An MILP Based Algorithm for Automatic System Level Synthesis

U.Nagaraj Shenoy Prith Banerjee Alok Choudhary

Technical Report No. CPDC-TR-9903-003

March 1999

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

An MILP Based Algorithm for Automatic System Level Synthesis *

U.Nagaraj Shenoy Prith Banerjee
Alok Choudhary
Center for Parallel and Distributed Computing
Northwestern University

Abstract

Automatic Synthesis of large realtime systems using an array of off-the-shelf general purpose processors, digital signal processors and Field Programmable Gate Arrays is increasingly gaining importance. Each of the subproblems in such a synthesis process, namely, pipelining, resource type selection, resource allocation, resource assignment, resource scheduling and resource reconfiguration are complex and are conventionally solved using a mixture of exact and heuristic techniques (often applied in a phased manner) with little guarantee on the optimality of the solution.

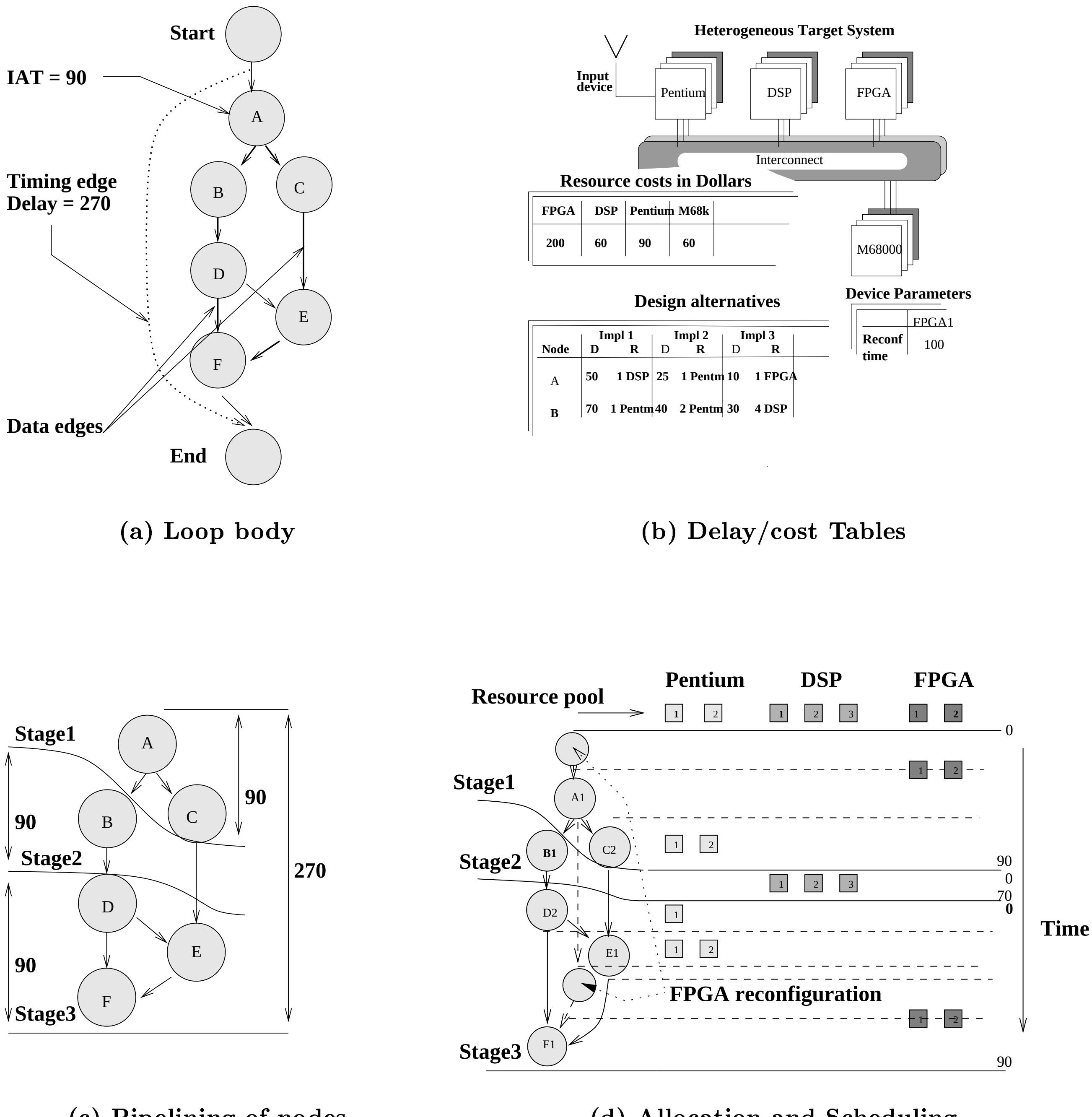
In contrast, this paper formulates the pipelining, selection, allocation, scheduling and reconfiguration problems as a single mixed integer linear programming (MILP) problem to ensure optimality. We show that for many practical size problems, our formulation can produce optimal solution in reasonable time. We believe that the uniqueness of our technique stems not only from the optimality of the solutions it can generate but also from the fact that the problem we have modeled is a generalization of several synthesis problems. Compared to the conventional MILP models for scheduling, the number of integer variables needed in our model are much less and is independent of the total cycle time (which can be very high in system level synthesis). Our model can handle *time constrained*, *resource constrained* as well as *time and resource constrained synthesis* problems.

1 Introduction

The current trend in building large time critical systems is to employ a combination of general purpose processing elements and application specific hardware. While the off-the-shelf processors act as convenient resources to perform routine (non time critical) parts of the computation, special purpose ASICs are employed to speedup the computation bottlenecks. With faster and higher capacity field programmable devices (FPGAs) becoming available, a typical system design has shifted to one of intelligently utilizing off-the-shelf general purpose processors and FPGAs.

The core part of the system design often starts with some compact representation of the processes involved [17, 14, 13, 12]. One of the most common and convenient internal representation of a realtime computation has been in terms of what are known as *control data flow graphs (CDFGs)* where the *nodes* indicate the computation tasks and the directed edges represent either the flow of

*This work was supported by DARPA under contract F30602-98-2-0144



(a) Loop body

(b) Delay/cost Tables

(c) Pipelining of nodes

(d) Allocation and Scheduling

Figure 1: Automatic Synthesis Example

execution control or the *data* across the tasks. Two nodes connected by a directed edge are said to be *control/data* dependent on each other. These *CDFGs* can be further augmented with additional directed edges to indicate the realtime constraints. Such a *timing edge* could indicate the total time allowed for the entire program or that between the end of one task and the beginning of another task. In real time systems involving repeated processing of some input data which arrives at a predefined rate, we could have an additional constraint that the processing has to be completed within the *inter arrival time (IAT)*.

1.1 A Motivating Synthesis Example

In Figure 1(a), we illustrate these concepts with the help of a synthetic program. There are six (labeled as A to F) tasks performed as part of the body of a loop and the various tasks have a dependence relation as indicated by the directed edges. The task A reads the incoming data which arrives every 90 ms. imposing a *throughput constraint* of 90 on this loop body. Further, the last task in the loop body, namely the task F, needs to take an action on the processed data in no more than 270 ms. after the data is read. This timing constraint (*latency*) is indicated as a timing edge with a weight of 270 in the figure.

Given a set of resource types (*candidate resource types*), it may be possible to implement each or all of these tasks by appropriately *selecting* the resources. Some of the resources may be most suitable for some kind of computations. What we really have is a large set of *design alternatives* from which we need to narrow down to a particular design which meets the requirements of a specific application. Each of these alternatives are associated with a *delay* (the time taken by the task if implemented using those resources) and a *cost* (could be the Dollar cost of the device or the real estate needed to implement the task on an FPGA and so on). Often, these design alternatives are encapsulated in what are known as *Delay/cost tables.* In Figure 1(b) we show one such delay cost table for our example.

As in our example program, an application may require that the incoming data is consumed at a faster rate and the processing of each data packet can take more time than the IAT. Instead of trying to design a system which can not only consume the incoming data at the specified rate but also process it at the same rate, a more cost effective strategy would be to *pipeline* the processing. As we have illustrated in Figure 1(c), dividing the original *CDFG* into three pipeline stages would meet both the timing requirements provided we ensure that each stage executes in not more than 90 ms.(IAT).

The computation load of a large realtime system may dictate that each of the tasks in the *CDFG* is implemented by employing more than one resource (same or of different types). Typically, data parallel implementations of the tasks utilize more than one resource of a given type. So, we need to not only decide on the resource type to be used for a given task, but we also have to *allocate* the required number of resources to each of the tasks. Further, to ensure maximum utilization of these resources we may wish to *schedule* these resources across the tasks wherever possible.

Use of programmable devices such as FPGAs adds one more dimension to the synthesis problem. These devices have to be *configured* before they can perform the computation. The time needed to configure these devices may not be negligible. One simple way to avoid these configuration times is to employ *pre-configured* devices. However, this prevents the use of the same device across different tasks. Better utilization of these devices can be achieved if we could reconfigure them just before the task starts execution, somehow hiding the reconfiguration times. It may or may not be possible to achieve this in all cases. In such cases we can decide to use some of these devices as *pre-allocated* (and hence pre-configured) exclusively to a task and others as reconfigurable on the fly.

In Figure 1(d) we have tried to illustrate how the *CDFG* in Figure 1(a) can be synthesized. We have selected Pentium, DSP and FPGA as the resource types, chosen specific implementations of the tasks using these resource types from among the several alternatives available in the delay/cost table, pipelined the *CDFG* into three pipeline stages, allocated altogether 2 Pentiums, 3 DSPs and 2 FPGAs to implement the entire *CDFG*, we have scheduled these devices across the tasks to ensure maximum utilization and finally we have scheduled the reconfiguration of the FPGAs in such a way that the same two FPGAs are used by both the tasks A and F.

1.2 Complexity of the Problem

The illustration in our example could give an impression that we can solve the synthesis problem by performing selection, pipelining, allocation, scheduling and hiding reconfiguration in successive phases. In fact, many of the current techniques do indeed employ such an approach. Unfortunately, these phases interact with each other so much that to achieve optimal solution we need to solve all these problems together rather than one after another in a phased manner. To make the matter worse, each of these subproblems are by themselves hard to solve (many are provably NP complete) and that is one of the reasons why current techniques employ heuristics [18, 6] or randomized search techniques [15] (such as annealing, genetic algorithms and so on) to solve each of these problems.

While the use of such non exact techniques has the advantage of arriving at a quick solution, in most cases there is no guarantee on the optimality of the result. Exact techniques based on mixed integer linear programming (MILP) have been attempted to solve some of the subproblems (scheduling for example [16]), but have two limitations. Firstly, these models don't address all the subproblems in a unified way and only guarantee optimality of the subproblem modeled. In effect, the result could still be suboptimal. Secondly, many of the MILP models which try to solve the scheduling problem are not very suitable when the number of time steps taken by each subtask is very large (which is typically the case in system level synthesis). Many of the MILP solvers are very sensitive to the number of integer variables in the model which could be very large even for a *CDFG* with a few nodes. These problems have discouraged the use of MILP techniques to solve the combined synthesis problem.

Recent work in application specific solution of MILP problems (see [5]) has once again regenerated interest in the powerful and elegant MILP modeling techniques. There is also work done in the area of pruning the search space so that the actual MILP solver spends less time finding the optimal solution.

1.3 Paper Outline

In the current paper, we present a combined MILP model to tackle all the subproblems listed earlier in such a way that the interaction between the subproblems are taken into account and the optimality of the result is guaranteed.

The rest of the paper is organized as follows. In the next section we formulate the system level synthesis problem and formally chalk out the main focus of our model. In Section 3 we introduce our MILP model wherein we discuss each of the subproblems and show how they fit in the combined model. We have evaluated our model using a combination of real and synthetic benchmarks. The results of this evaluation are presented in Section 4 where we show that our model produces results which are far superior to what could be produced using manual approach and the time taken to solve these models are not too high. After a brief overview of related work in Section 5, we conclude the paper with concluding remarks in Section 6.

2 The Automatic Synthesis Problem

There are broadly two kinds of problems namely *Time Constrained Synthesis (TCS)* and *Resource Constrained Synthesis (RCS)*. The former deals with minimizing the resource costs and the latter deals with minimizing the execution time assuming a set of resources to be used. More formally, these problems can be defined as

Given

1. A control data flow graph (*CDFG*) of the computation to be performed.
2. Total time and throughput constraints.
3. Possible resource types (in case of TCS) or a set of resources to be used (in case of RCS).
4. An optional bound on the number of resources of each type (combined TCS/RCS problem).
5. Design alternatives (in the form of delay/cost table).

Synthesize the program to minimize

- The total cost of the resources (in case of TCS).
- The total execution time (in case of RCS).

Such that

1. Timing constraints (indicated by timing edges) are not violated.
2. Throughput constraints are not violated.

Note that the complete problem of automatic synthesis involves many more aspects such as automatic problem partitioning, data flow analysis, construction of delay/cost tables and so on. In the present paper we only focus on the core problem discussed in Section 1.

2.1 Solution Strategy

We first formulate the MILP problem using various constraints as discussed in the next section. The solution of this MILP model would give us the number of resources of a given type needed to implement the complete *CDFG*. The solution also decides the pipeline stages and the start times of each task. All that we need to do is to assign the resources selected to each of the individual tasks and initiate the configuration of the reconfigurable devices (if need be) at times decided by the MILP solution. The resource assignment can be easily done by a simple variant of list scheduling. It should be noted however, that the introduction of this list scheduling does in no way make the solution suboptimal since the start times and the number of resources to be used by each task is decided by the MILP solution.

3 The MILP Formulation

Our MILP model combines pipelining, selection, allocation, scheduling and reconfiguration into a single model to ensure optimality. Further by suitably defining the objective function, we can either perform time constrained or resource constrained synthesis. The constraints which model each of the subproblems are discussed separately in successive subsections below. We use the notation listed in Table 1.

Table 1: Notation Used in the MILP Formulation

$\mathcal{T}$	Set of resource types.
N_t	# of resources of type t
l_t	Dollar cost of resource of type t.
r_t	Configuration delay of a device of type t.
T	Inter-arrival time (IAT)
$t_{c_{ij}}$	Weight of the timing edge between nodes i and j.
N_v	# of nodes in the CDFG
N_{a_i}	# of design alternatives for node i
a_{ik}	Stands for selection of the k^{th} alternative for the i^{th} node.
d_{ik}	Execution time of i^{th} node if k^{th} alternative is selected.
D_i	Execution time of i^{th} node after selection.
u_{ikt}	# of resources of type t needed by the k^{th} alternative for i^{th} node.
s_i	Start time of i^{th} node.
e_{ij}, e'_{ij}	Indicate overlap between nodes i and j.
p_i	Pipeline stage of node i
R_{it}	Total # of resources of type t used by i.
c_{it}	Indicates whether the resource of type t used by node i needs reconfiguration.
P_{it}	# of pre-configured resources of type t used by node i
C_{it}	# of resources of type t needing reconfiguration before used by node i
$\mathcal{Z}$	Set of boolean variables.
$\mathcal{N}$	set of positive integers.
N	Largest representable positive integer. (used in some of the constraints)

3.1 Selection

The selection problem involves choosing one of the many implementation alternatives for each node in a CDFG such that the execution times of the nodes with the selected implementations do not violate (1) the precedence requirements imposed by *data* edges between the nodes, (2) timing constraints imposed by timing edges and finally (3) the total delay constraint on the entire *CDFG* itself.

We use boolean variables $a_{ik} (1 \leq i \leq N_v, 1 \leq k \leq N_{a_i})$ to indicate whether a specific implementation k of a node v_i is used in the final selection. A value of 1 indicates that a specific implementation is used. Note that one and only one of the implementations will be finally selected. This leads to the following uniqueness constraint on these variables.

$$\sum_{k=1}^{N_{a_i}} a_{ik} \quad = \quad 1, \forall i, 1 \leq i \leq N_v \tag{1}$$

With these variables to indicate the selection of a particular implementation of a node v_i we can model the execution time of a node (after selection) as

$$D_i = \sum_{k=1}^{N_{a_i}} a_{ik} d_{ik}$$

We also use variables $s_i (1 \leq i \leq N_v)$ that indicate the start times of the nodes in the final synthesized *CDFG*. Using these variables we can express the following *selection* constraints to be met by our model.

Precedence Constraints A *data* edge between a pair of nodes v_i and v_j ($v_i \prec_{\text{data}} v_j$) imposes a constraint that the start time s_j of node v_j cannot be less than the end time of the node v_i. We can express this precedence constraint between v_i and v_j as

$$s_j \geq s_i + D_i \tag{2}$$

We need to impose such a precedence constraint for each pair of nodes connected by a *data* edge.

Timing Constraints As we discussed earlier, we have two types of timing constraints. The constraint could be imposed by a timing edge between a pair of nodes or it could be imposed by a bound on the total processing time of the entire *CDFG*.

A timing constraint of $t_{c_{ij}}$ units of time imposed by a timing edge between a pair of nodes v_i and v_j ($v_i \prec_{\text{timing}} v_j$) implies that the start time s_j of node v_j cannot be greater than the end time of node v_i by $t_{c_{ij}}$ units of time. This can be expressed as

$$s_j \leq (s_i + D_i) + t_{c_{ij}} \tag{3}$$

An overall processing time constraint of t units of time from the *start* node to *end* node can be imposed as

$$s_{end} \leq s_{start} + t \tag{4}$$

3.2 Pipelining

In case the timing constraints dictate the pipelining of a CDFG, we need to impose constraints such that the tasks in the CDFG get placed suitably in different pipeline stages. The requirements for pipelining are (1) the delay D_i of a node i assigned to any of the pipeline stages should not be more than the inter-arrival time T, (2) the execution time interval $[s_i, (s_i + D_i - 1)]$ of node i should fall within one of the pipeline stage intervals.

We use integer variables p_i to denote the pipeline stage to which a node i finally belongs after pipelining. The constraints that model pipelining are

$$\begin{aligned} D_i &\leq T & (5)\\ s_i &\geq T * p_i & (6)\\ s_i + D_i - 1 &< T * (p_i + 1) & (7)\\ p_i &< t/T & (8)\\ & \forall i, 1 \leq i \leq N_v \end{aligned}$$

3.3 Allocation

Let $\mathcal{T}$ denote the set of candidate resource types and let $t \in \mathcal{T}$ be a specific resource type. Let u_{ikt} denote the number of resources of type t needed to implement the k^{th} alternative for the node i. Allocation involves assigning a set of resources of a given type to a particular node. Using the selection variables, the number of resources of type t to be allocated to a given node v_i is given by

$$R_{it} = \sum_{k=1}^{N_{a_i}} a_{ik} u_{ikt} \tag{9}$$

If the resource t is reconfigurable (like FPGA for example) we need to finally decide on the number of such devices which are exclusively allocated to a given node (and hence not reconfigured) and those which are reconfigured (on the fly) just before the node starts execution. We use Boolean variables c_{it} to denote whether or not the resources of type t allocated to node i need reconfiguration. Let P_{it} denote the number of pre-configured devices of type t exclusively allocated to node i and let C_{it} denote those devices which need reconfiguration before being used for the node i. If the devices are not reconfigured (i.e. $c_{it} = 0$), then P_{it} is equal to R_{it}. Or else, P_{it} is equal to zero and C_{it} is equal to R_{it}. This can be indicated by the following constraints.

$$P_{it} \geq R_{it} - N c_{it} \tag{10}$$

$$C_{it} = R_{it} - P_{it} \tag{11}$$

3.4 Scheduling

The scheduling of the nodes is taken care by the constraints specified Equations 2,3 and 4. However, they by themselves don't ensure that the same resource t_k is not allocated to two or more nodes at the same time. We need to ensure this since our computation model requires that all the resources needed by a specific node get allocated exclusively to the node during the time of its execution. This implies that if the execution times of two tasks overlap, then only one of them can be allocated a specific resource t_k.

For a given resource type t, we need to find out the maximum number of resources of type t needed to execute the CDFG. If nodes $i_1, i_2, ..., i_m$ are scheduled in such a way that their execution times overlap, and all of them use at least one resource of type t, then the number of resources of type t needed to execute these m tasks is given by $\sum_{j=1}^{m} R_{i_j t}$. In order to find the total number of resources needed to implement the entire CDFG, we need to compute the maximum over all such possible sets of overlapping tasks using a given resource type t.

We use boolean variables e_{ij}, e'_{ij} to denote the overlap between two tasks i and j. These two tasks overlap if and only if $e_{ij} = 0, e'_{ij} = 0$. These variables can be set by imposing the following set of constraints.

$$\begin{aligned}
s_i - (s_j + D_j - 1) &> -N e_{ij} \\
s_j - (s_i + D_i - 1) &> -N e'_{ij} \\
(s_j + D_j - 1) - s_i &\geq -N(1 - e_{ij}) \\
(s_i + D_i - 1) - s_j &\geq -N(1 - e'_{ij})
\end{aligned}$$

Whether or not these two nodes actually overlap is decided based on the selection and scheduling of the tasks. However, it is possible to rule out those sets of nodes which cannot overlap in the final solution. In the case of non pipelined *CDFG*s we make the following observations.

Observation 1 *Two nodes v_i and v_j such that v_i is a* predecessor *(due to a data dependence) of v_j can not overlap.*

Observation 2 *Two nodes v_i and v_j which lie on mutually exclusive control paths can not overlap.*

Observation 3 *If the implementations of nodes v_i and v_j are such that $u_{ikt} = u_{jkt} = 0, \forall\ k$ for a specific resource of type t, these two nodes do not overlap (i.e. the overlap can be ignored as far as scheduling of a resource of type t is concerned).*

Observation 4 *If the* schedulable intervals *$\mathcal{I}_i$ and $\mathcal{I}_j$ of the two nodes v_i and v_j do not intersect, then these two nodes can not overlap. The intervals such as I_i are computed using the* ASAP *and* ALAP *schedules for the corresponding node.*

Considering pipelining, we make the following observations.

Observation 5 *If the corresponding implementations of the two nodes v_i and v_j are such that the sum of $m_i = \min d_{ik}$ and $m_j = \min d_{jk}$ is greater than the IAT T, then these two nodes overlap.*

Observation 6 *A loose bound on the cardinality of the set of mutually overlapping nodes is given by the product of the number of pipeline stages and the* width *of the* CDFG.

Observation 7 *A tighter bound on the cardinality of the set of mutually overlapping nodes is given by adding the cardinalities of the first t/T sets in an ordered (on the cardinality of the sets) partition of the nodes in the given* CDFG. *The sets in this partition are the largest subsets containing mutually independent nodes.*

When the CDFG is pipelined, the start times s_i and s_j of two potentially overlapping tasks have to be appropriately replaced by start times relative to the beginning of the pipeline stage (the start time of the corresponding pipeline stage needs to be subtracted). In that case, the modified set of constraints used to compute e_{ij} and e'_{ij} would be

$$\begin{aligned}
(s_i - Tp_i) - ((s_j - Tp_j) + D_j - 1) &> -Ne_{ij} \\
(s_j - Tp_j) - ((s_i - Tp_i) + D_i - 1) &> -Ne'_{ij} \\
((s_j - Tp_j) + D_j - 1) - (s_i - Tp_i) &\geq -N(1 - e_{ij}) \\
((s_i - Tp_i) + D_i - 1) - (s_j - Tp_j) &\geq -N(1 - e'_{ij})
\end{aligned}$$

Let $n_1, n_2, ..., n_k$ be one of the set of nodes which could possibly overlap and all of which use a resource of type t. The total number of resources of type t needed to implement these nodes is given by

$$N'_t \geq \sum_{i=1}^{k} \left(R_{it} - N \sum_{j=1}^{k} e_{ij} - N \sum_{j=1}^{k} e'_{ij} \right) \tag{12}$$

Note that when a device of type t is reconfigurable, we use C_{it} computed by Equation 11 in the place of R_{it} in the above constraint. Finally, the maximum number of resources N_t needed to implement the entire CDFG is given by taking the maximum over all such sums.

$$N_t \geq N'_t, \forall \text{ overlapping sets of nodes} \tag{13}$$

3.4.1 Scheduling of Mutually Exclusive Control Paths

If two nodes happen to fall on mutually exclusive control paths, we can allocate the same resource to both of them. The execution time intervals for these nodes may or may not overlap. Our MILP formulation can easily take care of these cases by assuming that these two nodes can never overlap (even if their execution time intervals do in reality).

3.5 Handling of Reconfigurable Devices

We use boolean variables c_{it} to denote whether the reconfigurable devices of type t used to implement the node i need to be reconfigured before the node starts execution. These variables will have a value 1 if and only if the implementation selected for the node uses a reconfigurable device of type t and this device is reused across nodes (thus needing reconfiguration). The first part of this condition is imposed by

$$c_{it} \leq \sum_{k=1}^{N_{a_i}} a_{ik}, \forall \text{ k, such that implementation k uses device type t} \tag{14}$$

The effective start time s'_i of the node i is advanced by the reconfiguration time r_t of the device if the device needs to be reconfigured. This is indicated by imposing the following condition.

$$s'_i = s_i - r_t c_{it} \tag{15}$$

It is this modified start time s'_i which is used while computing the overlap between two nodes if the nodes use a reconfigurable device to ensure that the reconfigurable device is assigned to the nodes not only during the execution of the nodes but also while the device is being reconfigured. In addition, we need to ensure that the reconfiguration takes place within the same pipeline stage in case the CDFG is pipelined. All this will result in the following modified constraints.

$$\begin{aligned}
(s'_i - Tp_i) - ((s'_j - Tp_j) + D_j - 1) &> -Ne_{ij} \\
(s'_j - Tp_j) - ((s'_i - Tp_i) + D_i - 1) &> -Ne'_{ij} \\
((s'_j - Tp_j) + D_j - 1) - (s'_i - Tp_i) &\geq -N(1 - e_{ij}) \\
((s'_i - Tp_i) + D_i - 1) - (s'_j - Tp_j) &\geq -N(1 - e'_{ij}) \\
s'_i &\geq Tp_i \\
s'_j &\geq Tp_j
\end{aligned}$$

3.6 Objective Function

In our synthesis problem we may have two different objectives. We could synthesize a system by minimizing the resource cost (Time Constrained Synthesis -TCS) or we could synthesize a system which results in the least execution time for a given set of resources (Resource Constrained Synthesis - RCS). In both cases we can set upper bounds on the number of resources of a given type. Also, the synthesized system has to necessarily meet all the timing and throughput constraints. Depending on our objective - TCS or RCS - we have a different objective function.

3.6.1 Time Constrained Synthesis

The objective of TCS is to minimize the cost of the synthesized program. There could be several cost criteria one could choose in a given situation. In the present model we assume the Dollar costs of the resources as the criterion. The objective function in this case is the sum of the Dollar costs of all the devices used to implement the CDFG.

$$C_{TCS} = \sum_{\forall t \in \mathcal{T}} l_t \hat{N}_t \tag{16}$$

Where $\hat{N}_t$ is the number of devices of a given type t used in the implementation of the CDFG and l_t is the cost of each device of type t. We need to consider three different cases while computing the number of devices $\hat{N}_t$ of type t.

Case 1: The device is a conventional processor reusable across the nodes. In this case $\hat{N}_t$ is nothing but N_t computed as a result of the constraint 13.

Case 2: The device is reusable across nodes but needs to be reconfigured unless exclusively allocated to a specific node. In this case some of these devices could be pre-allocated and others need reconfiguration. The number computed in equation 13 considers only those which are reconfigured. To this we need to add pre-allocated devices. Effectively, the device count in this case is given by

$$\hat{N}_t = N_t + \sum P_{it}, \ \forall \text{ nodes } i \text{ which are implemented using such devices.} \tag{17}$$

Case 3: The device is neither reconfigurable nor reusable across nodes. These devices could be ASICs or FPGAs which can not be reconfigured due to device/hardware limitations. In this case N_t is given by

$$\hat{N}_t = \sum R_{jt}, \ \forall \text{ nodes } j \text{ that are implemented using such devices} \tag{18}$$

3.6.2 Resource Constrained Synthesis

In this case, the number of devices N_t of a given type t (or an upper bound on that) is pre-specified and the synthesis has to arrive at a solution using these devices in such a way that the execution time is the minimum. The objective function for RCS is relatively simple. It is just the start time s_{end} of the end (last) node that we want to minimize. So, s_{end} forms the objective function.

$$C_{RCS} = s_{end} \tag{19}$$

3.6.3 Time and Resource Constrained Synthesis

We can combine time and resource constraints in order to satisfy the timing constraints as well as the limits on resource usage. The simplest of these constraints could be specifying an upper bound on the number of resources of a particular type. For example, we can have

$$N_{\text{FPGA}} \quad < \quad 10$$

Alternately, we could impose a constraint on the number of general purpose processors that can be used in the synthesis (may be due to engineering constraints). For example, we can set

$$N_{\text{Pentium}} + N_{\text{M68k}} + N_{\text{powerPC}} \quad < \quad 30$$

There could be cases where we may want to limit the power consumption of the synthesized system. Assuming that a resource t consumes w_t units of power, we could impose

$$w_{\text{Pentium}} N_{\text{Pentium}} + w_{\text{M68k}} N_{\text{M68k}} + w_{\text{FPGA}} N_{\text{FPGA}} \quad < \quad 1000$$

3.7 The Complete MILP Problem

Using the objective functions given in Equations 16 and 19, uniqueness constraints (Equation 1), precedence constraints (Equation 2), timing constraints (Equations 3 and 4), pipelining constraints (Equations 5 and 6), allocation constraints (Equations 9, 10 and 11), scheduling constraints (Equation 12 and 13) and reconfiguration constraints (Equation 14 and 15), we express the synthesis problem as follows.

min C
with

$$\mathrm{C} = s_{end} \qquad \text{(for RCS)}$$

$$\mathrm{C} = \sum_{\forall t \in \mathcal{T}} l_t \hat{N}_t \qquad \text{(for TCS)}$$

subject to

$$\begin{array}{rcl}
s_j & \geq & s_i + D_i \forall v_i \prec_{\text{data}} v_j \\
s_j & \leq & (s_i + D_i) + t_{c_{ij}} \forall v_i \prec_{\text{timing}} v_j \\
s_{end} & \leq & s_{start} + t \\
\sum_{k=1}^{N_{a_i}} a_{ik} & = & 1 \\
 & & \forall i, 1 \leq i \leq N_v
\end{array}$$

$$\begin{array}{rcl}
D_i & \leq & T \\
s_i & \geq & T * p_i \\
s_i + D_i - 1 & < & T * (p_i + 1) \\
 & & \forall i, 1 \leq i \leq N_v
\end{array}$$

$$\begin{array}{rcl}
R_{it} & = & \sum_{k=1}^{N_{a_i}} a_{ik} u_{ikt} \\
P_{it} & \geq & R_{it} - N c_{it} \\
C_{it} & = & R_{it} - P_{it} \\
 & & \forall t \in \mathcal{T}, 1 \leq i \leq N_v
\end{array}$$

$$\begin{array}{rcl}
N'_t & \geq & \sum_{i=1}^{k} \left(R_{it} - N \sum_{j=1}^{k} e_{ij} - N \sum_{j=1}^{k} e'_{ij} \right) \\
N_t & \geq & N'_t \\
 & & \forall \text{ overlapping sets of nodes}
\end{array}$$

$$\begin{array}{rcl}
c_{it} & \leq & \sum_{k=1}^{N_{a_i}} a_{ik} \\
s'_i & = & s_i - r_t c_{it} \\
 & & \forall \text{ j s.t. implementation k uses reconfigurable} \\
 & & \text{device of type t}
\end{array}$$

$$\begin{array}{rcl}
a_{ik}, c_{it}, e_{ij}, e'_{ij} & \in & \mathcal{Z} \\
p_i & \in & \mathcal{N}
\end{array}$$

4 Experimental Evaluation

In this section we present some of our initial experimental results. We have synthesized two real applications (STAP [11] and MPEG) and one synthetic *CDFG* using our MILP technique. The synthesis was performed for various combinations of timing constraints.

The manual synthesis was done by performing pipelining, selection and scheduling in phases. In all cases of manual synthesis, iterative refinement was used to improve the solution.

Table 2: Synthesis of STAP (7 nodes) using a homogeneous set of processors for various performance constraints.

Delay msec.	IAT msec.	Stages	# of processors Manual (no sched)	(with sched)	Automatic (MILP)	MILP solution time secs.
1250	1250	1	60	46	36	< 1
700	700	1	83	55	36	< 1
365	365	1	108	66	55	< 1
1400	700	2	68	60	36	< 1
700	350	2	88	72	48	9
360	**180**	**2**	**148**	**118**	**98**	**24**
1500	500	3	62	60	36	1
750	250	3	94	88	68	193
360	120	3	156	142	130	< 1

Table 3: Synthesis of MPEG (6 nodes) using processors and FPGAs for various performance constraints.

Delay msec.	IAT msec.	Stages	Dollar cost Manual (no sched)	(with sched)	Automatic (MILP)	MILP solution time secs.
110	110	1	375	195	90	< 1
90	90	1	450	270	150	< 1
100	50	2	495	315	225	10
90	45	2	515	395	240	3
75	25	3	540	480	420	6
60	20	3	680	605	530	1
60	15	4	805	805	775	2
50	**10**	**5**	**1145**	**1145**	**1145**	**<1**

The MILP model was automatically generated from the *CDFG* and delay/cost tables by a program written in Java. We used a commercial solver [3] available at [8] to solve this MILP problem. The results of these experiments are shown Tables 2,3 and 4. In these tables the *Delay* and em IAT columns indicate the timing constraints. The column *stages* indicates the number of pipeline stages into which the *CDFG* was divided. The two sub columns *no sched* and *with sched* under the *Manual* indicate the cost of the results arrived at manually without and with scheduling respectively. Similarly the column *Automatic* indicates the cost of the solution found by our MILP model and the last column lists the corresponding MILP solution times.

In all cases, the MILP model produced a solution which is far superior to that arrived at manually. For example, consider the highlighted row in Table 2. Here the total delay is 360 ms.

and inter-arrival time is 180 ms. Both the manual as well as the automatic techniques have pipelined the *CDFG* into 2 pipeline stages (in different ways though). While manual synthesis came up with a solution needing 118 processors (after scheduling), the MILP model came up with a better solution needing only 98 processors.

The main reason for this was the enormous number of design alternatives available for each case. Exhaustive manual searching of such huge design spaces is tedious and time consuming. Besides, the quality of the result also depends on how the *CDFG* was pipelined. We found that the MILP model arrives at a pipelining which is often different from the one found manually. Since manual design solves the problem in phases it ends up generating inferior solutions.

There were also cases where the MILP model produced the same result as that of the manual synthesis (see for example, the last row in Table 3). The reason for this was that due to the tight timing constraints both manual as well as MILP approaches had fewer choices which could be made and came up with same results.

As can be seen from these results, the price we need to pay for optimality is not too high. In most cases the MILP solver found the results quite quickly in a matter of few seconds. We expect this behavior at least in most reasonably sized real *CDFGs*.

Table 4: Synthesis of Random CDFG (15 nodes) using processors and FPGAs for various performance constraints.

Delay	Dollar cost			MILP solution time
msec.	Manual		Automatic	secs.
	(no sched)	(with sched)	(MILP)	
1000	1860	750	390	24
950	1890	990	420	61
900	1890	990	420	57
850	1980	960	480	24
800	2025	1230	510	15
750	2130	1320	585	22
700	2310	1170	735	23
650	2875	2050	930	12

5 Related Work

Various aspects of automatic synthesis of systems has been looked at by a number of researchers ([1, 4, 9, 10, 12, 13, 14, 15, 17, 18] are some of the representative works in this area).

There are basically two broad approaches followed by these works. Majority of them use some form of heuristics to solve the hardware/software co-synthesis problem (like in [18] for example). While other systems employ randomized search algorithms (like in [15] for example) based on evolutionary concepts. MILP models either for scheduling or for module selection have been proposed in the high level synthesis domain. To the best of our knowledge ours is the first MILP model which combines all the subproblems related to system level synthesis into a single model.

6 Conclusion

In this paper we presented a mixed integer linear programming (MILP) based model which captures pipelining, selection, allocation, scheduling and reconfiguration in a single formulation. Our model solves both time constrained (TCS) and resource constrained (RCS) synthesis problems.

Our initial experimental results indicate that the solution times of our model are acceptable for reasonably sized real applications. We are currently working on studying the effectiveness of our technique for large *CDFGs*. Also, we are focusing our attention on pruning techniques which could speed up the solution of large problems. We leave these for future work.

References

[1] A. Kalavade and E. Lee, "Hardware/Software Co-design Using Ptolemy - a Case Study" *Proc. of the IFIP International Workshop on Hardware/Software Co-Design*, 1992.

[2] Daniel Gajski, F. Vahid et.al., "Specification and Design of Embedded Systems", Prentice Hall, Inc.,Englewood Cliffs, New Jersey, 1994.

[3] XPRESS-MP Linear and Integer Programming Modeling and Optimization system, Dash Associates, UK. http://www.dash.co.uk

[4] Ireneusz Karkowski and Henk Corporaal, "Design Space Exploration Algorithm for Heterogeneous Multi-processor Embedded System Desgin", *Proc. of Design Automation Conf. (DAC-98)*, San Francisco,California, 1998, pp.82-87.

[5] M.Narasimhan and J.Ramanujam, "Improving the Computational Performance of ILP-based Problems", *Proc. of ICCAD98*,San Jose, CA, 1998.

[6] U.Nagaraj Shenoy, Prith Banerjee and Alok Choudhary,"Symphany : A System for Automatic Synthesis of Adaptive Systems", Technical Report CPDC-TR-9903-002, Northwestern University, March 1999.

[7] U.Nagaraj Shenoy, Prith Banerjee and Alok Choudhary,"An Exact Algorithm for Automatic System Level Synthesis",Technical Report CPDC-TR-9903-003, Northwestern University, March 1999.

[8] NEOS Server, http://www-unix.mcs.anl.gov/neos/Server

[9] Nohbyung Park and Alice Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications", *IEEE Trans. on Computer Aided Design*, Vol. 7. No. 3, March 1988, pp.356-370.

[10] P.H. Chou, R.B. Ortega and G. Borriello, "The Chinook Hardware/Software co-synthesis System", *Proc. of International Symp. on System Synthesis*, 1995.

[11] R. Brown and R. Linderman, "Algorithm Development for an Airborne Real-Time STAP Demonstration", *Proc. of IEEE National Radar Conference*,1997.

[12] R. Ernst, J. Henkel, and T.Benner, "Hardware-software cosynthesis for microcontrollers", *IEEE Design and Test of Computers*, 1994, pp. 64-75.

[13] R.K. Gupta and Giovanni De Micheli, "Hardware-software Cosynthesis for Digital Systems", *IEEE Design and Test of Computers*, 10(3),1993,pp. 29-41.

[14] Rajesh Gupta and Giovanni De Micheli, "System-level Synthesis using Re-programmable Components", *Proc. of Eur. Conf. on Design Automation (EDAC)*, Belgium, March 1992,pp. 2-7.

[15] Robert Dick and Niraj Jha, "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-time Distributed Embedded Systems",*Proc. of ICCAD98*,San Jose, CA, 1998.

[16] S.Chaudhury and R. Walker, "Analyzing and exploiting the structure of the constraints in the ILP approach to the scheduling problem",IEEE Trans. VLSI, 2(4), pp. 456-471, Dec.1994.

[17] S. Narayan, F. Vahid, and D. Gajski, "System Specification and Synthesis with the Spec-Charts Language", *Proc. of the IEEE International Conf. on Computer Aided Design*, 1991.

[18] Smita Bakshi and Daniel Gajski, "Hardware/Software Partitioning and Pipelining", *Proc. of Design Automation Conf. (DAC-97)*, Anaheim, CA, 1997, pp. 713-716.