Instruction Scheduling Using $\texttt{MAX} - \texttt{MIN}$ Ant Colony Optimization

ABSTRACT

Instruction scheduling is a fundamental step for a mapping an application to a computational device. It takes a behavioral application specification and produces a schedule for the instructions onto a collection of system processing units. The objective is to minimize the completion time of the given application while effectively utilizing the computational resources. The instruction scheduling problem is $\mathcal{NP}$-hard, thus it is necessary to develop effective heuristic methods in order to provide a qualitative scheduling solution. In this paper, we present a novel instruction scheduling algorithm using MAX-MIN Ant Colony Optimization approach. The algorithm utilizes a unique hybrid approach by combining the ant colony meta-heuristic with list scheduling. The algorithm uses local and global heuristics that dynamically adjust to the input application. Compared with a number of different list scheduling heuristics, our algorithm generates better results over all the tested benchmarks. Furthermore, we show that our algorithm consistently achieves a near optimal solution.

Keywords

Instruction Scheduling, List Scheduling, MAX-MIN Ant System

1. INTRODUCTION

As the fabrication technology advances and transistors become more plentiful, modern computing systems 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 will yield 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. In order to efficiently and effectively map applications onto these systems, designers will increasingly rely on system level synthesis tools. A fundamental process of system synthesis tools is creating a mapping of a behavioral model of an application 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 synthesis or compilation. Or the tool may take a transaction level behavior and create a register transfer level (RTL) circuit description. This is often called hardware or behavioral synthesis [22]. Both hardware and software synthesis flows are needed for the use and design of future computing systems.

Instruction scheduling (IS) is an important problem in system synthesis. An inappropriate scheduling of the instructions can fail to exploit the true potential of the system and can offset the gain from possible parallelization. Instruction scheduling appears in a number of different problems in system synthesis, e.g., compiler design for superscalar and VLIW microprocessors [28, 17], distributed clustering computation architectures [3] and behavioral synthesis of ASICs and FPGAs [22].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Instruction scheduling is done 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). As an example, Figure 1 gives the DFG for an Auto Regression (AR) filter.

Instruction scheduling can be classified as resource-constrained or performance-constrained. Given a DFG, clock cycle, resource count and resource delays, a resource-constrained scheduling finds the minimum number of control time-steps for executing the DFG. Performance-constrained scheduling tries to determine the minimum number of resources needed for a given scheduling deadline. Though performance constraints are imposed in many cases, resource-constrained scheduling is more frequent in practice. First, this is because in most of the cases, the DFGs are 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 performance constrained schedulers. The performance constraint of each block (DFG) is not easy to define since blocks are typically serialized and budgeting global performance constraint for each block is not trivial [20].

Instruction scheduling methods can be further classified as static scheduling and dynamic scheduling [28]. Static instruction 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 schedule decisions on-the-fly. In addition to minimizing of the program’s completion time, dynamic scheduling methods must also minimize the overhead paid for running the scheduler.

The instruction scheduling problem is $\mathcal{NP}$-hard [6]. Although it is possible to formulate and solve the problem using Integer Linear Programming (ILP) [33, 25], 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. These methods include list scheduling [32, 27, 2], forced-directed scheduling [24], genetic algorithm [16], tabu search [5], path-based scheduling [7], simulated annealing [31], graph theoretical and computational geometry approaches [20, 3]. Among them, list scheduling is most common due to its simplicity of implementation and capability of generating reasonably good results; there exist a wide variety of realizations of this approach and the success is heavily dependent on the input application [32].

In this paper, we focus our attention on the resource-constrained static instruction scheduling problem. In our algorithm, a collection of agents cooperate together to search for a good scheduling solution. Both global and local heuristics are combined in a stochastic decision making process in order to effectively and efficiently explore the search space. The quality of the resultant schedules is evaluated using a list scheduler and feed back to the agents to dynamically adjust the heuristic. The main contribution of our work is the formulation of an instruction scheduling algorithm that:

- Utilizes a unique hybrid approach combining list scheduling and the recently developed MAX-MIN ant system heuristic [30];
- Dynamically computes local and global heuristics based on the input application to adaptively search to solution space;
- Generates consistently good scheduling results over all testing cases compared to a variety of list scheduling algorithms and the optimal ILP solution. Furthermore, the algorithm demonstrates
robust stability over a variety of application benchmarks of large size.

The rest of the paper is arranged as following. We formally define the resource-constrained instruction scheduling problem in Section 2. In Section 3, we review the list scheduling method. In Section 4, we present a hybrid approach combining list scheduling and the MAX-MIN ant colony optimization to address the resource-constrained instruction scheduling problem. Experimental results for the new algorithm are presented in Section 5. We conclude in Section 6.

2. PROBLEM DEFINITION

Given a set of instructions and a collection of computational units, the instruction scheduling problem schedules the instructions onto the computing units such that the execution time of these instructions are minimized, while respecting the capacity limit imposed by the number of resources available.

The instructions and their dependencies are defined by a data flow graph (DFG) $G(V, E)$, where each of the nodes $v_i \in V (i = 1, \ldots , n)$ represents an operation $op_i$, and the edges $e_{ij}$ between the nodes denote the data dependency between operations $v_j$ and $v_i$. A DFG is a directed acyclic graph. The dependencies define a partially ordered relationship (denoted by the symbol $\preceq$) among the nodes. Without affecting the problem, we simplify our discussion by adding two virtual predecessors, and node root is the only starting node in the DFG, which has no predecessor, and node end is the only exiting node and has no successor.

Additionally, we have a collection of computing resources, e.g. ALUs, adders, and multipliers. There are $R$ different resource types and $r_j > 0$ gives the number of units of resource type $j(1 \leq j \leq R)$. Furthermore, each operation defined in the DFG can be performed by at least one type of resource. When each of the operations is uniquely associated with one resource type, we call it homogeneous instruction scheduling. If an operation can be performed by more than one resource types, we call it heterogeneous instruction scheduling [32]. Moreover, we assume the cycle delays for each operation on different type resources are known as $d(i, j)$. Of course, root and end always have zero delays. Finally, we assume the execution of the operations is non-preemptive, that is, once an instruction starts execution, it must finish without being interrupted.

A schedule is given by the vector $(s_{root}, f_{root}, (s_1, f_1), \ldots , (s_{end}, f_{end}))$ where $s_i$ and $f_i$ indicate the starting and finishing time of the operation $op_i$. The resource-constrained instruction scheduling problem is formally defined as $\text{min}(\Delta)$ 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 $r_j$, where $1 \leq j \leq R$.

3. LIST SCHEDULING

List scheduling is a commonly used heuristic for solving a variety of scheduling problems. A list scheduler takes a data flow graph and a priority list of all the nodes in the DFG as input. The list is sorted with decreasing magnitude of priority assigned to each of the operation. The list scheduler maintains a ready list, i.e. nodes whose predecessors have already been scheduled. In each iteration, the scheduler scans the priority list and operations with higher priority are scheduled first. Scheduling an operator to a control step makes its successor operations ready, which will be added to the ready list. This process is carried until all of the operations have been scheduled. When there exist more than one ready nodes sharing the same priority, ties are broken randomly. A pseudo code implementation of list scheduling is shown in Algorithm 1.

Algorithm 1: Resource-Constrained List Scheduling

It is easy to see that list scheduler always generates feasible schedule. Furthermore, it has been shown that a list scheduler is always capable of producing the optimal schedule for resource-constrained instruction scheduling problem if we enumerate the topological permutations of the DFG nodes with the input priority list [18].

The success of the list scheduler is highly dependent on the priority function [31, 22] and the structure of the input application (DFG) [18]. One simple, commonly used priority function assigns the priority inversely proportional the mobility, i.e., the greater the mobility the smaller the priority and vice-versa. This would ensure that operations with large mobility are deferred to later control steps because the number of control steps into which they could go is greater. Many other priority functions have been proposed [2, 18, 16, 4]. It is commonly agreed that there is no single good heuristic for prioritizing the DFG nodes across a range of applications. Our results in Section 5 confirm this.

4. MAX – MIN ANT SYSTEM FOR INSTRUCTION SCHEDULING
In this section, we present our algorithm of applying Ant System heuristic, or more specifically the MAX-MIN Ant System (MMAS) [30], for solving the optimal instruction scheduling problem under resource constraints. First, we give a brief introduction on the basics of Ant Colony Optimization and its applications. Then we give the details of our algorithm for instruction scheduling.

4.1 Basic Ant Colony Optimization

The Ant System (AS) or Ant Colony Optimization (ACO) algorithm, originally introduced by Dorigo et al. [11], is a cooperative heuristic searching algorithm inspired by the ethological study on the behavior of ants. It was observed [10] that ants - who lack sophisticated vision - could manage to establish the optimal path between their colony and the food source within a very short period of time. This is done by an indirect communication known as *stigmergy* via the chemical substance, or *pheromone*, left by the ants on the paths. Though any single ant moves essentially at random, it will make a decision on its direction biased by the “strength” of the pheromone trails that lie before it, where a higher amount of pheromone hints a better path. As an ant traverses a path, it reinforces that path with its own pheromone. A collective autocatalytic behavior emerges as more ants will choose the shortest trails, which in turn creates an even larger amount of pheromone on those short trails, which makes those short trails more likely to be chosen by future ants.

The ACO algorithm is inspired by such 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) [11], for which it gave competitive results comparing with traditional methods.

Researchers have since formulated ACO methods for a variety of traditional NPhard problems. These problems include the maximum clique problem [12], the quadratic assignment problem [15], the graph coloring problem [9], the shortest common super-sequence problem [19, 21], and the multiple knapsack problem [13]. ACO also has been applied to practical problems such as the vehicle routing problem [14], data mining [23], network routing problem [26] and the hardware/software partitioning problem [8].

4.2 Algorithm Formulation

As discussed in Section 3, the effectiveness of list scheduler heavily depends on the priority list. There exist many different heuristics on how to order the list, however, 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.

In our work, we address this problem in an evolutionary manner. The proposed algorithm is a build upon Ant System approach and the traditional list scheduling algorithm shown in Algorithm 1. In our algorithm, the instruction scheduling optimization is formulated as an iterative searching process. Each iteration consists of two stages. First, AS algorithm is applied in which a collection of ants traverse the DFG to construct individual instruction lists using global and local heuristics associated with the DFG nodes. Secondly, these results are given to a list scheduler in order to evaluate their quality. Based on this evaluation, the heuristics are adjusted to favor better solution components.

The hope is that further iterations will benefit from the adjustment and come up with better instruction list.

The first stage of our algorithm proceeds similarly as the basic Ant System algorithm reported in [11] for solving the TSP problem. In order to solve the instruction scheduling problem, each instruction or DFG node $op_k$ is associated with $n$ pheromone trails $\tau_{ij}$, where $j = 1, \ldots, n$. They indicate the global favorableness of assigning the $i$-th instruction at the $j$-th position in the priority list (See Figure 2).

Initially, $\tau_{ij}$ is set with some fixed value $\tau_0$.

For each iteration, $m$ ants are released and each starts to construct an individual priority list by filling the list with one instruction a step. Every ant will have memory about the instructions it has already selected in order to guarantee the validity of the constructed list. Upon starting step $j$, the ant has already selected $j-1$ instructions of the DFG. To fill the $j$-th position of the list, the ant chooses the next instruction $op_i$ probabilistically according to a probability:

$$P_{ij} = \frac{\tau_{ij}(t)^{\alpha} \eta_{ij}}{\sum_k(\tau_{ik}(t)^{\alpha} \eta_{ik})} \text{ if } op_k \text{ is not scheduled yet}$$

$$0 \text{ otherwise}$$

where the eligible instructions $op_k$ are those yet to be scheduled, $\eta_{ij}$ is a local heuristic for selecting instruction $k$, $\alpha$ and $\beta$ are parameters to control the relative influence of the distributed global heuristic $\tau_{ij}$ and local heuristic $\eta_{ij}$. Intuitively, the ant favors a decision on a choice that possesses higher volume of pheromone trail and better local heuristic.

The local heuristic $\eta$ gives the local favorableness of scheduling the $i$-th instruction at the $j$-th position of the priority list. In our work, we experimented with different well-known heuristics [22] for instruction scheduling.

1. **Instruction Mobility (IM):** The mobility of an instruction gives the range for scheduling the instruction. It is computed as the difference between ALAP and ASAP results. The smaller the mobility, the more urgent the instruction is. Especially, when the mobility is zero, the instruction is on the critical path.

2. **Instruction Depth (ID):** Instruction depth is the length of the longest path in the DFG from the instruction to the sink. It is an obvious measure for its priority of an operation as it gives number of instructions we must pass.

3. **Latency Weighted Instruction Depth (LWID):** LWID is computed in a similar manner as ID, except the nodes along the path are weighted using the latency of the operation that the node represents.

4. **Successor Number (SN):** The motivation of using the number of successors is to 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.

One important difference between our algorithm and other Ant System algorithms is that we use a dynamic local heuristic in the scheduling process. It is indicated by step 13 in Algorithm 2. This technique allows better local guidance to the ants for making the selection in the next iteration. We will illustrate this feature with the instruction mobility heuristic.

Typically, the mobility of an instruction is computed by using ALAP and ASAP results. One important input parameter in computing 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. As the algorithm proceeds, every time 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 instruction $i$ is computed as:

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

In the second stage of our algorithm, the lists constructed by the ants are evaluated using a traditional list scheduler as shown in Algorithm 1. The quality of the scheduling result from ant $h$ is judged by the schedule latency $L_h$. Upon finishing of each iteration, the pheromone trail is
updated according to the quality of instruction lists. In the mean time, a certain amount of the it will evaporate. More specifically, we have:

\[ \tau_{ij}(t) = \rho \cdot \tau_{ij}(t) + \sum_{h=1}^{m} \Delta \tau_{ij}^{h}(t) \quad \text{where } 0 < \rho < 1. \tag{3} \]

Here \( \rho \) is the evaporation ratio, and

\[ \Delta \tau_{ij}^{h} = \begin{cases} Q / L_h & \text{if instruction } i \text{ is scheduled at } j-\text{th position by ant } h \\ 0 & \text{otherwise} \end{cases} \tag{4} \]

\( Q \) is a fixed constant to control the delivery rate of the pheromone.

Two important operations are taken in this pheromone trail updating process. The evaporation operation is necessary for AS to be effective and diversified to explore different parts of the search space, while the reinforcement operation ensures that the favorable instruction orderings receive a higher volume of pheromone and will have better chance to be 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.

In the above algorithm, the ants construct a priority list using the same traversing method as that used in TSP formulation [11]. In fact, this turns out to be a naive 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 \( n \) instructions. 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 instructions 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. Though it would be generally hard to quantify the search space reduction as the totally number of topological sortings highly depends on the structure of the graph, it is believed to be significant. By adopting this technique, in the final version of our algorithm, the ant traverses the DFG similarly as the list scheduling process and fills instruction list one by one. At each step, the ant will select an instruction based on Equation 1 but only from all the ready instructions, that is all the instructions whose predecessors have all been positioned.

### 4.3 MAX - MIN Ant System

Premature convergence to local minima is a critical algorithmic issue that can be experienced by all evolutionary algorithms. Balancing exploration and exploitation is no trivial in these algorithms, especially for algorithms that use positive feedback such as the original Ant System [11]. The MAX - MIN Ant System (MMAS) is specially designed to address this problem.

MMAS [29, 30] is built upon the original Ant System algorithm. It improves the original algorithm by providing dynamically evolving bounds on the pheromone trails such that the heuristic is always within a limit comparing with that of the best path. As result, during run time, all possible paths will have a non-trivial probability of being selected and thus it encourages broader exploration of the search space.

MMAS forces the pheromone trails be limited with an 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 \( \tau_{\max} \) [30] is shown in (5) where \( S^{gb}() \) is the global best solution found so far:

\[ \tau_{\max}(t) = \frac{1}{1 - \rho f(S^{gb}(t - 1))} \tag{5} \]

The lower bound is defined as (6):

\[ \tau_{\min}(t) = \frac{\tau_{\max}(t)}{1 + \sqrt{p_{\text{best}}}} \tag{6} \]

where \( p_{\text{best}} \in [0, 1] \) is a controlling parameter to dynamically adjust the bounds of the pheromone trails. The physical meaning of \( p_{\text{best}} \) 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 \( \text{avg} \) is the average size of the decision choices over all the iterations. For a TSP problem of \( n \) cities, \( \text{avg} = n/2 \). It is noticed from (6) that lowering \( p_{\text{best}} \) will result tighter range for the pheromone heuristic. As \( p_{\text{best}} \to 0 \), \( \tau_{\min}(t) \to \tau_{\max}(t) \), which means more emphasis is given to search space exploration.

Theoretical treatment of using the pheromone bounds and other modifications on the original ant system algorithm are proposed in [30]. These include a pheromone updating policy that only utilizes the best performing ant, initializing pheromone with \( \tau_{\max} \) and combining local search with the algorithm. It was reported by the authors that MMAS was the best performing ACO approach and provided very high quality solutions.

In our experiments, we implemented both the basic AS and the MMAS algorithms. The latter consistently achieves better scheduling results, especially for large size testing cases. A pseudo code implementation of the final version of our algorithm using MMAS is shown as Algorithm 2, where the pheromone bounding step is indicated as step 12.

```plaintext
procedure MaxMinAntScheduling(G, R)
input: DFG G(V, E), resource set R
output: instruction schedule
1: initialize parameter \( \rho, \tau_{ij}, p_{\text{best}}, \tau_{\max}, \tau_{\min} \)
2: construct \( m \) ants
3: BestSolution \( \leftarrow \phi \)
4: while ending condition is not met do
5: for \( i = 0 \) to \( m \) do
6: \( \text{ant}(i) \) 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: \( \text{BestSolution} \leftarrow L(i) \)
10: end if
11: end for
12: update \( \tau_{\max} \) and \( \tau_{\min} \) based on (5) and (6)
13: update \( \eta \) if needed
14: update \( \tau_{ij} \) based on (3)
15: end while
16: return BestSolution
```

Algorithm 2: MAX-MIN Ant System Instruction Scheduling

### 5. EXPERIMENTAL RESULTS

We have implemented the MMAS Instruction Scheduling (MMAS-IS) algorithm and verified its performance by comparing them with the popularity used list scheduling algorithm. Furthermore, to better assess the quality of our algorithm, the same resource-constrained instruction scheduling tasks are also formulated as integer linear programming problems [22] and then optimally solved using CPLEX. The test DFGs consist of the AR filter, elliptic wave filter, two implementations of the cosine function, two FIR filters, and a second order differential equation implementation. These graphs range in size from 21 nodes to 82 nodes.

In our experiments, heterogeneous computing units are allowed, i.e. one type of instruction can be performed by different types of resources. For example, multiplication can be performed by either a faster multiplier or a regular one. Furthermore, multiple same type units are also allowed.

For each of the benchmark samples, we run the proposed algorithm with different choice of local heuristics. For each choice, we perform 5 runs where in each run we allow 100 iterations. The number of ants is set to 5. The evaporation rate \( \rho \) is configured to be 0.98. The scaling
parameters for global and local heuristics are set to be $\alpha = \beta = 1$ and delivery rate $Q = 1$. The best schedule delay is reported at the end of each run and then the average value is reported as the performance for the corresponding setting.

Table 1 summarizes our experiment results. Compared with the traditional list scheduling method, the proposed algorithm generates better results consistently over all testing cases. For some of the testing samples, it provides significant improvement on the schedule delay. The biggest saving achieved is 23%. This is obtained for the FIR1 benchmark data when LWID is used as the local heuristic for the proposed algorithm and also as the heuristic for constructing the priority list for the traditional list scheduler.

Besides the absolute schedule latency achieved, another important aspects for assessing the quality of a scheduling algorithm is its stability over different input applications. As indicated in Section 3, the performance of traditional list scheduler heavily depends on the input. This is echoed by the data in Table 1. 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 1, the average standard deviation for all the benchmarks over different heuristic choices is 0.8128, while that for the MMAS algorithm is only 0.1673. In other words, we can expect to achieve much more stable scheduling results on different application DFGs regardless the choice of local heuristic. This is a great attribute desired in practice.

Finally compared with the optimal scheduling results computed by 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. For all the benchmarks with known optima, our algorithm improves the average schedule latency by 44% compared with the list scheduling heuristics. For the larger size DFGs such as COSINE1 and COSINE2, CPLEX fails to generate optimal results after more than 7 hours of execution on a SPARC workstation with a 440MHz CPU and 384MByte memory. Experiment results of our algorithm are obtained on a Linux box with a 2GHz CPU. For all the benchmarks, the runtime of the proposed algorithm ranged from 0.1 seconds to 1.76 seconds.

The evolutionary effect on the global heuristics $\tau_{ij}$ is illustrated in Figure 2. It plots the pheromone values for the ARF testing sample after 100 iterations of the proposed algorithm. The x-axis is the index of instruction node in the DFG (shown in Figure 1), 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 instruction - the bigger the size of the dot, the stronger value of the pheromone.

It is clearly seen from Figure 2 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 instruction list only with topologically sorted order. It is also interesting to notice that though a good amount of instructions have a limited few alternative “good” positions (such as instruction 6 and 26), for some of the instructions the pheromone heuristics are strong enough to lock their positions. For example, according to its pheromone distribution, instruction 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 mentioned above. This shows that the proposed algorithm has the possibility to discover better ordering which may be hard to achieve intuitively.

<table>
<thead>
<tr>
<th>Benchmark (nodes/edges)</th>
<th>Resources</th>
<th>CPLEX (latency/runtime)</th>
<th>List Scheduling</th>
<th>MMAS-IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARF(28/30)</td>
<td>2a, 1fm, 2m</td>
<td>11/22</td>
<td>IM 11 ID 13 LWID 13 SN 11 MMAS-IS 11</td>
<td></td>
</tr>
<tr>
<td>FIR1(44/43)</td>
<td>1a, 1fm, 1m, 3i, 3o</td>
<td>14/11560</td>
<td>19 21 21 21 16.2 16.4 16.2 17</td>
<td></td>
</tr>
<tr>
<td>FIR2(40/39)</td>
<td>2a, 2m, 3i, 3o</td>
<td>13/232</td>
<td>19 19 19 18 17.2 17.2 17 17.8</td>
<td></td>
</tr>
<tr>
<td>COSINE1(66/76)</td>
<td>2a,2m, 1fm, 3i, 3o</td>
<td>†</td>
<td>19 20 18 18 17.4 18.2 17.6 17.6</td>
<td></td>
</tr>
<tr>
<td>COSINE2(82/91)</td>
<td>2a,2m, 1fm, 3i, 3o</td>
<td>†</td>
<td>23 23 23 23 21.2 21.2 21.2 21.2</td>
<td></td>
</tr>
<tr>
<td>EWF(34/47)</td>
<td>1a, 1fm, 1m</td>
<td>27/24000</td>
<td>28 31 31 28 27.2 27 27 27.2</td>
<td></td>
</tr>
<tr>
<td>HAL(21/25)</td>
<td>1a, 1fm, 1m, 3i, 3o</td>
<td>8/32</td>
<td>8 8 8 8 8 8 8 8</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Benchmark Evaluation Results
Schedule latency is in cycles; Runtime is in second; † indicates CPLEX failed to provide result after 7 hours of computing.
(Resource Labels: a=alu, fm=faster multiplier, m=multiplier, i=input, o=output)
(Heuristic Labels: IM=Instruction Mobility ID=Instruction Depth, LWID=Latency Weighted Instruction Depth, SN=Successor Number)

6. CONCLUSION

In this work, we presented a novel heuristic searching method for the resource constrained instruction scheduling problem based on the MAX − MIN Ant System algorithm. Our algorithm works as a collection of agents collaborate to explore the search space. A stochastic decision making strategy is proposed in order to combine global and local heuristics to effectively conduct this exploration. As the algorithm proceeds in finding better quality solution, dynamically computed local heuristics are utilized to better improve the efficiency of
searching. Proposed algorithm consistently provided near optimal partitioning results over tested examples and achieved good saving on the schedule length comparing with traditional list scheduling approach. Furthermore, the algorithm demonstrated robust stability over different applications and different selection of local heuristics, evidenced by a much smaller deviation over the results.

7. REFERENCES

[8] citation removed to maintain author confidentiality.