System Level Partitioning for Programmable Platforms Using the Ant Colony Optimization

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

Abstract

The flexibility, performance and cost effectiveness of programmable platforms have lead to their widespread use for embedded applications. The partitioning of an application tasks onto the different computational cores of a platform is an important step in the mapping of embedded applications onto these systems. This paper presents a novel approach for this problem based on the Ant Colony Optimization. In our algorithm, a collection of agents cooperate to effectively explore the search space. Experimental results show that our algorithm provides robust results that are qualitatively close to optimal with minor computational cost. Compared with the popularly used simulated annealing approach, the proposed algorithm gives better solutions with substantial reduction in execution time.

1 Introduction

The continued scaling of the feature size of the transistor will soon yield incredibly complex digital systems consisting of more than one billion transistors. This allows extremely complicated systems on a chip, which may consist of multiple processor cores, programmable logic cores, embedded memory blocks and dedicated application specific components. At the same time, the fabrication techniques have become increasingly complicated and expensive. Current day designs (below 150 nm feature size) already cost over one million dollars to fabricate. These forces have created a sizable and emerging market for programmable platforms.

A programmable platform is a device consisting of one or multiple programmable cores. Xilinx Virtex [29] and Altera Excalibur devices [3] are two examples of a programmable platform. These platforms may consist of hard cores, programmable cores and/or soft cores. A hard core is dedicated static processing unit, e.g. ARM processor in Excalibur or the PowerPC core in Virtex. A programmable core is some kind of programmable logic device (PLD) (e.g. FPGA, CPLD). A soft core is a processing unit implemented on programmable logic, e.g. CAST DSP core on Virtex or Nios on Excalibur.

Programmable platforms have emerged as a flexible, high performance, cost effective device for the implementation of embedded applications. Their programmability allows application development on the device after is fabricated. Therefore, the functionality of the device can change over time. This is especially important for embedded systems where the hardware cannot be easily upgraded (e.g. computers in cars). As standards change, you just need to reprogram the device, rather than physically replace the hardware device.

Programmable platforms provide a good price point for low volume applications. Also, it allows ”low” end users to create designs using newest, highest performance manufacturing process. Furthermore, programmable devices enable fast prototyping, which allows for faster time to market.

The programmability in these devices ranges from extremely fine grain control in PLD, to coarse grain control in the microprocessor. This allows for fine grain optimizations (bit level optimizations in the PLD), instruction level optimizations (on processor cores) and task level optimizations (across programmable cores).

These complex programmable platforms require a variety of system design techniques to allow system level design space exploration by application programmers. The system partitioning problem assigns computational tasks to the different computational cores of a platform. It is a key step in the mapping of embedded applications onto these systems.

At the system level, the application programmer designs applications using a set of interdependent tasks, where each task is a coarse grained set of computations with a well defined interface. The fundamental problem distills to automatically partitioning the tasks onto the system resources,

while optimizing system performance methods such as execution time, hardware cost and power consumption.

Some early works investigate the hardware/software partitioning problem [13, 16, 24, 26]; it is difficult to name a clear winner [11]. Partitioning issues for system architectures with reconfigurable logic components have also been studied [4, 17, 21]. These works assume a reconfigurable device coupling with a processor core in their partitioning problem.

The partitioning problem is $\mathcal{NP}$-complete [15]. Although it is possible to use brute force search or integer linear programming (ILP) formulations [22] for small problem instances, in the general case, the optimal solution is computationally intractable. Different heuristic methods have been proposed to try to effectively provide suboptimal solutions for the problem [13, 19, 12, 18, 1, 27]. These methods use a range of heuristics, including Simulated Annealing, Genetic Algorithm, Tabu Search, and Kernighan/Lin approach.Wiangtong *et al.* [28] compared three popularly used heuristic methods for the system level partitioning problem, and provided a good survey on the motivation of using task level abstraction and related work. Software tools are also developed for system level partitioning. For instance, in COSYMA [8] application tasks are mapped based on simulated annealing.

In this paper, we present a novel heuristic searching approach to the system partitioning problem based on the *Ant Colony Optimization* (ACO) algorithm [10]. In the proposed algorithm, a collection of agents cooperate together to search for a good partitioning solution. Both global and local heuristics are combined in a stochastic decision making process in order to effectively and efficiently explore the search space. Our approach is truly multi-way and can be easily extended to handle a variety of system requirements.

2 Ant Colony Optimization

The Ant Colony Optimization(ACO) algorithm, originally introduced by Dorigo *et al.* [10], is a cooperative heuristic searching algorithm inspired by the ethological study on the behavior of ants. It was observed [9] 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 on the "strength" of the pheromone trails that lie before it, where a higher amount of pheromone hints a better path. As an ant traverses a path, it reinforces that path with its own pheromone. A collective autocatalytic behavior emerges as more ants will choose the shortest trails, which in turn creates an even larger amount of pheromone on 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) [10], for which it gave competitive results comparing with traditional methods. Researchers have since formulated ACO methods for a variety of traditional $\mathcal{NP}$-hard problems. These problems include the maximum clique problem the quadratic assignment problem, the graph coloring problem, the shortest common super-sequence problem, and the multiple knapsack problem [7]. ACO also has been applied to practical problems such as the vehicle routing problem, data mining and network routing problem [7].

3 ACO for System Partitioning

3.1 Problem Definition

A crucial step in the design of systems with heterogenous computing resources is the allocation of the computation of an application onto the different computing components. This system partitioning problem plays a dominant role in the system cost and performance.

It is possible to perform partitioning at multiple levels of abstraction. For example, operation (instruction) level partitioning is done in the Garp project [5], while the work in [28][12] are on the functional task level. In this work, we focus on partitioning at the task or functional level. One of the reasons we select the task level partitioning is that it is commonly found that a bad partitioning in the task level is hard to correct in lower level abstraction. Additionally, task level partitioning is typically requested in the earlier stage of the design so that further hardware synthesis can be performed. Another variant in partitioning algorithms is the goal for optimization. Though the model and the algorithm presented in this paper has no limit in this respect, our experimental results reported here focus on minimizing the worse case execution time with pre-defined hardware cost.

We formally define the system partitioning problem as follows. For a given system architecture, a set of computing resources are defined for the system partitioning task. We use R to represent this set where $r = |R|$ is the number of resources in the system. The notation r_i $(i = 1, \ldots, r)$ refers to the ith resource R.

An application to be partitioned onto the system is given as a set of tasks $T = \{t_1, \ldots, t_N\}$, where the atomic partitioning unit is a task. A *task* is a coarse grained set of computation with a well defined interface. The precedence constraints between tasks are modeled using a task graph. A *task graph* is a directed acyclic graph (DAG) $G = (T, E)$, where $T = \{t_0, t_1, \ldots, t_n\}$ is a collection of tasks, and E is a

set of directed edges. Each task node defines a functional unit for the program, which contains information about the computation it needs to perform. There are two special nodes t_0 and t_n which are virtual task nodes. They are included for the convenience of having an unique starting and ending point of the task graph. An edge $e_{ij} \in E$ defines an immediate precedence constraint between t_i and t_j. For a given partitioning, the execution of a task graph runs in the following way: the tasks of different precedence levels are sequentially executed from the top level down, while tasks in the same precedence level but allocated on different system components can run concurrently.

Generally, there are r^N unique partitions for a task graph of size N. A *feasible* partitioning is a partitioning that satisfies the system constraints. An *optimal* partitioning is a feasible partitioning that minimizes the system objective function. The multi-way partitioning problem is then defined as: Find a set of partitions $P = \{P_1, \ldots, P_r\}$ on r resources, where $P_i \subseteq T$, $P_i \cap P_j = \phi$ for any $i \neq j$ that minimizes a system objective function under a set of system constraints.

The objective function may be a multivariate function of different system parameters (e.g. minimiz execution time or power consumption) while system cost (e.g. cost per device must be less than \$5) is an example of a system constraint.In this work, we use the critical path execution time of a task graph as the objective function and a fixed amount of area as the constraint.

3.2 Augmented Task Graph

We further introduce the Augmented Task Graph as the underlying model for the multi-way system partitioning problem. An *Augmented Task Graph* (ATG) $G' = (T, E', R)$ is an extension of the task graph G. It is derived from G as follows: Given a task graph $G = (T, E)$ and a system architecture R, each node $t_i \in T$ is duplicated in G'. For each edge $e_{ij} = (t_i, t_j) \in E$, there exist r directed edges from t_i to t_j in G', each corresponding to a resource in R. More specifically, we have

$$e'_{ijk} = (t_i, t_j, r_k), \text{ where } e_{ij} \in E, \text{ and } k = 1, ..., r$$

An edge e'_{ijk} represents the *binding* of edge e_{ij} with resource r_k. Our algorithm uses these augmented edges to make a local decision at task node t_i about the binding of the resource on task t_j[1]. We call this an *augmented edge*. The original task graph G is called the *support* of G'.

An example of ATG is shown in Figure 1(a) for a 3-way partitioning problem. In this case, we assume the system contains 3 computing resources, a Power PC microprocessor, a fixed size FPGA, and a digital signal processor (DSP). In the graph, the solid links indicate that the pointed task nodes are allocated to the DSP, while the dotted links for tasks partitioned onto Power PC and dot-dashed links for FPGAs. It is easy to see that partitioning algorithm based on the ATG model can be easily adapted if more resources are available. All we need to do is add additional augmented edges in the ATG.

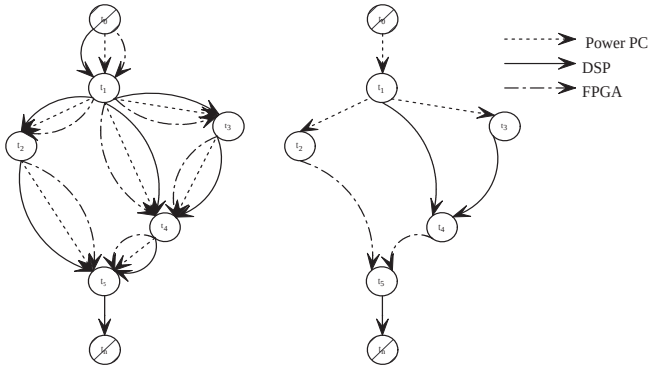

Figure 1: ATG for 3-way Partitioning

Based on the ATG model, a specific partitioning for the tasks on the multiple resources is a graph G_p, where G_p is a subgraph of G' that is isomorphic to its *support* G, and for every task node t_i in G_p, all the incoming edges of t_i are bounded with the same resource (say) r. Further, we say that partition G_p allocates task t_i to resource r. Figure 1(b) shows a sample partitioning for the ATG illustrated in Figure 1(a). In this partitioning, task 1, 2, and 3 are allocated onto the Power PC, task 4 is partitioned on to the DSP and task 5 for the FPGAs. As t_n is a virtual node, we do not care the status of the edge from t_5 to t_n.

3.3 ACO Formulation for System Partitioning

Based on the ATG model, our goal is to find a feasible partitioning G_p for G', which provides the optimal performance subject to the predefined system constraints. We introduce a new heuristic method for solving the multi-way system partitioning problem using the ACO algorithm. Essentially, the algorithm is a multi-agent[2] stochastic decision making process that combines local and global heuristics during the searching process. Each agent traverses the ATG and attempts to create a feasible partitioning by selecting the next move probabilistically according to the combined heuristics. The quality of the solution is measured by the overall execution time of the ATG, from t_0 to t_n, together with the consideration of the predefined constraints, such as hardware area cost limit. The quality measurement is used to reinforce the *good* solutions. The global heuristic information is distributed as pheromone trails on the edges of the ATG.

The proposed algorithm proceeds as follows:

1. Initially, associate each augmented edge e'_{ijk} in the ATG with a pheromone τ_{ijk}, a global heuristic indi-

[1]This will be further explained in Section 3.3

[2]We use the terms "agent" and "ant" interchangeably.

cating the favorableness for selecting the corresponding resource; the value of the pheromone on each augmented edge is initially set at the value τ_0;

2. Put m ants on task node t_0;
3. Each ant crawls over the ATG to create a feasible partitioning $P^{(l)}$, where $l = 1, \ldots, m$;
4. Evaluate the partitions generated by each of the m ants. The quality of a particular partition $P^{(l)}$ is measured by the overall execution time $time_{P^{(l)}}$.
5. Update the pheromone trails on the edges as follows:

$$\tau_{ijk} \leftarrow (1-\rho)\tau_{ijk} + \sum_{l=1}^{m} \Delta\tau_{ijk}^{(l)}$$

where $0 < \rho < 1$ is the evaporation ratio, $k = 1, \ldots, r$, and

$$\Delta\tau_{ijk}^{(l)} = \begin{cases} Q/time_{P^{(l)}} & \text{if } e'_{ijk} \in P^{(l)} \\ 0 & \text{otherwise} \end{cases}$$

and Q is a fixed constant to control the delivery rate of the pheromone.

6. If the ending condition is reached, stop and report the best solution found. Otherwise go to step 2.

Step 3 is an important part in the proposed algorithm. It describes how an individual ant "crawls" over the ATG and generates a solution. In step 3, each ant traverses the graph in a topologically sorted manner in order to satisfy the precedence constraints of task nodes. The trip of an ant starts from t_0 and ends at t_n, the two virtual nodes that do not require allocation. By visiting the nodes in the topologically sorted order, we insure that every predecessor node is visited before we visit the current node and that every incoming edge to the current node has been evaluated.

At each task node t_i where $i \neq n$, the ant makes a probabilistic decision on the allocation for each of its successor task nodes t_j based on the pheromone on the edge. The pheromone is manipulated by the distributed global heuristic τ_{ijk}) and a local heuristic such as the execution time and the area cost for a specific assignment of the successor node. More specifically, an ant at t_i guesses that node t_j be assigned to resource r_k according to the probability:

$$p_{ijk} = \frac{\tau_{ijk}^{\alpha}\eta_{jk}^{\beta}}{\sum_{l=1}^{r}\tau_{ijl}^{\alpha}\eta_{jl}^{\beta}}$$

Here η_{jk} is the local heuristic if t_j is assigned to resource r_k. In our work, we simply use the inverse of the cost of having task t_j allocated to resource r_k. It is intuitive to notice that the probability p_{ijk} favors an assignment that yields smaller local execution time and area cost, and an assignment that corresponds with the stronger pheromone. We focus on achieving the optimal execution time subject to hardware area constraint, therefore a simple weighted combination is used to estimate the cost:

$$cost_{jk} = w_t \cdot time_{jk} + w_a \cdot area_{jk}$$

where $time_{jk}$ and $area_{jk}$ are the execution time and hardware area cost estimates, constants w_t and w_a are scaling factors to normalize the balance of the execution time and area cost.

Upon entering a new node t_j, the ant also has to make a decision on the allocation of the task node t_j based on the guesses made by all of the immediate precedents of t_j. It is guaranteed those guesses are already made since that the ant travels the ATG in a topologically sorted manner. Different strategies can be used. For example, we can simply make the assignment based on the vote of the majority of the guesses. In our implementation, this decision is again made probabilistically based on the distribution of the guesses, i.e. the possibility of assigning t_j to r_k is:

$$p_{jk} = \frac{\text{count of guess } r_k \text{ for } t_j}{\text{count of immediate precedents of } t_j}$$

The above decision making process is carried by the ant until all the task nodes in the graph have been allocated.

At the end of each iteration, the pheromone trails on the edges are updated according to Step 5. First, a certain amount of pheromone is evaporated. From optimization point of view, the evaporation step helps the system escape from local minimums. Secondly, the *good* edges are reinforced. This reinforcement creates additional pheromone on the edges that are included on partition solutions that provide shortest execution time for the task graph. In each run of the algorithm, multiple iterations of the above steps are conducted until we reach the ending condition predefined.

3.4 Extensibility

Though detailed discussion on this topic is beyond the scope of this paper, it is worth mentioning that the above algorithm can be easily extended to fit different system requirements. For example, it is common that certain tasks are predetermined or preferred to run on certain resources. By modifying the decision strategy in step 3, we can easily accommodate this by either forcing a certain selection or biasing the chance for it to be chosen. Better local cost estimation model can be introduced and integrated as the local heuristics. Similarly, different target objective functions can be applied as the global heuristics bounded with the augmented edges. Other global information, such as profiling statistics on the tasks can also be considered in the same manner. Also, one can modify the proposed algorithm to handle different communication channels, each

with a different bandwidth and latency. These channels can either be associated with the augmented edges if they are bounded with the hardware configuration, or may be treated as a task attribute if the task can only use one certain type channel.

4 Testing Environment

4.1 Target Architecture and Benchmarks

Our experiments address the partitioning of multimedia applications on to a programmable, multiprocessor system platform. The target architecture contains one general purpose hard processor core, a soft DSP core, and one programmable core. The architecture is shown in Figure 2.

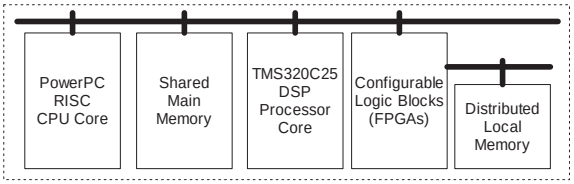

Figure 2: Target architecture

This model is similar to the Xilinx Virtex II Pro Platform FPGA [29], which contains up to four hard CPU cores, 13,404 configurable logic blocks (CLBs) and other peripherals. In our work, we target a system containing one PowerPC 405 RISC CPU core, separate data and instruction memory, and a fixed amount of reconfigurable logic with a capacity of 1,232 CLBs, among which, 724 CLBs are available to be used as general purpose reconfigurable logic (FPGA), and the remaining 508 CLBs embed an FPGA implementation (soft core) of the TMS320C25 DSP processor core [6]. Programmable routing switches provide communication between the different system resources.

This system imposes several constraints on the partitioning problem. The code length of both the PowerPC processor and the DSP processor must be less than the size of the instruction memory, and the tasks implemented on FPGAs must not occupy more than the total number of available CLBs. The execution time and required resources for each task on different resources depends on the implementation of the task. We assumed the tasks are static and precomputed. The communication time cost between interfaces of different processors, such as the interface between the PowerPC and the DSP processor, are known a priori.

Tasks allocated on either the PowerPC processor or the DSP processor are executed sequentially subject to the precedence constraints within the task (i.e. instruction level precedence constraints). Both the potential parallelism among the tasks implemented on FPGAs and the potential parallelism among all the processors are explored, i.e. concurrent tasks may execute in parallel on the different system resources. However, no hardware reuse between tasks assigned to FPGAs is considered. This would make an interesting extension to our work, however, it is outside the scope of this paper. The system constraints are used to determine whether a particular partition solution is feasible. For all the feasible partitions that do not exceed the capacity constraints, the partitions with the shortest execution time are considered the best.

Our experiments are conducted in a hierarchical environment for system design. An application is represented as a task graph in the top level. The task graph, formally described in Section 3.1, is a directed acyclic graph, which describes the precedence relationship between the computing tasks. A task node in the task graph refers to a function, which could be written in high-level languages, such as C/C++. It is analyzed using the SUIF [2] and Machine SUIF [23] tools; the result is imported in our environment as a control/data-flow graph (CDFG). CDFG reflects the control flow in a function, and may contain loops, branches, and jumps. Each node in CDFGs is a basic block, or a set of instructions that contains only one control-transfer instruction and several arithmetic, logic, and memory instructions.

Estimation is carried out for each task node to get performance characteristics, such as execution time, software code length, and hardware area. Based on the specification data of Virtex II Pro Platform FPGA [29] and the DSP processor core [6], we get the performance characteristics for each type of operations. Using these operation (instruction) characteristics, we estimate the performance of each basic block. This information for each task node is used to evaluate a partitioning solution. In each time an ant finds a candidate solution, we perform a critical path-based scheduling over the entire task graph to determine the minimum execution time. Additionally, we estimate the hardware cost and software code length for each task node. The software code length is estimated based on the number of instructions needed to encode the operations of the CDFG. The hardware is scheduled using ASAP scheduling. Based on that we can determine the approximate area needed to implement the task on the reconfigurable logic. We assume that there is no hardware reuse between different tasks.

We create a task level benchmark suite based on the MediaBench applications [20]. Each testing example is formed via a two step process that combines a randomly generated DAG with real life software functions. In order to better assess the quality of the proposed algorithm while the application scales, task graphs of different sizes are generated. For a given task graph, the computation definitions associated with the task nodes are selected from the same application within the MediaBench test suite.

4.2 Experimental Results

4.2.1 Absolute Quality Assessment

When the task graph size is not too big, it is possible to achieve definitive quality assessment for the proposed algorithm by first conducting brute force search on the given

problem. In our experiments, we apply the proposed ACO algorithm on the task benchmark set and evaluate the results with the statistics computed via the brute force search. We give 100 run of the ACO algorithm on each DAG in order to obtain enough evaluation data. For each run, the ant number is set as the average branch factor of the DAG. As a stopping condition, the algorithm is set to iterate 50 times i.e. $I = 50$. The solution with the best execution time found by the ants is reported as the result of each run. In all the experiments, we set $\tau_0 = 100$, $Q = 1,000$, $\rho = 0.8$, $\alpha = \beta = 1$, $w_t = 1$ and $w_a = 2$.

Figure 3 shows the cumulative distribution of the number of solutions found by the ACO algorithm plotted against the quality of those solutions for different problem sizes. The x-axis gives the solution quality compared to the overall number of solutions. The y-axis gives the total number of solutions (in percentage) that are worse than the solution quality. For example, looking at the x-axis value of 2% for size 13, less than 10% of the solutions that the ACO algorithm found were outside of the top 2% of the overall number of solutions. In other words, over 90% of the solutions found by the ACO algorithm are within 2% of all possible partitions. The number of solutions drops quickly showing that the ACO algorithm finds very good solutions in almost every run. In our experiments, 2,163 (or 86%) solutions found by ACO algorithm are within the top 0.1% range. Totally 2,203 solutions, or 88.12% of all the solutions, are within the top 1% range. The figure indicates that a majority of the results are qualitatively close to the optimal.

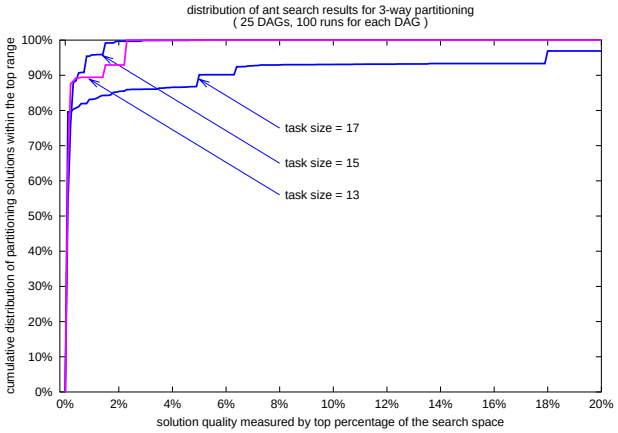

Figure 3: Result quality measured by top percentage

We also evaluate the capability of the algorithm with regard to discovering the optimal partition. In this test, our benchmarks contain task graphs with 13, 15 and 17 nodes and for each size there are 25 different testcases. We first perform the brute force search on each test case, which provides the optimal execution time and the number of partitions that achieve this execution time for the testcase. Based on this information, we can derive theoretical possibility of finding an optimal partition if random sampling is applied. When the task graph size is 13, for a search space with a size of $3^{13} = 1,594,323$, it is shown that over 2,500 runs across the 25 testcases, ACO algorithm found the optimal execution time 2,163 times. Based on this, the probability of finding the optimal solution with our algorithm for these task graphs is 86.44%. Related to this, we found that for 17 testing examples, or 68% of the testing set, our algorithm discovers the optimal partition every time in the 100 runs. This indicates that the proposed algorithm is statistically robust in finding close to optimal solutions. Similar analysis holds when task graph size is 15 or 17.

4.2.2 Compare with Simulated Annealing

In order to further investigate the quality of the proposed algorithm, we compared the results of the ACO algorithm with that of the simulated annealing (SA) approach. Our SA implementation is similar to the one reported in [28]. In our experiments, the most commonly known and used geometric cooling schedule [28] is applied and the temperature decrement factor is set to 0.9. Figure 4 compares the ACO results against that achieved by the SA search sessions of different iteration numbers. The graph is illustrated in the same way as Figure 3. Here SA50 has roughly the same execution time of the ACO, while respectively, SA500 and SA1000 runs approximately 10 times and 20 times longer. We can see that with substantial less execution time, the ACO algorithm achieves better results than the SA approach, even when it is compared with a much more exhaustive SA session such as SA1000.

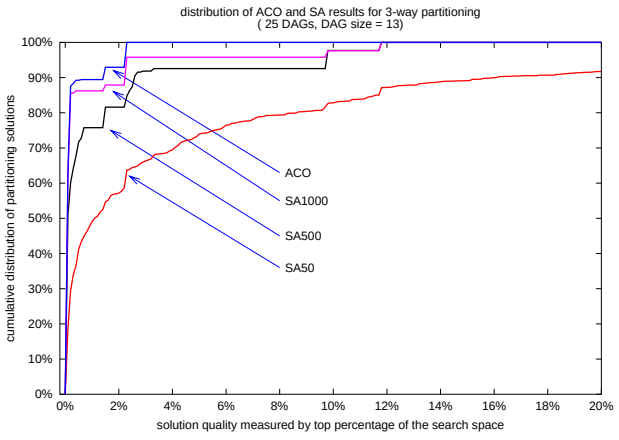

Figure 4: Comparing ACO with SA

When the problem size gets large, it becomes impossible to perform the brute force search to find the true optimal solution. However, we can still assess the quality of the proposed algorithm by comparing the relative difference between its results with that obtained with other

popularly used heuristic methods. Figure 5 shows the cumulative result quality distribution curves for task graphs with 25 nodes. The x axis now reads as the percentage difference on time performance of the partition found by the corresponding algorithm with respect to the *best* execution time over all the experiments. Here the ACO and SA500 take the same amount of execution time, while SA5000 runs at about 10 time longer. It is shown that ACO outperforms SA500 while a much more expensive SA performs comparably.

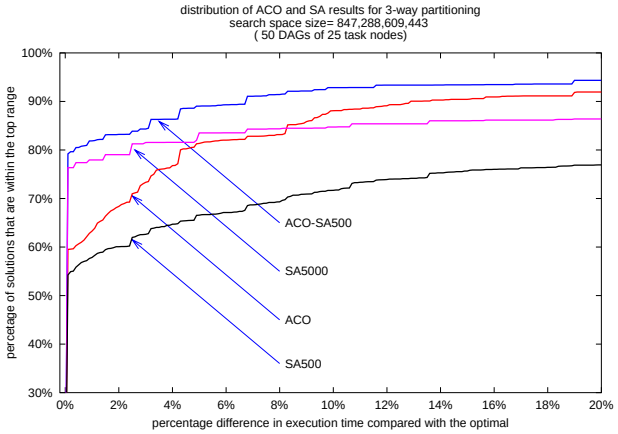

Figure 5: ACO, SA and ACO-SA on big size problem

4.2.3 Hybrid ACO with Simulated Annealing

One possible explanation for the proposed ACO approach to outperform the traditional SA method with regard to short computing time is that in the formulation of the SA algorithm, the problem is modeled with a flat representation, i.e. the task/resource partitioning is characterized as a vector, of which each element stores an individual mapping for a certain task. This model yields simplicity, while loses critical structural relationship among tasks comparing with the ATG model. This further makes it harder to effectively use structural information during the selection of neighbor solutions. For example, in the implementation tested, the internal correlation between tasks is fully ignored. To compensate for this, SA suffers from lengthy low temperature cooling process.

Another problem of SA, which may be more related with the stability of the results' quality than the long computing time, is its sensitivity to the selection of the initial seed solution. Starting with different initial partitions may lead to final results of different quality, besides the possibility of spending computing time on unpromising parts of the search space.

On the other hand, the ACO/ATG model makes effective use of the core structural information of the problem. The autocatalytic nature of how the pheromone trails are updated and utilized makes it more attractive in discovering "good" solutions with short computing time. However, this very behavior raises stagnation problem. For example, it is observed that allowing extra computing time after enough iterations of the ACO algorithm does not have significant benefit regarding to the solution quality. This stagnation problem has been discussed in other works [14, 25] and special problem-dependent recovery mechanisms have to be formulated to ease this artifact.

These complementary characteristics of the two methods motivate us to investigate a hybrid approach that combines the ACO and SA together. That is to use the ACO results as the initial seed partitions for the SA algorithm, it is possible for us to achieve even better system performance with a substantially reduced computing cost. In Figure 5, curve ACO-SA500 shows the result of this approach. It achieves definitively better results comparing with that of SA5000 while only taking 20% of its running time. Similar result holds for task graphs of with bigger size, such as 50 and 100 (For a test case with 100 task node, the computing time can be reduced from about 2 hours to 18 minutes using the hybrid ACO-SA approach).

Overall, we summarize the result quality comparison with Table 1 for large problem instances. It compares the average result qualities reported by ACO, SA500, SA5000 and the hybrid method ACO-SA500. The data is normalized with that obtained by SA500, and the smaller the better. It is easy to see that ACO always outperforms the traditional SA even when SA is allowed a much longer execution time, and the ACO-SA approach provides the best average results consistently with great runtime reduction.

(run time)	SA500 (t)	ACO (t)	SA5000 (10t)	ACO-SA500 (2t)
size = 25	1	0.86	0.90	0.85
size = 50	1	0.81	0.94	0.77
size = 100	1	0.84	0.92	0.80

Table 1: Average Result Quality Comparison

5 Conclusion

In this work, we presented a novel heuristic searching method for the system partitioning problem based on the ACO algorithm. Our algorithm works as a collection of agents work collaboratively 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. We introduced the Augmented Task Graph concept as a generic model for the system partitioning problem. Our approach is truly multi-way and can be easily extended to handle a variety of system requirements.

Preliminary results over our test cases for a 3-way system partitioning task showed promising results. The proposed algorithm consistently provided near optimal partitioning results over tested examples with very minor computational cost. Our algorithm is more effective in finding

the near optimal solutions and scales well as the problem size grows. It is also shown that with substantial less execution time the proposed method achieves better solutions than the popularly used simulated annealing approach. Furthermore, we provide a new hybrid approach combining the ACO and SA that yields even better results than using each of the algorithms individually.

References

[1] S. Agrawal and R. K. Gupta. Data-flow Assisted Behavioral Partitioning for Embedded Systems. In *Proceedings of the 34th Annual Conference on Design Automation Conference*, 1997.

[2] 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.

[3] Altera Corporation. *Excalibur Device Overview Data Sheet*, May 2002.

[4] M. Baleani, F. Gennari, Y. Jiang, Y. Pate, R. K. Brayton, and A. Sangiovanni-Vincentelli. HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, 2002.

[5] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33(4):62–69.

[6] CAST, Texas Instruments Inc. *C32025 Digital Signal Processor Core*, September 2002.

[7] D. Corne, M. Dorigo, and F. Glover, editors. *New Ideas in Optimization*. McGraw Hill, 1999.

[8] A. Österling, T. Benner, R. Ernst, D. Herrmann, T. Scholz, and W. Ye. *Hardware/Software Co-Design: Principles and Practice*, chapter The COSYMA System. Kluwer Academic Publishers, 1997.

[9] J. L. Deneubourg and S. Goss. Collective Patterns and Decision Making. *Ethology, Ecology & Evolution*, 1:295–311, 1989.

[10] 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.

[11] S. A. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models Validation, and Synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[12] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. Hardware/Software Partitioning with Iterative Improvement Heuristics. In *Proceedings of the Ninth International Symposium on System Synthesis*, 1996.

[13] R. Ernst, J. Henkel, and T. Benner. Hardware/Software Cosynthesis for Microcontrollers. *IEEE Design and Test of Computers*, 10(4):64–75, December 1993.

[14] L. M. Gambardella, E. D. Taillard, and M. Dorigo. Ant colonies for the quadratic assignment. *Journal of the Operational Research Society*, 50(2):167–176, 1996.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[16] R. K. Gupta and G. De Micheli. Constrained Software Generation for Hardware-Software systems. In *Proceedings of the Third International Workshop on Hardware/Software Codesign*, 1994.

[17] J. Harkin, T. M. McGinnity, and L. P. Maguire. Partitioning methodology for dynamically reconfigurable embedded systems. *IEE Proceedings - Computers and Digital Techniques*, 147(6):391–396, November 2000.

[18] J. I. Hidalgo and J. Lanchares. Functional Partitioning for Hardware - Codesign Codesign Using Genetic Algorithms. In *Proceedings of the 23rd Euromicro Conference*, 1997.

[19] A. Kalavade and E. A. Lee. A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem. In *Proceedings of the Third International Workshop on Hardware/Software Codesign*, 1994.

[20] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997.

[21] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Proceedings of the 37th Conference on Design Automation*, 2000.

[22] R. Niemann and P. Marewedel. An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming. *Design Automation for Embedded Systems*, 2(2):125–63, March 1997.

[23] 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.

[24] U. Steinhausen, R. Camposano, H. Gunther, P. Ploger, M. Theissinger, H. Veit, H. T. Vierhaus, U. Westerholz, and J. Wilberg. System-Synthesis using Hardware/Software Codesign. In *Proceedings of the Second International Workshop on Hardware/Software Codesign*, 1993.

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

[26] F. Vahid, J. Gong, and D. D. Gajski. A Binary-Constraint Search Algorithm for Minimizing Hardware during Hardware/Software Partitioning. In *Proceedings of the conference on European design automation conference*, 1994.

[27] F. Vahid and T. D. LE. Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning. *Design Automation for Embedded Systems*, 2(2):237–61, March 1997.

[28] T. Wiangtong, P. Y. K. Cheung, and W. Luk. Comparing Three Heuristic Search Methods for Functional Partitioning in Hardware-Software Codesign. *Design Automation for Embedded Systems*, 6(4):425–449, July 2002.

[29] Xilinx, Inc. *Virtex-II Pro Platform FPGA Data Sheet*, January 2003.