Chapter 13
Operation Scheduling: Algorithms and Applications

Gang Wang, Wenrui Gong, and Ryan Kastner

Abstract Operation scheduling (OS) is an important task in the high-level synthesis process. An inappropriate scheduling of the operations can fail to exploit the full potential of the system. In this chapter, we try to give a comprehensive coverage on the heuristic algorithms currently available for solving both timing and resource constrained scheduling problems. Besides providing a broad survey on this topic, we focus on some of the most popularly used algorithms, such as List Scheduling, Force-Directed Scheduling and Simulated Annealing, as well as the newly introduced approach based on the Ant Colony Optimization meta-heuristics. We discuss in details on their applicability and performance by comparing them on solution quality, performance stability, scalability, extensibility, and computation cost. Moreover, as an application of operation scheduling, we introduce a novel uniformed design space exploration method that exploits the duality of the time and resource constrained scheduling problems, which automatically constructs a high quality time/area tradeoff curve in a fast, effective manner.

Keywords: Design space exploration, Ant colony optimization, Instruction scheduling, MAX–MIN ant system

13.1 Introduction

As fabrication technology advances and transistors become more plentiful, modern computing systems can achieve better system performance by increasing the amount of computation units. It is estimated that we will be able to integrate more than a half billion transistors on a 468 mm^2 chip by the year of 2009 [38]. This yields tremendous potential for future computing systems, however, it imposes big challenges on how to effectively use and design such complicated systems.

As computing systems become more complex, so do the applications that can run on them. Designers will increasingly rely on automated design tools in order

P. Coussy and A. Morawiec (eds.) *High-Level Synthesis.*

to map applications onto these systems. One fundamental process of these tools is mapping a behavioral application specification to the computing system. For example, the tool may take a C function and create the code to program a microprocessor. This is viewed as software compilation. Or the tool may take a transaction level behavior and create a register transfer level (RTL) circuit description. This is called hardware or behavioral synthesis [31]. Both software and hardware synthesis flows are essential for the use and design of future computing systems.

Operation scheduling (OS) is an important problem in software compilation and hardware synthesis. An inappropriate scheduling of the operations can fail to exploit the full potential of the system. Operation scheduling appears in a number of different problems, e.g., compiler design for superscalar and VLIW microprocessors [23], distributed clustering computation architectures [4] and behavioral synthesis of ASICs and FPGAs [31].

Operation scheduling is performed on a behavioral description of the application. This description is typically decomposed into several blocks (e.g., basic blocks), and each of the blocks is represented by a data flow graph (DFG).

Operation scheduling can be classified as *resource constrained* or *timing constrained*. Given a DFG, clock cycle time, resource count and resource delays, a resource constrained scheduling finds the minimum number of clock cycles needed to execute the DFG. On the other hand, a timing constrained scheduling tries to determine the minimum number of resources needed for a given deadline.

In the timing constrained scheduling problem (also called fixed control step scheduling), the target is to find the minimum computing resource cost under a set of given types of computing units and a predefined latency deadline. For example, in many digital signal processing (DSP) systems, the sampling rate of the input data stream dictates the maximum time allowed for computation on the present data sample before the next sample arrives. Since the sampling rate is fixed, the main objective is to minimize the cost of the hardware. Given the clock cycle time, the sampling rate can be expressed in terms of the number of cycles that are required to execute the algorithm.

Resource constrained scheduling is also found frequently in practice. This is because in a lot of the cases, the number of resources are known a priori. For instance, in software compilation for microprocessors, the computing resources are fixed. In hardware compilation, DFGs are often constructed and scheduled almost independently. Furthermore, if we want to maximize resource sharing, each block should use same or similar resources, which is hardly ensured by time constrained schedulers. The time constraint of each block is not easy to define since blocks are typically serialized and budgeting global performance constraint for each block is not trivial [30].

Operation scheduling methods can be further classified as *static scheduling* and *dynamic scheduling* [40]. Static operation scheduling is performed during the compilation of the application. Once an acceptable scheduling solution is found, it is deployed as part of the application image. In dynamic scheduling, a dedicated system component makes scheduling decisions on-the-fly. Dynamic scheduling

methods must minimize the program's completion time while considering the overhead paid for running the scheduler.

13.2 Operation Scheduling Formulation

Given a set of operations and a collection of computational units, the resource constrained scheduling (RCS) problem schedules the operations onto the computing units such that the execution time of these operations are minimized, while respecting the capacity limits imposed by the number of computational resources. The operations can be modeled as a data flow graph (DFG) $G(V,E)$, where each node $v_i \in V (i = 1,\ldots,n)$ represents an operation op_i, and the edge e_{ij} denotes a dependency between operations v_j and v_i. A DFG is a directed acyclic graph where the dependencies define a partially ordered relationship (denoted by the symbol $\preccurlyeq$) among the nodes. Without affecting the problem, we add two virtual nodes *root* and *end*, which are associated with no operation (NOP). We assume that *root* is the only starting node in the DFG, i.e., it has no predecessors, and node *end* is the only exit node, i.e., it has no successors.

Additionally, we have a collection of computing resources, e.g., ALUs, adders, and multipliers. There are R different types and $r_j > 0$ gives the number of units for resource type j $(1 \leqslant j \leqslant R)$. Furthermore, each operation defined in the DFG must be executable on at least one type of the resources. When each of the operations is uniquely associated with one resource type, we call it *homogenous* scheduling. If an operation can be performed by more than one resource types, we call it *heterogeneous* scheduling [44]. Moreover, we assume the cycle delays for each operation on different type resources are known as $d(i, j)$. Of course, *root* and *end* have zero delays. Finally, we assume the execution of the operations is non-preemptive, that is, once an operation starts execution, it must finish without being interrupted.

A resource constrained schedule is given by the vector

$$\{(s_{root}, f_{root}), (s_1, f_1), \ldots, (s_{end}, f_{end})\}$$

where s_i and f_i indicate the starting and finishing time of the operation op_i. The resource-constrained scheduling problem is formally defined as $min(s_{end})$ with respect to the following conditions:

1. An operation can only start when all its predecessors have finished, i.e., $s_i \geqslant f_j$ if $op_j \preccurlyeq op_i$
2. At any given cycle t, the number of resources needed is constrained by r_j, for all $1 \leqslant j \leqslant R$

The timing constrained scheduling (TCS) is a dual problem of the resource constrained version and can be defined using the same terminology presented above. Here the target is to minimize total resources $\sum_j r_j$ or the total cost of the resources (e.g., the hardware area needed) subject to the same dependencies between operations imposed by the DFG and a given deadline D, i.e., $s_{end} < D$.

13.3 Operation Scheduling Algorithms

13.3.1 ASAP, ALAP and Bounding Properties

The simplest scheduling task occurs when we have unlimited computing resources for the given application while trying to minimize its latency. For this task, we can simply solve it by schedule an operation as soon as all of its predecessors in the DFG have completed, which gives it the name *As Soon As Possible*. Because of its ASAP, nature, it is closely related with finding the longest path between an operation and the starting of the application op_{root}. Furthermore, it can be viewed as a special case of resource constrained scheduling where there is no limit on the number computing unit. The result of ASAP provides the lower bound for the starting time of each operation, together with the lower bound of the overall application latency.

Correspondingly, with a given latency, we have the so called *As Late As Possible* (ALAP) scheduling, where each operation is scheduled to the latest opportunity. This can be done by computing the longest path between the operation node and the end of the application op_{end}. The result scheduling provides a upper bound for the starting time of each operation given the latency constraint on the application. However, different from ASAP, it typically does not have any significance regarding to how efficient the resources are used. On the contrary, it often yields a bad solution in the sense of timing constrained scheduling since the operations tends to cluster towards the end.

Though not directly useful in typical practice, ASAP and ALAP are often critical components for more advanced scheduling methods. This is because their combined results provide the possible scheduling choices for each operation. Such range is often referred as the *mobility* of an operation.

Finally, the upper bound of the application latency (under a given technology mapping) can be obtained by serializing the DFG, that is to perform the operations sequentially based on a topologically sorted sequence of the operations. This is equivalent to have only one unit for each type of operation.

13.3.2 Exact Methods

Though scheduling problems are $\mathcal{NP}$-hard [8], both time and resource constrained problems can be formulated using integer linear programming (ILP) method [27], which tries to find an optimal schedule using a branch-and-bound search algorithm. It also involves some amount of backtracking, i.e., decisions made earlier are changed later on. A simplified formulation of the ILP method for the time constrained problem is given below:

First it calculates the mobility range for each operation $M = \{S_j | E_k \leqslant j \leqslant L_k\}$, where E_k and L_k are the ASAP and ALAP values respectively. The scheduling problem in ILP is defined by the following equations:

$$Min(\sum_{k=1}^{n}(C_k \cdot N_k)) \text{ while } \sum_{E_i \leqslant j \leqslant L_i} x_{ij} = 1$$

where $1 \leqslant i \leqslant n$ and n is the number of operations. There are $1 \leqslant k \leqslant m$ operation types available, and N_k is the number of computing units for operation type k, and C_k is the cost of each unit. Each x_{ij} is 1 if the operation i is assigned in control step j and 0 otherwise. Two more equations that enforce the resource and data dependency constraints are: $\sum_{i=1}^{n} x_{ij} \leqslant N_i$ and $((q * x_{j,q}) - (p * x_{i,p})) \leqslant -1, p \leqslant q$, where p and q are the control steps assigned to the operations x_i and x_j respectively.

We can see that the ILP formulation increases rapidly with the number of control steps. For one unit increase in the number of control steps we will have n additional x variables. Therefore the time of execution of the algorithm also increases rapidly. In practice the ILP approach is applicable only to very small problems.

Another exact method is Hu's algorithm [22], which provides an optimal solution for a limited set of applications. Though can be modified to address generic acyclic DFG scheduling problem, the optimality only applies when the DFG is composed of a set of trees and each unit has single delay with uniformed computing units. Essentially, Hu's method is a special list scheduling algorithm with a priority based on longest paths [31].

13.3.3 Force Directed Scheduling

Because of the limitations of the exact approaches, a range of heuristic methods with polynomial runtime complexity have been proposed. Many timing constrained scheduling algorithms used in high level synthesis are derivatives of the force-directed scheduling (FDS) algorithm presented by Paulin and Knight [34, 35]. Verhaegh et al. [45, 46] provide a theoretical treatment on the original FDS algorithm and report better results by applying gradual time-frame reduction and the use of global spring constants in the force calculation.

The goal of the FDS algorithm is to reduce the number of functional units used in the implementation of the design. This objective is achieved by attempting to uniformly distribute the operations onto the available resource units. The distribution ensures that resource units allocated to perform operations in one control step are used efficiently in all other control steps, which leads to a high utilization rate.

The FDS algorithm relies on both the ASAP and the ALAP scheduling algorithms to determine the feasible control steps for every operation op_i, or the *time frame* of op_i (denoted as $[t_i^S, t_i^L]$ where t_i^S and t_i^L are the ASAP and ALAP times respectively). It also assumes that each operation op_i has a uniform probability of being scheduled into any of the control steps in the range, and zero probability of being scheduled elsewhere. Thus, for a given time step j and an operation op_i which needs $\triangle_i \geqslant 1$ time steps to execute, this probability is given as:

$$\mathsf{p}_j(op_i) = \begin{cases} (\sum_{l=0}^{\triangle_i} h_i(j-l))/(t_i^L - t_i^S + 1) & \text{if } t_i^S \leqslant j \leqslant t_i^L \\ 0 & \text{otherwise} \end{cases} \tag{13.1}$$

where $h_i(\cdot)$ is a unit window function defined on $[t_i^S, t_i^L]$.

Based on this probability, a set of *distribution graphs* can be created, one for each specific type of operation, denoted as q_k. More specifically, for type k at time step j,

$$q_k(j) = \sum_{op_i} \mathsf{p}_j(op_i) \qquad \text{if type of } op_i \text{ is } k \tag{13.2}$$

We can see that $q_k(j)$ is an estimation on the number of type k resources that are needed at control step j.

The FDS algorithm tries to minimize the overall concurrency under a fixed latency by scheduling operations one by one. At every time step, the effect of scheduling each unscheduled operation on every possible time step in its frame range is calculated, and the operation and the corresponding time step with the smallest negative effect is selected. This effect is equated as the force for an unscheduled operation op_i at control step j, and is comprised of two components: the self-force, SF_{ij}, and the predecessor–successor forces, PSF_{ij}.

The self-force SF_{ij} represents the direct effect of this scheduling on the overall concurrency. It is given by:

$$SF_{ij} = \sum_{l=t_i^S}^{t_i^L+\triangle_i} q_k(l)(H_i(l) - \mathsf{p}_i(l)) \tag{13.3}$$

where, $j \in [t_i^S, t_i^L]$, k is the type of operation op_i, and $H_i(\cdot)$ is the unit window function defined on $[j, j+\triangle_i]$.

We also need to consider the predecessor and successor forces since assigning operation op_i to time step j might cause the time frame of a predecessor or successor operation op_l to change from $[t_l^S, t_l^L]$ to $[\widetilde{t}_l^S, \widetilde{t}_l^S]$. The force exerted by a predecessor or successor is given by:

$$PSF_{ij}(l) = \sum_{m=\widetilde{t}_i^S}^{\widetilde{t}_i^L+\triangle_l} (q_k(m) \cdot \widetilde{\mathsf{p}}_m(op_l)) - \sum_{m=t_i^S}^{t_i^L+\triangle_l} (q_k(m) \cdot \mathsf{p}_m(op_l)) \tag{13.4}$$

where $\widetilde{\mathsf{p}}_m(op_l)$ is computed in the same way as (13.1) except the updated mobility information $[\widetilde{t}_l^S, \widetilde{t}_l^S]$ is used. Notice that the above computation has to be carried for all the predecessor and successor operations of op_i. The total force of the hypothetical assignment of scheduling op_i on time step j is the addition of the self-force and all the predecessor–successor forces, i.e.,

$$\text{total force}_{ij} = SF_{ij} + \sum_l PSF_{ij}(l) \tag{13.5}$$

where op_l is a predecessor or successor of op_i. Finally, the total forces obtained for all the unscheduled operations at every possible time step are compared. The operation and time step with the best force reduction is chosen and the partial scheduling result is incremented until all the operations have been scheduled.

The FDS method is "constructive" because the solution is computed without performing any backtracking. Every decision is made in a greedy manner. If there are two possible assignments sharing the same cost, the above algorithm cannot accurately estimate the best choice. Based on our experience, this happens fairly often as the DFG becomes larger and more complex. Moreover, FDS does not take into account future assignments of operators to the same control step. Consequently, it is likely that the resulting solution will not be optimal, due to the lack of a look ahead scheme and the lack of compromises between early and late decisions.

Our experiments show that a baseline FDS implementation based on [34] fails to find the optimal solution even on small testing cases. To ease this problem, a look-ahead factor was introduced in the same paper. A second order term of the displacement weighted by a constant η is included in force computation, and the value η is experimentally decided to be $1/3$. In our experiments, this look-ahead factor has a positive impact on some testing cases but does not always work well. More details regarding FDS performance can be found in Sect. 13.4.

13.3.4 List Scheduling

List scheduling is a commonly used heuristic for solving a variety of RCS problems [36, 37]. It is a generalization of the ASAP algorithm with the inclusion of resource constraints [25]. A list scheduler takes a data flow graph and a priority list of all the nodes in the DFG as input. The list is sorted with decreasing magnitude of priority assigned to each of the operation. The list scheduler maintains a ready list, i.e., nodes whose predecessors have already been scheduled. In each iteration, the scheduler scans the priority list and operations with higher priority are scheduled first. Scheduling an operator to a control step makes its successor operations ready, which will be added to the ready list. This process is carried until all of the operations have been scheduled. When there exist more than one ready nodes sharing the same priority, ties are broken randomly.

It is easy to see that list scheduler always generates feasible schedule. Furthermore, it has been shown that a list scheduler is always capable of producing the optimal schedule for resource-constrained instruction scheduling problem if we enumerate the topological permutations of the DFG nodes with the input priority list [25].

The success of the list scheduler is highly dependent on the priority function and the structure of the input application (DFG) [25,31,43]. One commonly used priority function assigns the priority inversely proportional to the mobility. This ensures that the scheduling of operations with large mobilities are deferred because they have more flexibility as to where they can be scheduled. Many other priority functions

have been proposed [2, 5, 18, 25]. However, it is commonly agreed that there is no single good heuristic for prioritizing the DFG nodes across a range of applications using list scheduling. Our results in Sect. 13.4 confirm this.

13.3.5 Iterative Heuristic Methods

Both FDS and List Scheduling are greedy constructive methods. Due to the lack of a look ahead scheme, they are likely to produce a sub-optimal solution. One way to address this issue is the iterative method proposed by Park and Kyung [33] based on Kernighan and Lin's heuristic [24] method used for solving the graph-bisection problem. In their approach, each operation is scheduled into an earlier or later step using the move that produces the maximum gain. Then all the operations are unlocked and the whole procedure is repeated with this new schedule. The quality of the result produced by this algorithm is highly dependent upon the initial solution. There have been two enhancements made to this algorithm: (1) Since the algorithm is computationally efficient it can be run many times with different initial solution and the best solution can be picked. (2) A better look-ahead scheme that uses a more sophisticated strategy of move selection as in [kris84] can be used. More recently, Heijligers et al. [20] and InSyn [39] use evolutionary techniques like genetic algorithms and simulated evolution.

There are a number of iterative algorithms for the resource constrained problem, including genetic algorithm [7, 18], tabu search [6, 44], simulated annealing [43], graph theoretic and computational geometry approaches [4, 10, 30].

13.3.6 Ant Colony Optimization (ACO)

ACO is a cooperative heuristic searching algorithm inspired by ethological studies on the behavior of ants [15]. It was observed [13] that ants – who lack sophisticated vision – manage to establish the optimal path between their colony and a food source within a very short period of time. This is done through indirect communication known as *stigmergy* via the chemical substance, or *pheromone*, left by the ants on the paths. Each individual ant makes a decision on its direction biased on the "strength" of the pheromone trails that lie before it, where a higher amount of pheromone hints a better path. As an ant traverses a path, it reinforces that path with its own pheromone. A collective autocatalytic behavior emerges as more ants will choose the shortest trails, which in turn creates an even larger amount of pheromone on the short trails, making such short trails more attractive to the future ants. The ACO algorithm is inspired by this observation. It is a population based approach where a collection of agents cooperate together to explore the search space. They communicate via a mechanism imitating the pheromone trails.

One of the first problems to which ACO was successfully applied was the Traveling Salesman Problem (TSP) [15], for which it gave competitive results comparing with traditional methods. Researchers have since formulated ACO methods for a variety of traditional $\mathcal{NP}$-hard problems. These problems include the maximum clique problem, the quadratic assignment problem, the graph coloring problem, the shortest common super-sequence problem, and the multiple knapsack problem. ACO also has been applied to practical problems such as the vehicle routing problem, data mining, network routing problem and the system level task partitioning problem [12, 48, 49].

It was shown [19] that ACO converges to an optimal solution with probability of exactly one; however there is no constructive way to guarantee this. Balancing exploration to achieve close-to-optimal results within manageable time remains an active research topic for ACO algorithms. MAX–MIN Ant System (MMAS) [42] is a popularly used method to address this problem. MMAS is built upon the original ACO algorithm, which improves it by providing dynamically evolving bounds on the pheromone trails so that the heuristic never strays too far away from the best encountered solution. As a result, all possible paths will have a non-trivial probability of being selected; thus it encourages broader exploration of the search space while maintaining a good differential between alternative solutions. It was reported that MMAS was the best performing ACO approach on a number of classic combinatory optimization tasks.

Both time constrained and resource constrained scheduling problems can be effectively solved by using ACO. Unfortunately, in the consideration of space, we can only give a general introduction on the ACO formulation for the TCS problem. For a complete treatment of the algorithms, including detailed discussion on the algorithms' implementation, applicability, complexity, extensibility, parameter selection and performance, please refer to [47, 50].

In its ACO-based formulation, the TCS problem is solved with an iterative searching process. the algorithms employ a collection of agents that collaboratively explore the search space. A stochastic decision making strategy is applied in order to combine global and local heuristics to effectively conduct this exploration. As the algorithm proceeds in finding better quality solutions, dynamically computed local heuristics are utilized to better guide the searching process. Each iteration consists of two stages. First, the ACO algorithm is applied where a collection of ants traverse the DFG to construct individual operation schedules with respect to the specified deadline using global and local heuristics. Secondly, these scheduling results are evaluated using their resource costs. The associated heuristics are then adjusted based on the solutions found in the current iteration. The hope is that future iterations will benefit from this adjustment and come up with better schedules.

Each operation or DFG node op_i is associated with D pheromone trails τ_{ij}, where $j = 1, \ldots, D$ and D is the specified deadline. These pheromone trails indicate the global favorableness of assigning the ith operation at the jth control step in order to minimize the resource cost with respect to the time constraint. Initially, based on ASAP and ALAP results, τ_{ij} is set with some fixed value τ_0 if j is a valid control step for op_i; otherwise, it is set to be 0.

For each iteration, m ants are released and each ant individually starts to construct a schedule by picking an unscheduled instruction and determining its desired control step. However, unlike the deterministic approach used in the FDS method, each ant picks up the next instruction for scheduling decision probabilistically. Once an instruction op_h is selected, the ant needs to make decision on which control step it should be assigned. This decision is also made probabilistically as illustrated in (13.6).

$$\mathsf{p}_{hj} = \begin{cases} \frac{\tau_{hj}(t)^{\alpha} \cdot \eta_{hj}^{\beta}}{\sum_l (\tau_{hl}^{\alpha}(t) \cdot \eta_{hl}^{\beta})} & \text{if } op_h \text{ can be scheduled at } l \text{ and } j \\ 0 & \text{otherwise} \end{cases} \tag{13.6}$$

Here j is the time step under consideration. The item η_{hj} is the local heuristic for scheduling operation op_h at control step j, and α and β are parameters to control the relative influence of the distributed global heuristic τ_{hj} and local heuristic η_{hj}. Assuming op_h is of type k, η_{hj} to simply set to be the inverse of the distribution graph value [34], which is computed based on partial scheduling result and is an indication on the number of computing units of type k needed at control step j. In other words, an ant is more likely to make a decision that is globally considered "good" and also uses the fewest number of resources under the current partially scheduled result. We do not recursively compute the forces on the successor nodes and predecessor nodes. Thus, selection is much faster. Furthermore, the time frames are updated to reflect the changed partial schedule. This guarantees that each ant will always construct a valid schedule.

In the second stage of our algorithm, the ant's solutions are evaluated. The quality of the solution from ant h is judged by the total number of resources, i.e., $Q_h = \sum_k r_k$. At the end of the iteration, the pheromone trail is updated according to the quality of individual schedules. Additionally, a certain amount of pheromone evaporates. More specifically, we have:

$$\tau_{ij}(t) = \rho \cdot \tau_{ij}(t) + \sum_{h=1}^{m} \Delta \tau_{ij}^{h}(t) \qquad \text{where } 0 < \rho < 1. \tag{13.7}$$

Here ρ is the evaporation ratio, and

$$\Delta \tau_{ij}^{h} = \begin{cases} Q/Q_h & \text{if } op_i \text{ is scheduled at } j \text{ by ant } h \\ 0 & \text{otherwise} \end{cases} \tag{13.8}$$

Q is a fixed constant to control the delivery rate of the pheromone. Two important operations are performed in the pheromone trail updating process. Evaporation is necessary for ACO to effectively explore the solution space, while reinforcement ensures that the favorable operation orderings receive a higher volume of pheromone and will have a better chance of being selected in the future iterations. The above process is repeated multiple times until an ending condition is reached. The best result found by the algorithm is reported.

Comparing with the FDS method, the ACO algorithm differs in several aspects. First, rather than using a one-time constructive approach based on greedy local decisions, the ACO method solves the problem in an evolutionary manner. By using simple local heuristics, it allows individual scheduling result to be generated in a faster manner. With a collection of such individual results and by embedding and adjusting global heuristics associated with partial solutions, it tries to learn during the searching process. By adopting a stochastic decision making strategy considering both global experience and local heuristics, it tries to balance the efforts of exploration and exploitation in this process. Furthermore, it applies positive feedback to strengthen the "good" partial solutions in order to speed up the convergence. Of course, the negative effect is that it may fall into local minima, thus requires compensation measures such as the one introduced in MMAS. In our experiments, we implemented both the basic ACO and the MMAS algorithms. The latter consistently achieves better scheduling results, especially for larger DFGs.

13.4 Performance Evaluation

13.4.1 Benchmarks and Setup

In order to test and evaluate our algorithms, we have constructed a comprehensive set of benchmarks named *ExpressDFG*. These benchmarks are taken from one of two sources: (1) popular benchmarks used in previous literature; (2) real-life examples generated and selected from the MediaBench suite [26].

The benefit of having classic samples is that they provide a direct comparison between results generated by our algorithm and results from previously published methods. This is especially helpful when some of the benchmarks have known optimal solutions. In our final testing benchmark set, seven samples widely used in instruction scheduling studies are included. These samples focus mainly on frequently used numeric calculations performed by different applications. However, these samples are typically small to medium in size, and are considered somewhat old. To be representative, it is necessary to create a more comprehensive set with benchmarks of different sizes and complexities. Such benchmarks shall aim to:

- Provide real-life testing cases from real-life applications
- Provide more up-to-date testing cases from modern applications
- Provide challenging samples for instruction scheduling algorithms with regards to larger number of operations, higher level of parallelism and data dependency
- Provide a wide range of synthesis problems to test the algorithms' scalability

For this purpose, we investigated the MediaBench suite, which contains a wide range of complete applications for image processing, communications and DSP applications. We analyzed these applications using the SUIF [3] and Machine SUIF [41] tools, and over 14,000 DFGs were extracted as preliminary candidates for our

Table 13.1 ExpressDFG benchmark suite

Benchmark name	No. nodes	No. edges	ID
HAL	11	8	4
horner_bezier†	18	16	8
ARF	28	30	8
motion_vectors†	32	29	6
EWF	34	47	14
FIR2	40	39	11
FIR1	44	43	11
h2v2_smooth_downsample†	51	52	16
feedback_points†	53	50	7
collapse_pyr†	56	73	7
COSINE1	66	76	8
COSINE2	82	91	8
write_bmp_header†	106	88	7
interpolate_aux†	108	104	8
matmul†	109	116	9
idctcol	114	164	16
jpeg_idct_ifast†	122	162	14
jpeg_fdct_islow†	134	169	13
smooth_color_z_triangle†	197	196	11
invert_matrix_general†	333	354	11

Benchmarks with † are extracted from MediaBench

benchmark set. After careful study, thirteen DFG samples were selected from four MediaBench applications: JPEG, MPEG2, EPIC and MESA.

Table 13.1 lists all 20 benchmarks that were included in our final benchmark set. Together with the names of the various functions where the basic blocks originated are the number of nodes, number of edges and instruction depth (assuming unit delay for every instruction) of the DFG. The data, including related statistics, DFG graphs and source code for the all testing benchmarks, is available online [17].

For all testing benchmarks, operations are allocated on two types of computing resources, namely MUL and ALU, where MUL is capable of handling multiplication and division, and ALU is used for other operations such as addition and subtraction. Furthermore, we define the operations running on MUL to take two clock cycles and the ALU operations take one. This definitely is a simplified case from reality. However, it is a close enough approximation and does not change the generality of the results. Other choices can easily be implemented within our framework.

13.4.2 Time Constrained Scheduling: ACO vs. FDS

With the assigned resource/operation mapping, ASAP is first performed to find the critical path delay L_c. We then set our predefined deadline range to be $[L_c, 2L_c]$, i.e.,

from the critical path delay to two times of this delay. This results 263 testing cases in total. For each delay, we run FDS first to obtain its scheduling result. Following this, the ACO algorithm is executed five times to obtain enough data for performance evaluation. We report the FDS result quality, the average and best result quality for the ACO algorithm and the standard deviation for these results. The execution time information for both algorithms is also reported.

We have implemented the ACO formulation in C for the TCS problem. The evaporation rate ρ is configured to be 0.98. The scaling parameters for global and local heuristics are set to be $\alpha = \beta = 1$ and delivery rate $Q = 1$. These parameters are not changed over the tests. We also experimented with different ant number m and the allowed iteration count N. For example, set m to be proportional to the average branching factor of the DFG under study and N to be proportional to the total operation number. However, it is found that there seems to exist a fixed value pair for m and N which works well across the wide range of testing samples in our benchmark. In our final settings, we set m to be 10, and N to be 150 for all the timing constrained scheduling experiments.

Based on our experiments, the ACO based operation scheduling achieves better or much better results. Our approach seems to have much stronger capability in robustly finding better results for different testing cases. Furthermore, it scales very well over different DFG sizes and complexities. Another aspect of scalability is the pre-defined deadline. The average result quality generated by the ACO algorithm is better than or equal to the FDS results in 258 out of 263 cases. Among them, for 192 testing cases (or 73% of the cases) the ACO method outperforms the FDS method. There are only five cases where the ACO approach has worse average quality results. They all happened on the *invert_matrix_general* benchmark. On average, we can expect a 16.4% performance improvement over FDS. If only considering the best results among the five runs for each testing case, we achieve a 19.5% resource reduction averaged over all tested samples. The most outstanding results provided by the ACO method achieve a 75% resource reduction compared with FDS. These results are obtained on a few deadlines for the *jpeg_idct_ifast* benchmark.

Besides absolute quality of the results, one difference between FDS and the ACO method is that ACO method is relatively more stable. In our experiments, it is observed that the FDS approach can provide worse quality results as the deadline is relaxed. Using the *idctcol* as an example, FDS provides drastically worse results for deadlines ranging from 25 to 30 though it is able to reach decent scheduling qualities for deadline from 19 to 24. The same problem occurs for deadlines between 36 and 38. One possible reason is that as the deadline is extended, the time frame of each operation is also extended, which makes the force computation more likely to clash with similar values. Due to the lack of backtracking and good look-ahead capability, an early mistake would lead to inferior results. On the other hand, the ACO algorithm robustly generates monotonically non-increasing results with fewer resource requirements as the deadline increases.

13.4.3 Resource Constrained Scheduling: ACO vs. List Scheduling and ILP

We have implemented the ACO-based resource-constrained scheduling algorithm and compared its performance with the popularly used list scheduling and force-directed scheduling algorithms.

For each of the benchmark samples, we run the ACO algorithm with different choices of local heuristics. For each choice, we also perform five runs to obtain enough statistics for evaluating the stability of the algorithm. Again we fixed the number of ants per iteration 10 and in each run we allow 100 iterations. Other parameters are also the same as those used in the timing constrained problem. The best schedule latency is reported at the end of each run and then the average value is reported as the performance for the corresponding setting. Two different experiments are conducted for resource constrained scheduling – the homogenous case and the heterogenous case.

For the homogenous case, resource allocation is performed before the operation scheduling. Each operation is mapped to a unique resource type. In other words, there is no ambiguity on which resource the operation shall be handled during the scheduling step. In this experiment, similar to the timing constrained case, two types of resources (MUL/ALU) are allowed. The number of each resource type is predefined after making sure they do not make the experiment trivial (for example, if we are too generous, then the problem simplifies to an ASAP problem).

Comparing with a variety of list scheduling approaches and the force-directed scheduling method, the ACO algorithm generates better results consistently over all testing cases, which is demonstrated by the number of times that it provides the best results for the tested cases. This is especially true for the case when operation depth (OD) is used as the local heuristic, where we find the best results in 14 cases amongst 20 tested benchmarks. For other traditional methods, FDS generates the most hits (ten times) for best results, which is still less than the worst case of ACO (11 times). For some of the testing samples, our method provides significant improvement on the schedule latency. The biggest saving achieved is 22%. This is obtained for the COSINE2 benchmark when operation mobility (OM) is used as the local heuristic for our algorithm and also as the heuristic for constructing the priority list for the traditional list scheduler. For cases that our algorithm fails to provide the best solution, the quality of its results is also much closer to the best than other methods.

ACO also demonstrates much stronger stability over different input applications. As indicated in Sect. 13.3.4, the performance of traditional list scheduler heavily depends on the input application, while the ACO algorithm is much less sensitive to the choice of different local heuristics and input applications. This is evidenced by the fact that the standard deviation of the results achieved by the new algorithm is much smaller than that of the traditional list scheduler. The average standard deviation for list scheduling over all the benchmarks and different heuristic choices is 1.2, while for the ACO algorithm it is only 0.19. In other words, we can expect to achieve

high quality scheduling results much more stably on different application DFGs regardless of the choice of local heuristic. This is a great attribute desired in practice.

One possible explanation for the above advantage is the different ways how the scheduling heuristics are used by list scheduler and the ACO algorithm. In list scheduling, the heuristics are used in a greedy manner to determine the order of the operations. Furthermore, the schedule of the operations is done all at once. Differently, in the ACO algorithm, local heuristics are used stochastically and combined with the pheromone values to determine the operations' order. This makes the solution exploration more balanced. Another fundamental difference is that the ACO algorithm is an iterative process. In this process, the pheromone value acts as an indirect feedback and tries to reflect the quality of a potential component based on the evaluations of historical solutions that contain this component. It introduces a way to integrate global assessments into the scheduling process, which is missing in the traditional list or force-directed scheduling.

In the second experiment, heterogeneous computing units are allowed, i.e., one type of operation can be performed by different types of resources. For example, multiplication can be performed by either a faster multiplier or a regular one. Furthermore, multiple same type units are also allowed. For example, we may have three faster multipliers and two regular ones.

We conduct the heterogenous experiments with the same configuration as for the homogenous case. Moreover, to better assess the quality of our algorithm, the same heterogenous RCS tasks are also formulated as integer linear programming problems and then optimally solved using CPLEX. Since the ILP solution is time consuming to obtain, our heterogenous tests are only done for the classic samples.

Compared with a variety of list scheduling approaches and the force-directed scheduling method, the ACO algorithm generates better results consistently over all testing cases. The biggest saving achieved is 23%. This is obtained for the FIR2 benchmark when the latency weighted operation depth (LWOD) is used as the local heuristic. Similar to the homogenous case, our algorithm outperforms other methods in regards to consistently generating high-quality results. The average standard deviation for list scheduler over all the benchmarks and different heuristic choices is 0.8128, while that for the ACO algorithm is only 0.1673.

Though the results of force-directed scheduler generally outperform the list scheduler, our algorithm achieves even better results. On average, comparing with the force-directed approach, our algorithm provides a 6.2% performance enhancement for the testing cases, while performance improvement for individual test sample can be as much as 14.7%.

Finally, compared to the optimal scheduling results computed by using the integer linear programming model, the results generated by the ACO algorithm are much closer to the optimal than those provided by the list scheduling heuristics and the force-directed approach. For all the benchmarks with known optima, our algorithm improves the average schedule latency by 44% comparing with the list scheduling heuristics. For the larger size DFGs such as COSINE1 and COSINE2, CPLEX fails to generate optimal results after more than 10 h of execution on a SPARC workstation with a 440 MHz CPU and 384 MB memory. In fact, CPLEX

crashes for these two cases because of running out of memory. For COSINE1, CPLEX does provide a intermediate sub-optimal solution of 18 cycles before it crashes. This result is worse than the best result found by the ACO algorithm.

13.4.4 Further Assessment: ACO vs. Simulated Annealing

In order to further investigate the quality of the ACO-based algorithms, we compared them with a simulated annealing (SA) approach. Our SA implementation is similar to the algorithm presented in [43]. The basic idea is very similar to the ACO approach in which a meta-heuristic method (SA) is used to guide the searching process while a traditional list scheduler is used to evaluate the result quality. The scheduling result with the best resource usage is reported when the algorithm terminates.

The major challenge here is the construction of a *neighbor* selection in the SA process. With the knowledge of each operation's mobility range, it is trivial to see the search space for the TCS problem is covered by all the possible combinations of the operation/timestep pairs, where each operation can be scheduled into any time step in its mobility range. In our formulation, given a scheduling S where operation op_i is scheduled at t_i, we experimented with two different methods for generating a neighbor solution:

1. *Physical neighbor*: A neighbor of S is generated by selecting an operation op_i and rescheduling it to a physical neighbor of its current scheduled time step t_i, namely either $t_i + 1$ or $t_i - 1$ with even possibility. In case t_i is on the boundary of its mobility range, we treat the mobility range as a circular buffer;
2. *Random neighbor*: A neighbor of S is generated by selecting an operation and rescheduling it to any of the position in its mobility range excluding its currently scheduled position.

However, both of the above approaches suffer from the problem that a lot of these *neighbors* will be invalid because they may violate the data dependency posed by the DFG. For example, say, in S a single cycle operation op_1 is scheduled at time step 3, and another single cycle operation op_2 which is data dependent on op_1 is scheduled at time step 4. Changing the schedule of op_2 to step 3 will create an invalid scheduling result. To deal with this problem in our implementation, for each generated scheduling, we quickly check whether it is valid by verifying the operation's new schedule against those of its predecessor and successor operations defined in the DFG. Only valid schedules will be considered.

Furthermore, in order to give roughly equal chance to each operation to be selected in the above process, we try to generate multiple neighbors before any temperature update is taken. This can be considered as a local search effort, which is widely implemented in different variants of SA algorithm. We control this local search effort with a weight parameter θ. That is before any temperature update taking place, we attempt to generate θN valid scheduling candidates where N is the number

of operations in the DFG. In our work, we set $\theta = 2$, which roughly gives each operation two chances to alter its currently scheduled position in each cooling step.

This local search mechanism is applied to both neighbor generation schemes discussed above. In our experiments, we found there is no noticeable difference between the two neighbor generation approaches with respect to the quality of the final scheduling results except that the *random neighbor* method tends to take significantly more computing time. This is because it is more likely to come up with an invalid scheduling which are simply ignored in our algorithm. In our final realization, we always use the *physical neighbor* method.

Another issue related to the SA implementation is how to set the initial seed solution. In our experiments, we experimented three different seed solutions: ASAP, ALAP and a randomly generated valid scheduling. We found that SA algorithm with a randomly generated seed constantly outperforms that using the ASAP or ALAP initialization. It is especially true when the *physical neighbor* approach is used. This is not surprising since the ASAP and ALAP solutions tend to cluster operations together which is bad for minimizing resource usage. In our final realization, we always use a randomly generated schedule as the seed solution.

The framework of our SA implementation for both timing constrained and resource constrained scheduling is similar to the one reported in [51]. The acceptance of a more costly neighboring solution is determined by applying the Boltzmann probability criteria [1], which depends on the cost difference and the annealing temperature. In our experiments, the most commonly known and used geometric cooling schedule [51] is applied and the temperature decrement factor is set to 0.9. When it reaches the pre-defined maximum iteration number or the stop temperature, the best solution found by SA is reported.

Similar to the ACO algorithm, we perform five runs for each benchmark sample and report the average savings, the best savings, and the standard deviation of the reported scheduling results. It is observed that the SA method provides much worse results compared with the ACO solutions. In fact, the ACO approach provides better results on every testing case. Though the SA method does have significant gains on select cases over FDS, its average performance is actually worse than FDS by 5%, while our method provides a 16.4% average savings. This is also true if we consider the best savings achieved amongst multiple runs where a modest 1% savings is observed in SA comparing with a 19.5% reduction obtained by the ACO method. Furthermore, the quality of the SA method seems to be very dependent on the input applications. This is evidenced by the large dynamic range of the scheduling quality and the larger standard deviation over the different runs. Finally, we also want to make it clear that to achieve this result, the SA approach takes substantially more computing time than the ACO method. A typical experiment over all 263 testing cases will run between 9 to 12 h which is 3–4 times longer than the ACO-based TCS algorithm.

As discussed above, our SA formulation for resource constrained scheduling is similar to that studied in [43]. It is relatively more straight forward since it will always provide valid scheduling using a list scheduler. To be fair, a randomly generated operation list is used as the seed solution for the SA algorithm. The neighbor

solutions are constructed by swapping the positions of two neighboring operations in the current list. Since the algorithm always generates a valid scheduling, we can better control the runtime than in its TCS counterpart by adjusting the cooling scheme parameter. We carried experiments using execution limit ranging from 1 to 10 times of that of the ACO approach. It was observed that SA RCS algorithm provides poor performance when the time limit was too short. On the other hand, once we increase this time limit to over five times of the ACO execution time, there was no significant improvement on the results as the execution time increased. It is observed that the ACO-based algorithm consistently outperforms it while using much less computing time.

13.5 Design Space Exploration

As a direct application of the operation scheduling algorithms, we examine the Design Space Exploration problem in this section, which is not only of theoretical interest but also encountered frequently in real-life high level synthesis practice.

13.5.1 Problem Formulation and Related Work

When building a digital system, designers are faced with a countless number of decisions. Ideally, they must deliver the smallest, fastest, lowest power device that can implement the application at hand. More often than not, these design parameters are contradictory. Designers must be able to reason about the tradeoffs amongst a set of parameters. Such decisions are often made based on experience, i.e., this worked before, it should work again. Exploration tools that can quickly survey the design space and report a variety of options are invaluable.

From optimization point of view, design space exploration can be distilled to identifying a set of Pareto optimal design points according to some objective function. These design points form a curve that provides the best tradeoffs for the variables in the objective function. Once the curve is constructed, the designer can make design decisions based on the relative merits of the various system configurations. Timing performance and the hardware cost are two common objectives in such process.

Resource allocation and scheduling are two fundamental problems in constructing such Pareto optimal curves for time/cost tradeoffs. By applying resource constrained scheduling, we try to minimize the application latency without violating the resource constraints. Here allocation is performed before scheduling, and a different resource allocation will likely produce a vastly different scheduling result. On the other hand, we could perform scheduling before allocation; this is the time constrained scheduling problem. Here the inputs are a data flow graph and a time deadline (latency). The output is again a start time for each operation, such that the latency is not violated, while attempting to minimize the number of resources that

are needed. It is not clear as to which solution is better. Nor is it clear on the order that we should perform scheduling and allocation.

Obviously, one possible method of design space exploration is to vary the constraints to probe for solutions in a point-by-point manner. For instance, you can use some time constrained algorithm iteratively, where each iteration has a different input latency. This will give you a number of solutions, and their various resource allocations over a set of time points. Or you can run some resource constrained algorithm iteratively. This will give you a latency for each of these area constraints.

Design space exploration problem has been the focus in numerous studies. Though it is possible to formulate the problems using Integer Linear Program (ILP), they quickly become intractable when the problem sizes get large. Much research has been done to cleverly use heuristic approaches to address these problems. Actually, one major motivation of the popularly used Force Directed Scheduling (FDS) algorithm [34] was to address the design space exploration task, i.e., by performing FDS to solve a series timing constrained scheduling problems. In the same paper, the authors also proposed a method called force-directed list scheduling (FDLS) to address the resource constrained scheduling problem. The FDS method is constructive since the solution is computed without backtracking. Every decision is made deterministically in a greedy manner. If there are two potential assignments with the same cost, the FDS algorithm cannot accurately estimate the best choice. Moreover, FDS does not take into account future assignments of operators to the same control step. Consequently, it is possible that the resulting solution will not be optimal due to it's greedy nature. FDS works well on small sized problems, however, it often results to inferior solutions for more complex problems. This phenomena is observed in our experiments reported in Sect. 13.4.

In [16], the authors concentrate on providing alternative "module bags" for design space exploration by heuristically solving the clique partitioning problems and using force directed scheduling. Their work focuses more on the situations where the operations in the design can be executed on alternative resources. In the Voyager system [11], scheduling problems are solved by carefully bounding the design space using ILP, and good results are reported on small sized benchmarks. Moreover, it reveals that clock selection can have an important impact on the final performance of the application. In [14, 21, 32], genetic algorithms are implemented for design space exploration. Simulated annealing [29] has also been applied in this domain. A survey on design space exploration methodologies can be found in [28] and [9].

In this chapter, we focus our attention on the basic design space exploration problem similar to the one treated in [34], where the designers are faced with the task of mapping a well defined application represented as a DFG onto a set of known resources where the compatibility between the operations and the resource types has been defined. Furthermore, the clock selection has been determined in the form of execution cycles for the operations. The goal is to find the a Pareto optimal tradeoff amongst the design implementations with regard to timing and resource costs. Our basic method can be extended to handle clock selection and the use of alternative resources. However, this is beyond the scope of this discussion.

13.5.2 Exploration Using Time and Resource Constrained Duality

As we have discussed, traditional approaches solve the design space exploration task solving a series of scheduling problems, using either a resource constrained method or a timing constrained method. Unfortunately, designers are left with individual tools for tackling either problem. They are faced with questions like: Where do we start the design space exploration? What is the best way to utilize the scheduling tools? When do we stop the exploration?

Moreover, due to the lack of connection amongst the traditional methods, there is very little information shared between time constrained and resource constrained solutions. This is unfortunate, as we are throwing away potential solutions since solving one problem can offer more insight into the other problem. Here we propose a novel design exploration method that exploits the duality of the time and resource constrained scheduling problems. Our exploration automatically constructs a time/area tradeoff curve in a fast, effective manner. It is a general approach and can be combined with any high quality scheduling algorithm.

We are concerned with the design problem of making tradeoffs between hardware cost and timing performance. This is still a commonly faced problem in practice, and other system metrics, such as power consumption, are closely related with them. Based on this, we have a 2-D design space as illustrated in Fig. 13.1a, where the x-axis is the execution deadline and the y-axis is the aggregated hardware cost. Each point represents a specific tradeoff of the two parameters.

For a given application, the designer is given R types of computing resources (e.g., multipliers and adders) to map the application to the target device. We define a specific design as a *configuration*, which is simply the number of each specific resource type. In order to keep the discussion simple, in the rest of the paper we

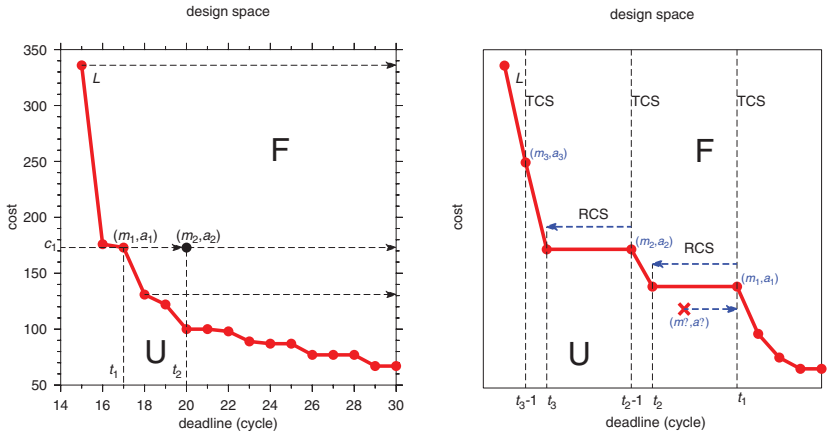

Fig. 13.1 Design space exploration using duality between schedule problems (curve L gives the optimal time/cost tradeoffs)

assume there are only two resource types M (multiply) and A (add), though our algorithm is not limited to this constraint. Thus, each configuration can be specified by (m,a) where m is the number of resource M and a is the number of A. For each specific configuration we have the following lemma about the portion of the design space that it maps to.

Lemma 1. *Let C be a feasible configuration with cost c for the target application. The configuration maps to a horizontal line in the design space starting at (t_{min},c), where t_{min} is the resource constrained minimum scheduling time.*

The proof of the lemma is straightforward as each feasible configuration has a minimum execution time t_{min} for the application, and obviously it can handle every deadline longer than t_{min}. For example, in Fig. 13.1a, if the configuration (m_1,a_1) has a cost c_1 and a minimum scheduling time t_1, the portion of design space that it maps to is indicated by the arrow next to it. Of course, it is possible for another configuration (m_2,a_2) to have the same cost but a bigger minimum scheduling time t_2. In this case, their feasible space overlaps beyond (t_2,c_1).

As we discussed before, the goal of design space exploration is to help the designer find the optimal tradeoff between the time and area. Theoretically, this can be done by finding the minimum area c amongst all the configurations that are capable of producing $t \in [t_{asap},t_{seq}]$, where t_{asap} is the ASAP time for the application while t_{seq} is the sequential execution time. In other words, we can find these points by performing time constrained scheduling (TCS) on all t in the interested range. These points form a curve in the design space, as illustrated by curve L in Fig. 13.1a. This curve divides the design space into two parts, labeled with F and U respectively in Fig. 13.1a, where all the points in F are feasible to the given application while U contains all the unfeasible time/area pairs. More interestingly, we have the following attribute for curve L:

Lemma 2. *Curve L is monotonically non-increasing as the deadline t increases.*[1]

Due to this lemma, we can use the dual solution of finding the tradeoff curve by identifying the minimum resource constrained scheduling (RCS) time t amongst all the configurations with cost c. Moreover, because the monotonically non-increasing property of curve L, there may exist horizontal segments along the curve. Based on our experience, horizontal segments appear frequently in practice. This motivates us to look into potential methods to exploit the duality between RCS and TCS to enhance the design space exploration process. First, we consider the following theorem:

Theorem 1. *If C is a configuration that provides the minimum cost at time t_1, then the resource constrained scheduling result t_2 of C satisfies $t_2 \leqslant t_1$. More importantly, there is no configuration C' with a smaller cost that can produce an execution time within $[t_2,t_1]$.*[2]

[1] Proof is omitted because of page limitation.

[2] Proof is omitted because of page limitation.

This theorem provides a key insight for the design space exploration problem. It says that if we can find a configuration with optimal cost c at time t_1, we can move along the horizontal segment from (t_1, c) to (t_2, c) without losing optimality. Here t_2 is the RCS solution for the found configuration. This enables us to efficiently construct the curve L by iteratively using TCS and RCS algorithms and leveraging the fact that such horizontal segments do frequently occur in practice. Based on the above discussion, we propose a new space exploration algorithm as shown in Algorithm 1 that exploits the duality between RCS and TCS solutions. Notice the *min* function in step 10 is necessary since a heuristic RCS algorithm may not return the true optimal that could be worse than t_{cur}.

By iteratively using the RCS and TCS algorithms, we can quickly explore the design space. Our algorithm provides benefits in runtime and solution quality compared with using RCS or TCS alone. Our algorithm performs exploration starting from the largest deadline t_{max}. Under this case, the TCS result will provide a configuration with a small number of resources. RCS algorithms have a better chance to find the optimal solution when the resource number is small, therefore it provides a better opportunity to make large horizontal jumps. On the other hand, TCS algorithms take more time and provide poor solutions when the deadline is unconstrained. We can gain significant runtime savings by trading off between the RCS and TCS formulations.

The proposed framework is general and can be combined with any scheduling algorithm. We found that in order for it to work in practice, the TCS and RCS algorithms used in the process require special characteristics. First, they must be fast, which is generally requested for any design space exploration tool. More importantly, they must provide close to optimal solutions, especially for the TCS problem. Otherwise, the conditions for Theorem 1 will not be satisfied and the generated curve L will suffer significantly in quality. Moreover, notice that we enjoy the biggest jumps when we take the minimum RCS result amongst all the configurations

Algorithm 1 Iterative design space exploration algorithm

procedure DSE
output: curve L
1: interested time range $[t_{min}, t_{max}]$, where $t_{min} \geqslant t_{asap}$ and $t_{max} \leqslant t_{seq}$.
2: $L = \phi$
3: $t_{cur} = t_{max}$
4: **while** $t_{cur} \geqslant t_{min}$ **do**
5: perform TCS on t_{cur} to obtain the optimal configurations C_i.
6: **for** configuration C_i **do**
7: perform RCS to obtain the minimum time t^i_{rcs}
8: **end for**
9: $t_{rcs} = \min_i (t^i_{rcs})$ /* find the best rcs time */
10: $t_{cur} = \min(t_{cur}, t_{rcs}) - 1$
11: extend L based on TCS and RCS results
12: **end while**
13: return L

that provide the minimum cost for the TCS problem. This is reflected in Steps 6–9 in Algorithm 1. For example, it is possible that both (m,a) and (m',a') provide the minimum cost at time t but they have different deadline limits. Therefore a good TCS algorithm used in the proposed approach should be able to provide multiple candidate solutions with the same minimum cost, if not all of them.

13.6 Conclusion

In this chapter, we provide a comprehensive survey on various operation scheduling algorithms, including List Scheduling, Force-Directed Scheduling, Simulated Annealing, Ant Colony Optimization (ACO) approach, together with others. We report our evaluation for the aforementioned algorithms against a comprehensive set of benchmarks, called ExpressDFG. We give the characteristics of these benchmarks and discuss suitability for evaluating scheduling algorithms. We present detailed performance evaluation results in regards of solution quality, stability of the algorithms, their scalability over different applications and their runtime efficiency. As a direct application, we present a uniformed design space exploration method that exploits duality between the timing and resource constrained scheduling problems.

Acknowledgment This work was partially supported by National Science Foundation Grant CNS-0524771.

References

1. Aarts, E. and Korst, J. (1989). *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Wiley, New York, NY.
2. Adam, T. L., Chandy, K. M., and Dickson, J. R. (1974). A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690.
3. Aigner, G., Diwan, A., Heine, D. L., Moore, M. S. L. D. L., Murphy, B. R., and Sapuntzakis, C. (2000). *The Basic SUIF Programming Guide*. Computer Systems Laboratory, Stanford University.
4. Aletà, A., Codina, J. M., and and Antonio G., Jesús S. (2001). Graph-Partitioning Based Instruction Scheduling for ClusteredProcessors. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture.*
5. Auyeung, A., Gondra, I., and Dai, H. K. (2003). Integrating random ordering into multi-heuristic list scheduling genetic algorithm. *Advances in Soft Computing: Intelligent Systems Design and Applications*. Springer, Berlin Heidelberg New York.
6. Beaty, Steve J. (1993). Genetic algorithms versus tabu search for instruction scheduling. In *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms.*
7. Beaty, Steven J. (1991). Genetic algorithms and instruction scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture.*
8. Bernstein, D., Rodeh, M., and Gertner, I. (1989). On the Complexity of Scheduling Problems for Parallel/PipelinedMachines. *IEEE Transactions on Computers*, 38(9):1308–1313.

9. C. McFarland, M., Parker, A. C., and Camposano, R. (1990). The high-level synthesis of digital systems. In *Proceedings of the IEEE*, vol. 78, pp. 301–318.
10. Camposano, R. (1991). Path-based scheduling for synthesis. *IEEE Transaction on Computer-Aided Design*, 10(1):85–93.
11. Chaudhuri, S., Blythe, S. A., and Walker, R. A. (1997). A solution methodology for exact design space exploration in a three-dimensional design space. *IEEE Transactions on very Large Scale Integratioin Systems*, 5(1):69–81.
12. Corne, D., Dorigo, M., and Glover, F., editors (1999). *New Ideas in Optimization*. McGraw Hill, London.
13. Deneubourg, J. L. and Goss, S. (1989). Collective Patterns and Decision Making. *Ethology, Ecology and Evolution*, 1:295–311.
14. Dick, R. P. and Jha, N. K. (1997). MOGAC: A Multiobjective Genetic Algorithm for the Co-Synthesis of Hardware-Software Embedded Systems. In *IEEE/ACM Conference on Computer Aided Design*, pp. 522–529.
15. Dorigo, M., Maniezzo, V., and Colorni, A. (1996). Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man and Cybernetics, Part-B*, 26(1):29–41.
16. Dutta, R., Roy, J., and Vemuri, R. (1992). Distributed design-space exploration for high-level synthesis systems. In *DAC '92*, pp. 644–650. IEEE Computer Society Press, Los Alamitos, CA.
17. ExpressDFG (2006). ExpressDFG benchmark web site. *http://express.ece.ucsb.edu/benchmark/*.
18. Grajcar, M. (1999). Genetic List Scheduling Algorithm for Scheduling and Allocationon a Loosely Coupled Heterogeneous Multiprocessor System. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation Conference*.
19. Gutjahr, W. J. (2002). Aco algorithms with guaranteed convergence to the optimal solution. *Information Processing Letters*, 82(3):145–153.
20. Heijligers, M. and Jess, J. (1995). High-level synthesis scheduling and allocation using genetic algorithms based on constructive topological scheduling techniques. In *International Conference on Evolutionary Computation*, pp. 56–61, Perth, Australia.
21. Heijligers, M. J. M., Cluitmans, L. J. M., and Jess, J. A. G. (1995). High-level synthesis scheduling and allocation using genetic algorithms. p. 11.
22. Hu, T. C. (1961). Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848.
23. Kennedy, K. and Allen, R. (2001). *Optimizing Compilers for Modern Architectures: A Dependence-basedApproach*. Morgan Kaufmann, San Francisco.
24. Kernighan, B. W. and Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307.
25. Kolisch, R. and Hartmann, S. (1999). Heuristic algorithms for solving the resource-constrained project scheduling problem: classification and computational analysis. *Project Scheduling: Recent Models, Algorithms and Applications*. Kluwer Academic, Dordrecht.
26. Lee, C., Potkonjak, M., and Mangione-Smith, W. H. (1997). Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*.
27. Lee, J.-H., Hsu, Y.-C., and Lin, Y.-L. (1989). A new integer linear programming formulation for the scheduling problem in data path synthesis. In *Proceedings of ICCAD-89*, pp. 20–23, Santa Clara, CA.
28. Lin, Y.-L. (1997). Recent developments in high-level synthesis. *ACM Transactions on Design of Automation of Electronic Systems*, 2(1):2–21.
29. Madsen, J., Grode, J., Knudsen, P. V., Petersen, M. E., and Haxthausen, A. (1997). LYCOS: The Lyngby Co-Synthesis System. *Design Automation for Embedded Systems*, 2(2):125–63.
30. Memik, S. O., Bozorgzadeh, E., Kastner, R., and MajidSarrafzadeh (2001). A super-scheduler for embedded reconfigurable systems. In *IEEE/ACM International Conference on Computer-Aided Design*.

31. Micheli, G. De (1994). *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York.
32. Palesi, M. and Givargis, T. (2002). Multi-Objective Design Space Exploration Using GeneticAlgorithms. In *Proceedings of the Tenth International Symposium on Hardware/Software-Codesign*.
33. Park, I.-C. and Kyung, C.-M. (1991). Fast and near optimal scheduling in automatic data path synthesis. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pp. 680–685. ACM Press, New York, NY.
34. Paulin, P. G. and Knight, J. P. (1987). Force-directed scheduling in automatic data path synthesis. In *24th ACM/IEEE Conference Proceedings on Design Automation Conference*.
35. Paulin, P. G. and Knight, J. P. (1989). Force-directed scheduling for the behavioral synthesis of asic's. *IEEE Transactions on Computer-Aided Design*, 8:661–679.
36. Poplavko, P., van Eijk, C. A. J., and Basten, T. (2000). Constraint analysis and heuristic scheduling methods. In *Proceedings of 11th Workshop on Circuits, Systems and Signal Processing(ProRISC2000)*, pp. 447–453.
37. Schutten, J. M. J. (1996). List scheduling revisited. *Operation Research Letter*, 18:167–170.
38. Semiconductor Industry Association (2003). National Technology Roadmap for Semiconductors.
39. Sharma, A. and Jain, R. (1993). Insyn: Integrated scheduling for dsp applications. In *DAC*, pp. 349–354.
40. Smith, J. E. (1989). Dynamic instruction scheduling and the astronautics ZS-1. *IEEE Computer*, 22(7):21–35.
41. Smith, M. D. and Holloway, G. (2002). *An Introduction to Machine SUIF and Its Portable Librariesfor Analysis and Optimization*. Division of Engineering and Applied Sciences, Harvard University.
42. Stützle, T. and Hoos, H. H. (2000). MAX–MIN Ant System. *Future Generation Computer Systems*, 16(9):889–914.
43. Sweany, P. H. and Beaty, S. J. (1998). Instruction scheduling using simulated annealing. In *Proceedings of 3rd International Conference on Massively Parallel Computing Systems.*
44. Topcuouglu, H., Hariri, S., and you Wu, M. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274.
45. Verhaegh, W. F. J., Aarts, E. H. L., Korst, J. H. M., and Lippens, P. E. R. (1991). Improved force-directed scheduling. In *EURO-DAC '91: Proceedings of the Conference on European Design Automation*, pp. 430–435. IEEE Computer Society Press, Los Alamitos, CA.
46. Verhaegh, W. F. J., Lippens, P. E. R., Aarts, E. H. L., Korst, J. H. M., van der Werf, A., and van Meerbergen, J. L. (1992). Efficiency improvements for force-directed scheduling. In *ICCAD '92: Proceedings of the 1992 IEEE/ACM international Conference on Computer-Aided Design*, pp. 286–291. IEEE Computer Society Press, Los Alamitos, CA.
47. Wang, G., Gong, W., DeRenzi, B., and Kastner, R. (2006). Ant Scheduling Algorithms for Resource and Timing Constrained Operation Scheduling. *IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 26(6):1010–1029.
48. Wang, G., Gong, W., and Kastner, R. (2003). A New Approach for Task Level Computational ResourceBi-partitioning. *15th International Conference on Parallel and Distributed Computing and Systems*, 1(1):439–444.
49. Wang, G., Gong, W., and Kastner, R. (2004). System level partitioning for programmable platforms using the ant colony optimization. *13th International Workshop on Logic and Synthesis, IWLS'04*.
50. Wang, G., Gong, W., and Kastner, R. (2005). Instruction scheduling using MAX–MIN ant optimization. In *15th ACM Great Lakes Symposium on VLSI, GLSVLSI'2005*.
51. Wiangtong, T., Cheung, P. Y. K., and Luk, W. (2002). Comparing Three Heuristic Search Methods for FunctionalPartitioning in Hardware-Software Codesign. *Design Automation for Embedded Systems*, 6(4):425–49.