Fast and extensive system-level memory exploration for ATM applications

Peter Slock, Sven Wuytack, Francky Catthoor, Gjalt de Jong†

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
†Alcatel Telecom, Francis Wellesplein 1, B-2018 Antwerp, Belgium

Abstract

In this paper, our memory architecture exploration methodology and CAD techniques for network protocol applications are presented. Prototype tools have been implemented, and applied on part of an industrial ATM application to show how our novel approach can be used to easily and thoroughly explore the memory organization search space at the system-level. An extended, novel method for signal to memory assignment is proposed which takes into account memory access conflict constraints. The number of conflicts is first optimized by our flow-graph balancing technique. Significant power and area savings were obtained by performing the exploration thoroughly at each of the degrees of freedom in the global search space.

1 Introduction

The complexity of modern telecommunication systems is rapidly increasing. A wide variety of services has to be transported, and elaborate network management is needed. Such complex systems require a combination of hardware and software components in order to deliver the required functionalities at the desired performance level. For certain speed critical functionalities, such as packet handling and congestion control, it is necessary to implement the corresponding components in hardware using custom Application Specific Integrated Circuits. These ASICs are usually at the heart of modern telecommunication systems.

For applications in this domain, the desired behavior is often characterized by complex algorithms that operate on large dynamically allocated stored data structures (e.g. linked list, trees, dynamic FIFOs). This includes especially the transport layer in ATM networks and LAN/WAN technology. Ideally, the specification should reflect the "conceptual" partitioning of the problem, which typically correspond to abstract data types along with services provided on them, and algorithms for the different processing tasks. As these conceptual entities can be readily specified in an object-oriented programming model, using data abstraction and class inheritance features, we use the C++ programming language for the behavioral algorithmic specification as motivated in our global MATISSE approach[5].

For this type of applications, typically a (very) large part of the area cost is due to memory units. Also the power for data-dominated applications is heavily dominated by the storage and transfers [8]. Hence, we believe that a dominating factor in the system level design is provided by the organization of the global communication and data storage. Therefore we have proposed a design methodology in which the storage related issues (virtual and physical memory management) are optimized as a first step [5], *before* doing the detailed scheduling, and data-path and controller synthesis.

Two main tasks in our memory management methodology are flow graph balancing (FGB) and memory allocation/assignment. FGB is executed prior to the more conventional memory allocation/assignment tasks [1, 7, 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 subsequent memory allocation/assignment tasks to come up with a memory architecture with a small number of memories and memory ports. The CAD techniques to support these tasks will be detailed in this paper and their feasibility will be demonstrated by prototype tools.

The extensive design exploration which is feasible by applying our methodology and its heavy impact on storage area and power will be illustrated based on a representative module of an industrial ATM application, provided by Alcatel.

The rest of the paper is organized as follows. Section 2 presents the related work. In Section 3 our physical memory management methodology and the CAD techniques for two key tasks are presented. Section 5 illustrates the extensive exploration which can be obtained with the proposed methodology and the use of our prototype tools on a representative ATM example, introduced in Section 4. Section 6 contains the conclusions.

2 Related Work

Although behavioral hardware synthesis has been an active field of research for over one decade (see e.g. [2]), and commercial behavioral synthesis tools are currently emerging (e.g. Synopsys' Behavioral Compiler, Mentor Graphics' DSPStation, and Synthesia's SYNT system), support for complex data structures and memory synthesis related problems is generally limited. An exception to this is the video and image processing area for which several groups have proposed memory management support [7, 12] including also our own work on the ATOMIUM methodology and partial prototype tool support [9, 17]. This is however restricted to the support of statically declared arrays and records.

The data structures in the communication network protocol domain are however different, as indicated above: they are mostly based on dynamically allocated lists, sets and tables. For this type of applications, partly different methodologies and design tool support are required. This has been motivated by us in our global methodology proposal on the MATISSE approach [5]. At that time, we did not provide any design techniques or tool support yet. Recently, we have also proposed a more detailed methodology for the flow graph balancing step [17] but also in that paper no CAD techniques have been presented yet.

In this paper, we focus on the new techniques and tools in both our flow graph balancing and memory allocation/assignment steps, targeted to the communication network protocol domain. They will be explained in detail in Section 3.

Several problems in the state-of-the-art literature are related to these tasks. 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 [7], or perform first a bandwidth estimation step [1] which is a crude ordering that does not 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 [7] or the *basic sets* in the ATOMIUM approach [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).

For the memory allocation and signal-to-memory assignment subtasks, our previous approach [1] is based on relatively greedy global optimization approaches which did not yet take into account memory access conflict constraints. As a result, the assignment technique could only work when the bandwidth constraints were not too tight and did not lead to conflict violations. In this paper, a novel method for signal-to-memory assignment is proposed which does incorporate a way to handle memory access conflict constraints. To achieve this, the original method had to be modified significantly. It also contains several improvements compared to other existing techniques because it takes into account the (extended) conflict graphs produced by the FG balancing step. Compared to other approaches, it also works on groups of scalars, thereby reducing the complexity significantly.

3 Exploration Environment

In this section our physical memory management script for network applications is explained. The input is a C or C++ description of the application, which is translated into a flow graph. The output is an optimized memory architecture. An overview of our exploration environment is shown in Fig. 1.

Given a flow graph of the application (cfr. upper left corner in Fig. 1), where for every memory access it is indicated which data is being accessed, an optimized memory architecture has to be derived during the allocation and assignment phase. However, because the memory architecture is derived before detailed scheduling, sufficient memory bandwidth must be foreseen such that the application can be scheduled within the given cycle budget afterwards. That is why the memory architecture is not derived directly from the flow graph. First a flow graph balancing step is performed, which tries to minimize the required memory bandwidth.

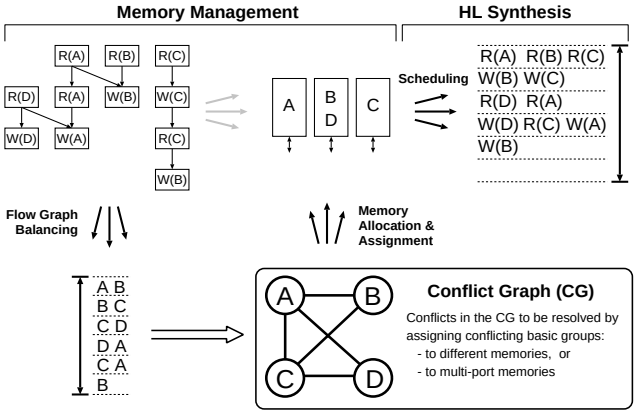

Figure 1: Physical Memory Management

Remark that due to the dynamic and data-dependent nature of the application domain, a number of steps of our High Level Memory Management (HLMM) script for Real-time Multi-dimensional Signal Processing (RMSP) applications [9] are not as useful in this context. In particular, the space reserved for dynamically allocated data of a certain type is assumed to be alive whole the time. This means that no in-place optimization [3, 4] can be done on it. Other very useful optimizations to lower the power consumption of RMSP applications, namely loop transformations to increase the locality of access and the exploitation of Data Reuse (leading to the introduction of memory hierarchy) [6], can also not be used in this context because no manifest loop nests are present and which data will be accessed is data-dependent and thus cannot be predicted.

In the next subsections, each subtask in our HLMM script for network applications will be explained.

3.1 Data Flow Analysis

The first task in the script translates the C/C++ description of the network component application into our flow graph format. It also performs data flow analysis to extract the parallelism from the sequential input description. For every data type in the application a virtual memory segment (VMS) is created. A VMS represents an amount of memory sufficient to store all objects of the corresponding data type. It also provides a dynamic memory management scheme to allocate (and deallocate) part of its memory space to data objects at run-time. For more details on how the dynamic allocated data is handled we refer to [5].

To increase the accuracy of the data flow analysis and to increase the freedom of subsequent memory management tasks, these virtual memory segments are partitioned further into non-overlapping *basic groups*. This is done in such a way that for every memory access, it is known at compile time which basic group is being accessed. For instance, assuming that for every memory access to a record, the field being accessed is known (a reasonable assumption in the application domain considered), all instances of a particular field (belonging to different objects of the same type) can be combined into one basic group. As a consequence the record layout plays an important role in the basic group partitioning. Indeed, fields grouped together in a common memory word, will be part of the same basic group. Currently our c2fg prototype tool is based on GNU's C compiler.

The basic groups are characterized by their bitwidth (extracted form the original specification), their number of words (extracted from the VMS definition), and the average number of memory accesses (derived from simulations). These figures are used for the area and power estimations.

3.2 Flow Graph Balancing for Memory Bandwidth Reduction

The flow graph balancing (FGB) step orders the memory accesses within the given cycle budget. In the context of ATM applications, nested loops almost never occur. The existing (single) loops are usually data-dependent (WHILE loop) spanning an entire task in a concurrent specification. In general, we can handle the memory management for the concurrent tasks separately, though also optimizations are feasible between tasks. The latter is however a topic of further research. So in this paper, we will assume that FGB has to operate on a single task, possibly generated after combining a number of initially concurrent tasks so that they are statically schedulable. Due to the very limited loop oriented characteristics, we can also restrict our current technique and prototype realization to handle a "flat" graph operating on the body of a single loop where typically complex dynamic data structures are accessed in a very complicated condition hierarchy.

Whenever two memory accesses to two basic groups (BGs) within the data structures occur in the same cycle, we say there is an access conflict between these two basic groups. All access conflicts are collected in a conflict graph, where the nodes represent basic groups, and the edges indicate a related conflict. These conflicts have to be resolved during the memory allocation and assignment steps. This can be done by assigning conflicting basic groups either to different memories or to a multiport memory. When all conflicts in the conflict graph are resolved during the memory assignment step, it is guaranteed that a valid schedule exists for the obtained memory architecture.

We have defined a cost function for these conflict graphs, such that more costly conflict graphs are likely to lead to more costly memory architectures. The cost function includes three weighted terms: 1) a measure for the minimum number of memories needed (the chromatic number[1] of the

[1] A *c-coloring* of a graph G is a partitioning of G's nodes in c partition

CG), 2) a term related to the number of conflicts in the conflict graph (each conflict is weighted with its importance), and 3) a term to minimize the number of self-conflicts (based on the notion of forces similar to IFDS [15]). For more details about the cost function we refer to [17]. The idea of flow graph balancing is then to come up with a partial ordering of the memory accesses that leads to a conflict graph with minimal cost.

When multiport memories are allowed in the memory architecture, more information is needed than present in a simple conflict graph. Therefore, we have defined an extended conflict graph (ECG). In an extended conflict graph, every conflict is annotated with the maximal number of simultaneous read accesses, the maximal number of simultaneous write accesses, and the maximal number of simultaneous memory accesses (i.e., read or write) that can occur between the conflicting basic groups during the execution of the algorithm. Also conflicts between more than two basic groups have to be taken into account, because several conflicting basic groups can be stored in a single multiport memory as long as the number of simultaneous memory accesses is not exceeding the access capabilities of the memory. This results in the inclusion of hyper edges in the ECG, indicating the conflicts between more than two basic groups. When multiport memories are available in the memory library, basic groups can be allowed to be in conflict with themselves, leading to self edges in the conflict graph. Obviously, such self conflicts will be very costly because they force the allocation of (expensive) multiport memories. Fig. 2 illustrates the difference between a conflict graph and an extended conflict graph for a given memory access ordering.

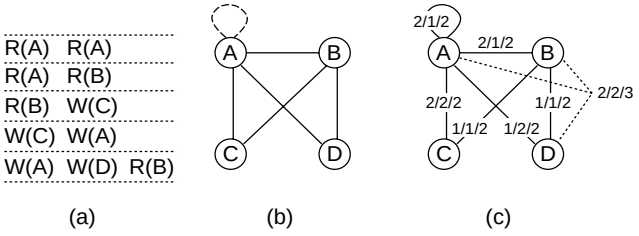

Figure 2: Extended Conflict Graph: (a) Schedule, (b) Conflict Graph, (c) Extended Conflict Graph. The R/W/RW numbers indicate the maximum number of simultaneous read operations (R), write operations (W), and read or write operations (RW) that occur for the given conflict.

Our current flow graph balancing tool uses an iterative search strategy to order the memory accesses similar to Improved Force Directed Scheduling of [15] but with totally different cost functions. As motivated above, no loops are supported as yet. That is a topic of current research. The prototype tool implements the principles introduced in [17].

3.3 Allocation and Assignment

The memory assignment technique discussed below is significantly different compared to existing techniques (including our own previous work [1]) because it has to take into account the (extended) conflict graphs produced by the FG balancing step. Compared to most other approaches, it also works on groups of scalars, thereby reducing the complexity significantly.

Once the (extended) conflict graph is available, we have all the inputs required to do a *valid and cost efficient* allocation and assignment. This is done by minimizing a cost function, containing weighted memory area and power terms, while taking into account all memory access constraints expressed by the conflict graph. The on-chip area and power models used in the cost function are proprietary from memory module vendors, so only relative figures are given in this paper.

During the *allocation* phase, the user decides on the *number of one-port memories* to be allocated. This should be at least the chromatic number of the conflict graph. Remark that this lower bound on the number of memories only holds for a library with one-port memories, which is always assumed further on in this paper[2]. The main reason for allocating more than the minimal number of memories is to reduce the power, as demonstrated further on. The practical upper bound on this number is the number of basic groups (cfr. Section 3.1).

During the *assignment* phase, each basic group is assigned *as a whole* to one of the allocated memories. This yields an assignment scheme. Usually a BG can only be assigned to *some* of the allocated memories, because of memory access conflicts with some of the basic groups assigned already. In order to find a minimum cost assignment scheme, we must explore the entire assignment search space, because classical global optimization approaches like (Mixed) Integer Linear Programming solvers do not work for our non-linear problem. The assignment search space can be represented as a tree, as shown in Fig. 3. This tree has M^N leaves, where M denotes the number of memories allocated and N the number of basic groups to be assigned, so there is a huge amount of assignment possibilities. We use a *branch-and-bound* algorithm (called *B&B* in the sequel) with an effective bounding to search the complete assignment tree.

In our specific context, we have implemented a very effective B&B strategy which is different from our original approach [1] which did not incorporate the access conflicts.

classes $V = X_1 + X_2 + \cdots + X_c$ such that every two adjacent nodes belong to a different partition class. In this case, when the members of partition X_i are colored with color i, adjacent nodes will receive different colors. The chromatic number $\chi(G)$ is the smallest number c for which there exists a c-coloring of G.

[2] We also have a methodology for dealing with multi-port memories, but that feature is not implemented yet in our prototype software environment.

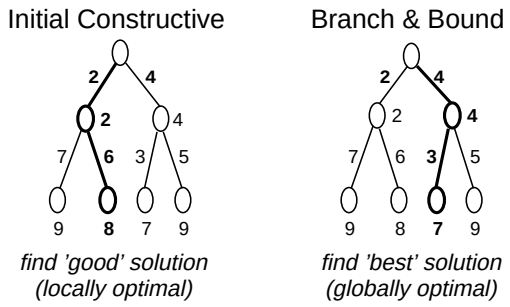

Figure 3: Different assignment algorithms for traversing the search tree: each tree level corresponds to the assignment of one basic group, each branch from a node corresponds to one memory.

Large parts of the tree can be cut away ('bound', i.e., pruning of subtrees) because of reasons, evaluated in this order:

1. paths which give rise to assignment schemes which are fully symmetric with already generated schemes can be discarded.

2. access conflicts between basic groups can be effectively checked due to the explicit ECG information, which allows to remove the corresponding sub trees

3. paths which have too high a cost from a certain basic group on can be pruned from that node. Currently we propose a simple but safe estimate of the minimum remaining cost which assumes storage in a common 1-port memory unit with minimum bitwidth for area and storage in separate 1-port memory units for power, and this for all remaining BGs.

Moreover, in order to have a low cost threshold (for pruning) at an early stage in the B&B process, we first perform an initial constructive assignment algorithm. This finds a local optimum in the search tree by iteratively assigning at each level in the search tree the corresponding basic group to the locally cheapest memory.

The order in which the basic groups are assigned also has an enormous effect on the run-time of the B&B algorithm. Experiments have shown that in most cases, ordering based on cost (i.e., assigning Basic Groups in the order from costly to cheap in terms of area/power consumption) leads to shorter run-times than ordering based on constraints (i.e., assigning BGs in the order from difficult-to-assign to easy-to-assign).

All the pruning criteria mentioned above have been implemented in our prototype tool. As a result, for the not very complex but still realistic example in Section 5, the CPU times can all be reduced below 1 minute on an HP 9000/715-50 workstation. Also for larger examples, experiments have confirmed that the run times remain very acceptable. Other performance improving measures have been developed by us and will be implemented in the future, such as splitting of the conflict graph into *(near-)independent clusters*, and additional cost pruning by *looking ahead* in the search tree.

4 Test Vehicle: STORM

As a test vehicle for our exploration experiments, we have selected part of the functional specification of an ASIC specified by Alcatel. The application is called STORM, which stands for ***S**TM1-level to **T**ransputer-netw**O**rk **R**elayer with **M**ultiple protocol handling*. It deals with protocol conversion between an ATM transportation layer and a transputer network (cfr. Fig. 4).

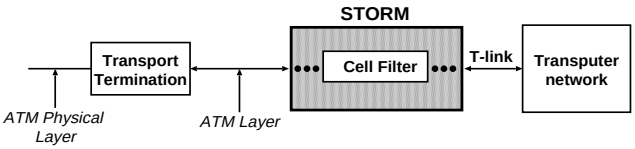

Figure 4: Position of STORM in ATM context

In this paper, we only focus on part of the STORM application, namely the cell filter. This part is representative, because, like the other subsystems, it is rather heavily IO-dominated: many background memory accesses have to be performed real-time within a limited number of cycles. It relates incoming ATM cells to allocated internal streams, by means of two search algorithms (cfr. Fig. 5), which can be combined in a pipelined way, i.e., they can be executed in parallel but acting on subsequent ATM cells.

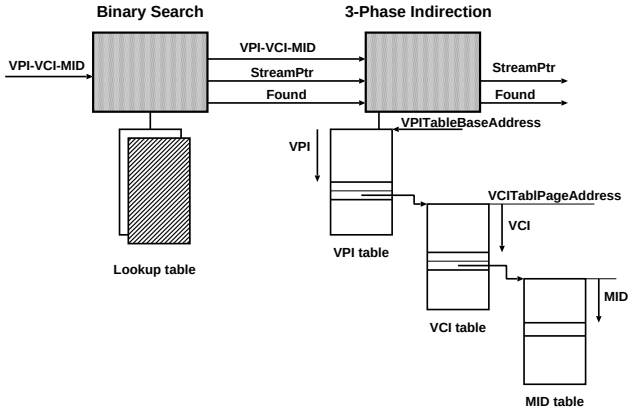

Figure 5: Cell filter

Both search algorithms, binary search and 3-phase indirection, are looking for the same information but in a different way. The first algorithm uses the combination of some information fields of an ATM cell as one key into one big

sorted table, stored in a *LookupRAM*. If the stream is not found in this table, the second algorithm looks for the stream in 3 steps. At each step, one of the subkeys (VPI, VCI, and MID extracted from the incoming ATM cell) is used as an index in the corresponding table, to find a pointer to the next table. The last table contains a pointer to the related stream.

5 Design Space Exploration: Results

In this section we show the results of our design space exploration experiments for the cell filter of the STORM application, by varying some of the design parameters. A similar exploration has also been done for the Segment Protocol Processor application from Alcatel [14], but due to space requirements these results are not presented in this paper.

5.1 Effect of Modified Record Layout

In network type applications, much of the data is stored in records. A record is a data structure that groups a number of related data items. These different data items are called the fields of the record. These fields can have different bitwidths. Several fields can be packed into a common word. How the different fields are packed into words is called the record layout.

The record layout heavily influences the optimal memory architecture, as it effects the number of memory accesses and the basic group partitioning. Here we present two extreme cases of record layout. In the first one, each field is stored in a separate memory word. The results are shown in Fig. 6. The access flow graph shows many memory accesses, because every field access requires its own memory access. There are many basic groups as can be seen in the conflict graph. The chromatic number of the conflict graph for a cycle budget of 6 cycles is 4. This means that in an architecture with only single port memories, at least 4 memory modules are needed to provide sufficient memory bandwidth.

A second case of record layout we have examined, is one where all fields of a given record are stored in a common memory word. The results for this are shown in Fig. 7. Now there are far less memory accesses, as can be seen in the access flow graph. Also the conflict graph is much simpler. In fact, in this case, there are no conflicts left in the graph which means that 1 memory is sufficient to meet the bandwidth requirements for a cycle budget of 6 cycles.

In between these two extreme cases, many other record layouts are possible. E.g., grouping the fields of a record such that they fit as good as possible into 32 bit words. This corresponds to the row labeled *32 bit* in Table 1.

The table entries contain relative area and power numbers for the total memory architecture resulting from allocation and assignment. The orders of magnitude are tens of mm^2

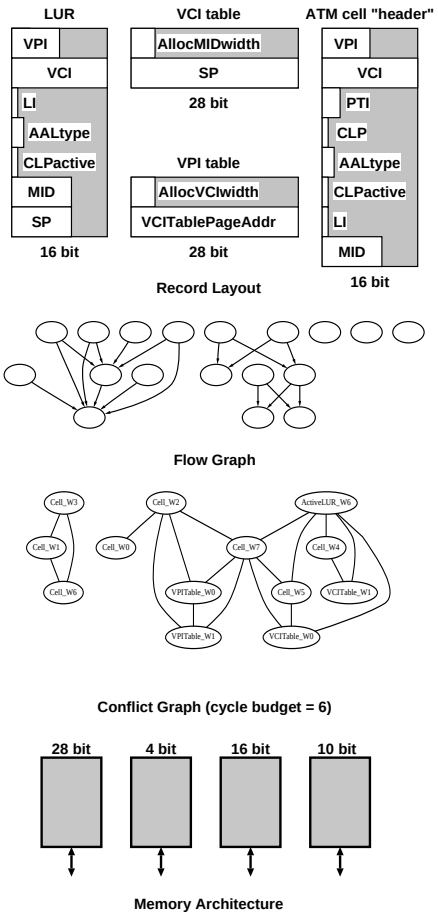

Figure 6: Record layout: 1 field per word

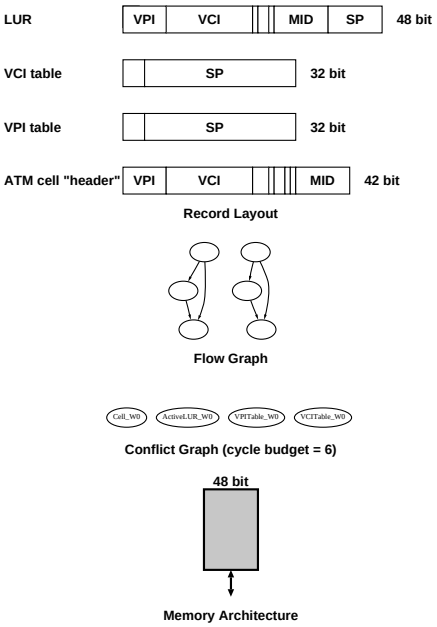

Figure 7: Record layout: 1 word per record

for area and a hundred mW for power. The area numbers are relative to the maximum area number of the table, whereas the power numbers are relative to the maximum power number.

	1 mem	2 mem	3 mem	4 mem
1 field 1 word	N/A	N/A	N/A	A: 1.00 P: 1.00
32 bit	N/A	A: 0.87 P: 0.96	A: 0.90 P: 0.60	A: 0.91 P: 0.50
1 word	A: 0.84 P: 0.92	A: 0.81 P: 0.64	A: 0.83 P: 0.39	A: 0.84 P: 0.34

Table 1: Optimal memory architecture for different record layouts and cycle budget = 6.

From this table we can conclude that, for this example, the area and power decrease when the records are packed into less words. Also fewer memories need to be allocated in case the records are packed into fewer words (the *N/A* entries denote that no valid assignment is possible for the corresponding number of allocated memories). Note also that the power decreases when more memories are allocated, while the area exhibits a minimum between the two extremes.

As indicated in subsection 3.1, it is also important to perform a partitioning of the original VMS data structures into basic groups. In this particular test-vehicle, this preprocessing results e.g. in a decrease of the number of self-conflicts (and hence minimal number of memory ports) from 4 to 2 for a cycle budget of 3 and the original record layout with 1 field stored per word.

5.2 Effect of the Cycle Budget

To see the effect of the cycle budget on the resulting memory architecture, we have varied the cycle budget from the critical path length (3 in the case of one field per word) to the number of memory accesses in the flow graph of the application (18 in this case). Assuming that every memory access requires exactly one cycle[3], the minimal bandwidth is reached (i.e., 1 memory port) for this amount of cycles. Allocating more cycles will have no further effect on the memory architecture. The resulting chromatic number is shown for each cycle budget in Fig. 8. Remark that in general this value decreases as more cycles are available. However, this is not always the case (e.g., cycle budget = 11) because during flow graph balancing also other cost terms than the chromatic number are taken into account (e.g., the number of conflicts, each weighed with their cost). Ideally the total cost should be decreasing monotonically (as is the case in Fig. 8). This is not guaranteed, however, with our FGB tool as it is a heuristic that does not necessarily find the global optimum. The CPU times for these experiments are all below 20 seconds on an HP 9000/715-50 workstation.

[3]This is usually the case for SRAM memories. If different types of RAMs are used, appropriate IO profiles (with larger latency values for instance) have to be used.

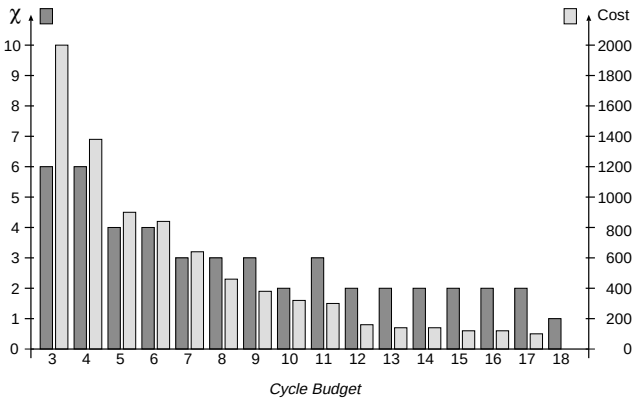

Figure 8: Chromatic number and FGB cost function for different cycle budgets.

5.3 Effect of Allocation/Assignment

To see the effect of the *number of allocated memory modules*, we have varied that number for a cycle budget equal to 6 and a record layout with 1 field per word. The results are shown in Table 2.

The minimal required number of 1 port memories is 4 (i.e., the chromatic number of the conflict graph for this cycle budget). The maximal number of memories equals the number of basic groups, because basic groups are assigned as a whole to a single memory. The values in the table were optimized both for area and power with equal weight factors. As can be seen in the table, the area decreases with increasing number of memories, up to a certain number of memories. The reason for this is that, because of the possibly different bit widths of the allocated memories, less bits have to be wasted when more memories are available in the memory architecture. Therefore basic groups with different bitwidths have a larger chance to be assigned to different memories, which will then have a bitwidth adapted to them.

However, from a certain number of memories on, the area starts increasing again with increasing number of memories, because more memories involves more overhead. The power keeps decreasing when more memories are allocated. Remark also that the optimal solution (w.r.t. weighted area/power cost) for these cost parameters is an architecture with 11 memories. Therefore, even when more than 11 memories are allocated, only 11 of them will be effectively used. Note, that the cost of interconnect is not yet taken into account in our cost functions. When this is done, the optimal

values will probably occur for a smaller number of memories.

#Mem	Area (rel)	Power (rel)	Weighted Cost
4	1.00	1.00	1.00
5	0.93	0.76	0.87
6	0.92	0.70	0.85
7	0.92	0.63	0.82
8	0.94	0.54	0.80
9	0.94	0.50	0.79
10	0.95	0.48	0.79
11	0.95	0.47	0.78
12	0.95	0.47	0.78
13	0.95	0.47	0.78

Table 2: Optimal cost results for different 1-port memory allocations for a cycle budget of 6.

The resulting basic group to memory assignment depends upon the exploration of the assignment search space, i.e., on the assignment algorithm used. We mainly used the *initial constructive* and the *branch-and-bound* assignment algorithms, as shown in Fig. 3. Experiments with both alternatives revealed for instance that, for the parameters mentioned above and 5 memories allocated, area and power for initial constructive are 10% resp. 30% higher than the area and power obtained with B&B. This shows that performing an extensive assignment search space exploration, as is done by the B&B algorithm, can reduce the area to some extent, and can have a significant effect on the power consumption of the final solution.

6 Conclusions

We have shown that significant better results than conventional design approaches can be obtained for the memory organization of network protocol applications when the full search space is well explored. The methodology and prototype tools presented in this paper allow for the first time to do this in a fast and thorough way starting from a high abstraction level. The run-times of our tools are all below 1 minute on an HP 9000/715-50 workstation for the application presented. In this way the system design time for these applications is significantly reduced while the final power and area cost are improved at the same time. This is of crucial importance for cost effective designs in the telecom network protocol industry.

Acknowledgments: We gratefully acknowledge the discussions with our colleagues at IMEC and Alcatel and especially the contributions of M.Miranda, M.Genoe, M.Eyckmans, and P.Six.

References

[1] 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.

[2] R.Camposano, W.Wolf (eds.), "Trends in high-level synthesis", Kluwer Academic Publishers, Boston, 1991.

[3] E.De Greef, F.Catthoor, H.De Man, "Memory Size Reduction through Storage Order Optimization for Embedded Parallel Multimedia Applications", *Intnl. Parallel Proc. Symp.(IPPS) in Proc. Wsh on "Parallel Processing and Multimedia"*, Geneva, Switzerland, April 1997.

[4] E.De Greef, F.Catthoor, H.De Man, "Array Placement for Storage Size Reduction in Embedded Multimedia Systems", *Intnl. Conf. on Applic.-Spec. Array Processors (ASAP)*, Zurich, Switzerland, July 1997.

[5] G.de Jong, B.Lin, C.Verdonck, S.Wuytack, F.Catthoor, "Background memory management for dynamic data structure intensive processing systems", *Proc. IEEE Int. Conf. Comp. Aided Design*, San Jose CA, pp.515-520, Nov. 1995.

[6] J.P.Diguet, S.Wuytack, F.Catthoor, H.De Man, "Formalized Methodology for Data Reuse Exploration in Hierarchical Memory Mappings", accepted for *Proceedings of IEEE International Symposium on Low Power Electronics and Design*, Monterey CA, Aug. 1997.

[7] 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.

[8] 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.

[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] Y.Therasse, G.H.Petit, M.Delvaux, "VLSI architecture of a SMDS/ATM router", *Annales des Télécommunications*, 48, no 3-4, pp.166-180, 1993.

[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, P.Lippens, E.Aarts, J.van Meerbergen, A.van der Werf, "Multi-dimensional periodic scheduling: model and complexity", *Proc. EuroPar Conference*, Lyon, France, August 1996. "Lecture notes in computer science" series, Springer Verlag, pp.226-235, 1996.

[17] S.Wuytack, F.Catthoor, G.de Jong, B.Lin, H.De Man, "Flow Graph Balancing for Minimizing the Required Memory Bandwidth", *Proc. 9th ACM/IEEE Intnl. Symp. on System-Level Synthesis*, La Jolla CA, pp.127-132, Nov. 1996.