

# An Optimization Methodology for Memory Allocation and Task Scheduling in SoCs via Linear Programming

Bastian Ristau and Gerhard Fettweis

TU Dresden, Vodafone Chair Mobile Communications Systems  
01062 Dresden, Germany  
{ristau, fettweis}@ifn.et.tu-dresden.de

**Abstract.** Applications for system on chips become more and more complex. Also the number of available components (DSPs, ASICs, Memories, etc.) rises continuously. These facts necessitate a structured method for selecting components, mapping applications and evaluating the chosen configuration and mapping. In this work we present a methodology for the last named. We will consider optimization of memory allocation and task scheduling as a packing problem and minimize needed memory area. The results can be used as one element of an automated performance analysis for a given system on a high abstraction level. This analysis is essential for establishing a framework that iterates over a large quantity of possible systems. Considering a part of the H.264 codec as an example we will illustrate the results. Furthermore we will show that results can be retrieved fast compared to other NP-hard problems due to intelligent formulation of conditions within the linear program

## 1 Introduction

In today's system on chip (SoC) design there is a big gap between register transfer level simulations and higher abstraction level models. Efforts have been made to close this gap in recent years. But commercial solutions still assume a given system and consider mainly the process of mapping tasks to the system components. The quality of the system still depends on the knowledge of the system engineer. From the research area frameworks are emerging that enable automatic iterations over various systems. Unfortunately, performance analysis of the chosen mapping often stays unrevealed, is forwarded to the next lower abstraction level or focuses on heuristics like list scheduling. In most cases also memory sizes are assumed as constraints and not as variables. Therefore, overall execution time or throughput is optimized.

But memory allocation and scheduling strongly influences die size, power consumption and moreover the whole communication architecture of the resulting SoC. Thus, an early proper evaluation for the chosen memory hierarchy is necessary. For evaluation exact algorithms are eligible, which are able to determine *how much memory is needed, when tasks have to start and how data can be stored without fragmentation*. We will present a methodology that solves this problem for application tailored SoCs via linear programming. In doing so we show that this problem can be regarded as modified packaging problem.

We will not focus on minimizing overall execution time but on minimizing used memory resources under a given timing constraint. This is done for mainly two reasons. First, in lots of applications in mobile networking a maximum overall execution time is given, e.g. time for decoding a picture within a video stream. Second, after a system is chosen and tasks are mapped to components and memories, the final die size is only influenced by used memory capacities. Hence, this minimization of used memory resources leads directly to minimization of necessary die size and silicon costs. Moreover, smaller memory capacities result in less energy consumption, which is crucial for embedded systems in mobile devices.

We think of the presented methodology as a starting point for evaluating systems configurations without time-consuming simulations and trial-and-error approaches. Together with automated analysis of other metrics, automatic iteration over and evaluation of various system configurations can be possible. The results can then be used for simulations on lower abstraction levels.

## 2 Related Work

A large contribution in closing the gap mentioned in section 1 has been done by the MESCAL [1] and Ptolemy [2] projects. These as many other tools (e.g. Artemis [3]) are based on the Y-chart approach [4]. It describes the need and a possible modus operandi for automatic iteration over various SoCs and different abstraction levels. Another approach based on integer linear programming and considering the entire mapping process is presented in [5].

As mentioned in section 1 the focus of this paper is restricted to scheduling. Pioneering work has been done by Liu and Layland back in 1973 [6] and Baruah et. al. [7]. Since then a lot of papers were presented treating scheduling in different ways. Many approaches focus on minimizing makespan, e.g. [8] and neglect optimization of memory requirements. An optimization methodology for energy consumption under a *given* memory size can be found in [9].

A possibility to minimize memory requirements is presented in [10]. Therein the authors focus on minimizing buffer requirements for all rate-optimal schedules. In contrast we consider all schedules meeting the (given) timing constraint. Furthermore multiple memories as well as restrictions on simultaneous memory accesses are included.

In some tools multi-objective optimization (MOO) is used, e.g. [11]. But we rejected optimizing both makespan and resources simultaneously. From the mathematical point of view MOO has some downsides. Firstly, it produces a set of pareto-optimal solutions, from which the preferred one has to be chosen manually. Secondly, the existence of more than one solution prevents solvers from efficiently making use of branch&bound techniques. Latter leads to significantly higher solving times and, therefore, the necessity to use suboptimal heuristics or genetic algorithms. However, if optimization of more than objective is wanted, our result is suitable for a first step optimization in an lexicographical objective environment.

A mathematical introduction with a collection of algorithms for classic scheduling problems mainly treating minimizing makespan can be found in [12]. An approach for a classification scheme regarding resource-constraint scheduling is presented in [13].

### 3 Methodology

The given problem "Minimizing total needed memory capacity via scheduling" can be formulated as a modified two-dimensional strip packaging problem (2D-SPP). The 2D-SPP describes the problem of packing boxes of fixed width and height into a strip of fixed width in such a manner, that total height is minimized and boxes are non-overlapping. It is described more detailed in [14].

We assume that the application is given as algorithm. Dependencies between the tasks (e.g. functions, operations) are data dependencies described by variables. In our model execution time of a task depends on the used component and memories. So execution time is fixed after mapping of tasks to components and memory. Components in this context are defined as elements processing tasks, such as Microprocessors, DSPs, ASICs, FPGAs, etc. A task itself consists of three phases:

1. *fetch*, in which needed data is transferred into the work memory,
2. *execute*, in which the task is performed by the assigned component,
3. *write back*, in which data is transferred into other memories for further processing.

Furthermore we presume a given maximal overall execution time by the standard specification. Therefore, the number of possible paths/branches caused by if/then conditions is finite and loops can be eliminated by series arranged tasks. In case of data dependent iteration counts each possible number of iterations is considered as one possible path.

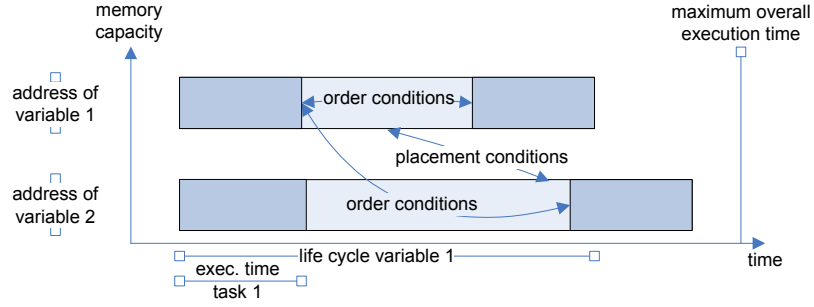
Applying this scenario to the problem of scheduling tasks and packing variables into memory the x-axis is time (where maximum execution time is determining the width of the strip) and of the y-axis is memory capacity (Fig. 1). The given problem placing variables and scheduling tasks necessitates a slight modification of the classic 2D-SPP, because there are not only one but two kinds of boxes to be packed:

1. *Outer boxes* represent the life cycles of variables. A life cycle starts when the variable is initialized in or transferred into memory and ends when it is needed no longer or transferred into another memory. The life cycles (respectively memory requirements) of the variables are the lengths (heights) of the boxes.
2. *Inner boxes*. Life cycles consist of different phases: phases, where the variable is used by a task or for transfer, and phases, in which the variable is stored, but not accessed. The inner boxes are characterized by the times the variable is accessed by tasks or transferred to another memory. Consequently, these boxes are nested in the outer boxes. The widths of the inner boxes are determined by the execution times of the corresponding tasks, the heights again are given by the memory requirements.

Note that the width of the inner boxes is given by the execution time of the referring task and fixed after mapping. In difference to classical 2D-SPPs the width of the outer boxes is variable due to the mutable storage times of variables between tasks. The two kinds of boxes are resulting in two different kinds of condition blocks in the later given mixed integer linear program (MILP) as visualized in Fig. 1:

1. *Order conditions*. As a result of data dependencies between tasks the inner boxes cannot be packed arbitrary regarding the horizontal position, e.g. if task  $i$  has to be completed before task  $j$  starts. These dependencies are included by one-dim. order conditions.

2. *Placement conditions.* This condition block is related to the two-dimensional placement conditions of the outer boxes (variables). In principle the positioning of these is free unless they do not overlap (respectively are not stored in the same memory address), but maybe restricted by order conditions referring to the inner boxes.



**Fig. 1.** Illustration of the scheduling problem interpreted as two-dimensional strip-packing problem. Outer boxes identify the life cycles and memory addresses of variables, inner boxes denote the execution times of tasks

Fig. 1 illustrates only one possible path in the flowchart and one memory. Optimization however has to be simultaneous for all possible paths and existing memories. This is done by ensuring that all tasks that exist in two paths start at the same time and variables, whose life cycles overlap in two paths, are placed at the same memory address. To consider multiple memories multiple 2D-SPPs are merged in one superior 2D-SPP. Both are described more detailed in the following section.

## 4 Detailed Problem Formulation

We will now present the mathematical formulation of the verbal given modified 2D-SPP as mixed integer linear program. But before we start, we need some definitions.

### 4.1 Definitions

With  $x_i \geq 0$  (respectively  $w_i \geq 0$ ) we denote the start time (execution time) of task  $i$ .  $y_i \geq 0$  ( $h_i \geq 0$ ) specifies the start memory address (memory requirements) of variable  $i$ .  $W$  represents the maximum overall execution time. Furthermore we use the variables  $u_{i,j}$ ,  $b_{i,j}$  and  $b'_{i,j}$  as follows:

$$u_{i,j} := \begin{cases} 1 & \text{if variable } i \text{ is placed below variable } j \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

$$b_{i,j} := \begin{cases} 1 & \text{if variable } i \text{ is placed before variable } j \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

$$b'_{i,j} := \begin{cases} 1 & \text{if task } i \text{ ends before task } j \text{ starts} \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

$c_n$  denotes the vector of available memory capacities for memory  $n$ ,  $a_n$  the vector of areas for the respective memory capacities and  $H \geq \max_n c_n$  a constant. Let

$$z_{n,k} := \begin{cases} 1 & \text{if capacity step } k \text{ is chosen for memory } n \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

Supplementary we need some set definitions. Let

$$M_{g,v}^n := \left\{ (p,t) \mid \begin{array}{l} \text{phase } p \text{ of task } t \text{ in path } g \text{ is accessing variable } v \text{ in mem-} \\ \text{ory number } n \end{array} \right\} \quad (5)$$

which is according to the set of all phase/task pairs accessing the variable  $v$  in path  $g$  and memory  $n$ . Within that set we denote the phase  $p$  and task  $t$  in which the life cycle of the variable  $v$  in memory  $n$  is beginning (ending) with  $s_{M_{g,v}^n}$  ( $e_{M_{g,v}^n}$ )  $\in M_{g,v}^n$ .

$$M_g^n := \{M_{g,1}^n, M_{g,2}^n, \dots\} \quad (6)$$

is the set of all variables allocated in memory  $n$ . Analogous we define  $E_g^n := \{e_1, e_2, \dots\}$  and  $S_g^n := \{s_1, s_2, \dots\}$ . These are the sets of tasks being end and respectively start of the lifecycle of a variable in memory  $n$  and path  $g$ .

## 4.2 Placement Conditions

First all boxes have to be placed within the strip or in other words have to end before maximum overall execution time. This is done by

$$x_i + w_i \leq W \quad \forall i \in M_{g,v}^n, \forall g, n \quad (7)$$

To implement the corresponding vertical condition block we have to go into more detail. Since classic 2D-SPPs are NP-complete the range of solvable problems in finite time is limited. But the good news is that this special case of the problem (or to be more precise the minimization of needed memory resources) does not have to be solved exactly. Memory capacity is only available in discrete capacity steps, usually to the power of 2. So if memory is available for example in 32KBit and 64KBit it makes no difference if the capacity needed is 52KBit or 33KBit, as long as optimization cannot result in  $\leq 32$ KBit and a solution  $\leq 64$ KBit is capable. This characteristic is implemented by

$$y_i + h_i \leq \sum_k c_{n,k} z_{n,k} \quad \forall i \in M_g^n, \forall g, n \quad (8)$$

and

$$\sum_k z_{n,k} = 1 \quad \forall n \quad (9)$$

(8) guarantees that all outer boxes are placed below the chosen memory size, (9) makes sure that exactly one memory size is chosen for each existing memory. Considering as

an example we will show in the following section that this minimization of memory capacity, not total needed memory, will result in considerable less time needed to find an optimal solution than minimizing total required memory resources.

Having ensured that all variables are placed within the memory and all tasks are completed before maximum execution time we have to ensure that two variables existing in the same path and memory are not allocated to the same memory address. That means all boxes (inner as well as outer ones) must not overlap. This is assured by

$$x_{e_i} + w_{e_i} - W + Wb_{i,j} \leq x_{s_j} \quad (10)$$

$$y_i + h_i - H + Hu_{j,i} \leq y_j \quad (11)$$

$$b_{j,i} + b_{i,j} \leq 1 \quad (12)$$

$$u_{j,i} + u_{i,j} \leq 1 \quad (13)$$

$$u_{j,i} + u_{i,j} + b_{j,i} + b_{i,j} \geq 1 \quad (14)$$

(10) – (14) hold  $\forall i, j \in M_g^n, i \neq j, e_i \in E_g^n, s_j \in S_g^n, \forall g, n$ . For each pair  $i, j$  of boxes (14) ensures at least one of the four conditions resulting from (10) and (11) is not redundant with  $x_i \geq 0$  or  $y_i \geq 0$  respectively. (12) and (13) prevent overlapping of the boxes. There is always one phase of a task constituting the start and end of the life cycle of a variable. So to prevent overlapping of the outer boxes, it is sufficient to postulate (10) only for the inner boxes delimiting an outer box.

To include if/then-conditions into the MILP (15) is added. It guarantees the placement of a variable existing in two possible paths  $g_1$  and  $g_2$  in the same vertical spot if at least one task is shared by the two paths.

$$y_{M_{g_1,v}^n} = y_{M_{g_2,v}^n} \quad \forall M_{g_1,v}^n \in M_{g_1}^n, M_{g_2,v}^n \in M_{g_2}^n : \exists (p, t) \in M_{g_1,v}^n \cap M_{g_2,v}^n, \forall n \quad (15)$$

Note, by definition a task existing in two possible paths is always started at the same time. However, if differentiated starting times are desired, a slight modification in terms of incorporating identifiers for the corresponding path by additional indices is necessary. It is also presumed that loops are eliminated. Remark that infinite loops do not exist due to the given maximum overall execution time resulting in a finite number of possible tasks.

The given set definitions also guarantee that two inner boxes with the same tag in different memories are located on the same spot horizontally. Hence, the starting time of the associated task is independent of the accessed memories. These facts and (15) are the only placement conditions between different paths and memories.

### 4.3 Order Conditions

The order conditions are defined by the application and can be derived from the flow-chart. There are two different kinds of order conditions. Firstly, (16) denotes that task  $j$  is dependent on task  $i$ .

$$x_i + w_i \leq x_j \quad (16)$$

Secondly, the number of simultaneously performed tasks by one component or the quantity of simultaneous memory accesses by different tasks can be limited. (17) and (18) are an example for such a limitation.

$$x_i + w_i - W + Wb'_{i,j} \leq x_j \quad (17)$$

$$b'_{i,j} + b'_{j,i} = 1 \quad (18)$$

In the given case the number of simultaneous accesses to one memory by tasks is limited to one. Formulation for other cases can be adopted easily.

#### 4.4 Objective Function

The objective function is given by

$$\sum_n \sum_k a_{n,k} z_{n,k} \rightarrow \min \quad (19)$$

With (19) total area used by memory components on the system is minimized. Area in this model is given by the sum of areas of all used memory elements. Other components of the system are unaccounted because these elements are fixed after mapping. More sophisticated definitions for calculating die size can be integrated easily by modifying the objective and adding additional constraints.

## 5 Results

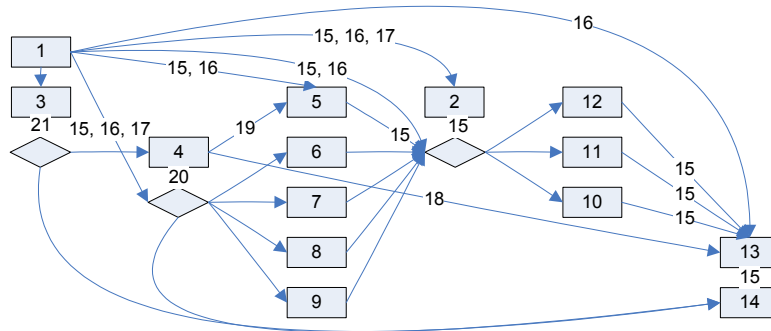
We have tested the methodology with a part of the H.264 video codec [15] from the Fraunhofer Hertz Institute Berlin, namely the function `decodeMBInter`. The problem characteristics are shown in table 1.

**Table 1.** Characteristics of the examined H.264-part

# of tasks	14
# of paths	14
# of variables	21

Fig. 2 illustrates the given data dependencies. Maximum overall execution time, times of tasks, memory requirements of the variables and memory capacity steps are a first estimate for a possible SoC configuration. Data dependencies were modified slightly to include possible parallel processing of tasks mapped to different components. Parallel processing of two tasks mapped to the same component was forbidden.

We assumed two components and two memories. One acting as system memory equipped with a single port and one representing a shared memory with double port. The mapping of tasks to components is shown in table 2. Note that tasks 1 and 14 denote dummy tasks identifying start and end. Consequently they do not have to be assigned to



**Fig. 2.** Flowchart of the examined H.264-part. Numbers in the boxes identify the tasks, numbers beside the arcs label variables. Temporary variables needed only within the tasks are later referred by the number of the corresponding task. Task 1 and 14 denote additional inserted starting and ending tasks, which are not assigned to components

any components. Local memory for both components is the shared memory. To simulate memory transfers and the three phases fetch, execute and write back in our model a temporary variable is transferred from system to shared memory at the beginning of each task. After execution the variable is transferred back to system memory.

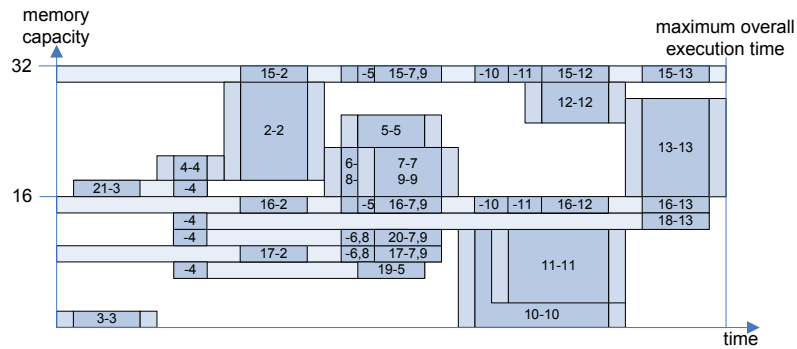
**Table 2.** Assumed mapping of tasks to components for the examined H.264-part

component executed tasks	
1	2 – 5
2	6 – 13

Optimization was done for the shared memory. The resulting MILP had 814 variables (thereof 748 binary) and 1491 constraints. Fig. 3 illustrates the result. As you can see the depicted solution is far away from minimum overall memory usage but optimal in respect to the chosen memory capacity step. The problem was solved in about 5 seconds on an AMD Dual Opteron System with 2.2 GHz processors each using CPLEX 9.1.

By contrast the solving time for minimizing needed memory capacity  $h$  (with modified condition (8):  $y_i + h_i \leq h$  and without condition (9)) took more than 500 sec. or in this case  $> 10,000\%$  compared to minimizing needed memory capacity step. It also illustrates the possible impact of a slightly different MILP formulation. The reason for this big difference is the utilization of the memory capacity steps as upper and lower bounds. If during the algorithm automatically generated lower bound and the capable solution are in between the same two memory capacity steps the algorithm terminates. This behavior is illustrated in Fig. 3, where optimization is stopped after a solution of 32 Kbit is capable and a solution of 16 Kbit is ruled out.

We also performed a trade off analysis with a different system configuration. The shared memory was replaced by a single ported memory for each component. Opti-



**Fig. 3.** Illustration of the optimized schedule of tasks and allocation of variables in respect to memory capacity step. Light blue boxes mark the outer boxes representing the life cycle of the variables, narrow boxes denote the phases in which a variable is transferred/initialized and blue boxes identify the inner boxes representing the tasks accessing the required variables. The labels of the boxes identify tasks – variables

mization resulted in a required memory capacity of at least 32 Kbit for each component. Although single ported memory is smaller compared to double ported, the overall needed die size was almost the same for both configurations. For calculation we used custom designed memory elements by UMC/Virtual Silicon Technology.

## 6 Conclusions

In section 5 we have demonstrated a methodology for memory allocation and task scheduling applicable to real life problems of small size. The solution time shows that the full potential of the methodology is not tapped by the passed problem.

Furthermore our approach is highly flexible. The assumption that a task consists of the three phases, is not a must. Inclusion of additional phases or reduction to one phase is possible without large effort. Latter can be reasonable, if exact execution and transfer times are not known in an early design stage, but a first estimation model for scheduling is desired. For this case also data dependencies can be reduced to simple task dependencies. Moreover the methodology does not care what is stored in the memory. The MILP formulation holds for data as well as instruction code. Also, the type of memory to be optimized is arbitrary. Thus, scratchpad memory, register files, etc. can be simulated.

As mentioned in section 2 there exist methods for optimizing memory allocation under given memory capacities. Examination of possible combination of these methods with our methodology could lead to a refinement of existing high abstraction level models and consequently could be the next step for automated iteration over and performance evaluation of several systems.

## References

1. Mihal, A., Kulkarni, C., Sauer, C., Vissers, K., Moskevicz, M., Tsai, M., Shah, N., Weber, S., Jin, Y., Keutzer, K., Malik, S. In: *Developing Architectural Platforms: A Disciplined Approach*. Volume 19. IEEE Design and Test of Computers (2002) 6–16
2. Lee, E.A.: Overview of the ptolemy project. Technical memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA. (2003)
3. Pimentel, A.D., Hertzberger, L.O., Lieverse, P., van der Wolf, P., Deprettere, E.F.: Exploring Embedded-Systems Architectures with Artemis. *Computer* **34** (2001) 57–63
4. Kienhuis, B., Deprettere, E., Vissers, K., van der Wolf, P.: An approach for quantitative analysis of application-specific dataflow architectures. In: *ASAP '97: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, Washington, DC, USA, IEEE Computer Society (1997) 338
5. Niemann, R., Marwedel, P.: Hardware/software partitioning using integer programming. In: *EDTC '96: Proceedings of the 1996 European conference on Design and Test*, Washington, DC, USA, IEEE Computer Society (1996) 473
6. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* **20** (1973) 46–61
7. Baruah, S.K., Gehrke, J.E., Plaxton, C.G.: Fast scheduling of periodic tasks on multiple resources. Technical Report CS-TR-95-02, University of Texas, Austin, Austin, TX, USA (1995)
8. Jin, Y., Satish, N., Ravindran, K., Keutzer, K.: An automated exploration framework for fpga-based soft multiprocessor systems. In: *CODES '05: Proceedings of the 2005 International Conference on Hardware/Software Codesign and System Synthesis*. (2005) 273–278
9. Verma, M., Wehmeyer, L., Marwedel, P.: Dynamic overlay of scratchpad memory for energy minimization. In: *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, New York, NY, USA, ACM Press (2004) 104–109
10. Govindarajan, R., Gao, G., Desai, P.: Minimizing Memory Requirements in Rate-Optimal Schedules. In: *ASAP '94: Proceedings of the International Conference on Application Specific Array Processors*. Volume Application Specific Array Processors, . Proceedings., International Conference on. (1994) 75–86
11. Dick, R.P., Jha, N.K.: MOGAC: a multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems. In: *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, Washington, DC, USA, IEEE Computer Society (1997) 522–529
12. Bruckner, P.: *Scheduling Algorithms*. 4th edn. Springer-Verlag (2003)
13. Brucker, P., Drexler, A., Möhring, R., Neumann, K., Pesch, E.: Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research* **112** (1999) 3–41
14. Belov, G., Chiglintsev, A., Filippova, A., Mukhacheva, E., Scheithauer, G., Shirgazin, R.: The two-dimensional strip packing problem: A numerical experiment with waste-free instances using algorithms with block structure. Preprint MATH-NM-01-2005 TU Dresden (2005)
15. Wiegand, T., Sullivan, G., Bjntegaard, G., Luthra, A.: Overview of the H.264/AVC video coding standard. In: *Circuits and Systems for Video Technology*, IEEE Transactions on. Volume vol.13, no.7., IEEE Circuits and Systems Society (2003) 560–576