The Journal of Supercomputing, 19(1), 57–75, 2001
© 2001 Kluwer Academic Publishers. Manufactured in The Netherlands.

Optimization of Dynamic Hardware Reconfigurations

JÜRGEN TEICH teich@date.upb.de
Computer Engineering, University of Paderborn, Germany

SÁNDOR P. FEKETE fekete@math.tu-berlin.de
Department of Mathematics, TU Berlin, Germany

JÖRG SCHEPERS joerg_schepers@de.ibmmail.com
IBM Germany, Köln, Germany

October 1, 1999

Abstract. Recent generations of Field Programmable Gate Arrays (FPGA) allow the dynamic reconfiguration of cells on the chip during run-time. For a given problem consisting of a set of tasks with computation requirements modeled by rectangles of cells, several optimization problems such as finding the array of minimal size to accomplish the tasks within a given time limit are considered. Existing approaches based on ILP formulations to solve these problems as multi-dimensional packing problems turn out not to be applicable for problem sizes of interest. Here, a breakthrough is achieved in solving these problems to optimality by using the new notion of *packing classes*. It allows a significant reduction of the search space such that problems of the above type may be solved exactly using a special branch-and-bound technique. We validate the usefulness of our method by providing computational results.

Keywords: reconfigurable hardware, multi-dimensional placement and packing

1. Introduction

Field-Programmable Gate Arrays (FPGA's) are a new and important class of hardware devices. Typically, they consist of a regular rectangular grid of equal configurable cells (logic blocks) that allow the prototyping of simple logic functions together with simple registers and with special routing resources (see Figure 1). A particular design is realized by customizing a configuration. In traditional SRAM-based chips, this can be done at power-up by loading a configuration bit-stream serially into the chip. These chips may only be reconfigured as a whole with typical reconfiguration times ranging in the order of milliseconds.

Today, new generations of FPGAs have become partitionable and dynamically reconfigurable, even partially. These chips (e.g., see (Atm; Xil96)) may support several independent or interdependent tasks and designs at a time, and parts of the chip can be reconfigured quickly during run-time.

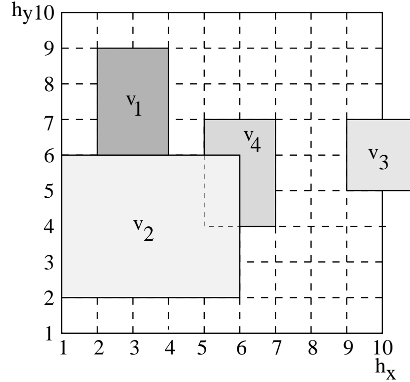

Figure 1. Example of an FPGA consisting of a rectangular grid of $h_x \times h_y = 9 \times 9$ logic cells. Each module (v_1, v_2, v_3, v_4) occupies a rectangle of cells. Modules must be placed inside the chip and must not overlap if executed simultaneously on the chip.

In the following, we consider architectures similar to the Xilinx 6200 FPGA (Xil96) architecture, where column readins and readouts of flipflop contents may be performed during run-time without interfering with other configured parts of the chip. Under these assumptions, a task may be represented by a 3D polytope with two spatial dimensions, the third one representing the time of computation, see Fig. 2.[1]

However, even if the configuration time is short, the compilation time for constructing the configuration stream for a task is still rather long. This diminishes the results that have been reported recently on on-line strategies for compiling and reconfiguring such devices. Important examples include speeding up computational problems in hardware by task compaction on hypercubes (HJ90), or approaches to dynamic allocation of a sequence of tasks on an FPGA of given size by using heuristics to compact tasks in execution on the chip during run-time (DG97b; DG97a).

Here, we consider *statically defined problems where a task set is given which may or may or not impose a partial order of execution*. Such a set of tasks may be described by a dependency graph, see Figure 2. From the static structure of the problem, we may optimize the layout of each task separately. For such a problem instance, we

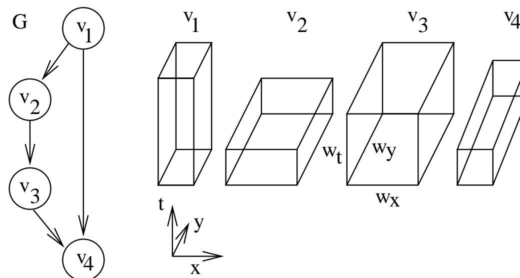

Figure 2. Dependency graph of tasks and shape of modules (3D boxes) with the spatial dimensions x and y and the temporal dimension t (execution time).

are interested in finding exact solutions to the following problems:

— Find the chip of smallest size to accommodate all tasks such that a given maximum execution-time of the set of tasks is guaranteed (*MinA&FindS*) together with a feasible schedule. A subproblem of this problem occurs when assuming a feasible schedule of all tasks is already given. We call this problem type *MinA&FixedS*.
— Check whether for a chip of given size and given maximum execution time, there is a feasible placement and a feasible schedule that accommodates a set of tasks (*FeasAT&FindS*).
— Find the smallest execution time of the set of tasks for a chip of fixed size (*MinT&FindS*).
— Check whether for a chip of given size and a given feasible schedule, there is a feasible placement (*FeasA&FixedS*).

A *schedule* is called feasible, if the execution intervals of tasks satisfy the precedence relation. In case of a given maximum execution time, the schedule must also satisfy the requirement that all tasks have finished their execution by this time. A *placement* is called feasible if the locations occupied by each pair of tasks that have overlapping execution intervals are disjoint.

First, we show how these problems relate to a special class of *higher-dimensional packing problems*. Since their one-dimensional counterparts are NP-complete in the strict sense (GJ79), these generalizations are also difficult to solve. Methods have been proposed to use ILP formulations to solve special problem cases such as the 2-dimensional knapsack problem (Bea85; HC95). Not surprisingly, the authors report of being able to solve only problems of very small sizes (e.g., for a grid of 30×30 cells).

Here, we exploit a new and more efficient way of representing the packing problem using *packing classes* (FS97; Sch97). We show that exact solutions for the problem types *MinA&FixedS* and *FeasA&FixedS* may be found for problems of technical interest using a new branch-and-bound technique. Other variants and extensions (like *FeasA&FindS* and *MinA&FindS*) require additional mathematical machinery and will be dealt with in future work.

The proposed methodology may be particularly useful for prototyping *piecewise regular algorithms* (MC94; TTZ97; Pla97) in hardware. In this sense, the task graph can be seen as a system level view of the dependencies of communicating regular subalgorithms, each being presynthesized and stored in an FPGA module library, and represented by a relocatable rectangle of cells and its execution time.

2. Reconfiguration and Packing

2.1. Definitions

Definition 1 (Problem instance). A *problem instance* is given by a directed, acyclic task graph $G = (V, A)$ where the nodes $v \in V$ denote tasks or designs, and the

arcs $a \in A$ denote partial order constraints on the execution of the tasks. To each task $v \in V$, there are assigned the weights

— $w_x(v) \in \mathbf{N}$, denoting the length of the layout (number of cells occupied) in the x-dimension.
— $w_y(v) \in \mathbf{N}$, denoting the length of the layout in the y-dimension, and
— $w_t(v) \in \mathbf{N}$, denoting the execution time of task v.

Definition 2 (Feasible schedule). For a given problem instance $G = (V, A)$, a *schedule* is given by a function $p_t : V \to \mathbf{N}$. A schedule p_t is called *feasible* if $\forall a \in A : p_t(\text{head}(a)) \geq p_t(\text{tail}(a)) + w_t(\text{tail}(a))$ where head(a) denotes the head of a and tail(a) denotes the tail of a. Under a given *maximum execution time* constraint $h_t \in \mathbf{N}$, a feasible schedule p_t must satisfy the additional constraint: $\forall v \in V : p_t(v) + w_t(v) \leq h_t$.

Definition 3 (Reconfigurable chip). A *reconfigurable chip* H is given by an array of $h_x \times h_y$ cells where $h_x, h_y \in \mathbf{N}$. Its area $A \in \mathbf{N}$ is given by $A = h_x \times h_y$.

A placement of tasks may be described as follows:

Definition 4 (Feasible placement). Given a problem instance $G = (V, A)$ and a reconfigurable chip H of size $h_x \times h_y$. A *feasible placement* is a three-dimensional vector function $p = (p_x, p_y, p_t) : V \to \mathbf{N}^3$, where the values $p_x(v), p_y(v), p_t(v)$ denote the leftmost cell, down-most cell, and the smallest time index occupied by v. Task v occupies the cells in the interval $[p_x(v), \ldots, p_x(v) + w_x(v))$ in the x, of $[p_y(v), \ldots, p_y(v) + w_y(v))$ in the y, and of $[p_t(v), \ldots, p_t(v) + w_t(v))$ in the t (time) dimension.

Example 5. See Figure 1. There are four tasks. Shown are the spatial dimensions of the tasks, i.e., $w_x(v_1) = 2$, $w_x(v_2) = 5$, $w_x(v_3) = 2$, $w_x(v_4) = 2$, and $w_y(v_1) = 3$, $w_y(v_2) = 4$, $w_y(v_3) = 2$, $w_y(v_4) = 3$. The execution times of the tasks are not shown.

A placement is feasible if and only if the rightmost cell and the topmost cell occupied by each task is smaller than or equal to the number of cells in that direction. Also, one must guarantee that for any two modules overlapping in time, the cells occupied by them in either the x- or the y-dimension must be disjoint in order for the corresponding *boxes* not to overlap.

Example 6. The placement shown in Figure 1 is infeasible. For one thing, task v_3 does not satisfy the constraint that it has to be placed completely within the chip. Also, in case v_2 and v_4 overlap in execution, these tasks occupy the same space at the same time.

A problem instance with a given feasible schedule p_t corresponds to a preplacement of the boxes in the t-dimension.

2.2. Packing

In order to show the relationship between the optimization and decision problems defined above and packing, we pick one special problem, namely that of *FeasA&FindS*, and consider the subclass of problems for which $G = (V, A)$ has no arcs (($A = \emptyset$), independent task set). Thus, we have to find a feasible placement on a chip of given size to accommodate all task boxes. This problem is known as the *Orthogonal Packing Problem* (OPP):

Definition 7 (OPP). Given a set of boxes V, a weight function w, and the container size function h. The OPP denotes the decision problem: Is there a feasible placement (packing) for (V, w, h)?

The OPP is the basis to several more-dimensional packing problems. One of them is the so-called *square-packing-problem* (SPP), where we want to determine a quadratic chip of smallest size that suffices to accommodate a given set of tasks:

Definition 8 (SPP). Given is set of boxes V and a vector weight function $w = (w_x, w_y, w_t)$. Also given h_t.

$$\begin{aligned} &\textbf{min} \quad h = h_x = h_y \in \mathbf{N} \\ &\textbf{s.t.} \quad \exists \text{ feas. placement for } (V, w, (h, h, h_t)) \end{aligned}$$

In the following, we write (V, w, h) to denote an instance of a *packing* (placement) *problem*, where V is a set of boxes, w is a vector weight (size) function, and h is a vector function denoting the extensions of the container. Also, we say that the obvious generalization of a feasible placement in Definition 4 to arbitrary dimensions defines a *packing* p.

All problems defined in the introduction belong to the class of *orthogonal packing problems* with *fixed orientations*: Each edge of a box must be placed parallel to an edge of the container, and boxes may not be rotated. Clearly, time and spatial orientations may not be interchanged, and the constraint of not interchanging x- and y-dimension typically arises by column- or row-oriented chips.

2.3. Previous Algorithms

In order to solve general optimization problems in the context of packing, it has to be checked whether a set of constraints such as given in Definition 4 is satisfied. A central role in this matter plays the OPP introduced in Definition 7. Using a grid decomposition, it can be formulated as a 0-1 program (Bea85). Unfortunately, experiments show that this approach can only be used to solve very small problems to optimality, even for our case where we already have a schedule. The reason lies in the exponential size of the resulting integer linear program: It has $O(|V||X||Y|)$ 0-1 variables and $O(|X||Y||T|)$ constraints where $|X|$, $|Y|$, $|T|$ are the dimensions of the underlying grid in the x-, y-, and t-direction.

The largest two-dimensional packing problems that have been solved with this technique place about 20 rectangles on a 30×30 grid (Bea85; HC95). To solve a three-dimensional problem with about 100 nodes is hopeless if these standard solution techniques are used.

3. Architecture Assumptions

3.1. Intermodule Communication

Intermodule communication is assumed to occur at the end of operation of the sending module (task model). The issuing module may store its result register values into an external memory connected to the FPGA interface (readout) via a bus interface. Memory is allocated to store temporarily intermediate results.[2] Afterwards, the receiving module will read in the communicated data into its registers over the bus interface. With this communication style, it is justifiable to ignore routing overhead between modules that otherwise might introduce additional placement constraints.

3.2. I/O-overhead

The communication time needed for writing out and reading in communicated data may be accounted for by considering this as an offset that is part of the execution time of a task.

3.3. Reconfiguration Overhead

The time needed for carrying out reconfigurations may be modeled by a constant (possibly an individual number for each task), depending on the target architecture. This may be considered a simplification because the reconfiguration time might depend on the result of the placement. However, many different models of taking into account reconfiguration times can be thought of, and have to be adapted individually to the target architecture.

4. A Solution Technique for the SPP

In the following, we propose an approach for solving the square-packing problem (SPP) in Definition 8 under the assumption that the OPP can be solved in a reasonable amount of time for problem sizes of interest. Techniques for an efficient solution of OPP's is presented in Section 5.

Obviously, a binary search technique may be applied to solve the SPP using the OPP:

— Use a heuristic to construct a packing for (V, w, h) returning an upper bound $h_u = \max\{h_x, h_y\}$ on the chip size.

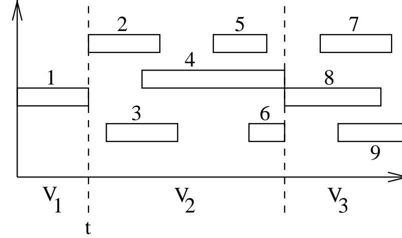

Figure 3. Extraction of independent subproblems for finding better upper bounds.

— Determine a lower bound h_l using a method as described in Section 5.
— In the interval $[h_l, \ldots, h_u]$, apply binary search to determine the smallest possible chip size.

4.1. A Trivial Upper Bound

A trivial upper bound h_u may be obtained for a given schedule as follows:

$$h_u = \max\left\{\sum_{v\in V} w_x(v), \sum_{v\in V} w_y(v)\right\}$$

4.2. Problem Pre-conditioning

Sometimes a better upper bound may be easily computed in case a given schedule reveals a certain structure, like in the example in Figure 3.

Shown is a timing diagram of the lifetimes of boxes. In case there exists a time step t where no two intervals overlap, this time step defines a cut that partitions the boxes V into those that live at time steps smaller than t (V_1) and those that do not start executing before this time step (V_2). Both subproblems may be solved independently because no two boxes $v_1 \in V_1$ and $v_2 \in V_2$ can overlap in any spatial placement within the container.

Let the above scheme recognize k independent subproblems $V_1, \ldots, V_k$. Then a better upper bound may be achieved by defining

$$h_u = \max_k\{h_u(k)\},$$

where $h_u(k)$ is the upper bound determined for the kth partition block of boxes.

5. Solving OPP's

In this section, we describe our new approach for solving OPP's that was first used for pure geometric packing in (Sch97; FS97; FS98a; FS98b). It makes use of two new ideas: a) a new way of characterizing feasible packings, and b) an approach to

get good lower bounds. Finally, we give a brief overview over how these techniques can be combined in a branch-and-bound framework to yield an efficient algorithm for solving *MinA&FixedS*-problems.

5.1. Packing Classes

For a given packing, the projections onto the different coordinate axes turn out to be *interval graphs* $G_i(V, E_i)$, see Figure 4. These graphs possess the following properties:

P1: G_i is an interval graph, $\forall i \in \{1, \ldots, d\}$,
P2: Any independent set S of G_i is i-admissible, $\forall i \in \{1, \ldots, d\}$. In other words, $w_i(S) = \sum_{v \in S} w_i(v) \leq h_i$, since all boxes in S must fit into the container in the ith dimension.
P3: $\cap_{i=1}^{d} E_i = \emptyset$. In other words, there must be at least one dimension in which the corresponding nodes do not overlap.

The G_i are called *component graphs* of $E = \{E_1, E_2, \ldots, E_d\}$.

Our search procedure works on d-tuples of component graphs with these properties; they are called *packing classes* in the following. It turns out that they represent

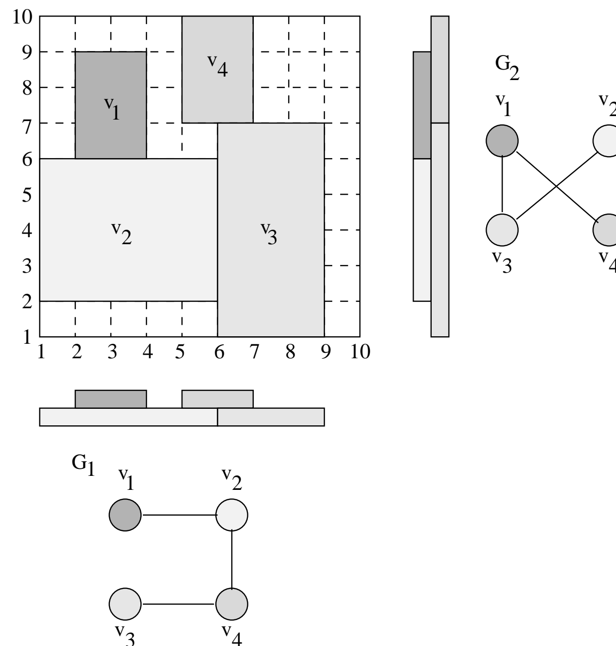

Figure 4. The projections of the boxes onto the coordinate axes define interval graphs (here in 2D: G_1 and G_2).

not only a single packing but a set of packings, which allows it to consider more than one possible candidate for an optimal packing at a time.

The way to obtain a packing class from a given packing is shown in Figure 4. Figure 5 demonstrates how to obtain packings from a given packing class.

Definition 9 (Orienting a packing class). Let E be a packing class and F_i, $i = 1, \ldots, d$, be a transitive orientation of the complements of the ith component graph G_i. Then $F := (F_1, \ldots, F_d)$ is called an orientation of the packing class E.

From an orientation F of a packing class E, one obtains a mapping $p^F : V \to \mathbf{Q}^d$ with

$$p_i^F(v) := \max\{1, p_i^F(u) + w_i(u) | \vec{uv} \in F_i\}, \ v \in V, \forall i = 1, \ldots, d$$

Example 10. Figure 5 shows an example of a transitive orientation and the corresponding packing.

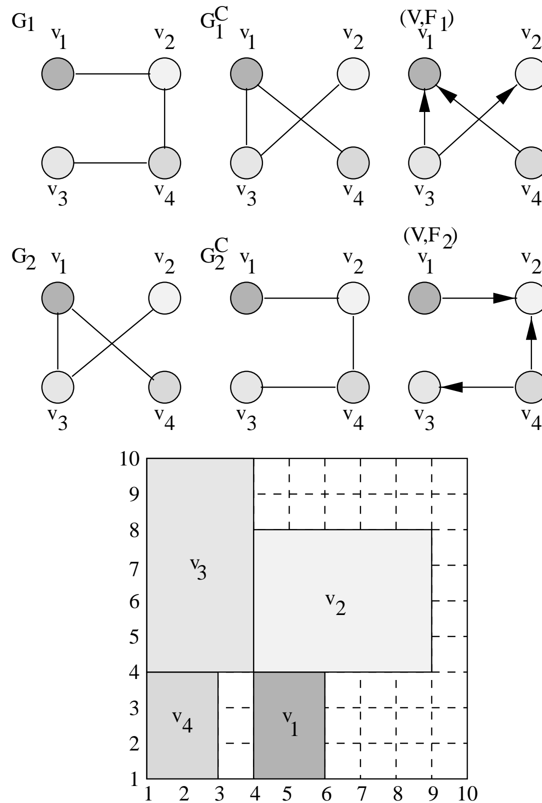

Figure 5. Obtaining a packing from a packing class. Each transitive orientation of the complement graphs of the component graphs defines a packing.

During the transition from packing to packing class, the information on the relative placement of any two nodes v_1 and v_2 not overlapping in the ith projection is lost. For such pairs of nodes, there is an edge in the corresponding complement graph G_i^C. An orientation of the edges in this graph reconstructs the relative position of nodes. Note that not just any orientation of edges in G_i^C is sufficient to define a packing, but only *transitive orientations* are feasible: if $(v_1, v_2) \in (V, F_i)$, and $(v_2, v_3) \in (V, F_i)$, then (v_1, v_3) must also be an element of (V, F_i).

For each orientation F of a packing class E is p^F a packing. $F := (F_1, F_2, \ldots, F_d)$ are transitive orientations of the complement of the ith graph G_i. Each orientation describes a different packing.

Example 11. The complement graphs shown in Figure 5 have two transitive orientations each. Figure 6 shows the total of 4 packings that are obtained as the combinations of these orders.

The main gain of talking about packing classes instead of packings is that existential statements about packings turn out to be equivalent to existential statements

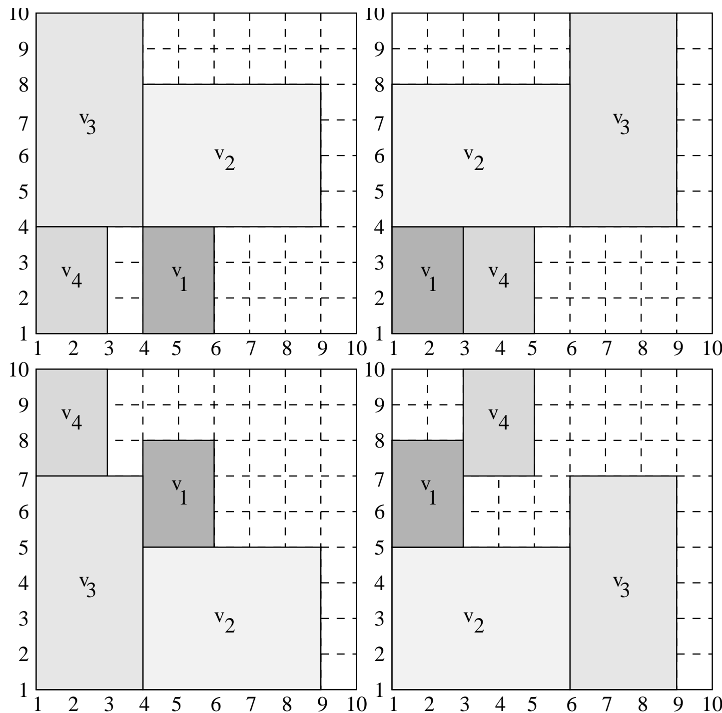

Figure 6. The packing class in Figure 5 has four different orientations which correspond to four different packings shown.

about packing classes:

Theorem 12. A d-tuple of graphs $G_i = (V, E_i)$ corresponds to a feasible packing, iff it is a packing class, i.e., if it satisfies conditions **P1, P2, P3**.

A proof can be found in (FS98b). This allows us to concentrate on packing classes instead of packings. Another important advantage of packing classes is the following: There are very powerful graph-theoretic characterizations for interval graphs and graphs with transitive orientations, see (Gol80) for more details.

The following terminology is used: An *induced cycle* in a graph G is given by a set of vertices $U = \{u_1, \dots, u_k\} \subseteq V$, such the cycle $C = \langle u_1, u_2, \dots, u_k, u_1\rangle$ of edges is contained in G. The length of the cycle is k. An induced cycle C is 2*-chordless*, if none of the edges (u_i, u_{i+2}) is contained in G; C is *chordless*, if only the edges (u_i, u_{i+1}) are contained in G.

Proposition 13. (GH64) *Let G be a graph whose complement graph has a transitive orientation. Then G is an interval graph, iff it does not contain an induced chordless cycle of length* 4.

Proposition 14. (GH62; GH64) *A graph has a transitive orientation, iff it does not contain a* 2*-chordless cycle of odd length.*

These characterizations can be used for fast subroutines for constructing packing classes via branch-and-bound (Section 6). Mathematical details and implementation aspects can be found in (FS98d).

5.2. Proving Infeasibility

A packing cannot exist if the sum of volumes of the boxes exceeds the container volumes.

Better criteria for infeasibility may be obtained by transformed volumes (Sch97). These techniques have been used in (FS97; FS98a; FS98c; TFS98) to obtain better bounds in two ways:

— use more than one (efficiently computable) volume criterion to obtain better bounds,
— use transformations of box sizes that allow to make exact decisions with better bounds.

In general, a combination of both of these approaches seems to provide convincing results.

Example 15. (Sch97; FS98a) Do 9 cubes of side length 2/5 fit into the unit cube? The sum of the individual cube volumes is given as $9(8/125) = 0.576 \leq 1$, so the volume criterion does not provide a proof of infeasibility. Using the transformation

function $u^{(2)}$ from (Sch97; FS98a), we obtain a transformed cube side length of $\frac{\lceil 3(2/5)-1 \rceil}{2} = 0.5$. Hence, the total sum of transformed volumes is $9(1/8) = 1.125 > 1$, proving infeasibility.

In (TFS98), it is shown how the transformed volume approach may be used in order to find better lower bounds for the SPP problem in Definition 8.

5.3. *Incorporation of Schedule Constraints*

So far, each dimension has been treated as being interchangeable with any spatial dimension. In case of our placement problem, however, time plays a special role. In particular, two special problem classes of packing problems must be considered:

5.3.1. FixedS-Problems. This class of problems assumes a given fixed schedule (temporal placement). In this case, the search space may be reduced to $d - 1 = 2$ dimensions by omitting the time component and by changing the definition of edges in the component graphs G_1 and G_2 as follows: In case two tasks $v_1 \in V$ and $v_2 \in V$ are not executed simultaneously, they may occupy the same chip area. Hence, condition P3 is already satisfied for edges between nodes E_1 and E_2 that are not executed simultaneously.

Example 16. Consider the d graphs $G_1, \ldots, G_d$ (corresponding to spatial dimensions) and two nodes $v_k, v_l \in V$ that are known to be scheduled at disjoint time slots. Hence, the d component graphs may all simultaneously contain an edge (v_k, v_l) see e.g. in Fig. 7 for $d = 2$, $k = 1$, $l = 2$. Then, none of the complement graphs can have an edge between v_k and v_l. Therefore, there cannot be an ordering of placement imposed between these two nodes in any spatial dimension.

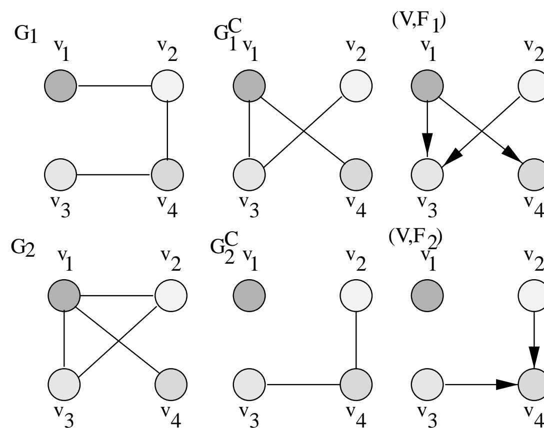

Figure 7. In case of a given schedule, all d spatial component graphs may contain an edge between two nodes, in case they are both scheduled at disjoint time slots. (Shown here for v_1 and v_2.)

Now, the above extensions due to schedule constraints make the problems introduced in Section 1 solvable as follows:

— *FeasA&FixedS*: Solve the 2D-instance of the OPP with the relaxation that packing classes may have an edge (v_i, v_j) in both component graphs $G_1 = (V, E_1)$ and $G_2(V, E_2)$, in case the execution intervals of v_i and v_j do not overlap (condition P3).
— *MinA&FixedS*: Apply the binary search proposed in Section 4 and bounds obtained by methods as described above and in (TFS98) for h with the same relaxation of P3 as for the *FeasA&FixedS*-problem.

6. A Branch-and-Bound Procedure

When solving the problem *FeasA&FixedS*, i.e., when checking whether a chip of given size is large enough to accommodate a set of tasks, we have to decide whether a set of boxes with fixed t-placements has a feasible packing. We approach this OPP in three steps, in ascending order of computational difficulty:

1. Try to disprove the existence of a packing with so-called *conservative scales* (FS97; TFS98). Stop in case of success.
2. Try to find a packing using an efficient (greedy) packing heuristic. Stop in case of success.
3. Try to construct a packing class by a tree search algorithm. In case of failure we can state that no packing for the given OPP–instance exists, otherwise, we get a feasible placement.

The search tree is traversed by Depth First Search, see (FS98d; Sch97) for details. Branching is done by fixing an edge $(b, c) \in E_i$ or $(b, c) \notin E_i$. After each branching step, it is checked if one of the three conditions P1, P2, P3 is violated, or whether a violation can only be avoided by fixing further edges. These tests as well as the test if the fixed edges already form a packing class are based on comparability graph recognition and on the calculation of maximal weighted cliques in comparability graphs, so they can be performed efficiently (Gol80).

In particular, Propositions 13 and 14, and condition P2 imply that the following forbidden configurations have to be dealt with:

1. induced chordless cycles of length 4 in E_i,
2. induced 2-chordless odd cycles in the set of edges excluded from E_i,
3. infeasible stable sets in E_i.

The use of P3 is obvious; Properties P1 and P2 are hereditary, so adding edges to E_i later will keep them satisfied. Each time we detect such a fixed subgraph, we can abandon the search on this node; if we detect a fixed subgraph, except for a set of "equivalent edges", we can fix one of them.

Our experience shows that these conditions are already useful when only small subsets of edges have been fixed: By excluding small sub-configurations like induced chordless cycles of length 4, each branching step triggers a cascade of more fixed edges.

7. Benchmarks

The first example is a numerical method for solving a differential equation (DE) with 11 nodes. The node operations are either multiplications or ALU-type operations (comparison, addition, subtraction). As a second example from the domain of high-level synthesis, we choose an elliptical wave filter (EWF, task graph see Fig. 8) with 34 nodes (8 multiplications and 26 additions).

Finally, a complete video-codec using the H.261 norm is optimized.

These examples are meant to demonstrate the general applicability of our method for practical problems; given other problem instances, or additional constraints, we can adapt our algorithm.

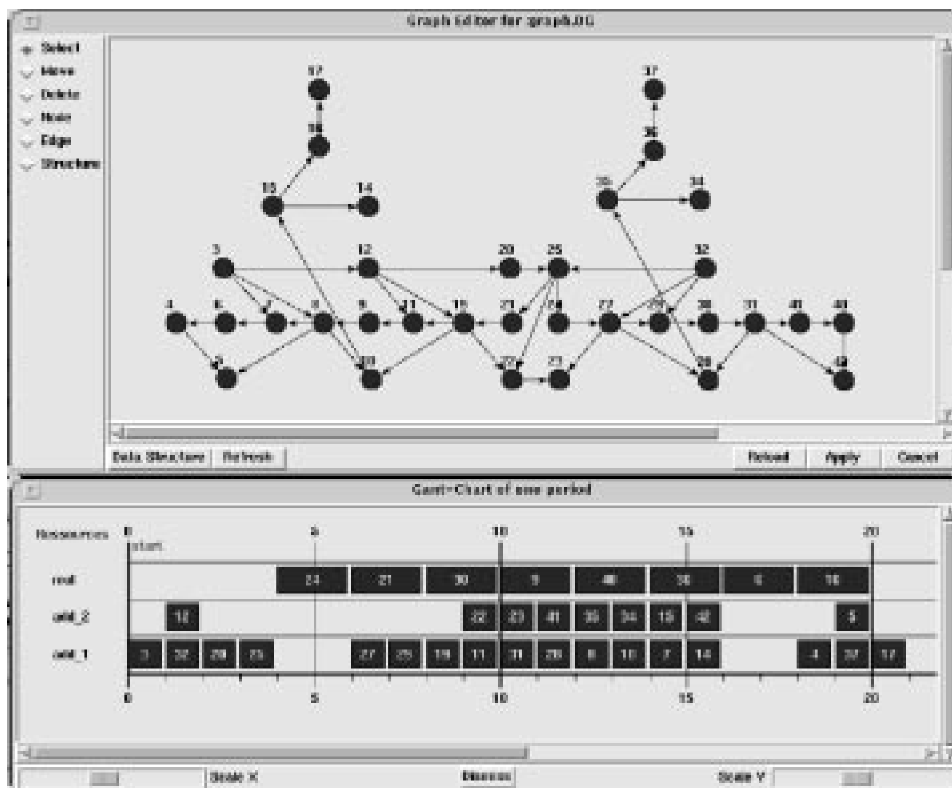

Figure 8. Problem graph and schedule of an elliptic wave filter, $h_t = 21$, one multiplier, two adders required (EWF, acyclic case).

7.1. DE and EWF Benchmarks

For the DE benchmark, the module library contains two hardware modules (box types): an array-multiplier and a module of type ALU that realizes all other node operations. For a word-length of n=16 bits, we assume a module geometry of 16×1 cells for the ALU module, and of 16×16 cells for the multiplier. Furthermore, let the execution time of an ALU node take one clock cycle and of a multiplication 2 clock cycles on our target chip.

For different resource constraints (maximal number of simultaneously active modules) shown in Table 1, we obtained schedules using a resource constrained scheduler, e.g., list scheduling, or using an ILP solver. Each schedule (*FixedS*) yields a test case for which the container size is minimized (*MinA*). Also shown are the number of iterations of the branch-and-bound procedure and the CPU-time needed for finding a solution.

The reported optimization times were measured as the CPU-times on a SUN-Ultra 30 architecture.

For the DE benchmark, it turns out that a chip of 32×32 freely programmable cells is necessary to obtain a latency of 6 clock cycles. As this turns out to be the longest path in the task graph, there does not exist any faster schedule. A reduction in size is possible if only one multiplication needs to be carried simultaneously, leading to a chip size of 17×17, and requiring 13 clock cycles. The minimal chip size was found using a binary search procedure for solving the SPP, with typically less than 3 applications of the OPP procedure.

For the EWF example, it can be seen that a 32×32 chip of freely programmable cells is necessary to obtain a latency of 16 clock cycles. A reduction in size is possible if only one multiplication needs to be carried out simultaneously, leading to a chip size of 18×18, and requiring 20 clock cycles. With a latency constraint of 27 clock cycles, the minimal chip size obtained is 17×17 cells, with a maximum degree of parallelism of one multiplication and one addition. Results for this benchmark are reported in Table 2.

7.2. Video-Codec

Figure 9 shows a block diagram of the operation of a hybrid image sequence coder/decoder. The purpose of the coder is to compress video images using the

Table 1. Computational results for optimizing reconfigurations for the DE benchmark

Test	Container sizes h_t	h_x	h_y	(#iterations/ CPU-time) (s)	Max. resources *multiplier/ALUs*
1	6	32	32	305/0.58	3/2
2	7	32	32	206/0.42	2/2
3	13	17	17	65/0.19	1/2
4	13	17	17	4/0.03	1/1
5	17	16	16	1/0.02	1/1

Table 2. Computational results for optimizing reconfigurations for the Elliptical Wave Filter

Test	Container sizes h_t	h_x	h_y	(#iterations/ CPU-time) (s)	Max. resources *multiplier/ALUs*
1	16	32	32	306/2.89	2/3
2	20	18	18	206/1.42	1/2
3	27	17	17	22/0.99	1/1

H.261 standard. In this device, transformative and predictive coding techniques are unified. The compression factor can be increased further by a predictive method for motion estimates: blocks inside a frame are predicted from blocks of previous images.

The blocks of the operational description in Figure 9 possess the granularity of more complex functions. However, this description contains no information corresponding to timing, architecture, and mapping of blocks onto an architecture.

Figure 10 shows a problem graph G of the video-codec in Figure 9. The problem graph contains a subgraph for the coder and one subgraph for the decoder.

For realizing the device, we have a library of three different modules. One is a simple processor core with a (normalized) area requirement of 625 units (25×25 cells, normalized to other modules in order to obtain a coarser grid) called PUM, denoted by "P" in Table 3. Secondly, there are two dedicated special-purpose modules: a block matching module (BMM, "B" in Table 3) that is used for motion estimation and requires $64 \times 64 = 4096$ cells; and a module DCTM ("D" in Table 3) for computing DCT/IDCT-computations, requiring $16 \times 16 = 256$ cells.

Again, the optimization was carried out for different latency constraints. Two instances of particular interest are shown in Table 3.

It can be concluded that solution 1 is preferable, since it provides better performance at equal cost. This conclusion could not be drawn from a resource constrained scheduling procedure that obtained the schedule of the first problem using an ILP solver for a resource constraint of 2 PUMs, 1 BMM, and 1 DCTM, and of 1 PUM, 1 BMM, and 1 DCTM in the second case. It should be noted that even better results may be obtained if also the schedule is determined by our packing algorithm. These *FindS* problems will be dealt with in a forthcoming paper.

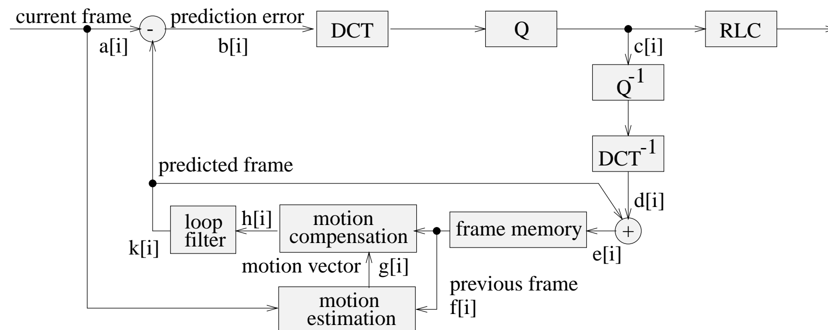

Figure 9. Block diagram of a video-codec (H.261).

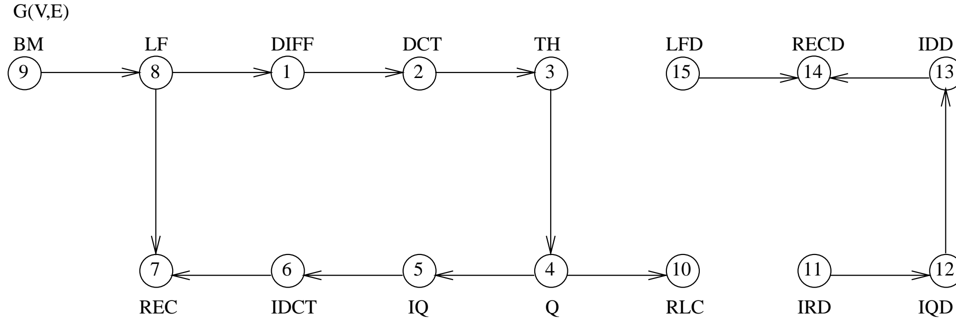

Figure 10. Problem graph G of the video-codec in Figure 9.

8. Conclusions

We describe a new general approach to solving dynamic hardware reconfiguration problems as more-dimensional packing problems. Using the concept of packing classes, new methods for obtaining fast lower bounds, and branch-and-bound techniques, we obtain an algorithm that solves these more-dimensional packing problems to optimality. Several practical benchmark instances as described here provide some evidence of the general applicability of our approach. Clearly, more complicated test instances only increase the necessity for our approach of using powerful tools from discrete optimization.

The presented methodology may be considered as a step towards prototyping communicating piecewise regular algorithms (Tei93; Pla97) in hardware using reconfigurable FPGA technology. The main advantages of such a methodology are 1) the reuse of resources in such computation intensive algorithms in time and space, and of course, 2) the programmability of the target architecture while being able to guarantee (near) real-time performance.

Concerning future research, we consider important extensions arising from requiring weaker time constraints for the tasks, such that only precedence constraints are given when optimizing the layout, instead of a fixed schedule. This is also a natural issue in the context of more-dimensional packing, where we may have constraints for the way of stacking boxes. For this purpose, we have to guarantee that the complement of a graph G_i in a packing class has a transitive orientation that contains the given partial order.

Another variation arises for instances where it may be possible to reuse parts of a given layout for similar tasks, such that some reconfiguration times for tasks can be

Table 3. Computational results for optimizing reconfigurations for the Video-Codec

Test	Container sizes h_t	h_x	h_y	(#iterations/ CPU-time) (s)	Max. resources $P/B/D$
1	59	89	89	35/0.11	2/1/1
2	61	89	89	40/0.18	1/1/1

saved. We hope to deal with this issue by treating problems of this type as knapsack problems with additional set cover constraints.

Notes

1. Here, each module is modeled by a cuboid. Such simplification, obtained by taking the axis-parallel hull over the 3D polytope, makes the following placement problem easier.
2. A static memory allocation may be deduced directly from the static placement.

References

1. Atmel. *AT6000 FPGA configuration guide*. Atmel Inc. [Atm]
2. J. E. Beasley. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33:49–64, 1985. [Bea85]
3. O. Diessel and H. El Ghindy. Partial FPGA rearrangement by local repacking. Technical Report 97-08, Dept. of Comp. Sci and Software Eng., Univ. of Newcastle, Australia, September 1997. [DG97a]
4. O. Diessel and H. El Ghindy. Run-time compaction of FPGA designs. In *Proc. of FPL'97—the 7th Int. Workshop on field-programmable logic and applications*, pages 131–140, Berlin, 1997. [DG97b]
5. S. P. Fekete and J. Schepers. A new exact algorithm for general orthogonal d-dimensional knapsack problems. In *Algorithms – ESA '97*, volume 1284, pages 144–156, Springer Lecture Notes in Computer Science, 1997. [FS97]
6. S. P. Fekete and J. Schepers. New classes of lower bounds for bin packing problems. In *Integer programming and Combinatorial Optimization (IPCO'98)*, volume 1412, pages 257–270, Springer Lecture Notes in Computer Science, 1998. [FS98a]
7. S. P. Fekete and J. Schepers. On more-dimensional packing I: Modeling. Technical Report 97-288, Angewandte Mathematik und Informatik, Universität Köln, Available at http://www.zpr.uni-koeln.de/~paper, 1998. [FS98b]
8. S. P. Fekete and J. Schepers. On more-dimensional packing II: Bounds. Technical Report 97-289, Angewandte Mathematik und Informatik, Universität Köln, Available at http://www.zpr.uni-koeln.de/~paper, 1998. [FS98c]
9. S. P. Fekete and J. Schepers. On more-dimensional packing III: Exact algorithms. Technical Report 97-290, Angewandte Mathematik und Informatik, Universität Köln, Available at http://www.zpr.uni-koeln.de/~paper, 1998. [FS98d]
10. A. Ghouilà-Houri. Caractérization des graphes non orientés dont on peut orienter les arrêtes de manière à obtenir le graphe d'une relation d'ordre. *C.R. Acad. Sci. Paris*, 254:1370–1371, 1962. [GH62]
11. P. C. Gilmore and A. J. Hoffmann. A characterization of comparability graphs and of interval graphs. *Canadian Journal of Mathematics*, 16:539–548, 1964. [GH64]
12. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979. [GJ79]
13. M. C. Golumbic. *Algorithmic graph theory and perfect graphs*. Academic Press, New York, 1980. [Gol80]
14. E. Hadjiconstantinou and N. Christofides. An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *European Journal of Operations Research*, 83:39–56, 1995. [HC95]
15. C.-H. Huang and J.-Y. Juang. A partial compaction scheme for processor allocation in hypercube multiprocessors. In *Proc. of 1990 Int. Conf. on Parallel Proc.*, pages 211–217, 1990. [HJ90]
16. G. M. Megson and D. Comish. *Systolic Algorithm Design Environments (SADEs)*, chapter 9, Transformational Approaches to Systolic Design, pages 205–239. 1994. [MC94]
17. T. Plaks. *Multidimensional Piecewise Regular Arrays.* PhD thesis, Chalmers University of Technology, Göteborg, Sweden, April 1997. [Pla97]
18. J. Schepers. Exakte Algorithmen für orthogonale Packungsprobleme. Technical Report 97-302, Doctoral thesis, Angewandte Mathematik und Informatik, Universität Köln, 1997. [Sch97]

19. J. Teich. *A Compiler for Application-Specific Processor Arrays*. Shaker (Reihe Elektrotechnik). Zugl. Saarbrücken, Univ. Diss, ISBN 3-86111-701-0, Aachen, Germany, 1993. [Tei93]
20. J. Teich, S. F. Fekete, and J. Schepers. Optimizing dynamic hardware configurations. Technical Report 98-336, Angewandte Mathematik und Informatik, Universität Köln, Available at http://www.zpr.uni-koeln.de/~paper, 1998. [TFS98]
21. J. Teich, L. Thiele, and L. Zhang. Partitioning processor arrays under resource constraints. *Int. Journal on VLSI and Signal Processing Systems*, 17(1):5–20, 1997. [TTZ97]
22. Xilinx. XC6200 field programmable gate arrays. Technical report, Xilinx, Inc., October 1996. [Xil96]