Flow Graph Balancing for Minimizing the Required Memory Bandwidth

Sven Wuytack, Francky Catthoor, Gjalt de Jong, Bill Lin, Hugo De Man

Abstract

In this paper we present the problem of flow graph balancing for minimizing the required memory bandwidth. Our goal is to minimize the required memory bandwidth within the given cycle budget by adding ordering constraints to the flow graph. This allows the subsequent memory allocation and assignment tasks to come up with a cheaper memory architecture with less memories and memory ports. The effect of flow graph balancing is shown on an example. We show that it is important to take into account which data is being accessed in parallel, instead of only considering the number of simultaneous memory accesses. This leads to the optimization of a conflict graph.

1. Introduction

Many important applications deal with large amounts of data. This is the case both in multi-dimensional signal processing applications like video and image processing, which handle indexed signals in the context of loops, and in communication network protocols, which handle large sets of records organized in tables. For this type of applications, typically a (very) large part of the area cost is due to memory units. Also the power is heavily dominated by the storage and transfers [4, 7]. Hence, we believe that the dominating factor in the system level design is provided by the organization of the global communication and data storage, and related algorithmic transformations. Therefore we have proposed a design methodology in which the memory architecture is optimized as a first step [14], *before* doing the detailed scheduling, and data-path and controller synthesis. Support for this is under development in our system-level exploration environment ATOMIUM [9].

Flow graph balancing (FGB) is one of the main tasks in our High Level Memory Management (HLMM) script. It is executed prior to the more conventional memory allocation/assignment tasks [1, 6, 12]. The goal of the task is to minimize the required memory bandwidth within the given cycle budget, by adding ordering constraints to the flow graph. This allows the memory allocation/assignment tasks to come up with a memory architecture with a small number of memories and memory ports.

The rest of the paper is organized as follows. Section 2 presents the related work. In Section 3 our flow graph balancing methodology is presented. Section 4 illustrates the proposed methodology on a representative example. Section 5 presents the proposed cost function for optimizing the conflict graph. Section 6 shows preliminary results obtained with a prototype tool. The conclusions are in Section 7.

2. Related Work

Several problems are related to flow graph balancing for minimizing the required memory bandwidth.

First, there is the register allocation domain which is fairly well understood by now. A nice literature overview of this domain can be found in [13]. The techniques used here start from a fully scheduled flow graph and are scalar-oriented. Many of these techniques construct a scalar conflict or compatibility graph and solve the problem using graph coloring or clique partitioning. This conflict graph is fully determined by the schedule. This means that no effort is spent in trying to come up with an optimal conflict graph.

In the less explored background memory allocation and assignment domain, the current techniques start from a given schedule [6], or perform first a bandwidth estimation step [1] which is a kind of crude ordering that doesn't really optimize the conflict graph either. These techniques have to operate on groups of signals instead of on scalars to keep the complexity acceptable, e.g. the *stream model* of Phideo [6] or the *basic sets* in the ATOMIUM environment [1].

In the scheduling domain, the techniques optimizing for the number of resources given the cycle budget are of interest to us. Also here most techniques operate on the scalar level, e.g. [10, 15]. The only exceptions currently are the Phideo stream scheduler [16] and the Notre-Dame rotation scheduler [11]. Many of these techniques try to reduce the memory related cost by estimating the required number of registers for a given schedule. Only few of them try to reduce the required memory bandwidth, which they do by minimizing the *number* of simultaneous memory accesses [15, 16]. They do not take into account *which* data is being accessed simultaneously. Also no real effort is spent to

optimize the memory access conflict graphs such that subsequent register/memory allocation tasks can do a better job.

The main difference between our flow graph balancing and the related work discussed here is that we try to minimize the required memory bandwidth in advance by optimizing the access conflict graph for groups of scalars within a given cycle budget. We do this by putting ordering constraints on the flow graph, taking into account *which* memory accesses are being put in parallel (i.e., will show up as a conflict in the access conflict graph). Also, our techniques operate on groups of scalars instead of on individual scalars.

3. Flow Graph Balancing

Conflict graphs are crucial to flow graph balancing. They are well known from register [13] and other assignment problems. However, since we operate on groups of scalars, we consider conflict graphs where the nodes correspond to basic groups (cfr. Subsection 3.1). When two basic groups are in conflict this means that they have to be assigned either to two different memories, or to a memory with at least two ports during the memory assignment phase.

The more conflicts there are between basic groups, the less freedom there is for the memory allocation/assignment tasks. Experiments have shown that this typically results in a higher cost of the memory architecture. Therefore, we will define a cost function for conflict graphs (cfr. Section 5) reflecting this. The idea of flow graph balancing is then to come up with a conflict graph with minimal cost such that it is still possible later on (after memory management) to schedule the flow graph within the cycle budget.

When multi-port memories are allowed in the memory architecture, it becomes useful to extend the conflict graph with more information to decide on memory types.

This annotation includes the type of conflicts that can occur. More specifically, one has to know for every conflict the maximum number of simultaneous read, write, and total number of memory accesses (i.e., read and write) that can occur. This information allows to decide which type of ports (Read, Write, or Read-Write) are needed on the multi-port memories when certain basic groups are assigned to it.

Secondly, when more than two memory accesses are scheduled in the same time slot, this results in a conflict between more than two basic groups. This type of conflict can be represented in the conflict graph by hyper edges, i.e., edges between more than two nodes.

Finally, it is also possible that a basic group is accessed several times in the same time slot, which results in a self conflict, represented by a self-loop on the corresponding node. Such a conflict forces a multi-port memory for that basic group.

All these extensions lead to the definition of the ECG which is the main output of FGB (cfr. Subsection 3.4).

3.1. Signal Sets: Basic Groups

The memory assignment task should assign groups of scalars to background memories to deal with realistic applications. We call these groups of scalars *basic groups* (BG). They form a partitioning of all data that has to be stored in background memory. This partitioning is decided earlier in our script, and is done in such a way that for every memory access (read or write) in the flow graph it is known which basic group is being accessed. In the case of multi-dimensional signal processing applications, the basic groups are (parts of) multi-dimensional arrays (cfr. [1]). The flow graph balancing and memory allocation tasks operate on the same basic groups as the memory assignment task [1, 3].

3.2. Input: Flow graph on the BG level

The main input of FGB is a flow graph at the BG level. In this flow graph there are two types of nodes: primitive nodes representing a single memory access to a given BG, and nodes that contain a subgraph that has to be repeated a number of times. These nodes have attributes, such as the number of times the node has to be repeated in time, and the period of the repetition expressed as a number of cycles.

3.3. Flow Graph Balancing Script

This subsection summarizes the proposed flow graph balancing script. The input of FGB consists of a flow graph at the BG level and a characterization of all the basic groups, which it gets from the data flow analysis task, as well as the cycle budget in which the flow graph has to be scheduled. The different tasks in our FGB script are:

1. **Initial IO-profile assignment**
   During memory assignment every BG will be assigned to a memory. Every memory has its characteristics on when the inputs have to be provided and when the outputs will be ready (i.e., latency of the memory). This information is formalized in a memory IO-profile. Because the IO-profile plays an important role during the ordering of the memory accesses (especially the latencies), it has to be taken into account as soon as possible. Therefore we assign an IO-profile to every BG at the beginning of FGB. Later in the script this assignment will be reconsidered.

2. **Pre-balancing for high-level in-place mapping**
   When part of the data is only temporary needed and can be overwritten as soon as it has been consumed for the last time, the order in which the different BGs are produced has a large effect on the required storage space due to in-place mapping considerations (cfr. [14]). This step inserts sequence edges in the flow graph at nodes

where this leads to a large reduction in the required number of storage locations.

3. **Conflict cost calculation**
   Because the BGs have different characteristics, some of them fit better together in a common memory than others. This leads to the introduction of *pairwise basic group conflict costs* between every two basic groups in the application (cfr. Section 5), such that during Conflict Directed Ordering (next step in the script) cheaper conflicts can be preferred over more costly ones.

4. **Conflict Directed Ordering (CDO)**
   This is the main step of our script. During CDO it is decided which memory accesses will be put in parallel. This is done globally and in such a way that the resulting conflict graph is as cheap as possible. The decisions are based on the conflict costs and probability that certain conflicts will occur.

5. **IO-profile assignment**
   This task examines the freedom left in the ordering of the memory accesses after Conflict Directed Ordering and uses it to optimize the latency values of the IO-profiles assigned to the different BGs.

6. **Extended conflict graph (ECG) construction**
   As a final step, the ECG is constructed. It is the primary input for the next tasks in our HLMM script [9]: memory allocation and assignment [1, 3].

The output of the FGB task are the ECG, the flow graph with the sequence edges added during pre-balancing for in-place, and the IO-profiles of the different basic groups.

3.4. Output: Extended Conflict Graph

The ECG represents the minimum amount of access conflicts between basic groups, such that it is still possible to find a schedule later on that fits within the cycle budget:

Definition: *An extended conflict graph (ECG)* $G(V, S, E, H)$ *is an undirected hyper graph, where the nodes (*V*) represent basic groups, and the self-edges (*S*), binary edges (*E*), and hyper edges (*H*) represent access conflicts between the basic groups. Every edge* $t \in S \cup E \cup H$ *is labeled with three numbers called* R_t*,* W_t*, and* RW_t*. Where* R_t*,* W_t*, and* RW_t *are respectively: the maximum number of simultaneous read operations, the maximum number of simultaneous write operations, and the maximum number of simultaneous memory accesses (i.e., read and write operations) that can occur for the given conflict during the execution of the algorithm.*

With the ECG as input, the memory allocation and assignment tasks can come up with a memory architecture that is optimal in terms of area and/or power cost. When all constraints contained in the ECG have been respected by these tasks, there will be enough memory bandwidth available to schedule the application within the specified cycle budget.

4. Illustration of the Proposed Methodology

In this section, the effect of flow graph balancing on the outcome of the memory allocation/assignment tasks is illustrated with an example. Several partial orderings will be proposed to illustrate the effect of flow graph balancing on the area cost of the memory architecture (after the memory allocation/assignment tasks). Area figures are given for a 1.2 μm CMOS technology. The values for the single port memories were obtained using an SRAM generator.The values for multi-port memories were obtained by adjusting the values for single port memories with the port dependent factor of Mulder's area model for on-chip memories [8]. The power cost is not taken into account in this example. The code and basic group information of the example is shown in Figure 1. The cycle budget is 550 cycles.

```
for (i = 1 to 50)
  tmp = input;
  A[2*i - 1] = tmp;
  A[2*i] = -tmp;
for (j = 1 to 50)
  B[j] = A[101 - j];
  G[j] = B[j] + A[2*j];
for (k = 1 to 50)
  C[2*k] = B[51 - k];
  C[2*k - 1] = B[k];
for (l = 1 to 50)
  D[l] = C[l] + A[l] + B[l];
for (m = 1 to 150)
  E[m] = m^2;
for (n = 1 to 50)
  F[n] = C[n] + C[2*n] + E[3*n];
```

```
A[]: 100 words of  8 bit
B[]:  50 words of 24 bit
C[]: 100 words of  8 bit
D[]: 100 words of 24 bit
E[]  150 words of 24 bit
F[]:  50 words of 24 bit
G[]:  50 words of  8 bit

Cycle budget = 550 cycles
```

Figure 1.

Every array in the code corresponds to one basic group. It is assumed that all produced basic groups are still needed at the end of the code fragment, such that no in-place optimization [9] can be performed. Notice that in the different stages we show structured code, whereas the technique really works on a (hierarchical) flow graph that corresponds to this code. We believe, however, that it is more clear to show the result of the FGB on the code rather than in flow graph format. One problem with this representation is that one could think that after FGB the code is fully ordered (or scheduled). This is not true. Only the constraints contained in the ECG are imposed. Every schedule that is compatible with it is still allowed. This means that FGB only

introduces a partial ordering, not a full ordering or schedule. After memory allocation and assignment it is possible that basic groups that were originally not in conflict are stored in different memories or even in a multi-port memory. In this case, these two basic groups can be accessed in parallel during the scheduling phase. So, after memory allocation/assignment there is even more freedom left for the scheduler, because it can come up with any schedule that is compatible with the memory architecture and basic group to memory assignment. In the figures depicting the ECGs, the hyper edges are represented in dashed lines, and the edge's R_t, W_t, and RW_t values are represented in the format: $R_t/W_t/RW_t$.

4.1. Partial Ordering 1

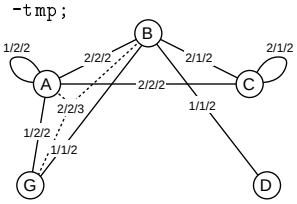
As a first experiment, we have maintained the original procedural execution order as specified in the algorithm. As it is often believed that simply balancing the number of simultaneous read/write leads to good results [15], we have given more cycles to the expressions that have to read many operands such that these memory accesses can be distributed over several time slots. In order to do this within the cycle budget, we had to reduce the number of cycles available to other expressions. For the m loop, this was done by unrolling it with a factor of two, such that two expressions can be scheduled in parallel. The resulting code and corresponding ECG is shown in Figure 2. The array signals are accesses to background memory. The *tmp* variables are registers (and therefore not in the ECG). The ECG can be obtained from the code by assuming that each line in the code has to be executed in one clock cycle. Remark that the number of simultaneous memory accesses is fairly well balanced.

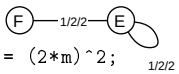
A good memory architecture that is compatible with this ECG is also given in Figure 2. The area cost is 25.529 mm^2. It contains two 2-port memories, which consume a large part of the area. The 2-port memories are a direct consequence of the two self loops in the ECG. Note that these memories could be combined into one 2-port memory, as can be seen in the ECG because there is no hyper edge between basic groups A[], C[], and E[], which means that they can be stored together in a 2-port memory. However, this would waste a lot of bits because E[] has a much larger bitwidth than the other two basic groups. It is clear that self loops in the ECG are costly and have to be avoided whenever possible.

4.2. Partial Ordering 2

In this experiment, we have derived an ECG directly from the procedural execution order of the original code (Figure 1), by assuming that every line has to be executed in one clock cycle. We did this to show that it is not necessarily

```
for (i = 1 to 50)
  tmp = input;
  A[2*i - 1] = tmp; A[2*i] = -tmp;
for (j = 1 to 50)
  B[j] = A[101 - j];
  G[j] = B[j] + A[2*j];
for (k = 1 to 50)
  C[2*k] = B[51 - k];
  C[2*k - 1] = B[k];
for (l = 1 to 50)
  tmp = C[l] + A[l];
  D[l] = tmp + B[l];
for (m = 1 to 75)
  E[2*m - 1] = (2*m - 1)^2; E[2*m] = (2*m)^2;
for (n = 1 to 50)
  tmp = C[n] + C[2*n];
  F[n] = tmp + E[3*n];
```





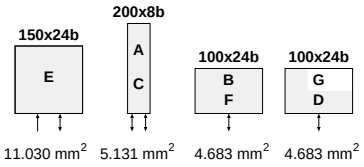


Figure 2. Partial ordering 1.

bad to have a large unbalance in the number of simultaneous read/write operations. The resulting ECG and memory allocation/assignment scheme are shown in Figure 3. Remark that this time there are less self loops in the ECG. As a result the total area cost is now reduced to 17.153 mm^2, despite the unbalance.

The reason is that it is also very important *which* memory accesses are put in parallel, something that has not been incorporated in previous work.

4.3. Partial Ordering 3

In the third experiment, we try to avoid the self loops in the ECG and at the same time reduce the maximum number of simultaneous memory accesses. We did this by applying some optimizing loop transformations [9]. More specifically, we have applied loop splitting (loop m), body splitting (loop j), body merge (loop i and part of loop m), loop reordering (part of loop m and part of loop j), and loop body reordering (loop n) to obtain an execution order shown in Figure 4.

The area cost is 16.292 mm^2, about 36% less than the first partial ordering. The architecture contains many (1-port) memories. This is caused by the size of the maximum clique[1] in the conflict graph (i.e., the graph obtained

[1] A *complete subgraph* is a subgraph in which every node is connected to every other node in the subgraph. A *clique* is a complete subgraph that is not contained in any other (larger) complete subgraph. A clique is a *maximum clique* of a graph when there is no other clique in the graph that contains more nodes.

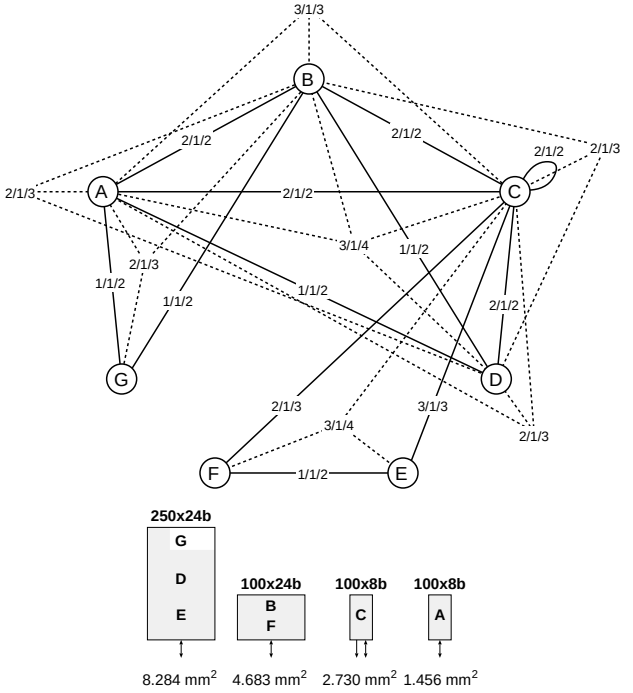
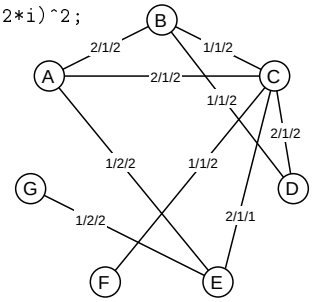


Figure 3. Partial ordering 2.

```
for (i = 1 to 50)
  tmp = input;
  A[2*i - 1] = tmp; E[2*i - 1] = (2*i - 1)^2;
  A[2*i] = -tmp; E[2*i] = (2*i)^2;
for (j = 1 to 50)
  B[j] = A[101 - j];
for (k = 1 to 50)
  C[2*k] = B[51 - k];
  C[2*k - 1] = B[k];
for (l = 1 to 50)
  tmp = C[l] + A[l];
  D[l] = tmp + B[l];
for (m = 1 to 50)
  tmp = B[m] + A[2*m];
  G[m] = tmp; E[m + 100] = (m + 100)^2;
for (n = 1 to 50)
  tmp1 = C[n]; tmp2 = E[3*n];
  F[n] = tmp1 + C[2*n] + tmp2;
```



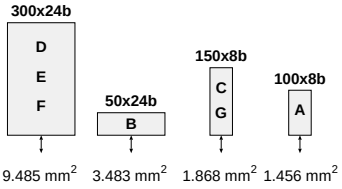


Figure 4. Partial ordering 3.

by removing the hyper and self edges from the ECG). The size of the maximum clique of this graph ($\{A, B, C\}$ and $\{B, C, D\}$) is 3, which means that we need at least 3 1-port memories in a memory architecture that consists entirely out of 1-port memories. However, because of the difference in bitwidth it turned out to be more cost effective to introduce a fourth memory in this case.

4.4. Partial Ordering 4

In this final experiment, we took the ordering of the previous experiment, with a little change in the ordering of the body in the loop producing D[]. The result is that the size of the maximum clique now becomes 2 instead of 3. This means that less memories are needed in the memory architecture. The results are shown in Figure 5. The area cost is 13.379 mm^2. This is a much better result than the one obtained in the first experiment: the area is about a factor 2 less; it contains only 2 memories instead of 4; and the total number of memory ports is 2 compared to 6, which means less interconnect.

This once more shows that it is very important to take into account *which* memory accesses are done in parallel. Even small changes can have large consequences. This makes it particularly difficult to optimize this by hand for large real life applications.

5. Cost Function

To leave as much freedom as possible for the memory allocation/assignment tasks, it is important to come up with an ECG with as few conflicts as possible. Not all conflicts are equally costly, though. For instance, when two basic groups have a large difference in bitwidth, it is not that bad that they have to be stored in different memories, because this saves bits that would otherwise be wasted. This means that such a conflict has to be prefered compared to a conflict between basic groups with equal bitwidth. This justifies the introduction of *pairwise basic group conflict costs* C_e corresponding to the binary edges $e \in E$ of an ECG $G(V, S, E, H)$. These pairwise conflict costs are calculated based on the properties of the two basic groups involved. This is an important difference with the scalar oriented techniques, where all scalars are considered to be more or less equal.

We propose the following cost function for optimizing the extended conflict graph: $Cost(G(V, S, E, H)) = \alpha \cdot \sum_{s \in S} RW_s + \beta \cdot MaxCliqueSize(G(V, E)) + \gamma \cdot \sum_{e \in E} C_e$ The first term penalizes self-loops in the ECG which reduces the number and size of multi-port memories in the final memory architecture, the second term reduces the number of required memories, and the last term minimizes the total weighted conflict cost of the extended conflict graph. The hyper edges are not included in the cost function because FGB comes before memory allocation/assignment. There-

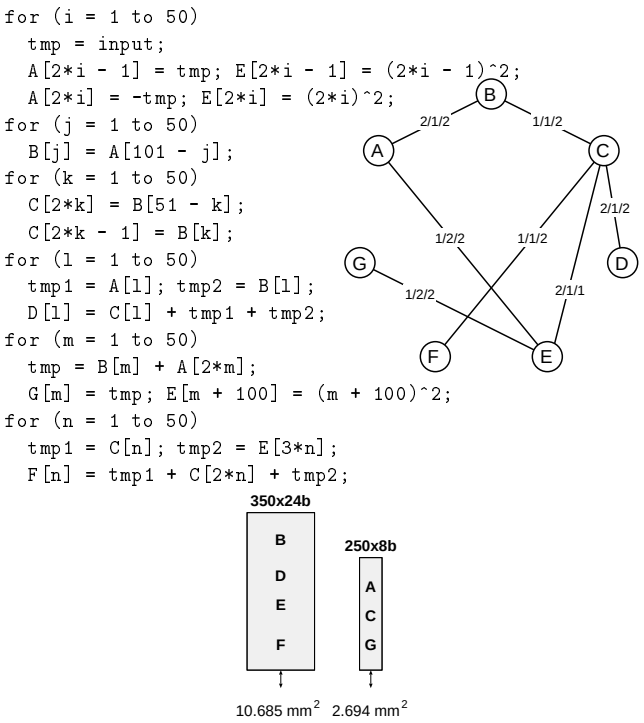


Figure 5. Partial ordering 4.

fore it is not known at this stage whether a conflict will be resolved by assigning the conflicting basic groups to different memories or not. Only when they are assigned to a multiport memory, the R_t, W_t, and RW_t values for $t \in E \cup H$ come into play. They contain vital information for the memory allocation and assignment tasks, though.

6. Results

Experiments with a prototype tool (that cannot yet handle loops) on a realistic ATM application [5] with 14 basic groups and 38 read/write operations showed some promising results. For a cycle budget of 17, the conflict graph obtained with our FGB technique contained 13 conflicts and had a maximum clique size of 3, which resulted in an optimal memory allocation of 3 one-port memories. The same application scheduled with Synopsys' Behavioral Compiler resulted in a conflict graph with 24 conflicts and a maximum clique of size 5, requiring 5 one-port memories for the same cycle budget.

7. Conclusions

In this paper we have shown that it is important to do a proper flow graph balancing to arrive at a good solution for the memory allocation/assignment problem. We have also shown that this flow graph balancing has to be done at the level of groups of scalars in such a way that the resulting conflict graph is optimized. This means that one has to take into account *which* data is being accessed in parallel, instead of only considering the *number* of parallel memory accesses. Currently, we have a prototype tool that shows already some very promising results, and we are working on the full implementation of the proposed technique.

References

[1] F.Balasa, F.Catthoor, H.De Man, "Dataflow-driven memory allocation for multi-dimensional processing systems", *Proc. IEEE Int. Conf. Comp. Aided Design*, San Jose CA, Nov. 1994.

[2] F.Balasa, F.Catthoor, H.De Man, "Background memory area estimation for multi-dimensional signal processing systems", *IEEE Trans. on VLSI Systems*, vol. 3, no. 2, pp. 157-172, June 1995.

[3] F.Balasa, "Background memory allocation for multi-dimensional signal processing", *Ph.D. dissertation*, IMEC, Leuven, Nov. 1995.

[4] F.Catthoor, W.Geurts, H.De Man, "Loop transformation methodology for fixed-rate video, image and telecom processing applications", *Proc. Intnl. Conf. on Applic.-Spec. Array Processors*, San Francisco, CA, pp.427-438, Aug. 1994.

[5] G.de Jong, B.Lin, C.Verdonck, S.Wuytack, F.Catthoor, "Background Memory Management for Dynamic Data Structures Intensive Processing Systems", *Proc. Int. Conference on CAD*, San Jose, CA, pp.515-520, Nov. 1995.

[6] P.Lippens, J.van Meerbergen, W.Verhaegh, A.van der Werf, "Allocation of Multiport Memories for Hierarchical Data Streams", *Proc. IEEE Int. Conf. Comp.-Aided Design*, pp. 728-735, Santa Clara, Nov. 1993.

[7] T.Meng, B.Gordon, E.Tsern, A.Hung, "Portable video-on-demand in wireless communication", special issue on "Low power design" of the *Proc. of the IEEE*, Vol. 83, No. 4, pp. 659-680, Apr. 1995.

[8] J.M.Mulder, N.T.Quach, M.J.Flynn, "An Area Model for On-Chip Memories and its Application", *IEEE J. Solid-state Circ.*, Vol.26, No.1, pp.98-105, Feb. 1991.

[9] L.Nachtergaele, F.Catthoor, F.Balasa, F.Franssen, E.De Greef, H.Samsom, H.De Man, "Optimization of memory organization and partitioning for decreased size and power in video and image processing systems", *IEEE Int'l Workshop on Memory Technology, Design and Testing*, pp. 82-87, San Jose CA, Aug. 1995.

[10] P.Paulin, J.Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's", *IEEE Trans. on CAD*, Vol. 8, No. 6, pp. 661-679, June 1989.

[11] N.Passos, E.Sha, "Push-up scheduling: optimal polynomial-time resource constrained scheduling for multi-dimensional applications", *Proc. IEEE Int. Conf. Comp. Aided Design*, San Jose CA, pp.588-591, Nov. 1995.

[12] L.Ramachandran, D.Gajski, V.Chaiyakul, "An algorithm for array variable clustering", *Proc. European Design and Test Conf.*, pp. 262-266, Paris, Mar. 1994.

[13] L.Stok, "Data path synthesis", *INTEGRATION, the VLSI journal*, Vol 18, pp. 1-71, June 1994.

[14] I.Verbauwhede, F.Catthoor, J.Vandewalle, H.De Man, "Background memory management for the synthesis of algebraic algorithms on multi-processor DSP chips", *Proc. VLSI'89, Int. Conf. on VLSI*, Munich, Germany, pp. 2098-218, Aug. 1989.

[15] W.Verhaegh, P.Lippens, E.Aarts, J.Korst, J.van Meerbergen, A.van der Werf, "Improved Force-Directed Scheduling in High-Throughput Digital Signal Processing", *IEEE Transactions on CAD and Systems*, Vol. 14, no 8, Aug. 1995.

[16] W.Verhaegh, "Multidimensional Periodic Scheduling", *Ph.D. dissertation*, Eindhoven University of Technology, Oct. 1995.