A System-level Synthesis Algorithm with Guaranteed Solution Quality[*]

U.Nagaraj Shenoy, Prith Banerjee, Alok Choudhary
Northwestern University, Evanston, IL, USA.

Abstract

Recently a number of heuristic based system-level synthesis algorithms have been proposed. Though these algorithms quickly generate good solutions, how close these solutions are to optimal is a question that is difficult to answer. While current exact techniques produce optimal results, they fail to produce them in reasonable time. This paper presents a synthesis algorithm that produces solutions of guaranteed quality (optimal in most cases or within a known bound) with practical synthesis times (few seconds to minutes). It takes a unified look (the lack of which is one of the main sources of sub-optimality in the heuristic techniques) at different aspects of system synthesis such as pipelining, selection, allocation, scheduling and FPGA reconfiguration. Our technique can handle both time constrained as well as resource constrained synthesis problems. We present results of our algorithm implemented as part of the Match project *[1] at Northwestern University.*

1. Introduction

The current trend in designing large time-critical systems is to employ a combination of general purpose processors, digital signal processors (DSPs) and field-programmable gate arrays (FPGAs). The core part of the system design often starts with some compact representation of the processes involved. 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 *execution control* or the *data* across the tasks (see [2] for an example).

These *CDFGs* can be further augmented with additional *timing* edges to indicate the realtime constraints. In 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).*

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

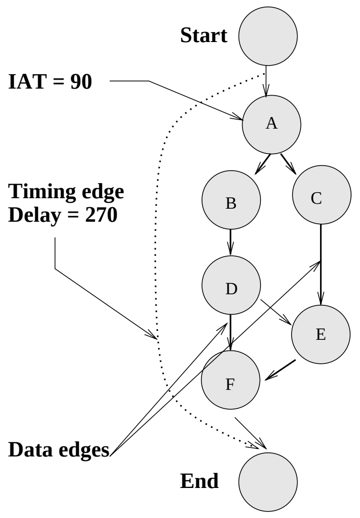


Figure 1. Dataflow graph of the example program loop

1.1. The Synthesis Problem Considered

We illustrate the synthesis problem addressed by our algorithm with the help of an example program (refer Figure 1). There are six (labeled as A to F) tasks performed as part of the body of a loop. 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. (*deadline*) after the data is read. This timing constraint is indicated as a timing edge with a weight of 270.

Given a set of resource types, we have 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 a set of resources) and a *cost* (could be the Dollar cost of the resource 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*.

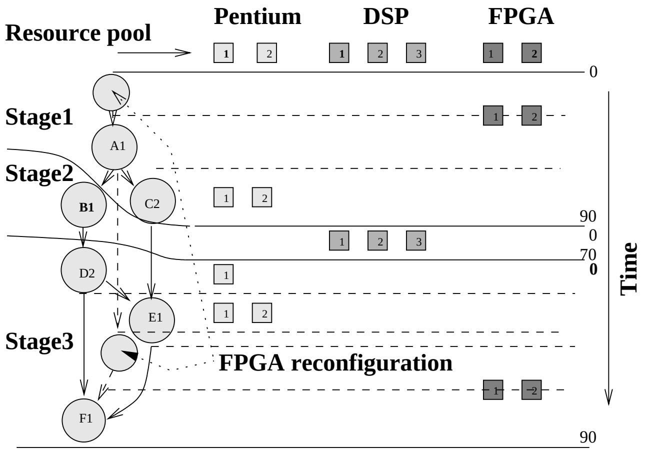


Figure 2. The example CDFG after selection, pipelining and scheduling

The computation load of a large realtime system may dictate that each of the tasks in the *CDFG* is implemented (possibly, in a data parallel manner) by employing more than one resource (same or of different types). We need to decide not only the resource type to be used for a given task, but also the number of resources needed by each of the tasks. To achieve high throughputs at lower resource costs, often these tasks are *pipelined*. Further, to ensure maximum utilization of these resources we may wish to *reuse* these resources across the tasks whenever possible.

Programmable devices such as FPGAs need to be *configured* before they can perform the computation. Better utilization of these devices can be achieved if we could 'hide' the reconfiguration times.

In Figure 2 we illustrate how the *CDFG* in Figure 1 can be synthesized. We have selected a general purpose processor, 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.

In this paper, we describe our synthesis algorithm that addresses all these issues in a unified manner. Our algorithm is based on Mixed Integer Linear Programming (MILP). The rest of the paper is organized as follows. We start with a brief overview of related work in Section 2. In Section 3 we describe our synthesis algorithm. We evaluate our algorithm using a large number of benchmarks in Section 4.

2. Related Work

Many of the system level synthesis subproblems have a corresponding counter-part in high level synthesis. Due to space restrictions we mainly concentrate on some of the works in system synthesis domain that are more closely related to ours in terms of synthesis issues, target architectures and solution techniques.

[3, 4, 5, 6] address problems similar to that discussed in this paper. While [3, 6] focus on inter-task communication related issues, [4, 5] emphasize on achieving high throughputs. Our current work takes into account both communication and throughput related issues.

Unlike the problem we have addressed in this paper, these algorithms assume single processor implementation of each task and also do not address issues related to reconfigurable devices. An algorithm which is targeted to multiprocessor implementation of the tasks is proposed by [7] which emphasizes mainly on program transformations and parallelization strategy for each task. [8] proposes an algorithm for design of systems using FPGAs and focuses mainly on reuse of the FPGAs across tasks.

In terms of the techniques used, [5, 6, 7] employ clever heuristics (based on iterative refinement) and [8] is based on Evolutionary programming. 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 nor any clue as to how close the solution is to the optimal.

[3, 4] use MILP techniques to solve the synthesis problem. However, their models are more restrictive than the one we are addressing. For example, the cost model in [4] does not seem to take into account resource sharing across tasks. Resource sharing is very important to reduce the cost of the synthesized system. Further, it is not clear whether these models are time efficient. While [3] deals with small task graphs (less than 10 nodes) and reports solution times of hours, [4] reports solution times (restricting to only algorithm selection) of the order of minutes. Neither of these algorithms address issues related to FPGAs.

[8] which focuses primarily on FPGAs, tries to minimize the reconfiguration times by optimally sequencing the tasks onto the FPGAs, whereas our algorithm tries to achieve that by *configuration in anticipation of future use (latency hiding)*. An interesting extension to our model can be to incorporate sequencing in addition to latency hiding.

To the best of our knowledge ours is the first MILP model which combines all the subproblems discussed in Section 1.1 into a single model and synthesizes relatively large (hundreds of nodes) graphs in reasonably short times (few minutes) with a guarantee on the solution quality.

3. The Synthesis Algorithm

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

Table 1. Notation Used in the MILP Formulation

	Predefined (target specific) constants:
l_t	Dollar cost of resource of type t.
r_t	Config delay of a device of type t.
	Constants for a problem instance:
N_v	# of nodes in the CDFG
T	Allowed total execution time (constraint).
$t_{c_{ij}}$	Wt. of timing edge between nodes i and j.
δ	Inter-arrival time (IAT).
N_{a_i}	# of design alternatives for node i
d_{ik}	Execution time of i^{th} node if k^{th} alternative is selected.
u_{ikt}	# of resources of type t needed by the k^{th} alternative for i^{th} node.
	Model variables:
D_i	Execution time of i^{th} node after selection.
s_i	Start time of i^{th} node.
N_t	# of resources of type t.
R_{it}	Total # of resources of type t used by i.
P_i	# of pre-configured FPGAs used by node i.
C_i	# of FPGAs used by node i needing reconfig.
a_{ik}	Indicates selection of k^{th} alternative for node i.
z_{ij}, z'_{ij}	Indicate overlap between nodes i and j.
p_i	Pipeline stage of node i.
c_i	Indicates whether the FPGA used by node i needs reconfig.

3.1. Resource Selection

Selecting the resources to implement a specific node in the CDFG involves choosing one of the many possible implementations for the node. This selection implies the *type* of the processing element, *number* of processing elements used (could be more than one if the implementation exploits data parallelism) as well as the *interconnect* used to link these processing elements. Depending on this selection, the cost of implementing the node (in terms of dollar costs, memory requirements, power consumption etc.) and the contribution of the node to the total execution time of the program get decided.

Selection of Processing Elements. We use boolean variables a_{ik} to denote whether an implementation k is chosen for node v_i. One and only one of the alternatives need to be selected for a given node. This implies the following uniqueness constraint.

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

Well known techniques[9, 10] exist that can exploit the *mutually exclusive property* of a_{ik} to speed up the solution.

Choice of a specific implementation for a node decides the execution time of the node. The actual execution time D_i of a node v_i can be expressed in terms of the execution times d_{ik} of individual alternatives (stored in the *delay/cost table*) as

$$D_i = \sum_{k=1}^{N_{a_i}} a_{ik} d_{ik} \tag{2}$$

Selection of Communication Links. The PEs used to implement a given node need to be interconnected to aide the inter-subtask (in a data parallel implementation of a macro task) communication. In addition to this inter-subtask communication, the macro tasks in a CDFG may need to communicate due to data dependencies across them. This communication time depends not only on the type of communication link used to connect the PEs in the two macro tasks, but also on the number of PEs used for each of these nodes.

A data edge e_l connecting two nodes v_i and v_j can be implemented using any one of the L link types. The communication time d'_{xyk} depends on the alternatives a_{ix} and a_{jy} used to implement the nodes v_i and v_j and the link type k used for implementing the edge e_l. Let b_{lk} denote that a link of type k is used to implement e_l. These boolean variables are constrained by (similar to Constraint 1 above)

$$\sum_{k=1}^{L} b_{lk} = 1 \tag{3}$$

In the presence of multiple alternatives for the implementation of a data edge e_l, the effective communication cost D'_l is given by

$$D'_l = \sum_{\forall xyk} f_{xyk} d'_{xyk} \tag{4}$$

Where f_{xyk} are implicit boolean variables (these variables are derived without *additional cost*) that have a value 1 if and only if a_{ix},a_{jy} and b_{lk} are all 1.

3.2. Scheduling the Nodes

The order in which the nodes in a CDFG start execution is largely decided by the data dependencies between them and the availability of resources to execute the tasks. Timing constraints can also influence this order. The start time s_i of node v_i cannot be earlier than the end time of any of its predecessors (source nodes of the associated data edges). If inter node communication times are taken into account, then s_i should be at least greater than the end time of each predecessor plus the time needed for communication. This is implied by the following constraint.

$$s_i \geq s_j + D_j + D'_l, \forall v_j \prec_{\text{data edge } e_l} v_i \tag{5}$$

Similarly, each timing edge e_l with a weight t between nodes v_i and v_j ($v_j \prec v_i$) implies that the node v_i has to start at most t units of time after the end time of node v_j imposing the following constraint.

$$s_i \leq s_j + D_j + t, \forall v_j \prec_{\text{time edge } e_l} v_i \tag{6}$$

3.3. Pipelining

Often, the nodes in a CDFG form part of a loop in which a stream of incoming data is read by one of the tasks and processed by subsequent tasks. The inter-arrival time δ between two successive packets of data can be much smaller compared to the total processing time T allowed. One way to meet this high throughput requirement is to pipeline the tasks into T/δ stages.

Pipelining of the tasks involves assigning the tasks to one of the T/δ stages. For this, each task has to fit in exactly one of the stages (say p_i) imposing the following constraints.

$$\begin{array}{rcl} D_i & \leq & \delta, \forall v_i \\ p_i\delta \leq & s_i & \leq p_i\delta + \delta - D_i \end{array} \tag{7}$$

The first constraint ensures that the execution time of the node does not exceed the pipeline stage delay (same as δ). The second constraint imposes the restriction that the node starts execution anytime after the beginning of stage p_i and the execution completes no later than the end of the same stage.

3.4. Resource Sharing

Often, the resources used to implement the nodes are not utilized beyond the start and end time of each node. These resources can be reused to execute some other nodes and hence bringing down the resource costs. Any two nodes with non overlapping execution intervals (start time to end time) can potentially share the same resource. From the resource's point of view, this means that the same resource would execute the two nodes one after another in the order specified by their respective start and end times.

The classical approach to model this resource scheduling problem [11] using boolean variables t_{ij}, that stand for usage of a resource of type t by the node v_i at time j, result in large number of integer variables (O(T)) for CDFGs with large total execution time T. Even the optimizations based on *execution time intervals* does not bring down this number substantially.

In our model we use a different strategy. We use boolean variables z_{ij} and z'_{ij} to capture the situation where a pair of nodes v_i and v_j may overlap. A value of $z_{ij} = z'_{ij} = 0$ indicates that the two nodes overlap (other combinations indicate non overlap). These variables can be set to the right value by imposing the following constraints.

$$\begin{array}{rcl} s_i - E_j & > & -N(1 - z_{ij}) \\ s_j - E_i & > & -N(1 - z'_{ij}) \\ E_j - s_i & \geq & -Nz_{ij} \\ E_i - s_j & \geq & -Nz'_{ij} \\ \text{where} & & \\ E_i & = & s_i + D_i - 1 \text{ (end time of } v_i) \\ E_j & = & s_j + D_j - 1 \text{ (end time of } v_j) \\ & & \text{N is a large constant} \gg D_k \end{array} \tag{8}$$

The idea here is to force the variables z_{ij} and z'_{ij} to zero whenever the execution intervals $[s_i, E_i]$ and $[s_j, E_j]$ of nodes v_i and v_j intersect. That means whenever both $s_i \leq E_j$ and $s_j \leq E_i$ are true. It is easy to see that the number of such variables needed are independent of the total execution time and much fewer than $O(N_v^2)$ (worst case) variables are needed in practice (refer [12]). Using these boolean variables, we proceed to compute the number of resources of a given type needed to synthesize the CDFG as follows.

Conventional Processors. Assuming that an alternative a_{ik} for a node v_i needs u_{ikt} resources of type t (known a-priori), we can compute the actual number of resources of that type needed by the node v_i by setting the following relation.

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

Now, if two nodes v_i and v_j both have implementations which use a resource of a given type and assuming that the execution times of these nodes overlap, the number of resources of that type needed to implement these two nodes is $R_i + R_j$ (since they cannot share resources).

One way to ensure that two nodes with overlapping execution intervals do not share a resource is by using boolean variables σ_{ik} to denote that node v_i uses the k^{th} resource of a given type. These variables should obey the following constraint

$$\begin{array}{rcl} \sigma_{ik} + \sigma_{jk} & \leq & 1 \text{ (at most one can be true)} \\ \text{whenever} & & z_{ij} = z'_{ij} = 0 \end{array} \tag{10}$$

A similar formulation is used in [3]. However, this formulation has the following problems.

Firstly, this formulation needs additional boolean variables σ_{ik} and the number of such variables can be of the order of total number of resources needed to implement the complete CDFG. Either such an estimate on the number of resources should be known a-priori or else a tight upper bound on them is needed (may not be always possible).

Secondly, in a given solution to the final synthesis problem, any renaming of the resources of a given type (effectively all permutations) is also a solution with exactly the same cost. That means the MILP solver can potentially spend lot more time in weeding out a large number of solutions which are all 'equivalent', before arriving at the optimal solution.

In our model, we use an alternate strategy that does not need any additional boolean variables nor an estimate on the upper bound on the number of resources. If two nodes v_i and v_j can potentially overlap, then we setup the following constraint to compute the total number of resources R of a given type needed by them.

$$\begin{array}{rcl} R & \geq & R_i \\ R & \geq & R_j \\ R & \geq & R_i + R_j - N(z_{ij} + z'_{ij}) \\ \text{where} & & \text{N is a constant} \gg R_k, \forall k \end{array} \tag{11}$$

It is easy to see that R takes the value $R_i + R_j$ whenever $z_{ij} = z'_{ij} = 0$(i.e. the nodes overlap), or else it takes a value of $\max(R_i, R_j)$. Note that this formulation uses no additional variables. These relations can be generalized to a set $\{v_{i_1}, v_{i_2}, ...v_{i_n}\}$ of n nodes that can potentially overlap as follows.

$$\begin{array}{rcl} R & \geq & \max(R_{i_1}, R_{i_2}, ..., R_{i_n}) \\ R & \geq & \sum_{k=1}^{n} R_{i_k} - N \sum_{\forall(ij)} (z_{ij} + z'_{ij}) \end{array} \tag{12}$$

Reconfigurable Devices. Sharing a reconfigurable resource such as FPGA by two or more nodes poses the following problems. The execution of such nodes can start only after the corresponding FPGAs are configured. In general, this process can take from several tens to several hundreds of milliseconds. A good scheduling algorithm should try to reuse these expensive resources as far as possible (without increasing the total execution time) to bring down the overall system cost.

In our formulation, we try to achieve this reuse to a large extent. We look at the usage of a reconfigurable device as two distinct consecutive tasks. Obviously, the first one involves configuring the FPGA and the second (not necessarily immediately following in time) task which actually performs the computation. Now, by scheduling the first one appropriately, it is possible to 'hide' the configuration delay and make it appear as if the FPGA was *pre-configured*. Effectively, no additional delay is added to the total execution time due to this re-usage of the FPGA.

In practice, however, it may or may not be possible to hide the configuration delays of all the FPGAs used in a design. In such cases some of the FPGAs may need to be preconfigured and others dynamically reconfigured. We need to compute the number C_{fp} of the FPGAs that can be reused (dynamically reconfigured) as well as the total number N_{fp} of the FPGAs (including the pre-configured ones) needed in the final design. We use boolean variables c_i to indicate whether the FPGAs used (if at all) by node v_i can be reused by some other node. This variable can have a value 1 *only if* the implementation selected (by the formulation described in Section 3.1) indeed uses FPGA as a resource. That means the following condition should hold.

$$c_i \leq \sum_k a_{ik}, \forall \text{ implementation k that uses FPGA} \tag{13}$$

For a given node v_i, the number of preconfigured FPGAs P_{fp_i}, the number of reconfigured FPGAs C_{fp_i} and the total number of FPGAs used R_{fp_i} are related by the following relations.

$$\begin{array}{rcl} R_{fp_i} & = & P_{fp_i} + C_{fp_i} \\ P_{fp_i} & = & R_{fp_i} - Nc_i \\ & & \text{where N is a constant} \gg R_{fp_i} \end{array} \tag{14}$$

The first constraint is obvious. The second one implies that when the FPGA allocated to node v_i needs to be reconfigured ($c_i = 1$), the node needs no preconfigured FPGAs ($P_{fp_i} = 0$).

We now need to impose further restrictions on the scheduling of the nodes so that the FPGA reconfiguration tasks get 'hidden' when possible. The implication of reconfiguring an FPGAs used by a node is that even though the node v_i starts execution at time s_i (as computed by the constraints described in Section 3.2), the FPGAs allocated to that node get 'locked up' even before the execution starts. That is to say that no other node can really use these FPGAs during the time period between the start of reconfiguration till the end of execution of the node. Let s'_i denote the time when the reconfiguration starts and r_t (a constant) be the reconfiguration time. The start times s_i and s'_i are related by the following constraints.

$$s_i \geq s'_i + c_i r_t \tag{15}$$

What this constraint implies is that if the FPGA is not reconfigured ($c_i = 0$) then it can be used anytime ($s_i \geq s'_i, s'_i$ is 0 by default) since it is pre-configured. In case a reconfiguration is needed ($c_i = 1$), then it can be used only after a delay of r_t from the time s'_i during which it is reconfigured. It is this s'_i (in the place of s_i) that is used in Constraint 8 to check the overlap if the nodes have an implementation that uses FPGAs. We need to impose further constraints if the CDFG is pipelined. We skip those details here and refer to [12].

Communication Interfaces. The number of communication interfaces needed is computed in a manner similar to that described in Section 3.4. We treat each communication interface as a resource and count not only those needed for intra node communication but also inter-node communication. Further, the number of communication interfaces needed for a given type of node is computed separately.

Mutually Exclusive Control Paths. Allocation of resources to mutually exclusive control paths is simple in our formulation assuming that a pre-processing step has identified all such paths. Two nodes v_i and v_j can share the same resources if they happen to be on two mutually exclusive paths. For such nodes we don't need the variables z_{ij}, z'_{ij} described in Section 3.4 and we can assume that their execution times don't overlap.

3.5. Objective Functions

Depending on the goals of an application, our MILP formulation can have a different objective function. The most common goals are to minimize the resource costs given a set of timing constraints to be met by the system. Alternatively, the goal could be to minimize the total execution time of the application given a set of resources. Other goals such as power minimization, area minimization are also possible [12].

Minimizing the Resource Cost. In a heterogeneous system one commonly used cost measure is the dollar cost of the resources used in the synthesized system. This cost has to be minimized without violating the timing constraints such as deadlines and throughput requirements. Obviously the objective function in this case is the sum of the costs of the compute and communication resources.

$$\begin{aligned} OBJ \;=\; & \textstyle\sum_i N_i l_i + \sum_j N_j l_j \\ & \forall \text{ comp. resource of type } i \\ & \forall \text{ comm. resource of type } j \end{aligned} \tag{16}$$

where l_i and l_j are dollar cost per resource.

When this objective function is minimized subject to all the constraints described in previous sections, the variables N_i and N_j evaluate to the number of resources of a given type needed to realize the least cost system that meets all the timing and throughput constraints.

Minimizing the Execution Time. In this case, the main objective is to choose the resources from a predefined pool of resources and schedule the nodes in such a way that the completion time of the program is minimum. The number of resources N_k in this case are predefined (they serve as upper bounds) and hence are constants. The total execution time of the CDFG is nothing but the start time s_{end} of the *exit node* v_{end} in the CDFG. We need to minimize this. So, the objective function in this case is simply the following.

$$OBJ = s_{end} \tag{17}$$

4. Experimental Evaluation

We have used a general purpose MILP solver [10] that uses *branch and bound search* technique combined with *LP relaxation*[9] to solve our model. A large set of real as well as synthetic benchmarks with upto 250 macro tasks (nodes in the CDFG) and more than 1000 subtasks is used to evaluate the algorithm. We have compared the quality of the results produced by our algorithm with that of a generic heuristic algorithm. This heuristic algorithm follows the same strategy as that followed by current heuristic algorithms [5, 6] where the main idea is to start with a low cost solution, iteratively refine it by improving the *critical paths* and schedule the resources until all the timing constraints are met.

As a specific example, we take one of the benchmarks namely the Space-time Adaptive Processing (STAP) [13] (refer to [12] for more details on benchmark specific results). STAP is a class of algorithms used in the area of airborne surveillance radars, used to detect weak target returns embedded in strong ground clutter. As can be seen in Table 2, for different combinations of total time T and inter-arrival time δ, the cost of the synthesized system using our algorithm is lower than that produced by the heuristic. The major part of this cost reduction comes from better resource selection and scheduling and in some cases from better pipelining.

Table 2. Synthesis of STAP application for various timing constraints

T	δ	Pipeline	Dollar cost		% Cost
ms	ms	stages	Heuristic	Our Algo	reduction
500	500	1	3480	2280	34
1400	700	2	3600	2160	40
500	250	2	5520	4320	22
750	250	3	5280	2880	45
400	100	4	9840	9360	5

We have done an exhaustive evaluation of our algorithm with more than 1000 test cases for CDFGs with upto 250 nodes and for different timing constraints. The evaluation is in terms of the cost reduction it can achieve as compared to heuristic, its ability to efficiently use the FPGAs, the closeness of the solutions to an estimated bound when the MILP search does not result in optimal solution and finally the time taken for the synthesis. We summarize these results in the following sections.

4.1. Improvement Over Heuristic

As can be seen in Figure 3, on an average, our algorithm produces results that are substantially better (20 to 50%) than the heuristic though for smaller graphs (upto 10 nodes) there are many cases where the heuristic also succeeds in producing the same quality result as our algorithm. Similar observation is made by others [6] where they assert that their heuristic algorithm produces the same results as that produced by exact techniques for small graphs. This is because, for small graphs, the design space is relatively small and so there is higher likelihood of a good heuristic finding an optimal solution. However, for larger CDFGs this does not seem to be the case.

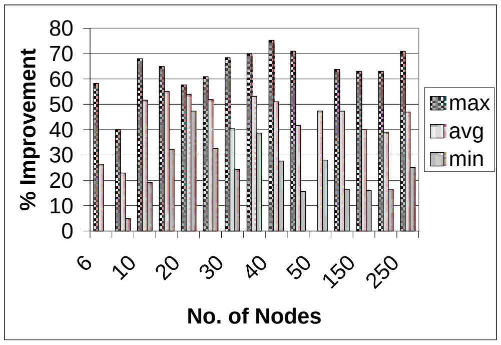


Figure 3. Improvement in the cost of the solution as compared to that produced by heuristic algorithm, for different test cases.

4.2. Reuse of FPGAs

We tried three different types of FPGAs with reconfiguration times of 10ms,100ms and 1000ms and ran a large number of cases to see how efficiently the algorithm utilizes the FPGAs. Figure 4 summarizes these results. For smaller reconfiguration times (10ms), the algorithm succeeds in reusing all the FPGAs in as many as 80% of the cases. For larger reconfiguration times (1000ms) though, less than 10% of the FPGAs are reused in half the cases. This indicates that for some applications we need to use FPGAs with faster reconfiguration times since there is no way of reusing the FPGAs without violating the timing constraints.

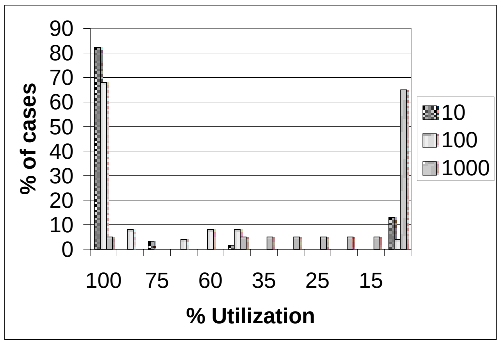


Figure 4. % Reuse of the FPGAs in different cases.

4.3. Closeness to the Bounds

Since the synthesis problem at hand is known to have several NP-complete subproblems, it is unlikely that any algorithm would produce optimal result in reasonable time for all cases. Fortunately, many of the MILP solvers allow us to terminate the branch and bound search after a specified time out. The solution obtained may not be optimal, but these solvers estimate how close these solutions are to optimal by computing a bound. Though these bounds in general may not be very tight, they do provide an idea of how close we are to the optimal when the search was terminated. In our test runs we imposed a stringent timeout of 10 minutes to see the quality of results that can be expected in such a short time. As can be seen from Figure 5, in 60% of the cases, the solver succeeded in finding the optimal solution much before the timeout.

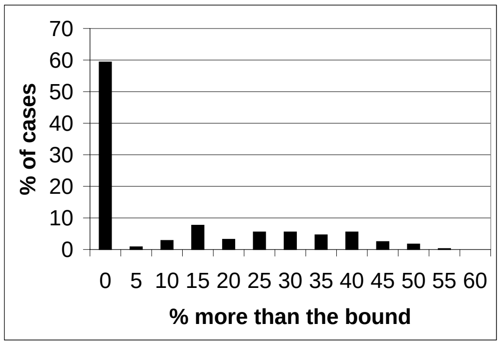
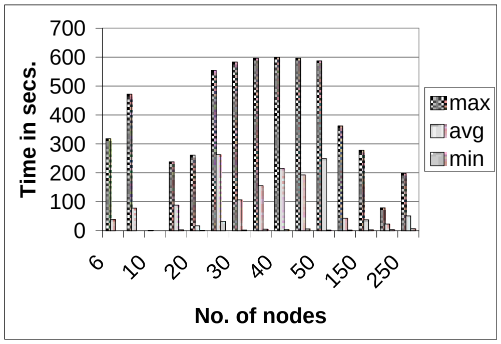
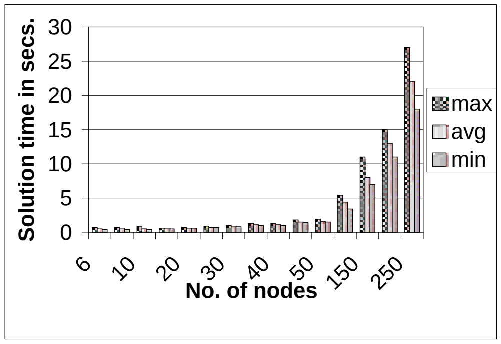


Figure 5. Deviation from an estimated bound when the search was terminated.

However, there were cases when the suboptimal solution was more expensive than the bound though it was generally much better than that found by heuristic. It is quite possible that these 'suboptimal' solutions would indeed turn out to be 'optimal' solutions given sufficient time and if the bound computed were sufficiently tight. In fact, when we increased the timeout to 30 minutes, 72% of the cases terminated with optimal results though the average decrease in the cost was marginal. A timeout of 10 minutes seems to be a good choice.



(a) Our Algorithm.



(b) Heuristic Algorithm.

Figure 6. Synthesis times for different sizes of test cases

4.4. Synthesis Time

One of the major concerns in automatic synthesis is the synthesis time and how it varies with the size of the problem. In Figure 6(a), we show the maximum/ average/ minimum time taken for synthesizing CDFGs of various sizes for different timing constraints. The average synthesis time for our algorithm is reasonably small (a few minutes), though there were cases that did not succeed in finding optimal solution within the preset timeout (10 mins). However, it is interesting to note that unlike the heuristic algorithm (refer Figure 6(b)), where it seems that the synthesis time is predominantly decided by the size of the CDFGs, there does not seem to be any strong relation between the number of nodes and the solution time of the MILP solver. The solver seems to be more sensitive to the structure of the CDFG (and hence the shape of the search space) rather than just the number of nodes.

5. Conclusion

In this paper we presented a system-level synthesis algorithm based on Mixed Integer Linear Programming technique that captures pipelining, selection, allocation, scheduling and reconfiguration of FPGAs in a single formulation. Our model solves both time constrained and resource constrained synthesis problems. Our experimental results indicate that in most cases our algorithm produces better results as compared to heuristic techniques (and never worse), succeeds in reusing the FPGAs across the tasks by hiding reconfiguration times, gives a clear idea about how close the solution is to the optimal in cases where optimal solution could not be found in limited time. It achieves all this in reasonable time making our technique useful for practical applications.

References

[1] Centre for Parallel and Distributed Computing, Northwestern University, “MATCH: A MATLAB Compilation Environment for Distributed Heterogeneous Adaptive Computing Systems”, http://www.ece.nwu.edu/cpdc/Match/Match.html,

[2] Jos T.J. and Leon Stok, “A Data Flow Graph Exchange Standard”, *Proc. of the Eur. Conf. on Design Automation (EDAC)*, Brussles, Belgium, March 1992, pp. 193-199.

[3] S. Prakash and A. Parker, “SOS: Synthesis of application-specific heterogeneous multiprocessor systems”, Jl. of Parallel and Dist Processing, pp. 338-351, Dec 1992.

[4] Y.G.Decastelo,M.Potkonjak and Alice Parker,“Optimal ILP-based Approach for Throughtput Optimization Using Simultaneous Algorithm/Architecture Matching and Retiming”,*Proc. of Design Automation Conf. (DAC-95)*, San Francisco, CA, 1995, pp. 113-118.

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

[6] Bharat Dave and Niraj Jha, “COHRA: Hardware-Software C-Synthesis of Hierarchical Distributed Embedded System Architectures”, *Proc. of VLSI Design’98*, pp.347-354.

[7] 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.

[8] Robert Dick and Niraj Jha, “CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-time Distributed Embedded Systems”,*Proc. of ICCAD98*,pp. 62-68, San Jose, CA, 1998.

[9] George Nemhauser and Laurence Wolsey, “Integer and Combinatorial Optimization”,Wiley-Interscience Publication.

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

[11] Daniel Gajski et.al., “High-level Synthesis”,Kluwer Academic Publications.

[12] U.Nagaraj Shenoy, Prith Banerjee and Alok Choudhary,“An MILP Based Algorithm for Automatic System Level Synthesis”,Technical Report CPDC-TR-9903-003, Northwestern University, March 1999.

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