A New Approach for Task Level Computational Resource Bi-partitioning

Gang Wang Wenrui Gong Ryan Kastner

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

ABSTRACT
An essential problem for hardware/software codesign is the partitioning of an application onto the computational resources. This paper presents a novel approach for the task level resource partitioning problem. Our approach is based on the Ant System algorithm, a meta-heuristic method inspired by the study of the behaviors of ants. In our algorithm, a collection of agents cooperate using distributed and local heuristic information to effectively explore the search space. An iterative stochastic decision making process is carried by the agents in order to optimally allocate tasks onto either the general purpose processor or the reconfigurable logic. Experiments show that the proposed algorithm provides robust results that are qualitatively close to the optimal with minor computational cost.

KEY WORDS
Embedded systems, resource allocation, ant system algorithm, hardware/software codesign

1 Introduction

An emerging trend in the design of embedded systems is the use of the architectures that couple a general purpose processor (GPP) with configurable logic [4, 2, 22] or ASIC unit such as an application specific DSP core. The two different types of computing resources have complementary characteristics making them an ideal pairing. The GPP performs admirably on control intensive application code with limited parallelism, while the configurable logic or ASIC unit can efficiently handle data intensive code with a large amount parallelism.

In order to bring about widespread acceptance of this architecture, we must automate the compilation of application code onto this configurable processor. An essential compilation task for is the allocation of the application onto the heterogeneous computation resources.

The partitioning problem is a fundamental task for embedded system. Partitioning algorithms assign application tasks to the system computing resources while optimizing system performance metrics such as the hardware cost, power consumption and worst case execution time. It is considered an integral part of the hardware/software codesign problem. Some early investigations on the partitioning problem include [9, 11, 18, 19]; it is difficult to name a clear winner [8]. More recently, partitioning issues for reconfigurable architectures have been studied [3, 12, 16]

The partitioning problem is $\mathcal{NP}$-complete [10]. It is possible to use brute force search or integer linear programming (ILP) formulations for small problem instances. However, in general, the optimal solution is computationally intractable. Various heuristics have been proposed to solve the partitioning problem. Ernst *et al.* [9] used Simulated Annealing (SA) in their Cosyma system. A global criticality/local phase driven algorithm [14] achieved results qualitatively close to the optimal. Hidalgo *et al.* [13] constructed a variant of Genetic Algorithm (GA) for the partitioning problem. Kernighan/Lin (KL) heuristic was utilized in [1]. Vahid *et al.* extended the KL heuristic [20] and they reported extremely fast execution times with results comparable with simulated annealing. Recently, the authors of [21] compared three popular heuristics for task level partitioning problem and concluded Tabu search performed better than SA or GA.

In this paper, we present a novel heuristic searching approach for the task level partitioning problem. Our approach is based on the *Ant System* (AS) algorithm [7]. In the proposed algorithm, a collection of agents cooperate together to search for a good partitioning solution. Our method can be extended to many generic partitioning problems.

We will focus our discussion under the context of hardware/software codesign, namely a system containing a GPP and another computing unit, either an ASIC design or a configurable logic such as FPGAs. The remainder of the paper is organized as follows. Section 2 gives a brief introduction to the AS algorithm. Section 3 details the proposed algorithm for the task level partitioning problem for hardware/software codesign. Section 4 presents our experimental results. We conclude and give some thoughts on future work in Section 5.

2 Ant System Algorithm

The ant search algorithm, originally introduced by Dorigo *et al.* [7], is a cooperative heuristic searching algorithm inspired by the ethological study on the behavior of ants. It was observed that ants – who lack sophisticated vision – could manage to establish the optimal path between their colony and the food source. The study [5] found that the ants use pheromone trails to communicate information amongst themselves. Though any single ant moves essentially at random, it will make a decision on its direction based on the "strength" of the pheromone on the paths that lie before it. As an ant traverses a path, it reinforces that path with its own pheromone.The pheromone evaporates over time; therefore the "shortest" paths will maintain a higher amount of pheromone as opposed to the "longer" paths. A collective autocatalytic behavior emerges as more ants will choose the shortest trails (because the short trails have a higher amount of pheromone), which in turn creates an even larger amount of pheromone on those short trails. This means that those short trails will be even more likely to be chosen by future ants.

The AS algorithm is inspired by such observation. It is a population based approach where a collection of agents cooperate together to explore the search space. They communicate via a mechanism imitating the pheromone trails. One of the first problems to which AS was successfully applied was the Traveling Salesman Problem (TSP) [7]. The AS algorithm gave competitive results comparing with traditional methods.

3 AS for Hardware/Software Codesign

A crucial step in the design of hardware/software codesign is the allocation of the application's computation onto the general purpose processor and configurable logic or ASIC unit. The partitioning plays a dominant role in the system cost and performance.

It is possible to perform partitioning at multiple levels of abstraction. For example, you can partition the application at the operation (instruction) level as is done in the Garp project [4]. We focus on partitioning at a higher abstraction level – the task or functional level. At this level, the atomic partitioning unit is a task. A *task* is a coarse grained set of computation with a well defined interface. Our work assumes that each task is specified as a C function. The tasks themselves may depend on the execution of other tasks. The dependencies are modeled using a task graph.

A *task graph* is a directed acyclic graph (DAG) $G = (T, E)$, where $T = (t_0, t_1, \ldots, t_n)$ is a collection of task nodes, and E is a set of directed edges. Each task node defines a functional unit for the program, which contains information about the computation the task 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 node t_i and t_j ($t_i \longrightarrow t_j$).

A task can be executed only after all the tasks with a higher precedence level have been executed. If the system contains both a GPP and configurable logic, the partitioning of the tasks onto the two resources becomes critical to the system performance. There are 2^N unique partitioning solutions, where N is the number of the tasks. However, some of these solutions may be infeasible as they violate system constraints. For example, a partitioning solution may allocate a large number of tasks to the configurable logic. However, the configurable logic has a fixed size, and the area occupied by those tasks must be less than the area of the configurable logic.

Our work applies partitioning at the functional task level for sequential scheduling. It focuses on minimizing the critical path of a task graph given area constraints on the size of the configurable logic. The tasks of different precedence levels are sequentially executed from the top level down, while tasks in the same precedence level can be allocated on different system components and run concurrently. Each task can be allocated either on the GPP or the configurable logic. In the remainder of this section, we will present a new heuristic algorithm based on AS for task level partitioning in hardware/software codesign. We will first discuss the mathematic model that forms the basis for the algorithm.

We model task graph partitioning as a bi-coloring problem. We associate the color c_1 with the GPP, and the color c_2 with the configurable logic. The partitioning problem finds the "optimal" coloring of the task nodes (excluding the virtual task nodes t_0 and t_n) using c_1 and c_2 such that the task graph provides the shortest execution time subject to a fixed amount of configurable logic. In order to effectively explore the search space, we also associate each edge in the task graph with the color. An edge e_{ij} can be colored with either c_1 or c_2. Each edge e_{ij} is associated with two global heuristics, denoted $\tau_{ij}(k)$, to indicate the likelihood for task t_j to be colored with the corresponding color c_k, where $k = 1, 2$.

We further model the partitioning problem as an agent based stochastic decision making process. An agent tries to construct a coloring on the task graph based on the distributed and local heuristics. For every task node t_i, two steps are carried out by the agent. First, the agent makes a decision on how the task node t_i should be colored. This decision is made by considering the choices made on t_i by its immediate predecessors. Once this decision is made, the agent forces every inbound edge of t_i to be colored the same color as t_i. That is, for every task node t_l that is an immediate predecessor of t_i, the edge e_{li} will be colored with the same color t_i - the current node under consideration. Then the agent guesses the color of each task node that immediately follows t_i, i.e. for every edge e_{ij} it guesses the color of t_j. The probability of assigning color c_k to t_j is based on a global heuristic $\tau_{ij}(k)$ and the local heuristic based on the local information from the task nodes t_i and t_j. We will describe our global and local heuristics in the following section.

Figure 1 gives an example of a bi-coloring of a task graph. We use a solid line to indicate the items colored with c_1 and dotted line for c_2. Consider task node t_3. When an agent encounters t_3, the edge e_{13} will already be colored based on the agent's previous decision at task node t_1. Since this is there is only one incoming edge for this node, t_3 will be assigned the color c_2, which corresponds to assigning task t_3 to the configurable logic. Then, we will use the local and global heuristics to assign a color to the edges e_{34} and e_{35}. Tasks 1, 2, 7 and 8 are partitioned on the GPP and task 3, 4, and 6 are allocated on the configurable logic. The inbound edges for each of the nodes are colored accordingly. As task node t_n is a virtual node, we do not care about the coloring of edge e_{8n}.

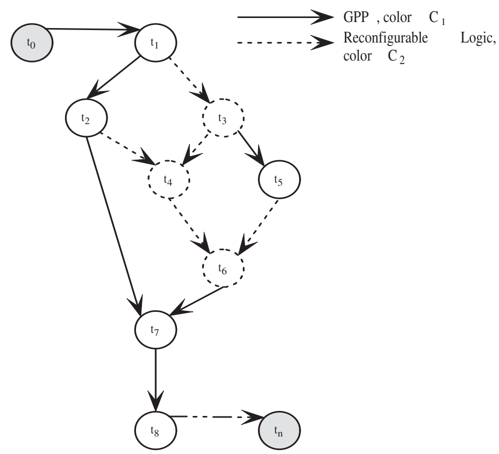

Figure 1: Partitioning as a graph bi-coloring problem

This stochastic process will be repeated multiple times. During the first iteration of the algorithm, the agent will perform random walk guided solely by the local heuristic because the global heuristic is based on the solution of previous iterations. Being that this is the first iteration, there is not yet any global information. After a coloring is created for the task graph, we calculate the quality of this coloring. Then for each e_{ij}, if it is colored by c_k, we will update the associated heuristic $\tau_{ij}(k)$. The hope is that by rewarding the distributed global heuristic, the agent will have better chance to find the optimal solution for the partitioning problem.

Based on the bi-coloring model, we introduce a new heuristic method for solving the computational resource partitioning problem using the AS algorithm. Each agent traverses the task graph and attempts to create a feasible bi-coloring by selecting the colors probabilistically according to the combined heuristics. The quality of the solution is measured by the overall execution time of the colored task graph together with the consideration of the predefined constraints, such as hardware area cost limit. The quality measurement is used to reinforce the *best* solutions. The global heuristic information is distributed as pheromone trails on the edges of the task graph.

The proposed algorithm proceeds as follows:

1. Initially, assign each of the edges in the task graph with a fixed pheromone τ_0 for both color c_1 and c_2, where c_1 corresponds to GPP, while c_2 for configurable logic;
2. Put m ants on t_0;
3. Each ant traverses the task graph to create a feasible bi-coloring solution s^i for the task graph, where $i = 1, \ldots, m$;
4. Evaluate all the m solutions. The quality of the solution s is measured by the overall execution time $time_s$. Among all solutions, find the best solution s_{best} which provides the minimum execution time and satisfies the configurable logic area constraint;
5. Update the pheromone for each color on the edges as follows:

$$\tau_{ij}(k) \leftarrow (1-\rho)\tau_{ij}(k) + \Delta\tau_{ij}(k)$$

where $0 < \rho < 1$ is the evaporation ratio, $k = 1, 2$, and

$$\Delta\tau_{ij}(k) = \begin{cases} Q/time_{s_{best}} & e_{ij} \text{ is colored with } c_k \text{ in } s_{best} \\ 0 & \text{otherwise} \end{cases}$$

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 commands how an individual ant "crawls" over the task graph to generates a solution. Two questions must to be addressed in this step: 1) how does the ant handle the precedence constraints between task nodes? 2) what heuristics should the ant use and how should it apply them?

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 assignment or coloring. By visiting the nodes in topologically sorted order, we insure that every predecessor node is visited before we visit the current node. Equivalently, it guarantees that every incoming edge to the current node is already colored.

At each task node t_i where $i \neq n$, the ant makes a probabilistic decision on the coloring for each of its successor task nodes t_j based on the pheromone on the edge More specifically, an ant at task node t_i guesses that node t_j is colored with c_k according to the probability:

$$\mathsf{p}_{ij}(k) = \frac{\tau_{ij}(k)^{\alpha}\eta_j(k)^{\beta}}{\sum_{l=1,2}\tau_{ij}(l)^{\alpha}\eta_j(l)^{\beta}}$$

Here $\eta_j(k)$ is the local heuristic if t_j is colored with c_k. In our work, we simply use the inverse of the cost of having task t_j assigned to either the GPP or the configurable logic. A simple weighted combination is used to estimate the cost: $cost = w_t \cdot time_j + w_a \cdot area_j$, where $time_j$ and $area_j$ 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. The constants α and β are used to weight the local and global heuristics.

Upon entering a new node t_i, the ant makes a decision on the coloring of the task node t_i based on the guesses made by all of the immediate precedents of t_i. It is guaranteed those guesses are already made since that the ant travels the task graph in a topologically sorted manner. In our implementation, this decision is made probabilistically based on the distribution of the guesses, i.e. the possibility of coloring t_i with c_k is:

$$\mathsf{p}_i(k) = \frac{\text{count of guess } c_k \text{ for } t_i}{\text{count of immediate precedents of } t_i}$$

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

In each run of the algorithm, multiple iterations of the above steps are conducted. Two ending conditions are explored: 1) the algorithm ends after a fix number of iterations, or 2) the algorithm ends when there is no improvement found after a certain number of iterations.

For each iteration, the ant search has a run time Ant_t confined by $O(2N^2)$, where N is the number of nodes in the task graph. For a run with I iterations using m ants, the time complexity for our algorithm is: $(Ant_t + E_t) * m * I$, where E_t is the time it takes to evaluate the quality of each solution. In practice, we found that $E_t \gg Ant_t$. Compared with brute force search that has a total run time of $(2^N) * E_t$, the speedup ratio we achieved by the AS algorithm is:

$$\mathrm{speedup} = \frac{(2^{\mathrm{N}}) * \mathrm{E_t}}{\mathrm{m} * \mathrm{I} * (\mathrm{Ant_t} + \mathrm{E_t})} \approx \frac{2^{\mathrm{N}}}{\mathrm{m} * \mathrm{I}}$$

Finally, we propose two possible ways to determine the number of ants, m, used in each iteration of the algorithm: 1) m equals the average branching factor of the task graph, or 2) m equals the maximum branch number of the task graph.

4 Experimental Results

Our experiments address the functional partitioning problem on a design using configurable processors targeting multimedia and communication applications. The target system contains one general purpose processor (PowerPC 405 RISC CPU core) and configurable computing logic (Xilinx VirtexII with 1,232 CLB capacity). The reference model is Xilinx Virtex II Pro Platform FPGA [22], which can contain up to four CPU cores, 13,404 CLBs and other components. Although our algorithm can extended for a system architecture with more computing resources, we restrict our system to contain one processor and configurable logic.

The execution time required to run a task on either resource depends on how the task is implemented. We assume that the tasks are static and precomputed. The communication time between resources are also predefined.

The tasks allocated on the GPP are sequentially executed subject to the precedence constraints. Task level parallelism between the GPP and configurable logic is explored, i.e. independent tasks may run concurrently on the GPP and configurable logic. Furthermore, instruction level parallelism within a particular task is exploited when the task is mapped to the configurable logic. However, no hardware reuse between tasks mapped to the configurable logic is considered since that it requires sophisticated scheduling that will greatly affect the execution time. This makes it difficult to evaluate the performance of our proposed algorithm. A predefined configurable logic area, i.e. predetermined number of CLBs, is used to judge whether a particular partition is feasible. For all the feasible partitions, i.e. the partitions that do not exceed this area cost, the best partition is the one with the shortest execution time.

Our experiments are conducted in a hierarchical environment for the target design. At the highest level of representation, an application is represented as a task graph. The task graph, as described in Section 3, is a directed acyclic graph, which shows the precedence relationship between tasks. Each task node is mapped to a function, which are written in a high level programming language, such as C/C++. The functions are analyzed using the SUIF and Machine SUIF tools. The resulting compiled function is imported in our environment as a control/data-flow graph (CDFG). A CDFG reflects the control flow within a function, and may contain loops, branches, and jumps. Each node in the CDFG is a basic block, i.e. a set of instructions that contains only one control-transfer instruction, and several arithmetic, logic, and memory instructions.

Estimation is carried out for each task node to get performance characteristics, such as execution time, software code length and hardware area. Based on the specification data of Virtex II Pro Platform FPGA [22], we get the typical performance characteristics for every operation, which are used estimate the performance of each basic block. The execution counts and the execution time of all basic blocks in a CDFG determine the execution time of the entire task. Using the HALT profiling tool included with Machine SUIF, we import profiling results of the MediaBench functions from the representative input data included with the benchmark suite. The maximum execution counts of those basic blocks are added into an ILP formulation with the objective of minimizing the total execution time; the ILP is solved using lp solve.

There are two types of commonly used scheduling approaches: pipelined scheduling and sequential scheduling. Partitioning for the pipelined scheduling and sequential scheduling can be very different. This is determined by the different objectives for these two approaches: while the pipelined scheduling tries to obtain the best pipelined stages and satisfy the throughput requirement of the system, sequential scheduling is concerned to minimize the total execution time of the task graph. The work reported in this paper only involves sequential scheduling as we target to achieve the best execution time under hardware cost constraint. That is for tasks of different level of precedence, they must be executed sequentially, while task with the same precedence level can run concurrently as long as the given resource partitioning allows. This allows us to determine the minimum execution time of the entire task graph by performing a critical path-based scheduling over a given partitioning. As no hardware reuse between the tasks is assumed, we calculate the hardware cost and software code length by simply summing these parameters from the individual task nodes. We schedule tasks based their dependencies given by the task graph. The scheduler reports the worst case execution time of the task. The synchronization time between the GPP and configurable logic and configuration time of the configurable logic is integrated as the task's execution time. Both the synchronization time and configuration time are configurable in our framework. We currently assume that both are minimal compared to the overall task execution time. The effect of varying these parameters is an interesting study, however it is out of the scope of this work.

Each testing example in our experiments is constructed as an application running on the previously described system architecture. An application is is formed using a two step process that combines randomly generated DAGs (corresponding to the task graph) with real life software functions for each task node of the task graph. The reasons for using randomly generated task graphs are: 1) there does not exist a widely accepted benchmark for the task level resource partitioning problem; 2) it has been a practice for the researchers in this field to use randomly generated task graphs for evaluating different partitioning algorithms, such as indicated by [21, 6, 14]. Our benchmarks characterize the tasks using realistic software functions extracted from real application instead synthetic task descriptions.

First, for a given number of task nodes, a set of random DAGs are created using GVF tool kit [17]. 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. We use these DAGs as the task graphs of the applications. In our experiments, the generated DAG has 25 task nodes and the average branching factor is 5. This gives a search space with the size of 2^{25}, i.e. over 33 million possible partitions. Benchmarks with smaller task graph sizes of 13, 15, and 20 nodes are also generated and tested.

We associate a function with each task node in the task graph. We use the MediaBench benchmark suite [15] as the resource for selecting these functions. For a given task graph, the functions are picked from one application within the benchmark suite.

A brute force search is done for each generated task graph. It provides important insights to the search space, such as the number of the optimal partitions with minimal execution time and the distribution of all the feasible partitions. However, it is computationally expensive and is the bottleneck in choosing the size of the task graphs (For example, on a Pentium machine with 2.8GHz CPU, it takes more than 100 hours to conduct brute force search on a benchmark of 50 DAGs and each contains 25 task nodes). Based on the results of brute force search, trivial examples, for which the number of the optimal partitions is statistically significant, are eliminated. After removing the "easy" partitioning problem instances, our testing suite contains 25 task graphs.

In our experiments, we apply the proposed AS algorithm on the testing example set and evaluate its results with statistics computed via the optimal, though computationally expensive, brute force search. Since the AS algorithm is a stochastic process, we run the AS algorithm 100 times for each task in order to obtain a large enough sample space. For each run, the number of ants per iteration, m, is set as the average branch factor of the DAG. We force the algorithm stop after 100 iterations. 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 $\tau_0 = 100$, $Q = 1,000$, $\rho = 0.2$, $\alpha = \beta = 1$, $w_t = 1$ and $w_a = 2$. We didn't extensively test the sensitivity of our algorithm with regard to different parameter settings. However, our results suggests that these values are robust over a wide range of testing examples.

Figure 2 shows the quality distribution for the result partitions achieved by using the proposed AS algorithm. Curve (a) shows the cumulative distribution of the number of solutions found by the AS algorithm plotted against the quality of those solutions. 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%, less than 23% of the solutions that the AS algorithm found were outside of the top 2% of the overall number of solutions. In other words, over 77% of the solutions found by the AS algorithm are within 2% of all possible partitions. The number of solutions drops quickly showing that the AS algorithm finds very good solutions in almost every run. 1,586 solutions, or 63.5% of all the solutions, found by our algorithm are within the top 0.1% range. The figure indicates that a majority of the results are qualitatively close to the optimal. In our experiments, totally 2,290 solutions, or 91.7% of all the solutions, are within the top 3% range.

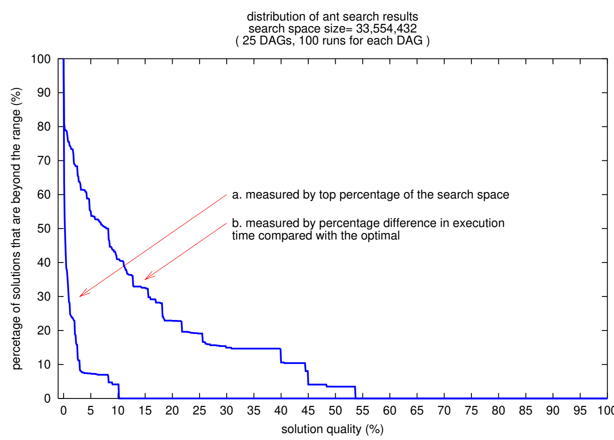

Figure 2: Quality for the result partitions provided by AS algorithm

We also try to evaluate the capability of the algorithm with respect to discovering the optimal partition. Out of a total of 2,500 results over 25 task graphs, the AS algorithm found the optimal execution time 460 times. Based on this, the probability of finding the optimal solution with our algorithm for these task graphs is 18.4%. With the same amount of computation time, a random sampling method has a chance of 8.5×10^{-7} to discover the optimal solutions for the same examples, which is close to zero. Therefore, the AS algorithm is statistically much more effective in finding the optimal solution than random sampling. Also, we found that for 5 testing examples, or 20% of the testing set, our algorithm discovers the optimal partition in more than half of the 100 runs. In other words, for a significant percentage of the testing examples, the algorithm has high possibility in producing one of the optimal partitions.

Curve (b) in Figure 2 provides another perspective regarding to the quality of our results. Here, the x axis is interpreted as the percentage difference comparing the execution time of the partition found by the AS 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 curve (a). The results in curve (a) show how well the AS algorithm finds solutions that are within a top percentage of overall solutions, while curve (b) measures the solution quality by directly comparing the result execution time with that of the optimal partition. These results differ because while the AS 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. Regardless, the quality of the solutions that we find are still very good. The majority 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 100, for a computational resource partitioning problem over a 25 node task graph, the proposed algorithm has a run time about 0.0014% of that using brute force search, or about 67,000 times faster. On a Linux machine with with a 2.80 GHz Intel Pentium IV CPU with 512 MByte memory, the average actual execution time for the brute force method is 130 minutes while, on average, our AS algorithm runs for 0.072 seconds. These runtimes are in scale with the theoretical speedup reported in Section 3. To summarize the experiment results, with a high probability (91.7%), we can expect to achieve a result within top 3% of the search space with a very minor computational cost. For benchmarks with smaller task node numbers (13, 16 and 20 nodes), our algorithm scales well and achieves good results similar to that obtained for the 25 node task graphs.

5 Conclusion

In this work, we presented a novel heuristic searching method for the task level computational resource partitioning problem based on the Ant System algorithm. The algorithm works as a collection of agents work collaboratively to explore the search space. A stochastic decision making strategy is proposed in order to combine global and local heuristics to effectively conduct this exploration. The graph bi-coloring model is proposed as the basis for the algorithm, which can be easily extended to handle multi-way partitioning problems. The results over task graph benchmark set shows extremely promising results. The proposed algorithm consistently provided near optimal partitioning results over test examples with very minor computational cost. Compared with random sampling method, our algorithm is significantly more effective in finding the optimal solution.

References

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

[2] P. M. Athanas and H. F. Silverman. Processor Reconfiguration through Instruction-set Metamorphosis. *Computer*, 26(3):11–18, 1993.

[3] 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.

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

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

[6] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task graphs for free. *Proc. Int. Workshop Hardware/Software Codesign*, pages 97–101, 1998.

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

[8] S. A. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models Validation, and Synthesis. *Proceedings of the IEEE*, 85(3):366–90, March 1997.

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

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] T. Wiangtong, P. Cheung, and W. Luk. Comparing three heuristic search methods for functional partitioning in hardware-software codesign. *Journal of Design Automation for Embedded Systems*, 6:425–449, 2002.

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