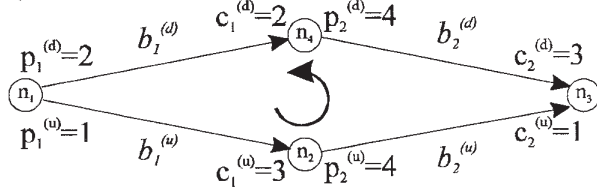


Figure 4. Example of a 4-node cycle.

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.

	n_1	n_4	n_1	n_1
$t_1^{(d)}$	0	2	0	2
$t_2^{(d)}$	0	0	4	4
$t_1^{(u)}$	0	1	1	2
$t_2^{(u)}$	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. 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 respectively up-edges.

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

We call Equation ② the *cycle equation*.

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.

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

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.

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

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:

$$e_1^{(d)} = 2 \cdot \left\{ 1 + \left\lfloor \frac{2}{2} \cdot \left[\left(\frac{3-1+0-0}{2} + \frac{1-1+0-0}{4} \cdot \frac{3}{1} \right) - \left(\frac{1}{2} + \frac{4}{4} \cdot \frac{2}{2} \right) \right] \right\rfloor \right\}, \\ e_2^{(d)} = 1 \cdot \left\{ 1 + \left\lfloor \frac{4}{1} \cdot \frac{2}{2} \cdot \left[\left(\frac{3-1+0-0}{2} + \frac{1-1+0-0}{4} \cdot \frac{3}{1} \right) - \left(\frac{1}{2} + \frac{4}{4} \cdot \frac{2}{2} \right) \right] \right\rfloor \right\}$$

or $e_1^{(d)} = 2$ and $e_2^{(d)} = 2$. We may hence arbitrarily chose to increase the length of either buffer $b_1^{(d)}$ either 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.

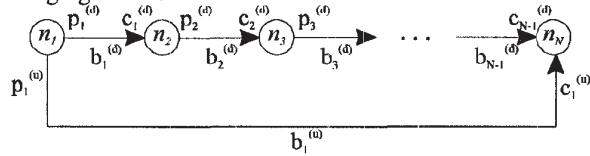


Figure 6. Special type of cycle: single chain with a bridge from source to sink.

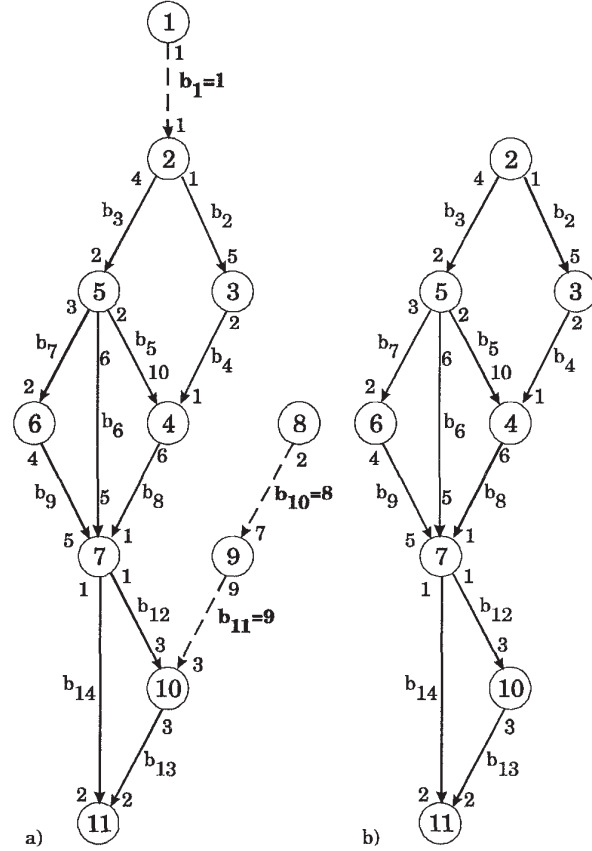


Figure 7. Applying Step 1 to the graph depicted in a) yields two chains: b_1 and (b_{10}, b_{11}) . Their buffer lengths, computed using Equation ①, are indicated next to the edge. Picture b) shows the graph after removing the chain-edges and the not-connected nodes. It is the input for Step 2.

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.

The procedure to compute the buffer lengths of an acyclic graph, minimizing its total buffer length, consists of three main steps, that will be more detailed in the sequel:

1. Determine all chains in the graph and compute for each edge the buffer length using Equation ①.

2. Group the cycles in clusters. Cycles belonging to the same cluster may influence each other's buffer lengths. For a cluster with e_c edges and n_c nodes, determine the $e_c - n_c + 1$ cycles for which independent cycle equations ② can be constructed. We call these $e_c - n_c + 1$ cycles the *primary cycles*; the other cycles of the cluster are called the *secondary cycles*.
3. For each cluster, order the primary cycles and compute in this order the buffer lengths using an equation similar to Equation ④.

We will now describe the consecutive steps in the procedure in more detail:

Step 1:

1. Determine all chains in the graph G.
2. Compute for each edge in each chain the minimum necessary buffer length using Equation ①.
3. Remove all computed edges from the graph G. Remove all nodes with zero edges (Figure 7).

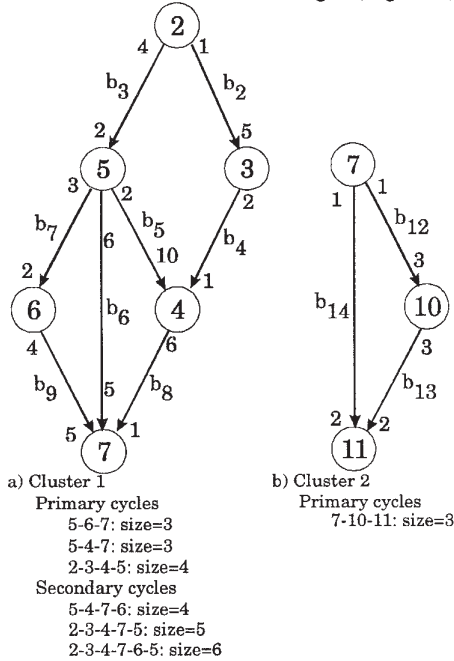


Figure 8. Results of applying Step 2 on the graph of Figure 7.b. For each of the two clusters, the primary and secondary cycles are indicated, with their size. The primary cycles are listed in the order they are found.

Step 2:

1. Determine all cycles in the graph G. Determine the size of each cycle. The size of a cycle is defined as its number of nodes. Mark all edges of the graph G as uncovered.
2. While graph G is not empty, do:
 - 2.1 Select an arbitrary cycle with the smallest size from the graph G and add it to a new cluster C_i ,

without removing it from the graph G. Indicate that this cycle has an independent cycle equation and mark its edges as covered. Note that $C_i = \langle N_i, E_i \rangle$ is also a graph.

- 2.2 While graph G contains cycles that have at least one uncovered and at least one covered edge, do:

Select an arbitrary cycle from graph G, with the smallest size and with at least one uncovered and at least one covered edge. Multiple covered edges should be contiguous. Add the cycle to cluster C_i without removing it from graph G. Indicate that this cycle has an independent cycle equation. Mark its edges as covered.

- 2.3 Remove all covered edges from graph G. Remove all nodes with zero edges.

Each cluster C_i will now contain $e_c - n_c + 1$ primary cycles and also some secondary cycles (Figure 8). By construction, buffer lengths of edges in one cluster do not influence the buffer length of any edge in any other cluster.

Step 3:

For each cluster C_i do:

1. Compute an initial buffer length for every edge according to Equation ① and mark the edge as not-frozen. Note that this buffer length may be increased in the sequel due to the influence of other edges in the cluster.
2. Determine the order in which the buffer lengths will be updated by grouping the primary cycles of the cluster in computation levels L_i . Each computation level contains one or more primary cycles of which the buffer lengths need to be computed together. The buffer lengths of another computation level may be computed separately to speed up the computations, at the cost of a sub-optimal result. The algorithm to determine the computation levels, is explained in Appendix A.
3. For computation level L_1 to $L_{\max}$ do:
 - 3.1 Construct a subgraph G_i for level L_i . It contains all edges and nodes of the primary cycles of L_i as well as all edges and nodes of the levels L_j (with $j < i$) that form a path between any two nodes of level L_i that are interconnected with frozen edges.
 - 3.2 While there are not-frozen edges in G_i do:
 - Select a primary cycle of L_i that contains at least one not-frozen edge and that preferably contains an edge that became temporarily-frozen during previous iteration. Select an arbitrary node of that cycle as a reference node. Number all nodes in the cycle counterclockwise, starting with '1' for the

reference node. For an up-edge, define $v_i=p_i$ and $w_i=c_i$; for a down-edge, define $v_i=c_i$ and $w_i=p_i$.

- Check whether a down-edge of the selected primary cycle needs to expand. Initialize sum S to zero. For every not-frozen up-edge of the selected cycle, add

$$\frac{c_k^{(d)} - d_k^{(d)} + \delta_k^{(d)} - t_k^{(d)} \prod_{j=1}^{k-1} \frac{w_j}{v_j}}{v_k^{(d)}} \quad \textcircled{5} \text{ to } S; \text{ for}$$

every not-frozen down-edge of the selected cycle, subtract $\frac{l_k^{(u)} - p_k^{(u)} + d_k^{(u)} - t_k^{(u)} \prod_{j=1}^{k-1} \frac{w_j}{v_j}}{v_k^{(u)}}$

$\textcircled{6}$ from S . If the cycle has only temp-frozen and frozen down-edges, then, for every contiguous path P of frozen edges of the selected cycle, determine all paths Q_i through frozen edges in G_i between the start and end nodes of P ; else, for every contiguous path P of frozen or temp-frozen edges of the selected cycle, determine all paths Q_i through frozen or temp-frozen edges in G_i between the start and end nodes of P . For each up-edge in Q_i , add a contribution $\textcircled{5}$ to a temporary sum S_j ; for each down-edge in Q_i , subtract a contribution $\textcircled{6}$ from S_j . Add the maximum of all temporary sums S_j to S . If the resulting S is non-negative, a down-edge of the selected cycle of L_i needs to be expanded. Compute for each not-frozen edge of the selected cycle the required expansion using Equation $\textcircled{7}$. If the cycle has no not-frozen edges, compute the required expansion of all temp-frozen edges using Equation $\textcircled{7}$. Expand the edge with the smallest $e_k^{(d)}$. Mark all not-frozen edges of the selected cycle as temp-frozen. Mark the temp-frozen edges, not belonging to the selected cycle, but belonging to a cycle of G_i containing the expanded edge, as not-frozen; this causes the algorithm to re-compute all cycles containing the expanded edge, to see whether this expansion requires other buffers to expand too.

$$e_k^{(d)} = \left(\left\lfloor \frac{v_k^{(d)} \prod_{j=1}^{k-1} \frac{v_j}{w_j} S}{d_k^{(d)} \prod_{j=1}^{k-1} \frac{v_j}{w_j}} \right\rfloor + 1 \right) d_k^{(d)} \quad \textcircled{7}$$

- If no down-edge needs to expand, check whether an up-edge needs to be expanded. Use the same procedure as outlined for the down-edge, but replace 'down' by 'up' and vice versa.

- Mark all not-frozen edges of the selected cycle as temp-frozen.

3.3 Mark all edges of L_i as frozen.

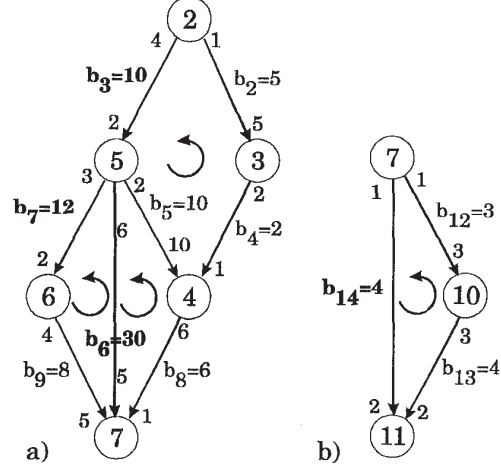


Figure 9. Results of applying Step 3 on the two clusters of Figure 8.

Figure 9 indicates the final buffer lengths of the two resulting clusters of Figure 8 as they have been computed using Step 3. All buffers that needed expansion above the value computed using Equation $\textcircled{1}$ are shown in bold. Cluster a) has two levels; L_1 consists of the primary cycles (5-6-7) and (5-7-4); L_2 consists of the primary cycle (2-5-4-3). Starting with L_1 , we build G_1 from the nodes n_4, n_5, n_6, n_7 and the edges b_5, b_6, b_7, b_8, b_9 . We select the cycle (5-6-7). Checking down-edge expansion, we find $S=1/6$; the only down-edge b_6 hence requires expansion. Equation $\textcircled{7}$ yields $e_6=2$, giving $b_6=10+2=12$. We mark b_6, b_7, b_9 as temp-frozen. We select cycle (5-7-4). Checking down-edge expansion, we find $S=-29/6$; hence no expansion is needed. Checking up-edge expansion, we find $S=17/6$; the only up-edge b_6 hence requires expansion. Equation $\textcircled{7}$ yields $e_6=18$, giving $b_6=12+18=30$. We mark b_7, b_9 as not-frozen and b_5, b_8 as temp-frozen. We select cycle (5-6-7). Checking down-edge expansion, we find $S=-19/6$; hence no expansion is needed. Checking up-edge expansion, we find $S=5/2$; either b_7 or b_9 needs expansion. Equation $\textcircled{7}$ yields $e_7=8$ and $e_9=16$, giving $b_7=4+8=12$. We mark b_7, b_9 as temp-frozen. We mark b_5, b_6, b_7, b_8, b_9 as frozen. For L_2 , G_2 equals the whole cluster. We select cycle (2-5-4-3). Checking down-edge expansion, we find $S=-11/2$; hence no expansion is needed. Checking up-edge expansion, we find $S=17/12$. We compute expansion for the only not-frozen up-edge b_3 : $e_3=6$ giving $b_3=4+6=10$. We mark b_2, b_3, b_4 as temp-frozen. We mark b_2, b_3, b_4 as frozen. Analogously, for cluster b), we find that b_{14} needs expansion to 4. The obtained buffer lengths are the minimum for which a deadlock-free schedule exists.

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 cycles and how their mutual influence may be computed. These algorithms were 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

Rudy Lauwereins is a senior research associate of the NFWO. This project is partly sponsored by the Belgian Interuniversity Pole of Attraction IUAP-50, by the Esprit OMI DIPSAP-II project 8068, by the European Space Agency project on Multi-DSP architectures and by an NFWO Krediet aan Navorsers. K.U.Leuven-ESAT is a member of the DSP Valley™ network.

6. Appendix A. Determine computation levels

While not all edges of the cluster C_i are frozen, do:

1. Choose one primary cycle to start the second step of this algorithm with, in following way: select the smallest primary cycle of which all not-frozen up-edges are shared with a single primary cycle and all not-frozen down-edges are shared with another primary cycle. If no such cycle is found, then select the smallest cycle of which either all up-edges either all down-edges are shared with one or more primary cycles. If still no cycle is selected, then select the smallest primary cycle of the cluster. We say that the selected cycle belongs to the set of cycles A with which we start the second step of the algorithm. All other primary cycles belong to the set Z.
2. In this step, we want to verify whether the buffer length of all cycles of Z can be computed after having fixed the buffer lengths of A, i.e. that all cycles of Z have enough freedom for buffer expansion both in the up- and down-direction. This induces that all cycles of Z must have at least one not-frozen up-edge that is not shared with A and one not-frozen down-edge also not shared with A. If that is the case, all cycles of A form a new

computation level L_i ; we mark its edges as frozen and jump back to the while loop. If at least one cycle of Z is detected that does not have enough freedom, then set A needs to be modified. We call the violating cycle B. If A and B share all their not-frozen edges, add cycle B to set A, since the buffer lengths of B cannot be computed independently of the ones of A. Repeat step 2 with the new set A. If A and B do not share all their not-frozen edges, cycle B replaces all cycles in set A since B has less freedom than A. Repeat step 2 with the new set A.

Mark all edges again as not-frozen.

7. 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", to be published in *IEEE Transactions on Signal Processing*, February 1996.
- 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 Codesign 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, "Hardware/software co-design in GRAPE, including data buffer size minimisation in SDF graphs" (Preliminary title), *Ph.D. Diss.*, KULeuven-ESAT, Oct 1996.
- 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.