UNIVERSITY OF CALIFORNIA

Santa Barbara

Ant Colony Metaheuristics for Fundamental Architectural Design Problems

A Dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Electrical and Computer Engineering

by

Gang Wang

Committee in charge:

Professor Ryan Kastner, Chair

Professor Malgorzata Marek-Sadowska

Professor Steve Butner

Professor Elaheh Bozorgzadeh

Professor Timothy Sherwood

September 2007

The dissertation of Gang Wang is approved:

Chair Date

Date

Date

Date

Date

University of California, Santa Barbara

September 2007

Ant Colony Metaheuristics for Fundamental Architectural Design Problems

by

Gang Wang

To my wife Fang Liu,

my children James, Justin, Jocelyn,

and my parents Tihao Wang and Shuchun Li.

Abstract

Ant Colony Metaheuristics for Fundamental Architectural Design Problems

by

Gang Wang

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Santa Barbara

Professor Ryan Kastner, Chair

As the fabrication technology advances, the number of transistors in modern computer systems keeps growing exponentially. This growth creates tremendous potential while imposing big challenges to the electronic design automation community on how to construct such complicated systems. In order to effectively utilize the computing resources, a number of fundamental problems need to be addressed. As the scale and complexity of these problems grow, we must look towards new optimizations methods, rather than simply perform iterative improvements on existing techniques.

In this dissertation, I will report our research work on constructing new heuristic algorithms using the Ant Colony metaheuristic for effectively and efficiently solving a range of difficult architectural design problems. We investigate three $\mathcal{NP}$-hard problems in this context, namely system partitioning, operation scheduling and design space exploration. Results show that Ant Colony metaheuristic is a very promising approach for solving these problems, and the algorithms we propose generally achieve better quality solutions with much improved stability compared to conventional methods.

Moreover, by establishing the theoretical link between timing and resource constrained scheduling, we propose an effective design exploration framework that leverages duality between the scheduling problems. To our best knowledge, our work is the first extensive study on applying the Ant Colony metaheuristics to the architectural design field.

Acknowledgments

I would like to show my sincerest gratitude to my advisor Professor Ryan Kastner for giving me copious amounts of insightful guidance, constant encouragement, constructive criticism, and expertise on every subject that arose throughout all these years. His enthusiasm and dedication to his students are truly inspiring; it is my very privilege to have been one of them. I would also like to thank Professor Malgorzata Marek-Sadowska, Professor Elaheh Bozorgzadeh, Professor Steven Butner, and Professor Tim Sherwood for being on my Ph.D. committee and for all their helps along the way.

I am very thankful for the many friends and fellow students I have had at UCSB, including Anup Hosangadi, Yen Meng, Brian DeRenzi, for their many stimulating discussions and warm friendship. Particularly, I want to thank Wenrui Gong, who I collaborated with on numerous research efforts and became close friend with over the years. Without them, my experience at UCSB won't be as rewarding as it is.

Finally, I want to thank my wife, Fang Liu, for her love, support, encouragement, sense of humor, and for being an intelligent partner on my research journey as well. I could not have accomplished this without her. Very special thanks to my parents for their selfless love and support, and to my lovely children James, Justin and Jocelyn.

Curriculum Vitæ

Gang Wang was born in Shaanxi, China in 1971. He received the Bachelor of Electrical Engineering degree from Xian Jiaotong University in 1992, and Master of Computer Science degree from Chinese Academy of Sciences in 1995, both in China. From 1995 to 1997, he conducted research work at Michigan State University (East Lansing, MI, US), and Carnegie Mellon University (Pittsburgh, PA, US), focusing on speech and image understanding. Since 1997, Mr. Wang held positions as software architect and technical manager in different leading companies of medical industry, including Computer Motion, Intuitive Surgical and Karl Storz Endoscope. His work focused on the research and development of complex surgical robotics systems, multi-modal human-computer interaction and intelligent operating room. His current research interests include reconfigurable and embedded computing, optimization algorithms and their applications, novel architectural design for FPGA and nanocomputing platforms, ubiquitous computing and its applications in medical/healthcare systems. He has authored or co-authored more than 20 technical papers in different journals and conferences on related topics.

Related Publications:

BOOKS/BOOK CHAPTERS

[B1] **Gang Wang**, Wenrui Gong, and Ryan Kastner. *Operation Scheduling: Algorithms and Design Space Exploration*, to appear in *High Level Synthesis Handbook: The State of Arts* published by Springer.

JOURNAL ARTICLES

[J1] **Gang Wang**, Wenrui Gong, Brian DeRenzi, and Ryan Kastner. *Exploring Time/Resource Tradeoffs by Solving Dual Scheduling Problems with the Ant Colony Optimization*, accepted by *ACM Transactions on Design Automation of Electronic Systems* (TODAES).

[J2] **Gang Wang**, Wenrui Gong, Brian DeRenzi, and Ryan Kastner. *Ant Scheduling Algorithms for Resource and Timing Constrained Operation Scheduling, IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems* (TCAD), Vol 26, Issue 6, pp 1010-1029, 2006.

[J3] **Gang Wang**, Satish Sivaswamy, Cristinel Ababei, Kia Bazargan, Ryan Kastner and Eli Bozorgzadeh. *Statistical Analysis and Design of HARP Routing Pattern FPGAs, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (TCAD), Vol 25, Issue 10, pp 2088-2102, October 2006.

[J4] **Gang Wang**, Wenrui Gong and Ryan Kastner, *Application Partitioning on Programmable Platforms Using the Ant Colony Optimization, Journal of Embedded Computing* (JEC), Vol 2, Issue 1, pp 119-136, 2006.

PEER-REVIEWED CONFERENCE/WORKSHOP PAPERS

[C1] Ted Huffmire, Brett Brotherton, **Gang Wang**, Ryan Kastner, and Tim Sherwood. *Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems, IEEE Symposium on Security and Privacy*, 2007.

[C2] **Gang Wang**, Wenrui Gong, and Ryan Kastner, *On the Use of Bloom Filters for Defect Maps in Nanocomputing*, In *International Conference on*

Computer-Aided Design (ICCAD), 2006.

[C3] **Gang Wang**, Wenrui Gong, Brian DeRenzi and Ryan Kastner, *Design Space Exploration using Time and Resource Duality with the Ant Colony Optimization*, In *43rd Design Automation Conference* (DAC), 2006.

[C4] **Gang Wang**, Wenrui Gong, and Ryan Kastner, *Defect-Tolerant Nanocomputing Using Bloom Filters for Defect Mapping*, In *IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM), pp 277-278, 2006.

[C5] Wenrui Gong, **Gang Wang** and Ryan Kastner. *Storage Assignment During High-level Synthesis for Configurable Achitectures*, In *International Conference on Computer Aided Design* (ICCAD), 2005.

[C6] **Gang Wang**, Wenrui Gong and Ryan Kastner, *Instruction Scheduling using MAX-MIN Ant Optimization*, In *ACM Great Lakes Symposium on VLSI* (GLSVLSI), 2005.

[C7] Satish Sivaswamy, **Gang Wang**, Cristinel Ababei, Kia Bazargan, Ryan Kastner, Eli Bozorgzadeh. *HARP: hard-wired routing pattern FPGAs*, In *Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays* (FPGA), 2005.

[C8] Wenrui Gong, Yan Meng, **Gang Wang**, Ryan Kastner, and Timothy Sherwood, *Data Partitioning for Reconfigurable Architectures with Distributed Block RAM*, In *International Conference on Engineering of Reconfigurable*

Systems and Algorithms (ERSA), 2005.

[C9] Wenrui Gong, **Gang Wang**, and Ryan Kastner. *Data Partitioning for Reconfigurable Architectures with Distributed Block Ram*, In *The Fourteenth International Workshop on Logic and Synthesis* (IWLS), 2005.

[C10] **Gang Wang**, Wenrui Gong and Ryan Kastner. *System Level Partitioning for Programmable Platforms Using the Ant Colony Optimization*, In *13th International Workshop on Logic and Synthesis* (IWLS), 2004.

[C11] Wenrui Gong, **Gang Wang** and Ryan Kastner. *A High Performance Application Representation for Reconfigurable Systems*, In *the International Conference on Engineering of Reconfigurable Systems and Algorithms* (ERSA), 2004.

[C12] **Gang Wang**, Wenrui Gong and Ryan Kastner, *A New Approach for Task Level Computational Resource Bi-Partitioning*, In *15th IASTED International Conference on Parallel and Distributed Computing and System* (PDCS), 2003. (***Best paper nomination.***)

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Due to the rapid advances in VLSI fabrication technology, modern computer systems continue to provide better performance by effectively utilizing an increasing number of transistors on a chip. The well-known Moore's law [73], which predicts that the number of transistors on a single chip would grow exponentially over a relative short period of time, has been very accurate so far. Over the past 30 years, the transistor density has doubled every 18-24 months. ITRS estimates that we will be able to integrate more than half billion transistors on a 468 mm^2 chip by the year of 2009[85]. This creates tremendous potential for future computing systems.

This also imposes big challenges to the Electronic Design Automation (EDA) community on how to effectively build such complicated systems. As the complexity of

digital systems increases, so does the complexity of the underlying EDA problems. One critical question we need to address is: as computing resources become abundant, how can we effectively utilize these resources so that we can fully exploit the technology advances to solve future computing problem? For example, recent studies by Professor Cong's group at UCLA [20, 21], indicate that the results obtained over a benchmark circuits with known optimal wire lengths using the current commercially available routing and placement tools are far from the optimal. They argued that just by improving these results we are potentially able to move today's technology one generation ahead. While these examples are contrived, and follow-up study [64] shows that real-world designs are not as dire as initially suggested, this is still a staggering difference. As we move into smaller sub-micro technologies, there exists room for improvement for these algorithms.

In answering this challenge, one trend that seems to be affirmed in recent years is the shift to parallel architectural design or spatial computational model in constructing computing systems. This differs significantly from timeplexing a single active computation among a large number of operations as we have been familiar with in traditional single processor architectures. Spatial computational designs dedicate specific computing hardware to individual operations [24]. Operations are then interconnected in space rather than in time. This model will allow us to exploit the full parallelism available in the applications. The spatial computing model trades increased area for better time performance and effective overall usage of the computing resources.

The trend of shifting to spatial computing has also been evidenced by the steady market adoption of reconfigurable computing technologies, especially the Field Programmable Gate Array (FPGA) platform. Reconfigurable hardware, such as FPGAs, provides a programmable substrate onto which descriptions of circuits can be loaded and executed at very high speeds. Because they are able to provide an attractive balance between performance, cost, and flexibility, many critical embedded systems make use of FPGAs as their primary source of computation. Their circuit-level flexibility allows system functionality to be updated arbitrarily and remotely. For example, the aerospace industry relies on FPGAs to control everything from satellites to the Mars Rover.

FPGAs lie along a continuum between general-purpose processors and application-specific integrated circuits (ASICs). They provide both high-performance and well-defined timing behavior, but they do not require the costly fabrication of custom chips. While general purpose processors can execute any program, this generality comes at the cost of serialized execution. On the other hand, ASICs can achieve impressive parallelism, but their function is literally hard wired into the device. The power of reconfigurable systems lies in their ability to flexibly customize an implementation down at the level of individual bits and logic gates without requiring a custom piece of silicon. This can often result in performance improvements on the order of magnitude as compared to, per unit silicon, a similar microprocessor [24, 14]. FPGA technology is now the leading design driver for almost every single foundry. In fact it is estimated that in 2005 alone there were over 80,000 different commercial FPGA design starts [67].

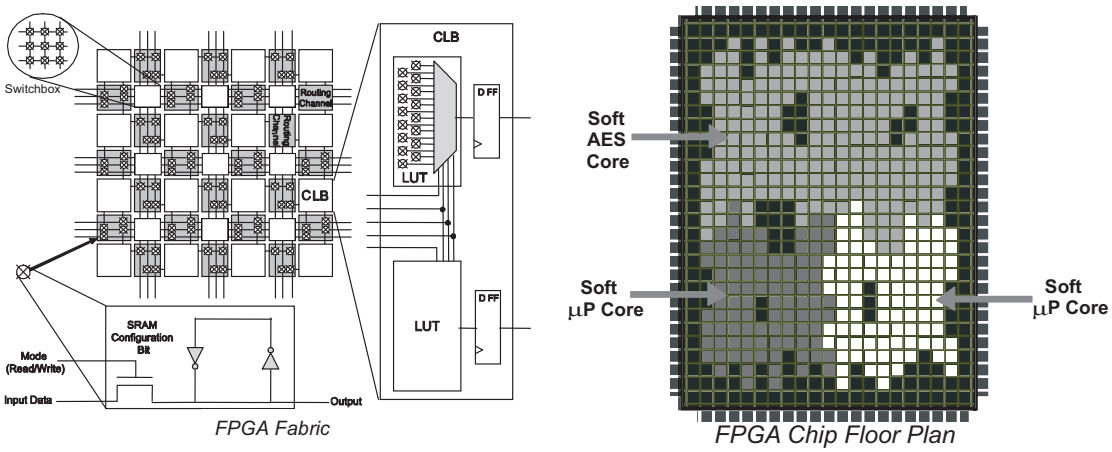


Figure 1.1: A simplified representation of an FPGA fabric is on the left. Configurable Logic Blocks (CLBs) perform logic level computation using Lookup Tables (LUTs) for bit manipulations and flipflops for storage. The switch boxes and routing channels provide connections between the CLBs. SRAM configuration bits are used throughout the FPGA (e.g., to program the logical function of the LUTs and connect a segment in one routing channel to a segment in an adjacent routing channel).The FPGA floor plan on the right illustratively shows a physical layout of FPGA after routing.

Figure 1.1 provides a simplified diagram of the modern FPGA architecture. FPGAs have very regular gate level patterns which differs from ASIC realization. Here the CLBs (configurable with LUT) are the basic logic/computing units. By connecting the CLBs using the interconnection resources (switchboxes and routing channels), an FPGA can be programmed to perform virtually any computation. It is worthwhile to note the tremendous flexibility the FPGA architecture provides to the designers. They have the capability to implement a wide variety of custom circuits, ranging from simple

adder/multiplier or multiple instances of them, to an entire computational function (e.g. digital filter, video codec), or even a complete CPU core (e.g. PowerPC 4507). This is very appealing; however, it also brings difficult questions regarding how to effectively use such flexibility. For example, for a given application and a known FPGA platform, what is the optimal configuration that achieves the best timing performance? Additionally, what is the impact on the system configuration (such as the number of adders and multipliers) if the requirements are relaxed? Moreover, is there a systematic method to help the designers explore the huge design decision space created by such increased flexibility?

Similar questions exist beyond the fine granularity level and the scope of pure FPGA-based platforms. In order to find cost effective ways to get the desired performance and maximize uses of resources, we often find designs that compose a hybrid technologies in today's systems. The most well-known example is the hardware/software co-design problem, where a system is composed of a general purpose CPU and a hardware-based computing resource (either ASIC or FPGA). More generally, a hybrid system can be organized to have *n* computing units with different capabilities and characteristics. How to assist the designer to effectively partition and distribute the computing tasks of an application over various computing units remains a system level challenge.

In general, as the complexity of digital systems increases, so does the complexity of these underlying EDA problems. To make it more difficult, many of these problems

are $\mathcal{NP}$-hard, which implies that finding optimal polynomial time algorithms for these problems are very unlikely. Due to this fact, almost all existing EDA systems apply heuristics to some extent. These heuristics were likely very successful and effective when they were invented. However, as the complexity of the problems increases, the conventional heuristic methods may fail in handling today's larger problems effectively. To face these challenges, we must look towards new optimization methods, rather than simply perform iterative improvements on existing techniques.

1.2 Research Overview

My dissertation research work is focused on constructing effective heuristic algorithms for solving difficult and fundamental design optimization problems. More specifically, the research focuses on devising new design automation algorithms based on the Ant Colony metaheuristics or Ant Colony Optimization (ACO) techniques. The ACO method is a relatively new meta-heuristic approach inspired by the ecological study of social insects (ants) and can be classified as a population based, self-organized meta-heuristic method; it was originally formulated to solve traditional $\mathcal{NP}$-hard combinatory problems in late 1990's and has been since successfully applied to solve a number of traditional $\mathcal{NP}$-hard combinatory problems.

ACO distinguishes itself from other conventional meta-heuristic methods (e.g. simulated annealing and genetic algorithms) with the following advantages:

- It formulates an optimization problem as a collaborative search process;

- It provides an effective way to combine global search experience with problem specific heuristics using pheromone sharing;
- It utilizes indirect communication in learning and employs positive feedback to achieve fast convergence;
- It offers a new and powerful way for solving optimization problems modeled as a graph, which is often the underlying model for various architectural problems.

Similar to other versatile meta-heuristic methods, such as Stimulated Annealing(SA), Genetic Algorithm (GA) and A* algorithm, it is possible to apply the Ant Colony metaheuristics to a slew of problems. However, careful attention has to be paid to consider the specific characteristics of the problem at hand and effectively integrate them in the final algorithms. In our study, we have selected to focus on three fundamental EDA problems, namely the system level partitioning problem, the operation scheduling problem and the design space exploration problem. These problems cover a good range of design granularity and are traditionally considered to be very difficult. We believe that these problems provide a good set of test cases for verifying the effectiveness of our methods. By addressing these problems with concretely constructed algorithms using Ant Colony metaheuristics, we hope to enrich and make contributions to the future system design methodologies.

As our first effort of applying the Ant Colony metaheuristics, we formulate new algorithms to address the system level task partitioning problem [102, 103, 105]. This problem is a fundamental $\mathcal{NP}$-hard challenge in a number of fields including high-level

system synthesis, parallel and distributed computing, and hardware/software co-design. It attempts to map application tasks onto multiple system resources w.r.t. the latency, hardware cost, power and other performance metrics. We construct a novel ACO-based algorithm to address this problem by introducing the Augmented Task Graph model. The concept can be easily extended to handle a variety of system requirements, including truly addressing the multi-way partitioning problem. The proposed algorithm consistently provides near optimal partitioning results on modestly-sized tested samples with very minor computational cost. For larger size problems, our algorithm scales well and achieves better solutions than the popularly used simulated annealing approach with substantially less execution time. Furthermore, we propose a hybrid approach that combines the ACO and simulated annealing together. This hybrid method leverages the complementary behaviors of the two algorithms and yields even better results than using them individually.

Operation scheduling is another fundamental architectural synthesis problem. An inappropriate scheduling of the operations will fail to exploit the full potential of the system. High quality scheduling solutions have direct impact in a number of different fields, such as compiler design for superscalar and VLIW microprocessors, distributed clustering computation architectures and hardware synthesis of ASICs and FPGAs. In our work, we introduce two novel algorithms [104, 99] using the ACO approach for the timing and resource constrained scheduling problems. We compile a comprehensive testing benchmark set (ExpressDFG) to verify the effectiveness and efficiency of the

proposed algorithms. For timing constrained scheduling, our algorithm achieves better results compared to force-directed scheduling on almost all the testing cases attaining a 19.5% reduction of resources. For resource constrained scheduling, our algorithm outperforms a number of list scheduling heuristics with better stability, and generates up to a 14.7% performance improvement. Our algorithms outperform the simulated annealing method for both scheduling problems in terms of quality, compute time and stability.

Finally, we look into the Design Space Exploration (DSE) problem, which tries to generate Pareto optimal tradeoffs among different system configurations. DSE is another critical challenge of high level synthesis. In practice, it is often addressed through ad-hoc probing of the solution space. This is not only time consuming but also very dependent on the designers experience. We propose a novel design exploration method that exploits the duality of the time and resource constrained scheduling problems [100, 101]. Our exploration automatically constructs a high quality time/area tradeoff curve in a fast, effective manner. In order to fully benefit from the duality attribute, we leverage the ACO-based optimization methods to solve both scheduling problems. We switch between these two algorithms to quickly traverse the design space. Compared with using force directed scheduling exhaustively at every time step, our approach provides a significant improvement on solution quality (average 17.3% reduction of resource counts) with similar run time on a comprehensive benchmark suite. It also scales well over different applications and problem sizes.

To summarize our research work, we focus on the essential algorithmic issues of applying Ant Colony metaheuristics to solve fundamental architectural design problems. We have successfully devised a series of algorithms for a number of problems across different levels of design granularity and achieved very promising results. We believe that Ant Colony metaheuristic is a framework of great potential in solving architectural design problems, and is not limited to the ones we studied. Moreover, we have developed a software tool, named CODES, to provide a uniform implementation for applying the ACO method to these architectural problems. To our best knowledge, our work is the first to introduce the Ant Colony metaheuristics to the architectural design field.

1.3 Organization of Dissertation

The dissertation is organized as follows: In Chapter 2, we give a review on the Ant Colony metaheuristic method with discussion on its characteristics. We present our work on applying the Ant Colony metaheuristic to solve system partitioning problem by introducing the Augmented Task Graph as the basic model in handling *n*-way partitioning problem in Chapter 3. In Chapter 4, we discuss how to solve timing and resource constrained operation scheduling problems. Also in this chapter, we will introduce the ExpressDFG benchmark suit we constructed. The same benchmark set will also be used in evaluating the design space exploration algorithm. We look into the parameter sensitivity issues experimentally. We introduce the duality based design

space exploration approach in Chapter 5. To lay the theoretical foundation, we will first prove an important theorem regarding duality between timing and resource constrained scheduling. Moreover, we will explain why ACO-based scheduling algorithms are favored in the proposed exploration framework. We conclude with Chapter 6.

Chapter 2

Ant Colony Metaheuristic

2.1 Nature Inspired Metaheuristics

As we have indicated in the previous chapter, fundamental architectural decisions often rely on solving $\mathcal{NP}$-hard combinatory optimization problems. With increasing complexity of these problems in today's applications, it becomes impossible to obtain the exact optimal solutions within a reasonable computation, and we have to use heuristic methods to hopefully obtain close-to-optimal results. One important approach for doing so is to select and utilize a metaheuristic method.

A metaheuristic is a heuristic method for solving a very general class of computational problems. It attempts to provide an efficient framework which combines user given black-box procedures. Such procedures are usually application specific heuristics themselves. The name combines the Greek prefix "meta" ("beyond", here in the

sense of “higher level”) and “heuristic” (from ευρισκειν, heuriskein, “to find”). Metaheuristics are generally applied to problems for which there is no satisfactory problem-specific algorithm or heuristic; or when it is not practical to implement such a method. Most commonly used metaheuristics are targeted to combinatorial optimization problems.

The simplest and most well known metaheuristic perhaps is the *Hill Climbing* method [82]. It is an optimization technique that belongs to the family of *local search* algorithms. The relative ease in implementation makes it a very popular first choice. However, this a simple method often fails to provide high quality results since it can easily get trapped within local minima.

In the past decades, a series of metaheuristic methods have been devised and successfully applied to a wide range of applications. It is interesting to notice that the best performing metaheuristics are almost inspired by nature.

Simulated annealing is a generic probabilistic meta-algorithm for finding global optima in large search space [56]. It was inspired by the annealing process in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. The heat causes the atoms to move from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; the slow cooling gives them a chance of finding configurations with lower internal energy than the initial one.

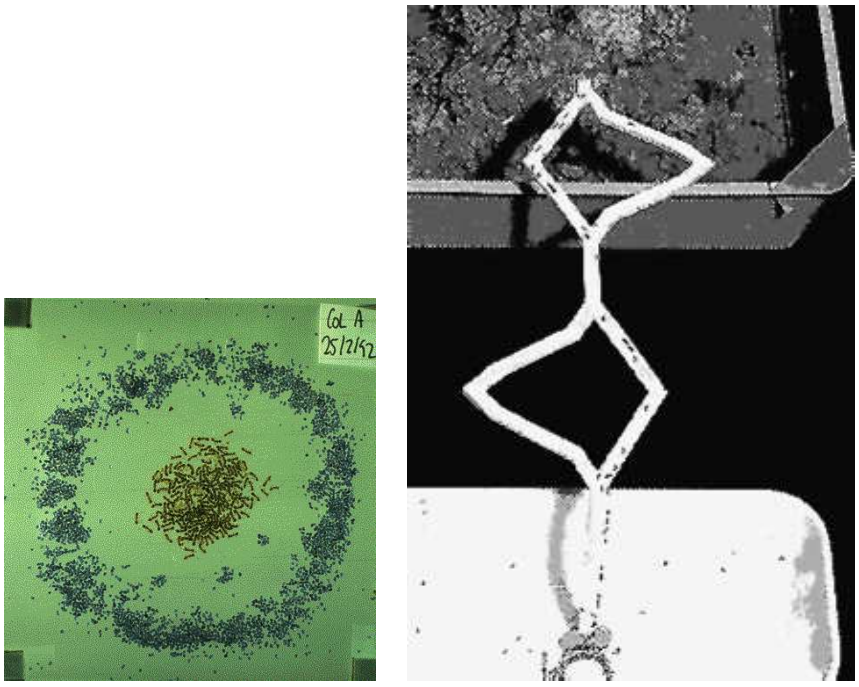
Genetic algorithms use techniques inspired by evolutionary biology such as inheri-

tance, mutation, selection, and crossover (also called recombination) [70]. They essentially solve the problems under consideration by simulating the evolutionary process, in which a population of abstract representations (called *chromosomes* or the genotype or the genome) of candidate solutions (called individuals, creatures, or phenotypes) evolves toward better solutions.

Artificial neural networks borrow the concept from how the human brain processes information by using an interconnected group of artificial neurons [1]. In solving an optimization problem, artificial neural networks use a mathematical model or computational model for information processing based on a connectionist approach, in which each processing unit is to simulate a individual neuron.

The Ant Colony Metaheuristic is a relatively new addition to the family of nature inspired algorithms for solving $\mathcal{NP}$-hard combinatory problems. Also known as Ant Colony Optimization (ACO) or Ant System (AS) algorithm[1] and originally introduced by Dorigo *et al.* [28] in 1996, it is a cooperative heuristic searching algorithm inspired by the ethological study on the behavior of ants. Figure 2.1(a) shows a laboratory nest constructed by a Leptothorax ant colony. The shown laboratory nest is made of two microscope slides separated by four 1mm thick cardboard pillars, one pillar at each corner. It closely approximates the rock crevices these ant colonies choose as nest sites in nature and facilitates easy observation on the ant behaviors. The blue dots are colored sand blocks from a pile provided outside the nest site that the ants have used

[1] In the rest of the discussion, we may use these terms interchangeably if not otherwise indicated



(a) (b)

Figure 2.1: (a) A laboratory nest of a Leptothorax ant colony; (b) Experiment settings used in [25].

for building a perimeter wall of the nest.

It was observed [25] that ants – who lack sophisticated vision – could manage to establish the optimal path between their colony and the food source within a very short period of time. This is done by an indirect communication known as *stigmergy* via the chemical substance, or *pheromone*, left by the ants on the paths. Though any single ant moves essentially at random, it will make a decision on its direction biased on the "strength" of the pheromone trails that lie before it, where a higher amount of pheromone hints a better path. As an ant traverses a path, it reinforces that path with its own pheromone. A collective autocatalytic behavior emerges as more ants will choose the shortest trails, which in turn creates an even larger amount of pheromone on those short trails, which makes those short trails more likely to be chosen by future ants. The experiment setup for the study done in [25] is shown in Figure 2.1(b), in which the ants converge to the shortest path between their nest and food source amongst four possible alternatives. The ACO algorithm is inspired by such observation. It is a population based approach where a collection of agents cooperate together to explore the search space. They communicate via a mechanism imitating the pheromone trails. The algorithm can be characterized by the following steps:

1. The optimization problem is formulated as a search problem on a graph;

2. A certain number of ants are released onto the graph. Each individual ant traverses the search space to create its solution based on the distributed pheromone trails and local heuristics;

3. The pheromone trails are updated based on the solutions found by the ants;

4. If predefined stopping conditions are not met, then repeat the first two steps; Otherwise, report the best solution found.

2.2 ACO for Travel Salesman Problem

One of the first problems to which ACO was successfully applied was the Traveling Salesman Problem (TSP) [28], for which it gave competitive results compared to traditional methods.

The objective of TSP is to find a Hamiltonian path for the given graph that gives the minimal length. More specifically, a TSP can be represented by a complete weighted directed graph $G = (V, E, d)$ where $V = \{1, 2, \ldots, n\}$ is a set of vertexes or cities, $E = \{(i, j) | (i, j) \in V \times V\}$ is a set of edges, and d is a weight function which associates a numeric weight d_{ij} for each edge (i, j) in E. This weight is naturally interpreted as the distance between city i and j in TSP. The objective is to find a Hamiltonian path for G which gives the minimal length.

In order to solve the TSP problem, ACO associates a pheromone trail for each edge in the graph. The pheromone indicates the attractiveness of the edge and serves as a global distributed heuristic. For each iteration, a certain number of ants are released randomly onto the nodes of the graph. An individual ant will choose the next node of the tour according to a probability that favors a decision of the edges that possesses

higher volume of pheromone. Upon finishing of each iteration, the pheromone on the edges is updated. Two important operations are taken in this pheromone updating process. First, the pheromone will evaporate, and secondly the pheromone on a certain edge is reinforced according to the quality of the tours in which that edge is included. The evaporation operation is necessary for ACO to effectively avoid local minima and diversify future exploration onto different parts of the search space, while the reinforcement operation ensures that frequently used edges and edges contained in better tours receive a higher volume of pheromone, which will have better chance to be selected in the future iterations of the algorithm. The above process is repeated multiple times until a certain stopping condition is reached. The best result found by the algorithm is reported as the final solution.

The algorithm associates a pheromone trail τ_{ij} for each edge (i, j) in E. It indicates the attractiveness of the edge and serves as a global distributed heuristic. Initially, τ_{ij} is set with some fixed value τ_0. For each iteration, m ants are released randomly on the cities, and each starts to construct a tour. Every ant will have memory about the cities it has visited so far in order to guarantee the constructed tour is a Hamiltonian path. If at step t the ant is at city i, the ant chooses the next city j probabilistically according to a probability:

$$p_{ij} = \begin{cases} \frac{\tau_{ij}(t)^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_k (\tau_{ik}^{\alpha}(t) \cdot \eta_{ik}^{\beta})} & \text{if is } j \text{ not visited} \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

where edges (i,k) are all the allowed moves from i, η_{ik} is a local heuristic which is defined as the inverse of d_{ij}, α and β are parameters to control the relative influence of

the distributed global heuristic τ_{ik} and local heuristic η_{ik}. Intuitively, the ant favors a decision on a edge that possesses higher volume of pheromone trail and better distance cost. Upon finishing of each iteration, the pheromone trail is updated according to the tours in which it is included. In the mean time, a certain amount of the it will evaporate. More specifically, we have:

$$\tau_{ij}(t) = \rho \cdot \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t) \qquad \text{where } 0 < \rho < 1. \tag{2.2}$$

Here ρ is the evaporation ratio, and $\Delta\tau_{ij}^{k} = Q/L_k$ if edge (i, j) is included in the tour ant k constructed, otherwise $\Delta\tau_{ij}^{k} = 0$. Q is a fixed constant to control the delivery rate of the pheromone, while L_k is the tour length for ant k. Two important operations are taken in this pheromone trail updating process. The evaporation operation is necessary for AS to be effective and diversified to explore different parts of the search space, while the reinforcement operation ensures that frequently used edges and edges contained in better tours receive a higher volume of pheromone and will have better chance to be selected in the future iterations of the algorithm. The above process is repeated multiple times until certain ending condition is reached. The best result found by the algorithm is reported. Figure 2.2 gives a visual illustration on how the above process works.

2.3 ACO for Other Combinatory Problems

When compared with existing algorithms over a set of difficult testing cases of the TSP, the ACO method achieved very competitive results [28] either on result quality

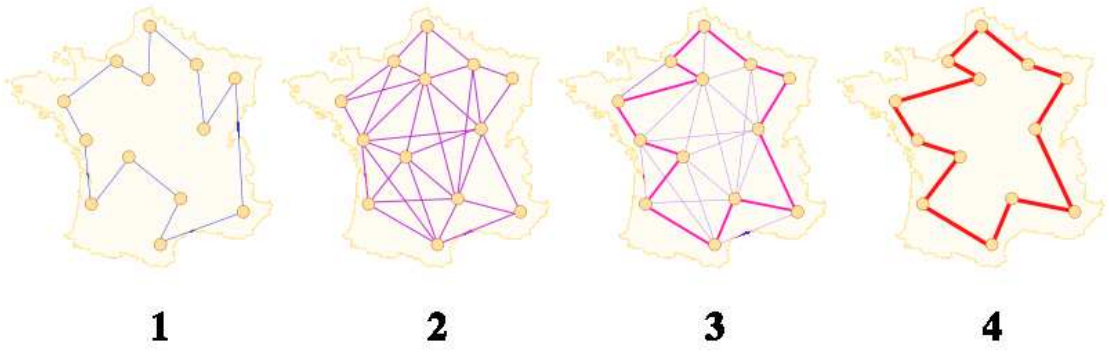


Figure 2.2: An illustration on how ACO-TSP works.

(1) Single ant constructs a solution; (2) Multiple solutions are constructed by all the ants individually;

(3) The pheromone trails adaptively adjust their values during the iterations; (4) The optimal solution emerges as the search learns from its experience.

or the computing time. Motivated by this success, researchers have since formulated ACO methods for a variety of traditional $\mathcal{NP}$-hard problems. These problems include the maximum clique problem [34], the quadratic assignment problem [37], the graph coloring problem [22], the shortest common super-sequence problem [61, 71], and the multiple knapsack problem [35]. ACO also has been applied to practical problems such as the vehicle routing problem [36], data mining [77] and network routing problem [83]. More recently, ACO approach was also successfully for bioinformatics application [87]. Table 2.1 gives a brief summary on the problems that ACO algorithms have been devised and related results.

Problem	Performance
Traveling salesman	state-of-the-art / good performance
Quadratic assignment	state-of-the-art / good performance
Job-Shop Scheduling	state-of-the-art / good performance
Vehicle routing	state-of-the-art / good performance
Sequential ordering	state-of-the-art performance
Shortest common supersequence	good results
Graph coloring and frequency assignment	good results
Bin packing	state-of-the-art performance
Constraint satisfaction	good performance
Multi-knapsack	poor performance
Timetabling	good performance
Optical network routing	promising performance
Set covering and partitioning	good performance
Parallel implementations and models	good parallelization efficiency
Routing in telecommunications networks	state-of-the-art performance
Protein Folding	state-of-the-art performance

Table 2.1: Applications of ACO method and their qualitative performance

2.4 Convergency of ACO Method

The convergence property of the ACO approach was investigated in [44, 46]. It was shown that ACO with a time-dependent evaporation factor or a time-dependent lower pheromone bound converges to an optimal solution with probability exactly one. The result enhanced the work presented in [45, 43, 91] for ACO algorithms to the strength of the well-known convergence property of the Simulated Annealing meta-heuristic. As in Simulated Annealing, it turns out that a convergence guarantee can be obtained

by a suitable speed of "cooling" (i.e., reduction of the influence of randomness). First, in the basic ACO formulation, the geometric pheromone decrement caused by constant evaporation factor on not reinforced arcs is too fast and leads (in general) to premature convergence to suboptimal solutions. On the other hand, introducing a fixed lower pheromone bound stops cooling at some point and leads to random-search-like behavior without convergence. In between lies a compromise of allowing pheromone trails to tend to zero, but slower than geometrically. This can be achieved either by decreasing evaporation factors, or else by "slowly" decreasing lower pheromone bounds. In a certain window of the cooling speed, we get convergence to the optimal solution with probability of one.

However, it is worth noting that that the theoretical cooling speeds indicated in [44] are also the most efficient ones, where efficiency is measured by the average runtime required to find a solution of a sufficiently good quality (say, only ρ% worse than the best solution with some pre-defined ρ). In the typical area of application for ACO, i.e. the area of $\mathcal{NP}$-complete combinatorial optimization problems, we cannot expect to obtain an algorithm providing optimal solutions in a short computation time. Again, as in Simulated Annealing, it might turn out that faster cooling than indicated by the theoretical scheme is advantageous for finite-time computing – that is for getting quick convergence, it may be worthwhile to pay the price of convergence to suboptimal solutions. However, experimental studies with slightly decreasing evaporation factors or lower pheromone bounds falling slightly slower than geometrically might be interest-

ing, especially for applications where the user is willing to invest a high amount of computation time for obtaining excellent solution quality. In addition to the (so-called "elitist") pheromone update mechanisms investigated in the paper, the author also suggests computational experiments with decreasing evaporation factors and/or decreasing lower pheromone bounds for other empirically successful update mechanisms, such as the rank-based update rule introduced by Bullheimer, Hartl and Strauss [15]. It would not be a surprise if some moderate form of retarding the cooling process could, in a considerable number of cases, be able to further improve the performance of present ACO implementations.

2.5 MAX-MIN Ant System (MMAS)

Premature convergence to local minima is a critical algorithmic issue that can be experienced by many heuristic optimization algorithms. As we have discussed in the previous section, though it was shown [44] that ACO with a time-dependent evaporation factor or a time-dependent lower pheromone bound converges to an optimal solution with probability of exactly one, it failed in providing any constructive approach. Balancing exploration and exploitation is not trivial in these algorithms, especially for algorithms that use positive feedback such as ACO.

The MAX-MIN Ant System (MMAS) [92] is one framework to provide such balance in an adaptive manner. It is built upon the original ACO algorithm and is specifically designed to address the premature convergence problem. It improves the original

ACO by providing dynamically evolving bounds on the pheromone trails such that the heuristic value is always within a limit to that of the best path. As a result, all possible paths will have a non-trivial probability of being selected and thus it encourages broader exploration of the search space.

More specifically, MMAS forces the pheromone trails to be limited within evolving bounds, that is for iteration t, $\tau_{min}(t) \leqslant \tau_{ij}(t) \leqslant \tau_{max}(t)$. If we use f to denote the cost function of a specific solution S, the upper bound τ_{max} [92] is shown in (2.3). Here $S^{gb}(\cdot)$ represents the global best solution found so far in all iterations.

$$\tau_{max}(t) = \frac{1}{1-\rho}\frac{1}{f(S^{gb}(t-1))} \tag{2.3}$$

The lower bound is defined as (2.4):

$$\tau_{min}(t) = \frac{\tau_{max}(t)(1-\sqrt[n]{p_{best}})}{(avg-1)\sqrt[n]{p_{best}}} \tag{2.4}$$

where $p_{best} \in (0,1]$ is a controlling parameter to dynamically adjust the bounds of the pheromone trails. The physical meaning of p_{best} is that it indicates the conditional probability of the current global best solution $S^{gb}(t)$ being selected given that all edges not belonging to the global best solution have a pheromone level of $\tau_{min}(t)$ and all edges in the global best solution have $\tau_{max}(t)$. Here avg is the average size of the decision choices over all the iterations. For a TSP problem of n cities, $avg = n/2$. It is noted from (2.4) that lowering p_{best} will result in a tighter range for the pheromone heuristic. As $p_{best} \to 0$, $\tau_{min}(t) \to \tau_{max}(t)$, which means more emphasis is given to search space exploration.

Theoretical treatment of using the pheromone bounds and other modifications on the original ACO algorithm are proposed in [92]. These include a pheromone updating policy that only utilizes the best performing ant, initializing pheromone with τ_{max} and combining local search with the algorithm. It was reported that MMAS was the best performing ACO approach and provided very high quality solutions.

Chapter 3

System Partitioning

Modern digital systems consist of a complex mix of computational resources, e.g. microprocessors, memory elements and reconfigurable logic. System partitioning – the division of application tasks onto the system resources – plays an important role for the optimization of the latency, area, power and other performance metrics. With the advent of complex heterogenous system architectures that contain a variety of computing components like microprocessors, memory elements and reconfigurable logic, system partitioning becomes an important step in the system design process, i.e. how to optimally assign computational tasks to the different system computing resources while respecting pre-defined design constraints.

In this chapter, we present a novel approach for this problem based on the Ant Colony Optimization, in which a collection of agents cooperate using distributed and local heuristic information to effectively explore the search space. The proposed model

can be flexibly extended to fit different design requirements. Experiments show that our algorithm provides robust results that are qualitatively close to the optimal with minor computational cost. Compared with the popularly used simulated annealing approach, the proposed algorithm gives better solutions with substantial reduction on execution time for large problem instances. Moreover, a hybrid approach that combines our algorithm and SA achieves even better results with great runtime reduction.

3.1 Introduction

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

A programmable platform is a device consisting of programmable cores. Its programmability allows application development after it is fabricated. Therefore, the functionality of the device can change over time. This is especially important for embedded systems where the hardware cannot be easily upgraded (e.g. computers in cars). As

standards change, one just need to reprogram the device, rather than physically replace the hardware. For these reasons, programmable platforms provide a good price point for low volume applications. It allows "low" end users to create designs using newest, highest performance manufacturing process. Furthermore, programmable devices enable fast prototyping, which allows for faster time to market.

Xilinx Virtex [108] and Altera Excalibur devices [6] are two examples of such programmable platform. These platforms may consist of hard cores, programmable cores and/or soft cores. A hard core is a dedicated static processing unit, e.g. ARM processor in Excalibur or the PowerPC core in Virtex. A programmable core is some kind of programmable logic device (PLD) (e.g. FPGA, CPLD). A soft core is a processing unit implemented on programmable logic, e.g. CAST DSP core [18] on Virtex or Nios [7] on Excalibur. The programmability in these devices ranges from extremely fine grain control in PLD, to coarse grain control in the microprocessor. This allows for fine grain optimizations (bit level optimizations in the PLD), instruction level optimizations (on processor cores) and task level optimizations (across programmable cores).

Comparing with the traditional single CPU architecture, these complex programmable platforms require more effective computer-aided design (CAD) techniques to allow design space exploration by application programmers. One special challenge resides at the system level design phase. At this stage, the application programmer works with a set of tasks, where each task is a coarse grained set of computations with a well defined interface based on the application. Different from single CPU

architecture, a key step in the mapping of applications onto these systems is to assign tasks to the different computational cores.

This partitioning problem is $\mathcal{NP}$-complete [38]. Although it is possible to use brute force search or ILP formulations [74] for small problem instances, generally, the optimal solution is computationally intractable. Thus it requires us to develop efficient algorithms in order to automatically partition the tasks onto the system resources, while optimizing performance metrics such as execution time, hardware cost and power consumption.

It is worth mentioning that though the above partitioning problem shares certain similarity with the Job Scheduling Problem (JSP) [40], another well-studied $\mathcal{NP}$-hard problem in the operation optimization community, they are fundamentally different. First, the *jobs* in JSP are independent from each other while the computational tasks are interrelated and constrained by data dependencies among different tasks. Secondly, for every job in JSP, each of its operations is explicitly associated with a resource known *a priori*, while a computational task on the programmable platform is possible to be allocated on different resources as long as the system requirements are met. Finally, the optimization target in JSP is only constrained by the condition that no two jobs are processed at the same time on the same resource. However, in the above task partitioning problem, besides this constraint, we also need respect other system design requirements, such as limits on power consumption and hardware cost.

Some early works [32, 42, 90, 95] investigate the hardware/software partitioning

problem, which is a special case of the system partitioning problem discussed here[1]. It is difficult to name a clear winner [30]. Partitioning issues for system architectures with reconfigurable logic components have also been studied [9, 47, 62]. These works assume a reconfigurable device coupling with a processor core in their partitioning problem.

Different heuristic methods have been proposed to try to effectively provide sub-optimal solutions for the problem. These methods include Simulated Annealing (SA), Tabu Search (TS), and Kernighan/Lin approach [32, 52, 31, 3, 96]. Evolutionary methods [50, 75] using Genetic Algorithm (GA) are also studied. Software tools based on these heuristics have been developed for system level partitioning problem. For instance, in COSYMA [23], the application tasks are mapped onto the system architecture using Simulated Annealing. Wiangtong *et al.* [106] compared three popularly used heuristic methods, and provided a good survey on the motivation and the related work of using task level abstraction. These methods provide practical algorithms for achieving acceptable the system partitioning solutions, however, they also have different drawbacks. Simulated Annealing suffers from long execution time for the low temperature cooling process. For Genetic Algorithm, special effort must be spent in designing the evolutionary operations and the problem-oriented chromosome representation, which makes it hard to adapt to different system requirements.

In this chapter, we present a novel heuristic searching approach to the system par-

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

titioning problem based on the *Ant Colony Optimization* (ACO) algorithm [28]. In the proposed algorithm, a collection of agents cooperate together to search for a good partitioning solution. Both global and local heuristics are combined in a stochastic decision making process in order to effectively and efficiently explore the search space. Our approach is truly multi-way and can be easily extended to handle a variety of system requirements.

The remainder of the chapter is organized as follows. Section 3.2 details the proposed algorithm for the constrained multi-way partitioning problem. As the basis of our algorithm, a generic mathematic model for multi-way partitioning is also introduced in this section. In Section 3.3.1, we present the experimental heterogenous architecture and the testing benchmark we used in our work. We analyze the experiment results and give assessment on the performance of the proposed algorithm in Section 3.3. We summarize our work on this topic with Section 3.5.

3.2 ACO for System Partitioning

3.2.1 Problem Definition

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

For example, operation (instruction) level partitioning is done in the Garp project [16], while the good deal of research work [52, 31, 106, 23] are on the functional task level.

In this work, we focus on partitioning at the task or functional level. One of the reasons we select the task level partitioning is that it is commonly found that a bad partitioning in the task level is hard to correct in lower level abstraction [53]. Additionally, task level partitioning is typically requested in the earlier stage of the design so that further hardware synthesis can be performed.

We formally define the system partitioning problem as follows:

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

An application to be partitioned onto the system is given as a set of tasks $T_{app} = \{t_1, \ldots, t_N\}$, where the atomic partitioning unit, a *task*, is a coarse grained set of computation with a well defined interface. The precedence constraints between tasks are modeled using a task graph. A *task graph* is a directed acyclic graph (DAG) $G = (T, E)$, where $T = \{t_0, t_n\} \cap T_{app}$, and E is a set of directed edges. Each task node defines a functional unit for the program, which contains information about the computation it needs to perform. There are two special nodes t_0 and t_n which are virtual task nodes. They are included for the convenience of having an unique starting and ending point of the task graph. An edge $e_{ij} \in E$ defines an immediate precedence constraint between t_i and t_j. For a given partitioning, the execution of a task graph runs in the following

way: the tasks of different precedence levels are sequentially executed from the top level down, while tasks in the same precedence level but allocated on different system components can run concurrently. Notice the precedence constraint is transitive. That is, if we let $\longrightarrow$ denote the precedence constraint, we have:

$$(t_a \longrightarrow t_b) \wedge (t_b \longrightarrow t_c) \Rightarrow t_a \longrightarrow t_c \tag{3.1}$$

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

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

Thus, the multi-way system partitioning problem is formally defined as: Find a set of partitions $P = \{P_1, \ldots, P_r\}$ on r resources, where $P_i \subseteq T$, $P_i \cap P_j = \phi$ for any $i \neq j$ that minimizes a system objective function under a set of system constraints.

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

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

3.2.2 Augmented Task Graph

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

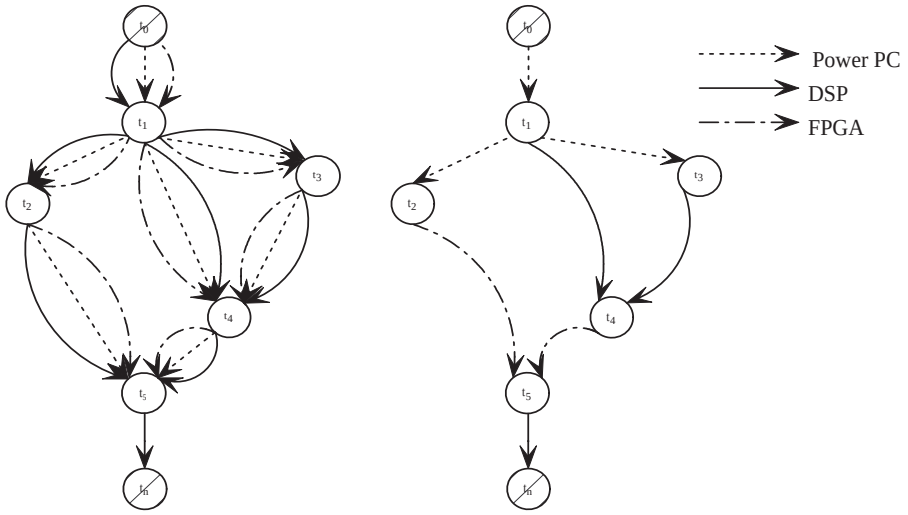
$$e'_{ijk} = (t_i, t_j, r_k), \text{ where } e_{ij} \in E, \text{ and } k = 1, ..., r \tag{3.2}$$

In ATG, an edge e'_{ijk} represents the *binding* of edge e_{ij} with resource r_k. Our algorithm uses these augmented edges to make a local decision at task node t_i about the binding of the resource on task t_j[3]. We call this an *augmented edge*. The original task graph G is called the *support* of G'.

An example of ATG is shown in Figure 3.1(a) for a 3-way partitioning problem. In

[3]This will be further explained in Section 3.2.3

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



(a) (b)

Figure 3.1: ATG for 3-way Partitioning

Based on the ATG model, a specific partitioning for the tasks on the multiple resources is a graph G_p, where G_p is a subgraph of G' that is isomorphic to its *support* G, and for every task node t_i in G_p, all the incoming edges of t_i are bounded with the same resource (say) r. Further, we say that partition G_p allocates task t_i to resource r. Figure 3.1(b) shows a sample partitioning for the ATG illustrated in Figure 3.1(a). In

this partitioning, task 1, 2, and 3 are allocated onto the PowerPC, task 4 is partitioned on to the DSP and task 5 for the FPGAs. As t_n is a virtual node, we do not care the status of the edge from t_5 to t_n.

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

$$f_{ik} = t_i \bullet r_k, \forall t_i \in T, \forall r_k \in R \tag{3.3}$$

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

$$f_{ik} = t_i \bullet r_k = \{time_{ik}, area_{ik}\} \tag{3.4}$$

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

3.2.3 ACO Formulation for System Partitioning

Based on the ATG model, our goal is to find a feasible partitioning G_p for G', which provides the optimal performance subject to the predefined system constraints. We introduce a new heuristic method for solving the multi-way system partitioning problem using the ACO algorithm. Essentially, the algorithm is a multi-agent [4] stochastic de-

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

cision making process that combines local and global heuristics during the searching process.

procedure ACOSystemPartition(G,R)

input: DFG $G(V,E)$, resource set R

output: system partition P_{best} to minimize latency under hardware cost constraint

1: construct ATG G' based on G and R, and $P_{best} \leftarrow \phi$

2: **while** ending conditioning is not met **do**

3: **for** $1 \leqslant l \leqslant m$ **do**

4: Initialize pheromone trail $\tau_{ijk} \leftarrow \tau_0$ for each e'_{ijk} in G'

5: $ant(l)$ crawls over G' to create a feasible partitioning P_l;

6: Evaluate P_l based on its execution time $time(P_l)$.

7: If P_l is better than P_{best}, update P_{best}.

8: **end for**

9: Update the pheromone trails on the edges as follows:

$$\tau_{ijk} \leftarrow (1-\rho)\tau_{ijk} + \sum_{l=1}^{m} \Delta\tau_{ijk}^{(l)} \tag{3.5}$$

$$\Delta\tau_{ijk}^{(l)} = \begin{cases} Q/time(P_l) & \text{if } e'_{ijk} \in P_l \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

where $0 < \rho < 1$ is the evaporation ratio, $k = 1,\ldots,r$, and Q is a fixed constant to control the delivery rate of the pheromone.

10: **end while**

11: return P_{best}

Algorithm 1: ACO Algorithm for System Partitioning

The proposed algorithm proceeds as illustrated by Algorithm 1. Step 5 is an important part in the proposed algorithm. It describe how an individual ant "crawls" over the ATG and generates a solution. Two problems must be addressed in this step:

1. How does the ant handle the precedence constraints between task nodes?

2. What are the global and local heuristics and how can they be applied?

3. Finally, how does the ant guarantee to find a feasible partition for the given application?

To answer these questions, each ant traverses the graph in a topologically sorted manner in order to satisfy the precedence constraints of task nodes. The trip of an ant starts from t_0 and ends at t_n, the two virtual nodes that do not require allocation. By visiting the nodes in the topologically sorted order, we ensure that every predecessor node is visited before we visit the current node and that every incoming edge to the current node has been evaluated. We can see later that by enforcing this ordering, we not only make sure that the found partition could be executed correctly but also provide an important preparation for the ant to make resource allocation decision upon entering a new task node.

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

$$p_{ijk} = \frac{\tau_{ijk}^{\alpha} \eta_{jk}^{\beta}}{\sum_{l=1}^{r} \tau_{ijl}^{\alpha} \eta_{jl}^{\beta}} \tag{3.7}$$

Here α and β are parameters to control the relative influence of the distributed global heuristic τ_{ijk} and local heuristic η_{jk} if t_j is assigned to resource r_k. In our work, we

simply use the inverse of the cost of having task t_j allocated to resource r_k as η_{jk}. We focus on achieving the optimal execution time subject to hardware area constraint, therefore a simple weighted combination is used to estimate the cost:

$$cost_{jk} = w_t \cdot time_{jk} + w_a \cdot area_{jk} \tag{3.8}$$

where $time_{jk}$ and $area_{jk}$ are the execution time and hardware area cost estimates, constants w_t and w_a are scaling factors to normalize the balance of the execution time and area cost. It is intuitive to notice that the probability p_{ijk} favors an assignment that yields smaller local execution time and area cost, and an assignment that corresponds with the stronger pheromone. Again $time_{jk}$ and $area_{jk}$ are obtained via the dot operation explained above in Section 3.2.2. Based on the proposed ATG model, by altering the dot operation, one can easily adapt the cost function to consider other constraints such as power consumption limit, while keep the algorithm essentially intact.

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

assigning t_j to r_k is:

$$p_{jk} = \frac{\text{count of guess } r_k \text{ for } t_j}{\text{count of immediate precedents of } t_j} \tag{3.9}$$

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

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

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

As illustrated in Step 5 by the algorithm, at the end of each iteration, the pheromone trails on the edges are updated according to Equation (3.5) and (3.6). First, a certain amount of pheromone is evaporated. From an optimization point of view, the evaporation step helps the system escape from local minimums. Secondly, the *good* edges

are reinforced. This reinforcement creates additional pheromone on the edges that are included on partition solutions that provide shorter execution time for the task graph. The given updating policy is similar to that reported in [28]. Notice here that every ant will contribute to the pheromone update independently based on the quality of the partition it finds. Alternative reinforcement methods [13] can also be applied here. For example, we explored the strategy of updating the pheromone trails on the edges that are included only in the best tour amongst all the returned partitions at each iteration, and we observed no noticeable difference regarding to the quality of the final results.

Finally, each run of the algorithm is composed of multiple iterations of the above steps. Two ending possible stopping conditions are: 1) the algorithm ends after a fixed number of iterations, or 2) the algorithm ends when there is no improvement found after a number of iterations. In the same run, the global pheromone trails τ_{ijk} are initialized once as indicated in step 1 at the start of the algorithm, updated at the end of each iteration, and inherited by the next iteration. The best partition found so far by the ants is also updated dynamically at the end of each iteration and reported as the final result of the run. Because of the stochastic nature of the algorithm, multiple runs can be conducted and may provide different results. Another reason to have multiple runs is to test the stability of the proposed algorithm in achieving high-quality results, as we will discuss in Section 3.3. For our experiments reported here, each run is independent and is started from scratch without using any result obtained in previous runs.

3.2.4 Complexity Analysis

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

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

$$\text{speedup} = \frac{(\mathrm{r^N}) * \mathrm{E_t}}{\mathrm{m} * \mathrm{I} * (\mathrm{Ant_t} + \mathrm{E_t})} \approx \frac{\mathrm{r^N}}{\mathrm{m} * \mathrm{I}} \tag{3.10}$$

The number of ants in each iteration m depends on the problem that is being solved by the ACO algorithm. For the TSP problem, the authors assigned m to be a constant multiple of total number of nodes in the TSP problem instance [28]. For the multiway partitioning problem based on the ATG, we propose two possible ways to determine the ant number: 1) based on the average branching factor of the original task graph G; or 2) the maximum branch number of the original task graph G.

3.2.5 Extending the ACO/ATG method

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

During system design phase, it is common that certain computational tasks are predetermined or preferred to run on certain resources. That is for each task $t_i \in T$, it is associated with a probability set $\{p_i^1, \ldots, p_i^r\}$ where r is the size of R. Among the elements of the set, some of them can be zero when the corresponding resources have been determined to be not suitable for the given task. By modifying the decision strategy in Equation (3.7), we can easily accommodate this requirement by using the following equation:

$$\mathsf{p}_{ijk} = \frac{p_i^k \tau_{ijk}^{\alpha} \eta_{jk}^{\beta}}{\sum_{l=1}^{r} p_i^l \tau_{ijl}^{\alpha} \eta_{jl}^{\beta}} \tag{3.11}$$

Similar to the above approach, other task dependent information, such as profiling statistics can also be considered. In this case, the probability distribution set is associated with the augmented edges in the ATG, instead with the resources. That is for each edge e'_{ijk} defined in Equation (3.2), there exists a frequency probability value p_{ijk}, which satisfies the following conditions:

$$\begin{cases} p_{ijk} = p_{i'j'k} & \text{if } i = i' \text{ and } j = j' \\ \sum p_{ijl} = 1 & \text{where } l = 1, \ldots, r \end{cases} \tag{3.12}$$

Using the two approaches discussed here, one can further modify the proposed algorithm to handle more complicated system features, such as different communication channels, where each channel has a different bandwidth and latency. These channels can either be associated with the augmented edges if they are bounded with the hardware realization, or may be treated as a task related attribute if the task can only use one certain type channel.

Finally, by altering the definition of the *dot* operation in Equation (3.3), better local cost estimation model can be introduced and integrated as the local heuristics. Similarly, different target objective functions for defining the global heuristic η in Equation (3.7) can be applied. For example, power consumption can be aggregated as part of the consideration during the process.

3.2.6 Comparing with the Original ACO

In this section, we will summarize the proposed algorithm by comparing it with the original ACO approach proposed in [28].

Perhaps the most fundamental contrast between our work and the original ACO reported in [28] is that they try to solve different domain problems. Though the ACO approach is known as a meta-heuristic method for addressing optimization problems, one still needs to form specific strategies in order to effectively utilize the domain specific characteristics for the problem in hand. To our best knowledge, the method we proposed here is the first approach in the literatures for solving application partitioning problem using the ACO heuristics. Comparing with the TSP problem that the original ACO algorithm was set to address, the application partitioning problem poses specific issues in formulating the ACO algorithm, even though both of them are $\mathcal{NP}$-complete.

First, there is a need to develop an appropriate graph model in formulating an ACO method for the application partitioning problem such that the global and local heuristics could be meaningfully fitted in. As discussed above, the ATG model is introduced in

our work as the answer, where the extended edges provide suitable attaching points for the global and local heuristics. In contrast, the modeling issue is relatively easier for the TSP problem since the connection graph of the problem is readily used as the model.

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

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

Finally, in our work, we propose a decision making process that is different from that in the original ACO method. In the TSP problem, the only decision to make is to which city the ant shall move to while constructing the Hamilton tour. However, in the

application partitioning problem, we have to visit all the child nodes in the ATG in a sorted order. Furthermore, when a task node is visited, we need to make decision on which computing resource it shall be mapped to. As discussed earlier, in our algorithm, a two step decision making process is adopted. First, at each node, the ant makes a "guess" for each immediate child node on how it should be mapped based on the global and local heuristics associated with these nodes. The final decision is delayed until the child node is visited by the ant and the partitioning for the node is done using yet another probabilistic approach over the previous "guesses", such as the one indicated by Equation (3.9).

3.3 Experimental Results and Performance Analysis

3.3.1 Target Architecture and Benchmarks

Our experiments address the partitioning of multimedia applications onto a programmable, multiprocessor system platform. The target architecture contains one general purpose hard processor core, a soft DSP core, and one programmable core (see Figure 3.2).

This model is similar to the Xilinx Virtex II Pro Platform FPGA [108], which contains up to four hard CPU cores, 13,404 configurable logic blocks (CLBs) and other peripherals. In our work, we target a system containing one PowerPC 405 RISC CPU core, separate data and instruction memory, and a fixed amount of reconfigurable logic

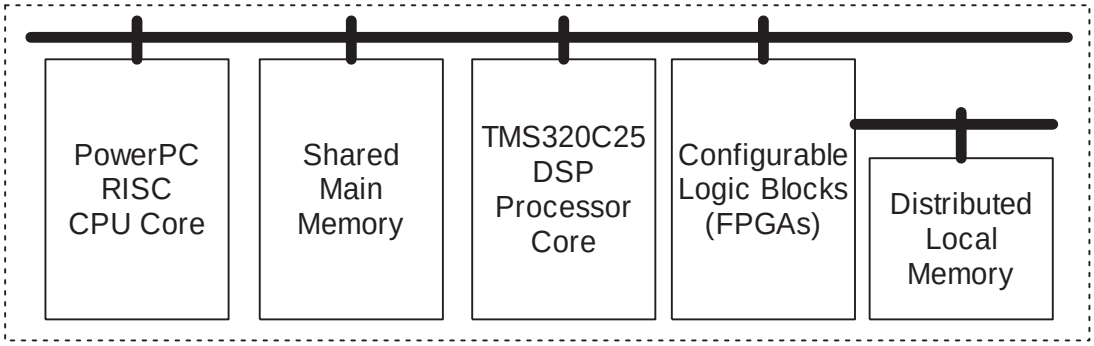


Figure 3.2: Target architecture

with a capacity of 1,232 CLBs, among which, 724 CLBs are available to be used as general purpose reconfigurable logic (FPGA), and the remaining 508 CLBs embed an FPGA implementation (soft core) of the TMS320C25 DSP processor core [18]. Programmable routing switches provide communication between the different system resources.

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

Tasks allocated on either the PowerPC processor or the DSP processor are executed sequentially subject to the precedence constraints within the task (i.e. instruction level

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

Our experiments are conducted in a hierarchical environment for system design. An application is represented as a task graph in the top level. The task graph, formally described in Section 3.2.1, is a directed acyclic graph, which describes the precedence relationship between the computing tasks. A task node in the task graph refers to a function, which could be written in high-level languages, such as C/C++. It is analyzed using the SUIF [4] and Machine SUIF [89] tools; the result is imported in our environment as a control/data-flow graph (CDFG). CDFG reflects the control flow in a function, and may contain loops, branches, and jumps. Each node in CDFGs is a basic block, or a set of instructions that contains only one control-transfer instruction and several arithmetic, logic, and memory instructions.

Estimation is carried out for each task node to get performance characteristics, such as execution time, software code length, and hardware area. Based on the specification data of the Virtex II Pro Platform FPGA [108] and the DSP processor core [18], we

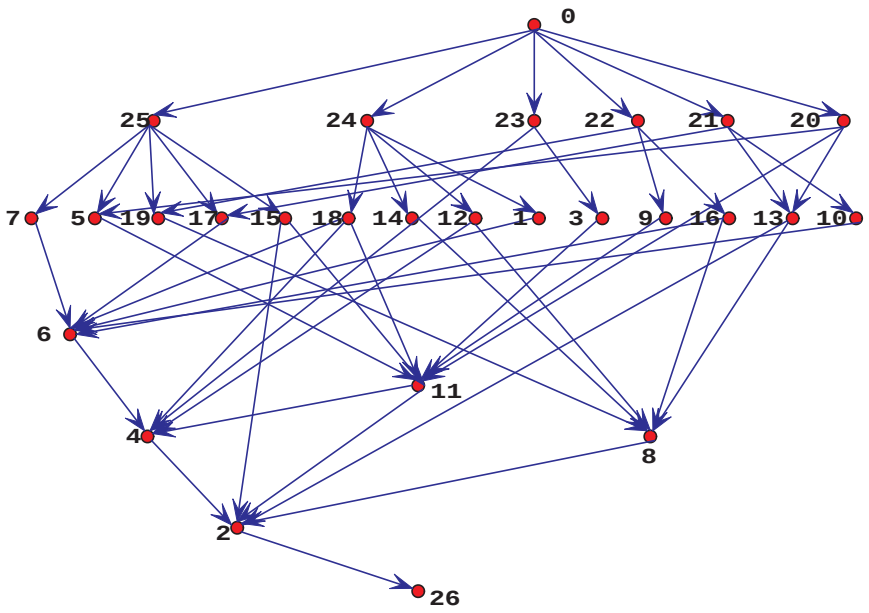


Figure 3.3: Example Task Graph

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

We create a task level benchmark suite based on the MediaBench applications [59].

Each testing example is formed via a two step process that combines a randomly generated DAG with real life software functions. The testing benchmarks are available online *http://express.ece.ucsb.edu*.In order to better assess the quality of the proposed algorithm while the application scales, task graphs of different sizes are generated. For a given task graph, the computation definitions associated with the task nodes are selected from the same application within the MediaBench test suite. Task graphs are created using GVF tool kit [68]. With this tool, we are able to control the complexity of the generated DAGs by specifying the total number of nodes or the average branching factor in the graph. Figure 3.3 gives a typical example for the task graph we used in our study.

3.3.2 Absolute Quality Assessment

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

are eliminated in our experiments to ensure that we are targeting the *hard* instances. We also provided an ILP formulation similiar to that reported in [51] for the given problem. However, the size of the problem prohibited it from being solvable. Unlike the brute force search the ILP formulation does not provide detailed information about the distribution of the solution quality over the complete search space, thus makes it hard to quantitatively judge the hardness of the testing samples.

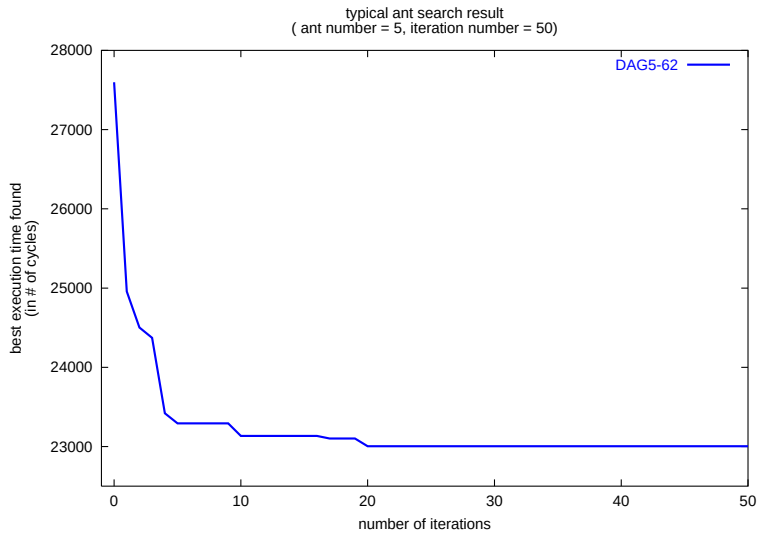


Figure 3.4: A typical run of ant search

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

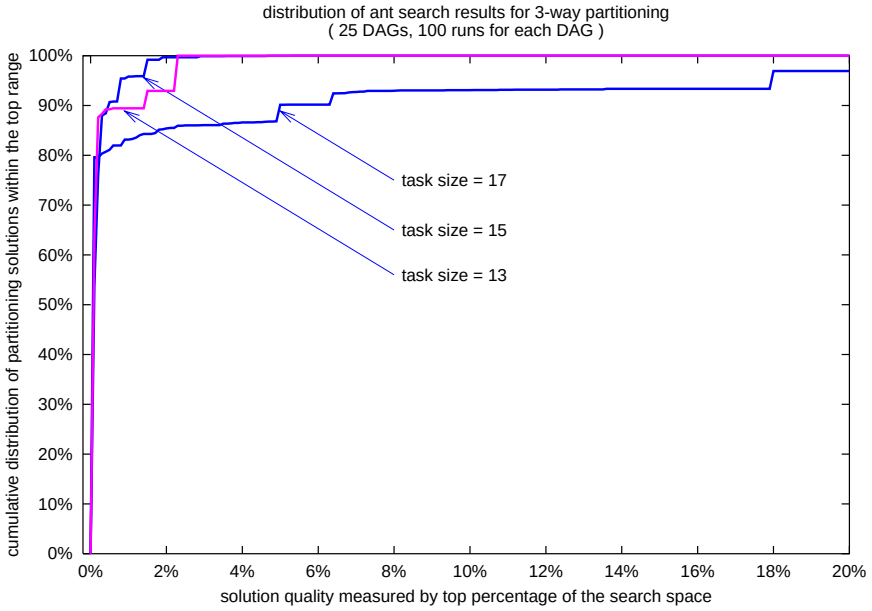


Figure 3.5: Result quality measured by top percentage

A typical run of our algorithm is shown in Figure 3.4. It shows the best execution time found by the ants after each iteration. In this case, the ants found a partition that provides the optimal execution time for DAG-5 very quickly,after only 20 iterations. This behavior is consistent over all the runs we conducted and agrees with the results reported in [28] for TSP problems.

Figure 3.5 shows the cumulative distribution of the number of solutions found by the ACO algorithm plotted against the quality of those solutions for different problem sizes. The x-axis gives the solution quality compared to the overall number of solutions. The y-axis gives the total number of solutions (in percentage) that are worse than the solution quality. For example, looking at the x-axis value of 2% for size 13, less than 10% of the solutions that the ACO algorithm found were outside of the top 2% of the

overall number of solutions. In other words, over 90% of the solutions found by the ACO algorithm are within 2% of all possible partitions. The number of solutions drops quickly showing that the ACO algorithm finds very good solutions in almost every run. In our experiments, $2,163$ (or 86%) solutions found by ACO algorithm are within the top 0.1% range. Totally $2,203$ solutions, or 88.12% of all the solutions, are within the top 1% range. The figure indicates that a majority of the results are qualitatively close to the optimal.

With the definitive description on the search space obtained from the brute force search, we can also evaluate the capability of the algorithm with regard to discovering the optimal partition. Table 3.1 shows a comparison between the proposed algorithm and random sampling when the task graph size is 13. The first column gives the testing case index. The second and third columns are the optimal execution time and the number of partitions that achieve this execution time for the testcase, respectively. This information is obtained through the brute force search. The fourth column gives the derived theoretical possibility of finding an optimal partition in 250 tries over a search space with a size of $3^{13} = 1,594,323$ if random sampling is applied. The last column is the number of times we found an optimal partition in the 100 runs of the ACO algorithm. It can be seen that over $2,500$ runs across the 25 testcases, we found the optimal execution time 2,163 times. Based on this, the probability of finding the optimal solution with our algorithm for these task graphs is 86.44%. With the same amount of computation time, random sampling method has a 14.21% chance of discovering the

Table 3.1: Comparing ACO results with the random sampling *

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

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

optimal solution. Therefore, our ACO algorithm is statistically 6 times more effective in finding the optimal solution than random sampling. Related to this, we found that for 17 testing examples, or 68% of the testing set, our algorithm discovers the optimal partition every time in the 100 runs. This indicates that the proposed algorithm is

statistically robust in finding close to optimal solutions. Similar analysis holds when task graph size is 15 or 17.

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

Figure 3.6 provides another perspective regarding to the quality of our results. In this figure, the x axis is the percentage difference comparing the execution time of the partition found by the ACO algorithm with respect to the optimal execution time. The y axis is the percentage of the solutions that fall in that range.

These results may seem somewhat conflicting with the results shown in Figure 3.5. The results in Figure 3.5 show the results on how the ACO algorithm finds solutions that are within a top percentage of overall solutions. This graph shows the solution quality found by ACO. The results differ because while the ACO algorithm may not find the optimal solution, it almost always finds the next best feasible solution. However, the quality the next feasible solution in terms of execution time may not necessarily be close to the optimal solution. We believe that this has more to do with the solution distribution of the benchmarks than the quality of the algorithm.

For example, larger benchmarks are more likely to have more solutions whose quality is close to optimal. If this is the case, the ACO algorithm will likely find a good solution with a good solution quality as is show in Figure 3.5. Regardless, the quality of

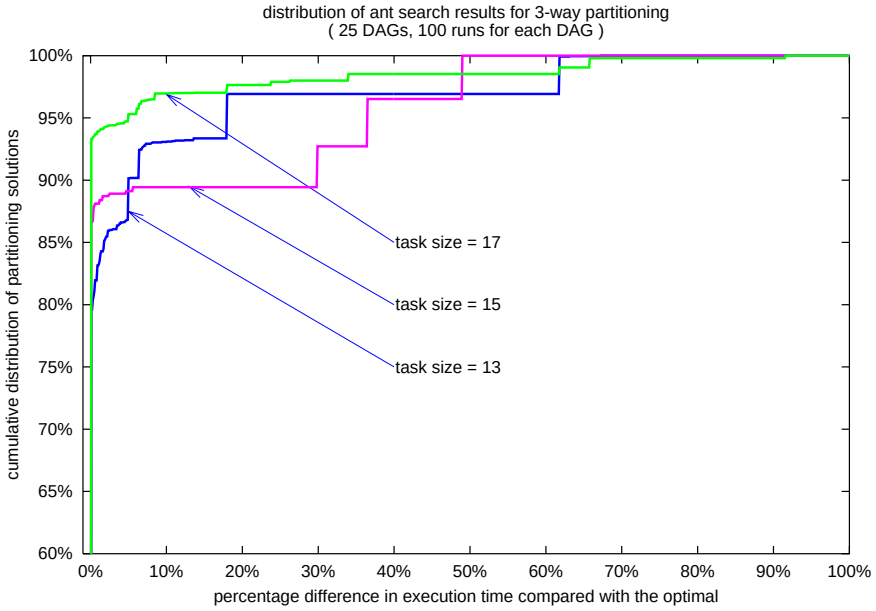


Figure 3.6: Execution time distribution

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

Based on the discussion in Section 3.2, when the ant number is 5 and iteration number is 50, for a three way partitioning problem over a 13 node task graph, the proposed algorithm has a theoretical execution time about 0.015% of that using brute force search, or 6,300 times faster. The experiments were conducted on a Linux machine with a 2.80 GHz Intel Pentium IV CPU with 512 MByte memory. The average actual execution time for the brute force method is 9.1 minutes while, on average, our ACO algorithm runs for 0.072 seconds. These runtimes are in scale with the theoretical speedup report in Section 3.2.4. To summarize the experiment results, with a high probability (88.12%), we can expect to achieve a result within top 1% of the search space with a very minor computational cost.

3.3.3 Comparing with Simulated Annealing

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

Our SA implementation is similar to the one reported in [106]. To begin the SA search, we randomly pick a feasible partition that obeys the cost constraint as the initial solution. The neighborhood of a solution contains all the feasible partitions that can be achieved by switching one of the tasks to a different computing resource from the one it is currently mapped to. The feasibility of the neighbors is computed in a similar way as in our ACO implementation. At every iteration of the SA search, a neighbor is randomly selected and the cost difference (i.e. execution time of the DAG) between the current solution and the neighboring solution is calculated. The acceptance of a more costly neighboring solution is then determined by applying the Boltzmann probability criteria [1], which depends on the cost difference and the annealing temperature. In our experiments, the most commonly known and used geometric cooling schedule [106] is applied and the temperature decrement factor is set to 0.9. When it reaches the pre-defined maximum iteration number or the stop temperature, the best solution found by SA is reported.

Because of the stochastic nature of the SA algorithm, for a given cooling approach, the more the iterations the better chance for SA to find higher quality results. However, as the iteration number increases, its execution time becomes longer. Figure 3.7 com-

pares the ACO results against those that achieved by the SA search sessions. The graph is illustrated in the same way as Figure 3.5. The SA sessions are configured in the same way except with different iteration numbers. Here SA50 has roughly the same execution time of our ACO implementation, while respectively, SA500 and SA1000 runs approximately 10 times and 20 times longer. We can see that with substantial less execution time, the ACO algorithm achieves better results than the SA approach, even when it is compared with a much more exhaustive SA session such as SA1000. In other words, in order to obtain comparable partition quality, SA suffers from much longer execution time. Furthermore, in order to compare the stability of the two different approaches, we also compared the variance of the results returned respectively by the SA and the proposed algorithm. The is done by carrying multiple runs of ACO and SA independently. This comparison indicates that the ACO approach consistently provides significantly more stable results than SA. For some testing cases, the variance on the SA results can be more than 3 times wider. Thus experimentally we perhaps can conclude that the ACO approach would have much better chance in obtaining high quality results than the SA method with the same execution cost.

Another benefit of conducting comparison between SA and ACO is that it provides a way for us to assess the quality of the proposed algorithm on bigger size testing cases. For such problems, it becomes impossible for us to perform the brute force search to find the true optimal solution for the problem. However, we can still assess the quality of the proposed algorithm by comparing relative difference between its results with

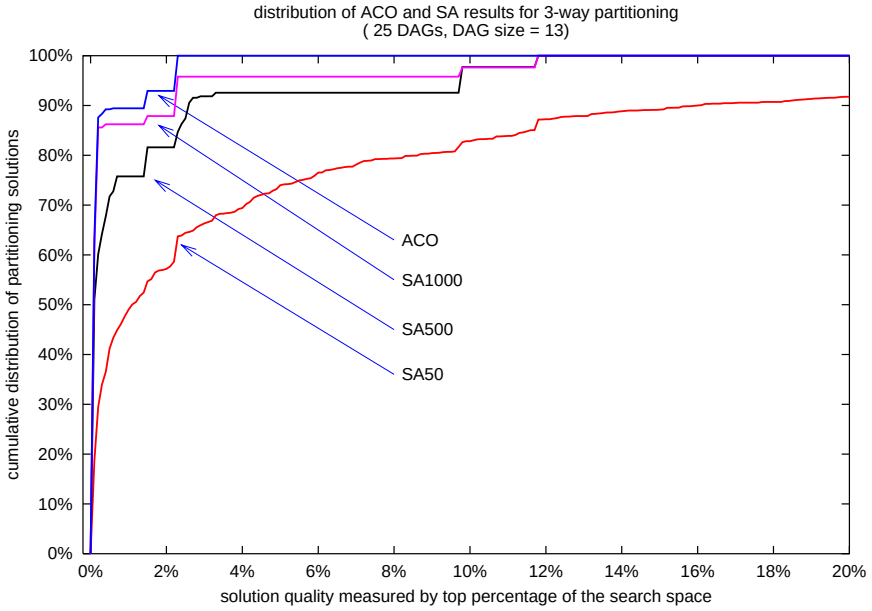


Figure 3.7: Comparing ACO with SA

that obtained by using other popularly used heuristic methods, such as SA. Figure 3.8 shows the cumulative result quality distribution curves for task graphs with 25 nodes. For these problems, it is estimated that the brute force method would take hundreds machine hours thus impractical for us to find the optimal exactly. In the figure, the x axis now reads as the percentage difference on the execution time of the partition found by the corresponding algorithm with respect to the *best* execution time over all the experiments using different approaches. Among them, the ACO and SA500 have the same amount of execution time, while SA5000 runs at about 10 time slower. It is shown that ACO outperforms SA500 while a much more expensive SA works comparably.

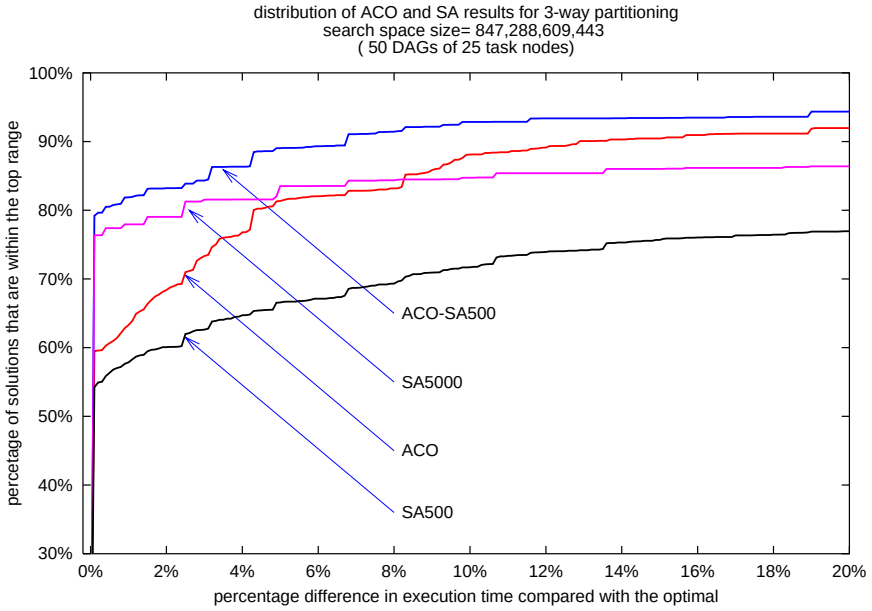


Figure 3.8: ACO, SA and ACO-SA on big size problems

3.3.4 Hybrid ACO with Simulated Annealing

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

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

On the other hand, the ACO/ATG model makes effective use of the core structural information of the problem. The autocatalytic nature of how the pheromone trails are updated and utilized makes it more attractive in discovering ”good” solutions with short computing time. However, this very behavior raises stagnation problem. For example, it is observed that allowing extra computing time after enough iterations of the ACO algorithm does not have significant benefit regarding to the solution quality. This stagnation problem has been discussed in other works [37, 28, 13, 27] and special problem-dependent recovery mechanisms have to be formulated to ease this artifact.

These complementary characteristics of the two methods motivate us to investigate a hybrid approach that combines the ACO and SA together. That is to use the ACO results as the initial seed partitions for the SA algorithm, it is possible for us to achieve even better system performance with a substantially reduced computing cost. In Figure 3.8, curve ACO-SA500 shows the result of this approach. It achieves definitively better results comparing with that of SA5000 while only taking about 20% of its running time. Similar result holds for task graphs with bigger sizes, such as 50 and 100 (for a test case with 100 task node, the computing time can be reduced from about 2 hours

to 18 minutes using the hybrid ACO-SA approach with comparable result quality).

Table 3.2: Average Result Quality Comparison

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

Overall, we summarize the result quality comparison with Table 3.2 for problems with big sizes. It compares the average result qualities reported by ACO, SA500, SA5000 and the hybrid method ACO-SA500. The data is normalized with that obtained by SA500, and the smaller the better. It is easy to see that ACO always outperforms the traditional SA even when SA is allowed a much longer execution time, and the ACO-SA approach provides the best average results consistently with great runtime reduction.

3.4 Application: Quick Design Parameter Estimation

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

For instance, considering the following case: assuming one design is realized with a PowerPC and a FPGA component (Architecture 1), while an alternative design contains an extra DSP core (Architecture 2), one needs to quickly evaluate design parameters associated with each of the two possible approaches. Does adding an extra DSP result in FPGA area reduction and if yes, how much can we save? Does the second design provide significant improvement of system's timing performance? Or by having an extra DSP, how much FPGA cost can be saved without tempting the system's time performance requirement? In order to address these questions, quick assessment on related design parameters is needed. Essentially, the above problem request us to provide insights for design parameters when the number of computing resources is incremented. The high quality and fast execution time of the proposed ACO multi-way application partitioning approach provides a possible method for certain situations for such a system level design task.

To see this, we cross examine the results of the proposed algorithm over the testing cases illustrated in Table 3.1 for the two architectures. Based on the available resources, they can be viewed as 3-way partitioning and bi-partitioning problems under our model respectively and the proposed ACO approach solves them in a uniformed way.

Based on this comparison, we find that with the same hardware area constraint, our algorithm robustly provides partitions with better or at least the same execution time for Architecture 2 for different test cases in our benchmarks. The speedup is dependent on the specific application, i.e. the application's ATG and the tasks associated with it.

With our testing cases, we have an average execution time speedup of 1.6% over the 25 testing examples, while over 11% speedup is observed for examples DAG-6 and DAG-17. More interestingly, based on the same test, we find that the 3-way partitioning results have an average 12.01% save in hardware area for the FPGA component compared with the bi-partitioning results. In 100 runs, the expected biggest area save over 25 DAGs is 12.61%, which is roughly in agreement with the average savings.

This motivates us to use the proposed ACO algorithm as a quick estimator for design parameters, such as the FPGA area cost constraint, when a new computing resource is included. The question the designer tries to answer here is: how much FPGA area can we save by adding a DSP core in the system while respecting the system delay constraint? Or what is the right FPGA area cost constraint we should provide for the incremented system? Without a quick design parameter assessment method, this constraint is hard to be made accurately. To address this problem, we propose a two step process using the ACO application partition as such quick estimator, as the process is diagramed in Figure 3.9.

First, we notice that Architecture 2, which contains an extra DSP, is expected to not make the FPGA cost worse. Based on this observation, a designer can first conduct bi-partitioning for the application over Architecture 1. The results will provide critical guidance regarding to the time performance and the upper bound of the FPGA area cost. The designer can then use the FPGA area cost result returned by our algorithm as the “desired” constraint for the 3-way partitioning problem over Architecture

2. Of course, this step may require multiple iterations if the optimal FPGA saving is expected. Thanks for the low computing cost of the proposed ACO approach, such iterative process is practical and can be conducted within reasonable time. As shown in Figure 3.9, for each of the iterations, we check if the system delay meets the time performance constraint. If yes, it implies that a more stringent area cost constraint can be used. Otherwise, we have found the optimal saving and the process terminates. By applying this method, without noticeable degradation on the execution time (less than 2%), our experiments on the testing cases show that an average hardware area reduction of 65.46% for the 3-way architecture comparing with original design which only uses PowerPC and FPGA.

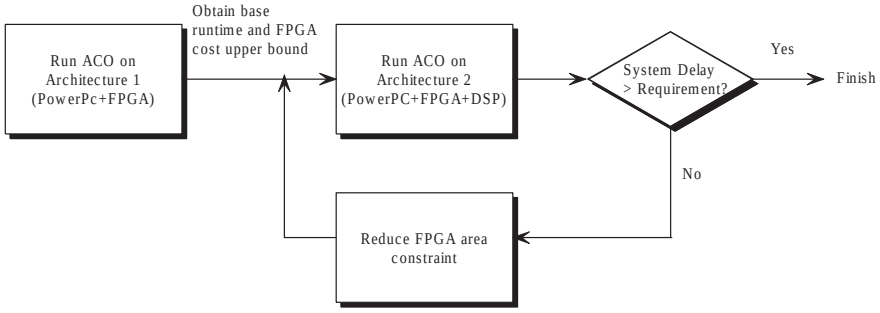


Figure 3.9: Estimate Design Parameters with ACO application partitioner on design choice with incremented resources

Notice this is just one of the possible scenarios that the proposed algorithm could help. There are other cases such a quick parameter estimator could be useful. For instance, by simply swapping the boxes associated with Architecture 1 and Architecture 2 in Figure 3.9, we can help to solve the reverse design problem, where we try to find how much extra FPGA resource we would need if we simplify the system design by

excluding the DSP core from the architecture.

3.5 Summary

In this work, we presented a novel heuristic searching method for the system partitioning problem based on the ACO techniques. Our algorithm proceeds as a collection of agents work collaboratively to explore the search space. A stochastic decision making strategy is proposed in order to combine global and local heuristics to effectively conduct this exploration. We introduced the Augmented Task Graph concept as a generic model for the system partitioning problem, which can be easily extended as the resource number grows and it fits well with a variety of system requirements.

Experimental results over our test cases for a 3-way system partitioning task showed promising results. The proposed algorithm consistently provided near optimal partitioning results over modestly sized tested examples with very minor computational cost. Our algorithm is more effective in finding the near optimal solutions and scales well as the problem size grows. It is also shown that for large size problems, with substantial less execution time, the proposed method achieves better solutions than the popularly used simulated annealing approach. With the observation of the complementary behaviors of the algorithms, we proposed a hybrid approach that combines the ACO and SA together. This method yields even better result than using each of the algorithms individually.

Chapter 4

Operation Scheduling

Operation scheduling is a fundamental problem in mapping an application to a computational device. It takes a behavioral application specification and produces a schedule solution for the operations onto a collection of processing units to either minimize the completion time or the computing resources required to meet a given deadline. The operation scheduling problem is $\mathcal{NP}$-hard, thus effective heuristic methods are necessary to provide qualitative solutions. We present novel operation scheduling algorithms using the Ant Colony Optimization approach for both timing and resource constrained scheduling problems. The algorithms use a unique hybrid approach by combining the MAX-MIN ant system meta-heuristic with traditional scheduling heuristics. We compiled a comprehensive testing benchmark set from real-world applications in order to verify the effectiveness and efficiency of our proposed algorithms. For timing constrained scheduling, our algorithm achieves better results compared with force-directed scheduling on almost all the testing cases with a maximum 19.5% reduction of the number of resources. For resource constrained scheduling, our algorithm outperforms

a number of different list scheduling heuristics with better stability, and generates better results with up to 14.7% improvement(on average 6.2% better). Furthermore, by solving the test samples optimally using ILP formulation, we show that our algorithm consistently achieves a near optimal solution. Our algorithms outperform the simulated annealing method for both scheduling problems in terms of quality, computing time and stability.

4.1 Introduction

As fabrication technology advances and transistors become more plentiful, modern computing systems can achieve better system performance by increasing the amount of computation units. It is estimated that we will be able to integrate more than a half billion transistors on a 468 mm^2 chip by the year of 2009 [85]. This yields tremendous potential for future computing systems, however, it imposes big challenges on how to effectively use and design such complicated systems.

As computing systems become more complex, so do the applications that can run on them. Designers will increasingly rely on automated design tools in order to map applications onto these systems. One fundamental process of these tools is mapping a behavioral application specification to the computing system. For example, the tool may take a C function and create the code to program a microprocessor. This is viewed as software compilation. Or the tool may take a transaction level behavior and create a register transfer level (RTL) circuit description. This is called hardware or behavioral

synthesis [72]. Both software and hardware synthesis flows are essential for the use and design of future computing systems.

Operation scheduling (OS) is an important problem in software compilation and hardware synthesis. An inappropriate scheduling of the operations can fail to exploit the full potential of the system. Operation scheduling appears in a number of different problems, e.g. compiler design for superscalar and VLIW microprocessors [54], distributed clustering computation architectures [5] and behavioral synthesis of ASICs and FPGAs [72]. In this work, we focus on operation scheduling for behavioral synthesis for ASICs/FPGAs. However, the basic algorithms proposed here can be modified to handle a wide variety of operation scheduling problems.

Operation scheduling is performed on a behavioral description of the application. This description is typically decomposed into several blocks (e.g. basic blocks), and each of the blocks is represented by a data flow graph (DFG). Figure 4.1 shows an example DFG for a one-dimensional 8-point fast discrete cosine transformation.

Operation scheduling can be classified as *resource constrained* or *timing constrained*. Given a DFG, clock cycle time, resource count and resource delays, a resource constrained scheduling finds the minimum number of clock cycles needed to execute the DFG. On the other hand, a timing constrained scheduling tries to determine the minimum number of resources needed for a given deadline.

In the timing constrained scheduling problem (also called fixed control step scheduling), the target is to find the minimum computing resource cost under a set

of given types of computing units and a predefined latency deadline. For example, in many digital signal processing (DSP) systems, the sampling rate of the input data stream dictates the maximum time allowed for computation on the present data sample before the next sample arrives. Since the sampling rate is fixed, the main objective is to minimize the cost of the hardware. Given the clock cycle time, the sampling rate can be expressed in terms of the number of cycles that are required to execute the algorithm.

Resource constrained scheduling is also found frequently in practice. This is because in a lot of the cases, the number of resources are known a priori. For instance, in software compilation for microprocessors, the computing resources are fixed. In hardware compilation, DFGs are often constructed and scheduled almost independently. Furthermore, if we want to maximize resource sharing, each block should use same or similar resources, which is hardly ensured by time constrained schedulers. The time constraint of each block is not easy to define since blocks are typically serialized and budgeting global performance constraint for each block is not trivial [69].

Operation scheduling methods can be further classified as *static scheduling* and *dynamic scheduling* [88]. Static operation scheduling is performed during the compilation of the application. Once an acceptable scheduling solution is found, it is deployed as part of the application image. In dynamic scheduling, a dedicated system component makes scheduling decisions on-the-fly. Dynamic scheduling methods must minimize the program's completion time while considering the overhead paid for running the

scheduler.

In this chapter, we focus on both resource and timing constrained static operation scheduling. We propose iterative algorithms based on the MAX-MIN Ant Colony Optimization for solving these problems. In our algorithms, a collection of agents (ants) cooperate together to search for a solution. Global and local heuristics are combined in a stochastic decision making process in order to efficiently explore the search space. The quality of the resultant schedules is evaluated and fed back to dynamically adjust the heuristics for future iterations. The main contribution of our work is the formulation of scheduling algorithms that:

- Utilize a unique hybrid approach combining traditional heuristics and the recently developed MAX-MIN ant system optimization [92];
- Dynamically use local and global heuristics based on the input application to adaptively search the solution space;
- Generate consistently good scheduling results over all testing cases compared with a range of list scheduling heuristics, force-directed scheduling, simulated annealing and the optimal ILP solution, and demonstrates stable quality over a variety of application benchmarks of large size.

This chapter is organized as follows. We formally define the timing constrained and resource constrained scheduling problems in Section 4.2. Then in Section 4.3 and Section 4.4, we present two hybrid approaches combining traditional scheduling heuristics with the MAX-MIN ant system optimization to solve the timing and resource constrained scheduling problems, respectively. We discuss the construction of our bench-

marks in Section 4.5. Experimental results for the new algorithms are presented and analyzed in Section 4.6. We summarize with Section 4.7.

4.2 Preliminaries

4.2.1 Operation Scheduling Problem Definition

Given a set of operations and a collection of computational units, the resource constrained scheduling (RCS) problem schedules the operations onto the computing units such that the execution time of these operations are minimized, while respecting the capacity limits imposed by the number of computational resources. The operations can be modeled as a data flow graph (DFG) $G(V,E)$, where each node $v_i \in V(i = 1,\ldots,n)$ represents an operation op_i, and the edge e_{ij} denotes a dependency between operations v_j and v_i. A DFG is a directed acyclic graph where the dependencies define a partially ordered relationship (denoted by the symbol $\preccurlyeq$) among the nodes. Without affecting the problem, we add two virtual nodes *root* and *end*, which are associated with no operation (NOP). We assume that *root* is the only starting node in the DFG, i.e. it has no predecessors, and node *end* is the only exit node, i.e. it has no successors.

Additionally, we have a collection of computing resources, e.g. ALUs, adders, and multipliers. There are R different types and $r_j > 0$ gives the number of units for resource type j $(1 \leqslant j \leqslant R)$. Furthermore, each operation defined in the DFG must be executable on at least one type of the resources. When each of the operations is uniquely associated with one resource type, we call it *homogenous* scheduling. If an

operation can be performed by more than one resource types, we call it *heterogeneous* scheduling [94]. Moreover, we assume the cycle delays for each operation on different type resources are known as $d(i, j)$. Of course, *root* and *end* have zero delays. Finally, we assume the execution of the operations is non-preemptive, that is, once an operation starts execution, it must finish without being interrupted.

A resource constrained schedule is given by the vector

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

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

1. An operation can only start when all its predecessors have finished, i.e. $s_i \geqslant f_j$ if $op_j \preccurlyeq op_i$;

2. At any given cycle t, the number of resources needed is constrained by r_j, for all $1 \leqslant j \leqslant R$.

The timing constrained scheduling (TCS) is a dual problem of the resource constrained version and can be defined using the same terminology presented above. Here the target is to minimize total resources $\sum_j r_j$ or the total cost of the resources (e.g. the hardware area needed) subject to the same dependencies between operations imposed by the DFG and a given deadline D, i.e. $s_{end} < D$.

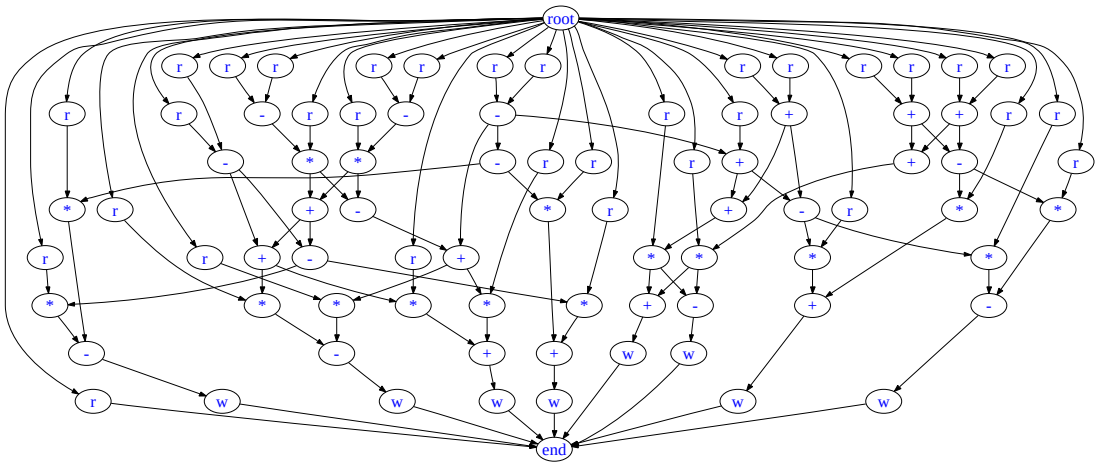


Figure 4.1: Data Flow Graph (DFG) of the *cosine2* benchmark
('r' is for memory read and 'w' for memory write).

4.2.2 Related Work

Many variants of the operation scheduling problem are $\mathcal{NP}$-hard [12]. Although it is possible to formulate and solve them using Integer Linear Programming (ILP) [107], the feasible solution space quickly becomes intractable for larger problem instances. In order to address this problem, a range of heuristic methods with polynomial runtime complexity have been proposed.

The integer linear programming (ILP) method [60]tries to find an optimal schedule using a branch-and-bound search algorithm. It also involves some amount of back-tracking, i.e., decisions made earlier are changed later on. A simplified formulation of the ILP method for the time-constrained problem is given below:

First it calculates the mobility range for each operation $M = \{S_j | E_k \leqslant j \leqslant L_k\}$, where E_k and L_k are the ASAP and ALAP values respectively. The scheduling problem

in ILP is defined by the following equations:

$$Min(\sum_{k=1}^{n}(C_k * N_k)) \text{ while } \sum_{E_i \leqslant j \leqslant L_i} x_{ij} = 1$$

where $1 \leqslant i \leqslant n$ and n is the number of operations. There are $1 \leqslant k \leqslant m$ operation types available, and N_k is the number of FUs of operation type k, and C_k is the cost of each FU. Each x_{ij} is 1 if the operation i is assigned in control step j and 0 otherwise. Two more equations that enforce the resource and data dependency constraints are:

$$\sum_{i=1}^{n} x_{ij} \leqslant N_i$$

and

$$((q * x_{j,q}) - (p * x_{i,p})) \leqslant -1, p \leqslant q$$

where p and q are the control steps assigned to the operations x_i and x_j respectively.

We can see that the ILP formulation increases rapidly with the number of control steps. For one unit increase in the number of control steps we will have n additional x variables. Therefore the time of execution of the algorithm also increases rapidly. In practice the ILP approach is applicable only to very small problems.

Many timing constrained scheduling algorithms used in high level synthesis are derivatives of the force-directed scheduling (FDS) heuristic presented by Paulin and Knight [78, 79]. Verhaegh *et al.* [97, 98] provide a theoretical treatment on the original FDS algorithm and report better results by applying gradual time-frame reduction and the use of global spring constants in the force calculation. Due to the lack of a look ahead scheme, the FDS algorithm is likely to produce a sub-optimal solution. One way

to address this issue is the iterative method proposed by Park and Kyung [76] based on Kernighan and Lin's heuristic [55] method used for solving the graph-bisection problem. In their approach, each operation is scheduled into an earlier or later step using the move that produces the maximum gain. Then all the operations are unlocked and the whole procedure is repeated with this new schedule. The quality of the result produced by this algorithm is highly dependent upon the initial solution. More recently, Heijligers *et al.* [48] and InSyn [86] use evolutionary techniques like genetic algorithms and simulated evolution.

There are a number of heuristic algorithms devised for the resource constrained problem, including list scheduling [84, 2, 80, 94, 2], forced-directed list scheduling [78], genetic algorithm [11, 41], tabu search [10], simulated annealing [93], critical path based heuristic [17], graph theoretic and computational geometry approaches [5, 69, 5]. Among them, list scheduling is the most common due to its simplicity of implementation and capability of generating reasonably good results for small sized problems. The success of the list scheduler is highly dependent on the priority function and the structure of the input application (DFG) [93, 72, 57]. One commonly used priority function assigns the priority inversely proportional to the mobility. This ensures that the scheduling of operations with large mobilities are deferred because they have more flexibility as to where they can be scheduled. Many other priority functions have been proposed [2, 57, 41, 8]. However, it is commonly agreed that there is no single good heuristic for prioritizing the DFG nodes across a range of applications using list

scheduling. Our results in Section 4.6 confirm this.

4.3 ACO for Timing Constrained Scheduling

In this section, we introduce our MMAS-based algorithms for solving the timing constrained scheduling problem. As discussed in Section 4.2, force-directed scheduling (FDS) is a commonly used heuristic as it generates "good" quality results for moderately sized DFGs. Our algorithm uses distribution graphs from FDS as a local heuristic. Additionally, we use the results produced by FDS to evaluate the quality of our algorithm. For these reasons, we provide some details of FDS in the following subsection. The remaining subsections describe our MMAS algorithm for timing constrained scheduling.

4.3.1 Force-Directed Scheduling

The force-directed scheduling algorithm (and its various forms) has been widely used since it was first proposed by Paulin and Knight [78]. The goal of the algorithm is to reduce the number of functional units used in the implementation of the design. This objective is achieved by attempting to uniformly distribute the operations onto the available resource units. The distribution ensures that resource units allocated to perform operations in one control step are used efficiently in all other control steps, which leads to a high utilization rate.

The FDS algorithm relies on both the ASAP and the ALAP scheduling algorithms to determine the feasible control steps for every operation op_i, or the ***time frame*** of op_i

(denoted as $[t_i^S, t_i^L]$ where t_i^S and t_i^L are the ASAP and ALAP times respectively). It also assumes that each operation op_i has a uniform probability of being scheduled into any of the control steps in the range, and zero probability of being scheduled elsewhere. Thus, for a given time step j and an operation op_i which needs $\triangle_i \geqslant 1$ time steps to execute, this probability is given as:

$$\mathsf{p}_j(op_i) = \begin{cases} (\sum_{l=0}^{\triangle_i} h_i(j-l))/(t_i^L - t_i^S + 1) & \text{if } t_i^S \leqslant j \leqslant t_i^L \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

where $h_i(\cdot)$ is a unit window function defined on $[t_i^S, t_i^L]$.

Based on this probability, a set of ***distribution graphs*** can be created, one for each specific type of operation, denoted as q_k. More specifically, for type k at time step j,

$$q_k(j) = \sum_{op_i} \mathsf{p}_j(op_i) \qquad \text{if type of } op_i \text{ is } k \tag{4.2}$$

We can see that $q_k(j)$ is an estimation on the number of type k resources that are needed at control step j.

The FDS algorithm tries to minimize the overall concurrency under a fixed latency by scheduling operations one by one. At every time step, the effect of scheduling each unscheduled operation on every possible time step in its frame range is calculated, and the operation and the corresponding time step with the smallest negative effect is selected. This effect is equated as the force for an unscheduled operation op_i at control step j, and is comprised of two components: the self-force, SF_{ij}, and the predecessor-successor forces, PSF_{ij}.

The self-force SF_{ij} represents the direct effect of this scheduling on the overall concurrency. It is given by:

$$SF_{ij} = \sum_{l=t_i^S}^{t_i^L+\triangle_i} q_k(l)(H_i(l) - \mathsf{p}_i(l)) \tag{4.3}$$

where, $j \in [t_i^S, t_i^L]$, k is the type of operation op_i, and $H_i(\cdot)$ is the unit window function defined on $[j, j+\triangle_i]$.

We also need to consider the predecessor and successor forces since assigning operation op_i to time step j might cause the time frame of a predecessor or successor operation op_l to change from $[t_l^S, t_l^L]$ to $[\widetilde{t}_l^S, \widetilde{t}_l^S]$. The force exerted by a predecessor or successor is given by:

$$PSF_{ij}(l) = \sum_{m=\widetilde{t}_i^S}^{\widetilde{t}_i^L+\triangle_l} (q_k(m) \cdot \widetilde{\mathsf{p}}_m(op_l)) - \sum_{m=t_i^S}^{t_i^L+\triangle_l} (q_k(m) \cdot \mathsf{p}_m(op_l)) \tag{4.4}$$

where $\widetilde{\mathsf{p}}_m(op_l)$ is computed in the same way as Equation (4.1) except the updated mobility information $[\widetilde{t}_l^S, \widetilde{t}_l^S]$ is used. Notice that the above computation has to be carried for all the predecessor and successor operations of op_i. The total force of the hypothetical assignment of scheduling op_i on time step j is the addition of the self-force and all the predecessor-successor forces, i.e.

$$\text{total force}_{ij} = SF_{ij} + \sum_l PSF_{ij}(l) \tag{4.5}$$

where op_l is a predecessor or successor of op_i. Finally, the total forces obtained for all the unscheduled operations at every possible time step are compared. The operation and time step with the best force reduction is chosen and the partial scheduling result is

incremented until all the operations have been scheduled. A pseudo implementation of FDS is given as Algorithm 2.

The FDS method is "constructive" because the solution is computed without performing any backtracking. Every decision is made in a greedy manner. If there are two possible assignments sharing the same cost, the above algorithm cannot accurately estimate the best choice. Based on our experience, this happens fairly often as the DFG becomes larger and more complex. Moreover, FDS does not take into account future assignments of operators to the same control step. Consequently, it is likely that the resulting solution will not be optimal, due to the lack of a look ahead scheme and the lack of compromises between early and late decisions.

Our experiments show that a baseline FDS implementation based on [78] fails to find the optimal solution even on small testing cases. To ease this problem, a look-ahead factor was introduced in the same paper. A second order term of the displacement weighted by a constant η is included in force computation, and the value η is experimentally decided to be $1/3$. In our experiments, this look-ahead factor has a positive impact on some testing cases but does not always work well. More details regarding FDS performance can be found in Section 4.6.

4.3.2 Algorithm Formulation

We address the timing constrained scheduling (TCS) problem in an evolutionary manner. The proposed algorithm is built upon the Ant System approach and the TCS

procedure FDS(G,R)

input: DFG $G(V,E)$, resource set R, and a map of operation to one resource in R

output: instruction schedule

initialize schedule result $S_{current}$ to be empty
while exists unscheduled instruction **do**
  perform ASAP and ALAP on partial schedule result $S_{current}$
  update time frame $[t_i^S, t_i^L]$ associated with each instruction op_i
  $Min = \infty$
  for each unscheduled instruction op_i **do**
    for $t_i^S \leqslant j \leqslant t_i^L$ **do**
      $S_{tmp} = schedule(S_{current}, op_i, j)$
      Update time frame and distribution graphs based on S_{tmp}
      Compute SF_{ij} and set $total\_force_{ij} = SF_{ij}$
      for each predecessor/sucessor op_l of op_i **do**
        Compute $PSF_{ij}(l)$
        $total\_force_{ij} += PSF_{ij}(l)$
      end for
      if $total\_force_{ij} < Min$ **then**
        $Min = total\_force_{ij}$
        $BestOp = op_i; BestStep = j$
      end if
    end for
  end for
  $S_{current} = schedule(S_{current}, BestOp, BestStep)$
  Update time frame and distribution graphs based on $S_{current}$
end while
return $S_{current}$ and the resource cost

Algorithm 2: Force-Directed Scheduling for Time-Constrained Optimization

problem is formulated as an iterative searching process. Each iteration consists of two stages. First, the ACO algorithm is applied in which a collection of ants traverse the DFG to construct individual operation schedules with respect to the specified deadline using global and local heuristics. Second, these results are evaluated using their resource costs. The heuristics are adjusted based on the solutions found in the current iteration. The hope is that future iterations will benefit from this adjustment and come up with better schedules.

Each operation or DFG node op_i is associated with D pheromone trails τ_{ij}, where $j = 1, \ldots, D$ and D is the specified deadline. These pheromone trails indicate the global favorableness of assigning the i-th operation at the j-th control step in order to minimize the resource cost with respect to the time constraint. Initially, based on ASAP and ALAP results, τ_{ij} is set with some fixed value τ_0 if j is a valid control step for op_i; otherwise, it is set to be 0.

For each iteration, m ants are released and each ant individually starts to construct a schedule by picking an unscheduled operation and determining its desired control step. However, unlike the deterministic approach used in the FDS method, each ant picks up the next operation probabilistically. The simplest way is to select an operation uniformly among all unscheduled operations. Once an operation op_h is selected, the ant needs to make a decision on which control step it should beassigned to. This decision is also made probabilistically according to Equation (4.6).

$$p_{hj} = \begin{cases} \frac{\tau_{hj}(t)^{\alpha} \cdot \eta_{hj}^{\beta}}{\sum_l (\tau_{hl}^{\alpha}(t) \cdot \eta_{hl}^{\beta})} & \text{if } op_h \text{ can be scheduled at } l \text{ and } j \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

Here j is the control step under consideration, which is between op_h's time frame $[t_h^S, t_h^L]$. The item η_{hj} is the local heuristic for scheduling operation op_h at control step j, and α and β are parameters to control the relative influence of the distributed global heuristic τ_{hj} and local heuristic η_{hj}. In our work, assuming op_h is of type k, we simply set η_{hj} to be the inverse of $q_k(j)$; that is the distribution graph value of type k at control step j (calculated in the same way as in FDS). Recalling our discussion in Section 4.3.1, q_k is computed based on partial scheduling result and is an indication on the number of computing units of type k needed at control step j. Intuitively, the ant favors a decision that possesses higher volume of pheromone and better local heuristic, i.e. a lower q_k. In other words, an ant is more likely to make a decision that is globally considered "good" and also uses the fewest number of resources under the current partially scheduled result. Similar to FDS, once an operation is fixed at a time step, it will not change. Furthermore, the time frames will be updated to reflect the changed partial schedule. This guarantees that each ant will always construct a valid schedule.

In the second stage of our algorithm, the ant's solutions are evaluated. The quality of the solution from ant h is judged by the total number of resources, i.e. $Q_h = \sum_k r_k$. At the end of the iteration, the pheromone trail is updated according to the quality of individual schedules. Additionally, a certain amount of pheromone evaporates. More

specifically, we have:

$$\tau_{ij}(t) = \rho \cdot \tau_{ij}(t) + \sum_{h=1}^{m} \Delta\tau_{ij}^{h}(t) \qquad \text{where } 0 < \rho < 1. \tag{4.7}$$

Here ρ is the evaporation ratio, and

$$\Delta\tau_{ij}^{h} = \begin{cases} Q/Q_h & \text{if } op_i \text{ is scheduled at } j \text{ by ant } h \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

Q is a fixed constant to control the delivery rate of the pheromone. Two important operations are performed in the pheromone trail updating process. Evaporation is necessary for ACO to effectively explore the solution space, while reinforcement ensures that the favorable operation orderings receive a higher volume of pheromone and will have a better chance of being selected in the future iterations. The above process is repeated multiple times until an ending condition is reached. The best result found by the algorithm is reported.

In our experiments, we implemented both the basic ACO and the MMAS algorithms. The latter consistently achieves better scheduling results, especially for larger DFGs. A pseudo code implementation of the final version of our TCS algorithm using MMAS is shown as Algorithm 3, where the pheromone bounding step is indicated as step 23.

procedure MaxMinAntSchedulingTCS(G,R)

input: DFG $G(V,E)$, resource set R

output: operation schedule

initialize parameter $\rho, \tau_{ij}, p_{best}, \tau_{max}, \tau_{min}$
construct m ants
$BestSolution \leftarrow \phi$
while ending condition is not met **do**
    for $i = 0$ **to** m **do**
        $ant(i)$ constructs a valid schedule timing constrained $S_{current}$ as following:
        $S_{current} \leftarrow \phi$
        perform ASAP and ALAP
        while exists unscheduled operation **do**
            update time frame $[t_i^S, t_i^L]$ associated with each operation op_i and the distribution graphs q_k.
            select one operation op_h among all unscheduled operations probabilistically
            for $t_h^S \leqslant j \leqslant t_h^L$ **do**
                set local heuristic $\eta_{hj} = 1/q_k(j)$ where op_h is of type k
            end for
            select time step l using η and τ as Equation (4.6).
            $S_{current} = schedule(S_{current}, op_h, l)$
            Update time frame and distribution graphs based on $S_{current}$
        end while
        if $S_{current}$ is better than that of *BestSolution* **then**
            $BestSolution \leftarrow S_{current}$
        end if
    end for
    update τ_{max} and τ_{min} based on Equation (2.3) and (2.4)
    update η **if needed**
    update τ_{ij} based on Equation (4.7)
end while
return *BestSolution*

Algorithm 3: MMAS for Timing Constrained Scheduling

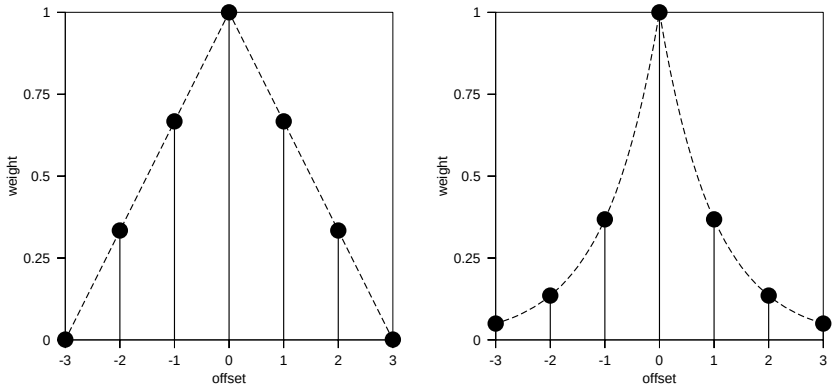
4.3.3 Refinements

Updating Neighboring Pheromone Trails

We found that a "better" solution can often be achieved from a "good" scheduling result by simply adjusting very few operations' scheduled positions within their time frames. Based on this observation, we can refine our pheromone update policy to encourage exploration of the neighboring positions. More specifically, in the pheromone reinforcement step indicated by Equation 4.8, we also increase the pheromone trails of the control steps adjacent position j subject to a weighted function window. Two such windowing functions are shown in Figure 4.2. Depending on the neighbor's offset from j, the two functions adjust its pheromone trail in a similar manner to Equation 4.8 but with an extra factor applied. Assuming we use x to represent the offset, then Figure 4.2(a) has a weight function of $1 - 1/3|x|$ while Figure 4.2(b) provides a weight function of $e^{-|x|}$. In our experiments, the latter provides relatively better performance. Ideally, the weight function window size shall be computed based on the mobility ranges of the operations. However, to keep the algorithm simple, we use a window size 5 across all our experiments, subject to the operation's time frame $[t_i^S, t_i^L]$. This number is estimated using the average mobility ranges of all testing cases.

Operation Selection

In our algorithm, the ants construct a schedule for the given DFG by making two decisions in sequence. First, it needs to select the next operation. Then a specific control



(a) (b)

Figure 4.2: Pheromone update windows

step is determined for the selected operation. As discussed earlier, the simplest approach for selecting an operation is to randomly pick one amongst all the unscheduled operations. Though it is simple and computationally effective, it does not appreciate the information accumulated in the pheromone from the previous iterations; it also ignores the dynamic time frame information. One possible refinement is to make the selection probability proportional to the pheromone and inversely proportional to the size of the operation's time frame at that instance. More precisely, we pick the next operation op_i probabilistically with the following equation:

$$\mathsf{p}_i = \frac{\frac{\sum_j \tau_{ij}}{(t_i^L - t_i^S + 1)}}{\sum_l \frac{\sum_k \tau_{lk}}{(t_l^L - t_l^S + 1)}} \tag{4.9}$$

Here the numerator can be viewed as the average pheromone value over all possible positions in the current time frame for operation op_i. The denominator is a normalization factor to bring the result to be a valid probability value between 0 and 1. It is basically the addition of the average of pheromone for all the unscheduled operations op_l. Notice

that as the time frames of the operations change dynamically depending on the partial schedule, the average pheromone trail is not constant during the schedule construction process. In other words, we only consider a pheromone τ_{ij} when $t_i^S \leqslant j \leqslant t_i^L$.

Intuitively, this formulation favors an operation with stronger pheromone and fewer possible scheduling alternatives. In the extreme case, $t_i^L = t_i^S$, which means operation op_i is on the critical path, we will have only one choice for op_i. If the pheromone for op_i at this position happens to be very strong, we will have better chance to pick op_i at the next step compared with other operations. Our experiments show that applying this operation selection policy makes the algorithm faster in identifying high quality results. Compared with the even possibility approach, there is an overhead to perform this operation selection policy. However, by making the selection more targeted, it allows us to reduce the overall iteration number of the algorithm thus the additional overhead is well worth it. In our experiments, we were able to reduce the total runtime by about 23% while achieving almost the same quality with our testing results by adopting this biased selection policy.

4.3.4 Extensions

Our proposed TCS algorithm applies the Ant Colony meta-heuristic at the high level. It poses little difficulty to extend it to handle different scheduling contexts. Most of the methods proposed previously for FDS can be readily implemented within our framework.

Resource Preference

In our work, the target is to minimize the total count of resources needed. Accordingly, we use the inverse of this total count as the quality of the scheduling result. This quality measurement is further used to adjust the pheromone trails. However, in practice, we may have unbalanced hardware costs for different resource types. With this consideration, we might find that we prefer a schedule that requires 3 multipliers and 4 adders rather than one that needs 4 multipliers and 3 adders, even though both schedules have the same total number (7) of resources. This issue can be handled in our algorithm simply by introducing a cost factor c_k for each resource type and modifying the quality of the schedule to this weighted resource cost,

$$Q_h = \sum_k (c_k r_k) \tag{4.10}$$

By adjusting the c_k assigned to different resource types, we can control the preference in our schedule results.

Multi-cycle Operation

No change is needed for our algorithm to handle multi-cycle operation since it uses dynamically computed time frames. Also, as presented in Section 4.3.1, the distribution graph handles multi-cycle operations naturally.

Mutually Exclusive Operations

Mutually exclusive operations occur when operations are located in different branches of the program. This happens in *if-then-else* and *case* statements in high-level languages. With the proposed algorithm, we do not need to add any extra constraints for handling such operations; thus the approach proposed in [78] is still valid.

Chained Operations

When the total delay of consecutive operations is less than a clock cycle, it is possible to chain the operations during scheduling. The same techniques used in [78] can be directly applied within our approach, where chaining is handled by extending the ASAP and ALAP computation to obtain the time frames for the operations.

Pipelining

For pipelined resources, there exists additional parallelism provided by functional pipelining. Here optimizing an individual control step becomes inappropriate and limited. We have to consider scheduling optimization over groups of control steps. We can solve this by slicing and superimposing the distribution graph in a manner depending on the latency [78]. Again, this method can also be applied to extend our algorithm to handle the pipelined scenario.

4.3.5 Complexity Analysis

As we can see, the construction of individual schedule by the ants, or the body of the inner loop in the proposed algorithm, is of the complexity $O(n^2)$, where n is the number of nodes in the DFG under consideration. Thus the total complexity of the algorithm is determined by the number of ants m and the iteration number N. Theoretically, the production of m and N shall be proportional to the production of n and the deadline D. In this case, we have a total complexity of $O(Dn^3)$ which is the same as the un-optimized version of FDS. However, in practice, we found it is possible to fix m and N for a large range of applications (see Section 4.6). This means that in practical use the algorithm can be expected to work with $O(n^2)$ complexity for most of the cases.

4.4 ACO for Resource Constrained Scheduling

In this section, we present our algorithm of applying Ant System heuristic, or more specifically the MAX-MIN Ant System (MMAS) [92], for solving the operation scheduling problem under resource constraints.

4.4.1 List Scheduling

List scheduling is a commonly used heuristic for solving a variety of scheduling problems. It is a generalization of the ASAP algorithm with the inclusion of resource constraints [57]. A list scheduler takes a data flow graph and a priority list of all the

nodes in the DFG as input. The list is sorted with decreasing magnitude of priority assigned to each of the operation. The list scheduler maintains a ready list, i.e. nodes whose predecessors have already been scheduled. In each iteration, the scheduler scans the priority list and operations with higher priority are scheduled first. Scheduling an operator to a control step makes its successor operations ready, which will be added to the ready list. This process is carried until all of the operations have been scheduled. When there exist more than one ready nodes sharing the same priority, ties are broken randomly. A pseudo code implementation of list scheduling is shown in Algorithm 4.

```
  procedure ListScheduling(G,R, L)
  input: DFG G(V,E), resource set R, priority list L
  output: instruction schedule
 1: cycle ← 0
 2: ReadyList ← successors of start
 3: while node end is not scheduled do
 4:    for op ∈ ReadyList in descending priority order do
 5:       if a resource exists for op to start then
 6:          schedule op at time cycle
 7:       end if
 8:       update ReadyList
 9:    end for
10:    cycle ← cycle + 1
11: end while
12: return cycle
```

Algorithm 4: Resource-Constrained List Scheduling

It is easy to see that list scheduler always generates feasible schedule. Furthermore, it has been shown that a list scheduler is always capable of producing the optimal

schedule for resource-constrained instruction scheduling problem if we enumerate the topological permutations of the DFG nodes with the input priority list [57].

The success of the list scheduler is highly dependent on the priority function [93, 72] and the structure of the input application (DFG) [57]. One simple, commonly used priority function assigns the priority inversely proportional to the mobility, i.e., the greater the mobility the smaller the priority and vice-versa. This would ensure that operations with large mobility are deferred to later control steps because the number of control steps into which they could go is greater. Many other priority functions have been proposed [2, 57, 41, 8]. It is commonly agreed that there is no single good heuristic for prioritizing the DFG nodes across a range of applications. Our results in Section 4.6 confirm this.

4.4.2 Algorithm Formulation

Based on this observation, we address the RCS problem in a similar manner to the ACO meta heuristic framework used to solve the TCS problem. The key idea is to combine ACO meta-heuristic with the traditional list scheduling algorithm, and formulate the problem as an iterative searching process over the operation list space. Our proposed algorithm dynamically explores different priority functions based on the structure of the input application. This allows us to adaptively create a priority function that is suited to the application at hand.

Similar to the algorithm formulated for the TCS problem, each operation, or DFG

node op_i, is associated with a set of pheromone trails τ_{ij}. The difference is that now each trail indicates the global favorableness of assigning the i-th operation at the j-th position in the priority list, where $j = 1, \ldots, n$. Since it is valid for the operation to be assigned to any of the position in the priority list, each pheromone trail will be valid. This is different from the timing-constrained formulation where some trails are fixed to be zero based on the allowed time frames of the operations. Initially, τ_{ij} is set with some fixed value τ_0.

A pseudo code implementation of our RCS algorithm using MMAS is shown as Algorithm 5, where the pheromone bounding step is indicated as step 12. For each iteration, m ants are released and each starts to construct an individual priority list by filling the list with one operation per step. Every ant will have memory about the operations it has already selected in order to guarantee the validity of the constructed list. Upon starting step j, the ant has already selected $j-1$ operations of the DFG. To fill the j-th position of the list, the ant chooses the next operation op_i probabilistically according to:

$$p_{ij} = \begin{cases} \frac{\tau_{ij}(t)^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_k (\tau_{kj}^{\alpha}(t) \cdot \eta_{kj}^{\beta})} & \text{if } op_k \text{ is not scheduled yet} \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

where the eligible operations op_k are those yet to be scheduled. Again, η_{ik} is a local heuristic for selecting operation op_k, and α and β are parameters to control the relative influence of the distributed global heuristic τ_{ik} and local heuristic η_{ik}.

The local heuristic η gives the local favorableness of scheduling the i-th operation at the j-th position of the priority list. In our work, we experimented with different

well-known heuristics [72] proposed for operation scheduling.

1. *Operation Mobility* (OM): The mobility of an operation gives the range for scheduling the operation. It is computed as the difference between ALAP and ASAP results. The smaller the mobility, the more urgent the scheduling of the operation is. When the mobility is zero, the operation is on the critical path.

2. *Operation Depth* (OD): Operation depth is the length of the longest path in the DFG from the operation to the sink. It is an obvious measure for the priority of an operation as it gives number of operations we must pass.

3. *Latency Weighted Operation Depth* (LWOD): LWOD is computed in a similar manner as OD, except that the nodes along the path are weighted using their operation latencies.

4. *Successor Number* (SN): The motivation of using the number of successors is the hope that scheduling a node with more successors has a higher possibility of making other nodes in the DFG free, thus increasing the number of possible operations to choose from later on.

The second stage of the algorithm, i.e. the result quality assessment and pheromone trail updating, proceeds similarly as the timing constrained algorithm discussed previously. The only exception is that now the quality Q_h in Equation 4.8 is replaced by the total latency L_h of the generated scheduling result.

procedure MaxMinAntSchedulingRCS(*G*,*R*)
input: DFG $G(V,E)$, resource set *R*
output: operation schedule

1: initialize parameter $\rho, \tau_{ij}, p_{best}, \tau_{max}, \tau_{min}$
2: construct *m* ants
3: $BestSolution \leftarrow \phi$
4: **while** ending condition is not met **do**
5: **for** $i = 0$ **to** *m* **do**
6: $ant(i)$ constructs a list $L(i)$ of nodes using τ and η
7: $Q_i = \text{ListScheduling}(G,R,L(i))$
8: **if** Q_i is better than that of $BestSolution$ **then**
9: $BestSolution \leftarrow L(i)$
10: **end if**
11: **end for**
12: update τ_{max} and τ_{min} based on Equation (2.3) and (2.4)
13: **update** η **if needed**
14: update τ_{ij} based on (4.7)
15: **end while**
16: return $BestSolution$

Algorithm 5: MMAS for Resource-Constrained Scheduling

4.4.3 Refinements

Dynamic Local Heuristics

One important difference between our algorithm and other Ant System algorithms is that we use a dynamic local heuristic in the resource constrained scheduling process. It is indicated by step 13 in Algorithm 5. This technique allows better local guidance to the ants for making the selection in the next iteration. We will illustrate this feature with the use of the operation mobility heuristic.

Typically, the mobility of an operation is computed by using ALAP and ASAP results. One important input parameter in computing the ALAP result is the estimated scheduling deadline. This deadline is usually obtained from system specifications or other quick heuristic methods such as a list scheduler. It is clear that more accurate deadline estimation will yield tighter mobility range thus better local guidance.

Based on the above observation, we use dynamically computed mobility as the local heuristic in our algorithm. As the algorithm proceeds, whenever a better schedule is achieved, we use the newly obtained scheduling length as the deadline for computing the ALAP result for the next iteration. That is, for iteration t, the local heuristic for operation i is computed as (see section 2.5 for definitions for f and S^{gb}):

$$\eta_i(t) = \frac{1}{ALAP(f(S^{gb}(t-1)), i) - ASAP(i) + 1} \tag{4.12}$$

Topologically Sorted Lists

In the above algorithm, the ants construct a priority list using the same traversing method that is used in the TSP formulation [28]. In fact, this turns out to be a naïve way. To illustrate this, one just need to notice that it will yield a search space of totally $n!$ possible lists, which is simply all the permutations of n operations. However, we know that the resultant schedules of the list scheduler are only a small portion of these lists. More precisely, they are all the possible permutations of the operations that are topologically sorted based on the dependency constraints imposed by the DFG. By leveraging this application dependent feature, it is possible for us to greatly reduce the search space. For instance, using this technique on a simple 11 node example [72] reduces the possible number of orderings from 11! to 59400, or 0.15%. Though it quickly becomes prohibitive to precisely compute such reduction for more complex graphs[1], it is generally significant. By adopting this technique, in the final version of our algorithm, the ant traverses the DFG in a similar manner to the list scheduling process and fills operation list one by one. At each step, the ant will select an operation based on Equation 4.11 but only from all the ready operations, that is, from all the operations whose predecessors have all been scheduled.

[1] We tried to compute the search space reduction for Figure 4.5 using GenLE [81]. It failed to produce any result within 100 computer hours.

4.4.4 Extensions

So far, our discussion on the operation scheduling problems has been limited to the the *homogeneous* case. In other words, each operation is mapped to a unique resource type, though a resource type might be able to handle different operations. In practice, this means that a *resource allocation* step needs to precede the operation scheduling process. We often need to handle the *heterogeneous* case, where one operation can be executed with different resource types. For example, a system might have two different realizations of multiplier, one is faster but more expensive while the other is slower but cheaper. Both are capable of executing a multiplication operation. Our challenge is to determine how to effectively use the resources to achieve the best time performance. In this situation, separating the resource allocation step from operation scheduling may be not a favorable approach, as the prior step could greatly limit the optimization opportunity for operation scheduling. This motivates us to consider the resource allocation issue within the operation scheduling problem.

It is possible to address this problem using ILP by extending the ILP formulation for the homogenous case. The basic idea is to introduce a new set of parameters m_{ik} which can take value 0 or 1, and describe the compatibility between operation op_i and resource type k. A set of new constraints are needed to make sure that only one type of resources among all those that are capable of processing op_i is used, i.e.

$$\sum_k m_{ik} = 1 \qquad \text{where } i = 1, \ldots, n \tag{4.13}$$

We can see it makes the ILP problem even more intractable.

However, this extra difficulty does not block the list scheduler or the proposed MMAS approach from working. The basic algorithm could be carried out with almost no changes except for the list construction. The major problem is, when there exists alternative resource types for one specific operation, estimating a certain attribute of the operation becomes more challenging. For example, with different execution delay on capable resource types, the mobility of the operation is variable. This has been studied in previous research, e.g. [94], where the average latency over a set of heterogenous resources is used to carry the scheduling task. In our work, we simply take the pessimistic approach by applying the longest execution latency amongst the alternative resources in computing such attributes. With this extension, our algorithm can be applied to heterogenous cases.

4.4.5 Complexity Analysis

List scheduling is a two step process. In the first step, a priority list is built. The second step takes n steps to solve the scheduling problem since it is a constructive method without backtracking. For different heuristics, the complexity of the first step is different. When operation mobility, operation depth and latency weight operation depth are used, it takes $O(n^2)$ steps to build the priority list since a depth-first or breadth-first graph transversal is involved. When the successor node number is adopted as the list construction heuristic, it only takes n steps. Thus the complexities for these methods are $O(n^2)$ or $O(n)$ respectively.

The force-directed resource constrained scheduling method is different. Though it is also a constructive method without backtracking, we need to compute the force of each operation at every step since the total latency is dynamically increased based on whether there is enough resources to handle the ready operations. Thus the FDS method has $O(n^3)$ complexity.

The complexity of the proposed MMAS solution is determined mainly by the complexity of constructing individual scheduling solutions, the number of ants m and the total iteration N in every run. In order to generate a schedule solution, each ant needs to first loop through n operations and for each operation determine its location, which has a complexity of $O(n)$. This list is then provided to a list scheduler with a complexity of $O(n)$ or $O(n^2)$. This makes overall complexity $O(n^2)$. Obviously, if mN is proportional to n, we will have one order higher complexity than the corresponding list scheduling approach. However, based on our experience, it is possible to fix such factor for a large set of practical cases so that the complexity of the MMAS solution is the same as the list scheduling approach.

4.5 ExpressDFG Benchmarks

In order to test and evaluate our algorithms, we have constructed a comprehensive set of benchmarks named as*ExpressDFG*. These benchmarks are taken from one of two sources:

- Popular benchmarks used in previous literature;

- Real-life examples generated and selected from the MediaBench suite [59].

The benefit of having classic samples is that they provide a direct comparison between results generated by our algorithm and that from previously published methods. This is especially helpful when some of the benchmarks have known optimal solutions. In our final testing benchmark set, seven samples widely used in operation scheduling studies are included. These samples focus mainly on frequently used numeric calculations performed by different applications. They are:

1. ARF: an implementation of an *Auto Regression Filter*.
2. EWF: an implementation of an *Elliptic Wave Filter*.
3. FIR1 and FIR2: two versions of a *Finite Impulse Response Filter)*.
4. COSINE1 and COSINE2: two implementations for a one dimensional 8-point fast discrete cosine transform, where COSINE1 assumes constant coefficients while the coefficients in COSINE2 are given as inputs.
5. HAL: an iterative solution of a second order differential equation. This perhaps is the most popularly used example in text books which originally appeared in [78].

However, these samples are typically small to medium in size, and are considered somewhat old. To be representative, it is necessary to create a more comprehensive set with benchmarks of different sizes and complexities. Such benchmarks shall aim to:

- Provide real-life testing cases from real-life applications;

- Provide more up-to-date testing cases from modern applications;
- Provide challenging samples for operation scheduling algorithms with regards to larger number of operations, higher level of parallelism and data dependency;
- Provide a wide range of synthesis problems to test the algorithms' scalability;

For this purpose, we have investigated the MediaBench suite, which contains a wide range of complete applications for image processing, communications and DSP applications. We analyzed these applications using the SUIF [4] and Machine SUIF [89] tools, and over 14,000 DFGs were extracted as preliminary candidates for our benchmark set. After careful study, thirteen DFG samples were selected from four MediaBench applications. These applications are:

JPEG JPEG is a lossy compression technique for digital images.The *cjpeg* application performs compression, while the *djpeg* application decompresses the JPEG image.

MPEG2 MPEG2 is a digital video compression standard, commonly used for high quality video compression including DVD compression. The *mpeg2enc* application encodes the video, while the *mpeg2dec* application decodes the video.

EPIC EPIC stands for Efficient Pyramid Image Coder and is another image compression utility.

MESA The Mesa project is a software 3-D graphics package. The primary application that we were concerned with was the *texgen* utility, which generates a texture mapped version of the Utah teapot.

From the JPEG project, four basic blocks were selected. The first came from the *write_bmp_header* function. The basic block was selected for its high level of parallelism. The second basic block came from the *h2v2_smooth_downsample* function. This function has 51 nodes for only one store operation at the end. The store is dependent on all but two of the operations, making it an interesting problem for scheduling. The third basic block was selected from the *jpeg_fdct_islow* function. The function performs an integer forward discrete cosine transform (DCT) using a slow-but-accurate algorithm and was chosen for its popularity amongst DSP applications. The final block was selected from the *jpeg_idct_ifast* function. Like the forward DCT, this was selected for its commonality. However, this implementation is a fast, and much less accurate, version of the inverse DCT.

Two basic blocks were selected from the MPEG section. The first came from the *idctcol* function in the *mpeg2dec* application. The function implements another version of the inverse DCT algorithm. In this case, the function is part of a 2-D inverse DCT, while the inverse DCT from the JPEG application is only 1-D. The large size of the DFG and complicated dependency structure provide a good test for scheduling algorithm. The second comes from the *motion_vectors* function in the *mpeg2enc* function. The basic block only contains 42 nodes and 38 edges, making it one of the smaller blocks selected from MediaBench, ensuring that the benchmark suite provides a wide range of synthesis problems to test scalability.

The EPIC project supplied one basic block. It came from the *collapse_pyr* function, which is a quadrature mirror filter bank. The block was selected for its medium size and common use in DSP applications.

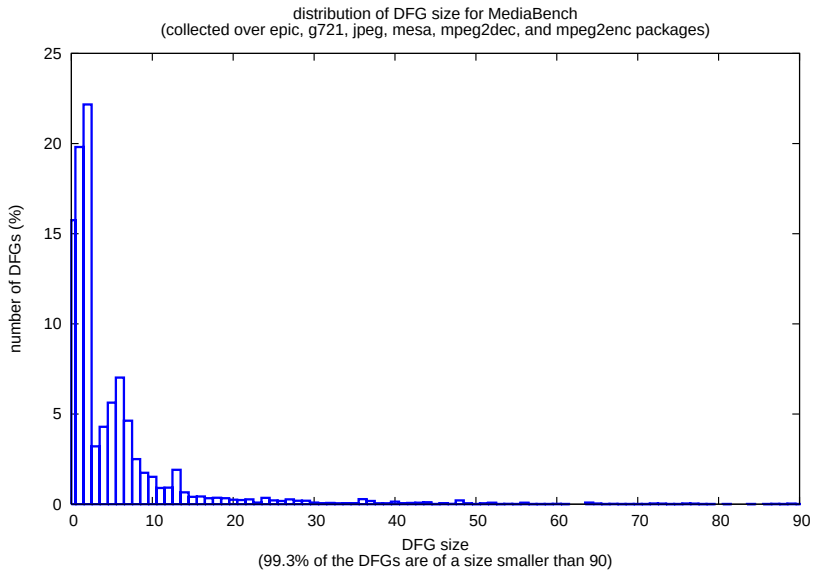


Figure 4.3: Distribution of DFG size for MediaBench

From the MESA application, six basic blocks were selected to be added to the benchmark suite. The *invert_matrix_general* and *matmul* functions were selected because they are general functions, not specific to the MESA application. Matrix operations, such as inversion and multiplication, are common in DSP applications where many filters are merely matrix multiplications with a set of coefficients. The next block selected came from the *smooth_color_z_triangle* function. The basic block is essentially four parallel computations without data dependencies, making it an ideal addition to the benchmark suite. The fourth benchmark is from the *horner_bezier* method. With only

18 nodes, the small size helps add variety to the benchmarks. The fifth block comes from the *interpolate_aux* function. The function performs four linear interpolation calculations, which can easily be run in parallel if the hardware is available. The final benchmark is from the *feedback_points* function, which calculates texture coordinates for a feedback buffer.

In order to justify the difficulty and representativeness of our testing cases, we analyze the distribution of the sizes of DFGs in practical software programs. Our analysis covers the *epic*, *jpeg*, *g721*, *mpeg2enc*, *mpeg2dec*, and *mesa* packages. The result is shown in Figure 4.3. We find that the maximum size of a DFG can be as big as 632. However, the majority of the DFGs are much smaller. In fact, more than 99.3% DFGs have fewer than 90 nodes. Moreover, the very largest ones are of little interest with respect to system performance. They are typically related with system initialization and are executed only once.

Table 4.1 lists all twenty benchmarks that were included in our final benchmark set. Together with the names of the various functions where the basic blocks originated are the number of nodes, number of edges and operation depth (assuming unit delay for every operation) of the DFG. The data, including related statistics, DFG graphs and source code for the all testing benchmarks, is available online [33].

Benchmark Name	*# Nodes*	*# Edges*	*OD*
HAL	11	8	4
horner_bezier	18	16	8
ARF	28	30	8
motion_vectors	32	29	6
EWF	34	47	14
FIR2	40	39	11
FIR1	44	43	11
h2v2_smooth_downsample	51	52	16
feedback_points	53	50	7
collapse_pyr	56	73	7
COSINE1	66	76	8
COSINE2	82	91	8
write_bmp_header	106	88	7
interpolate_aux	108	104	8
matmul	109	116	9
idctcol	114	164	16
jpeg_idct_ifast	122	162	14
jpeg_fdct_islow	134	169	13
smooth_color_z_triangle	197	196	11
invert_matrix_general	333	354	11

Table 4.1: ExpressDFG benchmark suite
(Benchmarks with † are extracted from MediaBench.)
(Benchmark node and edge count with the operation depth (OD) assuming unit delay.)

4.6 Experimental Results

4.6.1 Time Constrained Scheduling

In order to evaluate the quality of our proposed algorithm for timing constrained scheduling problem, we compare its results with that obtained by the widely used force-directed scheduling method. For all testing benchmarks, operations are allocated on two types of computing resources, namely MUL and ALU, where MUL is capable of handling multiplication and division, and ALU is used for other operations such as addition and subtraction. Furthermore, we define the operations running on MUL to take two clock cycles and the ALU operations take one. This definitely is a simplified case from reality. However, it is a close enough approximation and does not change the generality of the results. Other choices can easily be implemented within our framework.

Since there is no widely distributed and recognized FDS implementation, we implemented our own. The implementation is based on [78] and has all the applicable refinements proposed in the paper, including multi-cycle operation support, resource preference control, and look-ahead using second order of displacement in force computation. Actually, based on our experience, the look-ahead function for FDS is very critical. Without invoking this mechanism, basic FDS provides poor scheduling results even for small sized examples. In Table 4.2, we show the effect of look-ahead for the HAL benchmark originally presented in [78], which has only 11 operations and 8 data dependencies. Because of this, in our experiments, the look-ahead function is always

used to allow FDS to provide better results.

deadline	w/t look-ahead	w/ look-ahead
9	(2 1)	(2 1)
10	(2 1)	(2 1)
11	(2 1)	(2 1)
12	**(3 1)**	**(2 1)**
13	**(3 1)**	**(1 1)**
14	**(3 1)**	**(1 1)**

Table 4.2: Effect of Look-ahead Mechanism in FDS
(Result shown in MUL/ALU number pair. Deadline is in cycles.)

With the assigned resource/operation mapping, ASAP is first performed to find the critical path delay L_c. We then set our predefined deadline range to be $[L_c, 2L_c]$, i.e. from the critical path delay to 2 times of this delay. This results 263 testing cases in total. For each delay, we run FDS first to obtain its scheduling result. Following this, the proposed MMAS algorithm is executed 5 times to obtain enough data for performance evaluation. We report the FDS result quality, the average and best result quality for the proposed algorithm and the standard deviation for these results. The execution time information for both algorithms is also reported.

We have implemented our MMAS formulation in C for the TCS problem, with the refinements discussed in Section 4.3. The evaporation rate ρ is configured to be 0.98. The scaling parameters for global and local heuristics are set to be $\alpha = \beta = 1$ and delivery rate $Q = 1$. These parameters are not changed over the tests. We also experimented with different ant number m and the allowed iteration count N. For example, set m to

be proportional to the average branching factor of the DFG under study and N to be proportional to the total operation number. However, it is found that there seems to exist a fixed value pair for m and N which works well across the wide range of testing samples in our benchmark. In our final settings, we set m to be 10, and N to be 150 for all the timing constrained scheduling experiments.

Due to the large amount of data, we won't be able to report testing results for all 263 cases in details. Table 4.3 compares the testing results for *idctcol* and *invert_matrix_general*, two of the biggest samples. In this table, we provide a side by side comparison between FDS and our proposed method. The scheduling results are reported as MUL/ALU number pair required by the obtained scheduling. For MMAS method, we report both the average performance and the best performance in the 5 runs for each testing case, together with the saving percentage. The saving is measured by the reduction of computing resources. In order to keep the evaluation general and objective, we use the total count of resources as the quality metrics without considering their individual cost factors.

Besides absolute quality of the results, one difference between FDS and the proposed method is that our method is relatively more stable. In our experiments, it is observed that the FDS approach can provide worse quality results as the deadline is relaxed. Using the *idctcol* in Table 4.3 as an example, FDS provides drastically worse results for deadlines ranging from 25 to 30 though it is able to reach decent scheduling qualities for deadline from 19 to 24. The same problem occurs for deadlines between

Name (size)	Deadline	FDS	Average	Savings	Best	Savings	σ
	19	(6 8)	(5.0 6.0)	21.43%	(5 6)	21.43%	0.000
	20	(5 7)	(4.4 6.0)	13.33%	(4 6)	16.67%	0.219
	21	(4 7)	(4.2 5.8)	9.09%	(4 6)	9.09%	0.000
	22	(4 7)	(4.2 5.4)	12.73%	(4 5)	18.18%	0.219
	23	(4 7)	(4.0 5.4)	14.55%	(4 5)	18.18%	0.219
	24	(4 7)	(3.6 5.2)	20.00%	(3 5)	27.27%	0.335
	25	**(8 8)**	(3.8 5.0)	45.00%	(3 5)	50.00%	0.179
	26	**(8 8)**	(3.4 5.0)	47.50%	(3 5)	50.00%	0.219
	27	**(8 8)**	(3.0 5.0)	50.00%	(3 5)	50.00%	0.00
	28	**(8 8)**	(3.0 4.6)	52.50%	(3 4)	56.25%	0.219
idctcol	29	**(8 8)**	(3.0 4.4)	53.75%	(3 4)	56.25%	0.219
(114 164)	30	**(8 8)**	(3.0 4.6)	52.50%	(3 4)	56.25%	0.219
	31	(4 6)	(3.0 4.6)	24.00%	(3 4)	30.00%	0.219
	32	(4 5)	(3.0 4.0)	22.22%	(3 4)	22.22%	0.000
	33	(4 5)	(2.8 4.0)	24.44%	(2 4)	33.33%	0.179
	34	(4 5)	(3.0 4.0)	22.22%	(3 4)	22.22%	0.000
	35	(4 5)	(3.0 4.0)	22.22%	(3 4)	22.22%	0.000
	36	**(4 6)**	(3.0 3.8)	32.00%	(3 3)	40.00%	0.179
	37	**(4 6)**	(2.6 3.8)	36.00%	(3 3)	40.00%	0.219
	38	**(4 6)**	(2.8 3.4)	38.00%	(3 3)	40.00%	0.179
	15	(24 23)	(26.0 22.0)	***-2.13%***	(25 22)	0.00%	0.283
	16	(22 19)	(23.8 19.0)	***-4.39%***	(23 19)	***-2.44%***	0.179
	17	(19 17)	(21.8 17.4)	***-8.89%***	(21 17)	***-5.56%***	0.335
	18	(18 16)	(20.4 16.2)	***-7.65%***	(20 16)	***-5.88%***	0.219
	19	(17 16)	(19.2 16.0)	***-6.67%***	(19 15)	***-3.03%***	0.335
	20	(17 16)	(18.2 13.4)	4.24%	(18 13)	6.06%	0.358
invert_matrix_general	21	(16 16)	(17.2 12.8)	6.25%	(17 13)	6.25%	0.000
(333 354)	22	(16 16)	(16.4 12.2)	10.63%	(16 12)	12.50%	0.358
	23	(16 16)	(16.0 11.8)	13.12%	(16 11)	15.62%	0.179
	24	(16 16)	(15.4 11.2)	16.87%	(15 11)	18.75%	0.219
	25	(16 16)	(14.4 10.8)	21.25%	(14 11)	21.88%	0.179
	26	(16 16)	(14.2 10.2)	23.75%	(13 10)	28.12%	0.358
	27	(16 16)	(13.8 10.0)	25.62%	(13 10)	28.12%	0.179
	28	(16 16)	(13.4 10.2)	26.25%	(13 10)	28.12%	0.219
	29	(16 16)	(13.0 9.4)	30.00%	(13 9)	31.25%	0.219
	30	(16 16)	(12.6 9.6)	30.63%	(13 9)	31.25%	0.179

Table 4.3: Partial detailed results for Timing-Constrained Scheduling

(Size is given as DFG's node/edge number pair. Virtual nodes and edges are not counted. Average and standard deviation σ are computed over 5 runs. Saving is computed based on FDS results. No weight applied.)

Name	Size	Deadline	Avg. Savings (SA)	Best Savings (SA)	Avg. σ (SA)
HAL	11/8	(6 - 12)	7.1% (7.1%)	7.1% (7.1%)	0.000 (0.000)
horner_bezier_surf	18/16	(11 - 22)	9.9% (-4.6%)	13.2% (2.1%)	0.015 (0.051)
ARF	28/30	(11 - 22)	12.4% (-1.2%)	16.9% (3.1%)	0.093 (0.099)
motion_vectors	32/29	(7 - 14)	13.1% (-3.4%)	16.0% (2.8%)	0.072 (0.177)
EWF	34/47	(17 - 34)	11.5% (-4.4%)	18.1% (4.7%)	0.081 (0.136)
FIR2	40/39	(12 - 24)	16.8% (-15.7%)	22.8% (-1.9%)	0.106 (0.299)
FIR1	44/43	(12 - 24)	15.2% (-7.7%)	18.0% (-3.3%)	0.047 (0.116)
h2v2_smooth_downsample	51/52	(17 - 34)	19.3% (7.6%)	21.3% (11.0%)	0.042 (0.088)
feedback_points	53/50	(11 - 22)	5.9% (-12.8%)	9.2% (-6.4%)	0.103 (0.196)
collapse_pyr	56/73	(8 - 16)	18.3% (4.6%)	18.9% (9.6%)	0.044 (0.195)
COSINE1	66/76	(10 - 20)	21.5% (7.4%)	25.9% (14.1%)	0.150 (0.349)
COSINE2	82/91	(10 - 20)	5.6% (-14.8%)	12.0% (-7.3%)	0.232 (0.342)
write_bmp_header	106/88	(8 - 16)	0.9% (-5.3%)	1.0% (-3.4%)	0.064 (0.093)
interpolate_aux	108/104	(10 - 20)	0.2% (-36.5%)	2.0% (-27.9%)	0.109 (0.407)
matmul	109/116	(11 - 22)	3.7% (-30.8%)	5.1% (-21.4%)	0.088 (0.363)
idctcol	114/164	(19 - 38)	30.7% (12.6%)	34.0% (17.5%)	0.151 (0.231)
jpeg_idct_ifast	122/162	(17 - 34)	50.3% (36.9%)	52.1% (41.8%)	0.147 (0.336)
jpeg_fdct_islow	134/169	(16 - 32)	31.4% (7.5%)	34.2% (13.0%)	0.171 (0.335)
smooth_color_z_triangle	197/196	(15 - 30)	7.3% (-18.7%)	9.2% (-12.0%)	0.136 (0.472)
invert_matrix_general	333/354	(15 - 30)	11.2% (-29.4%)	13.2% (-22.9%)	0.237 (0.743)
Total Avg.			16.4% (-5.1%)	19.5% (1.0%)	0.104 (0.251)

Table 4.4: Result Summary for Timing-Constrained Scheduling

Data in parenthesis shows the results obtained using Simulated Annealing.

Deadline shows the tested range. Average σ is computed over the tested range.

Saving is computed based on FDS results. No weight applied.

36 and 38. One possible reason is that as the deadline is extended, the time frame of each operation is also extended, which makes the force computation more likely to clash with similar values. Due to the lack of backtracking and good look-ahead capability, an early mistake would lead to inferior results. On the other hand, our proposed algorithm robustly generates monotonically non-increasing results with fewer resource requirements as the deadline increases.

Table 4.4 summarizes the testing results for all of the benchmarks. We present the average and the best results for each testing benchmark, its tested deadline range, and the average standard deviations. The table is arranged in the increasing order of the complexity of the DFGs. The average result quality generated by our algorithm is better than or equal to the FDS results in 258 out of 263 cases. Among them, for 192 testing cases (or 73% of the cases) our MMAS method outperforms the FDS method. There are only five cases where our approach has worse average quality results. They all happened on the *invert_matrix_general* benchmark and are listed in Table 4.3, indicated by lines with the italic bold fonts. On average, as shown in Table 4.4, we can expect a 16.4% performance improvement over FDS. If only considering the best results among the 5 runs for each testing case, we achieve a 19.5% resource reduction averaged over all tested samples. The most outstanding results provided by our proposed method achieve a 75% resource reduction compared with FDS. These results are obtained on a few deadlines for the *jpeg_idct_ifast* benchmark.

From Table 4.4, it is easy to see that for all the examples, MMAS based operation

scheduling achieves better or much better results. Our approach seems to have much stronger capability in robustly finding better results for different testing cases. Furthermore, it scales very well over different DFG sizes and complexities. Another aspect of scalability is the pre-defined deadline. Based on the results presented in Table 4.3 and Table 4.4, the proposed algorithm also demonstrates better scalability over this parameter.

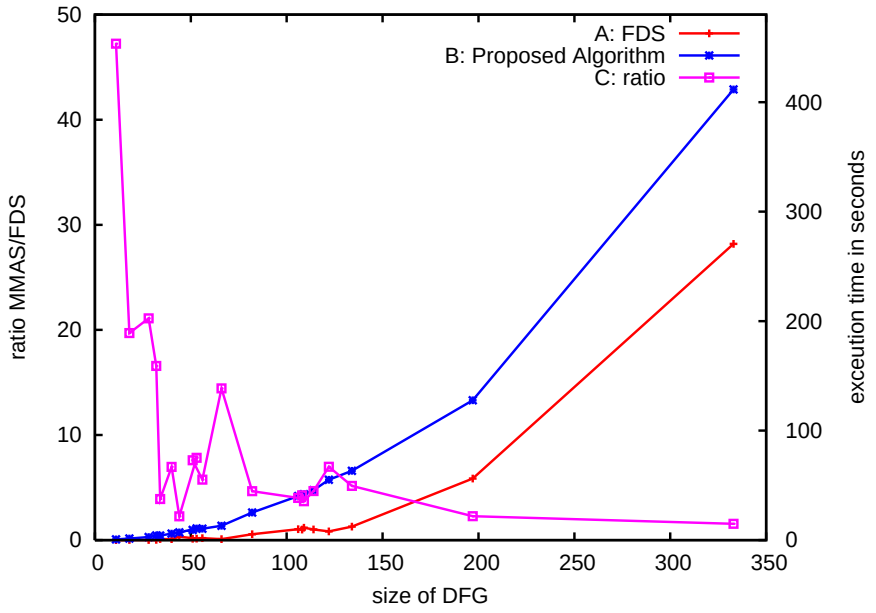


Figure 4.4: Execution Time for Timing-Constrained Scheduling.

(Ratio is MMAS time / FDS time)

All of the experimental results are obtained on a Linux box with a 2GHz CPU. Figure 4.4 diagrams the execution time comparison between the presented algorithm and FDS. Curve A and B shows the run time for FDS and the proposed method (respectively), where we use the average runtime for our MMAS solutions over 5 runs.

As discussed before, since we use fixed ant number m and iteration limit N in our experiments to make the algorithm simpler, there exists a big gap between the execution times for the smaller sized cases. For example, for the HAL example, which only has 11 operations, the execution time of FDS is 0.014 seconds while our method takes 0.66 seconds. This translates into a ratio of 47. However, as the size of the problem gets bigger, this ratio drops quickly. For the biggest cases *invert_matrix_genera*, FDS takes 270.6 seconds while our method spends about 411.7 seconds, which makes the ratio 1.5. To summarize, for smaller cases, our algorithm does have relatively larger execution times but the absolute run time is still very short. For the HAL example, it only takes a fraction of a second. For bigger cases, the proposed method has a runtime at the same scale as FDS. This makes our algorithm practical.

In Figure 4.4, we do see some spikes in the ratio curve. We attribute this to two main reasons. First, the recorded execution time is based on system time and it is relatively more unreliable when the execution time is small. Secondly but perhaps more important, the timing performance of both algorithms is not only determined by the DFG node count but also dependent on the predefined dependencies in the DFGs and the deadline D. This will introduce variance when the curves are drawn against the node count.

4.6.2 Resource Constrained Scheduling

We have implemented the proposed MMAS-based resource-constrained scheduling algorithm and compared its performance with the popularly used list scheduling and force-directed scheduling algorithms.

For each of the benchmark samples, we run the proposed algorithm with different choices of local heuristics. For each choice, we also perform 5 runs to obtain enough statistics for evaluating the stability of the algorithm. Again we fixed the number of ants per iteration 10 and in each run we allow 100 iterations. Other parameters are also the same as those used in the timing constrained problem. The best schedule latency is reported at the end of each run and then the average value is reported as the performance for the corresponding setting. Two different experiments are conducted for resource constrained scheduling – the homogenous case and the heterogenous case.

For the homogenous case, resource allocation is performed before the operation scheduling. Each operation is mapped to a unique resource type. In other words, there is no ambiguity on which resource the operation shall be handled during the scheduling step. In this experiment, similar to the timing constrained case, two types of resources (MUL/ALU) are allowed. The number of each resource type is predefined after making sure they do not make the experiment trivial (for example, if we are too generous, then the problem simplifies to an ASAP problem).

Name	Size	Resources	FDS	List Scheduling				MMAS(average over 5 runs)				SA
				OM	OD	LWOD	SN	OM	OD	LWOD	SN	(avg. 10 runs)
HAL	(8/11)	(2 1)	**8**	10	**8**	**8**	**8**	**8.0**	**8.0**	**8.0**	**8.0**	**8.0**
horner_bezier_surf	(16/18)	(2 1)	**12**	16	**12**	13	13	**12.0**	**12.0**	**12.0**	**12.0**	12.4
ARF	(30/28)	(3 1)	18	19	**16**	18	18	**16.0**	**16.0**	**16.0**	**16.0**	17.2
motion_vectors	(29/32)	(3 4)	**12**	15	**12**	**12**	14	**12.0**	**12.0**	**12.0**	**12.0**	13.3
EWF	(47/34)	(1 2)	**21**	22	**21**	**21**	22	**21.0**	**21.0**	**21.0**	**21.0**	21.3
FIR2	(39/40)	(2 3)	17	19	18	17	15	17.0	**16.8**	17.0	17.0	18.5
FIR1	(43/44)	(2 3)	**16**	22	22	21	**16**	**16.0**	**16.0**	**16.0**	**16.0**	21.1
h2v2_smooth_downsample	(52/51)	(1 3)	23	28	23	23	**22**	22.4	22.8	22.8	22.8	23.6
feedback_points	(50/53)	(3 3)	16	20	**14**	19	**14**	14.4	14.2	14.6	14.6	16.6
collapse_pyr	(73/56)	(3 5)	**11**	12	**11**	**11**	**11**	**11.0**	**11.0**	**11.0**	**11.0**	11.3
COSINE1	(76/66)	(4 5)	16	18	16	17	16	**14.0**	**14.0**	**14.0**	**14.0**	15.2
COSINE2	(91/82)	(5 8)	14	18	14	17	13	**12.4**	**12.4**	12.6	12.8	14.9
write_bmp_header	(88/106)	(1 9)	**12**	17	**12**	**12**	**12**	12.8	12.6	12.8	12.4	13.4
interpolate_aux	(104/108)	(9 8)	13	16	12	16	16	**11.0**	11.8	**11.0**	11.8	15.6
matmul	(116/109)	(9 8)	15	14	**13**	14	14	13.6	13.8	13.8	13.8	14.7
idctcol	(164/114)	(5 6)	21	26	21	21	21	20.6	**19.8**	20.2	20.0	24.3
jpeg_idct_ifast	(162/122)	(10 9)	**19**	21	20	**19**	**19**	**19.0**	**19.0**	**19.0**	**19.0**	20.8
jpeg_fdct_islow	(169/134)	(5 7)	**21**	28	22	22	**21**	22.0	22.0	21.8	21.8	23.8
smooth_color_z_triangle	(196/197)	(8 9)	**24**	25	25	23	**24**	**24.0**	**24.0**	**24.0**	**24.0**	25.5
invert_matrix_general	(354/333)	(15 11)	26	28	28	25	25	**24.0**	**24.2**	**24.2**	**24.2**	27.1

Table 4.5: Result Summary for Homogenous Resource-Constrained Scheduling

(Heuristic Labels: OM=Operation Mobility OD=Operation Depth, LWOD=Latency Weighted Operation Depth, SN=Successor Number)

Table 4.5 shows the testing results for the homogenous case. The best results for each case are shown in bold. Compared with a variety of list scheduling approaches and the force-directed scheduling method, the proposed algorithm generates better results consistently over all testing cases, which is demonstrated by the number of times that it provides the best results for the tested cases. This is especially true for the case when operation depth (OD) is used as the local heuristic, where we find the best results in 14 cases amongst 20 tested benchmarks. For other traditional methods, FDS generates the most hits (10 times) for best results, which is still less than the worst case of MMAS (11 times). For some of the testing samples, our method provides significant improvement on the schedule latency. The biggest saving achieved is 22%. This is obtained for the COSINE2 benchmark when operation mobility (OM) is used as the local heuristic for our algorithm and also as the heuristic for constructing the priority list for the traditional list scheduler. For cases that our algorithm fails to provide the best solution, the quality of its results is also much closer to the best than other methods.

Besides the absolute schedule latency, another important aspect of the quality of a scheduling algorithm is its stability over different input applications. As indicated in Section 4.2, the performance of traditional list scheduler heavily depends on the input application. This is echoed by the data in Table 4.5. Meantime, it is easy to observe that the proposed algorithm is much less sensitive to the choice of different local heuristics and input applications. This is evidenced by the fact that the standard deviation of the results achieved by the new algorithm is much smaller than that of the traditional list

Benchmark (nodes/edges)	Resources	CPLEX (latency/runtime)	Force Directed	List Scheduling				MMAS(average over 5 runs)			
				OM	OD	LWOD	SN	OM	OD	LWOD	SN
HAL(21/25)	1a, 1fm, 1m, 3i, 3o	8 / 32	**8**	**8**	**8**	9	**8**	**8**	**8**	**8**	**8**
ARF(28/30)	2a, 1fm, 2m	11 / 22	**11**	**11**	13	13	13	**11**	**11**	**11**	**11**
EWF(34/47)	1a, 1fm, 1m	27 / 24000	28	28	31	31	28	27.2	27.2	**27**	27.2
FIR1(40/39)	2a, 2m, 3i, 3o	13 / 232	19	19	19	19	18	17.2	17.2	**17**	17.8
FIR2(44/43)	1a, 1fm, 1m, 3i, 3o	14 / 11560	19	19	21	21	21	**16.2**	16.4	**16.2**	17
COSINE1(66/76)	2a,2m, 1fm, 3i, 3o	†	18	19	20	18	18	**17.4**	18.2	17.6	17.6
COSINE2(82/91)	2a,2m, 1fm, 3i, 3o	†	23	23	23	23	23	**21.2**	**21.2**	**21.2**	**21.2**

Table 4.6: Result Summary for Heterogenous Resource-Constrained Scheduling

Schedule latency is in cycles; Runtime is in seconds; † indicates CPLEX failed to provide final result before running out of memory.

(Resource Labels: a=alu, fm=faster multiplier, m=mutiplier, i=input, o=output)

(Heuristic Labels: OM=Operation Mobility OD=Operation Depth, LWOD=Latency Weighted Operation Depth, SN=Successor Number)

scheduler. Based on the data shown in Table 4.5, the average standard deviation for list scheduling over all the benchmarks and different heuristic choices is 1.2, while for the MMAS algorithm it is only 0.19. In other words, we can expect to achieve high quality scheduling results much more stably on different application DFGs regardless of the choice of local heuristic. This is a great attribute desired in practice.

One possible explanation for the above advantage is the different ways how the scheduling heuristics are used by list scheduler and the proposed algorithm. In list scheduling, the heuristics are used in a greedy manner to determine the order of the operations. Furthermore, the schedule of the operations is done all at once. Differently, in the proposed algorithm, local heuristics are used stochastically and combined with the pheromone values to determine the operations' order. This makes the solution exploration more balanced. Another fundamental difference is that the proposed algorithm is an iterative process. In this process, the pheromone value acts as an indirect feedback and tries to reflect the quality of a potential component based on the evaluations of historical solutions that contain this component. It introduces a way to integrate global assessments into the scheduling process, which is missing in the traditional list or force-directed scheduling.

In the second experiment, heterogeneous computing units are allowed, i.e. one type of operation can be performed by different types of resources. For example, multiplication can be performed by either a faster multiplier or a regular one. Furthermore, multiple same type units are also allowed. For example, we may have 3 faster multipli-

ers and 2 regular ones.

We conduct the heterogenous experiments with the same configuration as for the homogenous case. Moreover, to better assess the quality of our algorithm, the same heterogenous RCS tasks are also formulated as integer linear programming problems and then optimally solved using CPLEX. Since the ILP solution is time consuming to obtain, our heterogenous tests are only done for the classic samples in our benchmark set.

Table 4.6 summarizes our heterogenous experiment results. Here an extended HAL benchmark is used which includes extra memory access operations. Compared with a variety of list scheduling approaches and the force-directed scheduling method, the proposed algorithm generates better results consistently over all testing cases. The biggest saving achieved is 23%. This is obtained for the FIR2 benchmark when the latency weighted operation depth (LWOD) is used as the local heuristic. Similar to the homogenous case, our algorithm outperforms other methods in regards to consistently generating high-quality results. In Table 4.6, the average standard deviation for list scheduler over all the benchmarks and different heuristic choices is 0.8128, while that for the MMAS algorithm is only 0.1673.

Though the results of force-directed scheduler generally outperform the list scheduler, our algorithm achieves even better results. On average, comparing with the force-directed approach, our algorithm provides a 6.2% performance enhancement for the testing cases, while performance improvement for individual test sample can be as

much as 14.7%.

Finally, compared to the optimal scheduling results computed by using the integer linear programming model, the results generated by the proposed algorithm are much closer to the optimal than those provided by the list scheduling heuristics and the force-directed approach. For all the benchmarks with known optima, our algorithm improves the average schedule latency by 44% comparing with the list scheduling heuristics. For the larger size DFGs such as COSINE1 and COSINE2, CPLEX fails to generate optimal results after more than 10 hours of execution on a SPARC workstation with a 440MHz CPU and 384MByte memory. In fact, CPLEX crashes for these two cases because of running out of memory. For COSINE1, CPLEX does provide a intermediate sub-optimal solution of 18 cycles before it crashes. This result is worse than the best result found by our proposed algorithm.

Experiment results of our algorithm are obtained on a Linux box with a 2GHz CPU, as well as those for list scheduling and the force-directed scheduling. For all the benchmarks, the runtime of the proposed algorithm ranges from 0.1 to 1.76 seconds. List scheduling is always the fastest due to its one-pass nature. It typically finishes within a small fraction of a second. The force-directed scheduler runs much slower than the list scheduler because its complexity is cubic in the number of operations. For small testing cases, it is typically faster than our algorithm as we set a fixed iteration number for the ants to explore the search space. However, as the problem size grows, the force-directed scheduler has longer runtime than our algorithm. In fact, for COSINE1 and COSINE2,

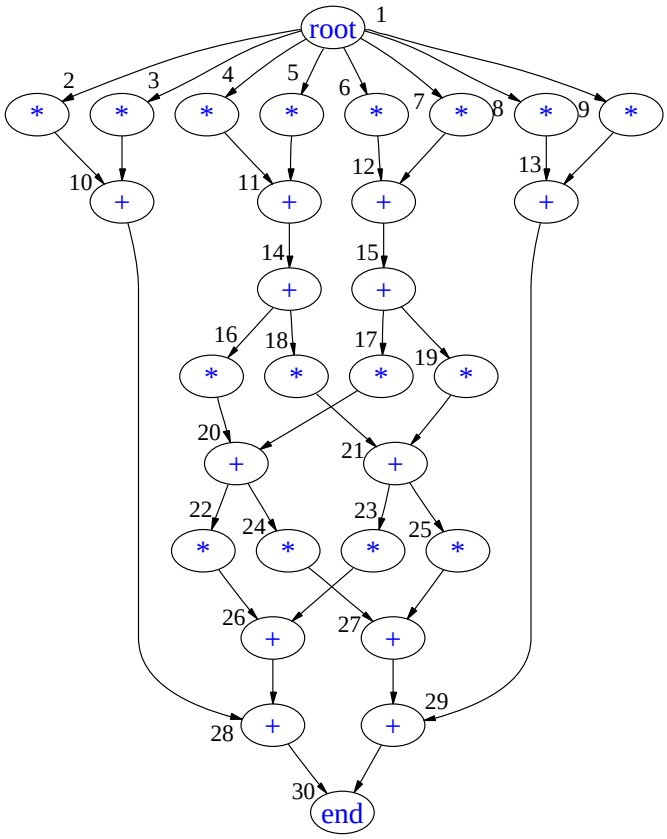


Figure 4.5: Data Flow Graph of AR Filter.
(The number by the node is the index assigned for the operation.)

the force-directed approach takes 12.7% and 21.2% more execution time respectively.

The evolutionary effect on the global heuristics τ_{ij} is illustrated in Figure 4.6. It plots the pheromone values for the ARF testing sample after 100 iterations of the proposed algorithm. The x-axis is the index of operation node in the DFG (shown in Figure 4.5), and the y-axis is the order index in the priority list passed to the list scheduler. There exist totally 30 nodes with node 1 and node 30 as the dummy source and sink of the DFG. Each dot in the diagram indicates the strength of the resultant pheromone trails for assigning corresponding order to a certain operation – the bigger the size of

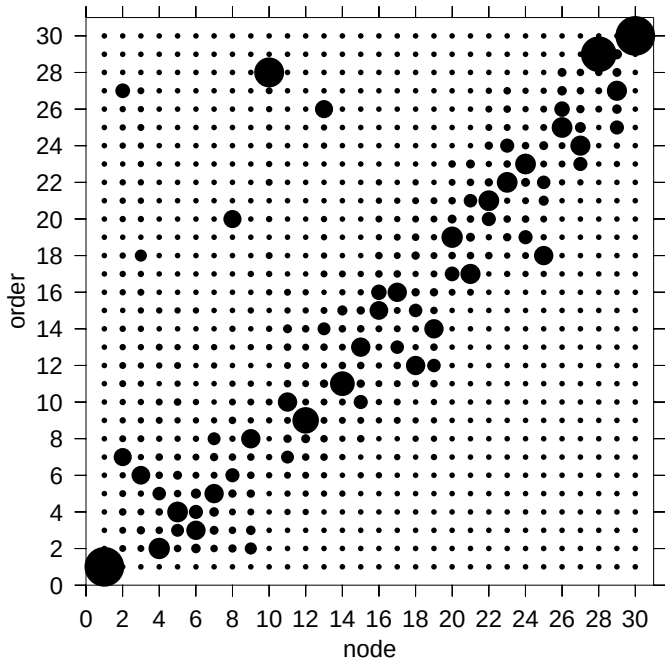


Figure 4.6: Pheromone Heuristic Distribution for ARF

the dot, the stronger the value of the pheromone.

It is clearly seen from Figure 4.6 that there are a few strong pheromone trails while the remaining pheromone trails are very weak. This might be explained by the strong symmetric structure of the ARF DFG and the special implementation in our algorithm of considering operation list only with topologically sorted order. It is also interesting to notice that though a good amount of operations have a limited few alternative "good" positions (such as operation 6 and 26), for some of the operations the pheromone heuristics are strong enough to lock their positions. For example, according to its pheromone distribution, operation 10 shall be placed as the 28-th item in the list and there is no other competitive position for its placement. After careful evaluation, this ordering preference cannot be trivially obtained by constructing priority lists with any of the

popularly used heuristics. This shows that the proposed algorithm has the possibility to discover better orderings which may be hard to achieve intuitively.

4.6.3 Comparison with Simulated Annealing

In order to further investigate the quality of the proposed algorithms, we compared them with a simulated annealing (SA) approach. For resource constrained scheduling, we implemented the algorithm presented in [93]. The basic idea is very similar to what we proposed in our MMAS approach in which a meta-heuristic method (SA) is used to guide the searching process while a traditional list scheduler is used to evaluate the result quality. The scheduling result with the best resource usage is reported when the algorithm terminates.

However, it is more difficult for the timing constrained scheduling problem since we have not found any SA-based approach in previously published works. Therefore, we formulated one ourselves. Consequently, we will give more emphasis on our SA based formulation for the timing constrained scheduling problem in the rest of this section.

A pseudo implementation of SA-based TCS algorithm is given as Algorithm 6. The major challenge here is the construction of a *neighbor* selection in the SA process. With the knowledge of each operation's mobility range, it is trivial to see the search space for the TCS problem is covered by all the possible combinations of the operation/timestep pairs, where each operation can be scheduled into any time step in its mobility range. In our formulation, given a scheduling S where operation op_i is scheduled at t_i, we experimented with two different methods for generating a neighbor solution:

1. *Physical neighbor*: A neighbor of S is generated by selecting an operation op_i and rescheduling it to a physical neighbor of its current scheduled time step t_i, namely either $t_i + 1$ or $t_i - 1$ with even possibility. In case t_i is on the boundary of its mobility range, we treat the mobility range as a circular buffer;

2. *Random neighbor*: A neighbor of S is generated by selecting an operation and rescheduling it to any of the position in its mobility range excluding its currently scheduled position.

However, both of the above approaches suffer from the problem that a lot of these *neighbors* will be invalid because they may violate the data dependency posed by the DFG. For example, say, in S a single cycle operation op_1 is scheduled at time step 3, and another single cycle operation op_2 which is data dependent on op_1 is scheduled at time step 4. Changing the schedule of op_2 to step 3 will create an invalid scheduling result. To deal with this problem in our implementation, for each generated scheduling, we quickly check whether it is valid by verifying the operation's new schedule against those of its predecessor and successor operations defined in the DFG. Only valid schedules will be considered.

Furthermore, in order to give roughly equal chance to each operation to be selected in the above process, we try to generate multiple neighbors before any temperature update is taken. This can be considered as a local search effort, which is widely implemented in different variants of SA algorithm. We control this local search effort with a weight parameter θ. That is before any temperature update taking place, we attempt

to generate θN valid scheduling candidates where N is the number of operations in the DFG. In our work, we set $\theta = 2$, which roughly gives each operation two chances to alter its currently scheduled position in each cooling step.

This local search mechanism is applied to both neighbor generation schemes discussed above. In our experiments, we found there is no noticeable difference between the two neighbor generation approaches with respect to the quality of the final scheduling results except that the *random neighbor* method tends to take significantly more computing time. This is because it is more likely to come up with an invalid scheduling which are simply ignored in our algorithm. In our final realization, we always use the *physical neighbor* method.

Another issue related to the SA implementation is how to set the initial seed solution. In our experiments, we experimented three different seed solutions: ASAP, ALAP and a randomly generated valid scheduling. We found that SA algorithm with a randomly generated seed constantly outperforms that using the ASAP or ALAP initialization. It is especially true when the *physical neighbor* approach is used. This is not surprising since the ASAP and ALAP solutions tend to cluster operations together which is bad for minimizing resource usage. In our final realization, we always use a randomly generated schedule as the seed solution.

The framework of our SA implementation for both timing constrained and resource constrained scheduling is similar to the one reported in [106]. The acceptance of a more costly neighboring solution is determined by applying the Boltzmann probability

procedure SA-TCS(G,R)

input: DFG $G(V,E)$, resource set R, and a map of operation to one resource in R

output: operation schedule

1: perform ASAP and ALAP on the DFG to obtain mobility ranges.
2: randomly initialize a valid seed scheduling $S_{current}$
3: set starting and ending temperature T_s and T_e.
4: set local search weight to θ.
5: set N to be the number of operations.
6: set t to T_s
7: set S_{best} to be $S_{current}$
8: **while** $t > T_e$ **do**
9: **for** $i = 0$; $i < \theta N$; i++ **do**
10: randomly generate a neighbor solution S_n
11: **if** S_n is invalid **then**
12: continue
13: **else**
14: compute the resource cost of S_n
15: randomly accept S_n to be $S_{current}$
16: update S_{best} if needed
17: **end if**
18: **end for**
19: update t based on cooling scheme
20: **end while**
21: return S_{best} and the resource cost

Algorithm 6: Simulated Annealing for Timing-Constrained Scheduling

criteria [1], which depends on the cost difference and the annealing temperature. In our experiments, the most commonly known and used geometric cooling schedule [106] is applied and the temperature decrement factor is set to 0.9. When it reaches the predefined maximum iteration number or the stop temperature, the best solution found by SA is reported.

The experimental results for TCS problem obtained using the above simulated annealing formulation are shown in Table 4.4, where the SA results are provided in parenthesis column by column with those achieved by using MMAS. Similar to the MMAS algorithm, we perform 5 runs for each benchmark sample and report the average savings, the best savings, and the standard deviation of the reported scheduling results. It can be seen from Table 4.4 that the SA method provides much worse results compared with the proposed MMAS solutions. In fact, the MMAS approach provides better results on every testing case. Though the SA method does have significant gains on select cases over FDS, its average performance is actually worse than FDS by 5%, while our method provides a 16.4% average savings. This is also true if we consider the best savings achieved amongst multiple runs where a modest 1% savings is observed in SA comparing with a 19.5% reduction obtained by MMAS method. Furthermore, the quality of the SA method seems to be very dependent on the input applications. This is evidenced by the large dynamic range of the scheduling quality and the larger standard deviation over the different runs. Finally, we also want to make it clear that to achieve this result, the SA approach takes substantially more computing time than the proposed

MMAS method. A typical experiment over all 263 testing cases will run between 9 to 12 hours which is 3 to 4 times longer than the MMAS-based TCS algorithm.

As discussed above, our SA formulation for resource constrained scheduling is similar to that studied in [93]. It is relatively more straight forward since it will always provide valid scheduling using a list scheduler. To be fair, a randomly generated operation list is used as the seed solution for the SA algorithm. The neighbor solutions are constructed by swapping the positions of two neighboring operations in the current list. Since the algorithm always generates a valid scheduling, we can better control the runtime than in its TCS counterpart by adjusting the cooling scheme parameter. We carried experiments using execution limit ranging from 1 to 10 times of that of the MMAS approach. It was observed that SA RCS algorithm provides poor performance when the time limit was too short. On the other hand, once we increase this time limit to over 5 times of the MMAS execution time, there was no significant improvement on the results as the execution time increased. In the rightmost column of Table 4.5, we present the typical RCS results using SA achieved with 10 times the MMAS execution time. The performance data is averaged over 10 runs for each testing sample. It is easy to see that the MMAS-based algorithm consistently outperforms it while using much less computing time.

4.6.4 Parameter Sensitivity

The proposed ACO-based algorithms belong to the category of stochastic search algorithms. This implies a certain sensitivity of the result to the choices of parameters which are at times difficult to determine. In order to better understand this issue and its relationship with the algorithms' performance, a study on their sensitivity to the parameter selection is in order. We have conducted extensive experiments in our work on this topic and will report our major findings in this section.

- α, β and Q: Variation on global heuristic weight α, local heuristic weight β and the pheromone delivery constant Q does not have noticeable impact on the performance of our algorithms. The algorithms consistently provide robust results when α and β are in the range of $[1, 100]$ and Q is between $[1, 5000]$ with small step size, while performance on benchmarks of smaller sizes tend to have more fluctuations than the bigger ones. Of course, a numerically precise limit should be a concern for parameter α and β in algorithm realization because they are used in power functions. Also, the scaling of local and global heuristics could be an issue with these parameters. In our study, we found setting $\alpha = \beta = 1$ worked well in our implementation over a comprehensive set of testing benchmarks. Moreover, the benefit is that it essentially eliminates the power function calls in Equation (2.1) which further reduces computing time.

- ρ: The pheromone evaporation factor ρ takes a value in the range of [0,1] and controls how much the existing pheromone trails will be reduced before any en-

hancement. The smaller this number, the more reduction is applied (see Equation (2.2)). When this number is too small, historical information accumulated in the search process will be essentially lost, and the the algorithms behave close to a random search. In our experiments, we found a value between 0.95 and 1 seems to be a good choice. In our final setup, parameter ρ is set to 0.98.

- p_{best}: This parameter, together with ρ, controls how the lower bound and upper bound of pheromone trails will be computed. Recall that when $p_{best} \rightarrow 0$, the difference between $\tau_{min}(t)$ and $\tau_{max}(t)$ gets smaller, which means the search is getting more random and more emphasis is given to search space exploration. In our experiments, we found that p_{best} should be bigger than 0.5. Once it is above this threshold, both algorithms for RCS and TCS problems perform robustly. In our final setup, p_{best} is set to 0.93.

- m and N: The ant count m and the iteration number N are closely related and have a direct impact on the algorithms execution time. Roughly, the product of m and N gives an estimation of how many scheduling instances the algorithms will cover. Theoretically, the bigger this product, the better the performance. Also, it is intuitive to see that these parameters should be positively correlated with the complexity of the test sample. In our work, we prefer to use a fixed setting for these parameters in order to make the algorithm simpler. As reported above, with $m = 10$ and N is set to be 150 and 100 for the TCS and RCS problem respectively, our algorithms work well over a wide range of testing samples. In

a further study, we varied m between 1 and 10, and N from 50 to 1000. We find that little performance improvement is seen after N is bigger than 250 when m is reasonable large ($\geqslant 4$). We contribute this to the fact that the pheromone trails converge after a large number of iterations.If N is smaller than 100, we will often miss the optimal solution because of premature termination. This is especially true for the TCS problem. Similarly, when m is bigger than 6, we see little improvement. The best tradeoff of m seems to be between 4 and 6. It is interesting to notice that these numbers are very close to the average branching factor of the testing samples. These results implies that we may still have room to fine tune these two parameters to further improve the performance/cost tradeoff of the algorithms.

4.7 Summary

In this chapter, we presented two novel heuristic searching methods for the resource and timing constrained scheduling problems based on the MAX-MIN Ant System. Our algorithms employ a collection of agents that collaborate to explore the search space. We proposed a stochastic decision making strategy in order to combine global and local heuristics to effectively conduct this exploration. As the algorithms proceed in finding better quality solutions, dynamically computed local heuristics are utilized to better guide the searching process.

A comprehensive set of benchmarks was constructed to include a wide range of

applications. Experimental results over our test cases showed promising results. The proposed algorithms consistently provided higher quality results over the tested examples and achieved very good savings comparing to traditional simulated annealing, list scheduling and force-directed scheduling approaches. Furthermore, the algorithm demonstrated robust stability over different applications and different selection of local heuristics, as evidenced by a much smaller deviation over the results.

To the best of our knowledge, the only other reported work on using Ant Colony Optimization to solve the operation scheduling problem is done by Kopuri *et al.* [58]. Compared to our work, their study is limited to the timing constrained scheduling problem.

To address the TCS problem, their algorithm has a different formulation and is more closely related to the classic force-directed scheduling algorithm. They use a modified self force computation, where predecessor and successor forces are dropped in the overall force consideration. This force is calculated by linear combination of normalized classic self-force and the pheromone trails. Since the resulting value can be both negative and positive, it is hard to act as an indicator for operation selection during the scheduling construction process. In their work, simple random selection is used.

Our algorithm uses a dynamically computed distribution graph for the corresponding resource k for the local heuristic and force calculation is not needed. We believe it provides the following benefits:

- It is directly tied with the optimization target, i.e. minimizing the resource cost.

- It is faster to compute.
- The value range for the distributed graph is non-negative, which enables more effective operation selection strategy than random selection as discussed in Section 4.3.3.

Moreover, as discussed in Section 4.3, our algorithm can be readily extended to handle different design scenarios such as multiple-cycle operations, mutually exclusive operations, operation chaining and pipelining. It is unclear if their algorithm can be easily extended to do so, and only single cycle operations were used in their study.

It is known that premature convergence is an important issue in ant based approaches and our experience shows this is an important factor for the operation scheduling problem. In order to cope with this, MAX-MIN formulation is used in our algorithms for both timing and resource constrained scheduling. No such mechanism was used in [58].

Finally, the effectiveness and efficiency of our algorithms is tested over a comprehensive benchmark suite compiled from real-world applications. The performance with respect to solution quality, stability, scalability, and timing performance is more thoroughly studied and reported here. Only limited results on a small number of samples were reported in [58].

Chapter 5

Design Space Exploration

Design space exploration during high level synthesis is often conducted through ad-hoc probing of the solution space using some scheduling algorithm. This is not only time consuming but also very dependent on designer's experience. We propose a novel design exploration method that exploits the duality of the time and resource constrained scheduling problems. Our exploration automatically constructs a time/area tradeoff curve in a fast, effective manner. It is a general approach and can be combined with any high quality scheduling algorithm. In our work, we use the MAX-MIN ant colony optimization technique to solve both the time and resource constrained scheduling problems. Our algorithm provides significant solution quality savings (average 17.3% reduction of resource counts) with similar run time compared to using force directed scheduling exhaustively at every time step. it also scales well across a comprehensive benchmark suite constructed with classic and real-life samples.

5.1 Introduction

When building a digital system, designers are faced with a countless number of decisions. Ideally, they must deliver the smallest, fastest, lowest power device that can implement the application at hand. More often than not, these design parameters are contradictory. For example, making the device run faster often makes it larger and more power hungry. Designers must also deal with increasingly strict time to market issues. Unfortunately, this does not afford them much time to make a decision.

Designers must be able to reason about the tradeoffs amongst a set of parameters. Such decisions are often made based on experience, i.e. this worked before, it should work again. Exploration tools that can quickly survey the design space and report a variety of options are invaluable.

From optimization point of view, design space exploration can be distilled to identifying a set of Pareto optimal design points according to some objective function. These design points form a curve that provides the best tradeoffs for the variables in the objective function. Once the curve is constructed, the designer can make design decisions based on the relative merits of the various system configurations. Timing performance and the hardware cost are two common objectives in such process.

Resource allocation and scheduling are two fundamental problems in constructing such Pareto optimal curves for time/cost tradeoffs. The two problems are tightly interwoven. Resource constrained scheduling takes as input an application modeled as data flow graph and a number of different types of resources. It outputs a start time for each

of the operations, such that the resource constraints are not violated, while attempting to minimize the application latency. Here allocation is performed before scheduling, and the schedule is obviously very dependent on the allocation; a different resource allocation will likely produce a vastly different scheduling result.

We could perform scheduling before allocation; this is the time constrained scheduling problem. Here the inputs are a data flow graph and a time deadline (latency). The output is again a start time for each operation, such that the latency is not violated, while attempting to minimize the number of resources that are needed. It is not clear as to which solution is better. Nor is it clear on the order that we should perform scheduling and allocation.

One possible method of design space exploration is to vary the constraints to probe for solutions in a point-by-point manner. For instance, you can use some time constrained algorithm iteratively, where each iteration has a different input latency. This will give you a number of solutions, and their various resource allocations over a set of time points. Or you can run some resource constrained algorithm iteratively. This will give you a latency for each of these area constraints.

An effective design space exploration strategy must understand and exploit the relationship between the time and resource constrained problems. Unfortunately, designers are left with individual tools for tackling either problem. They are faced with questions like: Where do we start the design space exploration? What is the best way to utilize the scheduling tools? When do we stop the exploration?

Moreover, due to the lack of connection amongst the traditional methods, there is very little information shared between time constrained and resource constrained solutions. This is unfortunate, as we are throwing away potential solutions since solving one problem can offer more insight into the other problem.

In this chapter, we describe a design space exploration strategy for scheduling and resource allocation. The ant colony optimization (ACO) meta-heuristic lies at the core of our algorithm. We switch between timing and resource constrained ACO heuristics to efficiently traverse the search space. Our algorithms dynamically adjust to the input application and produce a set of high quality solutions across the design space.

The rest of the chapter is organized as follows. We discuss related work in the next section. In Section 5.3, we present a design space exploration algorithm using duality between the time and resource scheduling problems. Together, we will discuss why ACO-based scheduling algorithms are suitable to be integrated within the proposed exploration framework. Experimental results for the new algorithms are presented and analyzed in Section 5.4. We summarize with Section 5.5.

5.2 Related Work

The scheduling and resource allocation problems form the basis for design space exploration during high level synthesis. These problems can be formulated as an Integer Linear Program (ILP) [107]; however it is typically impossible to solve large problem instances in this manner. Much research has been done to cleverly use heuristic

approaches to address these problems.

In [29], the authors concentrate on providing alternative "module bags" for design space exploration by heuristically solving the clique partitioning problems and using force directed scheduling. Their work focuses more on the situations where the operations in the design can be executed on alternative resources. In the Voyager system [19], scheduling problems are solved by carefully bounding the design space using ILP, and good results are reported on small sized benchmarks. Moreover, it reveals that clock selection can have an important impact on the final performance of the application. In [49, 26, 75], genetic algorithms are implemented for design space exploration. Simulated annealing [65] has also been applied in this domain. A survey on design space exploration methodologies can be found in [63] and [66].

Force directed scheduling (FDS) [78] is a popular scheduling algorithm. The original FDS algorithm is designed to solve the time constrained scheduling (TCS) problem, i.e. to reduce the number of functional units used in the implementation with a given execution deadline. This objective is achieved by attempting to uniformly distribute the operations onto the available resource units. The distribution ensures that resource units allocated to perform operations in one control step are used efficiently in all other control steps, which leads to a high utilization rate. A"force" is used to measure the parallel usage of a resource type. Each force is computed based on the operation's mobility range under the assumption that each operation op_i has a uniform probability of being scheduled into any of the control steps in this range. The algorithm proceeds

iteratively by selecting the operation and time step with the minimal force. The authors also proposed a method called force-directed list scheduling (FDLS) to address the resource constrained scheduling problem. Here, the priority function of the list scheduler is constructed using forces.

The FDS method is constructive since the solution is computed without backtracking. Every decision is made deterministically in a greedy manner. If there are two potential assignments with the same cost, the FDS algorithm cannot accurately estimate the best choice. Moreover, FDS does not take into account future assignments of operators to the same control step. Consequently, it is possible that the resulting solution will not be optimal due to its greedy nature. FDS works well on small sized problems, however, it often results to inferior solutions for more complex problems. This phenomena is observed in our experiments reported in Section 5.4.

In this work, we focus our attention on the basic design space exploration problem similar to the one treated in [78], where the designers are faced with the task of mapping a well defined application represented as a DFG onto a set of known resources where the compatibility between the operations and the resource types has been defined. Furthermore, the clock selection has been determined in the form of execution cycles for the operations. The goal is to find the a Pareto optimal tradeoff amongst the design implementations with regard to timing and resource costs. Our basic method can be extended to handle clock selection and the use of alternative resources. However, this is beyond the scope of this study.

5.3 Exploration Using Time and Resource Constrained Duality

5.3.1 Iterative Design Space Exploration Leveraging Duality

We are concerned with the design problem of making tradeoffs between hardware cost and timing performance. This is still a commonly faced problem in practice, and other system metrics, such as power consumption, are closely related with them. Based on this, we have a 2-D design space as illustrated in Figure 5.1(a), where the x-axis is the execution deadline and the y-axis is the aggregated hardware cost. Each point represents a specific tradeoff of the two parameters.

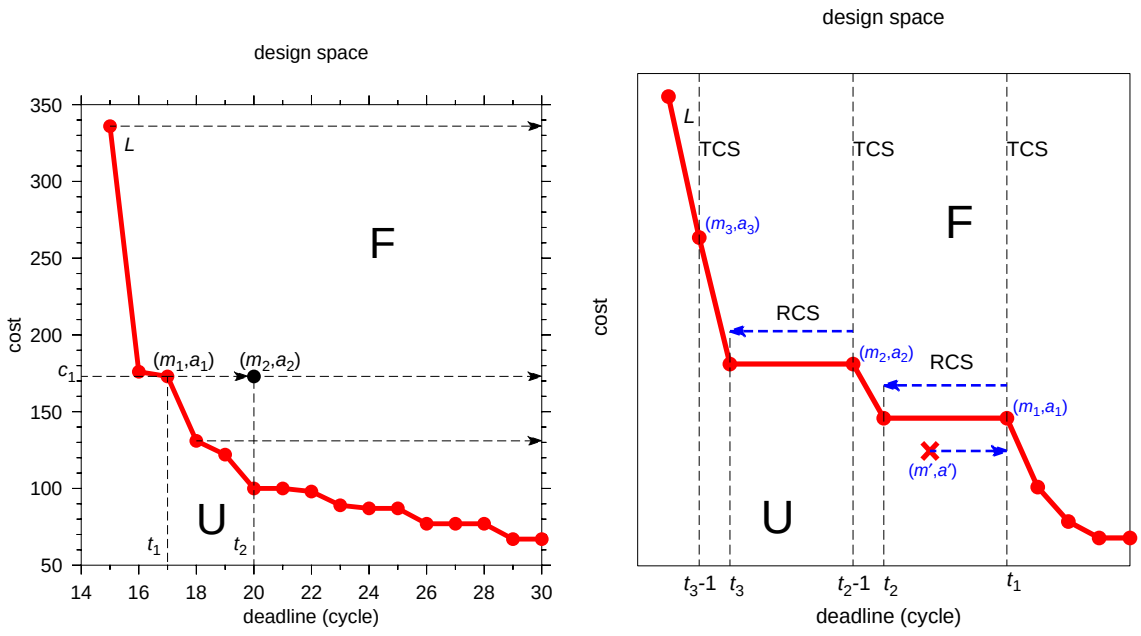


Figure 5.1: Design Space Exploration Using Duality between Schedule Problems (Curve *L* gives the optimal time/cost tradeoffs.)

For a given application, the designer is given R types of computing resources (e.g. multipliers and adders) to map the application to the target device. We define a specific design as a *configuration*, which is simply the number of each specific resource type. In order to keep the discussion simple, in the rest of the chapter we assume there are only two resource types M (multiply) and A (add), though our algorithm is not limited to this constraint. Thus, each configuration can be specified by (m,a) where m is the number of resource M and a is the number of A.

It is worth noticing that for each point in the design space shown in Figure 5.1(a), we might have multiple configurations that could realize it. For example, assuming unit cost for all resources, it is possible that a configuration with 10 multipliers and 10 adders can achieve the same execution time as another configuration with 5 multipliers and 15 adders, as both solutions have the same cost (20).

Studying the design space more carefully, reveals several key observations. First, the achievable deadlines are limited to the range $[t_{asap}, t_{seq}]$, where t_{asap} is the ASAP time for the application while t_{seq} is the sequential execution time when we have only one instance for each resource type. It is impossible to get a solution faster than the ASAP solution and any solution with a deadline beyond that of t_{seq} are not Pareto optimal. Furthermore, for each specific configuration we have the following lemma about the portion of the design space that it maps to.

Lemma 5.3.1 *Let C be a feasible configuration with cost c for the target application. The configuration maps to a horizontal line in the design space starting at* (t_{min}, c),

where t_{min} is the resource constrained minimum scheduling time.

The proof of the lemma is straightforward as each feasible configuration has a minimum execution time t_{min} for the application, and obviously it can handle every deadline longer than t_{min}. For example, in Figure 5.1(a), if the configuration (m_1, a_1) has a cost c_1 and a minimum scheduling time t_1, the portion of design space that it maps to is indicated by the arrow next to it. Of course, it is possible for another configuration (m_2, a_2) to have the same cost but a bigger minimum scheduling time t_2. In this case, their feasible space overlaps beyond (t_2, c_1).

As we discussed before, the goal of design space exploration is to help the designer find the optimal tradeoff between the time and area. Theoretically, this can be done by finding the minimum area c amongst all the configurations that are capable of producing $t \in [t_{asap}, t_{seq}]$. In other words, we can find these points by performing time constrained scheduling (TCS) on all t in the interested range. These points form a curve in the design space, as illustrated by curve L in Figure 5.1(a). This curve divides the design space into two parts, labeled with F and U respectively in Figure 5.1(a), where all the points in F are feasible to the given application while U contains all the unfeasible time/area pairs. More interestingly, we have the following attribute for curve L:

Lemma 5.3.2 *Curve L is monotonically non-increasing as the deadline t increases.*

Proof Assume the lemma is false. Therefore, we will have two points (t_1, c_1) and (t_2, c_2) on the curve L where $t_1 < t_2$ and $c_1 < c_2$. This means we have a specific configuration C with cost c_1 that is capable of producing an execution time t_1 for the ap-

plication. Since $t_1 < t_2$, and also from Lemma 5.3.1, we know that configuration C can produce t_2. This introduces a contradiction since c_2, which is worse than c_1, is the minimum cost at t_2.

Due to this lemma, we can use the dual solution of finding the tradeoff curve by identifying the minimum resource constrained scheduling (RCS) time t amongst all the configurations with cost c. Moreover, because the monotonically non-increasing property of curve L, there may exist horizontal segments along the curve. Based on our experience, horizontal segments appear frequently in practice. This motivates us to look into potential methods to exploit the duality between RCS and TCS to enhance the design space exploration process. First, we consider the following theorem:

Theorem 5.3.3 *If C is a configuration that provides the minimum cost at time t_1, then the resource constrained scheduling result t_2 of C satisfies $t_2 \leqslant t_1$. More importantly, there is no configuration C' with a smaller cost that can produce an execution time within $[t_2, t_1]$.*

Proof The first part of the theorem is obvious. Therefore, we focus on the second part. Assuming there is a configuration C' that provides an execution time $t_3 \in [t_2, t_1]$, then C' must be able to produce t_1 based on Lemma 5.3.1. Since C' has a smaller cost, this conflicts with the fact that C is the minimum cost solution (i.e. the TCS solution) at time t_1. Thus the statement is true. This is illustrated in Figure 5.1(b) with configuration (m_1, a_1) and (m', a').

This theorem provides a key insight for the design space exploration problem. It says that if we can find a configuration with optimal cost c at time t_1, we can move along the horizontal segment from (t_1, c) to (t_2, c) without losing optimality. Here t_2 is the RCS solution for the found configuration. This enables us to efficiently construct the curve L by iteratively using TCS and RCS algorithms and leveraging the fact that such horizontal segments do frequently occur in practice. Based on the above discussion, we propose a new space exploration algorithm as shown in Algorithm 7 that exploits the duality between RCS and TCS solutions. Notice the *min* function in step 10 is necessary since a heuristic RCS algorithm may not return the true optimal that could be worse than t_{cur}.

procedure DSE
output: curve L

1: interested time range $[t_{min}, t_{max}]$, where $t_{min} \geqslant t_{asap}$ and $t_{max} \leqslant t_{seq}$.
2: $L = \phi$
3: $t_{cur} = t_{max}$
4: **while** $t_{cur} \geqslant t_{min}$ **do**
5: perform TCS on t_{cur} to obtain the optimal configurations C_i.
6: **for** configuration C_i **do**
7: perform RCS to obtain the minimum time t^i_{rcs}
8: **end for**
9: $t_{rcs} = \min_i (t^i_{rcs})$ /* find the best rcs time */
10: $t_{cur} = \min (t_{cur}, t_{rcs}) - 1$
11: extend L based on TCS and RCS results
12: **end while**
13: return L

Algorithm 7: Iterative Design Space Exploration Algorithm

By iteratively using the RCS and TCS algorithms, we can quickly explore the design space. Our algorithm provides benefits in runtime and solution quality compared with using RCS or TCS alone. Our algorithm performs exploration starting from the largest deadline t_{max}. Under this case, the TCS result will provide a configuration with a small number of resources. RCS algorithms have a better chance to find the optimal solution when the resource number is small, therefore it provides a better opportunity to make large horizontal jumps. On the other hand, TCS algorithms take more time and provide poor solutions when the deadline is unconstrained. We can gain significant runtime savings by trading off between the RCS and TCS formulations.

5.3.2 Integrate with ACO-based Scheduling Algorithms

The proposed framework is general and can be combined with any scheduling algorithm. We found that in order for it to work in practice, the TCS and RCS algorithms used in the process require special characteristics. First, they must be fast, which is generally requested for any design space exploration tool. More importantly, they must provide close to optimal solutions, especially for the TCS problem. Otherwise, the conditions for Theorem 5.3.3 will not be satisfied and the generated curve L will suffer significantly in quality. Moreover, notice that we enjoy the biggest jumps when we take the minimum RCS result amongst all the configurations that provide the minimum cost for the TCS problem. This is reflected in Steps 6-9 in Algorithm 7. For example, it is possible that both (m, a) and (m', a') provide the minimum cost at time t but they have

different deadline limits. Therefore a good TCS algorithm used in the proposed approach should be able to provide multiple candidate solutions with the same minimum cost, if not all of them.

In order to select the suitable TCS and RCS algorithms, we studied different scheduling approaches for the two problems, including the popularly used force directed scheduling (FDS) for the TCS problem [78], various list scheduling heuristics, and the recently proposed Ant Colony Optimization (ACO) based instruction scheduling algorithms [104, 99].

We found that ACO-based scheduling algorithms offer the following major benefits over FDS, several variants of list scheduling and simulated annealing [99]:

- ACO-based scheduling algorithms generate better quality results that are close to the optimal with good stability for both the TCS and RCS problems;
- ACO-based methods provide reasonable runtime;
- Furthermore, as a population based method, ACO-based TCS approach naturally provides multiple alternative solutions. As we have discussed, this feature provides potential benefit in the proposed DSE process since we can select the largest jump provided by these candidates.

5.4 Experiments and Analysis

5.4.1 Benchmarks and Setup

We implemented four different design space exploration algorithms:

1. FDS: exhaustively step through the time range by performing time constrained force directed scheduling at each deadline;

2. MMAS-TCS: step through the time range by performing only MMAS-based TCS scheduling at each deadline.

3. MMAS-D: use the iterative approach proposed in Algorithm 7 by switching between MMAS-based RCS and TCS.

4. FDS-D: similar to the MMAS-D, except using FDS-based scheduling algorithms.

We implemented the MMAS-based TCS and RCS algorithms as described in Section 5.3.2. Since there is no widely distributed and recognized FDS implementation, we implemented our own. The implementation is based on [78] and has all the applicable refinements proposed in the paper, including multi-cycle instruction support, resource preference control, and look-ahead using second order of displacement in force computation.

For all testing benchmarks, the operations are allocated on two types of computing resources, namely MUL and ALU, where MUL is capable of handling multiplication and division, while ALU is used for other operations such as addition and subtraction.

Furthermore, we define the operations running on MUL to take two clock cycles and the ALU operations take one. This definitely is a simplified case from reality, however, it is a close enough approximation and does not change the generality of the results. Other operation to resource mappings can easily be implemented within our framework.

With the assigned resource/operation mapping, ASAP is first performed to find the critical path delay L_c. We then set our predefined deadline range to be $[L_c, 2L_c]$, i.e. from the critical path delay to two times of this delay. This results in 263 testing cases in total. Four design space exploration experiments are carried out. For the FDS and MMAS-TCS algorithms, we run force-directed or MMAS-based time constrained scheduling on every deadline and report the best schedule results together with the costs obtained. For the MMAS-D and FDS-D algorithms, we only run MMAS-based or FDS-based TCS on selected deadlines starting from $2L_c$ and make jumps based on the RCS results on the configurations previously obtained by performing TCS.

5.4.2 Quality Assessment

We first studied the effectiveness of the ACO approach for design space exploration. Two individual tests are carried out, one to verify its performance on TCS problem with a specific deadline, while the other tries to confirm its performance over the entire design space.

In the first tests, MMAS-based TCS is performed on the *idctcol* benchmark, an implementation of inverse discrete cosine transform, with deadline set to its ASAP time

19. We use 10 ants for each iteration, which provides 10 individual scheduling solutions. The total iteration limit is set to 200, which produces a total of 2000 scheduling results for this TCS problem. We want to examine the effectiveness of the algorithm. In other words, how does the quality of the solutions improve across iterations? Figure 5.2 shows this result by plotting the solution quality/frequency curves over time. Here each curve aggregates solutions found within certain iterations. For example, the curve labeled "1-200" diagrams the quality distribution for the first 200 scheduling results obtained in the first 20 iterations. The x-axis is the hardware cost for the schedule results, where we simply use resource number counts. The y-axis shows the number of solutions that iteration range produces at each specific cost.

From this graph, we can easily see the MMAS-based TCS is working. For example, comparing the initial 200 solutions (1-200) and the final 200 solutions (1801-2000). In the initial 200 solutions, there are 5 solutions with an area of 20, and the best solutions have area of 14 (there are 12 such solutions); by the last 200 solutions, there are 0 with an area of 20, 69 with an area of 14, and one with an area of 11. As the algorithm progresses, a positive trend emerges where the ants ignore the worst solutions and enforce the better ones.

To show the effectiveness of the algorithm over the whole design space, similar experiments are conducted across the range of interested deadlines. Figure 5.3 gives one example on the *idctcol* benchmark on deadlines from 19 to 32, where the x-axis is the deadline constraint and y-axis is the cost for scheduling results. The size of

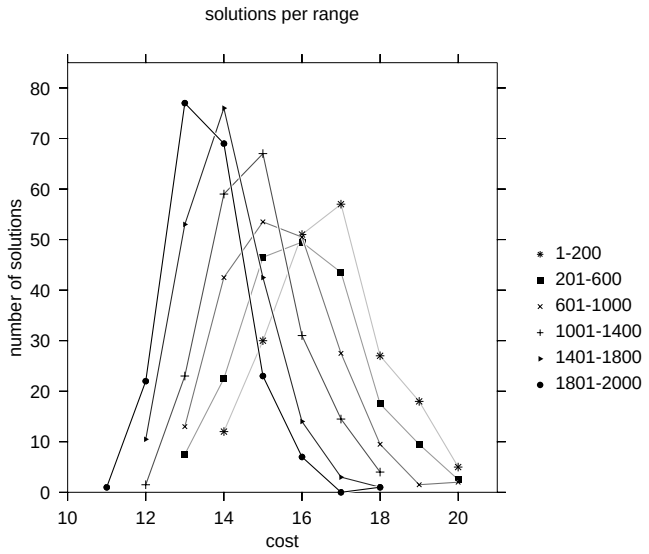


Figure 5.2: Distribution of the TCS ACO solution quality on idctcol benchmark with a deadline set to its ASAP time. Each line shows a different phase of the algorithm execution where each point gives the number of solutions of a particular resource cost. The line "1-200" denotes the first 200 solutions found by the ACO algorithm, while the line "1801-2000" gives last 200 solutions.

dots are proportional to the number of schedule results that the ants produce for the specific cost and deadline. It is easy to see that the focus area of algorithm adjusts as the constraints change. Moreover, if we inspect each column more carefully, we can see that the algorithm effectively explores the "best" part of the design space. This is evidenced by the movement of the dense area in the graph and the relatively invariant vertical spread.

We performed experiments on each benchmark using the four different design space exploration algorithms. First, time constrained FDS scheduling is used at every deadline. The quality of results is used as the baseline for quality assessment. Then MMAS-TCS, MMAS-D and FDS-D algorithms are executed; the difference is that MMAS-

TCS steps through the design space in the same way as FDS while MMAS-D and FDS-D utilize the duality between TCS and RCS. Because of their randomized nature, the MMAS-TCS and MMAS-D algorithms are executed five times in order to obtain enough statistics to evaluate their stability.

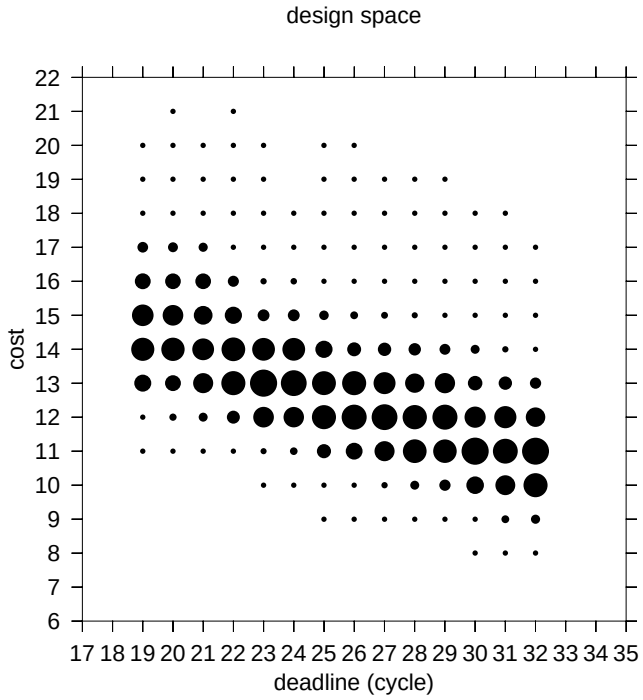


Figure 5.3: Solution quality of the TCS ACO on the idctcol benchmark. We run the TCS ACO algorithm at each deadline ranging from its ASAP time (19) to (32). The size of the dot indicates the proportion of solutions with a specific resource cost found at each deadline.

Detailed design space exploration results for six of the benchmarks are shown in Figure 5.4, where we compare the curves obtained by MMAS-D, FDS-D and FDS algorithms. Table 5.1 summarizes the experiment results. For each benchmark we give the node/edge count, and the average resource saving of the FDS-D, MMAS-TCS and MMAS-D algorithms comparing with FDS. We report the saving in percentage of total

resource counts (a negative result indicates a lower (better) resource cost). We weight the two resource types M and A equally, though we use different cost weights to bias alternative solutions (for example, solution (3M, 4A) is more favorable than (4M, 3A) as resource M has a large cost weight. We could easily vary the relative costs and number of the resources types. However, we feel this is would introduce confusion caused by different weight choices. The percentage savings is computed for every deadline of every benchmark. The average for a certain benchmark is reported in Table 5.1. It is easy to see that MMAS-TCS and MMAS-D both outperform the classic FDS method across the board with regard to solution quality, often with significant savings. Overall, MMAS-TCS achieves an average improvement of 16.4% while MMAS-D obtains a 17.3% improvement. Both algorithms scale well for different benchmarks and problem sizes. Moreover, by computing the standard deviation over the 5 different runs, the algorithms are shown to be very stable. For example, the average standard deviation on result quality for MMAS-TCS is only 0.104. On the other hand, the FDS-D algorithms has a minor performance degradation comparing with the FDS baseline. It outperforms FDS over 14 out of the 20 benchmarks, gives worse result on 4 samples, and shows no change on 2 testing cases. Though it provides modestly better results over two testing samples (i.e. *wbmpheader* and *interpolate*) when compared to MMAS-D, overall MMAS-D produces a much better result. Finally, FDS-D is much less stable with regard to the result quality. It seems to be more application dependent and yields bad results in certain cases (e.g. benchmark *jpegidictifast*).

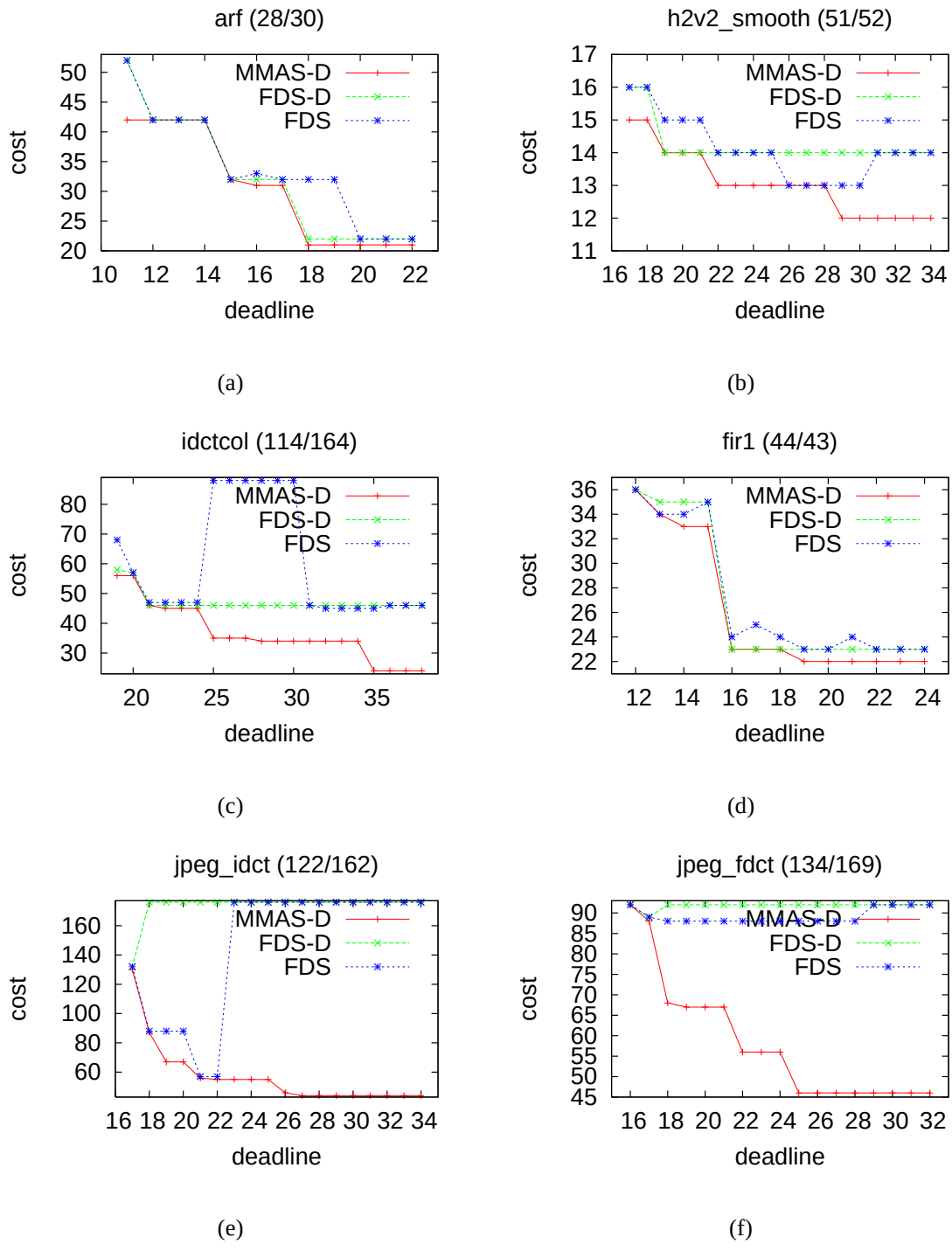


Figure 5.4: Design Space Exploration results: MMAS-D, FDS-D and FDS

Name	Nodes/Edges	Deadline	FDS-D	MMAS-TCS	MMAS-D
HAL	11/8	(6 - 12)	14.3%	-7.1%	-7.1%
hbsurf	18/16	(11 - 22)	0.0%	-9.9%	-13.2%
ARF	28/30	(11 - 22)	-4.7%	-12.4%	-18.6%
motionvectors	32/29	(7 - 14)	-8.0%	-13.1%	-16.0%
EWF	34/47	(17 - 34)	-5.6%	-11.5%	-21.9%
FIR2	40/39	(12 - 24)	-4.1%	-16.8%	-22.8%
FIR1	44/43	(12 - 24)	-3.9%	-15.2%	-18.0%
h2v2_smooth	51/52	(17 - 34)	4.2%	-19.3%	-20.5%
feedbackpoints	53/50	(11 - 22)	-1.2%	-5.9%	-9.1%
collapsepyr	56/73	(8 - 16)	-5.3%	-18.3%	-20.0%
COSINE1	66/76	(10 - 20)	-3.1%	-21.5%	-23.5%
COSINE2	82/91	(10 - 20)	0.7%	-5.6%	-8.1%
wbmpheader	106/88	(8 - 16)	-2.4%	-0.9%	-1.6%
interpolate	108/104	(10 - 20)	-2.3%	-0.2%	-1.8%
matmul	109/116	(11 - 22)	-4.7%	-3.7%	-5.6%
idctcol	114/164	(19 - 38)	-11.2%	-30.7%	-32.0%
jpegidctifast	122/162	(17 - 34)	35.2%	-50.3%	-52.1%
jpegfdctislow	134/169	(16 - 32)	16.2%	-31.4%	-34.6%
smoothcolor	197/196	(15 - 30)	-4.6%	-7.3%	-8.6%
invertmatrix	333/354	(15 - 30)	0.0%	-11.2%	-11.9%
Total Avg.			0.48%	-16.4%	-17.3%

Table 5.1: Summary for Design Space Exploration Results. Each line gives the benchmark name, the tested time range and the results of each design space exploration algorithm (FDS-D, MMAS-TCS, MMAS-D compared to the exhaustive FDS result. (A negative result indicates a smaller resource allocation, which is desired.)

It is interesting and initially surprising to observe that the MMAS-D always had no-worse performance than MMAS-TCS method. More careful inspection on the experiments reveals the reason: using the duality between TCS and RCS not only reduces the computation time but can also improves the quality of the result. To understand this, recall Theorem 5.3.3 and Figure 5.1(b). If we achieve an optimal solution at t_1, with MMAS-D we automatically extend this optimality from t_1 to t_2, while the MMAS-TCS algorithm can provide worse quality solutions on deadlines between t_1 and t_2.

This benefit is not specifically associated with MMAS scheduling algorithms, rather, it is also observed when other scheduling methods are used. For example, consider the curve generated by FDS-D in Figure 5.4(d). We can see that the configuration provided by TCS at deadline 24 can be pushed to deadline 16. FDS-D achieves better results over FDS at time stamps of 16, 17, 18 and 21. However, we will not always obtain this benefit. For the same curve, FDS-D actually suffers worse result at time 13 and 14. Extreme examples of this are shown in Figure 5.4(e) and Figure 5.4(f), the two worst samples for FDS-D. It is easy to realize that if a generous TCS result is generated at a bigger deadline, the following RCS step is misled to provide a very small deadline result. The effect is that the algorithm provides a poor tradeoff curve.

In conclusion, the proposed duality based design space exploration framework is a general approach and can be combined with any scheduling algorithm. However, the selection of such scheduling algorithms has a direct impact on the quality of the result-

ing tradeoff curves. This is not surprising in light of our discussion on Theorem 5.3.3 in Section 5.3.

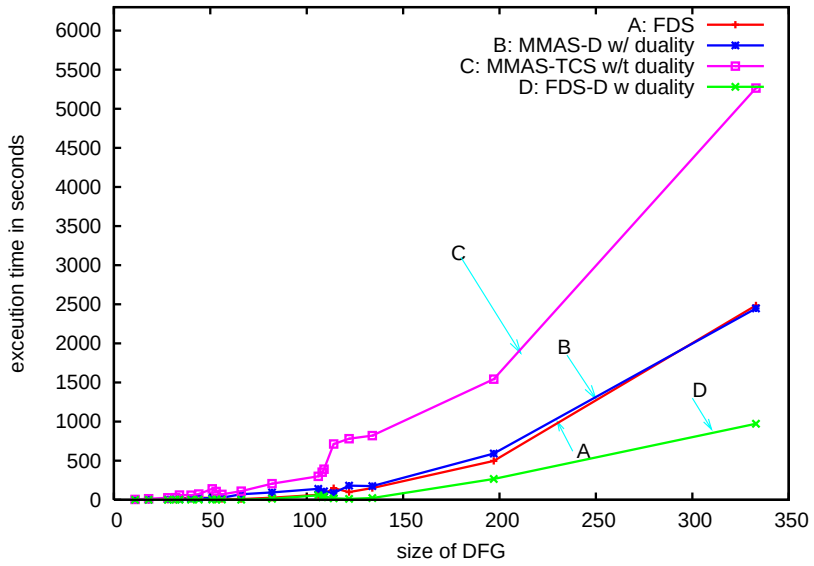


Figure 5.5: Timing Performance Comparison

Figure 5.5 diagrams the average execution time comparison for the four design space exploration approaches, ordered by the size of the benchmark. All of the experiment results use the same Linux box with a 2GHz CPU. It is easy to see that the all the algorithms have similar run time scale, where MMAS-TCS takes more time, while MMAS-D and FDS have very close run times–especially on larger benchmarks. The major execution time savings come from the fact that MMAS-D exploits the duality and only computes TCS on selected number of deadlines. Over 263 testing cases, we find on average MMAS-D skips about 44% deadlines with the help of RCS. The fact that MMAS-D achieves much better results than FDS with almost the same execution time makes it very attractive in practice.

5.5 Summary

We proposed a novel design space exploration method that bridges the time and resource constrained scheduling problems and exploits their duality. Our algorithm provides time/cost tradeoff curve in the design space in a systemic manner. We proved that it is possible to use the duality to help us effectively construct such a curve, while reducing the computing time and improving the quality of the result. The proposed method is general and can be combined with any high quality scheduling algorithms. However, the underlying scheduling algorithms has direct impact on the quality of the tradeoffs curve. We showed that ACO-based scheduling algorithms are ideal due to their robustness, high performance, reasonable execution time and the capability of providing multiple scheduling candidates. Our algorithms outperformed the popularly used force directed scheduling method with significant savings (average 17.3% savings on resource counts) and almost the same run time on comprehensive benchmarks constructed with classic and real-life samples. The algorithms also scaled well over different applications and problem sizes.

Chapter 6

Conclusions and Future Work

In this section, we first conclude our work on applying Ant Colony metaheuristics to solve the architectural design problems. Then we present some thoughts on potential future work in this area.

6.1 Conclusions

As VLSI technology advances, we have seen the steady shrink of feature sizes in integrated circuits and an exponential increase in capacity per die and per dollar at an exponential rate. Though we may see the end of the exponential feature size scaling in the next 20 years for conventional CMOS-based IC, there are promising evidence that advances in basic science may keep this trend forward with emerging technologies such as nanoscale material and molecular computing. Based on this observation, we project that system design techniques will soon become a necessity in order to tame

the immense complexity of future computing systems. This is especially true with the advent of complex system architectures that contain a variety of computing components like microprocessors, memory elements, ASIC, reconfigurable logic, even nanoscale devices. With increasing abundance and flexibility of computing resources, how to effectively use them to fully exploit the benefits becomes a renewed problem for the EDA community with much more difficult tasks in hand. To answer the challenge, we must look towards new optimizations methods, rather than simply perform iterative improvements on existing techniques.

In this study, we focus on constructing effective and efficient algorithms for solving a number of fundamental architectural design problems using the Ant Colony Optimzation, a relatively new meta-heuristic method inspired by the study of the behaviors of social insects. Comparing with other conventional metaheuristic methods, ACO-based approaches pose a set of unique advantages and have been proven effective in solving a wide range of traditionally hard combinatory problems. One special motivation for us to applying ACO to design problems is its natural connection with graph based modeling, which is often used in various system design problems.

To study the effectiveness of ACO method, we investigate three problems, namely system partitioning, operation scheduling and design space exploration problem, all of which are $\mathcal{NP}$-hard. By carefully examining the problem specific characteristics, we construct concrete algorithms to solve these problems under the ACO framework. Our algorithms utilize a unique hybrid approach by combining the ant colony meta-

heuristic with problem specific knowledge, where a collection of agents cooperate using distributed and local heuristic information to effectively explore the search space.

Our experiments over comprehensive benchmark suites show very promising results. For the system partitioning problem, the proposed algorithm provides robust results that are qualitatively close to the optimal with minor computational cost. Comparing with popularly used simulated annealing approach, the proposed algorithm gives better solutions with substantial reduction on execution time. For the operation scheduling problem, a comprehensive set of benchmarks was constructed to include a wide range of applications. The proposed algorithms consistently provide higher quality results over the tested examples and achieved very good savings comparing to traditional simulated annealing, list scheduling and force-directed scheduling approaches. Furthermore, the algorithm demonstrated robust stability over different applications and different selection of local heuristics, as evidenced by a much smaller deviation over the results.

Moreover, we propose a novel design space exploration method that bridges the time and resource constrained scheduling problems and exploits their duality. Our algorithm provides time/cost tradeoff curve in the design space in a systemic manner. We prove that it is possible to use the duality to help us effectively construct such a curve, while reducing the computing time and improving the quality of the result. The proposed method is general and can be combined with any high quality scheduling algorithms. However, the underlying scheduling algorithms have direct impact on

the quality of the tradeoffs curve. We showed that ACO-based scheduling algorithms are ideal due to their robustness, high performance, reasonable execution time and the capability of providing multiple scheduling candidates. Our algorithms outperformed the popularly used force directed scheduling method with significant savings (average 17.3% savings on resource counts) and almost the same run time on comprehensive benchmarks constructed with classic and real-life samples. The algorithms also scale well over different applications and problem sizes.

6.2 Future Work

Besides the promising results we have seen in the problems we investigated, we believe the Ant Colony metaheuristic method may also be helpful for the lower level system synthesis of reconfigurable computing system. The very nature that the system quality is distributively encoded as pheromone trails on the system representation makes it a promising model to handle dynamic changes required by the reconfigurable computing systems. As the system's computing characteristics change, it is natural to expect that the pheromone parameters accumulated over time will better reflect the dynamics of the system's behaviors, and thus, lead to quickly finding more effective configurations for the changed computing requirements. An application of this could be new ways for quickly calculating and deploying new FPGA placement and routing arrangement during the run time.

During our research work discussed above, we strongly felt the need of a better

application representation in the design process in order to effectively facilitate the system design/synthesis for the modern reconfigurable computing systems, which contain powerful hybrid architectures with multiple microprocessor cores, large reconfigurable logic arrays and distributed memory hierarchies. It is clear that traditional representations such as CDFG are not capable enough for optimizations that exploit fine and coarse grained parallelism. In our recent paper published on ERSA'2004 [39], we present an application representation based on the program dependence graph (PDG) incorporated with the static single-assignment (SSA) for synthesis to high performance reconfigurable devices. The PDG effectively describes control dependencies, while SSA yields precise data dependencies. When used together, these two representations provide a powerful, synthesizable form that exploits both fine and coarse grained parallelism. Our work showed that an intermediate representation based on PDG+SSA form supports a broad range of transformations and enables both coarse and fine grain parallelism. We described a method to synthesize this representation to a configurable logic array. Experimental results indicate that the PDG+SSA representation gives faster execution time using similar area when compared with commonly used CFG (Control Flow Graph) and PSSA (Predicted Static Single Assignment) forms.

Bibliography

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

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

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

[4] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lamand David L. Moore, Brian R. Murphy, and Constantine Sapuntzakis. *The Basic SUIF Programming Guide*. Computer Systems Laboratory, Stanford University, August 2000.

[5] Alex Aletà, Josep M. Codina, and Jesús Sánchez andAntonio González. Graph-Partitioning based Instruction Scheduling for ClusteredProcessors. In *Proceed-*

ings of the 34th Annual ACM/IEEE International Symposium onMicroarchitecture, 2001.

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

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

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

[9] Massimo Baleani, Frank Gennari, Yunjian Jiang, Yatish Pateand Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. HW/SW Partitioning and Code Generation of EmbeddedControl Applications on a Reconfigurable Architecture Platform. In *Proceedings of the Tenth International Symposium on Hardware/SoftwareCodesign*, 2002.

[10] Steve J. Beaty. Genetic algorithms versus tabu search for instruction scheduling. In *Proceedings of the International Conference on Artificial NeuralNets and Genetic Algorithms*, 1993.

[11] Steven J. Beaty. Genetic algorithms and instruction scheduling. In *Proceedings of the 24th annual international symposium on Microarchitecture*, 1991.

[12] David Bernstein, Michael Rodeh, and Izidor Gertner. On the Complexity of

Scheduling Problems for Parallel/PipelinedMachines. *IEEE Transactions on Computers*, 38(9):1308–13, September 1989.

[13] Eric Bonabeau, Macro Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, New York, NY, 1999.

[14] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan. Parallel fpga-based all-pairs shortest-paths in a directed graph. In *the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS06)*, April 2006.

[15] B. Bullnheimer, R. F. Hartl, and C. Strauss. A new rank based version of the ant system: A computational study. *Central European Journal for Operations Research and Economics*, 7(1):25–38, 1999.

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

[17] Raul Camposano. Path-based scheduling for synthesis. *IEEE Transaction on Computer-Aided Design*, 10(1):85–93, Janunary 1991.

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

[19] Samit Chaudhuri, Stephen A. Blythe, and Robert A. Walker. A solution method-

ology for exact design space exploration in a three-dimensional design space. *IEEE Trans. Very Large Scale Integr. Syst.*, 5(1):69–81, 1997.

[20] Jason Cong, Michail Romesis, and Min Xie. Optimality, scalability and stability study of partitioning and placement algorithms. In *ISPD '03: Proceedings of the 2003 international symposium on Physical design*, pages 88–94, New York, NY, USA, 2003. ACM Press.

[21] Jason Cong, Joseph R. Shinnerl, Min Xie, Tim Kong, and Xin Yuan. Large-scale circuit placement. *ACM Trans. Des. Autom. Electron. Syst.*, 10(2):389–430, 2005.

[22] D. Costa and A. Hertz. Ants can colour graphs. *Journal of the Operational Research Society*, 48:295–305, 1996.

[23] Achim Österling, Thomas Benner, Rolf Ernstand Dirk Herrmann, Thomas Scholz, and Wei Ye. *Hardware/Software Co-Design: Principles and Practice*, chapter The COSYMA System. Kluwer Academic Publishers, 1997.

[24] Andre DeHon. Very large scale spatial computing. In *UMC '02: Proceedings of the Third International Conference on Unconventional Models of Computation*, pages 27–36, London, UK, 2002. Springer-Verlag.

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

[26] Robert P. Dick and Niraj K. Jha. MOGAC: A Multiobjective Genetic Algorithm for the Co-Synthesis of Hardware-Software Embedded Systems. In *IEEE/ACM Conference on Computer Aided Design*, pages 522–529, 1997.

[27] Marco Dorigo and Luca Maria Gambardella. Ant Colony System: A Cooperative Learning Approach tothe Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[28] Marco Dorigo, Vittorio Maniezzo, and Alberto 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.

[29] R. Dutta, J. Roy, and R. Vemuri. Distributed design-space exploration for high-level synthesis systems. In *DAC '92*, pages 644–650, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[30] Stephen A. Edwards, Luciano Lavagno, Edward A. Lee, and AlbertoSangiovanni-Vincentelli. Design of Embedded Systems: Formal Models Validation, andSynthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[31] Petru Eles, Zebo Peng, and Alexa Doboli. System Level Hardware/Software Partitioning Based on SimulatedAnnealing and Tabu Search. *Design Automation for Embedded Systems*, 2(1):5–32, 1996.

[32] Rolf Ernst, Jorg Henkel, and Thomas Benner. Hardware/Software Cosynthesis for Microcontrollers. *IEEE Design and Test of Computers*, 10(4):64–75, December 1993.

[33] ExpressDFG. ExpressDFG benchmark web site. *http://express.ece.ucsb.edu/benchmark/*, 2006.

[34] Serge Fenet and Christine Solnon. Searching for maximum cliques with ant colony optimization. *3rd European Workshop on Evolutionary Computation in CombinatorialOptimization*, April 2003.

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

[36] L. M. Gambardella, E. D. Taillard, and G. Agazzi. *New Ideas in Optimization*, chapter A multiple ant colony system for vehicle routing problems with timewindows, pages 51–61. McGraw Hill, London, UK, 1999.

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

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

[39] Wenrui Gong, Gang Wang, and Ryan Kastner. A high performance application representation for reconfigurablesystems. *International Conference on Engineering of Reconfigurable Systemsand Algorithms, ERSA'04*, June 2004.

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

[41] Martin Grajcar. Genetic List Scheduling Algorithm for Scheduling and Allocationon a Loosely Coupled Heterogeneous Multiprocessor System. In *Proceedings of the 36th ACM/IEEE Conference on Design AutomationConference*, 1999.

[42] Rajesh K. Gupta and Giovanni De Micheli. Constrained Software Generation for Hardware-Software systems. In *Proceedings of the Third International Workshop on Hardware/SoftwareCodesign*, 1994.

[43] Walter J. Gutjahr. A graph-based ant system and its convergence. *Future Gener. Comput. Syst.*, 16(9):873–888, 2000.

[44] Walter J. Gutjahr. Aco algorithms with guaranteed convergence to the optimal solution. *Inf. Process. Lett.*, 82(3):145–153, 2002.

[45] Walter J. Gutjahr. A generalized convergence result for the graph-based ant system metaheuristic. *Probability in the Engineering and Informational Sciences*, 17:545 – 569, 2003.

[46] Walter J. Gutjahrs. A converging aco algorithm for stochastic combinatorial optimization. In *SAGA*, pages 10–25, 2003.

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

[48] M. Heijligers and J. Jess. High-level synthesis scheduling and allocation using genetic algorithms based on constructive topological scheduling techniques. In *International Conference on Evolutionary Computation*, pages 56–61, Perth, Australia, 1995.

[49] M. J. M. Heijligers, L. J. M. Cluitmans, and J. A. G. Jess. High-level synthesis scheduling and allocation using genetic algorithms. page 11, 1995.

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

[51] B. Jeong, S. Yoo, and K. Choi. Exploiting early partial reconfiguration of runtime reconfigurableFPGAs in embedded systems design. *7th ACM/SIGDA Int. Symposium on Field Programmable Gate Arrays*, page 247, 1999.

[52] Asawaree Kalavade and Edward A. Lee. A Global Criticality/Local Phase Driven Algorithm forthe Constrained Hardware/Software Partitioning Problem. In *codes94*, 1994.

[53] Ryan Kastner. *Synthesis Techniques and Optimizations for ReconfigurableSystems*. PhD thesis, University of California at Los Angeles, 2002.

[54] Ken Kennedy and Randy Allen. *Optimizing Compilers for Modern Architectures: A Dependence-basedApproach*. Morgan Kaufmann, 2001.

[55] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[56] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

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

[58] Shekhar Kopuri and Nazanin Mansouri. Enhancing scheduling solutions through ant colony optimization. In *International Symposium on Circuits and Systems (ISCAS'04)*, Vancouver, Canada, May 2004. IEEE.

[59] Chunho Lee, Miodrag Potkonjak, and William 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.

[60] Jiahn-Hung Lee, Yu-Chin Hsu, and Youn-Long Lin. A new integer linear programming formulation for the schedulingproblem in data path synthesis. In *Proceedings of ICCAD-89*, pages 20–23, Santa Clara, CA, USA, Nov 1989.

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

[62] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr andUday Kurkure, and Jon Stockwood. Hardware-Software Co-Design of Embedded Reconfig-urableArchitectures. In *Proceedings of the 37th Conference on Design Automation*, 2000.

[63] Youn-Long Lin. Recent developments in high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 2(1):2–21, 1997.

[64] Qinghua Liu and Malgorzata Marek-Sadowska. A study of netlist structure and placement efficiency. In *ISPD '04: Proceedings of the 2004 international symposium on Physical design*, pages 198–203, New York, NY, USA, 2004. ACM Press.

[65] J Madsen, J Grode, P V Knudsen, M E Petersen, and A Haxthausen. LYCOS: the Lyngby Co-Synthesis System. *Design Automation for Embedded Systems*, 2(2):125–63, March 1997.

[66] Michael C. McFarland, Alice C. Parker, and Raul Camposano. The high-level synthesis of digital systems. In *Proceedings of the IEEE*, volume 78, pages 301–318, Feb 1990.

[67] D. McGrath. Gartner dataquest analyst gives asic, fpga markets clean bill of health. *EE Times*, June 2005.

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

[69] Seda Ogrenci Memik, E. Bozorgzadeh, Ryan Kastner, and MajidSarrafzadeh. A super-scheduler for embedded reconfigurable systems. In *IEEE/ACM International Conference on Computer-Aided Design*, 2001.

[70] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs (2nd, extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1994.

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

[72] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[73] Gordon E. Moore. Cramming more components onto integrated circuits. pages 56–59, 2000.

[74] Ralf Niemann and Peter Marewedel. An Algorithm for Hardware/Software Partitioning Using MixedInteger Linear Programming. *Design Automation for Embedded Systems*, 2(2):125–63, March 1997.

[75] Maurizio Palesi and Tony Givargis. Multi-Objective Design Space Exploration Using GeneticAlgorithms. In *Proceedings of the Tenth International Symposium on Hardware/SoftwareCodesign*, 2002.

[76] In-Cheol Park and Chong-Min Kyung. Fast and near optimal scheduling in automatic data path synthesis. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 680–685, New York, NY, USA, 1991. ACM Press.

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

[78] P. G. Paulin and J. P. Knight. Force-directed scheduling in automatic data path synthesis. In *24th ACM/IEEE Conference Proceedings on Design Automation Conference*, 1987.

[79] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of asic's. *IEEE Trans. Computer-Aided Design*, 8:661–679, 1989.

[80] P. Poplavko, C.A.J. van Eijk, and T. Basten. Constraint analysis and heuristic scheduling methods. In *Proceedings of 11th. Workshop on Circuits, Systems and Signal Processing(ProRISC2000)*, pages 447–453, 2000.

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

[82] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, New York, NY, 2002.

[83] Ruud Schoonderwoerd, Owen Holland, Janet Bruten, and Leon Rothkrantz. Ant-based load balancing in telecommunications networks. *Adaptive Behavior*, 5:169–207, 1996.

[84] J. M. J. Schutten. List scheduling revisited. *Operation Research Letter*, 18:167–170, 1996.

[85] Semiconductor Industry Association. National Technology Roadmap for Semiconductors. 2003.

[86] Alok Sharma and Rajiv Jain. Insyn: Integrated scheduling for dsp applications. In *DAC*, pages 349–354, 1993.

[87] Alena Shmygelska and Holger H. Hoos. An ant colony optimisation algorithm for the 2d and 3d hydrophobic polar protein folding problem. *BMC Bioinformatics*, 6, 2005.

[88] James E. Smith. Dynamic instruction scheduling and the astronautics ZS-1. *IEEE Computer*, 22(7):21–35, 1989.

[89] Michael D. Smith and Glenn Holloway. *An Introduction to Machine SUIF and Its Portable Librariesfor Analysis and Optimization*. Division of Engineering and Applied Sciences, Harvard University, July 2002.

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

[91] T. Stützle and M. Dorigo. A short convergence proof for a class of ACO algorithms. *IEEE Transactions on Evolutionary Computation*, 6(4):358–365, 2002.

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

[93] Philip H. Sweany and Steve J. Beaty. Instruction scheduling using simulated annealing. In *Proceedings of 3rd International Conference on Massively ParallelComputing Systems*, 1998.

[94] Haluk Topcuouglu, Salim Hariri, and Min you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002.

[95] Frank Vahid, Jie Gong, and Daniel D. Gajski. A Binary-Constraint Search Algorithm for Minimizing Hardwareduring Hardware/Software Partitioning. In *Proceedings of the conference on European design automation conference*, 1994.

[96] Frank Vahid and THUY Dm LE. Extending the Kernighan/Lin Heuristic for Hardware and SoftwareFunctional Partitioning. *Design Automation for Embedded Systems*, 2(2):237–61, March 1997.

[97] W. F. J. Verhaegh, E. H. L. Aarts, J. H. M. Korst, and P. E. R. Lippens. Improved force-directed scheduling. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 430–435, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[98] W. F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aarts, J. H. M. Korst, A. van der Werf, and J. L. van Meerbergen. Efficiency improvements for force-directed scheduling. In *ICCAD '92: Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 286–291, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[99] Gang Wang, Wenrui Gong, Brian DeRenzi, and Ryan Kastner. Ant colony optimizations for resource and timing constrained operation scheduling. *IEEE Transaction on Computer-Aided Design*, 26(6):1010–1029, 2006.

[100] Gang Wang, Wenrui Gong, Brian DeRenzi, and Ryan Kastner. Design space exploration using time and resource duality with the ant colony optimization. In

DAC '06: Proceedings of the 43rd annual conference on Design automation, pages 451–454, New York, NY, USA, 2006. ACM Press.

[101] Gang Wang, Wenrui Gong, Brian DeRenzi, and Ryan Kastner. Exploring time/resource tradeoffs by solving dual scheduling problems with the ant colony optimization. *ACM Transactions on Design Automation of Electronic Systems (TODAES) (to appear)*, 2007.

[102] Gang Wang, Wenrui Gong, and Ryan Kastner. A New Approach for Task Level Computational ResourceBi-partitioning. *15th International Conference on Parallel and Distributed Computingand Systems*, 1(1):439–444, November 2003.

[103] Gang Wang, Wenrui Gong, and Ryan Kastner. System level partitioning for programmable platforms using the antcolony optimization. *13th International Workshop on Logic and Synthesis, IWLS'04*, June 2004.

[104] Gang Wang, Wenrui Gong, and Ryan Kastner. Instruction scheduling using MAX-MIN ant optimization. In *15th ACM Great Lakes Symposium on VLSI, GLSVLSI'2005*, April 2005.

[105] Gang Wang, Wenrui Gong, and Ryan Kastner. Application partitioning on programmable platforms using the ant colony optimization. *Journal of Embedded Computing*, 2(1):119–136, 2006.

[106] Theerayod Wiangtong, Peter Y. K. Cheung, and Wayne Luk. Comparing Three

Heuristic Search Methods for FunctionalPartitioning in Hardware-Software Codesign. *Design Automation for Embedded Systems*, 6(4):425–49, July 2002.

[107] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming languagedesign and implementation*, 2000.

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