

Ant Colony Optimizations for Resource and Timing Constrained Operation Scheduling

Gang Wang, *Member, IEEE*, Wenrui Gong, *Student Member, IEEE*, Brian DeRenzi, *Student Member, IEEE*,
and Ryan Kastner, *Member, IEEE*

Abstract—Operation scheduling is a fundamental problem in mapping an application to a computational device. It takes a behavioral application specification and produces a schedule to either minimize the completion time or the computing resources required to meet a given deadline. The operation scheduling problem is \mathcal{NP} -hard, thus effective heuristic methods are necessary to provide qualitative solutions. We present novel operation scheduling algorithms using the Ant Colony Optimization approach for both timing and resource constrained scheduling problems. The algorithms use a unique hybrid approach by combining the MAX-MIN ant system meta-heuristic with traditional scheduling heuristics. We compiled a comprehensive testing benchmark set from real-world applications in order to verify the effectiveness and efficiency of our proposed algorithms. For timing constrained scheduling, our algorithm achieves better results compared with force-directed scheduling on almost all the testing cases with a maximum 19.5% reduction of the number of resources. For resource constrained scheduling, our algorithm outperforms a number of different list scheduling heuristics with better stability, and generates better results with up to 14.7% improvement. Our algorithms outperform the simulated annealing method for both scheduling problems in terms of quality, computing time and stability.

Index Terms—Operation Scheduling, Force-Directed Scheduling, List Scheduling, MAX-MIN Ant System

I. 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 [1]. 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 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.

Manuscript received December 7, 2005; revised June 6, 2006. This work was supported by National Science Foundation Grant CNS-0524771.

The authors are with the Department of Electrical and Computer Engineering, University of California at Santa Barbara.

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 [2], distributed clustering computation architectures [3] and behavioral synthesis of ASICs and FPGAs [4]. In this work, we focus on operation scheduling for behavioral synthesis for ASICs/FPGAs. However, the basic algorithms proposed here can be modified to handle a wide variety of operation scheduling problems.

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). Figure 1 shows an example DFG for a one-dimensional 8-point fast discrete cosine transformation.

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.

Operation scheduling methods can be further classified as *static scheduling* and *dynamic scheduling* [5]. 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.

In this paper, we focus on both resource and timing constrained static operation scheduling. We propose iterative algorithms based on the MAX-MIN Ant Colony Optimization for solving these problems. In our algorithms, a collection of agents (ants) cooperate together to search for a solution. Global and local heuristics are combined in a stochastic decision making process in order to efficiently explore the search space. The quality of the resultant schedules is evaluated and fed back to dynamically adjust the heuristics for future iterations. The main contribution of our work is the formulation of scheduling algorithms that:

- Utilize a unique hybrid approach combining traditional heuristics and the recently developed MAX-MIN ant system optimization [6];
- Dynamically use local and global heuristics based on the input application to adaptively search the solution space;
- Generate consistently good scheduling results over all testing cases compared with a range of list scheduling heuristics, force-directed scheduling, simulated annealing and the optimal ILP solution, and demonstrates stable quality over a variety of application benchmarks of large size.

The paper is organized as follows. We formally define the timing constrained and resource constrained scheduling problems in Section II. In Section III, we give a brief review on the MAX-MIN Ant Colony Optimization (ACO). Then in Section IV and Section V, we present two hybrid approaches combining traditional scheduling heuristics with the MAX-MIN ant system optimization to solve the timing and resource constrained scheduling problems, respectively. We discuss the construction of our benchmarks in Section VI. Experimental results for the new algorithms are presented and analyzed in Section VII. In Section VIII, we compare our work with a related study. We conclude with Section IX.

II. PRELIMINARIES

A. Operation Scheduling Problem Definition

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 $v_i \in V (i = 1, \dots, n)$ 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 \preceq) 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 ($1 \leq j \leq R$). 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 [7]. 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

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

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. $s_i \geq f_j$ if $op_j \preceq op_i$;
- 2) At any given cycle t , the number of resources needed is constrained by r_j , for all $1 \leq j \leq R$.

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. $s_{end} < D$.

B. Related Work

Many variants of the operation scheduling problem are \mathcal{NP} -hard [8]. Although it is possible to formulate and solve them using Integer Linear Programming (ILP) [9], the feasible solution space quickly becomes intractable for larger problem instances. In order to address this problem, 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 [10], [11]. Verhaegh *et al.* [12], [13] 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. Due to the lack of a look ahead scheme, the FDS algorithm is likely to produce a sub-optimal solution. One way to address this issue is the

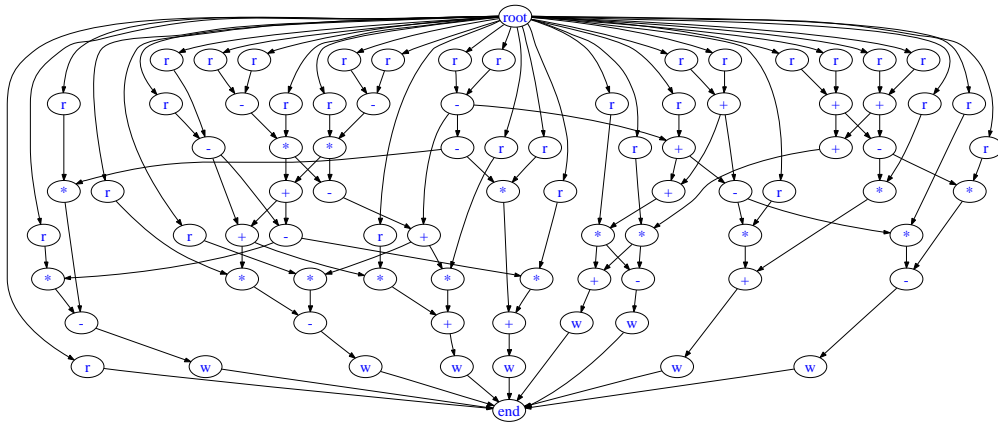


Fig. 1: Data Flow Graph (DFG) of the *cosine2* benchmark ('r' is for memory read and 'w' for memory write).

iterative method proposed by Park and Kyung [14] based on Kernighan and Lin's heuristic [15] 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. More recently, Heijligers *et al.* [16] and InSyn [17] use evolutionary techniques like genetic algorithms and simulated evolution.

There are a number of algorithms for the resource constrained problem, including list scheduling [7], [18], forced-directed scheduling [10], genetic algorithm [19], tabu search [20], simulated annealing [21], graph theoretic and computational geometry approaches [3]. Among them, list scheduling is the most common due to its simplicity of implementation and capability of generating reasonably good results for small sized problems. The success of the list scheduler is highly dependent on the priority function and the structure of the input application (DFG) [21], [4], [22]. 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 [18], [22], [19], [23]. 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 Section VII confirm this.

III. ANT COLONY OPTIMIZATION

Before we describe our ant colony optimizations for operation scheduling, we give a brief description of ant colony optimization (ACO) meta-heuristic and define terminology that we later use in our ACO formulations. Those familiar with ACO can skip or skim this section.

A. Basic Ant Colony Optimization

The Ant Colony Optimization algorithm, originally introduced by Dorigo *et al.* [24], is a cooperative heuristic searching

algorithm inspired by ethological studies on the behavior of ants. It was observed [25] 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 shorter trails, which in turn creates an even larger amount of pheromone on those short trails, making them more likely to be chosen by 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) [24], and it gave competitive results compared to traditional methods. The TSP can be modeled as a complete weighted directed graph $G = (V, E, d)$ where $V = \{1, 2, \dots, n\}$ is a set of vertices or cities, $E = \{(i, j) | (i, j) \in V \times V\}$ is a set of edges, and d is a function that associates a numeric weight d_{ij} for each edge (i, j) in E . This weight is naturally interpreted as the distance between city i and j . The objective is to find a Hamiltonian path for G which gives the minimal length.

In order to solve the TSP problem, ACO associates a pheromone trail τ_{ij} for each edge (i, j) in E . The pheromone indicates the attractiveness of the edge and serves as a distributed global heuristic. Initially, τ_{ij} is set with some fixed value τ_0 . For each iteration, m ants are released randomly on the cities, and each starts to construct a tour. Every ant will have memory about the cities it has visited so far in order to guarantee the constructed tour is a Hamiltonian path. If at step t the ant is at city i , the ant chooses the next city j

probabilistically using the following formula:

$$p_{ij} = \begin{cases} \frac{\tau_{ij}(t)^\alpha \cdot \eta_{ij}^\beta}{\sum_k (\tau_{ik}^\alpha(t) \cdot \eta_{ik}^\beta)} & \text{if } j \text{ not visited} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where edges (i, k) are all the allowed moves from i , η_{ik} is a local heuristics which is defined as the inverse of d_{ij} , α and β are parameters to control the relative influence of the distributed global heuristic τ_{ik} and local heuristic η_{ik} . Intuitively, the ant favors a decision on an edge that possesses higher volume of pheromone and better local distance. At the finish of each iteration, the pheromone trails are updated. More specifically, we have:

$$\tau_{ij}(t) = \rho \cdot \tau_{ij}(t) + \sum_{k=1}^m \Delta\tau_{ij}^k(t) \quad \text{where } 0 < \rho < 1. \quad (2)$$

Here ρ is the evaporation ratio within the range $[0,1]$, and $\Delta\tau_{ij}^k = Q/L_k$ if edge (i, j) is included in the tour ant k constructed, otherwise $\Delta\tau_{ij}^k = 0$. Q is a fixed constant to control the delivery rate of the pheromone, while L_k is the tour length for ant k . Two important operations are performed in this updating process. The evaporation operation is necessary for the ACO to be effective in exploring different parts of the search space, while the reinforcement operation ensures that frequently used edges and edges contained in the better tours receive a higher volume of pheromone and will have a better chance of being selected in the future iterations of the algorithm. The above process is repeated multiple times until certain ending condition is reached. The best result found by the algorithm is reported.

Researchers have since formulated ACO methods for a variety of traditional \mathcal{NP} -hard problems. These problems include the maximum clique problem [26], the quadratic assignment problem [27], the graph coloring problem [28], the shortest common super-sequence problem [29], [30], and the multiple knapsack problem [31]. ACO has also been applied to practical problems such as the vehicle routing problem [32], data mining [33], network routing problem [34] and the system level task partitioning problem [35], [36], [37].

Premature convergence to local minima is a critical algorithmic issue that can be experienced by all evolutionary algorithms. Balancing exploration and exploitation is not trivial in these algorithms, especially for algorithms that use positive feedback such as ACO. This problem was formally investigated in [38]. It was shown that ACO with a time-dependent evaporation factor or a time-dependent lower pheromone bound converges to an optimal solution with probability of exactly one. Similar to the optimality proof for the simulated annealing meta-heuristic, such a global convergence guarantee can be obtained by a suitable speed of *cooling* (i.e., reduction of the influence of randomness). Though they failed in providing any constructive approach, the authors suggested that it is theoretically achievable by decreasing the evaporation factors, or by slowly decreasing the lower pheromone bounds.

B. MAX – MIN Ant System (MMAS)

The MAX-MIN Ant System (MMAS) [6] is built upon the original ACO algorithm and is specifically designed to

address the premature convergence problem. It improves the original ACO by providing dynamically evolving bounds on the pheromone trails such that the heuristic value is always within a limit to that of the best path. As a result, all possible paths will have a non-trivial probability of being selected and thus it encourages broader exploration of the search space.

More specifically, MMAS forces the pheromone trails to be limited within evolving bounds, that is for iteration t , $\tau_{min}(t) \leq \tau_{ij}(t) \leq \tau_{max}(t)$. If we use f to denote the cost function of a specific solution S , the upper bound τ_{max} [6] is shown in (3). Here $S^{gb}(\cdot)$ represents the global best solution found so far in all iterations.

$$\tau_{max}(t) = \frac{1}{1 - \rho} \frac{1}{f(S^{gb}(t-1))} \quad (3)$$

The lower bound is defined as (4):

$$\tau_{min}(t) = \frac{\tau_{max}(t)(1 - \sqrt[p_{best}]{p_{best}})}{(avg - 1) \sqrt[p_{best}]{p_{best}}} \quad (4)$$

where $p_{best} \in (0, 1]$ is a controlling parameter to dynamically adjust the bounds of the pheromone trails. The physical meaning of p_{best} is that it indicates the conditional probability of the current global best solution $S^{gb}(t)$ being selected given that all edges not belonging to the global best solution have a pheromone level of $\tau_{min}(t)$ and all edges in the global best solution have $\tau_{max}(t)$. Here avg is the average size of the decision choices over all the iterations. For a TSP problem of n cities, $avg = n/2$. It is noted from (4) that lowering p_{best} will result in a tighter range for the pheromone heuristic. As $p_{best} \rightarrow 0$, $\tau_{min}(t) \rightarrow \tau_{max}(t)$, which means more emphasis is given to search space exploration.

Theoretical treatment of using the pheromone bounds and other modifications on the original ACO algorithm are proposed in [6]. These include a pheromone updating policy that only utilizes the best performing ant, initializing pheromone with τ_{max} and combining local search with the algorithm. It was reported that MMAS was the best performing ACO approach and provided very high quality solutions.

IV. MAX – MIN ANT SYSTEM FOR TIMING CONSTRAINED SCHEDULING

In this section, we introduce our MMAS-based algorithms for solving the timing constrained scheduling problem. As discussed in Section II, force-directed scheduling (FDS) is a commonly used heuristic as it generates “good” quality results for moderately sized DFGs. Our algorithm uses distribution graphs from FDS as a local heuristic. Additionally, we use the results produced by FDS to evaluate the quality of our algorithm. For these reasons, we provide some details of FDS in the following subsection. The remaining subsections describe our MMAS algorithm for timing constrained scheduling.

A. Force-Directed Scheduling

The force-directed scheduling algorithm (and its various forms) has been widely used since it was first proposed by Paulin and Knight [10]. The goal of the 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 $\Delta_i \geq 1$ time steps to execute, this probability is given as:

$$p_j(op_i) = \begin{cases} (\sum_{l=0}^{\Delta_i} h_i(j-l))/(t_i^L - t_i^S + 1) & \text{if } t_i^S \leq j \leq t_i^L \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

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 ,

$$q_k(j) = \sum_{op_i} p_j(op_i) \quad \text{if type of } op_i \text{ is } k \quad (6)$$

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:

$$SF_{ij} = \sum_{l=t_i^S}^{t_i^L + \Delta_i} q_k(l)(H_i(l) - p_i(l)) \quad (7)$$

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 $[j, j + \Delta_i]$.

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 $[\tilde{t}_l^S, \tilde{t}_l^L]$. The force exerted by a predecessor or successor is given by:

$$PSF_{ij}(l) = \sum_{m=\tilde{t}_l^S}^{\tilde{t}_l^L + \Delta_l} (q_k(m) \cdot \tilde{p}_m(op_l)) - \sum_{m=t_l^S}^{t_l^L + \Delta_l} (q_k(m) \cdot p_m(op_l)) \quad (8)$$

where $\tilde{p}_m(op_l)$ is computed in the same way as Equation (5) except the updated mobility information $[\tilde{t}_l^S, \tilde{t}_l^L]$ 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.

$$\text{total force}_{ij} = SF_{ij} + \sum_l PSF_{ij}(l) \quad (9)$$

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 [10] 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 Section VII.

B. MMAS for Timing Constrained Scheduling

We address the timing constrained scheduling (TCS) problem in an evolutionary manner. The proposed algorithm is built upon the Ant System approach and the TCS problem is formulated as an iterative searching process. Each iteration consists of two stages. First, the ACO algorithm is applied in which a collection of ants traverse the DFG to construct individual operation schedules with respect to the specified deadline using global and local heuristics. Second, these results are evaluated using their resource costs. The heuristics are 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 $j = 1, \dots, D$ and D is the specified deadline. These pheromone trails indicate the global favorableness of assigning the i -th operation at the j -th 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 operation and determining its desired control step. However,

unlike the deterministic approach used in the FDS method, each ant picks up the next operation probabilistically. The simplest way is to select an operation uniformly among all unscheduled operations. Once an operation op_h is selected, the ant needs to make a decision on which control step it should be assigned to. This decision is also made probabilistically according to Equation (10).

$$p_{hj} = \begin{cases} \frac{\tau_{hj}(t)^\alpha \cdot \eta_{hj}^\beta}{\sum_l (\tau_{hl}(t)^\alpha \cdot \eta_{hl}^\beta)} & \text{if } op_h \text{ can be scheduled at } l \text{ and } j \\ 0 & \text{otherwise} \end{cases} \quad (10)$$

Here j is the control step under consideration, which is between op_h 's time frame $[t_h^S, t_h^L]$. 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} . In our work, assuming op_h is of type k , we simply set η_{hj} to be the inverse of $q_k(j)$; that is the distribution graph value of type k at control step j (calculated in the same way as in FDS). Recalling our discussion in Section IV-A, q_k 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 . Intuitively, the ant favors a decision that possesses higher volume of pheromone and better local heuristic, i.e. a lower q_k . 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. Similar to FDS, once an operation is fixed at a time step, it will not change. Furthermore, the time frames will be 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:

$$\tau_{ij}(t) = \rho \cdot \tau_{ij}(t) + \sum_{h=1}^m \Delta\tau_{ij}^h(t) \quad \text{where } 0 < \rho < 1. \quad (11)$$

Here ρ is the evaporation ratio, and

$$\Delta\tau_{ij}^h = \begin{cases} Q/Q_h & \text{if } op_i \text{ is scheduled at } j \text{ by ant } h \\ 0 & \text{otherwise} \end{cases} \quad (12)$$

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.

In our experiments, we implemented both the basic ACO and the MMAS algorithms. The latter consistently achieves better scheduling results, especially for larger DFGs. A pseudo code implementation of the final version of our TCS algorithm

using MMAS is shown as Algorithm 1, where the pheromone bounding step is indicated as step 23.

```

procedure MaxMinAntSchedulingTCS( $G, R$ )
input: DFG  $G(V, E)$ , resource set  $R$ 
output: operation schedule
1: initialize parameter  $\rho, \tau_{ij}, p_{best}, \tau_{max}, \tau_{min}$ 
2: construct  $m$  ants
3:  $BestSolution \leftarrow \phi$ 
4: while ending condition is not met do
5:   for  $i = 0$  to  $m$  do
6:      $ant(i)$  constructs a valid schedule timing constrained
        $S_{current}$  as following:
7:      $S_{current} \leftarrow \phi$ 
8:     perform ASAP and ALAP
9:     while exists unscheduled operation do
10:      update time frame  $[t_i^S, t_i^L]$  associated with each
        operation  $op_i$  and the distribution graphs  $q_k$ .
11:      select one operation  $op_h$  among all unscheduled
        operations probabilistically
12:      for  $t_h^S \leq j \leq t_h^L$  do
13:        set local heuristic  $\eta_{hj} = 1/q_k(j)$  where  $op_h$  is
        of type  $k$ 
14:      end for
15:      select time step  $l$  using  $\eta$  and  $\tau$  as Equation (10).
16:       $S_{current} = schedule(S_{current}, op_h, l)$ 
17:      Update time frame and distribution graphs based
        on  $S_{current}$ 
18:    end while
19:    if  $S_{current}$  is better than that of  $BestSolution$  then
20:       $BestSolution \leftarrow S_{current}$ 
21:    end if
22:  end for
23:  update  $\tau_{max}$  and  $\tau_{min}$  based on Equation (3) and (4)
24:  update  $\eta$  if needed
25:  update  $\tau_{ij}$  based on Equation (11)
26: end while
27: return  $BestSolution$ 

```

Algorithm 1: MMAS for Timing Constrained Scheduling

C. Refinements

1) *Updating Neighboring Pheromone Trails:* We found that a "better" solution can often be achieved from a "good" scheduling result by simply adjusting very few operations' scheduled positions within their time frames. Based on this observation, we can refine our pheromone update policy to encourage exploration of the neighboring positions. More specifically, in the pheromone reinforcement step indicated by Equation 12, we also increase the pheromone trails of the control steps adjacent position j subject to a weighted function window. Two such windowing functions are shown in Figure 2. Depending on the neighbor's offset from j , the two functions adjust its pheromone trail in a similar manner to Equation 12 but with an extra factor applied. Assuming we use x to represent the offset, then Figure 2(a) has a weight function of $1 - 1/3|x|$ while Figure 2(b) provides a weight function of

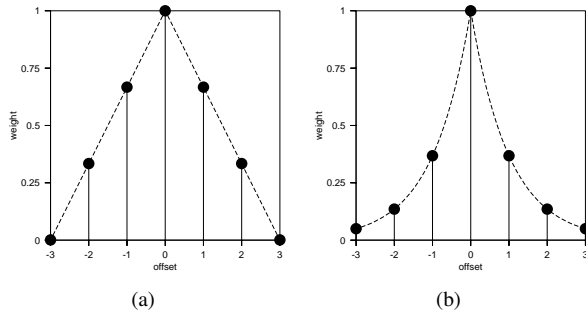


Fig. 2: Pheromone update windows

$e^{-|x|}$. In our experiments, the latter provides relatively better performance. Ideally, the weight function window size shall be computed based on the mobility ranges of the operations. However, to keep the algorithm simple, we use a window size 5 across all our experiments, subject to the operation's time frame $[t_i^S, t_i^L]$. This number is estimated using the average mobility ranges of all testing cases.

2) *Operation Selection*: In our algorithm, the ants construct a schedule for the given DFG by making two decisions in sequence. First, it needs to select the next operation. Then a specific control step is determined for the selected operation. As discussed earlier, the simplest approach for selecting an operation is to randomly pick one amongst all the unscheduled operations. Though it is simple and computationally effective, it does not appreciate the information accumulated in the pheromone from the previous iterations; it also ignores the dynamic time frame information. One possible refinement is to make the selection probability proportional to the pheromone and inversely proportional to the size of the operation's time frame at that instance. More precisely, we pick the next operation op_i probabilistically with the following equation:

$$p_i = \frac{\frac{\sum_j \tau_{ij}}{(t_i^L - t_i^S + 1)}}{\sum_l \frac{\sum_k \tau_{lk}}{(t_l^L - t_l^S + 1)}} \quad (13)$$

Here the numerator can be viewed as the average pheromone value over all possible positions in the current time frame for operation op_i . The denominator is a normalization factor to bring the result to be a valid probability value between 0 and 1. It is basically the addition of the average of pheromone for all the unscheduled operations op_l . Notice that as the time frames of the operations change dynamically depending on the partial schedule, the average pheromone trail is not constant during the schedule construction process. In other words, we only consider a pheromone τ_{ij} when $t_i^S \leq j \leq t_i^L$.

Intuitively, this formulation favors an operation with stronger pheromone and fewer possible scheduling alternatives. In the extreme case, $t_i^L = t_i^S$, which means operation op_i is on the critical path, we will have only one choice for op_i . If the pheromone for op_i at this position happens to be very strong, we will have better chance to pick op_i at the next step compared with other operations. Our experiments show that applying this operation selection policy makes the algorithm faster in identifying high quality results. Compared with the even possibility approach, there is an overhead to

perform this operation selection policy. However, by making the selection more targeted, it allows us to reduce the overall iteration number of the algorithm thus the additional overhead is well worth it. In our experiments, we were able to reduce the total runtime by about 23% while achieving almost the same quality with our testing results by adopting this biased selection policy.

D. Extensions

Our proposed TCS algorithm applies the Ant Colony meta-heuristic at the high level. It poses little difficulty to extend it to handle different scheduling contexts. Most of the methods proposed previously for FDS can be readily implemented within our framework.

1) *Resource Preference*: In our work, the target is to minimize the total count of resources needed. Accordingly, we use the inverse of this total count as the quality of the scheduling result. This quality measurement is further used to adjust the pheromone trails. However, in practice, we may have unbalanced hardware costs for different resource types. With this consideration, we might find that we prefer a schedule that requires 3 multipliers and 4 adders rather than one that needs 4 multipliers and 3 adders, even though both schedules have the same total number (7) of resources. This issue can be handled in our algorithm simply by introducing a cost factor c_k for each resource type and modifying the quality of the schedule to this weighted resource cost,

$$Q_h = \sum_k (c_k r_k) \quad (14)$$

By adjusting the c_k assigned to different resource types, we can control the preference in our schedule results.

2) *Multi-cycle Operation*: No change is needed for our algorithm to handle multi-cycle operation since it uses dynamically computed time frames. Also, as presented in Section IV-A, the distribution graph handles multi-cycle operations naturally.

3) *Mutually Exclusive Operations*: Mutually exclusive operations occur when operations are located in different branches of the program. This happens in *if-then-else* and *case* statements in high-level languages. With the proposed algorithm, we do not need to add any extra constraints for handling such operations; thus the approach proposed in [10] is still valid.

4) *Chained Operations*: When the total delay of consecutive operations is less than a clock cycle, it is possible to chain the operations during scheduling. The same techniques used in [10] can be directly applied within our approach, where chaining is handled by extending the ASAP and ALAP computation to obtain the time frames for the operations.

5) *Pipelining*: For pipelined resources, there exists additional parallelism provided by functional pipelining. Here optimizing an individual control step becomes inappropriate and limited. We have to consider scheduling optimization over groups of control steps. We can solve this by slicing and superimposing the distribution graph in a manner depending on the latency [10]. Again, this method can also be applied to extend our algorithm to handle the pipelined scenario.

E. Complexity Analysis

As we can see, the construction of individual schedule by the ants, or the body of the inner loop in the proposed algorithm, is of the complexity $O(n^2)$, where n is the number of nodes in the DFG under consideration. Thus the total complexity of the algorithm is determined by the number of ants m and the iteration number N . Theoretically, the production of m and N shall be proportional to the production of n and the deadline D . In this case, we have a total complexity of $O(Dn^3)$ which is the same as the un-optimized version of FDS. However, in practice, we found it is possible to fix m and N for a large range of applications (see Section VII). This means that in practical use the algorithm can be expected to work with $O(n^2)$ complexity for most of the cases.

V. MAX – MIN ANT SYSTEM FOR RESOURCE CONSTRAINED SCHEDULING

In this section, we present our algorithm of applying Ant System heuristic, or more specifically the MAX-MIN Ant System (MMAS) [6], for solving the operation scheduling problem under resource constraints.

A. Algorithm Formulation

As discussed in Section II, list scheduling is the most widely used method for resource constrained scheduling RCS problem. 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 operations. The list scheduler maintains a ready list, i.e. nodes whose predecessors have already been scheduled. In each iteration, the algorithm 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 repeated until all of the operations have been scheduled. When there exist more than one ready operations sharing the same priority, ties are broken randomly. The effectiveness of list scheduler heavily depends on the priority list. Although there exist many different heuristics on how to order the list, it is commonly believed that the best list depends on the structure of the input application. A priority list based on a single heuristic limits the exploration of the search space for the list scheduler.

Based on this observation, we address the RCS problem in a similar manner to the ACO meta heuristic framework used to solve the TCS problem. The key idea is to combine ACO meta-heuristic with the traditional list scheduling algorithm, and formulate the problem as an iterative searching process over the operation list space.

Similar to the algorithm formulated for the TCS problem, each operation, or DFG node op_i , is associated with a set of pheromone trails τ_{ij} . The difference is that now each trail indicates the global favorableness of assigning the i -th operation at the j -th position in the priority list, where $j = 1, \dots, n$. Since it is valid for the operation to be assigned to any of the position in the priority list, each pheromone trail will be valid. This is different from the timing-constrained formulation where some trails are fixed to be zero based on

the allowed time frames of the operations. Initially, τ_{ij} is set with some fixed value τ_0 .

A pseudo code implementation of our RCS algorithm using MMAS is shown as Algorithm 2, where the pheromone bounding step is indicated as step 12. For each iteration, m ants are released and each starts to construct an individual priority list by filling the list with one operation per step. Every ant will have memory about the operations it has already selected in order to guarantee the validity of the constructed list. Upon starting step j , the ant has already selected $j - 1$ operations of the DFG. To fill the j -th position of the list, the ant chooses the next operation op_i probabilistically according to:

$$p_{ij} = \begin{cases} \frac{\tau_{ij}(t)^\alpha \cdot \eta_{ij}^\beta}{\sum_k (\tau_{kj}(t)^\alpha \cdot \eta_{kj}^\beta)} & \text{if } op_k \text{ is not scheduled yet} \\ 0 & \text{otherwise} \end{cases} \quad (15)$$

where the eligible operations op_k are those yet to be scheduled. Again, η_{ik} is a local heuristic for selecting operation op_k , and α and β are parameters to control the relative influence of the distributed global heuristic τ_{ik} and local heuristic η_{ik} .

The local heuristic η gives the local favorableness of scheduling the i -th operation at the j -th position of the priority list. In our work, we experimented with different well-known heuristics [4] proposed for operation scheduling.

- 1) *Operation Mobility* (OM): The mobility of an operation gives the range for scheduling the operation. It is computed as the difference between ALAP and ASAP results. The smaller the mobility, the more urgent the scheduling of the operation is. When the mobility is zero, the operation is on the critical path.
- 2) *Operation Depth* (OD): Operation depth is the length of the longest path in the DFG from the operation to the sink. It is an obvious measure for the priority of an operation as it gives number of operations we must pass.
- 3) *Latency Weighted Operation Depth* (LWOD): LWOD is computed in a similar manner as OD, except that the nodes along the path are weighted using their operation latencies.
- 4) *Successor Number* (SN): The motivation of using the number of successors is the hope that scheduling a node with more successors has a higher possibility of making other nodes in the DFG free, thus increasing the number of possible operations to choose from later on.

The second stage of the algorithm, i.e. the result quality assessment and pheromone trail updating, proceeds similarly as the timing constrained algorithm discussed previously. The only exception is that now the quality Q_h in Equation 12 is replaced by the total latency L_h of the generated scheduling result.

B. Refinements

1) *Dynamic Local Heuristics*: One important difference between our algorithm and other Ant System algorithms is that we use a dynamic local heuristic in the resource constrained scheduling process. It is indicated by step 13 in Algorithm 2. This technique allows better local guidance to the ants for

```

procedure MaxMinAntSchedulingRCS( $G, R$ )
input: DFG  $G(V, E)$ , resource set  $R$ 
output: operation schedule
1: initialize parameter  $\rho, \tau_{ij}, p_{best}, \tau_{max}, \tau_{min}$ 
2: construct  $m$  ants
3:  $BestSolution \leftarrow \phi$ 
4: while ending condition is not met do
5:   for  $i = 0$  to  $m$  do
6:      $ant(i)$  constructs a list  $L(i)$  of nodes using  $\tau$  and  $\eta$ 
7:      $Q_i = \text{ListScheduling}(G, R, L(i))$ 
8:     if  $Q_i$  is better than that of  $BestSolution$  then
9:        $BestSolution \leftarrow L(i)$ 
10:    end if
11:  end for
12:  update  $\tau_{max}$  and  $\tau_{min}$  based on Equation (3) and (4)
13:  update  $\eta$  if needed
14:  update  $\tau_{ij}$  based on (11)
15: end while
16: return  $BestSolution$ 

```

Algorithm 2: MMAS for Resource-Constrained Scheduling

making the selection in the next iteration. We will illustrate this feature with the use of the operation mobility heuristic.

Typically, the mobility of an operation is computed by using ALAP and ASAP results. One important input parameter in computing the ALAP result is the estimated scheduling deadline. This deadline is usually obtained from system specifications or other quick heuristic methods such as a list scheduler. It is clear that more accurate deadline estimation will yield tighter mobility range thus better local guidance.

Based on the above observation, we use dynamically computed mobility as the local heuristic in our algorithm. As the algorithm proceeds, whenever a better schedule is achieved, we use the newly obtained scheduling length as the deadline for computing the ALAP result for the next iteration. That is, for iteration t , the local heuristic for operation i is computed as (see section III-B for definitions for f and S^{gb}):

$$\eta_i(t) = \frac{1}{ALAP(f(S^{gb}(t-1)), i) - ASAP(i) + 1} \quad (16)$$

2) *Topologically Sorted Lists:* In the above algorithm, the ants construct a priority list using the same traversing method that is used in the TSP formulation [24]. In fact, this turns out to be a naïve way. To illustrate this, one just need to notice that it will yield a search space of totally $n!$ possible lists, which is simply all the permutations of n operations. However, we know that the resultant schedules of the list scheduler are only a small portion of these lists. More precisely, they are all the possible permutations of the operations that are topologically sorted based on the dependency constraints imposed by the DFG. By leveraging this application dependent feature, it is possible for us to greatly reduce the search space. For instance, using this technique on a simple 11 node example [4] reduces the possible number of orderings from $11!$ to 59400, or 0.15%. Though it quickly becomes prohibitive to

precisely compute such reduction for more complex graphs¹, it is generally significant. By adopting this technique, in the final version of our algorithm, the ant traverses the DFG in a similar manner to the list scheduling process and fills operation list one by one. At each step, the ant will select an operation based on Equation 15 but only from all the ready operations, that is, from all the operations whose predecessors have all been scheduled.

C. Extensions

So far, our discussion on the operation scheduling problems has been limited to the the *homogeneous* case. In other words, each operation is mapped to a unique resource type, though a resource type might be able to handle different operations. In practice, this means that a *resource allocation* step needs to precede the operation scheduling process. We often need to handle the *heterogeneous* case, where one operation can be executed with different resource types. For example, a system might have two different realizations of multiplier, one is faster but more expensive while the other is slower but cheaper. Both are capable of executing a multiplication operation. Our challenge is to determine how to effectively use the resources to achieve the best time performance. In this situation, separating the resource allocation step from operation scheduling may be not a favorable approach, as the prior step could greatly limit the optimization opportunity for operation scheduling. This motivates us to consider the resource allocation issue within the operation scheduling problem.

It is possible to address this problem using ILP by extending the ILP formulation for the homogenous case. The basic idea is to introduce a new set of parameters m_{ik} which can take value 0 or 1, and describe the compatibility between operation op_i and resource type k . A set of new constraints are needed to make sure that only one type of resources among all those that are capable of processing op_i is used, i.e.

$$\sum_k m_{ik} = 1 \quad \text{where } i = 1, \dots, n \quad (17)$$

We can see it makes the ILP problem even more intractable.

However, this extra difficulty does not block the list scheduler or the proposed MMAS approach from working. The basic algorithm could be carried out with almost no changes except for the list construction. The major problem is, when there exists alternative resource types for one specific operation, estimating a certain attribute of the operation becomes more challenging. For example, with different execution delay on capable resource types, the mobility of the operation is variable. This has been studied in previous research, e.g. [7], where the average latency over a set of heterogenous resources is used to carry scheduling task. In our work, we simply take the pessimistic approach by applying the longest execution latency amongst the alternative resources in computing such attributes. With this extension, our algorithm can be applied to heterogenous cases.

¹We tried to compute the search space reduction for Figure 5 using GenLE [39]. It failed to produce any result within 100 computer hours.

D. Complexity Analysis

List scheduling is a two step process. In the first step, a priority list is built. The second step takes n steps to solve the scheduling problem since it is a constructive method without backtracking. For different heuristics, the complexity of the first step is different. When operation mobility, operation depth and latency weight operation depth are used, it takes $O(n^2)$ steps to build the priority list since a depth-first or breadth-first graph transverse is involved. When the successor node number is adopted as the list construction heuristic, it only takes n step. Thus the complexities for these methods are $O(n^2)$ or $O(n)$ respectively.

The force-directed resource constrained scheduling method is different. Though it is also a constructive method without backtracking, we need to compute the force of each operation at every step since the total latency is dynamically increased based on whether there is enough resources to handle the ready operations. Thus the FDS method has $O(n^3)$ complexity.

The complexity of the proposed MMAS solution is determined mainly by the complexity of constructing individual scheduling solutions, the number of ants m and the total iteration N in every run. In order to generate a schedule solution, each ant needs to first loop through n operations and for each operation determine its location, which has a complexity of $O(n)$. This list is then provided to a list scheduler with a complexity of $O(n)$ or $O(n^2)$. This makes overall complexity $O(n^2)$. Obviously, if mN is proportional to n , we will have one order higher complexity than the corresponding list scheduling approach. However, based on our experience, it is possible to fix such factor for a large set of practical cases so that the complexity of the MMAS solution is the same as the list scheduling approach.

VI. BENCHMARKS

In order to test and evaluate our algorithms, we have constructed a comprehensive set of benchmarks. These benchmarks are taken from one of two sources:

- Popular benchmarks used in previous literature;
- Real-life examples generated and selected from the MediaBench suite [40].

The benefit of having classic samples is that they provide a direct comparison between results generated by our algorithm and that 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 operation scheduling studies are included. These samples focus mainly on frequently used numeric calculations performed by different applications. They are:

- 1) ARF: an implementation of an *Auto Regression Filter*.
- 2) EWF: an implementation of an *Elliptic Wave Filter*.
- 3) FIR1 and FIR2: two versions of a *Finite Impulse Response Filter*.
- 4) COSINE1 and COSINE2: two implementations for a one dimensional 8-point fast discrete cosine transform, where COSINE1 assumes constant coefficients while the coefficients in COSINE2 are given as inputs.

- 5) HAL: an iterative solution of a second order differential equation. This perhaps is the most popularly used example in text books which originally appeared in [10].

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 operation 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 have 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 [41] and Machine SUIF [42] tools, and over 14,000 DFGs were extracted as preliminary candidates for our benchmark set. After careful study, thirteen DFG samples were selected from four MediaBench applications. These applications are:

JPEG JPEG is a lossy compression technique for digital images. The *cjpeg* application performs compression, while the *djpeg* application decompresses the JPEG image.

MPEG2 MPEG2 is a digital video compression standard, commonly used for high quality video compression including DVD compression. The *mpeg2enc* application encodes the video, while the *mpeg2dec* application decodes the video.

EPIC EPIC stands for Efficient Pyramid Image Coder and is another image compression utility.

MESA The Mesa project is a software 3-D graphics package. The primary application that we were concerned with was the *texgen* utility, which generates a texture mapped version of the Utah teapot.

From the JPEG project, four basic blocks were selected. The first came from the *write_bmp_header* function. The basic block was selected for its high level of parallelism. The second basic block came from the *h2v2_smooth_downsample* function. This function has 51 nodes for only one store operation at the end. The store is dependent on all but two of the operations, making it an interesting problem for scheduling. The third basic block was selected from the *jpeg_fdct_islow* function. The function performs an integer forward discrete cosine transform (DCT) using a slow-but-accurate algorithm and was chosen for its popularity amongst DSP applications. The final block was selected from the *jpeg_idct_ifast* function. Like the forward DCT, this was selected for its commonality. However, this implementation is a fast, and much less accurate, version of the inverse DCT.

Two basic blocks were selected from the MPEG section. The first came from the *idctcol* function in the *mpeg2dec* application. The function implements another version of the

inverse DCT algorithm. In this case, the function is part of a 2-D inverse DCT, while the inverse DCT from the JPEG application is only 1-D. The large size of the DFG and complicated dependency structure provide a good test for scheduling algorithm. The second comes from the *motion_vectors* function in the *mpeg2enc* function. The basic block only contains 42 nodes and 38 edges, making it one of the smaller blocks selected from MediaBench, ensuring that the benchmark suite provides a wide range of synthesis problems to test scalability.

The EPIC project supplied one basic block. It came from the *collapse_pyr* function, which is a quadrature mirror filter bank. The block was selected for its medium size and common use in DSP applications.

From the MESA application, six basic blocks were selected to be added to the benchmark suite. The *invert_matrix_general* and *matmul* functions were selected because they are general functions, not specific to the MESA application. Matrix operations, such as inversion and multiplication, are common in DSP applications where many filters are merely matrix multiplications with a set of coefficients. The next block selected came from the *smooth_color_z_triangle* function. The basic block is essentially four parallel computations without data dependencies, making it an ideal addition to the benchmark suite. The fourth benchmark is from the *horner_bezier* method. With only 18 nodes, the small size helps add variety to the benchmarks. The fifth block comes from the *interpolate_aux* function. The function performs four linear interpolation calculations, which can easily be run in parallel if the hardware is available. The final benchmark is from the *feedback_points* function, which calculates texture coordinates for a feedback buffer.

Benchmark Name	# Nodes	# Edges	OD
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

TABLE I: Benchmark node and edge count with the operation depth (OD) assuming unit delay.

Table I lists all twenty 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 operation depth (assuming unit delay for every operation) of the DFG. The data, including related statistics, DFG graphs and source code for the all

testing benchmarks, is available online [43].

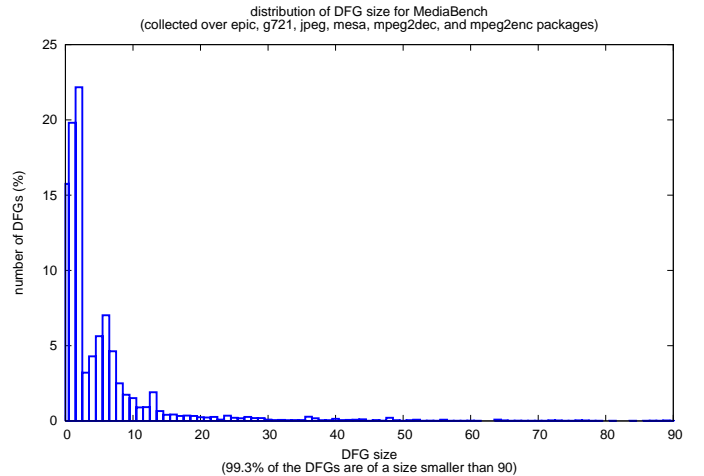


Fig. 3: Distribution of DFG size for MediaBench

In order to justify the difficulty and representativeness of our testing cases, we analyze the distribution of the sizes of DFGs in practical software programs. Our analysis covers the *epic*, *jpeg*, *g721*, *mpeg2enc*, *mpeg2dec*, and *mesa* packages. The result is shown in Figure 3. We find that the maximum size of a DFG can be as big as 632. However, the majority of the DFGs are much smaller. In fact, more than 99.3% DFGs have fewer than 90 nodes. Moreover, the very largest ones are of little interest with respect to system performance. They are typically related with system initialization and are executed only once.

VII. EXPERIMENTAL RESULTS

A. Time Constrained Scheduling

In order to evaluate the quality of our proposed algorithm for timing constrained scheduling problem, we compare its results with that obtained by the widely used force-directed scheduling method. 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.

Since there is no widely distributed and recognized FDS implementation, we implemented our own. The implementation is based on [10] and has all the applicable refinements proposed in the paper, including multi-cycle operation support, resource preference control, and look-ahead using second order of displacement in force computation. Actually, based on our experience, the look-ahead function for FDS is very critical. Without invoking this mechanism, basic FDS provides poor scheduling results even for small sized examples. In Table II, we show the effect of look-ahead for the HAL

benchmark originally presented in [10], which has only 11 operations and 8 data dependencies. Because of this, in our experiments, the look-ahead function is always used to allow FDS to provide better results.

deadline	w/t look-ahead	w/ look-ahead
9	(2 1)	(2 1)
10	(2 1)	(2 1)
11	(2 1)	(2 1)
12	(3 1)	(2 1)
13	(3 1)	(1 1)
14	(3 1)	(1 1)

TABLE II: Effect of Look-ahead Mechanism in FDS (Result shown in MUL/ALU number pair. Deadline is in cycles.)

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 2 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 proposed MMAS algorithm is executed 5 times to obtain enough data for performance evaluation. We report the FDS result quality, the average and best result quality for the proposed algorithm and the standard deviation for these results. The execution time information for both algorithms is also reported.

We have implemented our MMAS formulation in C for the TCS problem, with the refinements discussed in Section IV. 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.

Due to the large amount of data, we won't be able to report testing results for all 263 cases in details. Table III compares the testing results for *idctcol* and *invert_matrix_general*, two of the biggest samples. In this table, we provide a side by side comparison between FDS and our proposed method. The scheduling results are reported as MUL/ALU number pair required by the obtained scheduling. For MMAS method, we report both the average performance and the best performance in the 5 runs for each testing case, together with the saving percentage. The saving is measured by the reduction of computing resources. In order to keep the evaluation general and objective, we use the total count of resources as the quality metrics without considering their individual cost factors.

Besides absolute quality of the results, one difference between FDS and the proposed method is that our 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* in Table III 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, our proposed algorithm robustly generates monotonically non-increasing results with fewer resource requirements as the deadline increases.

Table IV summarizes the testing results for all of the benchmarks. We present the average and the best results for each testing benchmark, its tested deadline range, and the average standard deviations. The table is arranged in the increasing order of the complexity of the DFGs. The average result quality generated by our 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) our MMAS method outperforms the FDS method. There are only five cases where our approach has worse average quality results. They all happened on the *invert_matrix_general* benchmark and are listed in Table III, indicated by lines with the italic bold fonts. On average, as shown in Table IV, we can expect a 16.4% performance improvement over FDS. If only considering the best results among the 5 runs for each testing case, we achieve a 19.5% resource reduction averaged over all tested samples. The most outstanding results provided by our proposed method achieve a 75% resource reduction compared with FDS. These results are obtained on a few deadlines for the *jpeg_idct_ifast* benchmark.

From Table IV, it is easy to see that for all the examples, MMAS 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 predefined deadline. Based on the results presented in Table III and Table IV, the proposed algorithm also demonstrates better scalability over this parameter.

All of the experimental results are obtained on a Linux box with a 2GHz CPU. Figure 4 diagrams the execution time comparison between the presented algorithm and FDS. Curve A and B shows the run time for FDS and the proposed method (respectively), where we use the average runtime for our MMAS solutions over 5 runs. As discussed before, since we use fixed ant number m and iteration limit N in our experiments to make the algorithm simpler, there exists a big gap between the execution times for the smaller sized cases. For example, for the HAL example, which only has 11 operations, the execution time of FDS is 0.014 seconds while our method takes 0.66 seconds. This translates into a ratio of 47. However, as the size of the problem gets bigger, this ratio drops quickly. For the biggest cases *invert_matrix_genera*, FDS takes 270.6 seconds while our method spends about 411.7 seconds, which makes the ratio 1.5. To summarize, for smaller cases, our algorithm does have relatively larger execution times but the absolute run time is still very short. For the HAL

Name (size)	Deadline	FDS	Average	Savings	Best	Savings	σ
idctcol (114 164)	19	(6 8)	(5.0 6.0)	21.43%	(5 6)	21.43%	0.000
	20	(5 7)	(4.4 6.0)	13.33%	(4 6)	16.67%	0.219
	21	(4 7)	(4.2 5.8)	9.09%	(4 6)	9.09%	0.000
	22	(4 7)	(4.2 5.4)	12.73%	(4 5)	18.18%	0.219
	23	(4 7)	(4.0 5.4)	14.55%	(4 5)	18.18%	0.219
	24	(4 7)	(3.6 5.2)	20.00%	(3 5)	27.27%	0.335
	25	(8 8)	(3.8 5.0)	45.00%	(3 5)	50.00%	0.179
	26	(8 8)	(3.4 5.0)	47.50%	(3 5)	50.00%	0.219
	27	(8 8)	(3.0 5.0)	50.00%	(3 5)	50.00%	0.00
	28	(8 8)	(3.0 4.6)	52.50%	(3 4)	56.25%	0.219
	29	(8 8)	(3.0 4.4)	53.75%	(3 4)	56.25%	0.219
	30	(8 8)	(3.0 4.6)	52.50%	(3 4)	56.25%	0.219
	31	(4 6)	(3.0 4.6)	24.00%	(3 4)	30.00%	0.219
	32	(4 5)	(3.0 4.0)	22.22%	(3 4)	22.22%	0.000
	33	(4 5)	(2.8 4.0)	24.44%	(2 4)	33.33%	0.179
	34	(4 5)	(3.0 4.0)	22.22%	(3 4)	22.22%	0.000
	35	(4 5)	(3.0 4.0)	22.22%	(3 4)	22.22%	0.000
	36	(4 6)	(3.0 3.8)	32.00%	(3 3)	40.00%	0.179
37	(4 6)	(2.6 3.8)	36.00%	(3 3)	40.00%	0.219	
38	(4 6)	(2.8 3.4)	38.00%	(3 3)	40.00%	0.179	
invert_matrix_general (333 354)	15	(24 23)	(26.0 22.0)	-2.13%	(25 22)	0.00%	0.283
	16	(22 19)	(23.8 19.0)	-4.39%	(23 19)	-2.44%	0.179
	17	(19 17)	(21.8 17.4)	-8.89%	(21 17)	-5.56%	0.335
	18	(18 16)	(20.4 16.2)	-7.65%	(20 16)	-5.88%	0.219
	19	(17 16)	(19.2 16.0)	-6.67%	(19 15)	-3.03%	0.335
	20	(17 16)	(18.2 13.4)	4.24%	(18 13)	6.06%	0.358
	21	(16 16)	(17.2 12.8)	6.25%	(17 13)	6.25%	0.000
	22	(16 16)	(16.4 12.2)	10.63%	(16 12)	12.50%	0.358
	23	(16 16)	(16.0 11.8)	13.12%	(16 11)	15.62%	0.179
	24	(16 16)	(15.4 11.2)	16.87%	(15 11)	18.75%	0.219
	25	(16 16)	(14.4 10.8)	21.25%	(14 11)	21.88%	0.179
	26	(16 16)	(14.2 10.2)	23.75%	(13 10)	28.12%	0.358
	27	(16 16)	(13.8 10.0)	25.62%	(13 10)	28.12%	0.179
28	(16 16)	(13.4 10.2)	26.25%	(13 10)	28.12%	0.219	
29	(16 16)	(13.0 9.4)	30.00%	(13 9)	31.25%	0.219	
30	(16 16)	(12.6 9.6)	30.63%	(13 9)	31.25%	0.179	

TABLE III: Partial detailed results for Timing-Constrained Scheduling

(Size is given as DFG's node/edge number pair. Virtual nodes and edges are not counted.

Average and standard deviation σ are computed over 5 runs. Saving is computed based on FDS results. No weight applied.)

Name	Size	Deadline	Avg. Savings (SA)	Best Savings (SA)	Avg. σ (SA)
HAL	11/8	(6 - 12)	7.1% (7.1%)	7.1% (7.1%)	0.000 (0.000)
horner_bezier_surf	18/16	(11 - 22)	9.9% (-4.6%)	13.2% (2.1%)	0.015 (0.051)
ARF	28/30	(11 - 22)	12.4% (-1.2%)	16.9% (3.1%)	0.093 (0.099)
motion_vectors	32/29	(7 - 14)	13.1% (-3.4%)	16.0% (2.8%)	0.072 (0.177)
EWF	34/47	(17 - 34)	11.5% (-4.4%)	18.1% (4.7%)	0.081 (0.136)
FIR2	40/39	(12 - 24)	16.8% (-15.7%)	22.8% (-1.9%)	0.106 (0.299)
FIR1	44/43	(12 - 24)	15.2% (-7.7%)	18.0% (-3.3%)	0.047 (0.116)
h2v2_smooth_downsample	51/52	(17 - 34)	19.3% (7.6%)	21.3% (11.0%)	0.042 (0.088)
feedback_points	53/50	(11 - 22)	5.9% (-12.8%)	9.2% (-6.4%)	0.103 (0.196)
collapse_pyr	56/73	(8 - 16)	18.3% (4.6%)	18.9% (9.6%)	0.044 (0.195)
COSINE1	66/76	(10 - 20)	21.5% (7.4%)	25.9% (14.1%)	0.150 (0.349)
COSINE2	82/91	(10 - 20)	5.6% (-14.8%)	12.0% (-7.3%)	0.232 (0.342)
write_bmp_header	106/88	(8 - 16)	0.9% (-5.3%)	1.0% (-3.4%)	0.064 (0.093)
interpolate_aux	108/104	(10 - 20)	0.2% (-36.5%)	2.0% (-27.9%)	0.109 (0.407)
matmul	109/116	(11 - 22)	3.7% (-30.8%)	5.1% (-21.4%)	0.088 (0.363)
idctcol	114/164	(19 - 38)	30.7% (12.6%)	34.0% (17.5%)	0.151 (0.231)
jpeg_idct_ifast	122/162	(17 - 34)	50.3% (36.9%)	52.1% (41.8%)	0.147 (0.336)
jpeg_fdct_islow	134/169	(16 - 32)	31.4% (7.5%)	34.2% (13.0%)	0.171 (0.335)
smooth_color_z_triangle	197/196	(15 - 30)	7.3% (-18.7%)	9.2% (-12.0%)	0.136 (0.472)
invert_matrix_general	333/354	(15 - 30)	11.2% (-29.4%)	13.2% (-22.9%)	0.237 (0.743)
Total Avg.			16.4% (-5.1%)	19.5% (1.0%)	0.104 (0.251)

TABLE IV: Result Summary for Timing-Constrained Scheduling

Data in parenthesis shows the results obtained using Simulated Annealing.

Deadline shows the tested range. Average σ is computed over the tested range.

Saving is computed based on FDS results. No weight applied.

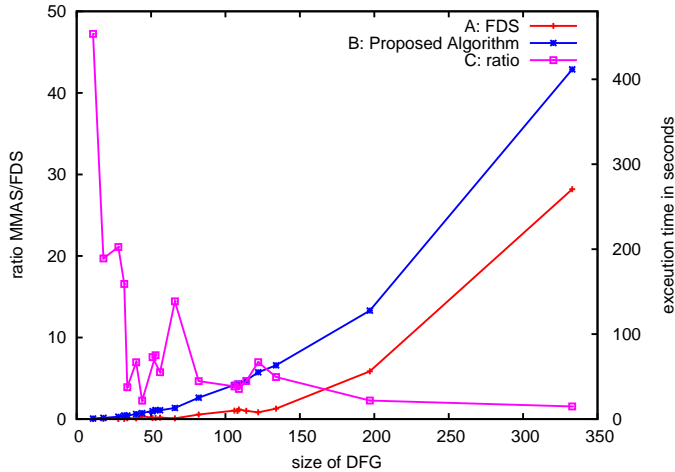


Fig. 4: Execution Time for Timing-Constrained Scheduling. (Ratio is MMAS time / FDS time)

example, it only takes a fraction of a second. For bigger cases, the proposed method has a runtime at the same scale as FDS. This makes our algorithm practical.

In Figure 4, we do see some spikes in the ratio curve. We contribute this to two main reasons. First, the recorded execution time is based on system time and it is relatively more unreliable when the execution time is small. Secondly but perhaps more important, the timing performance of both algorithms is not only determined by the DFG node count but also dependent on the predefined dependencies in the DFGs and the deadline D . This will introduce variance when the curves are drawn against the node count.

B. Resource Constrained Scheduling

We have implemented the proposed MMAS-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 proposed algorithm with different choices of local heuristics. For each choice, we also perform 5 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).

Table V shows the testing results for the homogenous case. The best results for each case are shown in bold. Compared with a variety of list scheduling approaches and the force-directed scheduling method, the proposed 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 (10 times) for best results, which is still less than the worst case of MMAS (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.

Besides the absolute schedule latency, another important aspect of the quality of a scheduling algorithm is its stability over different input applications. As indicated in Section II, the performance of traditional list scheduler heavily depends on the input application. This is echoed by the data in Table V. Meantime, it is easy to observe that the proposed 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. Based on the data shown in Table V, the average standard deviation for list scheduling over all the benchmarks and different heuristic choices is 1.2, while for the MMAS 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 proposed 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 proposed 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 proposed 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

Name	Size	Resources	FDS	List Scheduling				MMAS(average over 5 runs)				SA (avg. 10 runs)
				OM	OD	LWOD	SN	OM	OD	LWOD	SN	
HAL	(8/11)	(2 1)	8	10	8	8	8	8.0	8.0	8.0	8.0	8.0
homer_bezier_surf	(16/18)	(2 1)	12	16	12	13	13	12.0	12.0	12.0	12.0	12.4
ARF	(30/28)	(3 1)	18	19	16	18	18	16.0	16.0	16.0	16.0	17.2
motion_vectors	(29/32)	(3 4)	12	15	12	12	14	12.0	12.0	12.0	12.0	13.3
EWF	(47/34)	(1 2)	21	22	21	21	22	21.0	21.0	21.0	21.0	21.3
FIR2	(39/40)	(2 3)	17	19	18	17	15	17.0	16.8	17.0	17.0	18.5
FIR1	(43/44)	(2 3)	16	22	22	21	16	16.0	16.0	16.0	16.0	21.1
h2v2_smooth_downsample	(52/51)	(1 3)	23	28	23	23	22	22.4	22.8	22.8	22.8	23.6
feedback_points	(50/53)	(3 3)	16	20	14	19	14	14.4	14.2	14.6	14.6	16.6
collapse_pyr	(73/56)	(3 5)	11	12	11	11	11	11.0	11.0	11.0	11.0	11.3
COSINE1	(76/66)	(4 5)	16	18	16	17	16	14.0	14.0	14.0	14.0	15.2
COSINE2	(91/82)	(5 8)	14	18	14	17	13	12.4	12.4	12.6	12.8	14.9
write_bmp_header	(88/106)	(1 9)	12	17	12	12	12	12.8	12.6	12.8	12.4	13.4
interpolate_aux	(104/108)	(9 8)	13	16	12	16	16	11.0	11.8	11.0	11.8	15.6
matmul	(116/109)	(9 8)	15	14	13	14	14	13.6	13.8	13.8	13.8	14.7
idctcol	(164/114)	(5 6)	21	26	21	21	21	20.6	19.8	20.2	20.0	24.3
jpeg_idct_ifast	(162/122)	(10 9)	19	21	20	19	19	19.0	19.0	19.0	19.0	20.8
jpeg_fdct_islow	(169/134)	(5 7)	21	28	22	22	21	22.0	22.0	21.8	21.8	23.8
smooth_color_z_triangle	(196/197)	(8 9)	24	25	25	23	24	24.0	24.0	24.0	24.0	25.5
invert_matrix_general	(354/333)	(15 11)	26	28	28	25	25	24.0	24.2	24.2	24.2	27.1

TABLE V: Result Summary for Homogenous Resource-Constrained Scheduling

(Heuristic Labels: OM=Operation Mobility OD=Operation Depth, LWOD=Latency Weighted Operation Depth, SN=Successor Number)

Benchmark (nodes/edges)	Resources	CPLEX (latency/runtime)	Force Directed	List Scheduling				MMAS(average over 5 runs)			
				OM	OD	LWOD	SN	OM	OD	LWOD	SN
HAL(21/25)	1a, 1fm, 1m, 3i, 3o	8 / 32	8	8	8	9	8	8	8	8	8
ARF(28/30)	2a, 1fm, 2m	11 / 22	11	11	13	13	13	11	11	11	11
EWF(34/47)	1a, 1fm, 1m	27 / 24000	28	28	31	31	28	27.2	27.2	27	27.2
FIR1(40/39)	2a, 2m, 3i, 3o	13 / 232	19	19	19	19	18	17.2	17.2	17	17.8
FIR2(44/43)	1a, 1fm, 1m, 3i, 3o	14 / 11560	19	19	21	21	21	16.2	16.4	16.2	17
COSINE1(66/76)	2a,2m, 1fm, 3i, 3o	†	18	19	20	18	18	17.4	18.2	17.6	17.6
COSINE2(82/91)	2a,2m, 1fm, 3i, 3o	†	23	23	23	23	23	21.2	21.2	21.2	21.2

TABLE VI: Result Summary for Heterogenous Resource-Constrained Scheduling

Schedule latency is in cycles; Runtime is in seconds; † indicates CPLEX failed to provide final result before running out of memory.

(Resource Labels: a=alu, fm=faster multiplier, m=multiplier, i=input, o=output)

(Heuristic Labels: OM=Operation Mobility OD=Operation Depth, LWOD=Latency Weighted Operation Depth, SN=Successor Number)

be performed by either a faster multiplier or a regular one. Furthermore, multiple same type units are also allowed. For example, we may have 3 faster multipliers and 2 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.

Table VI summarizes our heterogenous experiment results. Here an extended HAL benchmark is used which includes extra memory access operations. Compared with a variety of list scheduling approaches and the force-directed scheduling method, the proposed 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. In Table VI, the average standard

deviation for list scheduler over all the benchmarks and different heuristic choices is 0.8128, while that for the MMAS 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 proposed 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 hours of execution on a SPARC workstation with a 440MHz CPU and 384MByte 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 our proposed algorithm.

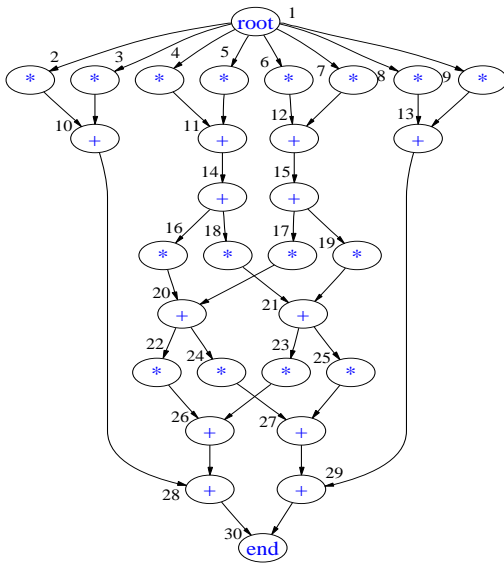


Fig. 5: Data Flow Graph of AR Filter.

(The number by the node is the index assigned for the operation.)

Experiment results of our algorithm are obtained on a Linux box with a 2GHz CPU, as well as those for list scheduling and the force-directed scheduling. For all the benchmarks, the runtime of the proposed algorithm ranges from 0.1 to 1.76 seconds. List scheduling is always the fastest due to its one-pass nature. It typically finishes within a small fraction of a second. The force-directed scheduler runs much slower than the list scheduler because its complexity is cubic in the number of operations. For small testing cases, it is typically faster than our algorithm as we set a fixed iteration number for the ants to explore the search space. However, as the problem size grows, the force-directed scheduler has longer runtime than our algorithm. In fact, for COSINE1 and COSINE2, the force-directed approach takes 12.7% and 21.2% more execution time respectively.

The evolutionary effect on the global heuristics τ_{ij} is illustrated in Figure 6. It plots the pheromone values for the ARF testing sample after 100 iterations of the proposed algorithm. The x-axis is the index of operation node in the DFG (shown in Figure 5), and the y-axis is the order index in the priority list passed to the list scheduler. There exist totally 30 nodes with node 1 and node 30 as the dummy source and sink of the DFG. Each dot in the diagram indicates the strength of the resultant pheromone trails for assigning corresponding order to a certain operation – the bigger the size of the dot, the stronger the value of the pheromone.

It is clearly seen from Figure 6 that there are a few strong pheromone trails while the remaining pheromone trails are very weak. This might be explained by the strong symmetric structure of the ARF DFG and the special implementation in our algorithm of considering operation list only with topologically sorted order. It is also interesting to notice that though a good amount of operations have a limited few alternative

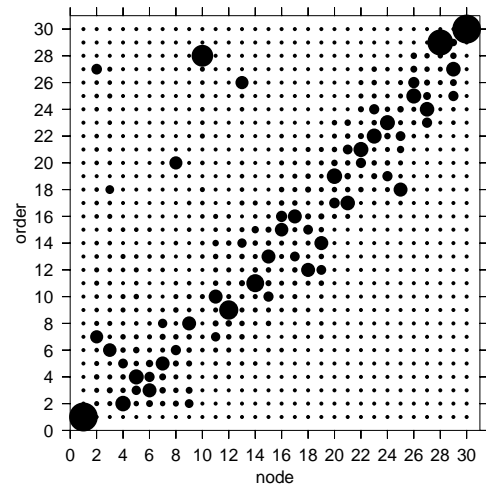


Fig. 6: Pheromone Heuristic Distribution for ARF

“good” positions (such as operation 6 and 26), for some of the operations the pheromone heuristics are strong enough to lock their positions. For example, according to its pheromone distribution, operation 10 shall be placed as the 28-th item in the list and there is no other competitive position for its placement. After careful evaluation, this ordering preference cannot be trivially obtained by constructing priority lists with any of the popularly used heuristics. This shows that the proposed algorithm has the possibility to discover better orderings which may be hard to achieve intuitively.

C. Comparison with Simulated Annealing

In order to further investigate the quality of the proposed algorithms, we compared them with a simulated annealing (SA) approach. For resource constrained scheduling, we implemented the algorithm presented in [21]. The basic idea is very similar to what we proposed in our MMAS 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.

However, it is more difficult for the timing constrained scheduling problem since we have not found any SA-based approach in previously published works. Therefore, we formulated one ourselves. Consequently, we will give more emphasis on our SA based formulation for the timing constrained scheduling problem in the rest of this section.

A pseudo implementation of SA-based TCS algorithm is given as Algorithm 3. 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 [44]. The acceptance of a more costly neighboring solution is determined by applying the Boltzmann probability criteria [45], which depends on the cost difference and the annealing temperature. In our experiments, the most commonly known and used geometric cooling schedule [44]

procedure SA-TCS(G, R)

input: DFG $G(V, E)$, resource set R , and a map of operation to one resource in R

output: operation schedule

- 1: perform ASAP and ALAP on the DFG to obtain mobility ranges.
- 2: randomly initialize a valid seed scheduling $S_{current}$
- 3: set starting and ending temperature T_s and T_e .
- 4: set local search weight to θ .
- 5: set N to be the number of operations.
- 6: set t to T_s
- 7: set S_{best} to be $S_{current}$
- 8: **while** $t > T_e$ **do**
- 9: **for** $i = 0; i < \theta N; i++$ **do**
- 10: randomly generate a neighbor solution S_n
- 11: **if** S_n is invalid **then**
- 12: continue
- 13: **else**
- 14: compute the resource cost of S_n
- 15: randomly accept S_n to be $S_{current}$
- 16: update S_{best} if needed
- 17: **end if**
- 18: **end for**
- 19: update t based on cooling scheme
- 20: **end while**
- 21: return S_{best} and the resource cost

Algorithm 3: Simulated Annealing for Timing-Constrained Scheduling

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.

The experimental results for TCS problem obtained using the above simulated annealing formulation are shown in Table IV, where the SA results are provided in parenthesis column by column with those achieved by using MMAS. Similar to the MMAS algorithm, we perform 5 runs for each benchmark sample and report the average savings, the best savings, and the standard deviation of the reported scheduling results. It can be seen from Table IV that the SA method provides much worse results compared with the proposed MMAS solutions. In fact, the MMAS 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 MMAS 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 proposed MMAS method. A typical experiment over all 263 testing cases will

run between 9 to 12 hours which is 3 to 4 times longer than the MMAS-based TCS algorithm.

As discussed above, our SA formulation for resource constrained scheduling is similar to that studied in [21]. 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 MMAS 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 5 times of the MMAS execution time, there was no significant improvement on the results as the execution time increased. In the rightmost column of Table V, we present the typical RCS results using SA achieved with 10 times the MMAS execution time. The performance data is averaged over 10 runs for each testing sample. It is easy to see that the MMAS-based algorithm consistently outperforms it while using much less computing time.

D. Parameter Sensitivity

The proposed ACO-based algorithms belong to the category of stochastic search algorithms. This implies a certain sensitivity of the result to the choices of parameters which are at times difficult to determine. In order to better understand this issue and its relationship with the algorithms' performance, a study on their sensitivity to the parameter selection is in order. We have conducted extensive experiments in our work on this topic and will report our major findings in this section.

- α , β and Q : Variation on global heuristic weight α , local heuristic weight β and the pheromone delivery constant Q does not have noticeable impact on the performance of our algorithms. The algorithms consistently provide robust results when α and β are in the range of $[1, 100]$ and Q is between $[1, 5000]$ with small step size, while performance on benchmarks of smaller sizes tend to have more fluctuations than the bigger ones. Of course, a numerically precise limit should be a concern for parameter α and β in algorithm realization because they are used in power functions. Also, the scaling of local and global heuristics could be an issue with these parameters. In our study, we found setting $\alpha = \beta = 1$ worked well in our implementation over a comprehensive set of testing benchmarks. Moreover, the benefit is that it essentially eliminates the power function calls in Equation (1) which further reduces computing time.
- ρ : The pheromone evaporation factor ρ takes a value in the range of $[0, 1]$ and controls how much the existing pheromone trails will be reduced before any enhancement. The smaller this number, the more reduction is applied (see Equation (2)). When this number is too small, historical information accumulated in the search

process will be essentially lost, and the the algorithms behave close to a random search. In our experiments, we found a value between 0.95 and 1 seems to be a good choice. In our final setup, parameter ρ is set to 0.98.

- p_{best} : This parameter, together with ρ , controls how the lower bound and upper bound of pheromone trails will be computed. Recall that when $p_{best} \rightarrow 0$, the difference between $\tau_{min}(t)$ and $\tau_{max}(t)$ gets smaller, which means the search is getting more random and more emphasis is given to search space exploration. In our experiments, we found that p_{best} should be bigger than 0.5. Once it is above this threshold, both algorithms for RCS and TCS problems perform robustly. In our final setup, p_{best} is set to 0.93.
- m and N : The ant count m and the iteration number N are closely related and have a direct impact on the algorithms execution time. Roughly, the product of m and N gives an estimation of how many scheduling instances the algorithms will cover. Theoretically, the bigger this product, the better the performance. Also, it is intuitive to see that these parameters should be positively correlated with the complexity of the test sample. In our work, we prefer to use a fixed setting for these parameters in order to make the algorithm simpler. As reported above, with $m = 10$ and N is set to be 150 and 100 for the TCS and RCS problem respectively, our algorithms work well over a wide range of testing samples. In a further study, we varied m between 1 and 10, and N from 50 to 1000. We find that little performance improvement is seen after N is bigger than 250 when m is reasonable large (≥ 4). We contribute this to the fact that the pheromone trails converge after a large number of iterations. If N is smaller than 100, we will often miss the optimal solution because of premature termination. This is especially true for the TCS problem. Similarly, when m is bigger than 6, we see little improvement. The best tradeoff of m seems to be between 4 and 6. It is interesting to notice that these numbers are very close to the average branching factor of the testing samples. These results implies that we may still have room to fine tune these two parameters to further improve the performance/cost tradeoff of the algorithms.

VIII. RELATED WORK

To the best of our knowledge, the only other reported work on using Ant Colony Optimization to solve the operation scheduling problem is done by Kopuri *et al.* [46]. Compared to our work, their study is limited to the timing constrained scheduling problem.

To address the TCS problem, their algorithm has a different formulation and is more closely related to the classic force-directed scheduling algorithm. They use a modified self force computation, where predecessor and successor forces are dropped in the overall force consideration. This force is calculated by linear combination of normalized classic self-force and the pheromone trails. Since the resulting value can be both negative and positive, it is hard to act as an indicator for operation selection during the scheduling construction process. In their work, simple random selection is used.

Our algorithm uses a dynamically computed distribution graph for the corresponding resource k for the local heuristic and force calculation is not needed. We believe it provides the following benefits:

- It is directly tied with the optimization target, i.e. minimizing the resource cost.
- It is faster to compute.
- The value range for the distributed graph is non-negative, which enables more effective operation selection strategy than random selection as discussed in Section IV-C.

Moreover, as discussed in Section IV, our algorithm can be readily extended to handle different design scenarios such as multiple-cycle operations, mutually exclusive operations, operation chaining and pipelining. It is unclear if their algorithm can be easily extended to do so, and only single cycle operations were used in their study.

It is known that premature convergence is an important issue in ant based approaches and our experience shows this is an important factor for the operation scheduling problem. In order to cope with this, MAX-MIN formulation is used in our algorithms for both timing and resource constrained scheduling. No such mechanism was used in [46].

Finally, the effectiveness and efficiency of our algorithms is tested over a comprehensive benchmark suite compiled from real-world applications. The performance with respect to solution quality, stability, scalability, and timing performance is more thoroughly studied and reported here. Only limited results on a small number of samples were reported in [46].

IX. CONCLUSION

In this work, we presented two novel heuristic searching methods for the resource and timing constrained scheduling problems based on the MAX-MIN Ant System. Our algorithms employ a collection of agents that collaborate to explore the search space. We proposed a stochastic decision making strategy in order to combine global and local heuristics to effectively conduct this exploration. As the algorithms proceed in finding better quality solutions, dynamically computed local heuristics are utilized to better guide the searching process.

A comprehensive set of benchmarks was constructed to include a wide range of applications. Experimental results over our test cases showed promising results. The proposed algorithms consistently provided higher quality results over the tested examples and achieved very good savings comparing to traditional simulated annealing, list scheduling and force-directed scheduling approaches. Furthermore, the algorithm demonstrated robust stability over different applications and different selection of local heuristics, as evidenced by a much smaller deviation over the results.

REFERENCES

- [1] Semiconductor Industry Association, "National Technology Roadmap for Semiconductors," 2003.
- [2] K. Kennedy and R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.
- [3] A. Aletà, J. M. Codina, J. Sánchez, and A. González, "Graph-Partitioning based Instruction Scheduling for Clustered Processors," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 2001.
- [4] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
- [5] J. E. Smith, "Dynamic instruction scheduling and the astronautics ZS-1," *IEEE Computer*, vol. 22, no. 7, pp. 21–35, 1989.
- [6] T. Stütze and H. H. Hoos, "MAX-MIN Ant System," *Future Generation Comput. Systems*, vol. 16, no. 9, pp. 889–914, September 2000.
- [7] H. Topcuoğlu, S. Hariri, and M. you Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, 2002.
- [8] D. Bernstein, M. Rodeh, and I. Gertner, "On the Complexity of Scheduling Problems for Parallel/Pipelined Machines," *IEEE Transactions on Computers*, vol. 38, no. 9, pp. 1308–13, September 1989.
- [9] K. Wilken, J. Liu, and M. Heffernan, "Optimal instruction scheduling using integer programming," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000.
- [10] P. G. Paulin and J. P. Knight, "Force-directed scheduling in automatic data path synthesis," in *24th ACM/IEEE Conference Proceedings on Design Automation Conference*, 1987.
- [11] —, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 661–679, 1989.
- [12] W. F. J. Verhaegh, E. H. L. Aarts, J. H. M. Korst, and P. E. R. Lippens, "Improved force-directed scheduling," in *EURO-DAC '91: Proceedings of the conference on European design automation*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1991, pp. 430–435.
- [13] W. F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aarts, J. H. M. Korst, A. van der Werf, and J. L. van Meerbergen, "Efficiency improvements for force-directed scheduling," in *Proceedings of the 1992 IEEE/ACM International Conference on Computer-aided Design*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 286–291.
- [14] I.-C. Park and C.-M. Kyung, "Fast and near optimal scheduling in automatic data path synthesis," in *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*. New York, NY, USA: ACM Press, 1991, pp. 680–685.
- [15] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, February 1970.
- [16] M. Heijligers and J. Jess, "High-level synthesis scheduling and allocation using genetic algorithms based on constructive topological scheduling techniques," in *International Conference on Evolutionary Computation*, Perth, Australia, 1995, pp. 56–61.
- [17] A. Sharma and R. Jain, "Insyn: Integrated scheduling for DSP applications," in *DAC*, 1993, pp. 349–354.
- [18] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, 1974.
- [19] M. Grajcar, "Genetic List Scheduling Algorithm for Scheduling and Allocation on a Loosely Coupled Heterogeneous Multiprocessor System," in *Proceedings of the 36th ACM/IEEE Conference on Design Automation Conference*, 1999.
- [20] S. J. Beaty, "Genetic algorithms versus tabu search for instruction scheduling," in *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, 1993.
- [21] P. H. Sweany and S. J. Beaty, "Instruction scheduling using simulated annealing," in *Proceedings of 3rd International Conference on Massively Parallel Computing Systems*, 1998.
- [22] R. Kolisch and S. Hartmann, *Project Scheduling: Recent models, algorithms and applications*. Kluwer Academic Publishers, 1999, ch. Heuristic Algorithms for Solving the Resource-Constrained Project Scheduling problem: Classification and Computational Analysis.
- [23] A. Auyeung, I. Gondra, and H. K. Dai, *Advances in Soft Computing: Intelligent Systems Design and Applications*. Springer-Verlag, 2003, ch. Integrating random ordering into multi-heuristic list scheduling genetic algorithm.
- [24] M. Dorigo, V. Maniezzo, and A. Coloni, "Ant System: Optimization by a Colony of Cooperating Agents," *IEEE Transactions on Systems, Man and Cybernetics, Part-B*, vol. 26, no. 1, pp. 29–41, February 1996.
- [25] J. L. Deneubourg and S. Goss, "Collective Patterns and Decision Making," *Ethology, Ecology & Evolution*, vol. 1, pp. 295–311, 1989.
- [26] S. Fenet and C. Solnon, "Searching for maximum cliques with ant colony optimization," *3rd European Workshop on Evolutionary Computation in Combinatorial Optimization*, April 2003.
- [27] L. M. Gambardella, E. D. Taillard, and M. Dorigo, "Ant colonies for the quadratic assignment," *Journal of the Operational Research Society*, vol. 50, no. 2, pp. 167–176, 1996.
- [28] D. Costa and A. Hertz, "Ants can colour graphs," *Journal of the Operational Research Society*, vol. 48, pp. 295–305, 1996.

- [29] G. Leguizamón and Z. Michalewicz, "A new version of ant system for subset problems," in *Proceedings of the 1999 Congress of Evolutionary Computation*. IEEE Press, 1999, pp. 1459–1464.
- [30] R. Michel and M. Middendorf, *New Ideas in Optimization*. London, UK: McGraw Hill, 1999, ch. An ACO algorithm for the shortest supersequence problem, pp. 51–61.
- [31] S. Fidanova, "Evolutionary Algorithm for Multiple Knapsack Problem," in *Proceedings of PPSN-VII, Seventh International Conference on Parallel Problem Solving from Nature*, ser. Lecture Notes in Computer Science. Springer Verlag, Berlin, Germany, 2002.
- [32] L. M. Gambardella, E. D. Taillard, and G. Agazzi, *New Ideas in Optimization*. London, UK: McGraw Hill, 1999, ch. A multiple ant colony system for vehicle routing problems with time windows, pp. 51–61.
- [33] R. S. Parpinelli, H. S. Lopes, and A. A. Freitas, "Data mining with an ant colony optimization algorithm," *IEEE Transaction on Evolutionary Computation*, vol. 6, no. 4, pp. 321–332, August 2002.
- [34] R. Schoonderwoerd, O. Holland, J. Bruten, and L. Rothkrantz, "Ant-based load balancing in telecommunications networks," *Adaptive Behavior*, vol. 5, pp. 169–207, 1996.
- [35] G. Wang, W. Gong, and R. Kastner, "A New Approach for Task Level Computational Resource Bi-partitioning," *15th International Conference on Parallel and Distributed Computing and Systems*, vol. 1, no. 1, pp. 439–444, November 2003.
- [36] —, "System level partitioning for programmable platforms using the ant colony optimization," *13th International Workshop on Logic and Synthesis, IWLS'04*, June 2004.
- [37] —, "Application partitioning on programmable platforms using the ant colony optimization," *Journal of Embedded Computing*, vol. 2, no. 1, 2006.
- [38] W. J. Gutjahr, "ACO algorithms with guaranteed convergence to the optimal solution," *Inf. Process. Lett.*, vol. 82, no. 3, pp. 145–153, 2002.
- [39] G. Pruesse and F. Ruskey, "Generating linear extensions fast," *SIAM J. Comput.*, vol. 23, no. 2, pp. 373–386, 1994.
- [40] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997.
- [41] G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, and C. Sapuntzakis, *The Basic SUIF Programming Guide*, Computer Systems Laboratory, Stanford University, August 2000.
- [42] M. D. Smith and G. Holloway, *An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization*, Division of Engineering and Applied Sciences, Harvard University, July 2002.
- [43] "http://express.ece.ucsb.edu/benchmark/."
- [44] T. Wangtong, P. Y. K. Cheung, and W. Luk, "Comparing Three Heuristic Search Methods for Functional Partitioning in Hardware-Software Codesign," *Design Automation for Embedded Systems*, vol. 6, no. 4, pp. 425–49, July 2002.
- [45] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. New York, NY: John Wiley & Sons, 1989.
- [46] S. Kopuri and N. Mansouri, "Enhancing scheduling solutions through ant colony optimization," in *International Symposium on Circuits and Systems (ISCAS'04)*. Vancouver, Canada: IEEE, May 2004.



Gang Wang (M'98) is currently a PhD student at the Department of Electrical and Computer Engineering, University of California, Santa Barbara. He received the Bachelor of Electrical Engineering degree from Xi'an Jiaotong University in 1992, and Master of Computer Science degree from Chinese Academy of Sciences in 1995, both in China. From 1995 to 1997, he conducted research work at Michigan State University (East Lansing, MI, US), and Carnegie Mellon University (Pittsburgh, PA, US), focusing on speech and image understanding. In 1997, he joined

Computer Motion Inc. and worked as a Principle Engineer on research and development of surgical robotics systems. His research areas include evolutionary computation, reconfigurable computing, nanocomputing, computer aided design and design automation. He has been an IEEE member since 1998.



Wenrui Gong (S'02) received his Bachelor of Engineering degree in Computer Science from Sichuan University, China, in 1999. He received his Master of Science degree in Electrical and Computer Engineering, in 2002, from the University of California at Santa Barbara, where he is currently pursuing the Ph.D. degree. His research interests include architectural synthesis and compilation techniques for reconfigurable computing systems, and combinatorial optimization algorithms and their applications.



Brian DeRenzi is an undergraduate in the Department of Electrical and Computer Engineering at the University of California, Santa Barbara. He will be graduating in June 2006, with a Bachelor of Science degree in Computer Engineering. His current research focuses on high level synthesis and compilation techniques for reconfigurable systems.



Ryan Kastner (S'00M'04) is an associate professor in the Department of Electrical and Computer Engineering at the University of California, Santa Barbara. He received a PhD in Computer Science (2002) at UCLA, a masters degree in engineering (2000) and bachelor degrees (BS) in both electrical engineering and computer engineering (1999), all from Northwestern University. His current research interests lie in the realm of embedded system design, in particular, the use of reconfigurable computing devices for digital signal processing.

Professor Kastner has published over 70 technical articles, and is the author of the book, "Synthesis Techniques and Optimizations for Reconfigurable Systems", available from Kluwer Academic Publishing (now Springer). He is a member of numerous conference technical committees including the *International Conference on Computer Aided Design (ICCAD)*, the *Design Automation Conference (DAC)*, *Design, Automation and Test in Europe, GLOBECOM*, the *International Conference on Computer Design (ICCD)*, the *Great Lakes Symposium on VLSI (GLSVLSI)*, the *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)* and the *International Symposium on Circuits and Systems (ISCAS)*. He serves on the editorial board for the *Journal of Embedded Computing*.