An Algorithm for Synthesis of Large Time-Constrained Heterogeneous Adaptive Systems

NAGARAJ SHENOY, ALOK CHOUDHARY, and PRITHVIRAJ BANERJEE
Northwestern University

Large time-constrained applications are highly computer-intensive and are often implemented as a complex organization of pipelined data parallel tasks on a pool of embedded processors, DSP processors, and FPGAs. The large number of design alternatives available at each task level, the application as a whole, and the special needs of the reconfigurable devices (such as the FPGA) make the manual synthesis of such systems very tedious.

The automatic synthesis algorithm in this paper combines exact (MILP-based) and heuristic techniques to solve this problem, which basically involves (1) propagation of timing constraints; (2) pipelining the loops to meet throughput requirements; (3) resource selection and allocation, keeping the processing requirements and the timing constraints in view; (4) scheduling the resources across the tasks to ensure maximum utilization; and (5) hiding the reconfiguration delays of the FPGAs.

While the use of MILP techniques helps in getting high-quality results, combining them with heuristics ensures acceptable synthesis times, striking a good balance between quality of results and synthesis time. Our experimental evaluation of the algorithm shows an average 40% in resource cost reduction (compared to manual synthesis) with synthesis times from minutes to as low as a few seconds in some cases.

Categories and Subject Descriptors: J.6 [**Computer Applications**]: Computer-Aided Engineering—*Computer-aided design* (CAD); C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Signal processing systems*; *Real-time and embedded systems*

General Terms: Algorithm, Design, Experimentation

Additional Key Words and Phrases: Delay/cost table, hierarchical control data-flow graph, list scheduling, mixed integer linear programming, pipelining, reconfigurable computing, time-constrained synthesis

This work was supported by DARPA under contract F30602-98-2-0144 and F30602-97-C-0026.
Authors' address: Electrical and Computer Engineering, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208–3118; email: banerjee@ece.nwu.edu.

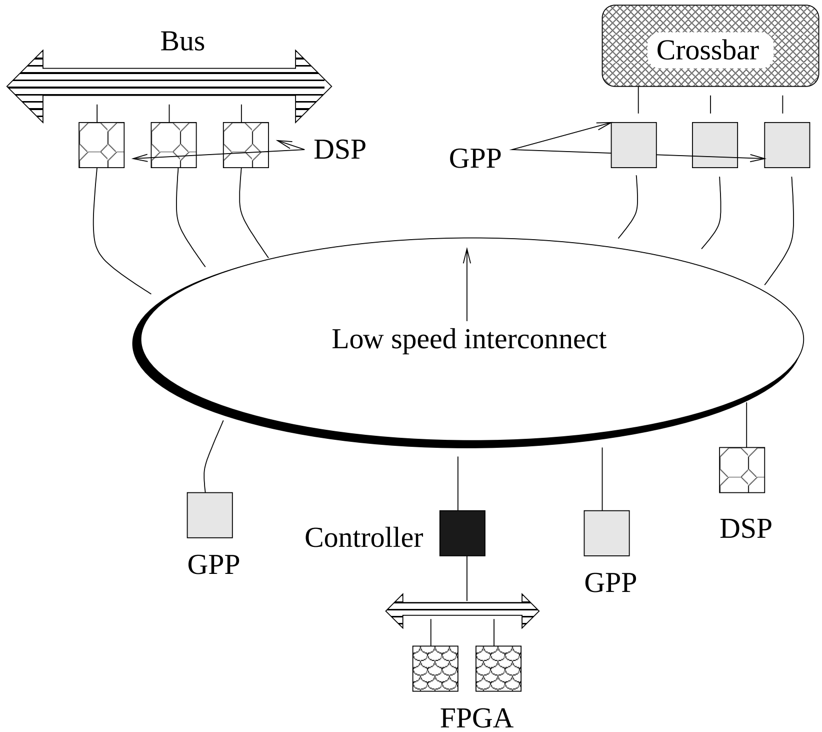

Fig. 1. A large time-constrained system implementation using heterogeneous resources (DSP and FPGA are generic devices in their respective categories).

1. INTRODUCTION

Several computer-intensive real-time signal-processing applications such as the *space time adaptive processing (STAP)* [Brown and Linderman 1997] in the area of airborne surveillance radars, have stringent requirements on processing and response times. Different phases of these applications exhibit different computation granularities and degrees of inherent parallelism.

The current trend is to implement each subtask in the computation by using a combination of off-the-shelf devices such as general-purpose processors, DSP processors, and field programmable gate arrays (FPGAs), exploiting both data and functional parallelism. While general-purpose processors and DSPs handle the bulk of the coarse-grained computational requirements of the application, FPGAs provide fast hardware implementations of time-critical parts. These FPGAs can be reconfigured on the fly to make the system *adaptive* to the requirements of the application at various phases of computation. Figure 1 shows a typical scenario.

These systems are heavily pipelined at various levels and each macro task is implemented as a collection of tightly-coupled data parallel tasks, as shown in Figure 2. The introduction of heterogeneity and adaptability adds additional dimensions to the complexity of the design. Manual synthesis of

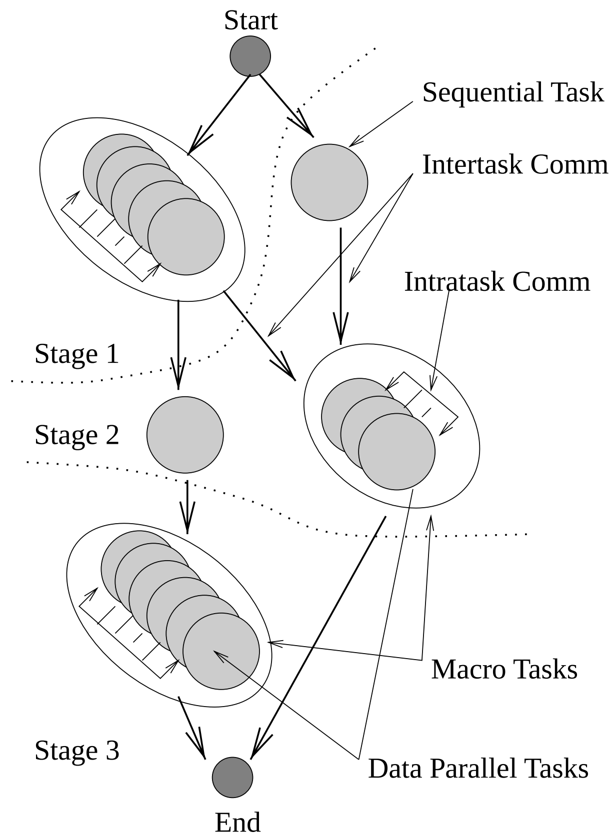

Fig. 2. Task/data parallel implementation of a large time-critical application.

such systems is extremely tedious due to the availability of a large number of implementation alternatives, making automatic synthesis techniques very attractive.

In a typical scenario, a synthesis algorithm works on what are known as *hierarchical control data-flow graphs* (HCDFG), which abstract various macro tasks and their interdependencies, of both data-flow and timing requirements (see Figure 3 for a simple example of such a HCDFG). An HCDFG is a directed graph with the nodes representing the *macro tasks* to be performed and the edges indicating the flow of data/control across these tasks. These edges can also represent timing constraints, where they indicate the allowable time from a source task (typically a task that reads the input) to a destination task (typically the one that initiates some action). Some of the tasks (*abstract tasks*) in turn stand for a lower level HCDFG. These abstract tasks can be loop bodies comprising several tasks, in which one of the tasks reads the incoming data and others perform various processing assignments, and the final one initiates some action based on the processed data. This can impose both *throughput* and *total delay* constraints on such an abstract task.

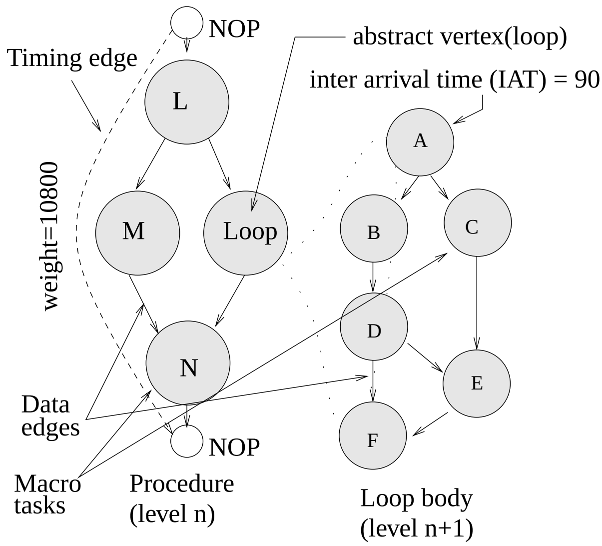

Fig. 3. An example hierarchical control data-flow graph (HCDFG).

Given such an abstract representation of the computation, a synthesis algorithm has to solve several subproblems (many of them provably NP-complete) before it can arrive at the right architecture to perform the computation which meets all the constraints. Some of these subproblems include constraint propagation, pipelining, resource selection, allocation, scheduling, and hiding the reconfiguration times of the FPGAs. More formally, it needs to do the following:

Given

(1) a hierarchical control data-flow graph (HCDFG) capturing the computation to be performed;
(2) timing and throughput constraints;
(3) possible resource types to be used in the design;
(4) design alternatives in the form of delay/cost tables.

Synthesize the program to arrive at a system with minimal total resource cost.

Such that the timing (indicated by timing edges) and throughput (indicated by interarrival-time IAT) constraints are not violated.

In this paper we describe an algorithm that *propagates* the timing constraints from a higher-level HCDFG to lower levels; *selects* and *allocates* the right type/ number of devices to implement each macro task; *pipelines* the tasks (if necessary) to achieve throughput requirements; *schedules* the resources to ensure optimal utilization and takes into account reconfiguration delays of the FPGAs.

We model pipelining, selection, and allocation of resources as a mixed integer-linear programming problem (MILP) to minimize the cost (dollar cost, for example) of the synthesized system. To further reduce the cost of the system, we use a simple variant of the *list-scheduling algorithm* [De Micheli 1998] to schedule the resources, taking into account pipelining, mutually exclusive control paths, and the reconfiguration delays of the FPGAs. To guide our synthesis process, we use *delay/cost* tables (*delay* meaning the execution time for a specific implementation of a node and the *cost* indicating the corresponding resource cost) that encapsulate the various design alternatives for each node in the HCDFG.

The remainder of this paper is organized as follows. We briefly discuss related work in system-level synthesis in Section 2. Our synthesis algorithm is covered in Section 3. We present the experimental results in Section 4, and conclude in Section 5.

2. RELATED WORK

Various aspects of automatic synthesis in real-time systems have been examined by several researchers [Kalavade and Lee 1992; Chou et al. 1995; Ernst et al. 1994; Gupta and De Micheli 1993; 1992; Bakshi and Gajski 1997; Dick and Jha 1998; Dave and Jha 1998; Prakash and Parker 1992; Decastelo et al. 1995; and Karkowski and Corporaal 1998]. Many of the problems in system-level synthesis have their counterparts in high-level synthesis [Gajski et al. 1998; De Micheli 1998]. In this section we discuss some of the earlier efforts that are closely related to our work and compare and contrast them in terms of problems, architectures, and techniques.

In terms of the problems, Prakash and Parker [1992]; Dave and Jha [1998]; and Decastelo et al. [1995] all deal with automatically synthesizing task graphs using heterogeneous components. However, each has a different emphasis. While Prakash and Parker [1992] and Dave and Jha [1998] focus on intertask communication-related issues, Decastelo et al. [1995] and Bakshi and Gajski [1997] emphasize achieving high throughputs. Our current work takes into account both communication and throughput-related issues.

Unlike the problem we address in this paper, these algorithms assume single-processor implementation of each task and do not address issues related to reconfigurable devices. An algorithm for multiprocessor task implementation is proposed by Karkowski and Corporaal [1998] with an emphasis on program transformations and a parallelization strategy for each task. Dick and Jha [1998] propose an algorithm for system design using FPGAs, and focus on reusing the FPGAs across tasks. It tries to minimize reconfiguration time by sequencing the tasks optimally onto the FPGAs, whereas our algorithm tries to achieve this by *configuring in anticipation of future use (latency hiding)*.

In terms of techniques, Bakshi and Gajski [1997]; Dave and Jha [1998]; and Karkowski and Corporaal [1998] employ greedy heuristics (based on iterative refinement) and Dick and Jha [1998] base their technique on list scheduling and evolutionary programming. While the use of greedy heuristics has an advantage in terms of a fast solution, the use of randomized algorithms may not necessarily result in low synthesis times.

In the context of multiprocessor task implementation, not only the scheduling but the selection and allocation of resources to each task (type and number of resources) is crucial. The design space becomes very large and an algorithm based purely on a greedy heuristic is less likely to find aa good solutions in a reasonable time.

As an alternative to heuristic-based solutions, Prakash and Parker [1992] and Decastelo et al. [1995] proposed MILP techniques to solve the synthesis problem which have the potential to produce optimal solutions. However, their models are restrictive and do not address all the issues we deal with in the this paper. For example, the cost model in Decastelo et al. [1995] does not seem to take resource sharing across tasks into account. Resource sharing is very important in reducing the cost of a synthesized system. Further, it is not clear whether these models are time-efficient. While Prakash and Parker [1992] deal with small task graphs (fewer than 10 nodes) and report solution times in hours; Decastelo et al. [1995] report solution times (for algorithm selection only) on the order of minutes. Neither of these algorithms addresses issues related to FPGAs or parallel implementation of individual tasks.

Our algorithm is unique because it combines the power of MILP techniques for the optimal solution of the selection, allocation, and pipelining problems with the speed of heuristic techniques for solving the constraint propagation and scheduling problems. We believe that this is a good balance between speed and the quality of the result. Our algorithm aims at designing large systems with a heterogeneous pool of resources, which exploits parallelism not only across tasks but also within each task. It makes good use of pipelining techniques to increase throughput. And it addresses one of the main problems in the use of reconfigurable devices, namely reconfiguration delays. By cleverly overlapping reconfiguration of the FPGAs with the computations in the preceding tasks, our algorithm allows efficient use of the FPGAs.

3. THE SYNTHESIS ALGORITHM

Our algorithm works on a hierarchical control data-flow graph (HCDFG), which is captured from a high-level sequential description of the computation after flow analysis. We employ a hierarchical synthesis algorithm to match the hierarchy in the control data-flow graph representation of the given program. The Synthesis algorithm goes recursively through each abstract vertex in the given HCDFG and synthesizes each one in a bottom-up fashion. A higher-level graph is synthesized after all its lower-level graphs (represented by abstract vertices) are synthesized.

Our algorithm is comprised of (1) a heuristic-based constraint propagation phase; (2) an MILP-based selection and pipelining phase; and (3) a heuristic-based scheduling phase. The following sections discuss these phases.

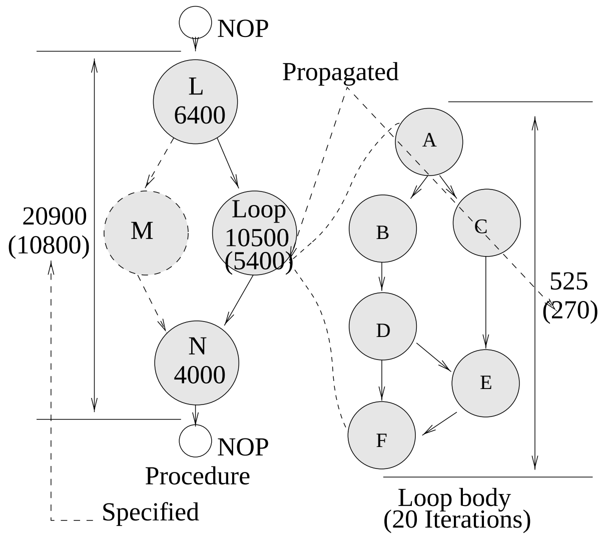

Fig. 4. Constraint propagation for the example HCDFG (the numbers without parenthesis show median delays and those within parenthesis show propagated delays).

3.1 Propagating Timing Constraints

In order to set timing bounds for each of the lower-level graphs, we propagate the timing constraints specified at a higher- to lower-level graph. These propagated timing constraints are set as timing edges in the lower level. As discussed in Section 1, a timing constraint in a HCDFG is suggested by a timing edge from a vertex v_s to a vertex v_e. There can be one or more paths in the HCDFG that include both v_s and v_e. We call the sequence of vertices in these paths, starting from v_s and ending at v_e, a *path segment*, and the sum of the execution times of each vertex in a given path segment (excluding v_s and v_e), the *delay of the path segment*.

A timing edge with a weight t indicates that $\forall p$, $delay(p) \leq t$, where p is any path segment between v_s and v_e. This puts a constraint on the execution times of the vertices that lie on these path segments. Section 3 discusses how these timing constraints are handled in general; but the hierarchical synthesis algorithm requires that the timing constraints at a specific level of the graph hierarchy be propagated to each of the abstract vertices at that level if they happen to lie on any one of the path segments constrained by a timing edge.

To illustrate our constraint propagation algorithm, let us assume a timing edge of weight t between vertices v_s and v_e. Let $s = <v_i, v_{i+1}, \ldots, v_{i+n}>$ be the set of vertices on a given path segment between v_s and v_e (excluding v_s and v_e). Let $v_{i+k} \in s$ be an abstract vertex. We need to propagate the timing constraint t as an execution time bound on the execution of the vertex v_{i+k}. We do this as follows: We first compute the median delays for each of the vertices in s. Let m_{tot} denote the sum of the

```
ALGORITHM : Propagate
INPUT :     HCDFG G
            delay/cost table T
OUTPUT :    G with time bounds
            for abstract vertices.
begin
    for each vertex v ∈ G
        Delay(v)← median delay
        Bound(v)← ∞
    End
    for each timing edge < v_s, v_e >∈ G
        for each path p_i from v_s to v_e
            s ← sum of median delays of vertices ∈ p_i
            for each abstract vertex v_k ∈ p_i
                new_bound = Weight(< v_s, v_e >) Delay(v_k)/s
                Bound(v_k)← min(Bound(v_k),new_bound)
            End
        End
    End
end
```

Fig. 5. Algorithm: Propagate.

median delay of each of the vertices in s and let m_a denote the median delay of vertex v_{i+k}. We estimate the execution time bound on v_{i+k} in the path segment s as

$$t_s = t\frac{m_a}{m_{tot}}.$$

Further, if the vertex v_{i+k} lies on path segments $<s_1, s_2, \ldots, s_m>$, then the time constraint t_a on the abstract vertex v_{i+k} is given by

$$t_a = \min_{s \in <s_1, \ldots, s_m>} t_s.$$

For our example program (shown in Figure 3), assuming a set of design alternatives (skipped for brevity), our algorithm propagated a timing constraint of 10800 (specified at the procedure level) to the loop body (an abstract vertex in the procedure) as shown in Figure 4. We outline our *propagation algorithm* in Figure 5.

Once the constraints are propagated, our algorithm synthesizes the HCDFG following the inherent hierarchy in the graph. At each level of hierarchy, the algorithm solves the pipelining, selection, allocation, and scheduling problems guided by the estimates provided by the delay/cost tables.

Table I. Notation In the Formulation

t	Total processing delay.
T	Interarrival time (IAT)
$t_{c_{ij}}$	Weight of the timing edge between nodes i and j.
N_v	Number of nodes in the CDFG
N_e	Number of data edges in the CDFG
N_{a_i}	Number of design alternatives for node i
N_{b_i}	Number of design alternatives for edge i
d_{ik}	Execution time of i^{th} node if k^{th} alternative is selected.
x_{ik}	Communication time of i^{th} edge if k^{th}alternative is selected.
c_{ik}	Cost (dollar cost, for example) of k^{th} implementation of i^{th} node.
l_{ik}	Cost (dollar cost, for example) of k^{th} implementation of i^{th} edge.
D_i	Execution time of i^{th} node after selection.
S	Number of pipeline stages.
	Model variables
s_i	Start time of i^{th} node.
$a_{ik} \in [0, 1]$	Stands for selection of the k^{th} alternative for the i^{th} node.
$b_{ik} \in [0, 1]$	Stands for selection of the k^{th} alternative for the i^{th} edge.
$p_i \in [0..S - 1]$	Pipeline stage of node i

3.2 Selection and Pipelining

Our selection and pipelining algorithm is based on Mixed Integer Linear Programming (MILP). We model the various constraints imposed by the control data-flow graph (in terms of data and timing edges) and minimize the objective function that is the total cost of the resources used in the synthesis. Table I lists the notation in our formulation.

3.2.1 *The MILP Formulation*. The *selection* has to do with choosing a specific implementation for each task as well as each data edge from among various available alternatives. Depending on the alternative chosen, the cost of implementing the node as well as the interconnection network could change. The resource cost of a macro task depends on the processing elements used and the interconnect for *intra* and *inter task* communication.

Communication costs incurred by a macro task could be due to intramacro task communication (among the data parallel tasks in the macro task) or the intermacro task communication due to the data dependence between macro tasks.

Intramacro task communication costs depend on the interconnect used to support the communication (refer to Table II). The *effective time* taken by the communication across the tasks within a macro task for a given interconnect/implementation pair contributes to the total *execution time* of the macro task. The resource cost of the interconnect is split into two components. A fixed *interconnect cost* is incurred (for the system as a whole) whenever an interconnect of a given type is used in the final synthesis by any of the devices. In addition, *an interface cost* is charged to each device that communicates using such an interconnect.

Intermacro task communication depends on a variety of things. Since each *data edge* in the HCDFG represents one such instance of the communication,

Table II. Intramacro Task Communication Cost Table

Task	Impl. 1		Impl. 2		Impl. 3	
	Res.	Time	Res.	Time	Res.	Time
t1	xbar	10	lan	90	bus	50
t2	bus	10	bus	20	NA	
t3	xbar	60	xbar	70	bus	80
...	...	...	...	...	...	...
tn	bus	5	bus	10	NA	

Table III. Intermacro Task Communication Cost Table (A '*' is any implementation; a list of names within parenthesis means a list of implementations)

Edge	Impl. 1		Impl. 2	
	Res.	Time	Res.	Time
e1	$<S1, D2, xbar>$	30	$<*, *, bus>$	40
e2	$<S3, *, bus>$	50	$<*, *, xbar>$	10
e3	$<S1, (D2..D5), lan>$	90	$<*, *, bus>$	40
...	...	...	...	...
en	$<*, *, xbar>$	30	$<*, *, bus>$	80

our communication cost table needs to list different implementation alternatives that support such a communication. Depending on the situation, such a communication could depend on the *sender task* only, the *receiver task*, or both. Further, the implementations chosen for the sender and receiver tasks may influence the cost of the communication in addition to the actual interconnect used for the communication.

In general, each implementation alternative for a data edge is a triplet $<s, d, r>$ with an associated communication time. The s and d in the triplets stand for source and destination implementations, and r stands for the type of interconnect used for the communication. The *source* and the *destination* can be specified in a variety of ways (see Table III).

One and only one of the implementations for a given node/edge can be selected. This results in the following constraints.

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

$$\sum_{k=1}^{N_{b_i}} b_{ik} = 1, \forall i, 1 \leq i \leq N_e \tag{2}$$

The selection of an implementation for a data edge is further bound by the constraint that the selection should match the source and destination implementation selections. That is, b_{jk} can be true *only if* the a_{ik} corresponding to s and d in the corresponding triplet $<s, d, r>$ are true. For example, if the triplet is $<S1, D2, r>$, then the constraint $b_{jk} \leq y$ (where

y is an auxiliary variable, which is set to true if and only if both the variables a_{s1} and a_{d2} are true) should be met.

The execution time D_i of a node v_i (after selection) is modeled as the sum of C_i (sum of *computation time* and *intramacro task communication times*) and X_i (sum of the *intermacro task communication time* corresponding to each data edge emanating from the node).

$$D_i = \sum_{k=1}^{N_{a_i}} a_{ik} d_{ik} + \sum_{\forall e \in \text{ out edge}} \sum_{k=1}^{N_{b_e}} b_{ek} x_{ek}$$

Using the start time variables s_i, we impose 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 ($v_i \prec_{\text{data}} v_j$) implies that the start time s_j of node v_j cannot be less than the end time of the node v_i.

$$s_j \geq s_i + D_i \tag{3}$$

Timing constraints. 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.

$$s_j \leq s_i + D_i + t_{c_{ij}} \tag{4}$$

An overall processing time constraint of t units of time (either specified or propagated) from the *start* node to *end* node implies

$$s_{end} \leq s_{start} + t \tag{5}$$

Pipelining constraints. 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 interarrival 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. Using the variables p_i we model the pipelining requirements as

$$D_i \leq T \tag{6}$$

$$T * p_i \leq s_i \leq T * (p_i + 1) - D_i \tag{7}$$

Objective function.The main objective of the selection problem is to arrive at a solution that employs resources whose total cost is the minimum.

```
ALGORITHM : ScheduleAllocate
INPUT :     HCDFG G with ALAP schedule
            and selection.
OUTPUT :    G with allocation
begin
      resPool← ∅
      Mark start node
      clock← 0
      while there are unmarked nodes
            for each unmarked node v ∈ G
                if State(v)=RUNNING then
                   if clock=Start(v)+Delay(v) then
                      Free resources
                      Mark node v
                   end if
                else if Marked(v_p)=true ∀v_p = Pred(v)
                    and (resources are available
                           or Alap(v)=clock) then
                       State(v)←RUNNING
                       Allocate resources to v
                end if
            end for
            clock← clock +1
      end while
end
```

Fig. 6. Resource allocation and scheduling.

Using the boolean variables a_{ik}, b_{jk}, and cost values c_{ik}, l_{jk} we can express this objective function as

$$C = \sum_{i=1}^{N_v} \sum_{k=1}^{N_{a_i}} c_{ik} a_{ik} + \sum_{j=1}^{N_e} \sum_{k=1}^{N_{b_j}} l_{jk} b_{jk} \tag{8}$$

3.3 Allocation and Scheduling

After the implementations for each node and edge are selected (as discussed in the previous section), the scheduler first computes the *as late as possible (ALAP)* schedule to decide how a given node can be delayed before starting execution. The scheduler allocates both the processing elements (PE) and interconnects (IC) in a conservative fashion guided by these schedules.

The algorithm *ScheduleAllocate* (simplified for the sake of illustration) sketched in Figure 6 starts with an empty pool of resources and picks a node for scheduling when all its predecessors have finished. If required resources are available in the pool, the same resources are allocated to the node and the node is scheduled. If the required resources are currently held by some other node, then the scheduling is delayed until either the resources become available or the ALAP schedule for the node is reached.

If by that time some other node scheduled earlier has released all or some of the resources needed by the node, same are allocated to the current node. If sufficient resources are not available in the pool, then new resources are allocated. The resources are held with a node until the clock advances (the advancement is shown as single time step in Figure 6 for simplicity) by the time denoted by the delay of the node. It should be noted that this scheduling is done at synthesis time (static) and not at runtime.

3.3.1 *Allocation of Interconnects*. The scheduler needs to allocate not only the interconnects needed by the PEs implementing a node (for intramacro task communication), but also those needed by the data edges (intermacro task communication) between these nodes. Allocation of interconnects is slightly more involved as compared to allocation of PEs. A PE is always viewed as attached to an IC and remains so permanently. While a PE can be simultaneously attached to more than one IC (assuming that it has that many interfaces), a specific IC may have limits on the number of PEs that can be attached to it. The cost of an IC may depend on the number of PEs connected to it and it may often increase in discrete steps. For example, crossbar switches may come with capacities of multiples of 4 PEs.

Our strategy to allocate the ICs is to use the same IC for multiple nodes/edges wherever possible, effectively reducing the resource costs. When we need to allocate a new IC, our scheduler tries to meet this requirement by an already allocated IC. If required, it could upgrade an existing IC to a higher capacity to accommodate new PEs. It also tries to change the selection of an IC for a node/edge if doing so does not violate the constraints and reduces the resource cost. If everything fails, then it could allocate a new IC. A brief sketch of this allocation strategy is outlined in Figure 7.

3.3.2 *Handling Reconfigurable Devices*. Our scheduler takes care of hiding the reconfiguration time of the FPGAs wherever possible. For this purpose it modifies the original CDFG by inserting additional *reconfiguration tasks*. Let us assume that a vertex v with a delay d_e is mapped to an FPGA. Let d_c denote the configuration time for this vertex. We can visualize the vertex v as two vertices v_c (the configuration task) and v_e (the computation task) each with corresponding delays d_c and d_e. We introduce a dummy dependence edge from v_c to v_e to indicate that v_e can start execution only when v_c has finished (i.e., FPGA is configured). Also, all the incoming edges of vertex v become incoming edges of vertex v_e. The vertex v_c has a single incoming edge from the *start* vertex.

With this modification of the HCDFG, for each vertex mapped to reconfigurable devices, we proceed as follows. A vertex such as v_c is scheduled as soon as its resource requirements are met by the current resource pool. If there are multiple vertices of the type v_c contending for a free resource, then we use their successor's ALAP schedule as priority to break the tie. The resources are returned to the pool as usual after the vertex v_e finishes.

```
ALGORITHM : Allocate
INPUT :     node (current node), respool(resource pool),
            HCDFG G with selection.

OUTPUT :    G with allocation
begin
     /* First allocate node resources */
     if there are no free PEs in respool then
        Add new PEs to respool.
     end if
     if these PEs are not already associated with reqrd IC then
        if they can be associated with a free IC then
           Associate them with the IC.
        else if a free IC can be upgraded then
                 Upgrade the IC and associate the PEs.
        else allocate a new IC and associate the PEs.
     end if
     Allocate the PEs with IC.

     /* Now allocate edge resources */
     for each incoming edge e
          if PEs in source and dest don't meet selection then
             if source IC can accommodate dest PE then
                Use source IC for the edge.
             else if dest IC can accommodate source PE then
                      Use dest IC.
             else if source/dest can be upgraded then
                      Upgrade and use the IC.
             Else
                      Allocate new IC and associate PEs.
             end if
          end if
     end for
end
```

Fig. 7. Resource allocation.

In case the vertex v_c cannot be scheduled before the clock reaches the ALAP schedule for vertex v_e, then v_e is assumed to be preprogrammed and the FPGAs are not reused for that vertex.

3.3.3 *Scheduling of Mutually Exclusive Control Paths.* The vertices between a *branch* vertex and the corresponding *merge* vertex could fall in nonoverlapping path segments. Since only one of the control paths emanating from a branch vertex can be active at any given instant of time, the vertices lying on all such mutually exclusive path segments can share resources.

Our scheduling algorithm takes advantage of this fact and tries to allocate the same resources to two or more vertices if they happen to fall in different mutually exclusive control paths. When a vertex v_j is picked up for scheduling and if the resource pool does not have the necessary resources, the scheduler checks all the currently *RUNNING* vertices and does the following. If a currently *RUNNING* vertex v_i has allocated resources needed by v_j and if v_i and v_j fall on two mutually exclusive control paths, it allocates some or all of the resources allocated to v_i also to v_j. The resources are freed only when both v_i and v_j finish.

3.3.4 *Refining Selection While Scheduling*. To take into account the interaction between selection and scheduling, we have incorporated the capability to change the selection (under some conditions) in our scheduler. Whenever a node becomes *ready* for execution and it becomes necessary to allocate new resources, the scheduler tries to choose an alternate implementation for the node if the resources for that implementation is available in the pool and the delay associated with that implementation is no more than that associated with the previously selected implementation.

4. EXPERIMENTAL EVALUATION

We currently have a Java-based implementation of our synthesis algorithm. The MILP problem is automatically generated from a CDFG, which in turn is fed to a public domain MILP solver [Berkelaar 2001]. The output of the solver is input to the scheduler.

We used a large number of benchmarks to evaluate our synthesis algorithm, including real applications such as *space time adaptive processing (STAP)*, MPEG decoder, as well as several synthetic benchmarks. In each case we compare the results generated by the automatic synthesizer with what could be produced by manual synthesis. We assume that a typical manual synthesis involves pipelining and resource allocation based on the relative computation weights of each of the tasks. The initial decisions are iteratively refined by using runtime measurements until the computation load is balanced and communication delays are minimized. At each step the allocation is done in a conservative fashion. We take the manual design techniques employed in Choudhary et al. [1998] as a guideline in designing large systems. Some of the results of the evaluation of our synthesis algorithm are listed in Tables IV(a)–(c) and V(a)–(c).

We synthesized each benchmark for various combinations of $<Delay, IAT>$ pairs. We compared the cost of the automatically synthesized system with the manually synthesized one. We separated the cost of the system before scheduling (shown as NS) and after scheduling (shown as WS) to highlight the contributions of these two synthesis phases. We also show the percentage cost reduction as compared to manual synthesis. The last column in each table shows the time taken for automatic synthesis.

In almost all cases, our algorithm generated substantially better quality results compared to the manually synthesized ones. The cost reduction is as high as 70% in some cases. The cost reduction is significant when, for various reasons, the CDFG are large (Synthetic programs 2 and 3). First, a

Table IV. Automatic synthesis of the benchmarks (real applications) for various timing constraints (*Delay* is the total delay; *IAT* the interarrival time; *Man.* and *Auto* for manual and automatic synthesis; *WS* and *NS* for costs with and without scheduling; *CR* for percentage cost reduction; *Syn.time* time for synthesis)

(a) STAP using homogeneous resources

Delay (msecs.)	IAT (msecs.)	#of Procs Man.	Auto. NS	Auto. WS	Auto. CR	Syn. time (secs.)
1250	1250	60	60	44	27%	0.1
700	700	83	78	50	40%	0.1
365	365	108	106	60	44%	0.1
1400	700	68	60	48	29%	0.1
700	350	88	82	60	32%	0.5
360	180	148	136	102	31%	0.6
1500	500	62	62	56	10%	0.1
750	250	94	94	88	6%	1.7
360	120	156	152	140	10%	0.2

(b) STAP using heterogeneous resources

Delay (msecs.)	IAT (msecs.)	Cost in $ Man.	Auto. NS	Auto. WS	Auto. CR	Syn. time (secs.)
1250	1250	2800	2800	1920	31%	0.1
700	700	4375	3960	2140	51%	0.2
365	365	7710	6360	5460	29%	0.2
1400	700	3250	2800	2410	26%	0.1
700	350	4840	4345	3895	20%	0.7
360	180	11270	8250	7350	35%	0.5
1500	500	2920	2920	2520	14%	0.1
750	250	5140	4930	4240	18%	1.2
360	120	8600	8380	8380	3%	0.1

(c) MPEG using heterogeneous resources

Delay (msecs.)	IAT (msecs.)	Cost in $ Man.	Auto. NS	Auto. WS	Auto. CR	Syn. time (secs.)
110	110	375	375	135	64%	0.1
90	90	450	405	225	50%	0.1
100	50	495	495	315	36%	0.5
90	45	515	510	390	24%	0.3
75	25	540	540	540	0%	0.2
60	20	680	680	680	0%	0.4
60	15	805	775	775	4%	0.1
50	10	1145	1145	1145	0%	0.1

large graph provides a large number of possibilities to pipeline the graph, and the quality of the result depends on the right pipelining. Second, the size of the search space for resource selection is very large in the case of large graphs, making the simple greedy heuristics used by the manual

Table V. Automatic synthesis of benchmarks (synthetic) for various timing constraints (*Delay* is total delay; *IAT* is interarrival time; *Man.* and *Auto.* for manual and automatic synthesis; *WS* and *NS* for costs with and without scheduling; *CR* stands for percentage cost reduction; *Syn.time* is time taken for synthesis)

(a) Synthetic program 1 using heterogeneous resources

Delay (msecs.)	IAT (msecs.)	Cost in $ Man.	Auto. NS	Auto. WS	Auto. CR	Syn. time (secs.)
1400	200	1785	1785	1725	3%	2.0
1800	300	1605	1320	1080	33%	1.2
2400	400	1530	1260	960	37%	1.6
2000	500	1320	1305	825	38%	2.8
1800	600	1605	1260	765	52%	0.7
1400	700	1545	1380	840	46%	3.1
800	800	2025	1880	950	53%	0.7
650	650	2875	2255	1355	53%	1.7

(b) Synthetic program 2 using heterogeneous resources

Delay (msecs.)	IAT (msecs.)	Cost in $ Man.	Auto. NS	Auto. WS	Auto. CR	Syn. time (secs.)
2800	400	2670	2115	1500	44%	2.1
4200	600	2235	2115	1435	36%	0.2
3500	700	2475	2115	1380	44%	0.2
2400	800	2550	2145	915	16%	1.2
1800	900	2430	2295	1170	52%	0.4
2500	500	2835	2130	1530	46%	2.1
1200	1200	3345	3080	1055	68%	0.3
900	900	4700	3975	1800	62%	0.2

(c) Synthetic program 3 using heterogeneous resources

Delay (msecs.)	IAT (msecs.)	Cost in $ Man.	Auto. NS	Auto. WS	Auto. CR	Syn. time (secs.)
1400	700	5370	4665	1710	68%	**>300**
2600	650	5450	3405	1500	72%	228.0
4200	600	4665	3375	2025	57%	2.2
4500	500	3795	3375	2715	28%	3.2
4000	1000	3885	3375	1545	60%	2.7
1600	800	5025	4310	1760	65%	**>300**
3600	1200	3705	3375	1605	57%	2.1
2800	1400	3375	3375	1110	67%	0.4

synthesis unable to find a good solution (in spite of iterative refinement) in a given amount of time. In most cases, the contribution of scheduling to cost reduction is also significant, and larger graphs no doubt provide more opportunities for scheduling.

For a given graph, the cost reductions are not significant (in many cases it is almost zero) whenever the timing constraints are *tight* (low delays and low IAT). This is because most of the design alternatives become infeasible (for a given delay/cost table) for those timing constraints making the search space relatively small. Both manual and automatic techniques produce comparable results in such cases.

It is interesting to analyze the exact reason why a huge cost reduction was achieved in specific cases. For example, consider the case where our algorithm achieved a maximum cost reduction of 72% (second row of Table V(c)). More than half of this reduction is mainly due to better selection and pipelining (see column NS). While the manual synthesis employed 41 M68k, 24 PowerPC, 11 Pentium, and 1 FPGA to design this system, our algorithm chose 30 M68k, 13 PowerPC, and 7 Pentium processors. In both cases, cheaper resources are given preference, expensive devices such as FPGAs are selected only when essential, both partitioned the graph into the same number of stages (4 in this case). But the MILP-based pipeliner/selector resulted in a better assignment of pipeline stages to the tasks that not only helped in choosing fewer resources, but also in better scheduling.

In many cases the cost reduction was not because the MILP selector preferred low-cost devices (in fact it chose expensive devices), as would any greedy algorithm, but it chose the resources (even if it increased the implementation cost of some of the nodes), keeping the overall cost in view. There were other cases where the cost reduction came partly from reuse of FPGAs across the nodes by dynamically reconfiguring them.

As can be seen from Tables IV and V, the synthesis time in most cases is very small. In rare cases the MILP solver took an unduly large amount of time to come up with an optimal solution (see the highlighted cases in Table V(c)). We set a timeout of 300 secs, after which the MILP solver terminates the search and returns the best suboptimal solution. As can be seen from Table V(c), even in such cases the cost reduction achieved was quite impressive (around 65%).

5. CONCLUSION

In this paper we presented a synthesis algorithm that automatically performs constraint propagation, resource selection, allocation, pipelining, scheduling, and hiding of reconfiguration delays in the context of the design of large time-constrained parallel heterogeneous adaptive systems. Our algorithm combines the power and the elegance of MILP techniques to solve the selection and pipelining problems with the fast list scheduling-based heuristic to perform scheduling. Experimental evaluation of our algorithm using a large number of benchmarks shows that high-quality results can be obtained in reasonable amount of time.

REFERENCES

BAKSHI, S. AND GAJSKI, D. D. 1997. Hardware/software partitioning and pipelining. In *Proceedings of the 34th Annual Conference on Design Automation* (DAC '97, Anaheim, CA, June 9–13), E. J. Yoffa, G. De Micheli, and J. M. Rabaey, Chairs. ACM Press, New York, NY, 713–716.

BERKELAAR. 2001. lp_solve version 2.1. ftp://ftp.es.ele.tue.nl/pub/lp_solve.

BROWN, R. AND LINDERMAN, R. 1997. Algorithm development for an airborne real-time STAP demonstration. In *Proceedings of the IEEE National Conference on Radar*. IEEE Computer Society Press, Los Alamitos, CA.

CHOU, P. H., ORTEGA, R. B., AND BORRIELLO, G. 1995. The Chinook hardware/software co-synthesis system. In *Proceedings of the Eighth International Symposium on System Synthesis* (Cannes, France, Sept. 13–15), P. G. Paulin and F. Mavaddat, Eds. ACM Press, New York, NY, 22–27.

CHOUDHARY, A., LIO, W., WEINER, D., VARSHNEY, P., LINDERMAN, R., AND LINDERMAN, M. 1998. Design, implementation and evaluation of parallel pipelined STAP on parallel computers. In *Proceedings of the 12th International Symposium on Parallel Processing*.

DAVE, B. AND JHA, N. 1998. CORHA: Hardware-software C-synthesis of hierarchical distributed embedded system architectures. *IEEE Trans. Comput.-Aided Des. 17* (Oct.), 900–919.

DECASTELO, Y. G., POTKONJAK, M., AND PARKER, A. C. 1995. Optimal ILP-based approach for throughput optimization using simultaneous algorithm/architecture matching and retiming. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation* (DAC '95, San Francisco, CA, June 12–16), B. T. Preas, Ed. ACM Press, New York, NY, 113–118.

DE MICHELI, G. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., New York, NY.

DICK, R. P. AND JHA, N. K. 1998. CORDS: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '98, San Jose, CA, Nov. 8-12), H. Yasuura, Chair. ACM Press, New York, NY, 62–67.

ERNST, R., HENKEL, J., AND BENNER, T. 1994. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test* (Dec.), 64–75.

GAJSKI, D., DUTT, N., LIN, S., AND WU, A. 1992. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Hingham, MA.

GUPTA, R. K. AND DE MICHELI, G. 1993. Hardware-software cosynthesis for digital systems. *IEEE Des. Test 10*, 3 (Sept.), 29–41.

GUPTA, R. AND DEMICHELI, G. 1992. System-level synthesis using re-programmable components. In *Proceedings of the European Conference on Design Automation* (EDAC '92, Brussels, Belgium, Mar. 16 - 19). IEEE Computer Society Press, Los Alamitos, CA, 2–7.

KALAVADE, A. AND LEE, E. 1992. Hardware/software co-design using Ptolemy: A case study. In *Proceedings of the IFIP International Workshop on Hardware/Software Co-Design*. IFIP.

KARKOWSKI, I. AND CORPORAAL, H. 1998. Design space exploration algorithm for heterogeneous multi-processor embedded system design. In *Proceedings of the 35th Annual Conference on Design Automation* (DAC '98, San Francisco, CA, June 15–19), B. R. Chawla, R. E. Bryant, and J. M. Rabaey, Chairs. ACM Press, New York, NY, 82–87.

PRAKASH, S. AND PARKER, A. C. 1992. SOS: Synthesis of application-specific heterogeneous multiprocessor systems. *J. Parallel Distrib. Comput. 16*, 4 (Dec.), 338–351.

Received: May 1999; accepted: September 2000