Journal of Embedded Computing 1 (2005) 1–18 1
IOS Press

Application partitioning on programmable platforms using the ant colony optimization

Gang Wang*, Wenrui Gong and Ryan Kastner
Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA 93106-9560, USA
Tel.: +1 805 893 3985; E-mail: wanggang@ece.ucsb.edu, gong@ece.ucsb.edu, kastner@ece.ucsb.edu

Abstract. Modern digital systems consist of a complex mix of computational resources, e.g. microprocessors, memory elements and reconfigurable logic. System partitioning – the division of application tasks onto the system resources – plays an important role for the optimization of the latency, area, power and other performance metrics. This paper presents a novel approach for this problem based on the Ant Colony Optimization, in which a collection of agents cooperate using distributed and local heuristic information to effectively explore the search space. The proposed model can be flexibly extended to fit different design requirements. Experiments show that our algorithm provides robust results that are qualitatively close to the optimal with minor computational cost. Compared with the popularly used simulated annealing approach, the proposed algorithm gives better solutions with substantial reduction on execution time for large problem instances. Moreover, a hybrid approach that combines our algorithm and SA achieves even better results with great runtime reduction.

Keywords: System partitioning, ant colony optimization, hardware/software codesign, evolutionary computing, CAD

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 system-on-a-chip (SoC), 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, which have emerged as a flexible, high performance, cost effective choice for embedded applications.

A programmable platform is a device consisting of programmable cores. Its programmability allows application development after it 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, one just need to reprogram the device, rather than physically replace the hardware. For these reasons, programmable platforms provide a good price point for low volume applications. 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.

Xilinx Virtex [45] and Altera Excalibur devices [4] are two examples of such programmable platform. These platforms may consist of hard cores, programmable cores and/or soft cores. A hard core is a 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 [9] on Virtex or Nios [5] on Excalibur.

*Corresponding author. Gang Wang, Tel.: +1 805 893 3985; E-mail: wanggang@ece.ucsb.edu.

ISSN 1740-4460/05/$17.00

Comparing with the traditional single CPU architecture, these complex programmable platforms require more effective computer-aided design (CAD) techniques to allow design space exploration by application programmers. One special challenge resides at the system level design phase. At this stage, the application programmer works with a set of tasks, where each task is a coarse grained set of computations with a well defined interface based on the application. Different from single CPU architecture, a key step in the mapping of applications onto these systems is to assign tasks to the different computational cores. This partitioning problem is NP-complete [23]. Although it is possible to use brute force search or ILP formulations [35] for small problem instances, generally, the optimal solution is computationally intractable. Thus it requires us to develop efficient algorithms in order to automatically partition the tasks onto the system resources, while optimizing performance metrics such as execution time, hardware cost and power consumption.

It is worth mentioning that though the above partitioning problem shares certain similarity with the Job Scheduling Problem (JSP) [24], another well-studied NP-hard problem in the operation optimization community, they are fundamentally different. First, the *jobs* in JSP are independent from each other while the computational tasks are interrelated and constrained by data dependencies among different tasks. Secondly, for every job in JSP, each of its operations is explicitly associated with a resource known *a priori*, while a computational task on the programmable platform is possible to be allocated on different resources as long as the system requirements are met. Finally, the optimization target in JSP is only constrained by the condition that no two jobs are processed at the same time on the same resource. However, in the above task partitioning problem, besides this constraint, we also need respect other system design requirements, such as limits on power consumption and hardware cost.

Some early works [18,25,40,41] investigate the hardware/software partitioning problem, which is a special case of the system partitioning problem discussed here.[1] It is difficult to name a clear winner [16]. Partitioning issues for system architectures with reconfigurable logic components have also been studied [6,26, 32]. These works assume a reconfigurable device coupling with a processor core in their partitioning problem.

Different heuristic methods have been proposed to try to effectively provide sub-optimal solutions for the problem. These methods include Simulated Annealing (SA), Tabu Search (TS), and Kernighan/Lin approach [2,17,18,28,42]. Evolutionary methods [27,36] using Genetic Algorithm (GA) are also studied. Software tools based on these heuristics have been developed for system level partitioning problem. For instance, in COSYMA [11], the application tasks are mapped onto the system architecture using Simulated Annealing. Wiangtong et al. [44] compared three popularly used heuristic methods, and provided a good survey on the motivation and the related work of using task level abstraction. These methods provide practical algorithms for achieving acceptable the system partitioning solutions, however, they also have different drawbacks. Simulated Annealing suffers from long execution time for the low temperature cooling process. For Genetic Algorithm, special effort must be spent in designing the evolutionary operations and the problem-oriented chromosome representation, which makes it hard to adapt to different system requirements.

In this paper, we present a novel heuristic searching approach to the system partitioning problem based on the *Ant Colony Optimization* (ACO) algorithm [14]. 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.

The remainder of the paper is organized as follows. In Section 2, we give a brief introduction to the ACO approach. Section 3 details the proposed algorithm for the constrained multi-way partitioning problem. As the basis of our algorithm, a generic mathematic model for multi-way partitioning is also introduced in this section. In Section 4, we present the experimental heterogenous architecture and the testing benchmark we used in our work. We analysis the experiment results and give assessment on the performance of the proposed algorithm in Section 5. We conclude with Section 7.

2. Ant colony optimization

The Ant Colony Optimization (ACO) algorithm, originally introduced by Dorigo et al. [14], is a coop-

[1] Hardware/software partitioning is equivalent to the system partitioning problem where there is only one microprocessor and one "hardware" resource i.e. ASIC.

erative heuristic searching algorithm inspired by the ethological study on the behavior of ants. It was observed [12] 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. The algorithm can be characterized by the following steps:

1. The optimization problem is formulated as a search problem on a graph;
2. A certain number of ants are released onto the graph. Each individual ant traverses the search space to create its solution based on the distributed pheromone trails and local heuristics;
3. The pheromone trails are updated based on the solutions found by the ants;
4. If predefined stopping conditions are not met, then repeat the first two steps; Otherwise, report the best solution found.

One of the first problems to which ACO was successfully applied was the Traveling Salesman Problem (TSP) [14], for which it gave competitive results comparing with traditional methods. The objective of TSP is to find a Hamiltonian path for the given graph that gives the minimal length. In order to solve the TSP problem, ACO associates a pheromone trail for each edge in the graph. The pheromone indicates the attractiveness of the edge and serves as a global distributed heuristic. For each iteration, a certain number of ants are released randomly onto the nodes of the graph. An individual ant will choose the next node of the tour according to a probability that favors a decision of the edges that possesses higher volume of pheromone. Upon finishing of each iteration, the pheromone on the edges is updated. Two important operations are taken in this pheromone updating process. First, the pheromone will evaporate, and secondly the pheromone on a certain edge is reinforced according to the quality of the tours in which that edge is included. The evaporation operation is necessary for ACO to effectively avoid local minima and diversify future exploration onto different parts of the search space, while the reinforcement operation ensures that frequently used edges and edges contained in better tours receive a higher volume of pheromone, which will have better chance to be selected in the future iterations of the algorithm. The above process is repeated multiple times until a certain stopping condition is reached. The best result found by the algorithm is reported as the final solution.

Researchers have since formulated ACO methods for a variety of traditional NP-hard problems. These problems include the maximum clique problem [19], the quadratic assignment problem [22], the graph coloring problem [10], the shortest common super-sequence problem [31,34], and the multiple knapsack problem [20]. ACO also has been applied to practical problems such as the vehicle routing problem [21], data mining [37], and network routing problem [38]. Recently, it has been applied to tackle the hardware/software codesign problem [43], which is special case of the partitioning problem we discuss here.

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 [8], while the good deal of research work [11,17,28,44] 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 [29]. Additionally, task level partitioning is typically requested in the earlier stage of the design so that further hardware synthesis can be performed.

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 $T_{app} = \{t_1, \ldots, t_N\}$, where the atomic partitioning unit, 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 $T = \{t_0, t_n\} \cap T_{app}$, 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. Notice the precedence constraint is transitive. That is, if we let $\longrightarrow$ denote the precedence constraint, we have:

$$(t_a \longrightarrow t_b) \wedge (t_b \longrightarrow t_c) \Rightarrow t_a \longrightarrow t_c \tag{1}$$

In a task graph, a task can only be executed when all the tasks with higher precedence level have been executed.

If a system contains only one processing resource, e.g. a general purpose processor, it is trivial to determine the system performance; only the sequential constraints between tasks need to be respected. For a system that contains r heterogenous computing resources, the partitioning of the tasks onto different resources becomes critical to the system performance. There are r^N unique partitioning solutions, where N is the number of the actual tasks. Some of these solutions may be infeasible as they violate system constraints.[2] We call a partitioning *feasible* when it satisfies the system constraints. An *optimal* partitioning is a feasible partitioning that minimizes the objective function of the system design.

Thus, the multi-way system partitioning problem is formally 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. minimize 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

To solve the multi-way application partitioning problem, introduce the Augmented Task Graph as the underlying model. An *Augmented Task Graph* (ATG) $G' = (T, E', R)$ is an extension of the traditional task graph G discussed above. 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

$$\begin{aligned} e'_{ijk} = (t_i, t_j, r_k), \text{ where } e_{ij} \in E, \text{ and } k \\ = 1, \ldots, r \end{aligned} \tag{2}$$

In ATG, 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.[3] We call this an *augmented edge*. The original task graph G is called the *support* of G'.

An example of ATG is shown in Fig. 1(a) for a 3-way partitioning problem. In this case, we assume the system contains 3 computing resources, a PowerPC 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 PowerPC 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.

[2] For example, a partitioning solution may allocate a large number of tasks to the reconfigurable logic. However, the reconfigurable logic has a fixed size, and the area occupied by those tasks must be less than the area of the reconfigurable logic.

[3] This will be further explained in Section 3.3.

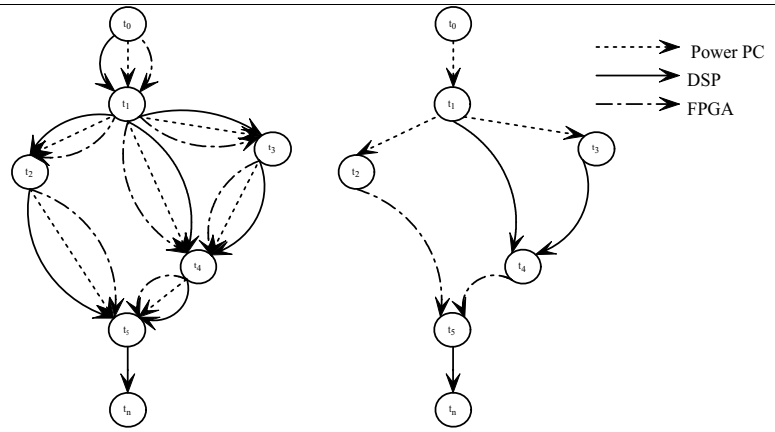

Fig. 1. ATG for 3-way Partitioning.

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 Fig. 1(a). In this partitioning, task 1, 2, and 3 are allocated onto the PowerPC, 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.

To make our model complete, a *dot* operation is defined, which is a bivariate function between a task and a resource:

$$f_{ik} = t_i \bullet r_k, \forall t_i \in T, \forall r_k \in R \tag{3}$$

It provides a local cost estimation for assigning task t_i to resource r_k. Assuming we are only concerned with the execution time and hardware area in our partitioning, we can let f_{ik} be a two item tuple, i.e.

$$f_{ik} = t_i \bullet r_k = \{time_{ik}, area_{ik}\} \tag{4}$$

Obviously, other items, such as power consumption estimation, can be easily added if they are considered. The dot operation can be viewed as an abstraction of the work performed by the cost estimator.

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[4] stochastic decision making process that combines local and global heuristics during the searching process. 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 indicting 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:

$$\tau_{ijk} \leftarrow (1-\rho)\tau_{ijk} + \sum_{l=1}^{m} \Delta\tau_{ijk}^{(l)} \tag{5}$$

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

$$\Delta\tau_{ijk}^{(l)} = \begin{cases} Q/time_{P^{(l)}} & \text{if } e'_{ijk} \in P^{(l)} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

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

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

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 describe how an individual ant "crawls" over the ATG and generates a solution. Two problems must be addressed in this step:

1. How does the ant handle the precedence constraints between task nodes?
2. What are the global and local heuristics and how can they be applied?
3. Finally, how does the ant guarantee to find a feasible partition for the given application?

To answer these questions, 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 ensure that every predecessor node is visited before we visit the current node and that every incoming edge to the current node has been evaluated. We can see later that by enforcing this ordering, we not only make sure that the found partition could be executed correctly but also provide an important prepare for the ant to make resource allocation decision upon entering a new task node.

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 to be assigned to resource r_k according to the probability:

$$\mathsf{p}_{ijk} = \frac{\tau_{ijk}^{\alpha}\eta_{jk}^{\beta}}{\sum_{l=1}^{r}\tau_{ijl}^{\alpha}\eta_{jl}^{\beta}} \quad (7)$$

Here α and β are parameters to control the relative influence of the distributed global heuristic τ_{ijk} and local heuristic η_{jk} 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 as η_{jk}. We focus on achieving the optimal execution time subject to hardware area constraint, therefore a simple weighted combination is used to estimate the cost:

$$cost_{jk} = w_t \cdot time_{jk} + w_a \cdot area_{jk} \quad (8)$$

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. 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. Again $time_{jk}$ and $area_{jk}$ are obtained via the dot operation explained above in Section 3.2. Based on the proposed ATG model, by altering the dot operation, one can easily adapt the cost function to consider other constraints such as power consumption limit, while keep the algorithm essentially intact.

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. Recall that the ant travels the ATG in a topologically sorted manner, it is guaranteed that those guesses are already made. Different strategies can be used on how such allocation decision is made. 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:

$$\mathsf{p}_{jk} = \frac{\text{count of guess } r_k \text{ for } t_j}{\text{count of immediate precedents of } t_j} \quad (9)$$

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

Of course, during the above resource allocation process for the node t_j, it is possible that we encounter the situation where some of the allocation choices become invalid. For example, we may find that the current available FPGA area is not sufficient to hold the realization of t_j. For these cases, we simply reject the invalid resource allocations by making the number of such guesses zero.

Once task node t_j is allocated on resource r_k, it remains unchanged during the current tour for an ant. This ensures that each task is uniquely assigned to one specific resource. Furthermore, we can obtain the cost (such as its execution time and area cost) for t_j on resource r_k by the querying the pre-computed cost information for t_j on r_k using the dot operation discussed previously. In turn, the critical path of the application up to this point will be updated together with the refreshed resource availabilities. By carefully applying all the above measurements, we can guarantee that a partition constructed by the ant is feasible.

As illustrated in Step 5 by the algorithm, at the end of each iteration, the pheromone trails on the edges are

updated according to Eqs (5) and (6). First, a certain amount of pheromone is evaporated. From an 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 shorter execution time for the task graph. The given updating policy is similar to that reported in [15]. Notice here that every ant will contribute to the pheromone update independently based on the quality of the partition it finds. Alternative reinforcement methods [7] can also be applied here. For example, we explored the strategy of updating the pheromone trails on the edges that are included only in the best tour amongst all the returned partitions at each iteration, and we observed no noticeable difference regarding to the quality of the final results.

Finally, each run of the algorithm is composed of multiple iterations of the above steps. Two ending possible stopping conditions are: 1) the algorithm ends after a fixed number of iterations, or 2) the algorithm ends when there is no improvement found after a number of iterations. In the same run, the global pheromone trails τ_{ijk} are initialized once as indicated in step 1 at the start of the algorithm, updated at the end of each iteration, and inherited by the next iteration. The best partition found so far by the ants is also updated dynamically at the end of each iteration and reported as the final result of the run. Because of the stochastic nature of the algorithm, multiple runs can be conducted and may provide different results. Another reason to have multiple runs is to test the stability of the proposed algorithm in achieving high-quality results, as we will discuss in Section 5. For our experiments reported in this paper, each run is independent and is started from scratch without using any result obtained in previous runs.

3.4. Complexity analysis

The space complexity of the proposed algorithm is bounded by the complexity of the ATG, namely $\mathrm{O}(rN^2)$, where N is the number of nodes in the task graph.

For each iteration, each ant has a run time Ant_t confined by $\mathrm{O}(rN^2)$. For a run with I iterations using m ants, the time complexity of the proposed algorithm is $(Ant_t + E_t) * m * I$, where E_t is the evaluation time for each generated partitioning. In the practical situation, $E_t \gg Ant_t$. Comparing with brute force search which has a total run time of $(r^N) * E_t$, the speedup ratio we can achieve is:

$$\mathrm{speedup} = \frac{(\mathrm{r^N}) * \mathrm{E_t}}{\mathrm{m} * \mathrm{I} * (\mathrm{Ant_t} + \mathrm{E_t})} \approx \frac{\mathrm{r^N}}{\mathrm{m} * \mathrm{I}} \quad (10)$$

The number of ants in each iteration m depends on the problem that is being solved by the ACO algorithm. For the TSP problem, the authors assigned m to be a constant multiple of total number of nodes in the TSP problem instance [14]. For the multiway partitioning problem based on the ATG, we propose two possible ways to determine the ant number: 1) based on the average branching factor of the original task graph G; or 2) the maximum branch number of the original task graph G.

3.5. Extending the ACO/ATG method

Besides the ability to adjust itself as the number of computing resource numbers in the system varies, the ACO/ATG method can be easily extended to fit different system requirements. Here we will discuss a few possible ways for some commonly encountered design scenarios.

During system design phase, it is common that certain computational tasks are predetermined or preferred to run on certain resources. That is for each task $t_i \in T$, it is associated with a probability set $\{p_i^1, \ldots, p_i^r\}$ where r is the size of R. Among the elements of the set, some of them can be zero when the corresponding resources have been determined to be not suitable for the given task. By modifying the decision strategy in Eq. (7), we can easily accommodate this requirement by using the following equation:

$$\mathsf{p}_{ijk} = \frac{p_i^k \tau_{ijk}^\alpha \eta_{jk}^\beta}{\sum_{l=1}^r p_i^l \tau_{ijl}^\alpha \eta_{jl}^\beta} \quad (11)$$

Similar to the above approach, other task dependent information, such as profiling statistics can also be considered. In this case, the probability distribution set is associated with the augmented edges in the ATG, instead with the resources. That is for each edge e'_{ijk} defined in Eq. (2), there exists a frequency probability value p_{ijk}, which satisfies the following conditions:

$$\begin{cases} p_{ijk} = p_{i'j'k} & \text{if } i = i' \text{ and } j = j' \\ \sum p_{ijl} = 1 & \text{where } l = 1, \ldots, r \end{cases} \quad (12)$$

Using the two approaches discussed here, one can further modify the proposed algorithm to handle more complicated system features, such as different communication channels, where each channel has a different bandwidth and latency. These channels can either be as-

sociated with the augmented edges if they are bounded with the hardware realization, or may be treated as a task related attribute if the task can only use one certain type channel.

Finally, by altering the definition of the *dot* operation in Eq. (3), better local cost estimation model can be introduced and integrated as the local heuristics. Similarly, different target objective functions for defining the global heuristic η in Eq. (7) can be applied. For example, power consumption can be aggregated as part of the consideration during the process.

3.6. Comparing with the original ACO

In this section, we will summarize the proposed algorithm by comparing it with the original ACO approach proposed in [14].

Perhaps the most fundamental contrast between our work and the original ACO reported in [14] is that they try to solve different domain problems. Though the ACO approach is known as a meta-heuristic method for addressing optimization problems, one still needs to form specific strategies in order to effectively utilize the domain specific characteristics for the problem in hand. To our best knowledge, the method we proposed here is the first approach in the literatures for solving application partitioning problem using the ACO heuristics. Comparing with the TSP problem that the original ACO algorithm was set to address, the application partitioning problem poses specific issues in formulating the ACO algorithm, even though both of them are NP-complete.

First, there is a need to develop an appropriate graph model in formulating an ACO method for the application partitioning problem such that the global and local heuristics could be meaningfully fitted in. As discussed above, the ATG model is introduced in our work as the answer, where the extended edges provide suitable attaching points for the global and local heuristics. In contrast, the modeling issue is relatively easier for the TSP problem since the connection graph of the problem is readily used as the model.

Secondly, a different solution construction strategy has to be developed in our work for individual ant to come up with its partition result. In the original ACO method for the TSP problem, this issue is also relatively trivial as the connection between different cities are undirectional and there is no specific constraint on the ordering of how the cities are visited. However, in the application partitioning problem, to guarantee the correctness of the application, stringent dependencies between tasks have to be respected. In our formulation, a topological sorted ordering is used for individual ant to transverse the ATG. This also has fundamental impact on how the partitioning decision is made for a task node when it is visited.

Local heuristic definition is by nature problem dependent in the ACO framework and has to be formulated in a domain specific manner. In the original ACO method for the TSP problem, it is straightforward to select the distance between two cities as the local heuristic. In our work, we use a weighted combination for this purpose since multiple considerations are involved in defining the cost of mapping certain task onto a resource.

Finally, in our work, we propose a decision making process that is different from that in the original ACO method. In the TSP problem, the only decision to make is to which city the ant shall move to while constructing the Hamilton tour. However, in the application partitioning problem, we have to visit all the child nodes in the ATG in a sorted order. Furthermore, when a task node is visited, we need to make decision on which computing resource it shall be mapped to. As discussed earlier, in our algorithm, a two step decision making process is adopted. First, at each node, the ant makes a "guess" for each immediate child node on how it should be mapped based on the global and local heuristics associated with these nodes. The final decision is delayed until the child node is visited by the ant and the partitioning for the node is done using yet another probabilistic approach over the previous "guesses", such as the one indicated by Eq. (9).

4. Target architecture and benchmarks

Our experiments address the partitioning of multimedia applications onto a programmable, multiprocessor system platform. The target architecture contains one general purpose hard processor core, a soft DSP core, and one programmable core (see Fig. 2).

This model is similar to the Xilinx Virtex II Pro Platform FPGA [45], 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

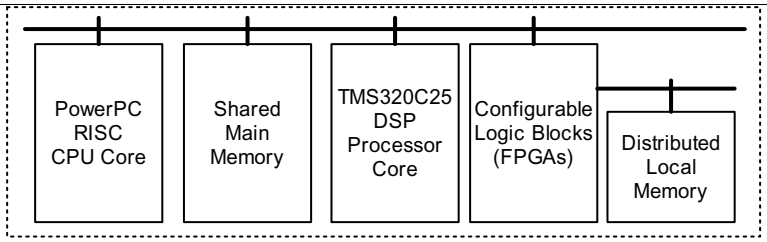

Fig. 2. Target architecture.

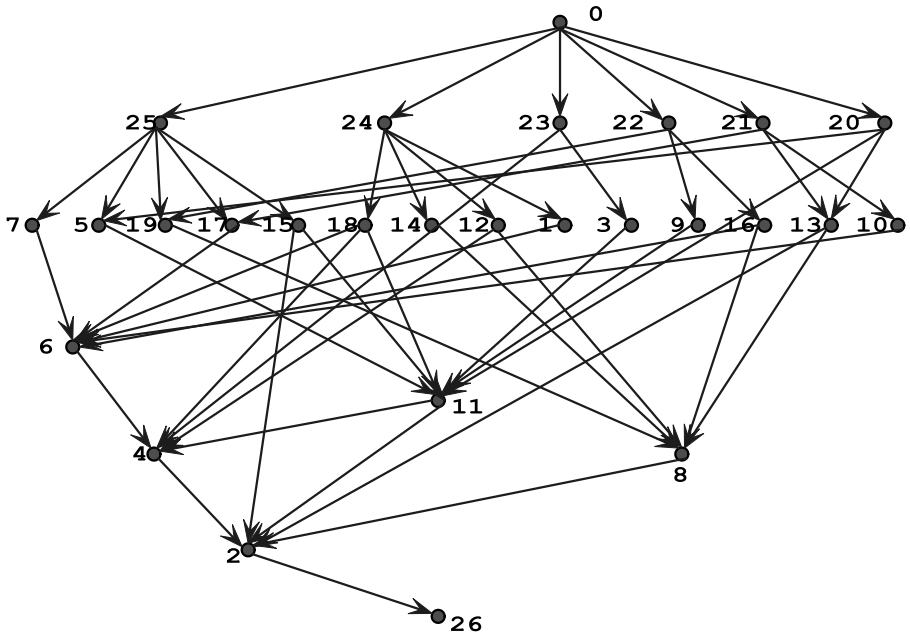

Fig. 3. Example Task Graph.

(soft core) of the TMS320C25 DSP processor core [9]. 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 pre-computed. 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 be-

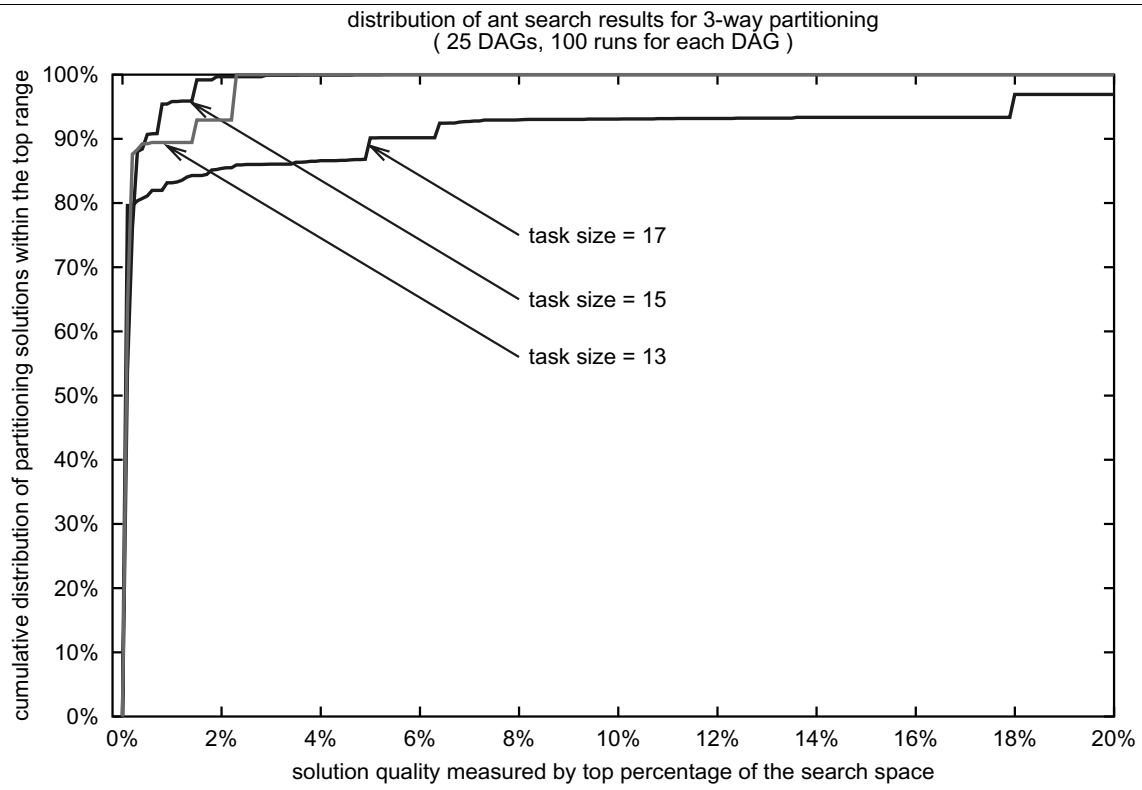

Fig. 4. Result quality measured by top percentage.

tween 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 [3] and Machine SUIF [39] 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 the Virtex II Pro Platform FPGA [45] and the DSP processor core [9], 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 [30]. Each testing example is formed via a two step process that combines a randomly generated DAG with real life software functions. The testing benchmarks are available online http://express.ece.ucsb.edu. 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. Task graphs are created using GVF tool kit [33]. With this tool, we are able to control the complexity of the generated DAGs by specifying the total number of nodes or the average branching factor in the graph. Figure 3 gives a typical example for the task graph we used in our study.

Table 1
Comparing ACO results with the random sampling*

Testcase	Optimal Execution Time	Total # Optimal Partitions	Random Sampling Prob.	Optimal # ACO Runs
DAG-1	23991	2187	29.05	100
DAG-2	11507	1215	17.35	100
DAG-3	13941	2187	29.05	100
DAG-4	60120	1664	22.98	3
DAG-5	23004	729	10.80	100
DAG-6	12174	81	1.26	100
DAG-7	26708	2187	29.05	100
DAG-8	51227	486	7.34	71
DAG-9	11449	1458	20.45	100
DAG-10	140197	1024	14.84	0
DAG-11	138387	1215	17.35	98
DAG-12	10810	243	3.74	100
DAG-13	33193	2187	29.05	100
DAG-14	16460	81	1.26	100
DAG-15	30919	1215	17.35	100
DAG-16	49910	1856	25.26	92
DAG-17	22934	135	2.09	100
DAG-18	47161	243	3.74	100
DAG-19	152088	1024	14.84	2
DAG-20	6157	27	0.42	97
DAG-21	29877	610	9.12	100
DAG-22	14141	729	10.80	100
DAG-23	15718	2187	29.05	100
DAG-24	9905	108	1.68	100
DAG-25	48141	486	7.34	98

*100 ACO runs on 25 testing task graphs with size 13.

5. Experimental results and performance analysis

5.1. Absolute quality assessment

It is possible to achieve definitive quality assessment for the proposed algorithm on small task graphs. 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. By conducting thorough evaluation on the search space, we obtain important insights to the search space, such as the optimal partitions with minimal execution time and the distribution of all the feasible partitions. More, the brute force results can be used to quantify the hardness of the testing instances, i.e. by computing the theoretical expectation for performing random sampling on the search space. Trivial examples, for which the number of the optimal partitions is statistically significant, are eliminated in our experiments to ensure that we are targeting the *hard* instances.

We give 100 runs 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 4 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.

With the definitive description on the search space obtained from the brute force search, we can also evaluate the capability of the algorithm with regard to dis-

covering the optimal partition. Table 1 shows a comparison between the proposed algorithm and random sampling when the task graph size is 13. The first column gives the testing case index. The second and third columns are the optimal execution time and the number of partitions that achieve this execution time for the testcase, respectively. This information is obtained through the brute force search. The fourth column gives the derived theoretical possibility of finding an optimal partition in 250 tries over a search space with a size of $3^{13} = 1,594,323$ if random sampling is applied. The last column is the number of times we found an optimal partition in the 100 runs of the ACO algorithm. It can be seen that over $2,500$ runs across the 25 testcases, we 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%. With the same amount of computation time, random sampling method has a 14.21% chance of discovering the optimal solution. Therefore, our ACO algorithm is statistically 6 times more effective in finding the optimal solution than random sampling. 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.

There exist three testcases (DAG-4, DAG-10, and DAG-10) for which the proposed algorithm only finds the optimal solution in few times among the 100 runs. Further analysis of the results shows that all the solutions returned for these testing samples are within the top 3% of the solution space.

Figure 5 provides another perspective regarding to the quality of our results. In this figure, the x axis is the percentage difference comparing the execution time of the partition found by the ACO algorithm with respect to the optimal execution time. The y axis is the percentage of the solutions that fall in that range.

These results may seem somewhat conflicting with the results shown in Fig. 4. The results in Fig. 4 show the results on how the ACO algorithm finds solutions that are within a top percentage of overall solutions. This graph shows the solution quality found by ACO. The results differ because while the ACO algorithm may not find the optimal solution, it almost always finds the next best feasible solution. However, the quality the next feasible solution in terms of execution time may not necessarily be close to the optimal solution. We believe that this has more to do with the solution distribution of the benchmarks than the quality of the algorithm.

For example, larger benchmarks are more likely to have more solutions whose quality is close to optimal. If this is the case, the ACO algorithm will likely find a good solution with a good solution quality as is show in Fig. 4.

Regardless, the quality of the solutions that we find are still very good. The majority (close to 90%) of our results are within the range of less than 10% worse compared with the optimal execution time.

Based on the discussion in Section 3, when the ant number is 5 and iteration number is 50, for a three way partitioning problem over a 13 node task graph, the proposed algorithm has a theoretical execution time about 0.015% of that using brute force search, or 6,300 times faster. The experiments were conducted on a Linux machine with a 2.80 GHz Intel Pentium IV CPU with 512 MByte memory. The average actual execution time for the brute force method is 9.1 minutes while, on average, our ACO algorithm runs for 0.072 seconds. These runtimes are in scale with the theoretical speedup report in Section 3.4. To summarize the experiment results, with a high probability (88.12%), we can expect to achieve a result within top 1% of the search space with a very minor computational cost.

5.2. Comparing with Simulated Annealing

In order to further investigate the quality of the proposed algorithm, we compared the results of the proposed ACO algorithm with that of the simulated annealing (SA) approach.

Our SA implementation is similar to the one reported in [44]. To begin the SA search, we randomly pick a feasible partition that obeys the cost constraint as the initial solution. The neighborhood of a solution contains all the feasible partitions that can be achieved by switching one of the tasks to a different computing resource from the one it is currently mapped to. The feasibility of the neighbors is computed in a similar way as in our ACO implementation. At every iteration of the SA search, a neighbor is randomly selected and the cost difference (i.e. execution time of the DAG) between the current solution and the neighboring solution is calculated. The acceptance of a more costly neighboring solution is then determined by applying the Boltzmann probability criteria [1], which depends on the cost difference and the annealing temperature. In our experiments, the most commonly known and used geometric cooling schedule [44] is applied and

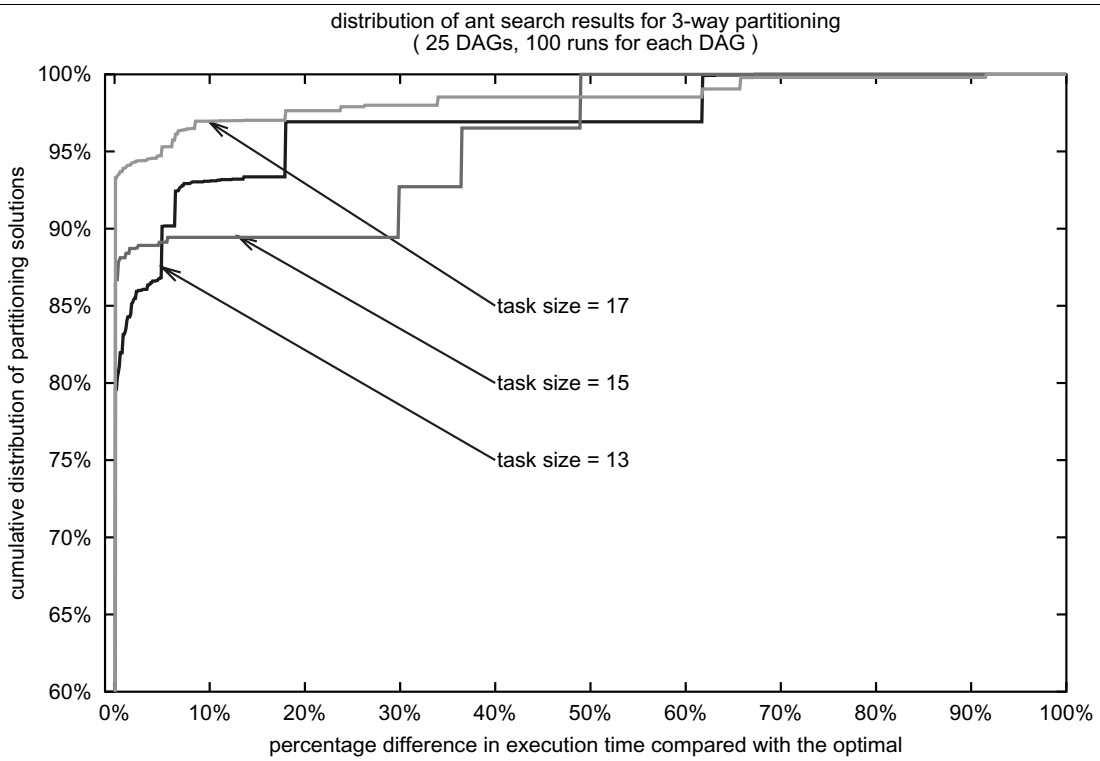

Fig. 5. Execution time distribution.

the temperature decrement factor is set to 0.9. When it reaches the pre-defined maximum iteration number or the stop temperature, the best solution found by SA is reported.

Because of the stochastic nature of the SA algorithm, for a given cooling approach, the more the iterations the better chance for SA to find higher quality results. However, as the iteration number increases, its execution time becomes longer. Figure 6 compares the ACO results against those that achieved by the SA search sessions. The graph is illustrated in the same way as Fig. 4. The SA sessions are configured in the same way except with different iteration numbers. Here SA50 has roughly the same execution time of our ACO implementation, 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. In other words, in order to obtain comparable partition quality, SA suffers from much longer execution time. Furthermore, in order to compare the stability of the two different approaches, we also compared the variance of the results returned respectively by the SA and the proposed algorithm. The is done by carrying multiple runs of ACO and SA independently. This comparison indicates that the ACO approach consistently provides significantly more stable results than SA. For some testing cases, the variance on the SA results can be more than 3 times wider. Thus experimentally we perhaps can conclude that the ACO approach would have much better chance in obtaining high quality results than the SA method with the same execution cost.

Another benefit of conducting comparison between SA and ACO is that it provides a way for us to assess the quality of the proposed algorithm on bigger size testing cases. For such problems, it becomes impossible for us to perform the brute force search to find the true optimal solution for the problem. However, we can still assess the quality of the proposed algorithm by comparing relative difference between its results with that obtained by using other popularly used heuristic methods, such as SA. Figure 7 shows the cumulative result quality distribution curves for task graphs with 25 nodes. For these problems, it is estimated that the brute force method would take hundreds machine hours thus impractical for us to find the optimal exactly. In the figure, the x axis now reads as the percentage difference on the execution time of the partition found by

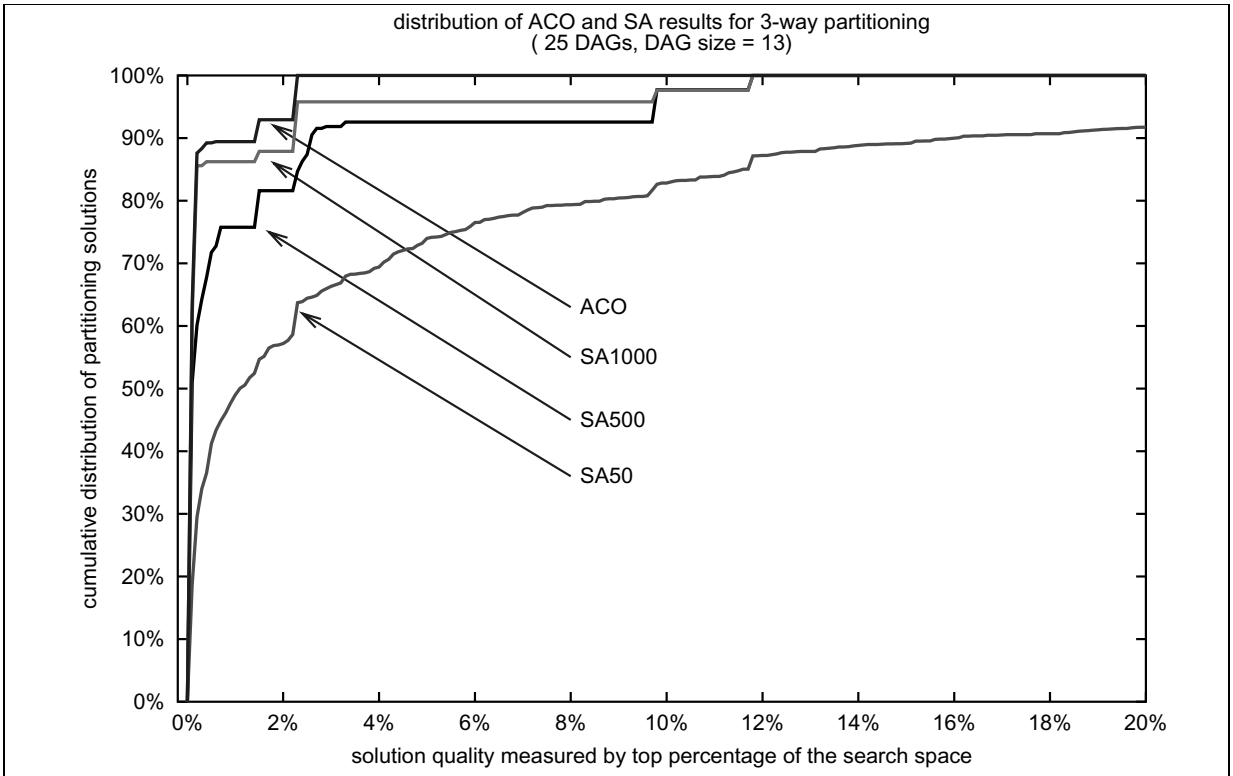

Fig. 6. Comparing ACO with SA.

the corresponding algorithm with respect to the *best* execution time over all the experiments using different approaches. Among them, the ACO and SA500 have the same amount of execution time, while SA5000 runs at about 10 time slower. It is shown that ACO outperforms SA500 while a much more expensive SA works comparably.

5.3. 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 compensates this, SA suffers from lengthy low temperature cooling process.

Another problem of SA, which may be more related with the stability of the quality of the results 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 qualities, 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 [7,13,15,22] 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

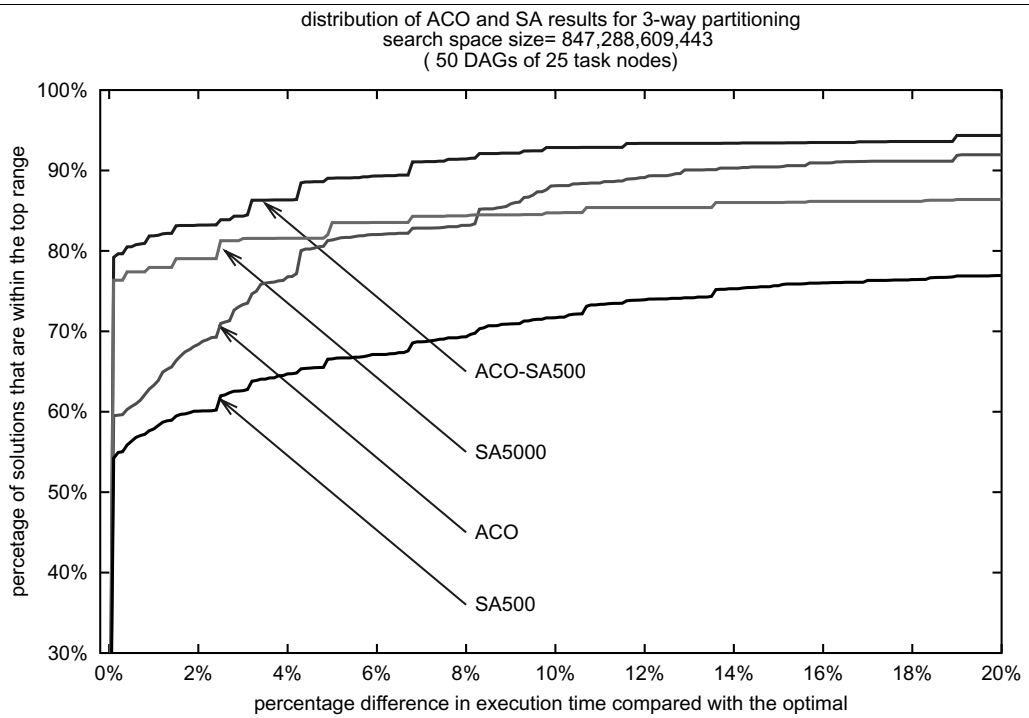

Fig. 7. ACO, SA and ACO-SA on big size problems.

Table 2
Average Result Quality Comparison

(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

the SA algorithm, it is possible for us to achieve even better system performance with a substantially reduced computing cost. In Fig. 7, curve ACO-SA500 shows the result of this approach. It achieves definitively better results comparing with that of SA5000 while only taking about 20% of its running time. Similar result holds for task graphs with bigger sizes, 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 with comparable result quality).

Overall, we summarize the result quality comparison with Table 2 for problems with big sizes. 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.

6. Estimating design parameters with ACO

One possible application of using the proposed ACO approach for application partitioning is to help make high level design choices by estimating design parameters at the early stage. At this point, a critical problem that the system designer faces is to make choice among alternative designs. One common question that the system designer has to answer is whether an extra computing device is needed in the system design.

For instance, considering the following case: assuming one design is realized with a PowerPC and a FPGA component (Architecture 1), while an alternative design contains an extra DSP core (Architecture 2), one needs to quickly evaluate design parameters associated with each of the two possible approaches. Does adding an extra DSP result in FPGA area reduction and if yes, how much can we save? Does the second design provide significant improvement of system's timing performance? Or by having an extra DSP, how much FPGA cost can be saved without tempting the system's time performance requirement? In order to address these

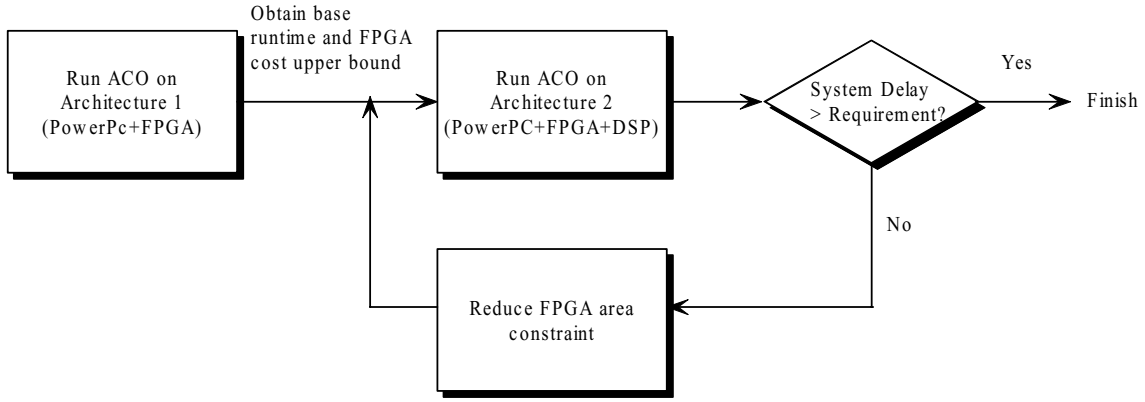

Fig. 8. Estimate Design Parameters with ACO application partitioner on design choice with incremented resources.

questions, quick assessment on related design parameters is needed. Essentially, the above problem request us to provide insights for design parameters when the number of computing resources is incremented. The high quality and fast execution time of the proposed ACO multi-way application partitioning approach provides a possible method for certain situations for such a system level design task.

To see this, we cross examine the results of the proposed algorithm over the testing cases illustrated in Table 1 for the two architectures. Based on the available resources, they can be viewed as 3-way partitioning and bi-partitioning problems under our model respectively and the proposed ACO approach solves them in a uniformed way.

Based on this comparison, we find that with the same hardware area constraint, our algorithm robustly provides partitions with better or at least the same execution time for Architecture 2 for different test cases in our benchmarks. The speedup is dependent on the specific application, i.e. the application's ATG and the tasks associated with it. With our testing cases, we have an average execution time speedup of 1.6% over the 25 testing examples, while over 11% speedup is observed for examples DAG-6 and DAG-17. More interestingly, based on the same test, we find that the 3-way partitioning results have an average 12.01% save in hardware area for the FPGA component compared with the bi-partitioning results. In 100 runs, the expected biggest area save over 25 DAGs is 12.61%, which is roughly in agreement with the average savings.

This motivates us to use the proposed ACO algorithm as a quick estimator for design parameters, such as the FPGA area cost constraint, when a new computing resource is included. The question the designer tries to answer here is: how much FPGA area can we save by adding a DSP core in the system while respecting the system delay constraint? Or what is the right FPGA area cost constraint we should provide for the incremented system? Without a quick design parameter assessment method, this constraint is hard to be made accurately. To address this problem, we propose a two step process using the ACO application partition as such quick estimator, as the process is diagramed in Fig. 8.

First, we notice that Architecture 2, which contains an extra DSP, is expected to not make the FPGA cost worse. Based on this observation, a designer can first conduct bi-partitioning for the application over Architecture 1. The results will provide critical guidance regarding to the time performance and the upper bound of the FPGA area cost. The designer can then use the FPGA area cost result returned by our algorithm as the "desired" constraint for the 3-way partitioning problem over Architecture 2. Of course, this step may require multiple iterations if the optimal FPGA saving is expected. Thanks for the low computing cost of the proposed ACO approach, such iterative process is practical and can be conducted within reasonable time. As shown in Fig. 8, for each of the iterations, we check if the system delay meets the time performance constraint. If yes, it implies that a more stringent area cost constraint can be used. Otherwise, we have found the optimal saving and the process terminates. By applying this method, without noticeable degradation on the execution time (less than 2%), our experiments on the testing cases show that an average hardware area reduction of 65.46% for the 3-way architecture comparing with original design which only uses PowerPC and FPGA.

Notice this is just one of the possible scenarios that the proposed algorithm could help. There are other cases such a quick parameter estimator could be useful. For instance, by simply swapping the boxes associated with Architecture 1 and Architecture 2 in Fig. 8, we can help to solve the reverse design problem, where we try to find how much extra FPGA resource we would need if we simplify the system design by excluding the DSP core from the architecture.

7. Conclusion

In this work, we presented a novel heuristic searching method for the system partitioning problem based on the ACO techniques. Our algorithm proceeds 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, which can be easily extended as the resource number grows and it fits well with a variety of system requirements.

Experimental 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 modestly sized 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 for large size problems, with substantial less execution time, the proposed method achieves better solutions than the popularly used simulated annealing approach. With the observation of the complementary behaviors of the algorithms, we proposed a hybrid approach that combines the ACO and SA together. This method yields even better result than using each of the algorithms individually.

In future work, we plan to further refine the algorithm for more sophisticated testing scenarios, e.g. to handle looping and conditional jump among tasks. Compilation techniques have to be introduced into the algorithm to achieve this. More, this may pave the way to explore the ability of the ACO approach at finer granularity levels, such as basic blocks or even instruction level. It will also be interesting to explore other strategies for the ant decision making process in order to further improve the effectiveness of the algorithm. For example, introducing Tabu heuristic could make the ants more efficient in avoiding bad solutions. Comparing with other heuristic methods mentioned here, ACO is more tightly tied with the topological characteristics of the application. This makes it possible to deeply couple ACO with task scheduling. One direction for doing so is to investigate on how profiling information can be effectively used to guide the algorithm.

References

[1] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatoria Optimization and Neural Computing*, John Wiley & Sons, New York, NY, 1989.

[2] 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.

[3] 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.

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

[5] Altera Corporation, *Nios Embedded Processor System Development*, 2003. http://www.altera.com/products/devices/nios.

[6] 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.

[7] E. Bonabeau, M. Dorigo and G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*, Oxford University Press, New York, NY, 1999.

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

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

[10] D. Costa and A. Hertz, Ants can colour graphs, *Journal of the Operational Research Society* **48** (1996), 295305.

[11] 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.

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

[13] M. Dorigo and L.M. Gambardella, Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem, *IEEE Transactions on Evolutionary Computation* **1**(1) (April 1997), 53–66.

[14] 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) (February 1996), 29–41.

[15] 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) (February 1996), 29–41.

[16] 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) (March 1997), 366–390.

[17] 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.

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

[19] S. Fenet and C. Solnon, *Searching for maximum cliques with ant colony optimization*, 3rd European Workshop on Evolutionary Computation in Combinatorial Optimization, April 2003.

[20] S. Fidanova, *Evolutionary Algorithm for Multiple Knapsack Problem*, In Proceedings of PPSN-VII, Seventh International Conference on Parallel Problem Solving from Nature, Lecture Notes in Computer Science. Springer Verlag, Berlin, Germany, 2002.

[21] 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, London, UK, 1999, 511.

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

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

[24] R.L. Graham, E.L. Lawler, J.K. Lenstra and A.H.G.R. Kan, Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics* **5** (1979), 287–326.

[25] 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.

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

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

[28] 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.

[29] R. Kastner, *Synthesis Techniques and Optimizations for Reconfigurable Systems*, PhD thesis, University of California at Los Angeles, 2002.

[30] 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.

[31] G. Leguizamon and Z. Michalewicz, *A new version of ant system for subset problems*, In Proceedings of the 1999 Congress of Evolutionary Computation, IEEE Press, 1999, 1459–1464.

[32] 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.

[33] G. Melancon and I. Herman, Dag drawing from an information visualization perspective, Technical Report INS-R9915, CWI, November 1999.

[34] R. Michel and M. Middendorf, *New Ideas in Optimization*, chapter An ACO algorithm for the shortest supersequence problem, McGraw Hill, London, UK, 1999, 51–61.

[35] R. Niemann and P. Marewedel, An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming, *Design Automation for Embedded Systems* **2**(2) (March 1997), 125–163.

[36] M. Palesi and T. Givargis, *Multi-Objective Design Space Exploration Using Genetic Algorithms*, In Proceedings of the Tenth International Symposium on Hardware/Software Codesign, 2002.

[37] 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) (August 2002), 321–332.

[38] R. Schoonderwoerd, O. Holland, J. Bruten and L. Rothkrantz, Ant-based load balancing in telecommunications networks, *Adaptive Behavior* **5** (1996), 169–207.

[39] 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.

[40] 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.

[41] 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.

[42] F. Vahid and T.D. LE, Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning, *Design Automation for Embedded Systems* **2**(2) (March 1997), 237–261.

[43] 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* **1**(1) (November 2003), 439–444.

[44] 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) (July 2002), 425–449.

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