Available online at www.sciencedirect.com
SCIENCE DIRECT®
Applied Soft Computing 4 (2004) 303–322
Applied Soft Computing
www.elsevier.com/locate/asoc

FPGA implementation of population-based ant colony optimization

B. Scheuermann [a,*], K. So [b], M. Guntsch [a], M. Middendorf [c], O. Diessel [b], H. ElGindy [b], H. Schmeck [a]

[a] *Institute AIFB, University of Karlsruhe (TH), Germany*
[b] *Computer Science and Engineering, University of New South Wales, Australia*
[c] *Institute of Computer Science, University of Leipzig, Germany*

Received 26 January 2004; accepted 6 March 2004

Abstract

We present a hardware implementation of population-based ant colony optimization (P-ACO) on field-programmable gate arrays (FPGAs). The ant colony optimization meta-heuristic is adopted from the natural foraging behavior of real ants and has been used to find good solutions to a wide spectrum of combinatorial optimization problems. We describe the P-ACO algorithm and present a circuit architecture that facilitates efficient FPGA implementations. The proposed design shows modest space requirements but leads to a significant reduction in runtime over software-based solutions. Several modifications and extensions of the basic algorithm are also presented, including the approximation of the heuristic function by a small, dynamically changing set of favorable decisions.

Keywords: Ant colony optimization; Ant algorithm; Field-programmable gate array; FPGA

1. Introduction

Natural evolution has yielded biological systems in which complex collective behavior emerges from the local interaction of simple components. One example where this phenomenon can be observed is the foraging behavior of ant colonies [1,2]. Ant colonies are capable of finding shortest paths between their nest and food sources. This complex behavior of the colony is possible because the ants communicate indirectly by disposing traces of pheromone as they walk along a chosen path. Following ants most likely prefer those paths possessing the strongest pheromone information, thereby refreshing or further increasing the respective amounts of pheromone. Since ants on short paths are quicker, pheromone traces on these paths are increased very frequently. On the other hand, pheromone information is permanently reduced by evaporation, which diminishes the influence of formerly chosen unfavorable paths. This combination focuses the search process on short, favorable paths.

Inspired by this biological paradigm, Dorigo et al. [3–5] introduced a metaheuristic known as ant colony optimization (ACO). In ACO, a set of artificial ants searches for good solutions for the optimization problem under consideration. Each ant constructs a solution by making a sequence of local decisions. Its decisions are guided by pheromone information and

* Corresponding author. Tel.: +49-721-608-3924; fax: +49-721-693717.
E-mail address: scheuermann@aifb.uni-karlsruhe.de (B. Scheuermann).

1568-4946/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.asoc.2004.03.008

some additional heuristic information (if applicable). After a number of ants have constructed solutions, the best ants are allowed to update the pheromone information along their path through the decision graph. Evaporation is accomplished by globally reducing the pheromone information by a certain percentage. This process is repeated iteratively until a stopping criterion is met. ACO has shown good performance on several combinatorial optimization problems, including scheduling [6], vehicle routing [7], constraint satisfaction [8], and the quadratic assignment problem [9].

Parallel versions of ACO algorithms have been studied by several authors [10–18]. Most of these authors use the parallel approach simply to gain speedup by performing the solution construction of the ants in parallel without any significant changes to the solution construction and pheromone update mechanisms [10–13,16]. More interesting parallel, multi-colony ACO algorithms, in which every processor holds a colony of ants with their own pheromone information and where cooperation between the colonies is done by the exchange of good solutions, have been studied in [14,17,18]. A different and more fine-grained parallel ACO approach that is suitable for large processor arrays (RMesh model) has been proposed in [15]. The RMesh is a standard model for reconfigurable processor arrays, in which the processors are connected by a dynamically reconfigurable bus system [19]. The RMesh efficiently supports algorithmic tasks that are typical of ACO algorithms such as bit-summation and finding the rank of a number in a set.

In this paper, we adapt an ACO algorithm to field-programmable gate arrays (FPGAs). To the best of our knowledge, this is the first implementation of ACO on commercially available FPGA devices. This implementation is based on design-concepts introduced by the authors in [20]. FPGAs are used for a wide range of applications, e.g. network communication [21], video communication and processing [22,23] and cryptographic applications [24]. It has been shown that FPGAs are suitable for the implementation of soft computing techniques like Neural Networks [25–27] and Genetic Algorithms [28–30]. We show that ACO can also be implemented on FPGAs, leading to significant speedups in runtime compared to implementations in software on sequential machines.

We demonstrate that a straightforward hardware mapping of the standard ACO algorithm is not very well suited to implementation on the resources provided by current commercial FPGA architectures. Instead we suggest using the Population-based ACO (P-ACO), in which pheromone information is replaced by a small set (population) of good solutions discovered during the preceding iterations [31]. Accordingly, the combination of pheromone updates and evaporation has been replaced by inserting a new good solution into the population, replacing the oldest solution from the population. Experimental results indicate that P-ACO performs at least as well as the standard ACO approach [31]. We show that asymptotically the proposed hardware implementation of the Population-based ACO allows for essential reductions in hardware resources needed in comparison to the standard ACO algorithm.

The remainder of this paper is organized as follows. Section 2 provides a brief introduction to FPGA technology. Sections 3 and 4 describe the standard, respectively the Population-based ACO algorithms and discuss their characteristics with respect to an FPGA implementation. The conversion of the P-ACO algorithm and the mapping into FPGA hardware is then described in Section 5. Experimental studies in Section 6 gauge the performance of the presented design, followed by several interesting modifications and extensions in Section 7. Finally, in Section 8 we give a conclusion and an outline of our future work.

2. Field-programmable gate arrays

A variety of devices is currently available for developing and implementing digital systems. Usually, a designer can choose from a wealth of software-oriented devices like general-purpose-processors, micro-controllers, digital signal processors, or application-specific instruction set processors (ASIPs). On the other hand, hardware devices—so-called application-specific integrated circuits (ASICs)—are available, which are designed for predefined processing tasks.

By providing programmable selection of alternate logic and routing structures, field-programmable gate arrays can be considered to be located at the intersection of software and hardware-oriented systems. Using modern design software, circuits can be designed and implemented very rapidly, thereby avoiding the up-front cost of designing custom circuits and

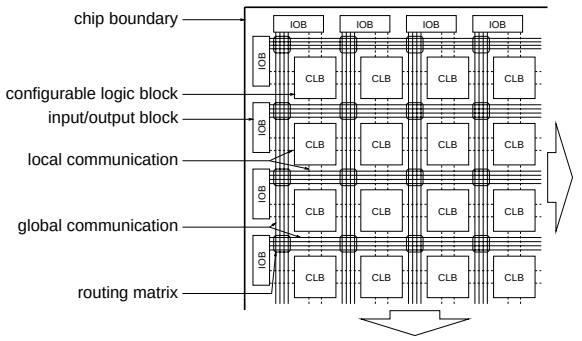

Fig. 1. Simplified schematic of an FPGA, top-left corner.

providing a quick means of correcting design errors. After configuring the circuit onto the FPGA chip, it is switched into operational mode, upon which the inherent parallelism and pipelined design style can offer considerable speedup over instruction stream processors. FPGAs facilitate system development and allow easy and quick design changes and verification. Not only are they suitable for rapid prototyping, they can also substitute for standard logic and gate array solutions in small and medium volume productions. However, their high level of flexibility demands additional switches and routing, which in turn increase circuit delays and chip area relative to custom fabricated circuits.

Fig. 1 depicts a simplified representation of an FPGA comprising the three major configurable elements commonly present:

- Configurable logic blocks (CLBs) provide the basic functional components for implementing logic and registers.
- Input/output blocks (IOBs) form interfaces between the routing network and package pins.
- Routing network consisting of horizontal and vertical multi-track channels and configurable switches allow logic blocks to be interconnected to form complex computational structures.

Generally, CLBs consist of two or three look-up tables (LUTs) and two flip-flops. Any LUT can be configured to either compute an arbitrary boolean function of three or four input signals, or they can be used as a small RAM providing storage for up to 16 bits. Depending on the respective device, additional dedicated storage elements, carry logic and multiplication circuits or even complete RISC micro-processors might be embedded into the FPGA chip area [32–34].

Horizontal and vertical communication resources provide configurable connections among CLBs or between CLBs and IOBs. Global busses of different length connect components across longer distances, whereas local busses allow for a fast communication between direct neighbours. User-defined linkages between bus lines can be established either within routing matrices or at specific programmable interconnect points (omitted in Fig. 1).

3. Standard ant colony optimization

In this section, the standard ACO approach that has been followed by most ACO algorithms so far is introduced. We describe the generic decision process, which is then exemplified by means of three combinatorial optimization problems. Afterwards, we discuss some relevant aspects when mapping the ACO algorithm onto an FPGA.

3.1. Description

The objective of ACO is to find good solutions for a given combinatorial optimization problem [3–5]. The problems considered usually allow solutions to be expressed as a permutation π of n given items. The pheromone information is encoded in an $n \times n$ pheromone matrix $[\tau_{ij}]$. Depending on the problem the pheromone value τ_{ij} expresses the desirability to assign item j to place i of the permutation (place $\times$ item coding) or the desirability to position item j immediately after item i in the permutation (item $\times$ item coding).

Three examples for combinatorial optimization problems and the corresponding types of pheromone encoding are given below:

- For the single machine total tardiness problem (SMTTP) [35], n given jobs have to be scheduled onto a single machine. For every job j, its deadline d_j and the processing time p_j are given. If C_j denotes the completion time of job j in a schedule, then $L_j = C_j - d_j$ defines its lateness and $T_j = \max(0, L_j)$ its tardiness. The objective is to find a schedule minimizing the total tardiness of all

jobs $\sum_{j=0}^{n-1} T_j$. Since the relative position of a job in the schedule is more important than its predecessor or successor in the schedule a place × item pheromone matrix is used.

- For the quadratic assignment problem (QAP), n facilities, n locations, and two $n \times n$ matrices $[d_{ij}]$ and $[f_{hl}]$ are given, where d_{ij} is the distance between locations i and j and f_{hl} is the flow between facilities h and l. The goal is to find an assignment of facilities to locations, i.e. a permutation π of $[0, n-1]$, such that the sum of distance-weighted flows between facilities $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} d_{\pi(i)\pi(j)} f_{ij}$ is minimized. Solutions are constructed by successively assigning facilities to places (locations) in the permutation, which means that, like for SMTTP, a place × item pheromone matrix is used.
- For the traveling salesperson problem (TSP), n cities are given with distances d_{ij} between cities i and j for $i; j \in [0, n-1]$. The goal is to find a distance minimal Hamiltonian cycle, i.e. a mono-cyclic permutation π of [0: $n-1$] which minimizes $\sum_{i=0}^{n-1} d_{i\pi(i)}$. Since the neighborhood of cities in the permutation is important for this problem an item × item pheromone matrix is used.

Typically, when constructing a solution, ants do not rely solely on pheromone information, but also have access to some heuristic information η_{ij}, which signifies the immediate impact that a local decision might have on solution quality. For example, in TSP the heuristic value of choosing to visit city j after the last chosen city i is considered to be inversely proportional to the distance separating them, $\eta_{ij} = 1 = d_{ij}$.

The standard ACO algorithm (see Fig. 2) starts by initializing the pheromone matrix, setting every pheromone entry to an initial value $\tau_{\text{init}} > 0$. For problems with an item × item encoded pheromone matrix, e.g. TSP, the pheromone entries on the diagonal are set to 0, since no city can be its own successor/predecessor. In every iteration of the algorithm, m ants generate solutions $\pi_0, \ldots, \pi_{m-1}$. An ant builds a solution by making a sequence of local decisions, i.e. successive selections of items. Every decision is made randomly according to a probability distribution over the so far unchosen items in selection set S:

$$\forall_j \in S : p_{ij} = \frac{\tau_{ij}^{\alpha}\eta_{ij}^{\beta}}{\sum_{z\in S}\tau_{iz}^{\alpha}\eta_{iz}^{\beta}} \quad (1)$$

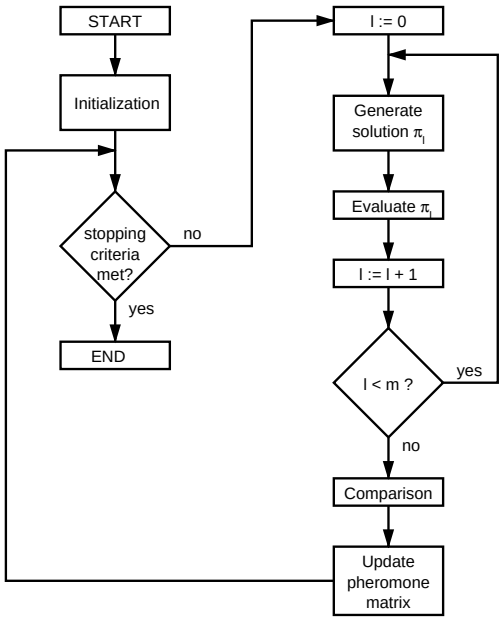

Fig. 2. ACO processing flow.

where parameters α and β are used to determine the relative influence of pheromone values and heuristic values. Initially, the selection set S contains all items; after each decision, the selected item is removed from S. Every solution is evaluated according to the respective objective function. After m solutions have been generated, the solution qualities are compared to determine the best solution π^* of the current iteration. The pheromone matrix is then updated in two steps:

(1) *Evaporation*: All pheromone values in the matrix are reduced by a relative amount: $\forall i, j \in [0, n-1] : \tau_{ij} \mapsto (1-\rho)\tau_{ij}$.
(2) *Intensification*: The pheromone values along the best solution π^* are increased by an absolute amount: $\forall i \in [0, n-1] : \tau_{i\pi^*(i)} \mapsto \tau_{i\pi^*(i)} + \Delta$.

Note that in some variants of ACO (see [5] for an overview), not only the best solution π^*, but the $\bar{m} < m$ best solutions of the iteration are allowed to increment pheromone values. Other approaches keep track of the best solution π^e found so far, called the elitist solution, and the pheromone update is also performed according to this solution.

The ACO algorithm executes a number of iterations until a specified stopping criterion has been met, e.g. a predefined maximum number of iterations has been

executed, a specific level of solution quality has been reached, or the best solution has not changed over a certain number of iterations.

3.2. Problems mapping standard ACO to FPGA

When mapping ACO to hardware, we have to consider the restrictions set by the target architecture. These restrictions concern the number, the type, and the distribution of available computational, memory and I/O resources.

A straightforward approach would be to directly map the pheromone matrix onto the FPGA. This design comprises all processing and memory resources for each element of the statically allocated matrix. Ants could then be piped through the matrix in a systolic fashion as proposed in [15]. Each ant then occupies the row of the matrix that corresponds to its actual decision. Obviously, this approach would result in space requirements that increase quadratically with problem size n.

When developing an ACO design for FPGAs, we encounter several restrictions that make a hardware realization difficult:

- Pheromone values and the random numbers used require a floating point representation. Such a representation does not lend itself to a fine-grained programmable logic implementation.
- Evaporation, and the integration of heuristic information, requires multiplication operations. However, the computational resources available on an FPGA do not efficiently support the implementation of multiplication circuits. Only a few FPGAs have dedicated multiplication blocks and these are restricted in size.
- To make a selection according to the probability distribution p_{ij}, we have to calculate prefix sums of the products in the numerator over the as yet unchosen items in the selection set S (refer to Eq. (1)). Therefore, n circuits for prefix sum calculations have to be provided, one circuit per row in the pheromone matrix. The required space and time complexity is prohibitive even for the comparatively large present day programmable gate arrays.

For these reasons we suggest implementing the alternative P-ACO approach introduced in the following section.

4. Population-based ant colony optimization

In this section, we explain the P-ACO approach (see [31]) and describe the differences to standard ACO. Furthermore, we discuss the advantages of PACO with respect to an FPGA implementation.

4.1. Description

One important aspect of P-ACO which makes it interesting for a hardware-based implementation is that it transfers less and only the most important information from one iteration of the algorithm to the next. Instead of a complete pheromone matrix as in ACO, P-ACO transfers a small population P of the k best solutions that have been found in past iterations.

Since each solution is a permutation of the n items, the population can be stored in an $n \times k$ matrix $Q = [q_{ij}]$, where each column of Q contains one solution. This matrix is called the population matrix. It contains the best solution of each of the preceding k iterations. When employing an elitism strategy, column $j = 0$ contains the best solution found so far by the algorithm in all iterations, and the remaining columns $j = 1, \ldots, k-1$ contain the respective best solution of each of the preceding $k-1$ iterations. Unless noted otherwise, we employ an elitism strategy.

After an iteration of ants has constructed solutions, instead of updating a pheromone matrix, P-ACO updates the population. The best solution of the current iteration is added, and it replaces the oldest solution in P, if P already contains k solutions, i.e. the population size remains $|P| = k$. This means that one column $j \in \{1, \ldots, k-1\}$ in the population matrix Q has to be changed. Additionally, if this solution is also better than the elitist solution, then column $j = 0$ is also overwritten by the new solution. Initially the population is empty. Thus, during the first $k-1$ iterations the population has to be filled, which is done by inserting the best solution of the respective iteration without removing an older solution.

Note that columns $j \in \{1, \ldots, k-1\}$ of the population matrix Q are maintained like a FIFO-queue. Hence, each solution in this queue has an influence on the decisions of the ants over exactly $k-1$ subsequent iterations. Other schemes for deciding which solutions should enter/leave the population are discussed in [36].

In P-ACO, the pheromone matrix (τ_{ij}) that is used by ants for solution construction is determined by the population matrix. Each pheromone value is set to the initial value τ_{init} and is increased, if there are corresponding solutions in the population:

$$\forall i, j \in [0, n-1]; \tau_{ij} \mapsto \tau_{\text{init}} + \zeta_{ij}\Delta \quad (2)$$

with ζ_{ij} denoting the number of solutions $\pi \in P$ with $\pi(i) = j$, or, using the population matrix Q, $\zeta_{ij} = |\{h : q_{ih} = j\}|$.

Observe that a pheromone value possesses one of $k+1$ possible discrete values and a population update corresponds to an implicit update of the pheromone values:

- A solution π entering the population corresponds to a positive update:

  $\forall i \in [0, n-1] : \tau_{i\pi(i)} \mapsto \tau_{i\pi(i)} + \Delta$

- A solution σ leaving the population corresponds to a negative update:

  $\forall i \in [0, n-1] : \tau_{i\sigma(i)} \mapsto \tau_{i\sigma(i)} - \Delta$

This approach is different to the standard ACO algorithm, in which evaporation is used to reduce all pheromone values after an iteration (see [4,37]). The increment value $\Delta > 0$ and the population size k, along with the number of ants per iteration m, are all parameters of the ACO algorithm. The exact value of τ_{init} is arbitrary, as Δ can simply be scaled accordingly to produce an identical probability distribution as per Eq. (1).

4.2. Advantages of mapping population-based ACO to FPGA

Whereas the ACO approach requires floating point representations of pheromone values, in P-ACO the pheromone information is implicitly stored within the population. The FPGA implementation can therefore restrict itself to managing a population of solutions represented by an $n \times k$ matrix of integer numbers that is better suited to current field-programmable technology.

For every ant decision in ACO, the probability distribution in Eq. (1) has to be computed. On a sequential machine, this operation requires $O(n)$ multiplication and addition steps. By implementing in hardware or by parallelizing this operation the time complexity could be reduced to $O(\log n)$ steps.

Using the P-ACO approach, we show that it is not necessary to perform time consuming prefix sum calculations as in ACO. An ant decision in P-ACO is based on $k+1$ possible discrete pheromone values. Moreover, at least $n-k$ of the pheromone values in a row are the same. Time requirements for a decision can be reduced to $\Theta(k)$ simple steps as will be further explained in Section 5. Note that k is typically a small constant because empirical studies have shown that a population size of $1 \leq k \leq 8$ is recommendable [31]. For a problem of size n, this implies $\Theta(nk)$ time is required per ant to construct a solution. In comparison, a sequential processor that uses the ACO approach requires $O(n^2)$ time per ant to construct a solution. In addition, the functional parallelism embodied in processing m ants in parallel on an FPGA allows m solutions to be formed in $\Theta(nk)$ time, compared to $O(mn^2)$ time on a sequential processor using the ACO approach.

5. Implementation of P-ACO on FPGA

In this section, we first give a general top-level overview of the P-ACO hardware implementation. We then explain the implementation of the main modules in greater detail. This section describes the P-ACO implementation for SMTTP.

5.1. Overview

At high level, the mapping of the P-ACO algorithm into the corresponding FPGA design is straightforward (see Fig. 3) and consists of three main modules:

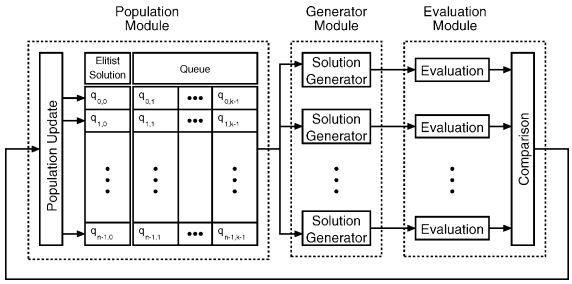

Fig. 3. P-ACO design with Population, Generator and Evaluation modules.

Population, Generator and Evaluation modules. Note that, for the sake of clarity, the Control module has been omitted. The Population module contains the population matrix $Q = [q_{ij}]_{n\times k}$ comprising the elitist solution in column $j = 0$ and the FIFO-queue in columns $j \in \{1, \ldots, k-1\}$. For SMTTP, each item q_{ij} is the number of a job to be scheduled. The Population module is responsible for broadcasting items q_{ih} ($h \in \{0, \ldots, k-1\}$) in the ith row of the population matrix to the Generator module. Furthermore, at the end of the current iteration it receives the best solution from the Evaluation module, which is then inserted into the queue. The Generator module holds m Solution Generators working concurrently, one Solution Generator per ant. The solutions are transferred from there to m parallel Evaluation Blocks in the Evaluation module. It is also possible to have less than m Solution Generators and Evaluation circuits, which will be discussed in greater detail in Section 7. The evaluation results of these m solutions are collected in a Comparison block, which determines the best solution of the current iteration. This best solution also becomes the new elitist solution, if it is better than the current elitist solution.

5.2. Generator module

The Generator module contains m identical Solution Generators, each of them simulates the behavior of an artificial ant constructing a solution using the P-ACO metaheuristic.

The Solution Generator (cmp. Fig. 4) consists of the three blocks S-Array, Match Buffer and Selector. The S-Array stores and maintains selection set $\{S, \ldots, n-1\}$ of items $s_i \in S$ with $i \in \{0, \ldots, c\}$ and $c = |S| - 1$. Furthermore, the S-Array can be queried for its contents. When it receives a broadcast item q_{ih} from the Population module, it compares this item with all remaining items in S. If there exists an item $s_l \in S$ with $s_l = q_{ih}$ (i.e. item s_l has been matched), then the S-Array returns the address a_l, which is the address of s_l within the S-Array. The Match Buffer stores these match-addresses a_l into a register. The Selector builds-up a probability distribution and randomly selects an item $s_{l^*} \in S$ which is then sent to the Evaluation module.

The processing flow within a Solution Generator is depicted in Fig. 5. The Solution Generator starts off by filling the selection set S with numbers $0, \ldots, n-1$ and initializing the S-Counter c. This S-Counter is implemented in an external logic in the Control module. The current S-Counter value is available to all Solution Generators. The Match Counter M expresses the number of match addresses stored in the Match Buffer. In the loop, which starts after initialization, all items q_{ih} with $h \in \{0, \ldots, k-1\}$ in row i of the population are broadcast to the S-Array. Whenever an item s_l in the S-Array has been matched, i.e. $s_l = q_{ih}$, then the corresponding match-address a_l is stored into the Match Buffer. Note that if an item matches multiple times, its address is also transferred multiple times. After the last repetition of the loop the Match Buffer contains all items which appear in row i of the population as well as in selection set S. From these matches we are able to derive a probability distribution over all items in row i of the pheromone matrix which is implicitly represented by row i of the population:

$$\forall j \in S : p_{ij} = \frac{\tau_{ij}}{\sum_{z\in S}\tau_{iz}} = \frac{\tau_{\text{init}} + \zeta_{ij}\Delta}{\sum_{z\in S}(\tau_{\text{init}} + \zeta_{iz}\Delta)} = \frac{\zeta_{ij}\Delta + 1}{c + M\Delta + 1} \tag{3}$$

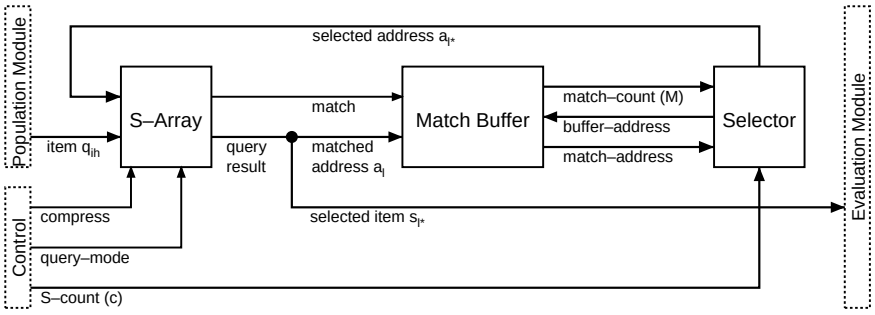

Fig. 4. Solution Generator.

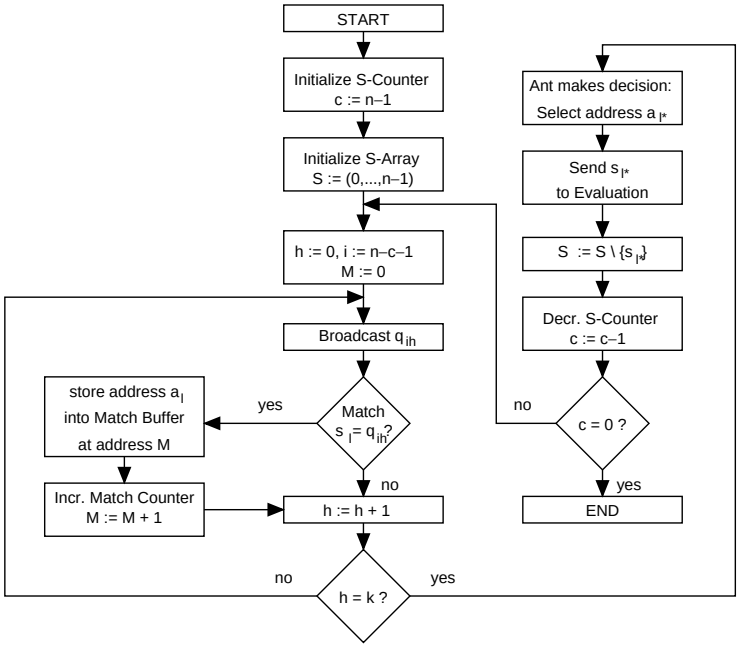

Fig. 5. Solution generation.

Note that in our P-ACO hardware implementation we set $\tau_{\text{init}} := 1$. Heuristic information is disregarded, but we outline the integration of a discretized heuristic as a possible modification in Section 7.2. According to the probability distribution in Eq. (3) the ant (Solution Generator) makes a decision for the ith place in the solution vector. It randomly selects an address a_{l^*} which is sent to the S-Array to query the selected item s_{l^*} stored at address a_{l^*}. The selected item s_{l^*} is then transferred to the respective Evaluation block. With respect to SMTTP, the selected job is evaluated by calculating its job-tardiness which is then added to the intermediate tardiness sum over all previously scheduled job. Afterwards, s_{l^*} is removed from selection set S and the process continues making decisions for the remaining n–c places in the solution vector.

The organization of an S-Array within the Generator module is shown in Fig. 6. It consists of n S-Cells connected by a fanout bus broadcasting queries. Each S-Cell is connected to its successor S-Cell to realize a right-shift of items during array compression (removal of the selected item from set S) which is done to guarantee that the items in S are always stored in S-Cells with addresses $a_c, \ldots, a_{n-1}$. Only active S-Cells contain valid items in S as indicated by the Active Flag (AF). An OR-tree generates the match detection bit for the entire array. An n-to-$\lceil \log_2 n \rceil$ encoder is required to produce the address of the S-Cell which asserted the hit signal. This cell-address represents the select signal of the multiplexer connecting the respective q-res bus to the query result output sent to the S-Array.

Shaded cells are deactivated, they no longer contain an item in S. An S-Cell performs the basic operations during solution generation and its high level structure is shown in Fig. 7. It contains two comparators, a register buffering the selected address a_{l^*}, and two control signals encoding the current phase.

The operation of the S-Array consists of three phases: the Broadcast Phase, the Selection Phase and the Compression Phase. During the Broadcast Phase the k items in population row i are broadcast to the S-Cells (i.e. query $= q_{ih}$). The eq-comparator in the S-Cell compares the broadcast item with its own stored item s. If they are equal, the S-Cell emits its hard-coded address a as query result and asserts a hit signal, if the respective S-Cell is active ($AF = 1$).

After the Selector has randomly selected item s_{l^*}, the S-Array enters the Selection Phase. The selected address a_{l^*} is broadcast to the S-Cells. Each S-Cell

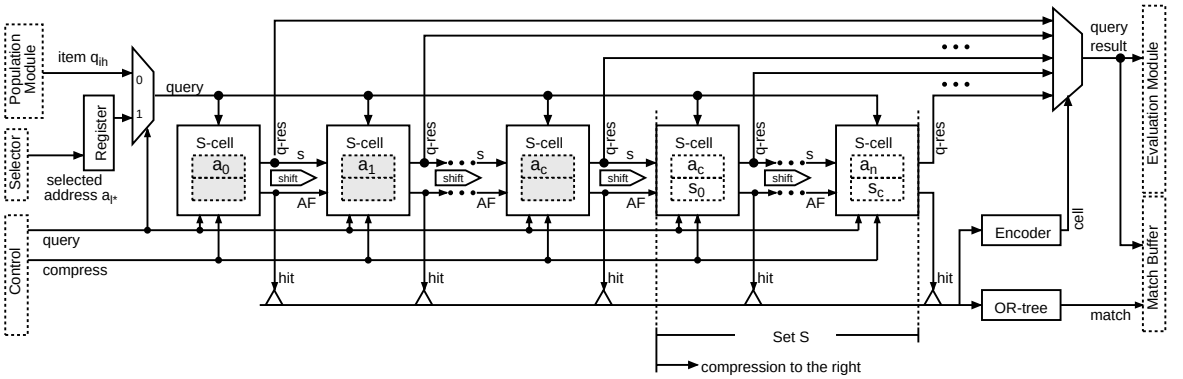

Fig. 6. S-Array. $a_i = i$ is the index of cell i, and s_i is the item in S at this address.

uses its eq-comparator to compare the broadcast address to its own address a.

S-Cell l^* forwards its stored item s_{l^*} as query result to the Evaluation module. Each S-Cell j also uses its own le-comparator to check whether it is a predecessor of the selected S-Cell, i.e. $a_j \leq a_{l^*}$. All predecessor S-Cell set their Match Flag to MF := 1, all others set MF := 0. These flags are to be used in the Compression Phase.

The final phase for the S-Array is the Compression Phase. Each S-Cell sends its Active Flag (AF) and its stored item (s) to its immediate successor SCell. An S-Cell latches the corresponding data arriving from its immediate predecessor S-Cell, if its own Match Flag was set (MF = 1) in the Selection Phase. A logical zero is loaded into the AF flip-flop of the first S-cell. The overall effect is that the selected item s_{l^*} is overwritten, and all items to the left of the selected cell (i.e. $a_j < l^*$) are shifted one cell to the right. After an iteration is complete, the original values of the data registers in the S-Array are re-initialized by loading the hard-coded S-Cell address values a into the s-registers.

The Match Buffer is a simple circuit which stores incoming match-addresses into a register of size k, if the match signal is asserted. After the last broadcast M indicates the total number of Matches sent to the Selector. The structure of the Match Buffer is shown in Fig. 8a.

The Selector (see Fig. 8b) receives the number of matches M from the Match buffer. The Selector has been designed to realize the selection of an item according to Eq. (3). Here, the upper bound $R = c + M\Delta$ for a pseudo random number generator is calculated, where $\Delta = 2^y$ has been chosen to be a power of 2 in order to replace multiplications and divisions, which

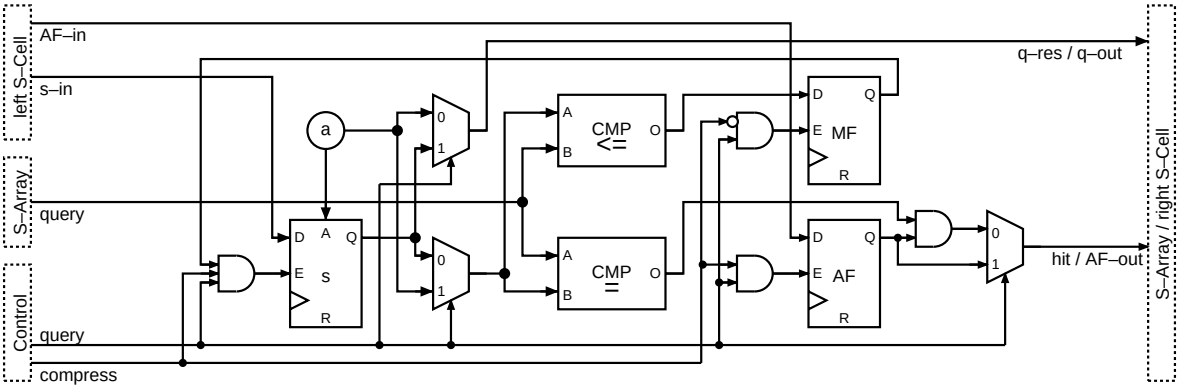

Fig. 7. S-Cell architecture.

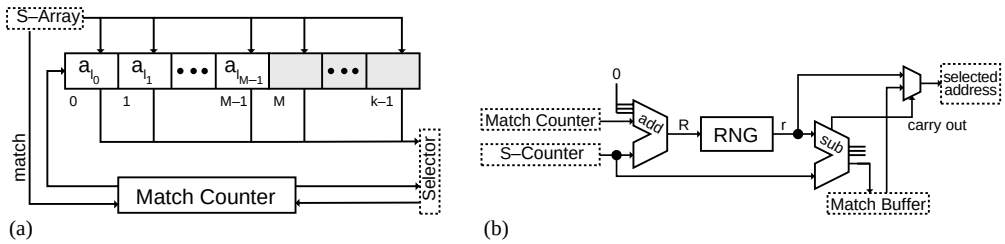

Fig. 8. (a) Match Buffer (shaded area is empty) and (b) Selector.

are not readily supported by current FPGA technology, by simple left and right shifts respectively. A random number r is drawn uniformly from the interval $[0, R]$. Without large multiplier or divider circuits, it is difficult to draw a random number uniformly from an arbitrary interval. Hardware-based random number generators creating random bits can be used to create random numbers from the interval $[0, 2^x - 1]$ by drawing x bits. However, since $R + 1$ will most likely not be a power of 2, we suggest repeatedly drawing a random number $r \in [0, R']$ until $r \leq R$, where $R' = 2^{\lceil \log(R) \rceil} - 1$ is the smallest power of 2 greater than or equal to R. Let $p = \text{Prob}(r \leq R) = (R + 1/R' + 1)$ denote the probability of drawing a random number $r \in [0, R]$, where $1/2 < p \leq 1$. Then the probability of drawing a random number $r \in [0, R]$ after $u - 1$ unsuccessful trials is $P(U = u) = p(1 - p)^{u-1}$, decreasing exponentially in u. Hence, the expected value is $E(U) = p\sum_{u=1}^{\infty} u(1 - p)^{u-1} = 1/p$. On average, random numbers have to be drawn $E(U)$ times, with $1 \leq E(U) < 2$, in order to receive a uniformly distributed random number $r \in 2\,[0, R]$. The random bits necessary for generating the random number are created with the RNG introduced by Ackermann et al. [38], which is well suited in terms of quality of the random bits and space requirements.

Once r has been generated, we compare the number with c to determine whether an item from the S-Array or a buffer-address in the Match Buffer has been chosen. If $r \leq c$, then $a_{l^*} = a_r$ is the index of the selected item from S. Otherwise, $a_{l^*} = a_{M^*}$, with $M^* = l_{(r-c) \gg y}$, i.e. the Match Buffer is queried for the actual match-address a_{l^*}. Recall that the Match Buffer consists of the addresses $\{a_{l_0}, \ldots, q_{l_{M-1}}\}$. After a_{l^*} has been determined, the corresponding item s_{l^*} is queried from the S-Array and sent to the Evaluation module.

5.3. Evaluation module

The Evaluation module is used to evaluate the solutions generated by the Solution Generators. A specific Evaluation block is required for each distinct optimization problem to be solved using the P-ACO metaheuristic. Such an Evaluation block takes problem-specific evaluation parameters (e.g. job processing times and deadlines for the SMTTP) and calculates the objective function value of a solution arriving from a Solution Generator.

The Comparison block determines the best solution of the iteration and the new elite solution (if found) to be stored into the population. The design of the Comparison block for SMTTP is shown in Fig. 9. A comparator tree reduces the logic delay of the Comparison block. The index (best-index) of the Solution Generator, which constructed the best solution, is forwarded to the Population module. An additional comparator compares the quality of the best solution in the current iteration with the quality of the elite solution. If the currently best solution is also better than the elite solution, then an elite flag is-elite is asserted.

5.4. Population module

The Population module (see Fig. 10) provides the storage for the population matrix. This storage makes use of the block select RAM (BRAM) on the Virtex and Virtex E/II/II pro architectures. The Population module receives the index (best-index) of the best solution from the Evaluation module. A so-called solution forwarding unit (SFU) is responsible for transferring the respective best solution to the BRAM. It also calculates the write-addresses of the forwarded solution. This solution is then written into

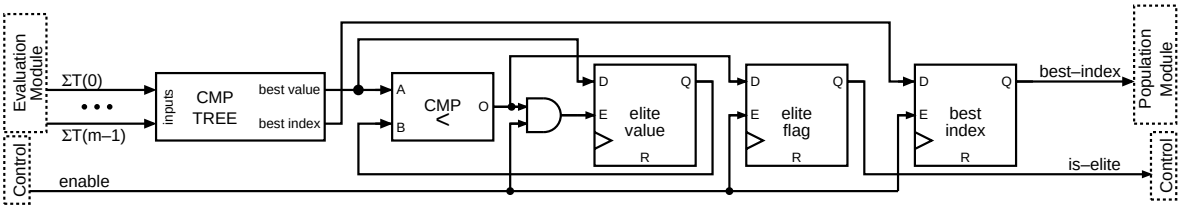

Fig. 9. Comparison block for the SMTTP. Reset and clock wires are omitted.

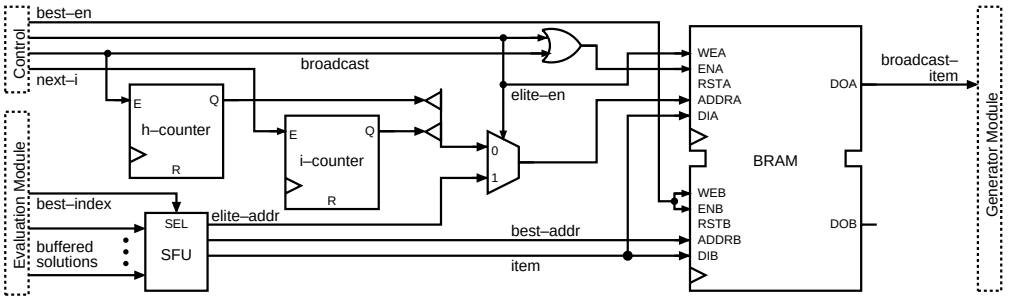

Fig. 10. Population module.

the FIFO-queue (columns $j \in \{1, \ldots, k-1\}$ of the population matrix) via port B of the BRAM. If the arriving solution is also a new elitist solution (elite-en asserted), then it is also written into column $j = 0$ via port A. During the Query Phase the Population module sends all k items q_{ih} stored in population row i to the Generator modules. Therefore it contains two counters producing indices h and i as described in Fig. 5. The outputs of both counters are joint to form a address bus for the items to be read via port A.

6. Experimental results

6.1. Experimental method

We compare the hardware implementation of P-ACO with its software counterpart using contemporary FPGA technologies. For the hardware implementation, we encoded the design in register-transfer level VHDL targeting the Xilinx Virtex family of FPGAs [32], but we exhausted the largest device in that family with some of the problem instances. In order to gather a reasonable amount of sample points to analyse the hardware requirements statistically, we performed timing estimations on the latest and largest architecture from Xilinx, the Virtex-II Pro Platform FPGA XC2VP125 [34]. We did not utilize the PowerPC cores present in the Virtex-II Pro architecture, because these are very special to the Virtex-II Pro. How these embedded CPU cores can be used efficiently for our design is a direction of future work. We used XST 5.2sp1 on normal optimization effort for high level logic synthesis, and Xilinx ISE 5.2sp1 Place and Route for implementation on the FPGA. In software, we programmed the P-ACO to execute on a AMD Athlon uniprocessor machine clocked at 1540 MHz to measure timing performance of the software implementation. In this paper, we restrict to comparing the implementation on a single FPGA with a software-based solution on a single processor. Considering an implementation on a multi-FPGA-board or a parallel variant of the software implementation is an interesting aspect for future research.

The optimization problem regarded is SMTTP, where the problem size, i.e. the number of jobs, was scaled within the range from $40 \leq n \leq 320$. Processing times were chosen randomly from the interval [1:64]. The number of Solution Generators (which equals the number of ants per iteration)

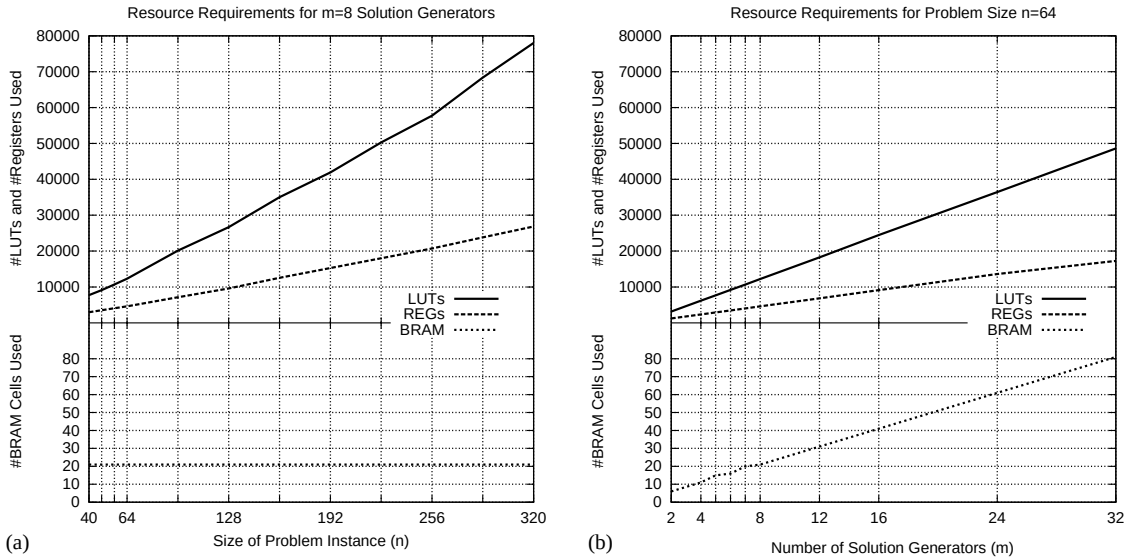

Fig. 11. FPGA resource utilization—number of BRAM cells, number of LUTs, number of slice registers—for varying problem sizes and number of implemented Solution Generators. (a) Varying problem sizes n, fixed number of Solution Generators ($m = 8$). (b) Varying number of Solution Generators m, fixed problem size ($n = 64$).

ranged from $m = 2$ to 32. For the hardware implementation, we recorded the utilization of FPGA resources, circuit delays, and operational frequencies. For every problem instance of the software counterpart, we recorded the execution time per iteration as an average over 100,000 iterations of the P-ACO algorithm.

6.2. Resource requirements

We measured the resource requirements by counting the number of look-up tables, slice registers (REGs), and Block Select RAM cells used by our design implementations. In Fig. 11a, resource requirements are depicted for a fixed number of Solutions Generators $m = 8$ and variable problem size n. The resource requirements shown in Fig. 11b are for a fixed problem size $n = 64$ and a variable number of Solution Generators.

As the problem size increases (see Fig. 11a), the number of S-Cells, i.e. the width of an S-Array, grows proportionally. The height of the S-Array, grows logarithmically, because each S-Cell stores one item represented by $\log(n)$ bits. Since the S-Arrays occupy the largest portion of the total P-ACO circuitry, the overall consumption of LUTs and REGs grows according to $\Theta(n \log(n))$.

This theoretical assumption is confirmed by the regression results (see Table 1) which correspond to the values shown in Fig. 11. As indicated by the Bravais–Pearon index $r \approx 1$, all resource requirements show a very strong correlation to m and n, respectively. The number of BRAM cells used remains on a constant level of 21. For an increasing number of Solution Generators (see Fig. 11b), all resource requirements grow linearly. Given a fixed problem size of $n = 64$, we need extra 1516 LUTs, 542 REGs,

Table 1
Regression results for number of LUTs, REGs and BRAMs

Fixed	Resource	Regression function f	r
$m = 8$	LUTs	$28.2257n\log_2(n) + 1531.71$	0.999079113
	REGs	$9.74471n\log_2(n) + 929.175$	0.999861506
	BRAM	21	Undefined
$n = 64$	LUTs	$1516.17m + 97.2749$	0.999987262
	REGs	$542.244m + 244.764$	0.998796397
	BRAM	$2.4802m + 1.5297$	0.999374327

For every regression function, the Bravais–Pearon correlation index r is given.

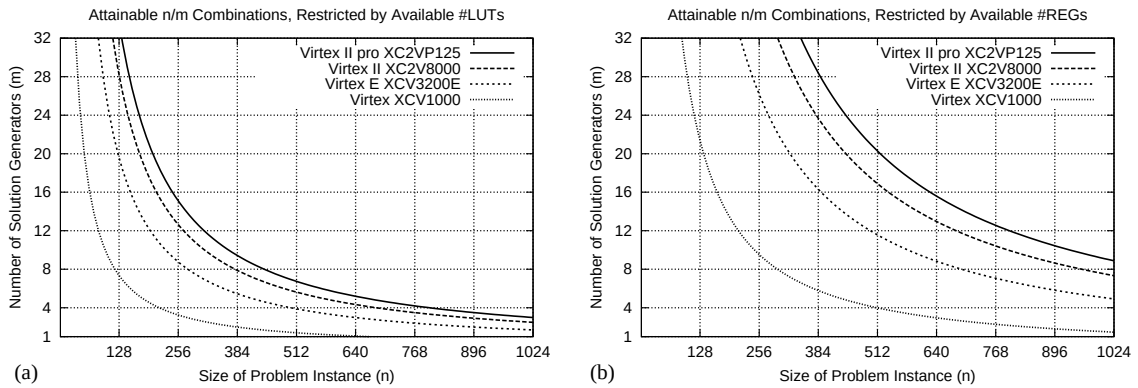

Fig. 12. Attainable combinations of n and m for different types of FPGAs. Restrictions set by the number of available LUTs (a) and REGs (b).

and 2.5 BRAM cells approximately, if we intend to extent the P-ACO circuit by an additional Solution Generator (cmp. Table 1).

Fig. 12 provides an overview of the maximal attainable combinations of m and n for the largest members of the Xilinx Virtex, Vitex E, Virtex II, and Virtex II pro FPGA families. Note that in practice it is inefficient to use very large numbers of ants (Solution Generators). Therefore, in the diagrams the upper limit in the range of Solution Generators was set to $m = 32$. The left Figure shows all combinations, if the implementations were only restricted by the number of available LUTs; whereas the right figure shows all implementations, if they were only restricted by the number of available REGs. On each of the considered FPGA devices, the number of LUTs is equal to the number of REGs. However, in the design implementation the number of configured LUTs is about 170% higher than the number of REGs used. Hence the curves in the right figure are considerably higher than the respective curves on the left hand side.

6.3. Time requirements

The generation of a solution for SMTTP consists of n decision steps (cmp. Fig. 13). For each decision, k queue items are broadcast and matched. The time requirements for this operation is denoted by t_{bm}. Afterwards, item s_{l^*} is selected and evaluated, which takes t_s and t_e time. The time to compress the S-Array is denoted by t_{cps}. Due to the generic implementation of the Solution Generators, for a given problem size n, the total time for any decision t_d is constant and does not depend on the optimization problem under consideration. On the other hand, the time to evaluate an item does strongly depend on the optimization objective. Thus the time for generating a complete solution is $t_g = nt_d$, if $t_{\text{cps}} \geq t_e$, otherwise $t_g = \max\{nt_d + t_e, nt_e + t_d\} - t_{\text{cps}}$. We implemented the P-ACO circuit for SMTTP so that evaluation does not retard the beginning of the next decision, i.e. $t_g = nt_d + t_e - t_{\text{cps}}$. If a solution cannot gradually be evaluated after each decision, like in QAP, then the complete solution is evaluated after the last decision has been made. In every iteration m, solutions are generated in parallel. Afterwards, these solutions are compared by their evaluation quality and the population is updated, which requires t_{cmp} and t_u time respectively. Thus the time to finish a complete iteration is $t_{it} = t_g + t_{\text{cmp}} + t_u$. The time to (re-)initialize the Solution Generators is disregarded.

Table 2 shows the maximum global clock frequencies f_{gl} for a fixed number of Solution Generators $m = 8$ and varying problem sizes n. It further gives an overview of the required clock tics: number of clock tics for generating a solution $c_g = t_g f_{gl}$, clock tics for comparison $c_{\text{cmp}} = t_{\text{cmp}} f_{gl}$, clock tics per population update $c_u = t_u f_{gl}$, and clock tics per iteration $c_{it} = t_{it} f_{gl} = c_g + c_{\text{cmp}} + c_u$. Accordingly, the maximum frequency per iteration can be calculated: $f_{it} = f_{gl}/c_{it}$.

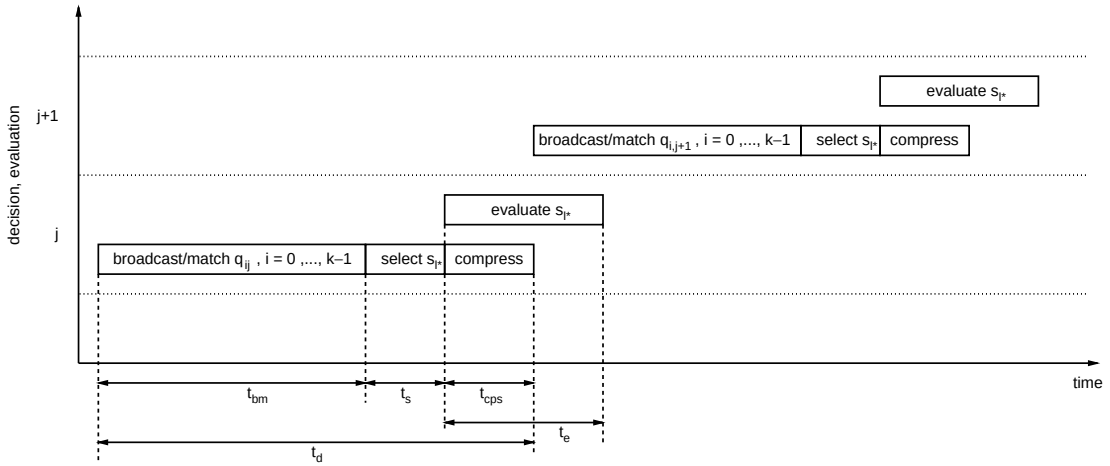

Fig. 13. Schedule of two decisions and evaluations. Here, only one Solution Generator is considered.

In Fig. 14, we present timing results obtained by synthesizing the VHDL description of our design. The gap at $m = 7$ is present as the endpoints of the critical path changes. For $m \leq 7$, the address counter to the S-Array output is the critical path; whereas for $m \geq 8$, a control signal is the source of the critical path. In Fig. 14, the increase in critical path length can be attributed to the increased logic levels as the design grows in size; the growth in critical path length for increasing numbers of Solutions Generators implemented is likely to be due to the increased difficulty of solving the placement and routing problems for large circuit sizes. In all instances, routing delay is the major component of the total critical path delay. Clock skew was a negligible problem for the larger circuits due to the dedicated clock routing resources of the FPGA. As we increase the number of Solution Generators, propagation delays grow only slightly.

Table 2
Operating frequencies for the hardware P-ACO implementation with $m = 8$ Solution Generators on the Virtex-II Pro X2VP125-7 device

n	Clock Tics				Max f_{gl} (MHz)	Max f_{it} (kHz)
	c_g	c_{cmp}	c_u	c_{it}		
40	403	2	40	445	27.651	61.935
48	483	2	48	533	32.219	60.448
56	563	2	56	621	30.806	49.607
64	643	2	64	709	25.997	36.667
96	963	2	96	1061	26.832	25.289
128	1283	2	128	1413	22.522	15.939
160	1603	2	160	1765	19.899	11.274
192	1923	2	192	2117	21.011	9.925
224	2243	2	224	2469	19.001	7.695
256	2563	2	256	2821	19.789	7.015
288	2883	2	288	3173	15.273	4.813
320	3203	2	320	3525	14.494	4.112

In Figs. 15 and 16, we compare the P-ACO algorithm implemented in software against the hardware realization. Since the software and hardware versions produce the same solutions, their performances are compared by means of the required time per iteration. Note that, except for the first k iterations, the execution time for every iteration is constant. The figures on the left compare the time per iteration, whereas the figures on the right depict the respective speedup values, where the speedup is defined by:

$$\text{speedup} = \frac{\text{time per iteration in software}}{\text{time per iteration in hardware}}$$

In Fig. 15, the execution time per iteration for both the software and the hardware implementations grow with a linear trend as the problem size n is increased. However, for every problem size, the hardware version runs faster than its software counterpart. Speedup values range from a minimum of 2.04 for a problem size of $n = 320$ to maximum value of 4.07 for $n = 48$.

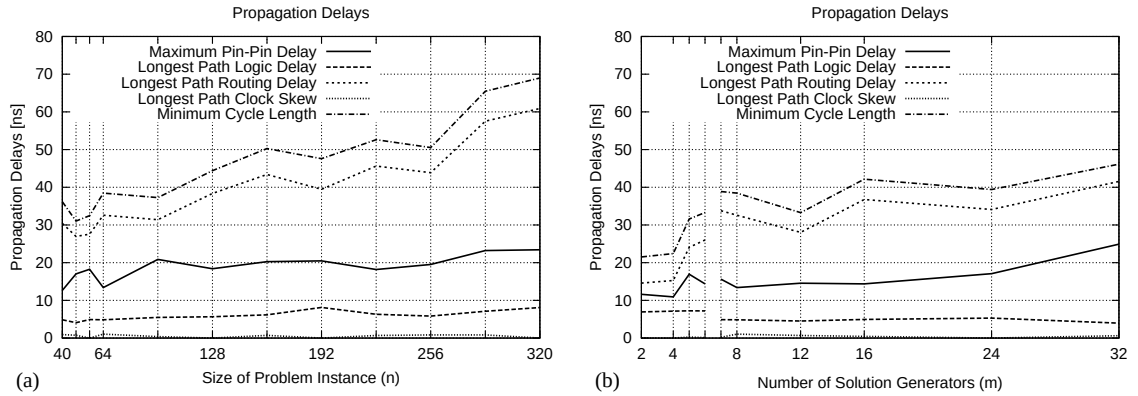

Fig. 14. Delay components contributing to critical path for different problem sizes and number of implemented Solution Generators. (a) Number of implemented Solution Generators $m = 8$, varying problem size n. (b) Problems of size $n = 64$, varying number of implemented Solution Generators m.

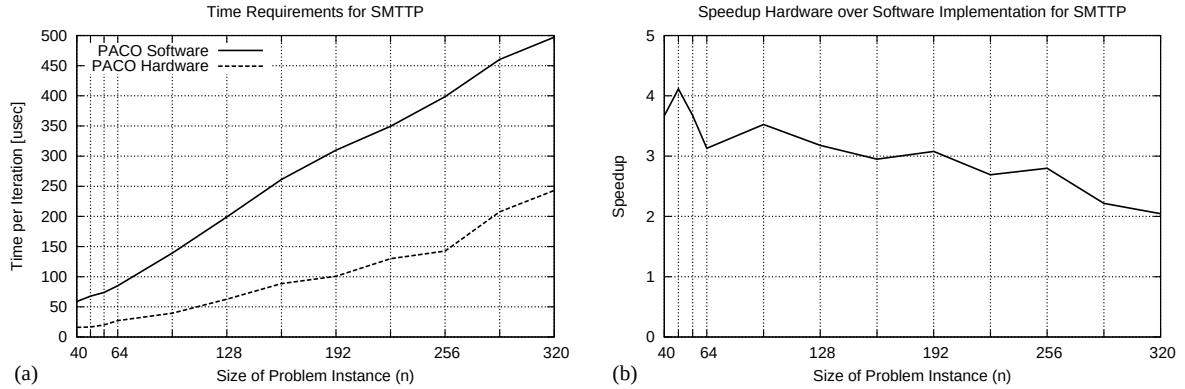

Fig. 15. Comparison of hardware and software implementation of P-ACO for fixed $m = 8$. (a) Time requirements per iteration. (b) Speedup of hardware over software implementation.

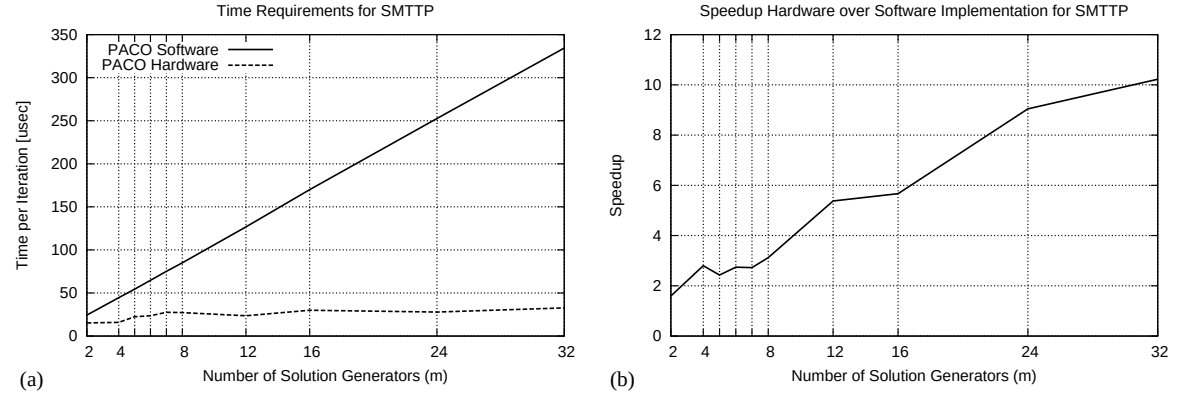

Fig. 16. Comparison of hardware and software implementation of P-ACO for fixed $n = 64$. (a) Time requirements per iteration. (b) Speedup of hardware over software implementation.

Fig. 16 shows that the time per iteration in software grows with a linear trend as the number of Solution Generators is increased; whereas for the hardware implementation, the time requirements remain on an almost constant level of approximately 28 μs. The corresponding speedup values range from 1.59 for $m = 2$ Solution Generators to 10.22 at $m = 32$. We achieve speedup linear in m, taking into account that the task of placing and routing the design becomes considerably more difficult for larger m. From the P-ACO perspective this means that within certain constraints we can easily increase the degree of parallelism (in terms of concurrently working ants) with only a marginal decline in execution speed.

7. Modifications

This section deals with modifications to the FPGA implementation of P-ACO described in Section 5. Specifically, we propose a technique for reducing the space needed by the P-ACO algorithm and discuss how to enable the algorithm to use heuristic information.

7.1. Space constraints

Depending on the size of the FPGA designated for mapping the P-ACO algorithm, we will at some point be dealing with problem instances too large to fit the entire algorithm as introduced in Section 5. Specifically, the "height" of the P-ACO algorithm increases only logarithmically with n whereas the "width" increases in a linearly, resulting in an increasingly flat rectangle shape as a basic structure. Then, fitting the algorithm can be accomplished by folding, which is standard technique for this kind of problem.

If a problem size n is given, the available resources might constrain the number of Solution Generators that can be implemented on the FPGA device. We propose implementing only $m' < m$ Solution Generators operating in g cycles, where $m = gm'$ (see Fig. 17). In every cycle m', solutions are generated in parallel. Afterwards, these solutions are compared by their evaluation quality. After each cycle, the newly created solutions are compared among each other, and the best solution of the current cycle is compared with the best solution of all preceding cycles. Comparison takes tcmp time, and can be done while the next solution is being generated. Thus the time to finish a complete iteration is $t_{it} = \max\{gt_g + t_{\text{cmp}}, gt_{\text{cmp}} + t_g\} + t_u$.

Another method to make better use of the available space takes into account the fact that the selection set S decreases in size over time. It is possible at certain points in time to move the currently active selection sets to an S-Array with a smaller number of S-Cells. Two examples for this modification are given in Fig. 18.

The 2/4 S-Array configurations in Fig. 18a and b save 25/37.5% space at the cost of having the simulated ants be in different stages of completion due to their respective starting delays. In both configurations, the left side shows all S-Arrays after a new ant was started, the right side the moment just before the selection sets are shifted down and a new ant starts.

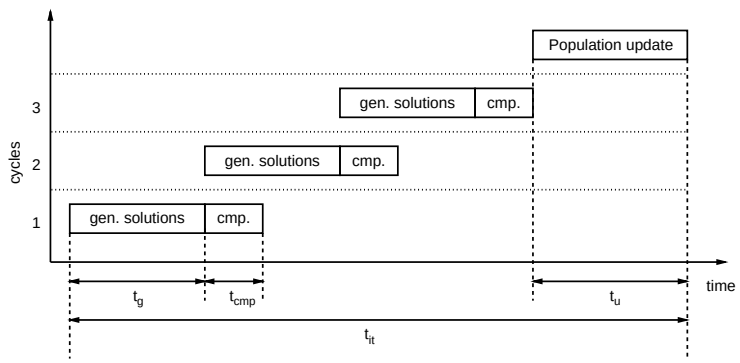

Fig. 17. Schedule of a complete iteration consisting of $g = 3$ cycles. After the last cycle, the population is updated. Here, only one Solution Generator is considered.

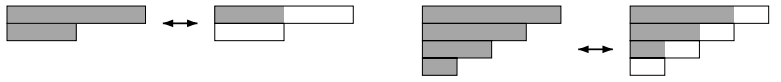

Fig. 18. Example of 2 S-Array (a) and 4 S-Array configuration (b) with decreasing size. The shaded area represents the active part of the row holding set S.

7.2. Heuristic information

Another aspect of ACO algorithms in general that was neglected in the basic layout described in Section 5 is the use of heuristic information about the problem instance. A characteristic of ACO in general is that heuristic information can often be easily integrated into the algorithm and is used successfully by most ACO algorithms for combinatorial optimization problems [39]. In this subsection, we describe a method of utilizing heuristic information without giving up the short run times of the FPGA implementation of the P-ACO algorithm.

The integration of heuristic information into the P-ACO hardware algorithm poses two problems: heuristic values are generally real-valued, and they exist for all items of the set S, not just an $O(k)$ size subset. Therefore, we propose to transform the information in every row i of the heuristic matrix $\vec{\eta}_i = (\eta_{i,0}, \ldots, \eta_{i,n-1}) \in \mathbb{R}^n$, e.g. the reciprocal of the distances from city i to all others in TSP, into a set of l heuristic-vectors $\{\vec{h}_{i,0}, \ldots, \vec{h}_{i,l-1}\}$ with $\vec{h}_{i,u} \in [0, n-1]^t$ and $u \in \{0, \ldots, l-1\}$, where each vector $\vec{h}_{i,u}$ holds items which have a high heuristic value. The following method is proposed:

(1) calculate

$$\delta_i = \frac{1}{tl}\sum_{j=0}^{n-1} n_{ij}$$

(2) do the following n times

   (a) determine j^* so that $\eta_{ij^*} = \max_{j=0,\ldots,n-1}\eta_{ij}$
   (b) add item j^* to the pool of numbers for building the heuristic-vectors.
   (c) update $n_{j^*} \mapsto \eta_{j^*} - \delta_i$.

The quality of the approximation attainable by this method depends on the values of η_{ij}, t and l. Deciding which items should be placed into a given vector is accomplished by placing any one item in as many different vectors as possible and combining it with the maximum amount of other items. The order of the items in each vector $\vec{h}_{iu}$ is arbitrary. If $l > 1$, i.e. there is more than one heuristic-vector for the given row, then we declare one of the heuristic-vectors $\vec{h}_{iu^*}$ as active, and the other $l-1$ as inactive. Only the active heuristic-vector affects the decision-process of the ant. After an iteration of m ants has finished, the active heuristic-vector $\vec{h}_{iu^*}$ is replaced by some other $\vec{h}_{iu}$ with $u \in [0, l-1]\backslash\{u^*\}$. Note that the creation of the heuristic-vectors $\vec{h}_{i,0}, \ldots, \vec{h}_{i,l-1}$ must take place before the algorithm is programmed onto the FPGA. Therefore, problems which require an online computation of the heuristic values, e.g. most scheduling problems, cannot be handled by this method.

In the P-ACO algorithm, the pheromone matrix of the standard ACO algorithm was replaced by a population of k good solutions. The necessary calculations to transform row i in the population matrix Q into the respective values τ_{ij} were described in Eq. (2). Likewise, we are now able to transform any heuristic-vector $\vec{h}_{iu^*}$ into the corresponding heuristic value $\hat{\eta}_{ij}$ describing the heuristical impact of item j when making a decision in row i:

$$\forall i, j \in [0, n-1] : \hat{\eta}_{ij} \mapsto \eta_{\text{init}} + \gamma_{ij}\Delta_{\text{H}} \tag{4}$$

where $\eta_{\text{init}} \geq 0$ denotes the base heuristic value assigned to every item j and γ_{ij} describes the number of occurences of item j in the active heuristic-vector, i.e. $\gamma_{ij} = |\{v : h_{iu^*v} = j, v = 0, \ldots, t-1\}|$. These occurences are all weighted by the same value ΔH.

As in standard ACO (cmp. Eq. (1)), ants make random decisions according to a probability distribution p_{ij}:

$$\forall j \in S : p_{ij} = \frac{\tau_{ij}\hat{\eta}_{ij}}{\sum_{z\in S}\tau_{iz}\hat{\eta}_{iz}} = \frac{(\tau_{\text{init}} + \zeta_{ij}\Delta_{\text{P}})(\eta_{\text{init}} + \gamma_{ij}\Delta_{\text{H}})}{\sum_{z\in S}(\tau_{\text{init}} + \zeta_{iz}\Delta_{\text{P}})(\eta_{\text{init}} + \gamma_{ij}\Delta_{\text{H}})} \tag{5}$$

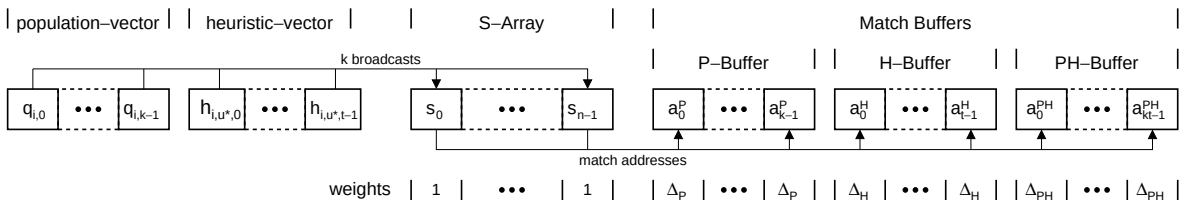

Fig. 19. Items in the current Population row i and the respective heuristic-vector $\vec{h}_{iu^*}$ are broadcast. Match addresses are stored in Match Buffers. Weights for items in S and match addresses are indicated.

In order to distinguish the two types of weights, Δ in Eq. (2) has been renamed to Δ_P. Unlike in standard ACO, no exponentiations by α and β are required anymore. Pheromone and heuristic values are weighted by choosing the init and Δ parameters accordingly. The required arithmetic operations better suit the resources provided by the FPGA architecture. Furthermore, no prefix sum calculations are required to allow the ant to draw from this distribution. Instead we extend the Match Buffer concept introduced in Section 5.2 to also consider the knowledge stored in the heuristic-vectors.

Let population-vector $(q_{i,0}, \ldots, q_{i,k-1})$ be the current row in the population, and $\vec{h}_{i,u^*} = (h_{i,u^*,0}, \ldots, h_{i,u^*,t-1})$ the current heuristic-vector (cmp. Fig. 19).

All $k + t$ items in both vectors are broadcast to the S-Array that contains the items of selection set S. The addresses of matched items are then stored into three different types of Match Buffers. The P-Buffer stores $M_P \leq k$ addresses of items which are in the population-vector as well as in set S, i.e. the P-Buffer is equivalent to the Match Buffer shown in Fig. 8a. The respective $M_H \leq t$ addresses of items which occur in the heuristic-vector and in selection set S are copied into a separate location called the H-Buffer. Furthermore, since the pheromone and heuristic values are multiplied in the ACO algorithm, we need an additional buffer—called PH-Buffer—storing the items which are in the heuristic-vector as well as in the population-vector and in set S. Let $\Delta_P = 2^{y_P}$ be the weight associated with the population-vector and $\Delta_H = 2^{y_H}$ the weight of the heuristic-vector derived from the heuristic information. Then, $\Delta_{PH} = \Delta_P \Delta_H = 2^{y_P + y_H}$ is the weight for an item j of which the address is stored $\phi_{ij} = \zeta_{ij}\gamma_{ij}$ times in the PH-Buffer. After all broadcasts, the PH-Buffer contains $M_{PH} \leq kt$ addresses. Using the buffer concept and setting $\tau_{init} = \eta_{init} = 1$, Eq. (5) can now be transformed to:

$$\forall j \in S : p_{ij} = \frac{\tau_{ij}\hat{\eta}_{ij}}{\sum_{z \in S} \tau_{iz}\hat{\eta}_{iz}} = \frac{1 + \zeta_{ij}\Delta_P + \gamma_{ij}\Delta_H + \phi_{ij}\Delta_{PH}}{|S| + M_P\Delta_P + M_H\Delta_H + M_{PH}\Delta_{PH}} \quad (6)$$

In order to make an ant decision based on this distribution, the Selector from the basic layout in Fig. 8b must be modified as well, since we now choose from four sets of items with different weights instead of two. These modifications, however, are essentially only doing two further subtractions and expanding the multiplexer which chooses the address of the S-Cell. An efficient parallel implementation of the Match Buffers and the Selectors on an FPGA allows to perform an ant decision in $\Theta(k + t)$ time, where k and t can be considered as small constants.

8. Conclusion

We have presented a mapping of population-based ant colony optimization to an FPGA architecture. In doing so, we have implemented new ways for dealing with the pheromone information, which asymptotically have led to significant improvements in runtime and area requirements in comparison to the standard ACO algorithm if it was implemented in hardware. Test results of an FPGA implementation for the SMTTP problem have shown a considerable speedup over a software implementation, especially for a large number of ants per iteration. Furthermore, we have shown possibilities for compacting the algorithm and including heuristic information in the process

of constructing the solutions without asymptotically increasing runtime or required space.

Our future work will include the implementation of compaction techniques and item × item encoded problem instances (e.g. TSP). Furthermore, the effect of the discretized heuristic on solution quality will be investigated.

Acknowledgements

This work was supported by the Xilinx University Programme and the International Office (IB/DLR) of the German Ministry of Education and Research (BMBF) within the scope of WTZ-project AUS 00/002.

References

[1] J.L. Deneubourg, J.M. Pasteels, J.C. Verhaege, Probabilistic behavior in ants: a strategy of errors? J. Theor. Biol. 105 (1983) 259–271.

[2] R. Beckers, J.L. Deneubourg, S. Goss, Trails and U-turns in the selection of the shortest path by the ant Lasius Niger, J. Theor. Biol. 159 (1992) 397–415.

[3] M. Dorigo, V. Maniezzo, A. Colorni, Positive Feedback as a Search Strategy, Tech. Rep. 91-016, Politecnico di Milano, Italy, 1991.

[4] M. Dorigo, Optimization, Learning and Natural Algorithms (in Italian), Ph.D. Thesis, Dipartimento di Elettronica, Politecnico di Milano, 1992.

[5] M. Dorigo, G. Di Caro, The ant colony optimization meta-heuristic, in: D. Corne, M. Dorigo, F. Glover (Eds.), New Ideas in Optimization, McGraw-Hill, 1999, pp. 11–32.

[6] D. Merkle, M. Middendorf, H. Schmeck, Ant colony optimization for resource-constrained project scheduling, IEEE Trans. Evolut. Comput. 6 (4) (2002) 333–346.

[7] L.M. Gambardella, E.D. Taillard, G. Agazzi, MACS-VRPTW: a multiple ant colony system for vehicle routing problems with time windows, in: New Ideas in Optimization, McGraw Hill, London, UK, 1999, pp. 63–76.

[8] C. Solnon, Ants can solve constraint satisfaction problems, IEEE Trans. Evolut. Comput. 6 (4) (2002) 347–357.

[9] L.-M. Gambardella, E. Taillard, M. Dorigo, Ant colonies for the quadratic assignment problem, J. Operat. Res. Soc. 50 (1999) 167–176.

[10] M. Bolondi, M. Bondaza, Parallelizzazione di un algoritmo per la risoluzione del problema del comesso viaggiatore, Master's thesis, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1993.

[11] M. Dorigo, Parallel ant system: an experimental study, unpublished manuscript, 1993.

[12] B. Bullnheimer, G. Kotsis, C. Strauss, Parallelization strategies for the ant system, in: R.D. Leone, A. Murli, P. Pardalos, G. Toraldo (Eds.), High Performance Algorithms and Software in Nonlinear Optimization, Vol. 24 of Applied Optimization, Kluwer, 1998, pp. 87–100.

[13] E.-G. Talbi, O. Roux, C. Fonlupt, D. Robillard, Parallel ant colonies for combinatorial optimization problems, in: J.R. et al. (Eds.), Parallel and Distributed Processing, 11 IPPS/SPDP'99 Workshops, no. 1586 in LNCS, Springer-Verlag, 1999, pp. 239–247.

[14] S. Iredi, D. Merkle, M. Middendorf, Bi-criterion optimization with multi colony ant algorithms, in: E.Z. et al. (Eds.), Evolutionary Multi-Criterion Optimization, First International Conference (EMO'01), LNCS 1993, Springer-Verlag, 2001, pp. 359–372.

[15] D. Merkle, M. Middendorf, Fast ant colony optimization on runtime reconfigurable processor arrays, Genet. Programming Evol. Machines 3 (4) (2002) 345–361.

[16] M. Rahoual, R. Hadji, V. Bachelet, Parallel ant system for the set covering problem, in: Ant Algorithms, Proceedings of Third International Workshop ANTS 2002, LNCS 2463, Springer-Verlag, Brussels, Belgium, 2002, pp. 262–267.

[17] M. Middendorf, F. Reischle, H. Schmeck, Multi colony ant algorithms, J. Heuristics 8 (3) (2002) 305–320.

[18] M. Randall, A. Lewis, A parallel implementation of ant colony optimization, J. Parallel Distrib. Comput. 62 (9) (2002) 1421–1432.

[19] R. Miller, V.K. Prasanna-Kumar, D.I. Reisis, Q.F. Stout, Parallel computations on reconfigurable meshes, IEEE Trans. Comput. 42 (6) (1993) 678–692 (a preliminary version of this paper was presented at 5th MIT Conference on Advanced Research in VLSI, 1988).

[20] O. Diessel, H. ElGindy, M. Middendorf, M. Guntsch, B. Scheuermann, H. Schmeck, K. So, Population based ant colony optimization on FPGA, in: IEEE International Conference on Field-Programmable Technology (FPT), 2002, pp. 125–132.

[21] O. Cheung, P. Leong, Implementation of an FPGA based accelerator for virtual private networks, in: IEEE International Conference on Field Programmable Technology (FPT), Hong Kong, 2002, pp. 34–43.

[22] J. Gause, P. Cheung, W. Luk, Static and dynamic reconfigurable designs for a 2D shape-adaptive DCT, in: R. Hartenstein, H. Grüunbacher (Eds.), Field-Programmable Logic and Applications, FPL, Springer-Verlag, 2000, pp. 96–105.

[23] J. Villasenor, C. Jones, B. Schoner, Video communications using rapidly reconfigurable hardware, IEEE Trans. Circuits Syst. Video Technol. (1995) 565–567.

[24] A. Hämäläinen, M. Tommiska, J. Skyttä, 6.78 gigabits per second implementation of the IDEA cryptographic algorithm, in: M. Glesner, P. Zipf, M. Renovell (Eds.), Field-Programmable Logic and Applications. Reconfigurable Computing Is Going Mainstream, LNCS 2438, Springer-Verlag, 2002, pp. 760–769.

[25] S. Bade, B. Hutchings, Fpga based stochastic neural network implementation, in: Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, 1994, pp. 189–198.

[26] P. Lysaght, J. Stockwood, J. Law, D. Girma, Artificial neural network implementation on a fine-grained fpga, in: Field-Programmable Logic, 1994, pp. 421–431.
[27] R. Gadea, J. Cerd, F. Ballester, A. Mochol, Artificial neural network implementation on a single fpga of a pipelined on-line backpropagation, in: Proceedings of the 13th Conference on International Symposium on System Synthesis, 2000, pp. 225–230.
[28] P. Graham, B. Nelson, Genetic algorithms in software and in hardware—a performance analysis of workstation and custom computing machine implementations, in: IEEE Symposium on FPGAs for Custom Computing Machines, 1996, pp. 216–225.
[29] I.M. Bland, G.M. Megson, The systolic array genetic algorithm, an example of systolic arrays as a reconfigurable design methodology, in: IEEE Symposium on FPGAs for Custom Computing Machines, 1998, pp. 260–261.
[30] C. Aporntewan, P. Chongstitvatana, Hardware implementation of the compact genetic algorithm, in: Proceedings of the 2001 IEEE Congress on Evolutionary Computation, Seoul, South Korea, 2001, pp. 624–629.
[31] M. Guntsch, M. Middendorf, A population based approach for ACO, in: S. Cagnoni et al. (Eds.), Applications of Evolutionary Computing—EvoWorkshops 2002: EvoCOP, EvoIASP, EvoSTIM/EvoPLAN, no. 2279 in Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 72–81.
[32] Xilinx, Virtex 2.5V FPGA Complete Data Sheet, http://direct.xilinx.com/bvdocs/publications/ds003.pdf (2001).
[33] Xilinx, Virtex-II Platform FPGA, complete data sheet, http://direct.xilinx.com/bvdocs/publications/ds031.pdf (2002).
[34] Xilinx, Virtex-II Pro Platform FPGA Complete Data Sheet, http://direct.xilinx.com/bvdocs/publications/ds083.pdf (2003).
[35] J. Du, J.Y.-T. Leung, Minimizing total tardiness on one machine is NP-hard, Math. Oper. Res. 15 (1990) 483–485.
[36] M. Guntsch, M. Middendorf, Applying population based aco to dynamic optimization problems, in: M. Dorigo et al. (Eds.), Ant Algorithms: 3rd International Workshop, ANTS2002, Vol. 2463 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 111–122.
[37] M. Dorigo, V. Maniezzo, A. Colorni, The ant system: optimization by a colony of cooperating agents, IEEE Trans. Syst. Man Cybern.Part B 26 (1996) 29–41.
[38] J. Ackermann, U. Tangen, B. Bödekker, J. Breyer, E. Stoll, J. McCaskill, Parallel random number generator for inexpensive configurable hardware cells, Comput. Phys. Commun. 140 (3) (2001) 293–302.
[39] M. Dorigo, G. Di Caro, L.M. Gambardella, Ant algorithms for discrete optimization, Artificial Life 5 (2) (1999) 137–172.