Instruction Scheduling Using $\mathcal{MAX} - \mathcal{MIN}$ Ant System Optimization

Gang Wang Wenrui Gong Ryan Kastner
Department of Electrical and Computer Engineering
University of California at Santa Barbara
Santa Barbara, CA 93106-9560
{wanggang, gong, kastner}@ece.ucsb.edu

ABSTRACT

Instruction scheduling is a fundamental step for mapping an application to a computational device. It takes a behavioral application specification and produces a schedule for the instructions onto a collection of 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 effective heuristic methods are necessary to provide a qualitative scheduling solution. In this paper, we present a novel instruction scheduling algorithm using MAX-MIN Ant System Optimization approach. The algorithm utilizes a unique hybrid approach by combining the ant system meta-heuristic with list scheduling, where the local and global heuristics are dynamically adjusted to the input application in an iterative manner. Compared with force-directed scheduling and a number of different list scheduling heuristics, our algorithm generates better results over all the tested benchmarks with better stability. Furthermore, by solving the test samples optimally using ILP formulation, we show that our algorithm consistently achieves a near optimal solution.

Categories and Subject Descriptors: J.6 [Computer-Aided Engineering]: Computer-Aided Design (CAD)

General Terms: Algorithms, Design, Theory

Keywords: Instruction Scheduling, Force-Directed 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 automated design tools. One fundamental process of these 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 [11]. 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 the above process. An inappropriate scheduling of the instructions can fail to exploit the true potential of the system and can offset the gain from possible parallelism. Instruction scheduling appears in a number of different important problems, e.g. compiler design for superscalar and VLIW microprocessors distributed clustering computation architectures [3] and behavioral synthesis of ASICs and FPGAs [11].

Instruction scheduling is often 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 2 gives the DFG for an Auto Regression (AR) filter. Based on the constraint, 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. 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 is not easy to define since blocks are typically serialized and budgeting global performance constraint for each block is not trivial [10].

The instruction scheduling problem is $\mathcal{NP}$-hard [5]. Although it is possible to formulate and solve the problem using Integer Linear Programming (ILP) , 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 [17, 2] , forced-directed scheduling [14] , genetic algorithm [8], tabu search, simulated annealing, graph theoretic and computational geometry approaches[10, 3].

Among them, list scheduling is 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) [11, 9]. One commonly used priority function assigns the priority inversely proportional to the mobility. Many other priority functions have been proposed [2, 9, 8, 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 4 confirm this. Force-directed scheduling proposed by Paulin and Knight [14] is another popularly used method. Essentially, it is an extension of list scheduling. In force-directed approach, the selection of a candidate operation to be scheduled in a given time step is done using the concept of *force*, which tries to balance the operation execution on different computing units. The results of force-directed scheduling have been shown to be superior to list scheduling.

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 heuristics. 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 [16];
- Dynamically computes local and global heuristics based on the input application to adaptively search the solution space;
- Generates consistently good scheduling results over all testing cases compared with a range of list scheduling heuristics, force-directed scheduling and the optimal ILP solution, and demonstrates stable quality 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 present a hybrid approach combining list scheduling and the MAX-MIN ant system optimization to address the defined instruction scheduling problem. Experimental results for the new algorithm are presented in Section 4. We conclude with Section 5.

2. PROBLEM DEFINITION

Given a set of instructions and a collection of computational units, the resource constrained 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 the dependencies among them are defined by 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. The dependencies define a partially ordered relationship (denoted by the symbol $\preccurlyeq$) among the nodes. Without affecting the problem, we simplify our discussion by adding two virtual nodes $root$ and end, for which no instruction is associated with. We assume that $root$ 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 types and $r_j > 0$ gives the number of units of resource type $j(1 \leqslant j \leqslant R)$. Furthermore, each operation defined in the DFG can be performed by at least one type of resource. 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

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

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

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

3. $\mathcal{MAX} - \mathcal{MIN}$ 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) [16], for solving the optimal instruction scheduling problem under resource constraints. First, we give a brief introduction on the basics of Ant System Optimization and its applications. Then we give the details of our algorithm for instruction scheduling.

3.1 Basic Ant System Optimization

The Ant System (AS) or Ant Colony Optimization (ACO) algorithm, originally introduced by Dorigo *et al.* [6], is a cooperative heuristic searching algorithm inspired by the ethological study on the behavior of ants. It was observed 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 AS 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 AS was successfully applied was the Traveling Salesman Problem (TSP) [6], for which it gave competitive results comparing with traditional methods.

Researchers have since formulated AS methods for a variety of traditional $\mathcal{NP}$-hard problems. These problems include the maximum clique problem, the quadratic assignment problem, the graph coloring problem, the shortest common super-sequence problem, and the multiple knapsack problem [7]. AS also has been applied to practical problems such as the vehicle routing problem , data mining, network routing problem and the hardware/software partitioning problem [13, 18].

3.2 Algorithm Formulation for IS

As discussed in Section 1, 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 built upon Ant System approach and the traditional list scheduling algorithm. 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.

In order to solve the instruction scheduling problem, each instruction or DFG node op_i is associated with n pheromone trails τ_{ij}, where $j = 1, \dots, n$. They indicate the global favorableness of assigning the i-th instruction at the j-th position in the priority list. Initially, τ_{ij} is set with some fixed value τ_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:

$$\mathsf{p}_{ij} = \begin{cases} \frac{\tau_{ij}(t)^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_k (\tau_{kj}^{\alpha}(t) \cdot \eta_{kj}^{\beta})} & \text{if } op_k \text{ is not scheduled yet} \\ 0 & \text{otherwise} \end{cases} \qquad (1)$$

where the eligible instructions op_k are those yet to be scheduled, η_{ik} is a local heuristic for selecting instruction k, 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 a choice that possesses higher volume of pheromone trail and better local heuristic.

The local heuristic η 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 [11] 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 1. 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 (see section 3.3 for definitions for f and S^{gb}):

$$\eta_i(t) = \frac{1}{ALAP(f(S^{gb}(t-1)), i) - ASAP(i) + 1} \qquad (2)$$

In the second stage of our algorithm, the lists constructed by the ants are evaluated using a traditional list scheduler. 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 it will evaporate. More specifically, we have:

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

Here ρ is the evaporation ratio, and

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

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 [6]. 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. For instance, using this technique on a simple 11 node example [11] 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 graphs[1], it is generally significant. By adopting this technique, in

[1]We tried to compute the search space reduction for Figure 2 using GenLE [15]. It failed to produce any result within 100 computer hours.

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.

3.3 $\mathcal{MAX}-\mathcal{MIN}$ 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 [6]. The MAX-MIN Ant System (MMAS) is specially designed to address this problem.

MMAS [16] 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 a result, 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 to be limited with an evolving bounds, that is for iteration t, $\tau_{min}(t) \leqslant \tau_{ij}(t) \leqslant \tau_{max}(t)$. If we use f to denote the cost function of a specific solution S, the upper bound τ_{max} [16] is shown in (5) where $S^{gb}(\cdot)$ is the global best solution found so far:

$$\tau_{max}(t) = \frac{1}{1-\rho}\frac{1}{f(S^{gb}(t-1))} \qquad (5)$$

The lower bound is defined as (6):

$$\tau_{min}(t) = \frac{\tau_{max}(t)(1-\sqrt[n]{p_{best}})}{(avg-1)\sqrt[n]{p_{best}}} \qquad (6)$$

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 noticed from (6) that lowering p_{best} will result tighter range for the pheromone heuristic. As $p_{best} \to 0$, $\tau_{min}(t) \to \tau_{max}(t)$, 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 [16]. 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 by the authors that MMAS was the best performing AS 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 1, where the pheromone bounding step is indicated as step 12.

```
procedure MaxMinAntScheduling($G$,$R$)
input: DFG $G(V,E)$, resource set $R$
output: instruction 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:       $q(i) =$ 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 (5) and (6)
13:    update $\eta$ if needed
14:    update $\tau_{ij}$ based on (3)
15: end while
16: return $BestSolution$
```

Algorithm 1: MAX-MIN Ant System Instruction Scheduling

4. EXPERIMENTAL RESULTS

We have implemented the MMAS Instruction Scheduling (MMAS-IS) algorithm and compared its performance with the popularly used list scheduling and force-directed scheduling algorithms. Furthermore, to better assess the quality of our algorithm, the same resource constrained instruction scheduling tasks are also formulated as integer linear programming problems [11] and then optimally solved using CPLEX.

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 example, we may have 3 faster multipliers and 2 regular ones.

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. Figure 1 shows such a distribution for the MediaBench benchmarks, in which a wide range of complete applications for image processing, communications and DSP applications are included. Our analysis covers the *epic, jpeg, g721, mpeg2enc, mpeg2dec,* and *mesa* packages. 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, almost all bigger DFGs are of little interest with respect to system performance. They are typically related with system initialization and are executed only once.

Based on the above analysis, our test DFGs are selected to consist of the AR filter, elliptic wave filter, two implementations of the cosine function, two FIR filters, and a second order differential equation implementation. The sizes of these samples range from 21 nodes to 82 nodes. They present a set of difficult or very difficult testing samples for instruction scheduling problem and were used in various literatures [10, 12].

For each of the benchmark samples, we run the proposed algorithm with different choices of local heuristics. For each choice, we perform 5 runs where in each run we allow 100 iterations. The number of ants per iteration is set to 5. 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 $Q = 1$. 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.

Table 1 summarizes our experiment results. Compared with the a variety of list scheduling approaches and the force-directed 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 latency. The biggest saving achieved is 23%. This is obtained for the FIR2 benchmark

Benchmark (nodes/edges)	Resources	CPLEX (latency/runtime)	Force Directed	List Scheduling				MMAS-IS(average over 5 runs)			
				IM	ID	LWID	SN	IM	ID	LWID	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 1: Benchmark Evaluation Results
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=mutiplier, i=input, o=output)
(Heuristic Labels: IM=Instruction Mobility ID=Instruction Depth, LWID=Latency Weighted Instruction Depth, SN=Successor Number)

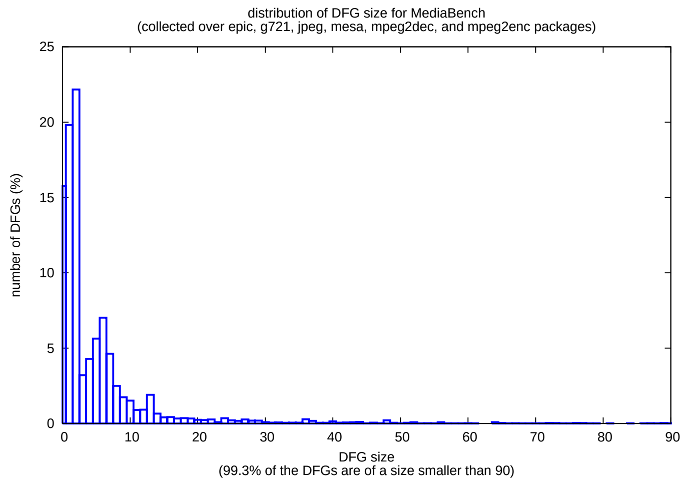

Figure 1: Distribution of DFG size for MediaBench

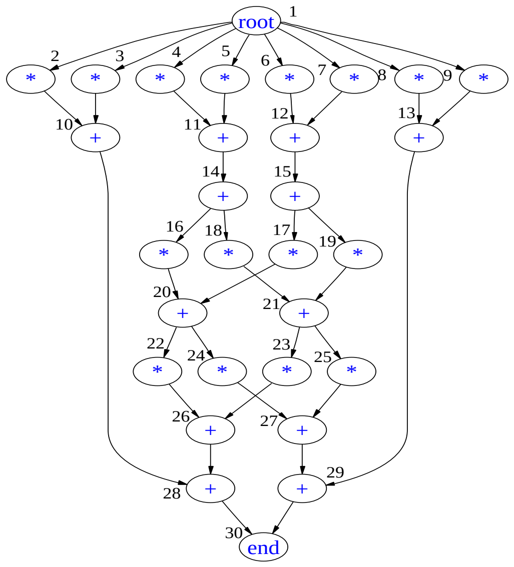

Figure 2: Data Flow Graph of AR Filter.
(The number by the node is the index assigned for the instruction.)

when LWID is used as the local heuristic for our algorithm and also as the heuristic for constructing the priority list for the traditional list scheduler.

Though the results of force-directed scheduler are generally superior to that of 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 with 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.

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 1, 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 list scheduler over all the benchmarks and 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.

One possible explanation for the above attribute 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 instructions. Furthermore, the schedule of the instructions is done all in once. Differently, in the proposed algorithm, local heuristics are used stochastically and combined with the pheromone values to determine the instructions' 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 on 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-driven scheduling.

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 seconds 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-driven 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, it takes 12.7% and 21.2% more execution time respectively.

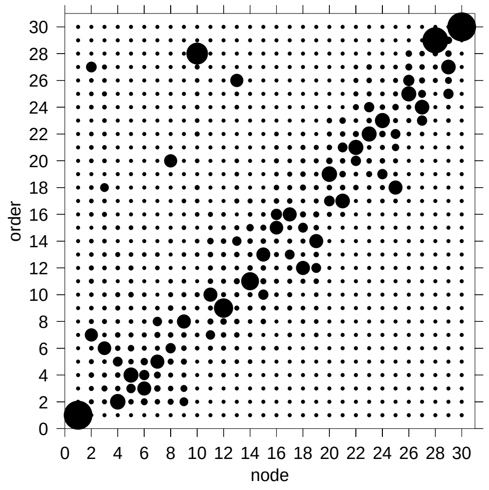

Figure 3: Pheromone Heuristic Distribution for ARF

The evolutionary effect on the global heuristics τ_{ij} is illustrated in Figure 3. 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 2), 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 the value of the pheromone.

It is clearly seen from Figure 3 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.

5. 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 guide the searching process.

Experimental results over our test cases showed promising results. The proposed algorithm consistently provided near optimal partitioning results over tested examples and achieved good saving on the schedule length comparing with traditional list scheduling and force-directed 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.

6. REFERENCES

[1] National Technology Roadmap for Semiconductors. 2003.

[2] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, 1974.

[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] A. Auyeung, I. Gondra, and H. K. Dai. *Advances in Soft Computing: Intelligent Systems Design and Applications*, chapter Integrating random ordering into multi-heuristic list scheduling genetic algorithm. Springer-Verlag, 2003.

[5] D. Bernstein, M. Rodeh, and I. Gertner. On the Complexity of Scheduling Problems for Parallel/Pipelined Machines. *IEEE Transactions on Computers*, 38(9):1308–13, September 1989.

[6] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man and Cybernetics, Part-B*, 26(1):29–41, February 1996.

[7] L. M. Gambardella, E. D. Taillard, and G. Agazzi. *New Ideas in Optimization*, chapter A multiple ant colony system for vehicle routing problems with time windows. McGraw Hill, 1999.

[8] 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.

[9] R. Kolisch and S. Hartmann. *Project Scheduling: Recent models, algorithms and applications*, chapter Heuristic Algorithms for Solving the Resource-Constrained Project Scheduling problem: Classification and Computational Analysis. Kluwer Academic Publishers, 1999.

[10] S. O. Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh. A super-scheduler for embedded reconfigurable systems. In *IEEE/ACM International Conference on Computer-Aided Design*, 2001.

[11] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[12] M. Narasimhan and J. Ramanujam. A fast approach to computing exact solutions to the resource-constrained scheduling problem. *ACM Trans. Des. Autom. Electron. Syst.*, 6(4):490–500, 2001.

[13] R. S. Parpinelli, H. S. Lopes, and A. A. Freitas. Data mining with an ant colony optimization algorithm. *IEEE Transaction on Evolutionary Computation*, 6(4):321–332, August 2002.

[14] 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.

[15] G. Pruesse and F. Ruskey. Generating linear extensions fast. *SIAM J. Comput.*, 23(2):373–386, 1994.

[16] T. Stützle and H. H. Hoos. MAX-MIN Ant System. *Future Generation Comput. Systems*, 16(9):889–914, September 2000.

[17] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transaction on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[18] 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, PDCS'2003*, 1(1):439–444, November 2003.