A Fast Approach to Computing Exact Solutions to the Resource-Constrained Scheduling Problem

M. NARASIMHAN and J. RAMANUJAM[1]
Louisiana State University

This paper presents an algorithm that substantially reduces the computational effort required to obtain the exact solution to the Resource Constrained Scheduling (RCS) problem. The reduction is obtained by (a) using a branch-and-bound search technique, which computes *both* upper and lower bounds, and (b) using efficient techniques to accurately estimate the possible time-steps at which each operation can be scheduled and using this to prune the search space. Results on several benchmarks with varying resource constraints indicate the clear superiority of the algorithm presented here over traditional approaches using integer linear programming, with speed-ups of several orders of magnitude.

Categories and Subject Descriptors: B.5.1 [**Register-Transfer-Level Implementation**]: Design—*Data-path design*; B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic synthesis, hardware description languages, optimization*; B.m [**Miscellaneous**]: Design management

General Terms: Design, Optimization, Experimentation

Additional Key Words and Phrases: Design automation, high-level synthesis, resource-constrained scheduling, exact scheduling, lower and upper bounds, enumeration

1. INTRODUCTION

The scheduling problem plays a key role in high-level synthesis. Gajski et al. [1986] call it "perhaps the most important step during structure synthesis." In this paper, we consider the problem of Resource Constrained Scheduling (RCS), i.e., minimizing the latency of an implementation given a fixed set of resources. This problem is known to be NP-complete [Ullman 1975] in general, and so the existence of a polynomial time algorithm for solving this problem is highly unlikely. In many high-level synthesis systems, heuristics are employed since a designer may have to perform many iterations of the entire design process in a limited amount of time. These algorithms are generally fast and there has been much work in the development of good heuristics [Paulin and Knight 1989; McFarland et al. 1990; Camposano and Wolf 1991; Lin 1997]. In general these algorithms do not produce optimal solutions. Even in those cases where they produce an optimal solution, the methods do not guarantee that the solution produced is indeed optimal; so one would need to use other techniques to verify optimality.

In this paper, we present an efficient branch-and-bound algorithm that computes the optimal solution to the resource-constrained scheduling problem. Our algorithm tries to decompose the scheduling problem into a number of smaller sub-problems

[1]Corresponding author: J. Ramanujam. Address of the authors: M. Narasimhan and J. Ramanujam, Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. Email: {mukund,jxr}@ee.lsu.edu. Phone: (225) 578-5628, Fax: (225) 578-5200.
The material presented in this article is based on research supported in part by J. Ramanujam's NSF Young Investigator Award CCR-9457768 and NSF grant CCR-9210422.

many of which need not be solved because of the use of bounding techniques. It tries to bound the time for execution both from below and from above, rather than just bounding it from below as is done in conventional branch-and-bound. In addition, it does not use linear relaxation [Chaudhuri and Walker 1994] as the bounding technique, unlike ILP-based approaches to exact scheduling and design space exploration. These improvements enable our algorithm to quickly prune away large parts of the solution space, producing guaranteed optimal solutions quickly.

This paper is organized as follows. Section 2 presents the relevant terminology and notation, followed by a brief discussion of related work. Section 3 presents our search technique for the exact solution to the scheduling problem. Section 4 describes a technique for refining ASAP and ALAP (see Section 2 for definitions) values. In Section 5, we present descriptions of the lower and upper bounding techniques used in our search algorithm. We present experimental results on several benchmarks in Section 6; these results clearly demonstrate the superiority of our technique for over ILP approaches. Section 7 concludes with a summary and discussion.

2. TERMINOLOGY AND NOTATION

We assume that the problem to be scheduled is represented in the form of a Data Flow Graph (DFG). We will denote by $V(G) = \{x_1, \ldots, x_N\}$ the set of operations in the DFG, and by $E(G)$ the dependence constraints between those operations. A dependence constraint $\langle x_i, x_j \rangle$ implies that operation x_j depends on the result of operation x_i; therefore, operation x_i must be completed before operation x_j can begin executing. If there is a path from x_i to x_j in the DFG then x_i is a predecessor of x_j; we use $\Gamma(x_i, x_j)$ to denote the length of the longest path from x_i to x_j. The set of all predecessors of x is denoted by $\mathsf{Pred}(x)$ and $G_{\mathsf{Pred}}(x)$ denotes the sub-DFG induced by $\mathsf{Pred}(x)$; the set of all successors of x is denoted by $\mathsf{Succ}(x)$ and $G_{\mathsf{Succ}}(x)$ denotes the sub-DFG induced by $\mathsf{Succ}(x)$. The length of the longest path between any pairs of nodes in a graph G is denoted by $\mathsf{CP}(G)$.

Let $\mathcal{M}$ denote the set of resource classes (multipliers, adders, etc.). The number of available instances of each resource class ℓ, denoted M_ℓ, is fixed. Each operation in $x \in V$ is associated with a unique member $\mathsf{type}(x) \in \mathcal{M}$; operation x can be executed only on an instance of resource class $\mathsf{type}(x)$; further, each operation x executes in $\mathsf{delay}(x)$ time steps.

A schedule is a mapping of operations to time steps that satisfies the dependency and resource constraints. Suppose that there is an upper bound of λ time steps on the latency of the algorithm. Then, each operation x must be scheduled to begin executing on a unit of a resource of $\mathsf{type}(x)$ at some time step between 1 and $\lambda - \mathsf{delay}(x) + 1$. The value $\mathsf{ASAP}(x)$ (resp. $\mathsf{ALAP}(x)$) is the earliest (resp. latest) time step to which operation x can be scheduled given a latency λ. We will denote by $\mathsf{TimeFrame}(x)$ the set $\{\mathsf{ASAP}(x), \mathsf{ASAP}(x)+1, \ldots, \mathsf{ALAP}(x)\}$. We assume, without loss of generality, that the operations in $V(G)$ are numbered in decreasing order of their ASAP values. Notice that this implies that if $x_i \in \mathsf{Pred}(x_j)$ then $i < j$.

2.1 Previous Work

Most algorithms which yield exact (optimal) solutions model the scheduling problem using Integer Linear Programming. The ILP model is then solved using a

```
procedure Enumerate(S, i)
if (i = N + 1)                                          /* Scheduled all the elements ? */
   if (Feasible(S))
      if (best_latency > latency(S))
         best_schedule ← ⟨X_1, ..., X_N⟩;
         best_latency ← Latency(⟨X_1, ..., X_N⟩)
      endif
   endif
else
   for step ← ASAP(x_i) to ALAP(x_i) do
      Save ASAP Values
      if (ResourceUsed(step, type(x_i)) < M_type(x_i))
         S(i) ← step;                                   /* Schedule this node ... */
         Increment ResourceUsed(step, type(x_i));
         UpdateASAP(i, step);
         Enumerate(S, i + 1);                           /* and then schedule the rest */
         Decrement ResourceUsed(step, type(x_i));
      endif
      Restore ASAP Values
   enddo
endif
end procedure                                           /* Enumerate */

procedure UpdateASAP(i, step)
for j ← i + 1 to N do
   if (x_j is a successor of x_i)
      ASAP(x_j) ← max(ASAP(x_j), step + Γ(x_i, x_j));
   endif
enddo
end procedure                                           /* UpdateASAP */
```

Fig. 1. Naïve Complete Enumeration

generic ILP solver to get the optimal schedule. See Lin [1997] for a recent survey on techniques for high-level synthesis. Many different ILP models have been proposed [Camposano and Wolf 1991; Hwang et al. 1991]. Chaudhuri and Walker [1994] and Gebotys and Elmasry [1993] have shown that efficient formulations lead to reduced execution times for the ILP approach. However, as the number of variables in the formulations increases very rapidly with the size of the DFG, even with efficient formulations, tight resource constraints can cause ILP models to take inordinately long to solve.

3. EFFICIENT BRANCH-AND-BOUND SOLUTION

A naïve way of finding an optimal schedule is to enumerate every possible schedule, and then pick the optimal schedule. Such an algorithm is illustrated in Fig. 1. However, such an algorithm is too inefficient to be of any practical use, since the number of schedules is (in the worst case) exponential in the number of nodes. In order to improve the performance of the algorithm, we must reduce the number of valid schedules that are actually examined. By doing this, we are in effect, pruning the search space. Next, we show how that can be done using upper and lower bounding routines.

```
procedure Enumerate(⟨X_1, ..., X_i⟩)
if (i = N + 1)                                        /* Scheduled all the elements ? */
   if (best_latency > Latency(⟨X_1, ..., X_N⟩))
      best_schedule ← ⟨X_1, ..., X_N⟩;
      best_latency ← Latency(⟨X_1, ..., X_N⟩)
   endif
else
   for step ← ASAP(x_i) to ALAP(x_i) do
      Save ASAP Values
      if (ResourceUsed(step, type(x_i)) < M_type(x_i))
         l ← LowerBound(⟨X_1, ..., X_i⟩);
         u ← UpperBound(⟨X_1, ..., X_i⟩);
         if (u < best_latency)                        /* Found a better schedule */
            best_latency ← u;
            best_schedule ← UpperBoundSchedule(⟨X_1, ..., X_i⟩);
         endif
         if (l < best_latency)        /* Can this sub-tree contain a better schedule ? */
            X_{i+1} ← step;                           /* Schedule this node ... */
            Increment ResourceUsed(step, type(x_i));
            UpdateASAP(i, step);
            Enumerate(⟨X_1, ..., X_i, X_{i+1}⟩);      /* and then schedule the rest */
            Decrement ResourceUsed(step, type(x_i));
         endif
      endif
      Restore ASAP Values
   enddo
endif
end procedure                                         /* Enumerate */

procedure UpdateASAP(i, step)
for j ← i + 1 to N do
   if (x_j is a successor of x_i)
      ASAP(x_j) ← max(ASAP(x_j), step + Γ(x_i, x_j));
   endif
enddo
end procedure                                         /* UpdateASAP */
```

Fig. 2. Refined Complete Enumeration

3.1 Improving algorithm efficiency by more effective pruning

Consider a k-tuple $\langle X_1, X_2, \ldots, X_k \rangle$, $1 \le k \le |T|$. We call this k-tuple a partial schedule if all the following three conditions hold:

(1) $X_i \in \mathsf{TimeFrame}(t_i)\ \forall i \in \{1, 2, \ldots, k\}$;

(2) $\langle t_i, t_j \rangle \in E \ \Rightarrow\ X_i < X_j \qquad \forall i, j \in \{1, 2, \ldots, k\}$; (and)

(3) $|\{t_i \in T \,|\, X_i = n \text{ and } \mathsf{Resource}(t_i) = \ell\}| \le M_\ell$
$$\forall n \in Z, \ell \in \{1, 2, \ldots, R\}, i \in \{1, 2, \ldots, k\}.$$

We denote by $\langle\rangle$ the partial schedule corresponding to $k = 0$. For a partial schedule, decisions have been made about scheduling the first k nodes (in the ordering specified). Let Ψ be the set of all partial schedules, and ϕ the set of (complete) schedules. Clearly $\phi \subset \Psi$. Also notice that the partial schedule $\langle\rangle$ is an element of Ψ for which no scheduling decisions have been made. For any element ψ of Ψ, $|\,\psi\,| = k \Leftrightarrow \psi$ is a k-tuple.

Construct a tree as follows. The leaves are the elements of ϕ. The interior nodes are formed of the elements of $\Psi - \phi$. An element of $\psi_1 = \langle X_1, \ldots, X_{k_1} \rangle \in \Psi - \phi$ is the parent of an element of $\psi_2 = \langle Y_1, \ldots, Y_{k_2} \rangle \in \Psi$ if

$$k_1 = \mid \psi_1 \mid = \mid \psi_2 \mid -1 = k_2 - 1 \qquad \text{and} \qquad X_i = Y_i \;\; \forall i \in \{1, 2, \ldots, k_1\}.$$

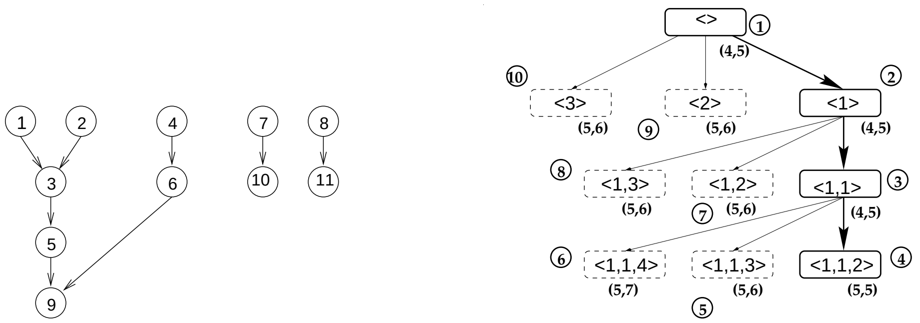
So the ancestor of an element in the tree represents a partial schedule for which fewer decisions have been made, and all the decisions that have been made for the ancestor are identical to the decisions that have been made for the descendant. Therefore the root of the tree is $\langle\rangle$, and the leaves correspond to valid schedules. What we require is the leaf of the tree corresponding to the least latency. It should be clear that a naive algorithm that enumerates every possible schedule is essentially a tree searching algorithm (or alternatively a depth-first traversal of the graph corresponding to this tree). Notice that this tree is very similar to the enumeration tree discussed in [Nemhauser and Wolsey 1988]. We will now show how the efficiency of this algorithm can be improved by pruning parts of the tree.

Suppose we had a lower (resp. upper) bound estimating function, which when given a DFG G, a set of resource constraints and a partial schedule ψ, will return a lower (resp. upper) bound on the optimal latency of any valid schedule (leaf of the tree) that is a descendant of ψ in the tree T corresponding to G. Let $\mathsf{Lower}(\psi)$ and $\mathsf{Upper}(\psi)$ represent the lower and upper bounding functions. We will also require that the upper bounding function return an upper bound that is achievable, i.e., there should actually be a schedule that is a descendant of ψ with latency $\mathsf{Upper}(\psi)$. Let $\mathsf{UpperBoundSchedule}(\psi)$ be one such schedule.

Suppose, for some partial schedule ψ, $\mathsf{Lower}(\psi) = \mathsf{Upper}(\psi)$. Then, the schedule returned by $\mathsf{UpperBoundSchedule}(\psi)$ is a schedule with the least latency in the subtree rooted at ψ. Therefore, we can obtain the optimal schedule in this subtree without having to exhaustively explore it. Thus, computing lower and upper bounds for every interior node of the tree results in a need to explore a sub-tree if and only if the lower and upper bounds differ. We use this concept to derive the refined algorithm shown in Fig. 2.

We will now illustrate this algorithm with an example. Consider the DFG shown in Fig. 3(a). We will assume that the operations shown (1 thru 11) are to be executed on two classes of resources. Operations 1, 2, 3, 4, 6 and 8 are to be executed on resource class A, and the remaining operation are to be executed on resource class B. We will assume that there are 2 instances of resource class A and 1 instance of resource class B. Further, for simplicity, we will assume that the delay of all operations is one time step.

A portion of the enumeration tree is shown in Fig. 3(b). For the purpose of our example, we will use very simplistic bounds estimators. The values in parenthesis below each node (partial schedule) represent the values returned by our bounds estimators for that partial schedule. So, the optimal schedule has latency at least 4 and at most 5, and all schedules in which the first operation is scheduled to time step 2 (represented by the subtree rooted at the node (partial schedule) $\langle 2 \rangle$) must have latency at least 5, and there is at least one such schedule with latency ≤ 6. The encircled numbers besides the node indicate the order of exploration of the tree. Therefore, node $\langle\rangle$ is the first node to be explored, followed by node $\langle 1 \rangle$, $\langle 1, 1 \rangle$ and $\langle 1, 1, 2 \rangle$.



(a) Example DFG

(b) Search Tree for DFG in (a)

Fig. 3. An example DFG and its search tree

Initially the algorithm finds the upper and lower bounds of all schedules. Since the values of the upper (5) and the lower (4) bound do not match, we must further explore by subdividing the search space into sub-spaces (or subtrees) and searching each of those spaces (trees).

When the algorithm reaches node (partial schedule) $\langle 1, 1, 2 \rangle$, the values of the upper and lower bound are equal and so it does not have to further enumerate the subtree rooted at this node.

After this when it attempts to bound the latencies of the optimal schedule in the subtrees rooted at nodes $\langle 1, 1, 3 \rangle$ and $\langle 1, 1, 4 \rangle$, since the lower bounds for both these trees are not less than the latency of the best schedule found so far (5), these subtrees cannot possibly contain a schedule with latency less than the optimal schedule found so far. Therefore, these subtrees can be pruned away, and do not have to be explored.

The nodes marked by dashed lines represent those nodes whose lower and upper bound information (coupled with the latency of the best schedule so far) allow us to determine that the subtree rooted at this node cannot possibly contain a solution better than the one we have already found. Notice that the node $\langle 1, 1, 2 \rangle$ comes under this category because it cannot possibly contain a schedule with latency less than 5, and a schedule with latency equal to 5 (returned by the function $\mathsf{UpperBoundSchedule}(\langle 1, 1, 2 \rangle)$) exists. The nodes marked in solid lines, however, represent the nodes whose bounds do not provide enough information to determine if this subtree can possibly contain an optimal schedule, and therefore must be explored further.

4. REFINING THE ASAP AND ALAP VALUES

We had earlier defined the ASAP (resp. ALAP) value of an operation $x \in V(G)$ to be the earliest (resp. latest) possible time step to which the operation can be scheduled. What this means is that for any valid schedule $\mathcal{S}$, $\mathsf{ASAP}(x) \leq \mathcal{S}(x) \leq \mathsf{ALAP}(x)$. In practice, these values are generated by scheduling the DFG while ignoring all the resource constraints. So the actual values of ASAP used are really lower bounds on the actual "As Soon As Possible" values. So, for any $\mathcal{S} \in \phi$ and $x \in V(G)$, $\mathsf{ASAP}(x) \leq \mathcal{S}(x)$. However, if for some $x \in V(G)$, $\mathsf{ASAP}(x) < \mathcal{S}(x)$ $\forall \mathcal{S} \in \phi$, then the ASAP value is not as tight as it could possibly be.

An operation cannot begin executing before all of its predecessors (direct and indirect) have completed executing. Therefore, another lower bound on the execution step of an operation is the minimum time taken to completely execute $G_{\mathsf{Pred}}(x)$. So, if the algorithm in [Rim and Jain 1994] returns l as a lower bound on the latency of $G(x)$ then, for any $\mathcal{S} \in \phi$ and $x \in V(G)$, $l \leq \mathcal{S}(x)$. This idea is used to develop tighter bounds in [Langevin and Cerny 1996]. The algorithm in [Rim and Jain 1994] is called recursively to compute (possibly) *better* values of ASAP for all the nodes, and then the [Rim and Jain 1994] can be used with the *new and improved* ASAP as the release step. The same technique can also be used to generate tighter ALAP values of the nodes. To show this, we first state some facts.

THEOREM 1. *Let $\mathcal{S} \in \phi$ be a valid schedule for the DFG G. Let G^r be another DFG, with $V(G) = V(G^r)$ and $\langle x_i, x_j \rangle \in E^r(G) \Leftrightarrow \langle x_j, x_i \rangle \in E(G)$. If $\mathcal{S}^r(x_j) = \mathsf{latency}(\mathcal{S}) - \mathcal{S}(x_j) + 1$, then $\mathcal{S}^r$ is a valid schedule for G^r and* $\mathsf{latency}(\mathcal{S}) \geq \mathsf{latency}(\mathcal{S}^r)$.

COROLLARY 2. *If $\mathcal{S}$ is an optimal schedule for G, then $\mathcal{S}^r$ is an optimal schedule for G^r.*

A key point to notice is that the ASAP values of the operations of the DFG G become the ALAP values of the corresponding operations of the DFG G^r. This is because, the set of predecessors of an operation become the set of successors of the corresponding operation in the reversed DFG. So finding the ASAP values of G (resp. G^R) is the same as finding the ALAP values of G^R (resp. G).

COROLLARY 3. *If $\mathcal{S}(x_j) \geq \mathsf{ASAP}(x_j)$ $\forall x_j \in V(G)$ and $\mathcal{S}$ is a valid schedule, then $\mathcal{S}^r(x_j) \leq \mathsf{latency}(\mathcal{S}) - \mathsf{ASAP}(x_j) + 1$.*

Therefore, we can calculate tighter estimates on the ALAP time of a node by simply running the procedure described in [Langevin and Cerny 1996] on the reversed DFG. Additional benefits of running the algorithm on G^r include possible tighter lower bounds on latency of the optimal schedule.

5. LOWER AND UPPER BOUND ESTIMATION

5.1 Lower Bound Estimation

The lower-bound estimation function that we used is one proposed by Langevin and Cerny [1996], which in turn is based on [Rim and Jain 1994]. Besides just lower-bounding the latency of all schedules of the graph G, we also need to lower-bound the latency of just those schedules for which the first m operations (in order of increasing ALAP values) are assigned to specific steps; in other words, we require a lower bound on the latency of the set of schedules which are descendants of a partial schedule, i.e., a lower bound on

$$\min_{\substack{\mathcal{S} \in \phi \\ \mathcal{S}(x_i) = s_i, 1 \leq i \leq m}} \mathsf{latency}(\mathcal{S}).$$

This can be done by a slight modification of the algorithm in [Rim and Jain 1994]. The modified algorithm is shown in Fig. 4.

```
procedure Lower(⟨X_1, X_2, ... X_m⟩)
Resources ← ResourceUsed
latency_estimate ← max_{i∈{1,2,...,m}} {X_i + CP(G_Succ(x_i))};
for i ← m+1 to |V(G)| do
   k ← ASAP(x_i);
   while (Resources(k, type(x_i)) = M_type(x_i)) do
      k ← k+1;
   enddo
   Increment Resources(k, type(x_i));
   latency_estimate ← max(latency_estimate, k + CP(G_Succ(x_i)));
enddo
end procedure                                                /* Lower */
```

Fig. 4. Partial Schedule Lower Bound Estimation

```
procedure Upper(⟨X_1, X_2, ... X_m⟩)
Resources ← ResourceUsed
latency_estimate ← max_{i∈{1,2,...,m}} {X_i + CP(G_Succ(x_i))};
for i ← m+1 to |V(G)| do
   k ← ASAP(x_i);
   while (there is a member of T_pred(t) scheduled on of after step k) do
      k ← k+1;
   enddo
   while (Resources(k, type(x_i)) = M_type(x_i)) do
      k ← k+1;
   enddo
   Increment Resources(k, type(x_i));
   latency_estimate ← max(latency_estimate, k);
enddo
end procedure                                                /* Upper */
```

Fig. 5. Partial Schedule Upper Bound Estimation

5.2 Upper bound estimation

The latency of any valid schedule is a valid upper bound on the latency of the optimal schedule. Therefore, to obtain tight estimates on the upper bound of the latency, we just employ a good heuristic to compute a valid schedule, and return the latency of that schedule as the required upper bound. Notice that this fits in with our requirement that the upper bound returned must be feasible. There are a number of good heuristics available. In our implementation, we used a list scheduler that uses the ALAP values of the operations as the priority function. The algorithm is shown in Fig. 5.

We chose a list scheduler over other algorithms such as force-directed scheduling [Paulin and Knight 1989] because of its speed. This procedure will be called several times and so speed is important.

6. EXPERIMENTAL RESULTS

Tables I and II report the times taken to solve the ILP formulations described in [Chaudhuri and Walker 1994] (referred to as ILP in the tables) and the times taken by our approach (referred to as BULB[2] in the tables) on several benchmarks. For all benchmarks other than the FFT, we assume that an ALU takes 1 clock cycle to execute, a multiplier takes 2 clock cycles and the pipeline initiation interval for a pipelined multiplier is 1 clock cycle. The multiplier in the FFT example is 3-stage pipelined. All times represent the running time on a Sun UltraSPARC 1 with 64 MBytes memory. The execution time listed for the ILP approach is the time required to solve the model alone. The time required to generate the model is not factored in. Cases for which no solution is produced within one hour of CPU time are denoted by a †. All operations other than multiplication were mapped to the ALU. The *EWF* and *FDCT* benchmarks problems that we used were part of the NEAT distribution [Eindhoven Design Automation Group WWW Server 1994]. The ILP solver used on all benchmarks is *lp_solve* version 2.1 [Berkelaar 1997].

It can be seen that in almost every case, our algorithm outperformed the ILP based technique, even though the ILP model used was extremely efficient. For problems with relatively small number of variables, ILP does well. However, as the resource constraints are tightened, the number of variables, and hence the time taken to solve the formulation also increases. Our algorithm generates the optimal schedule very fast in most of the cases. It can be seen that for only two cases did it fail to produce a solution within an hour. It should also be noted that the use of better heuristics will improve the performance of this algorithm.

7. SUMMARY

In this paper, we presented a branch-and-bound algorithm that finds optimal solutions to the resource-constrained scheduling problem in high-level synthesis. Unlike other approaches to branch-and-bound, we use both upper and lower bounds that are computed quickly. These bounds are problem-specific and therefore are more effective in pruning the search space. The effectiveness of this approach relies on the quality of bounds. Given the number of good heuristics (that produce upper bounds) for scheduling [Camposano and Wolf 1991; De Micheli 1994] and recent results on lower bounds [Rim and Jain 1994; Langevin and Cerny 1996; Narasimhan 1998; Narasimhan and Ramanujam 1997], such an approach is very effective. We have demonstrated the effectiveness of this approach by comparing the running times of our algorithm with the time taken to solve this problem using the ILP formulation of Chaudhuri and Walker [1994]; our approach demonstrates improvement of several orders of magnitude. Work is in progress in extending these techniques to complete design space exploration.

ACKNOWLEDGMENTS

The work of J. Ramanujam is supported in part by an NSF Young Investigator Award CCR–9457768 and an NSF Research Initiation Award CCR–9210422. We

[2]BULB stands for Branch with Upper and Lower Bounds

Table I. Scheduling times for various benchmarks for ILP and our branch and bound technique **Note:** † denotes no solution after one hour of CPU time. $*_m$: non-pipelined multiplier. $*_p$: pipelined multiplier. a: ALU. ILP: Integer Linear Programming. BULB: our branch-and-bound technique using both lower and upper bounds.

Bench-mark	$*_m$	a	#steps	ILP (sec.)	BULB (sec.)	$*_p$	a	#steps	ILP (sec.)	BULB (sec.)
EWF	1	1	28	†	<0.01	1	1	28	†	<0.01
	1	2	21	3.65	0.01	1	2	19	0.42	0.04
	2	2	18	0.05	<0.01	1	3	18	0.03	<0.01
	3	3	17	0.06	<0.01	2	2	18	0.08	<0.01
						2	3	17	<0.01	<0.01
ARF	1	1	34	†	11.22	1	1	19	1403.75	<0.01
	2	1	18	†	0.29	2	1	16	26.62	0.01
	2	2	18	†	2.34	1	2	19	0.65	<0.01
	3	1	16	726.26	<0.01	2	2	13	0.67	<0.01
	4	1	16	3.06	<0.01	2	3	13	0.12	<0.01
	3	2	15	908.15	10.90	3	2	13	0.26	<0.01
	3	3	15	933.08	11.35	4	2	11	0.02	<0.01
	4	2	11	0.02	<0.01					
FDCT	1	1	34	†	0.01	1	1	26	†	<0.01
	2	1	26	†	<0.01	2	1	26	†	0.01
	2	2	–	†	†	1	2	19	†	<0.01
	2	3	18	†	<0.01	2	2	13	†	<0.01
	3	2	14	†	213.25	3	2	13	†	<0.01
	3	3	14	†	1.03	2	3	12	410.57	0.17
	3	4	–	†	†	3	3	10	4.55	<0.01
	4	2	13	†	<0.01	4	4	09	0.16	<0.01
	4	3	11	988.51	107.03	8	4	08	0.04	<0.01
	4	4	11	40.63	<0.01					
	4	5	11	21.60	<0.01					
	5	4	10	2.54	<0.01					
	5	5	10	1.60	<0.01					
	8	4	08	0.03	<0.01					
2 times unfolded EWF	1	1	56	†	<0.01	1	1	56	†	<0.01
	2	1	56	†	<0.01	2	1	56	†	<0.01
	1	2	41	3504.76	0.04	1	2	37	31.22	0.01
	2	2	35	10.36	0.02	2	2	36	14.48	0.01
	1	3	41	689.23	0.04	1	3	35	0.14	<0.01
	2	3	35	0.62	0.01	2	3	33	0.03	<0.01
	3	3	33	0.02	<0.01					
3 times unfolded EWF	1	1	84	†	<0.01	1	1	84	†	<0.01
	2	1	84	†	<0.01	2	1	84	†	<0.01
	1	2	61	2381.28	0.08	1	2	55	†	0.01
	2	2	52	462.15	0.04	2	2	53	398.35	0.01
	1	3	61	†	0.08	1	3	52	0.4	<0.01
	2	3	52	20.89	2460	2	3	49	0.06	<0.01
	3	3	49	0.04	<0.01					
32 point FFT	1	1	241	†	< 0.01	1	1	160	†	< 0.01
	1	2	241	†	< 0.01	1	2	83	†	< 0.01
	2	1	160	†	< 0.01	2	2	80	†	< 0.01
	2	2	121	†	< 0.01	8	8	21	†	< 0.01

thank the anonymous referees whose constructive comments have helped us improve the presentation of this paper.

References

BERKELAAR, M. 1997. *lp_solve* version 2.1. ftp://ftp.es.ele.tue.nl/pub/lp_solve.

CAMPOSANO, R. AND WOLF, W. 1991. *High-Level VLSI Synthesis.* Kluwer Academic.

CHAUDHURI, S. AND WALKER, R. A. 1994. Analyzing and exploiting the structure of the constraints in the ILP approach to the scheduling problem. *IEEE Transactions on Very Large Scale Integration 2*, 4 (Dec.), 456–471.

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

EINDHOVEN DESIGN AUTOMATION GROUP WWW SERVER. 1994. *Online Neat Sources: http://www.es.ele.tue.nl/neat* (pre-release ed.). Eindhoven University of Technology.

GAJSKI, D., DUTT, N., AND PANGRLE, B. 1986. Silicon compilation (tutorial). In *Proceedings of the IEEE Custom Integrated Circuits Conference* (May 1986). pp. 102–110.

GEBOTYS, C. H. AND ELMASRY, M. I. 1993. Global optimization approach for architectural synthesis. *IEEE Trans. Computer-Aided Design 12*, 9 (Sept.), 1266–1278.

HWANG, C. T., LEE, T. H., AND HSU, Y. C. 1991. A formal approach to the scheduling problem in high level synthesis. *IEEE Trans. Computer-Aided Design 10*, 4, 464–475.

LANGEVIN, M. AND CERNY, E. 1996. A recursive technique for computing lower-bound performance of schedules. *ACM Trans. Design Automation of Electronic Systems 1*, 4 (Oct.), 443–456.

LIN, Y.-L. 1997. Recent developments in high-level synthesis. *ACM Trans. Design Automation of Electronic Systems 2*, 1 (Jan.), 2–21.

MCFARLAND, M., PARKER, A., AND CAMPOSANO, R. 1990. The high-level synthesis of digital systems. *Proceedings of the IEEE 78*, 2 (Feb.), 301–318.

NARASIMHAN, M. 1998. Exact scheduling techniques for high-level synthesis. M.S. thesis, Louisiana State University.

NARASIMHAN, M. AND RAMANUJAM, J. 1997. The effect of tight partial schedule bounds and better heuristics on branch-and-bound solutions to scheduling. Technical Report TR 97-07-01 (July 97, revised October 97), Louisiana State University.

NEMHAUSER, G. L. AND WOLSEY, L. A. 1988. *Integer and Combinatorial Optimization.* Wiley–Interscience.

PAULIN, P. AND KNIGHT, J. 1989. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Trans. Computer-Aided Design 8*, 6 (June), 661–679.

RIM, M. AND JAIN, R. 1994. Lower bound performance estimation for the high-level synthesis scheduling problem. *IEEE Trans. Computer-Aided Design 13*, 4 (April), 451–458.

ULLMAN, J. 1975. NP-complete scheduling problems. *Journal of Computer and System Sciences 10*, 3 (June), 384–393.