Digital Systems Design Using Constraint Logic Programming

Szymanek, Radoslaw; Gruian, Flavius; Kuchcinski, Krzysztof

Published in:
Proceedings of the Practical Application of Constraint Logic Programming (PACLP) Conference

2000

Link to publication

Citation for published version (APA):
Szymanek, R., Gruian, F., & Kuchcinski, K. (2000). Digital Systems Design Using Constraint Logic Programming. In *Proceedings of the Practical Application of Constraint Logic Programming (PACLP) Conference*

Total number of authors:
3

DIGITAL SYSTEMS DESIGN USING CONSTRAINT LOGIC PROGRAMMING

RADOSLAW SZYMANEK, FLAVIUS GRUIAN, KRZYSZTOF KUCHCINSKI

Lund University, Dept. of Computer Science, Lund, SWEDEN

Radoslaw.Szymanek@cs.lth.se

Abstract. This paper presents an application of finite domain constraint logic programming methods to digital system synthesis problems. The modeling methods address basic synthesis problems of high-level synthesis and system-level synthesis. Based on the presented models, the synthesis algorithms are then defined. These algorithms aim to optimize digital systems with respect to cost, power consumption, and execution time.

Keywords. digital systems design, constraint logic programming

1. Introduction

Recently, the system-on-chip concept has been proposed which is integration of many different system parts on a single chip. Current developments in the VLSI technology make it possible to build complex systems consisting of millions of components thus there is the potential of reducing cost while improving many design parameters, such as performance and reliability. The design process of a system-on-chip requires new design methods which can help the designer to explore the design space early in the design process.

The typical design problems which have to be solved during system synthesis include system partitioning, allocation of resources, assignment of basic system parts into allocated components and scheduling (Eles et al., 1997). These design steps are performed sequentially with possible iterations when the results are unsatisfactory. Many heterogeneous constraints have to be taken into account during the design process. In addition to performance and area constraints, we often would like to consider memory or power consumption constraints. These constraints are usually difficult to include in automatic design methods and they have to be handled separately, which

This work was supported in part by the Wallenberg Foundation project “Information Technology for Autonomous Aircraft” and the Foundation for Strategic Research, Integrated Electronic Systems program.

reduces the chance of reaching good final results.

System synthesis can be defined as an optimization problem. The system is modeled as a set of constraints on fabrication cost, number of components, timing, and the goal is to find a solution which minimize a given cost function. For example, we would like to implement a given functionality on a number of processors while minimizing the execution time. This optimization will provide the fastest possible implementation with the available resources. Other optimization criteria, such as cost, power consumption or a combination of them, can be also considered.

The synthesis process of a digital system starts with defining the system in an abstract way, usually by describing its functionality. This abstract representation is then refined and it becomes a description of the hardware (e.g. ASIC) and possibly some software modules, which together implement the system. We believe that the use of Constraint Logic Programming (CLP) can improve both quality and efficiency of system design steps. Below we will present a number of examples where CLP has been used for high-level and system-level synthesis.

This paper is organized as follows. Section 2 defines the computational model for our framework and presents the basic finite domain modeling techniques. Both the high-level and system-level synthesis approaches using finite domain constraints are then presented in section 3 and 4 together with a discussion on experimental results. Finally, the conclusions are presented in section 5.

2. Basic Modeling Techniques

In this paper, we consider that digital systems are modeled using graphs where the nodes represent computations and the arcs data dependencies between them. The computation nodes can be either simple operations, such as additions and multiplications, or complex tasks representing, for example, signal or image processing algorithms. Figure 1 depicts an example of such graph. Each node in this graph, (T_1, T_2, T_3, T_4, T_5, T_6), corresponds to a computation. An arc connects two nodes if and only if there is a data dependency between the corresponding nodes. There is an arc in the graph connecting a node with another node if the first node (sender) activates communication to another node (receiver). For example, the arc between nodes T_1 and T_3 models the communication between these nodes.

In general, the functionality represented by a node is activated when the communication on all its input arcs took place. The graph models a single computation as a partial order of nodes' executions. The graph is acyclic but an implicit iterative computation is assumed. An example of such graph is depicted in Figure 1, each computation starts from the execution of nodes T_1 and T_2 and finishes with execution of node T_6.

Each node has a deterministic execution time. A communication time is also deterministic. Both the execution time and communication time can be decided before the model is built. Communication between nodes is allowed only at the beginning or at the end of the node execution.

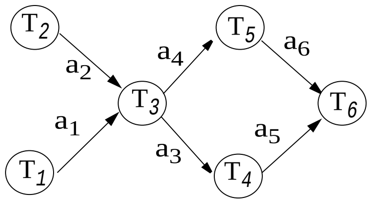


Figure 1. An example of a computation graph

2.1 Finite Domain Constraints Model

The graph introduced above is modeled as a set of finite domain constraints. The modeling constraints enforce the node ordering and a realistic resource access. We define first the variables which represent the basic parameters of nodes and resources and then introduce the constraints on these variables.

A node is modeled as a 3-tuple $T = (\tau, \delta, \rho)$ where τ, δ and ρ are finite domain variables representing the activation time of the node (τ), the execution time of the node (δ), and the resource used to execute the node (ρ).

For example, for the graph depicted in Figure 1, the following definition of the node T_1 can be made:

$$T_1 = (\tau_1, \delta_1, \rho_1), \text{ where } \tau_1::0..50, \delta_1::[2,5], \rho_1::3..4.$$

Node T_1 is activated sometime between 0 and 50, its execution time is either 2 or 5 time units and it uses either resource 3 or 4. A constraint solver assigns then single value from domains of related finite domain variables τ, δ, and ρ, providing a possible solution. For example, $T_1 = (0, 2, 4)$.

A single node specification does not include graph information on the execution order between nodes. This is modeled as inequality constraints. If there is an arc from a node T_i to a node T_j in the graph then the following inequality constraint is defined:

$$\tau_i+\delta_i \le \tau_j$$

Two arbitrary nodes can not, in general, use the same resource at the same time. This is usually expressed through disjunctive constraints imposed on each pair of nodes (Kuchcinski, 1997). These constraints have to be defined for all nodes which can be executed in parallel, and thus, we can avoid overlapping execution of tasks. This leads to the creation of $\frac{n \cdot (n-1)}{2}$ constraints, in the worst case. The work presented in this paper uses CHIP 5 constrained logic programming system (Beldiceanu et al., 1997), therefore we use the global constraint `diffn/1` to disallow overlapping execution of tasks. `Diffn/1` makes use of a rectangle interpretation of nodes. A node, represented by a 3-tuple, can be interpreted as a rectangle in the time/resource space having the following coordinates $((\tau_i, \rho_i), (\tau_i, \rho_i+1), (\tau_i+\delta_i, \rho_i), (\tau_i+\delta_i, \rho_i+1))$. This is not a limitation in practice since the nodes, which need more resources, can be modeled as several nodes. The `diffn/1` constraint takes as an argument a list of *n*-dimensional rectangles and ensures that for each pair of *i*, *j* ($i \neq j$) of *n*-dimensional rectangles, there exist at least one dimension where *i* is after *j* or *j* is after *i*. The *n*-dimensional rectangle is defined by a tuple $[O_1, ..., O_n, L_1, ..., L_n]$, where O_i and L_i are called the origin and the length of the n-dimensional rectangle in the *i*-th dimension respectively. The `diffn/1` constraint can replace the set of disjunctive constraints by imposing that all rectangles representing nodes, such as $Rect_i = [\tau_i, \rho_i, \delta_i, 1]$ and $Rect_j = [\tau_j, \rho_j, \delta_j, 1]$, can not overlap.

A node execution time is deterministic but it is not the same for all resources. For example, different microprocessors require different execution time for the same task depending on the clock frequency and architecture of processor. Our model makes it possible to define the node execution time as a finite domain variable capturing several execution time values. The relation between δ_i and ρ_i can be expressed using the `element/3` constraint. The execution time, δ_i, for a node T_i on a resource ρ_i, is defined by the following constraint:

```
element(ρ_i, [τ_i1, τ_i2, ..., τ_iN], δ_i).
```

This constraint enforces a finite relation between the first and the third variable. The finite relation is given by the vector of values passed as the second argument.

2.2 Redundant Constraints

The formulation presented above fully describes the graph model of the digital system and can be directly used for synthesis. However, in most of the developed applications we used redundant constraints to improve the constraint propagation. The most important redundant constraint we used is `cumulative/8`.

The `cumulative/8` constraint has been defined in CHIP to specify requirements on the tasks which need to be scheduled on a limited number of resources. It expresses the requirement that, at any time instant, the corresponding total of the resources for the tasks does not exceed a given limit. The following four parameters are used: a list of the start times of tasks O_i, a list of durations D_i of tasks, a list of the amount of resources R_i required by the task, and the upper limit of the amount of resources UL. All parameters can be either domain variables or integers. Formally, `cumulative/8` enforces the following constraint:

$$\forall i \in \left[\min_{1 \le j \le n}(O_j), \max_{1 \le j \le n}(O_j + D_j)\right] : \sum_{k : O_k \le i < O_k + D_k} R_k \le UL$$

where n is the number of tasks, while *min* and *max* are the minimum and maximum values in the domain of the variable respectively.

The cumulative constraint can be used to describe two types of constraints. In the first formulation, O_i is replaced by τ_i, D_i by δ_i and, finally, R_i by 1. This models the task allocation and scheduling on the limited number of resources represented by UL. The second formulation represents the bin packing problem: O_i is replaced by ρ_i, D_i is always 1 and finally R_i is replaced by δ_i. The variable UL is constrained to the value lower or equal the execution time of the graph. Please note that the first formulation uses only τ_i and δ_i while the second one only ρ_i and δ_i. Therefore they are able to offer different types of propagation.

We use also another redundant constraint `precedence/5`. This constraint takes into account, in addition to precedence constraints expressed by inequalities, the resource limitations on which jobs can be scheduled. This redundant constraint gives better propagation than inequalities alone.

3. High-Level Synthesis

High-Level Synthesis (HLS) refers to that step in the synthesis of a digital system where a functional (behavioral) specification of a system or sub-system is transformed into a Register-Transfer Level (RTL) representation. This RTL representation will be later implemented as hardware.

3.1 Introduction

During HLS one must decide the type and number of resources (adders, multipliers, ALUs, registers) needed, the right time to perform each operation, and resource which will perform it. These three problems are referred to as resource allocation, operation scheduling, and binding. Each of these problems has been proven to be NP complete. At high-level, the data operations are simple arithmetic operations such as additions, multiplications, and comparisons. Most of the digital systems are synchronous, meaning the operations are performed at well established moments triggered by a global signal named clock. The time units characterizing the delays of the operations are, at this level, always related to the clock signal frequency. The right times to perform certain operations are thus expressed in clock cycles. HLS usually targets minimization of execution time or resources cost, but other parameters, such as, power consumption, need to be addressed sometimes. Depending on the user specifications, HLS can be constrained by the availability of different resources or execution time.

The input to the HLS process, the functional specification, can be represented as a control dataflow graph (CDFG). A CDFG and the graph presented in Figure 1 are very much alike. Therefore, the modeling techniques introduced in the previous section are directly applicable. The nodes in CDFG represent simple operations, such as additions, multiplications, comparisons, while the arcs describe the conditional or non-conditional data flow between different operations. For clarity we will consider here only data flow graphs (DFG), although control information can be handled as shown in (Kuchcinski, 1997).

The constraints described in the previous section form the basic model for the general HLS problem. Additional constraints for modeling application specific issues, such as, register assignment and power consumption minimization, are described in the following part.

Registers are assigned to input, output, and temporary variables during high-level synthesis. To allow register sharing, the lifetimes of the variables, representing the period that the variable occupies a register, are computed and a related analysis determines register assignment. The lifetimes of the variables are modeled in our approach using rectangles which span on time axes over define-use time of the variable. This resembles a definition of variable lifetimes used in left-edge algorithm (see, for example (Beldiceanu et al., 1997)). Defining the lifetimes of variables as rectangles provides a natural way to use both `diffn/1` and `cumulative/8` constraints (Kuchcinski, 1998).

3.2 Advanced Features

A number of useful extensions to the basic formulation introduced in section 2 can be defined to consider special features such as pipelined components, chaining, algorithmic pipelining, and conditional execution. They are discussed in this section.

Modeling *pipelined components* can be accomplished by defining 3-dimensional rectangles, in which the third dimension represents subsequent stages of the component. For example, Figure 2 depicts a design which uses a two stage pipelined component. The first stage, S_1, is represented by the cube of height 1 located between τ_0 and τ_1 and originated at coordinate 0 in the third dimension. The second stage, S_2, is represented by the cube of height 1 located between τ_1 and τ_2 and originated at coordinate 1 in the third dimension. All non-pipelined operations, such as the operation Op_j depicted in Figure 2, have heights of 2 and therefore can not be placed together with neither the first, nor the second stage of the pipelined sub-task. “Packing” of operations represented by 3-dimensional rectangles enables placement of the stage one and two of different operations at the same resource/time location since they do not overlap in the third dimension. Other non-pipelined operations can not collide with the pipelined ones, since they have the height 2. The finite domain constraint definition for the example in Figure 2 is the following:

$\texttt{diffn([[}\tau_{i,S1},\rho_i,0,\delta_{i,S1},1,1\texttt{], [}\tau_{i,S2},\rho_i,1,\delta_{i,S2},1,1\texttt{], [}\tau_j,\rho_j,0,\delta_j,1,2\texttt{]])} \wedge \tau_{S1} + \delta_{S1} = \tau_{S2}$.

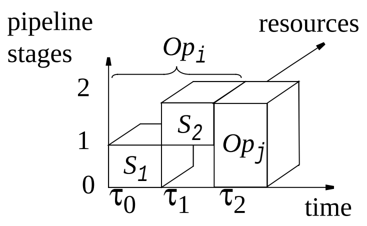


Figure 2. Resource sharing constraints for pipelined components.

This formulation can be extended into n-dimensions, if there are more different pipelined components.

Chaining refers to the high-level synthesis technique of scheduling two or more data-dependent operations during the same clock cycle. It is achieved by connecting the output of one functional unit directly to the input of the following functional unit without storing a temporary value in a register. During the same clock cycle the functional unit can not be reused by another operation because it still propagates results which are stored at the end of this clock cycle. This introduces additional constraints on resource sharing in chaining.

Figure 3 illustrates the basic idea of modeling chaining using finite domain constraints. Three dimensional rectangles are used for this purpose. The three dimensions are used to represent resources, clock cycle and a relative position of an operation within a clock cycle, called here a step. Each clock cycle can be filled with several operations as long as they fit within the limits of the clock cycle (the rectangle boundaries). Two `diffn/1` constraints are used to impose basic requirements on the implementation. The first `diffn/1` constraint specifies the structure depicted in Figure 3 and is defined by the following constraint:

$\texttt{diffn([[}\tau^s_i,\rho_i,\tau^c_i,\delta_i,1,1\texttt{] , ..., [}\tau_j,\rho_j,\tau^c_j,\delta_j,1,1\texttt{]]).}$

The second constraint is used to forbid situations when the same resource is shared within the same clock cycle. It is defined using a projection of rectangles on the resource/clock cycle surface as a `diffn/1` constraint on two dimensional rectangles as given below.

$\texttt{diffn([[}\tau^c_i,\rho_i,1,1\texttt{], ..., [}\tau^c_j,\rho_j,1,1\texttt{]]).}$

The relation between previously introduced start time of an operation, τ_i, and the two new parameters τ^c_i and τ^s_i is defined for every operation by the following equation:

$\tau_i = \tau^c_i * N + \tau^s_i$

where *N* is the number of steps in the clock cycle.

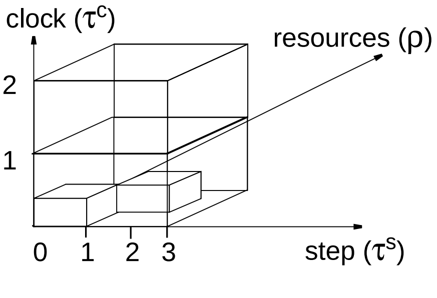


Figure 3. Rectangle representation of chaining.

Pipelining a data-flow graph is an efficient way of accelerating a design (Kuchcinski, 1997). It introduces, in fact, new constraints on the location of rectangles. This method is well known in computer architecture area, where two dimensional reservation tables are used for pipeline analysis. This approach is compatible with our methodology. Introducing an *n* stage pipeline of the initiation rate of *k* time units is equivalent to a placement of *n* copies of existing rectangles, starting at positions *k, 2·k, 3·k,* etc. This prevents placing operations in forbidden locations, which are to be used by subsequent pipeline instances. Since the operation parameters are defined by domain variables, the copies of the current rectangles do not define final operation positions but these positions will be adjusted during an assignment of values to domain variables.

The following constraints define two-stage pipeline for two operations Op_i and Op_j, depicted in Figure 4, with initiation rate *k*:

$\tau_{i,k} = \tau_i + k, \ \tau_{j,k} = \tau_j + k,$

$\texttt{diffn}([\ [\tau_i,\rho_i,\delta_i,1],\ [\tau_j,\rho_j,\delta_j,1],\ [\tau_i,k,\rho_i,\delta_i,1],\ [\tau_j,k,\rho_j,\delta_j,1]\ ]).$

The graphical, rectangle representation of these constraints is depicted in Figure 4

The rectangle based resource constraints can be easily extended to handle *conditional* nodes. The conditional node is executed only if the conditions assigned to its input arcs are true. The value of this condition can not be statically determined and therefore we need to schedule both true and false execution cases. The presented formulation of the resource constraints, which uses 2-dimensional rectangles in the time/resource space, needs to be extended to cover conditional execution. The main idea of representing conditional nodes is to extend rectangles into higher dimensions. In principle, one more dimension is used for each new condition. The conditional nodes start in the third dimension either at 0 or 1, depending on the condition, and have height 1. They can share the same time/resource place since they can be placed “one on top of the other”. Other computational nodes can not be placed together with conditional ones since in this formulation they have height 2.

3.3 Power Consumption Minimization

System power consumption is another important design issue. For CMOS digital circuits the power consumption depends mainly on the supply voltage (V_{dd}), clock frequency (f), and switched capacitance (α*C):

$$P_{switching} = \alpha \cdot C \cdot f \cdot V_{dd}^2$$

Considering that the voltage and frequency are usually fixed as design requirements, and the capacitance is determined by the technological process, the only way to minimize power consumption is by minimizing switching activity α. The switching activity of a node is a measure

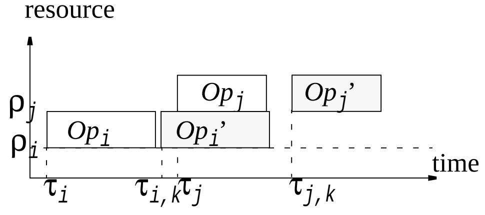


Figure 4. A graphical representation of the resource constraint for algorithmic pipelining.

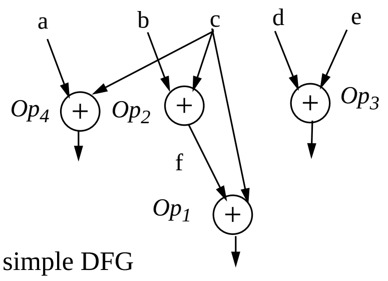


Figure 5. A simple DFG

of how much a certain node in a CMOS circuit has to switch from 1 to 0 to compute something. In other words, if the signals in a circuit are changing as little as possible during computation then the circuit will consume less power. With this observation, one could carefully schedule the order of operations on each resource such that the data is changing as little as possible at the inputs and inside the resource. Briefly, binding and scheduling influence the values and the sequence of signals applied to each resource

Consider simple DFG shown in Figure 5. There are several possible schedules and bindings for this graph using two adders. Each solution yields different switching activities, thus different power consumptions. Two of these possible bindings and schedules are depicted in Figure 6. First let us consider that each operation of the DFG is executed on its own functional unit yielding a switching activity that can be calculated using signal probabilities or computed by a fast RT level simulator. Let us call this switching activity the unbounded switching ($Sw_{0,i}$) for operation Op_i. In general, during high-level synthesis, several operations will be bound to the same resource determining the switching activity of the design. For example, if on a certain resource, operations Op_i and Op_j are executed in that order, the switching activity cannot be computed as a simple sum $Sw_{0,i} + Sw_{0,j}$ since the switching produced by Op_j is dependent on the signal values produced by the previous operation Op_i. It is closer to reality to consider switching as $Sw_{0,i} + Sw_{i,j}$ where $Sw_{i,j}$ is the relative switching between operations Op_i and Op_j. The relative switching activity describes the bit correlation of two signals and is defined as the number of different bit values of the two signals (Raghunathan et al, 1994).

What we finally need to minimize is exactly the total switching yielded by certain sequences of operations on their resources. For that we have to know the sequence of operations on each resource which can be obtained in CHIP using the `cycle/n` constraint. Actually we have to deal with a slightly modified travelling salesman problem (TSP) (Reeves, 1993) where there are as many cycles as there are resources. The nodes in the graph are the operations, Op_i, and the weights assigned to the arcs in the graph are the relative switching values. For example $Sw_{i,j}$ is the weight of the arc going from Op_i to Op_j. The unbounded switchings can be seen as arcs from a dummy node representing a resource i to a normal operation node (see Figure 8).

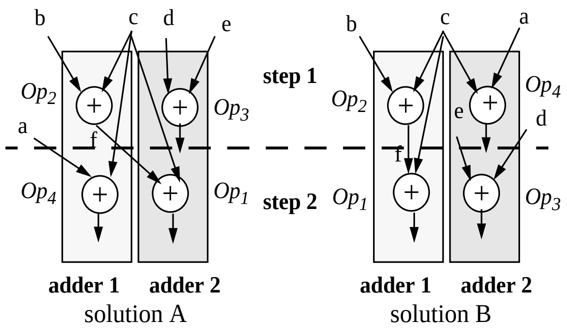


Figure 6. Two possible schedules with different bindings for the DFG in Figure 5

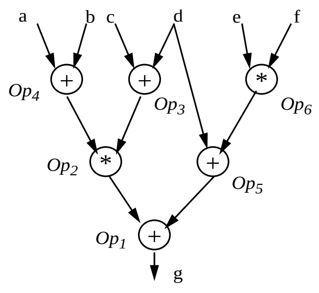


Figure 7. Another simple DFG

3.4 Example

For the DFG example depicted in Figure 7, a possible design which uses three resources, two adders and one multiplier, is described in Figure 8. The operations Op_4, Op_5, Op_1 are executed on resource *1* in this order, Op_6, Op_2 on resource *2* in this order and Op_3 on resource *3*. The switching activity is the sum of the weights of the arcs involved:

$$Sw = (Sw_{0,4} + Sw_{4,5} + Sw_{5,1})+(Sw_{0,6} + Sw_{6,2}) + Sw_{0,3}$$

Observe that the arcs closing the cycles, back to the dummy nodes, have weight zero. In particular we used `cycle/9` to group the *N* operations in sets for each resource:

cycle(R, [S_1, S_2, ..., S_R, S_{R+1}, ..., S_{R+N}], [0, ..., 0], MinimalCycle-Length, MaxCycleLength, [1, 2, ..., R], unused, [1, 2, ..., R, ρ_1, ρ_2, ..., ρ_N], [unused, ..., unused, τ_1, τ_2, ..., τ_N]).

where *R* is the number of resources used, S_i, $1 \le S_i \le N$, is the domain variable indicating an immediate successor of operation *i* on the specific resource, and τ_i, ρ_i are the same as defined before. Having the ordering of operations on each resource, it is easy to compute the overall switching activity which is the objective function to be minimized. To extract exactly the switching values needed for the computation of this function, we used an additional `cycle/11` constraint. For more information please refer to (Gruian et al., 1998).

3.5 High-Level Synthesis - Experimental Results

We carried out several experiments using a prototype of the synthesis system implemented in CHIP 5, a constrained logic programming system (Cosytec, 1996). This is a Prolog system with constraints solvers over finite and rational domains. In the experiments, we used only the finite domain solver. All experiments have been run on a 50 MHz SPARCCenter 2000 machine.

Four HLS benchmarks have been selected for experiments: differential equation (DIFFEQ), fifth order elliptic wave filter (EWF), AR lattice filter (AR) and discrete cosine transform (DCT). The benchmarks varies much in complexity. The simplest example, DIFFEQ, has only 11 operations

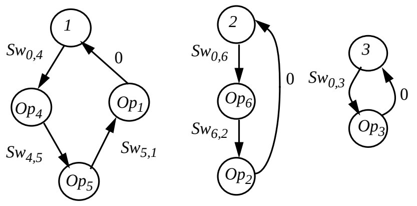


Figure 8. Example of cycle generation for the DFG

and 16 variables, the AR benchmark has 28 operations and 47 variables, the EWF has 34 operations and 41 variables, and the DCT has 48 operations and 48 variables. All benchmarks use adders and multipliers. We assumed that addition requires one and multiplication two clock cycles. This assumption is realistic as indicated in many research reports.

We evaluated our modeling method by making synthesis, using different design styles, for existing benchmarks (Kuchcinski, 1998). Each example has been synthesized using multicycle components (two-cycle multipliers), two-stage pipelined multipliers and chaining. In chaining, different lengths of the clock cycle has been tried.

The optimal assignment of functional units and the schedule was obtained for all examples in several seconds. This result is very surprising since known ILP based synthesis results, e.g. (Lee et al., 1989), usually produce the same solutions in tenths of seconds. Even heuristic solutions which can not guarantee optimal results usually require several seconds to come out with a solution. For example, simulated annealing based algorithm used in SALSA II needs up to 13 seconds for the DCT benchmark to produce the solution (Rhinehart et al., 1993). The register assignment optimization performed after the synthesis requires a fraction of a second.

Finite domain constraints offer a convenient way of combining different design constraints and solve them in one framework. We achieved this when we combine register, functional units assignment, scheduling, and register assignment constraints into one synthesis step. This approach, as expected, provides better synthesis results even in cases when optimal results can not be computed due to the problem complexity. In many cases, the system has been able to generate lower number of registers for the same number of functional units and steps than the other reported approaches.

We have also synthesized the four examples using pipelining of the whole algorithm. We assumed first a two and then a three stage-pipeline. The synthesis algorithm optimizes a number of steps in the pipeline stage satisfying resources constraints. For DIFFEQ and EWF optimal results for different resource constraints have been obtained. For other two examples, in the situations of a low number of resources (AR- 2 adders and 4 multipliers and DCT- 4 adders and 4 multipliers), a partial search method had to be used to generate good solutions. In all other cases optimal results have been obtained.

To show the behavior of our low-power oriented modeling method, based on switching minimization, as described in 3.3, we used three benchmarks: DIFFEQ, AR, and DCT. For each of these we assumed different allocations of resources and examine how scheduling and binding influence the switching activity. For DIFFEQ, because of its simplicity, we could obtain optimal results, while for the other two we used credit-based partial search and obtain near optimal results. We found out that, in some cases, having a switching activity sensitive synthesis strategy may give up to 60% decrease in power consumption, for the same allocation. In worst cases, this decrease was 2% only. The power consumption can be decreased even more if the constraints on resources and deadline are relaxed (Gruian et al., 1998).

4. System-Level Synthesis

Given the specification of the system functionality, the main goal of system-level synthesis is to make decisions concerning the system architecture and the system implementation on this architecture. The functional specification of the system is compiled into a task graph. The graph introduced in section 2 is interpreted as a task graph where the nodes represent tasks and the arcs represent communications between them. Each task must be executed on a single processor, so for each task we need to reserve a time slot, code and data memory on the chosen processor. In our approach, we assume that there is no need for communication when two tasks are executed on the same processor since both tasks have access to the same local memory.

An architecture consists of processors and communication devices, such as busses and links. Figure 9 depicts an example target architecture which consists of four processors, P_1, P_2, P_3, and P_4, two links, L_1 and L_2, and a bus, B_1.

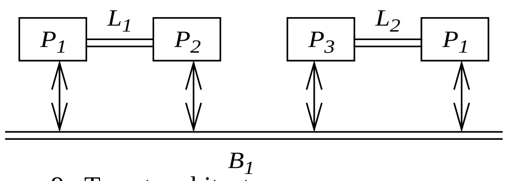


Figure 9. Target architecture

In our view, the goal of the system-level synthesis is to find an architecture with a minimal cost which can execute all tasks while fulfilling timing and memory constraints. The architecture is created from a set of components specified by the designer. The whole process is guided by the designer. The constraint system enforces the correctness of the solution by rejecting all the decisions which violate constraints.

4.1 System Modeling

The constraints taken into account in the presented synthesis system can be classified into two groups:

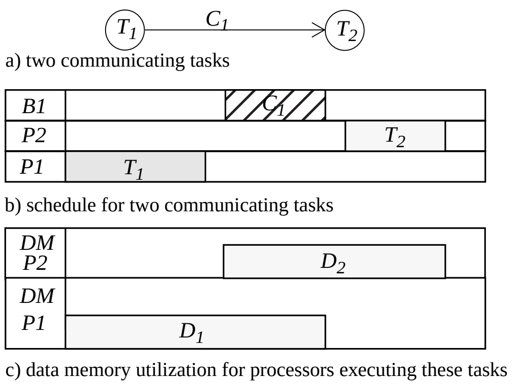
- timing constraints and
- resource constraints.

The data dependency constraints belong to the first group and they are modeled using inequalities, as presented in section 2.1. There are two kinds of data dependency between tasks. Indirect data dependency exists when two communicating tasks, for example T_1 and T_3, are executed on different processors. In this case, communication a_1 depends on task T_1 and task T_3 depends on communication a_1. Direct data dependency occurs when two communicating tasks are executed on the same processor. These two possibilities of data dependency are encoded using conditional constraints.

The problems of binding tasks to processors and communication to communication devices and scheduling them are modeled, as indicated in section 2.1, by `diffn/1` constraint. This constraint requires the task duration to be greater than zero. Since, in our model, some communications can be performed in zero time, using local memory, we have to distinguish them from tasks and other communications. The way of handling “disappearing” communication is by introducing a third dimension in the `diffn/1` constraint in addition to time and resource dimensions. These communications will have different values in the third dimension. This policy ensures that non-existing communications do not restrict the solution space.

Code memory is used to store programs implementing tasks. The amount of code memory needed to implement a task depends on the processor type, but it is fixed during the execution of the whole task graph. We used the reified version of the `sequence/5` constraint to obtain matrix ρ, where ρ_{mi} equals 1 denotes that m-th task is executed by i-th processor. Multiplication of two vectors c_i and ρ_i, where c_{im} denotes amount of code memory required to execute m-th task on i-th processor and ρ_i is the i-th column from matrix ρ, gives the overall utilization of the code memory on the i-th processor. This utilization must not exceed the available memory.

Data memory constraint is the most complex since data memory utilization changes during tasks’



T1
C1
T2
a) two communicating tasks
B1
C1
P2
T2
P1
T1
b) schedule for two communicating tasks
DM P2
D2
DM P1
D1
c) data memory utilization for processors executing these tasks

Figure 10. Data memory requirements

execution. Data are associated with communications and tasks. Each task requires fixed amount of data memory during the execution. Before we can start executing a task we need all input data of the task stored in a local memory therefore some of the data memory requirement appear before execution of the task. When task finishes its execution we need to reserve data memory for output data until it is consumed by successor task. In the case of transferring the data from one processor to the other we have to reserve memory on both processors during the transmission. Each time there is a need to reserve data memory this is done dynamically. The memory is freed when not needed any more. Dynamic allocation schema makes the handling of data memory much more difficult than handling code memory.

4.2 An Illustrative Example

Consider two tasks and the communication between them as depicted in Figure 10a, where task T_1 is executed on processor P_1 and task T_2 is executed on processor P_2. Communication C_1 is scheduled on bus B_1. The data transfer can occur between finishing time of task T_1 and starting time of task T_2 which is expressed by the following inequalities:

$$\tau_{t1} + \delta_{t1} \le \tau_{c1} \ \wedge \tau_{c1} + \delta_{c1} \le \tau_{t2}$$

Each communication results in two data requirements as depicted in Figure 10c. Processor P_1 must reserve data memory, denoted by D_1, for task T_1 from τ_{t1} until $\tau_{c1}+\delta_{c1}$, where τ_{c1}, δ_{c1} denote the start time and duration of the communication respectively. Processor P_2 reserves data memory for task T_2, denoted by D_2, from τ_{c1} until $\tau_{t2}+\delta_{t2}$. D_1 and D_2 have the same height denoting the memory size. Since tasks T_1 and T_2 are executed on different processors, the data memory during communication must be reserved on both processors. This results in higher overall data memory utilization than in the case when task T_1 and task T_2 are executed on the same processor.

For each processor, one cumulative constraint is created as depicted in Figure 11. The data requirement D_1 appears in the cumulative for both processors, P_1 and P_2, because both processors

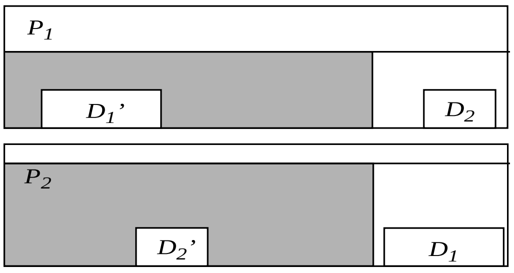


Figure 11. Data memory constraint

can execute task T_1. Task T_2 can also be executed on both processors, so D_2 exists in both cumulative constraints. Since processor P_1 executes task T_1, rectangle D_1 in the cumulative constraint for processor P_1, denoted by D_1', is placed in the dotted area and rectangle D_1 in the cumulative constraint for P_2 is placed outside dotted area. The same principle applies to task T_2 and its data requirement, D_2. The actual data requirements are represented by rectangles D_1' and D_2'.

In addition to `cumulative/8` we have to use conditional and `element/3` constraints in order to assure that there is only one D_1' and D_2' and following equalities hold:

$$\tau_{D1'} = \tau_{T1} \wedge \tau_{D1'} + \delta_{D1'} = \tau_{C1} + \delta_{C1} \wedge \tau_{D2'} = \tau_{C1} \wedge \tau_{T2} + \delta_{T2} = \tau_{D2'} + \delta_{D2'}$$

when T_1 and T_2 are executed on different processors or

$$\tau_{D2'} = \tau_{D1'} + \delta_{D1'} = \tau_{T2} \quad \wedge \tau_{T2} + \delta_{T2} = \tau_{D2'} + \delta_{D2'} \wedge \tau_{D1'} = \tau_{T1}$$

when T_1 and T_2 are executed on the same processor. Using this formulation, we can ensure that cumulative utilization of data memory depicted as rectangles in the dotted area does not exceed the available data memory.

4.3 Optimization Heuristic

The task assignment and scheduling are NP-complete problems. The inclusion of memory constraints makes the problem even more complex. We developed new heuristic which performs assignment and scheduling. The new heuristic takes into account memory constraints.

4.3.1 Parameters Estimations

A solution to a synthesis problem is an assignment of each task to a processor and a time slot. In addition, each communication task has to be assigned to a communication device and scheduled. A number of parameters are estimated to guide our heuristic during the process of finding good solution.

A distributed execution of the task graph results in bigger data memory requirement than execution on the single resource as discussed in section 4.2. Delaying the execution of not urgent tasks for the favor of the tasks which belong to the critical path decrease the schedule length but it increases data memory requirement. There are two conflicting goals. Either the schedule length is decreased or data memory requirement is decreased. Our heuristic tries first to schedule tasks from the critical path until the estimate of data memory utilization is below memory size. When the estimate of data memory utilization exceeds memory size then our heuristic aims at choosing a task which will lower the estimated data memory utilization. Since the actual data memory utilization depends on the schedule, it is difficult to know in advance exactly how the assignment of task T_j on processor P_i will influence the peak of data memory requirements on all processors.

We use two estimates of data memory utilization. First one is computed as a sum of ingoing communications of ready to execute tasks. These communications represent data that were produced by already scheduled tasks and have to be stored somewhere. This estimate is fast but not accurate because it does not take time into consideration. The second estimate is used when the first one cannot guarantee with high degree of probability that data memory will not be overused. It obtains more precise estimate of the upper bound of the data memory requirement. To have a more accurate estimate we use the lower bound of the schedule length denoted by E_m. The tightest possible schedule, defined by E_m, means the best possible parallelism and this can potentially result in the highest data memory requirement. In addition, we compute the latest possible start time for each task denoted by $\max(\tau_j)$. The latest possible start time of task T_j means that incoming data to this task will need to be stored for the longest possible time. Both E_m and $\max(\tau_j)$ are then used in the cumulative constraint to estimate data memory requirement. The second data memory estimate cannot guarantee that data memory will not be overused on one of the processors, but it is accurate enough too be useful.

Two kinds of measures, *UCM* (utilization of code memory) and *UTS* (utilization of processor time slots), are used by our heuristic. Based on them we decide on which processor P_i execute task T_j. *UCM* and *UTS* use lower bounds for used amount of code memory (*LCM*) and processor time units (*LTS*).

$UCM = \frac{LCM}{ACM}$, where *ACM* - available code memory

$UTS = \frac{LTS}{ATS}$, where *ATS* - available processor time units

Similar measures are defined for the situation when task T_j will be executed on processor P_i. These measures are denoted by UCM_{ij} and UTS_{ij}.

These two kinds of measures are used when computing the cost V_{ij} of implementing task T_j on processor P_i. The cost function uses, in addition, the amount of code memory (C_{ij}) needed to execute task T_j on processor P_i and T_{ij} which represents the time needed to execute task T_j on processor P_i.

$$V_{ij} = \begin{cases} \dfrac{C_{ij}}{LCM_{ij}} & \text{if } -1 < Ind < -L_1 \quad (1) \\ \dfrac{C_{ij}}{LCM_{ij}} + \dfrac{T_{ij}}{LTS_{ij}} \times (1 - |Ind|) & \text{if } -L_1 \le Ind < -L_2 \quad (2) \\ \dfrac{C_{ij}}{LCM_{ij}} + \dfrac{T_{ij}}{LTS_{ij}} & \text{if } -L_2 \le Ind \le L_2 \quad (3) \\ \dfrac{C_{ij}}{LCM_{ij}} \times (1 - |Ind|) + \dfrac{T_{ij}}{LTS_{ij}} & \text{if } L_2 < Ind \le L_1 \quad (4) \\ \dfrac{T_{ij}}{LTS_{ij}} & \text{if } L_1 < Ind < 1 \quad (5) \end{cases}$$

where $Ind = UTS - UCM$. The value *Ind* is in the range <-1, 1>. L_1 and L_2 are heuristic constants and are equal 0.16 and 0.08 respectively.

In case (1), when $Ind < -L_1$, the code memory is much more used than processor time and therefore only code memory contributes to cost V_{ij}. The heuristic should minimize further increase in code memory utilization. On the other hand, when value *Ind* is greater than L_1, the heuristic aims at minimizing further increase of processors utilization. When the utilization of processor time and code memory is balanced (3) then both factors are taken into consideration with the same weight. The remaining cases describe situation when one of the resources is slightly overused. To counteract this, the weight of the other resource is decreased.

4.3.2 The Pseudo Code of the Heuristic

In this section we present the pseudo-code of our heuristic as depicted in Figure 12. In each iteration of the while loop we first choose a task to be scheduled. The selection of the actual task depends on data memory requirement estimate. Next step is to assign the chosen task to a processor which is selected according to the introduced cost function. After assignment of the task to the processor, ingoing communications are assigned and scheduled on the communication device and finally the task is scheduled.

```
while (R ≠ ∅) {
// S - set of tasks which are already scheduled
// T - set of all tasks
// R- set of all ready tasks {x | x ∈ T - S ∧ pred(x) ⊆ S}
    if ($\frac{\sum_{t_j \in R} DI_j}{ADM} < L_0$) {
            // ADM  - available data memory
            // $L_0$   - heuristic constant equals 0.4
            // $\tau_j$    - start time of task $T_j$
            // Choose the task according to schema minimize_schedule_length

            find a task $T_j$ among tasks in R with smallest max($\tau_j$).
            (the second criteria is smallest $\Delta d_j$.)
            //        $\Delta d_j = DO_j - DI_j$, where
            //        $DO_j$      - amount of data transmitted from task $T_j$
            //        $DI_j$      - amount of data transmitted to task $T_j$
    else {
            estimate $E_m$
            estimate max($\tau_j$) for all tasks in R
            estimate data memory utilization (EDMU) using cumulative/8
                    and previous estimates
            if (EDMU < ADM) {
                    // Choose the task according to schema minimize_schedule_length
                    find a task $T_j$ among tasks in R with smallest max($\tau_j$).
                    (the second criteria is smallest $\Delta d_j$)}
            else {
                    // Choose the task according to schema minimize_data_memory
                    find a task $T_j$ among tasks in R with smallest $\Delta d_j$.
                    (the second criteria is smallest max($\tau_j$))}}}

    Compute $V_{ij}$ for each processor which can execute task $T_j$. Assign task $T_j$ to processor $P_i$ with the smallest $V_{ij}$.
    Schedule incoming communications of task $T_j$, in such a way, that the start time of task $T_j$ - ($\tau_j$) is minimal.
    Schedule task $T_j$
    // Task $T_j$ and all incoming communications are assigned and scheduled }
```

Figure 12. The pseudo-code of the heuristic

The heuristic described above balances the utilization of the code memory and available time slots. The relation between the distributed execution and data memory requirement is also addressed and coped with.

4.4 System-Level Synthesis - Experimental Results

At the system level, a video coding algorithm H.261 has been used for evaluation. The task graph contains 12 sub-tasks and 14 interconnections between them. We conducted three experiments. The first one is the non-pipeline implementation. It was generated in a fraction of a second and proved to be optimal. The pipelined designs use 3 stage pipeline and two buses has the stage latency 1154 and the total execution time of 3373. This result is different from the one known from (Bender, 1996) which provides 1320 latency time and 3027 total execution time. The difference in the results comes from the additional constraint introduced in (Bender, 1996). They do not allow to start a new computation on a given resource before all previous computations did not finish their executions. Our approach does not need this simplifying assumption and therefore can produce better results. Finally, we have generated the pipelined designs with one and three buses instead of two. All pipeline designs improve the performance.

The heuristic presented in section 4.3 was applied to a number of randomly generated task graphs. Each of these examples consists of 100 computation tasks and 120 communication tasks. Searching for a solution for these examples was done with different initial constraints for the execution deadline. The results of the heuristic do not degrade when the constraint for the execution deadline was considerably tighten. An important advantage of this heuristic is its ability to exploit the parallelism existing in the graph. The heuristic was able to obtain very good results. Providing average utilization over a number of different execution deadlines for buses between 73% and 80% and for processors between 86% and 90%. The resources used in the architecture were also constraining significantly the solution space. The average utilization of data memory of the processors was between 80% and 88% and the average utilization of code memory was between 85% and 89%. The average values were obtained from totally 100 experiments.

5. Conclusions

In this paper, we have presented methods for digital system modeling and synthesis using finite domain constraints and CLP paradigm. We have addressed both high-level and system-level synthesis targeting different optimization goals. First, the basic formulation of the computation graphs has been introduced and formalized using finite domain constraints. Then we have shown how to use this formulation together with different extensions for high-level synthesis. The presented methods make it possible to optimize execution time of the design, resource utilization and power consumption. System-level synthesis has been defined in a similar way but it was extended with important code and data memory constraints. The introduced modeling techniques have been later used for synthesis by applying optimization methods based on B&B algorithms and domain specific heuristics.

Standard CLP optimization method is based on branch and bound (B&B) algorithm. It can be successfully applied to middle size problems, but large problems with heterogeneous constraints require more sophisticated optimization methods. The big advantage of CLP is the possibility to create new heuristics using the available meta-heuristics. In our systems, we use credit search heuristic (Beldiceanu et al., 1997), as well as our domain specific heuristic. Using credit search, we are able to partially explore the search space and to avoid situations when the search is stuck at one part of the tree.

We carried out extensive experiments for broad class of applications. The modeling framework incorporates different design styles. Different design constraints, such as power consumption, memory size, timing constraints and designer specific constraints, guide the constraint solver towards a better final solution. The final solution can be further optimized using different synthesis goals, such as cost, performance. The experimental results presented in [(Gruian et al., 1998), (Kuchcinski, 1997), (Kuchcinski, 1998), (Szymanek et al, 1999)] prove the usability of the proposed methods for large scale designs which contain up to ~200 computational and communication tasks. They show that CLP with finite domain constraints and particularly the CHIP system provide a good basis for solving many problems from the area of digital system design which require combinatorial optimization methods. These methods are especially well suited for the cases when many heterogeneous constraints are required for the problem specification.

6. References

1. Beldiceanu N., Bourreau E., Simonis H. and Chan P.: Partial search strategy in CHIP, Presented at 2nd Metaheuristic International Conference MIC97, Sophia Antipolis, France, 21-24 July 1997
2. Bender A.: Design an Optimal Loosely Coupled Heterogeneous Multiprocessor System, In *Proc. of the European Design and Test Conference*, March 11-14, 1996, Paris, France, pp. 275-281.
3. COSYTEC: CHIP, System Documentation, 1996
4. Eles P., Kuchcinski K. and Peng Z.: System Synthesis with VHDL, Kluwer Academic Publisher, 1997
5. Gruian F. and Kuchcinski K.: Operation Binding and Scheduling for Low Power Using Constraint Logic Programming, Proc. 24th Euromicro Conference, Workshop on Digital System Design, Västerås, Sweden, August 25-27, 1998
6. Kuchcinski K.: Embedded System Synthesis by Timing Constraints Solving, Proc. of the 10th Int. Symposium on System Synthesis, Sep. 17-19, 1997, Antwerp, Belgium
7. Kuchcinski K.: An Approach to High-Level Synthesis Using Constraint Logic Programming, Proc. 24th Euromicro Conference, Workshop on Digital System Design, Västerås, Sweden, August 25-27, 1998
8. Lee J-H., Hsu Y-Ch. and Lin Y-L.: A New Integer Linear Programming Formulation for The Scheduling Problem in Data Path Synthesis, *Proc. IEEE International Conference on Computer-Aided Design*, November 5-9, 1989.
9. Raghunathan A. and Jha N. K.: Behavioral Synthesis for Low Power, Proceedings of ICCD 1994
10. Reeves C. R.: Modern Heuristic Techniques for Combinatorial Problems, Blackwell Scientific Publications, 1993
11. Rhinehart M. R. and Nestor J. A.: SALSA II: A Fast Transformational Scheduler for High-Level Synthesis, *Proc. of IEEE International Symposium on Circuits and Systems*, May 1993, pp. 1678-1681.
12. Szymanek R. and Kuchcinski K.: Design Space Exploration in System Level Synthesis under Memory Constraints, 25th Euromicro Conference, Workshop on Digital System Design, Milan, Italy, September 8-10, 1999