ABSTRACT

Title of dissertation: System Synthesis for Embedded Multiprocessors

Vida Kianzad, Doctor of Philosophy, 2006

Dissertation directed by: Professor Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering

Modern embedded systems must increasingly accommodate dynamically changing operating environments, high computational requirements, flexibility (e.g., for the emergence of new standards and services), and tight time-to-market windows. Such trends and the ever-increasing design complexity of embedded systems have challenged designers to raise the level of abstraction and replace traditional ad-hoc approaches with more efficient synthesis techniques. Additionally, since embedded multiprocessor systems are typically designed as final implementations for *dedicated* functions, modifications to embedded system implementations are rare, and this allows embedded system designers to spend significantly larger amounts of time to optimize the architecture and the employed software. This dissertation presents several system-level synthesis algorithms that employ thorough and hence time-intensive optimization techniques (e.g. evolutionary algorithms) that allow the designer to explore a significantly larger part of the design space. It looks at critical issues that are at the core of the synthesis process — selecting the architecture, partitioning the functionality over the components of the architecture, and scheduling activities such that design constraints and optimization objectives are satisfied.

More specifically for the scheduling step, a new solution to the two-step (clustering and cluster-merging) multiprocessor scheduling problem is proposed. For the first step or pre-processing step of clustering a simple yet highly efficient genetic algorithm is proposed. Several techniques for the second step of merging or cluster scheduling are proposed and finally a complete two-step effective solution is presented. Also, a randomization technique is applied to existing deterministic techniques to extend these techniques so that they can utilize arbitrary increases in available optimization time. This novel framework for extending deterministic algorithms in our context allows for accurate and fair comparison of our techniques against the state of the art.

To further generalize the proposed clustering-based scheduling approach, a complementary two-step multiprocessor scheduling approach for heterogeneous multiprocessor systems is presented. This work is amongst the first works that formally studies the application of clustering to heterogeneous system scheduling. Several techniques are proposed and compared and conclusive results are presented.

A modular system-level synthesis framework is then proposed. It synthesizes multi-mode, multi-task embedded systems under a number of hard constraints; optimizes a comprehensive set of objectives; and provides a set of alternative trade-off points in a given multi-objective design evaluation space. An extension of the framework is proposed to better address dynamic voltage scaling, memory optimization, and efficient mappings of applications onto dynamically reconfigurable hardware.

Additionally, to address the increasing importance of managing power consumption for embedded systems and the potential savings during the scheduling step of synthesis, an integrated framework for energy-driven scheduling onto embedded multiprocessor

systems is proposed. It employs a solution representation (for the GA-based scheduler) that encodes both task assignment and ordering into a single chromosome and hence significantly reduces the search space and problem complexity. It is shown that a task assignment and scheduling that result in better performance (less execution time) do not necessarily save power, and hence, integrating task scheduling and voltage scheduling is crucial for fully exploiting the energy-saving potential of an embedded multiprocessor implementation.

System Synthesis for Embedded Multiprocessors

by

Vida Kianzad

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Prof. Shuvra S. Bhattacharyya for his invaluable guidance and support during the past seven years. Shuvra's vision and personality make him a great advisor one could ever wish for. He taught me how to do research and gave me enormous freedom to select the final direction of this work. His extraordinary editorial skills also effectively helped me to improve my technical writing skills.

I would like to thank my committee members, Prof. Qu, Prof. Barua, Prof. Hollingsworth and Prof. Varshney for their insightful feedbacks. I also gratefully acknowledge Prof. Qu's help and advice on the final technical chapter of this thesis. I would also like to thank Prof. Barua for his helpful suggestions on improving my presentation skills.

I am thankful to the various faculty from whom I have learnt numerous lessons. Particularly, I am sincerely grateful to Prof. André Tits, the former associate chair of graduate studies, and Prof. Donald Yeung who despite their busy schedules, have always been readily available for help, advice, or simply a word of encouragement.

Over the past seven years, I have had an excellent groups of lab-mates and friends to interact with, that I would like to acknowledge. Numerous early discussions with Nitin Chandrachoodan, a truly bright individual helped keep me on right path. Neal Bambha, an outstanding senior graduate student in the lab helped me with my numerous technical questions. Ming-Yung Ko has been a wonderful friend and officemate, my walking partner during endless summer days that we spent as interns together, who with no doubt is one of the most helpful individuals I will ever meet. I owe a big thank you to Mainak Sen for proofreading my thesis. I am also grateful to Ming-Yung Ko, Dong-Ik Ko, Sankalita Saha, Shahrooz Shahparnia, Mainak Sen, and Ankush Varma for a number of insightful discussions. I would also like to thank other friends in the lab, Celine Badr, Bishnupriya Bhattacharya, Ivan Corretjer, Chia-Jui Hsu, Mukul Khandelia, Sumit Lohani, Chung-Ching Shen, Sadagopan Srinivasan and Lin Yuan.

I would like to acknowledge my many great friends who made my UMCP experience one to always remember; Nasim Vakili, Mehdi Kalantari, Anna Secka, Radost Koleva, Gelareh Taban, Behnam Neekzad and Farangis Soroushian. I specially would like to thank Afshin Sepehri for being a truly outstanding computer scientist and teacher whose help, advice and friendship has helped me through many technical challenges that I faced during my graduate studies and completing this thesis.

My parents, Reza and Jaleh, have been an endless source of encouragement and love. They always inspired and motivated me to do my best and to never give up. My sister Aida, who is everything that is good and beautiful in this world, has been a never ending source of love and moral support for me. My best friend and husband Pouria has always been there for me through the worst and best time with patience, love and understanding and has never given up on me. My deepest gratitude and love goes to these four.

To my wonderful parents and husband, I gratefully dedicate this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

An embedded computing system is a computer that is a part of a larger system and is meant to help implement the system functionality. Embedded systems are everywhere — homes, offices, automobiles, manufacturing systems, hospitals, industrial plants and consumer electronics (see Figure 1.1.) Today most people use more embedded computers (or systems) in their daily lives (e.g. pagers, telephones, cars, etc) than traditional computers (e.g. PCs). For example the total shipment of microprocessor units and micro control units in 1997 was over 4.4 billion units, and of this about 98% related to embedded systems applications [118].

Depending on the core functionality, embedded computing systems can be classified as control-oriented or data-stream processing-oriented. An example of the control-oriented functionality of embedded systems can be seen in automobiles where the system operation (brake, fuel-injection, AC, etc,) is controlled by the embedded processors based on the various inputs and signals they read from different sensors (e.g the BMW 7 series has more than 80 embedded control units.) Data-stream processing, or digital signal processing (DSP) aspect of embedded systems is evident in cellular phones, modems, multi-media devices, radar application, etc. In this thesis we are interested in developing tools that facilitate and optimize the design of the latter class of embedded systems, i.e the signal processing (DSP) functionality of the embedded system.

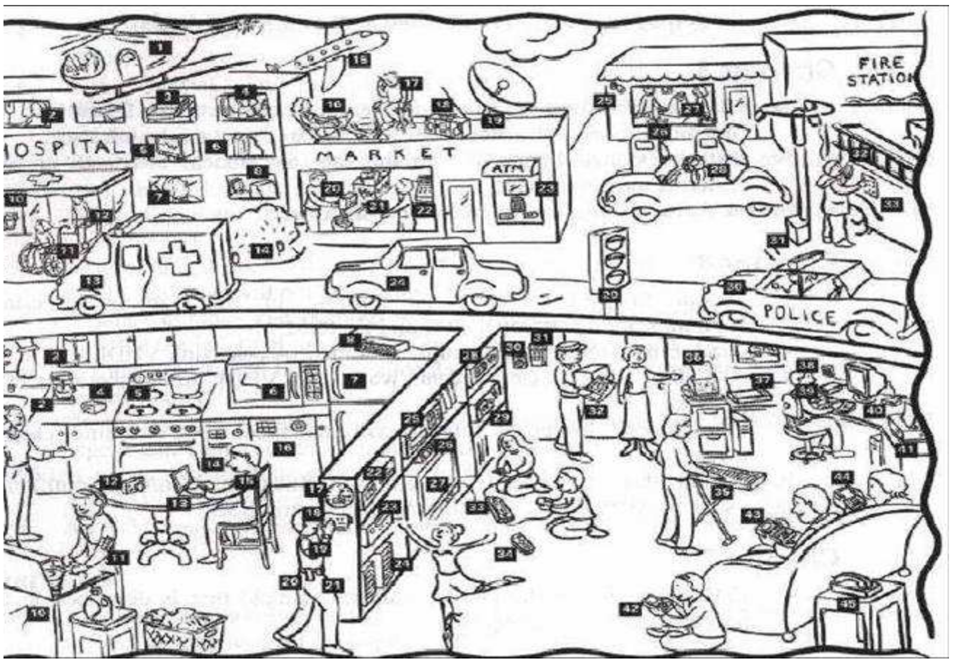


Figure 1.1: Embedded Systems are everywhere.

1.1 Multiprocessor Embedded Systems

Design of embedded systems requires the implementation of a set of functionalities that satisfy a number of constraints such as cost, power dissipation, performance, etc. In recent years, not only the number of functionalities that an embedded system can perform has increased but the functionalities have grown in complexity as well. More specifically, today's real time image and signal processing applications are characterized by an increase in computational requirements and algorithm complexity. The real-time realization of these applications often calls for heterogeneous architectures, providing extremely high computational and throughput rates, which can only be achieved by aggressive application of parallel processing as provided by multiprocessor systems. These systems consist of dedicated hardware (e.g. ASICs, FPGAs) and/or programmable processor elements to perform composite schemes of DSP algorithms with filtering, coding, block matching, etc. The use of multiprocessor systems could be either in forms of distributed systems or systems-on-chip. The former trend i.e. connecting several processing-elements (specially microprocessors) together and performing a complex task collaboratively has been evident in many system companies. The latter trend has recently become popular and feasible due to the progress of semiconductor industry and feature size reduction that has made it possible for multiple processing elements to be placed on a single die. However, regardless of the hardware architecture or software programming paradigm used, there are fundamental difficulties that arise when trying to make processors cooperate on a common application. The difficulties involved are interaction of several complex factors including scheduling, inter-processor communication, iterative execution, etc. Addressing any one of these factors in isolation is itself typically intractable in any optimal sense. Such (ever-increasing) complexities added to the constantly changing requirements of embedded systems have made it very difficult for designers to predict an accurate development time while meeting all the requirements and delivering an error free system. A recent example of problems encountered in developing distributed embedded systems in automotive industry is the recall of 1.3 million Mercedes Benz in March 2005 due to software bugs in the embedded control unit that was in charge of optimizing battery usage and the braking systems [118]. Most of today's embedded systems are designed manually with an ad hoc approach that is heavily based on earlier experience with similar products and on manual design. Design automation has the potential to help designers keep pace with increasing problem complexity. This thesis explores algorithms and techniques to develop such automated tools for multiprocessor embedded system synthesis.

1.2 Embedded Systems Design Automation

Advances in the integrated circuit technology has made it possible to put an order of millions of transistors on a single chip and fit more functionality into smaller units. With such increased density it is clear that the design of these chips and complex systems built upon them is only possible via use of advanced design techniques and computer-aided design (CAD) tools. There are several design methods and CAD tools that are widely used for specification, simulation, verification, and to some extent synthesis. In particular, tools that deal with lower level of abstractions (i.e. physical or logical) such as layout tools and logic synthesizer have been well studied and developed. Ideally, one would like to develop design tools and methodologies that involve the entire design flow from system-level description to actual hardware implementation in a single design tool or environment. The advantage of such a system is that it allows a single designer to get a better overall view of the system being designed, and opens up the possibility of much better overall designs. A very important additional advantage is that the overall "Time-to-Market" of the design can be greatly reduced, and this is a crucial factor in determining the economic viability of any system. Another factor, as mentioned in [115], is the fact that more power-efficient designs can be made by making appropriate decisions at a higher level of the design, than by concentrating on circuit level improvements. The ultimate goal is to have a single tool that can take abstract designs and go through the entire process of system design automatically. In the near term, it is equally or more important to consider techniques that aid the designer by exploring large parts of the design space automatically, and presenting a set of useful designs to human designers, who can

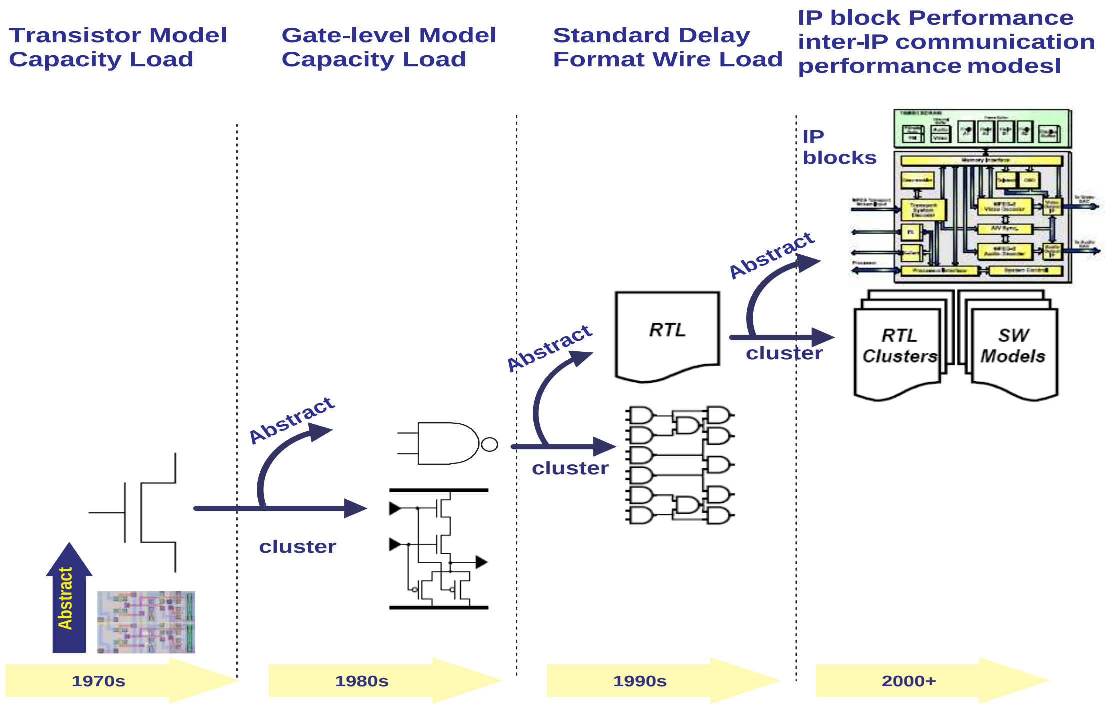


Figure 1.2: Raising the Level of Abstraction [117].

then use their experience to choose a suitable candidate. As shown in Figure 1.2 design automation has followed the historical trend from automation of low-level stages of the design process toward automation of increasingly high-level stages of the design process. More specifically, as presented in [117] the following design automation methodology and tool development efforts can be identified in the electronic design automation (EDA) history:

- 1964 – 1978 — The foundations of EDA was laid by the industry pioneers. The fundamental contributions can be grouped in the following five areas: circuit simulation; logic simulation and testing; MOS timing simulation; wire routing; and regular arrays.

- 1979 – 1993 — It is fair to say that the EDA field exploded in all its aspects in this period. The main contributions from this age cluster into several distinct topics: Verification and testing, Layout, Logic synthesis, Hardware description languages, Hardware acceleration, High-level design.

- 1993 – 2002 — Due to emergence of web and its application, the field of EDA got less attention in this period. Additionally, the key EDA problems were at higher abstraction levels, where the problems encompass wider ranges of decisions and hence generally harder to formulate clearly and (when formulated) often more complex to solve. Hence EDA had less commercial impact, or less influence on the actual design process. At the same time, the semiconductor sector continued to drive technology along the lines of Moore's law, increasing the technical challenges to EDA. In this period system on chip (SoC) also became a reality. It is hard to clearly point out the fundamental contributions of this period, but some of the important topics addressed in this period are as follows: Physical verification (due to submicron range challenges), self-test as cost-efficient test methods, asynchronous design methods and the associated synthesis problem, hardware-software co-design and embedded software.

The future of EDA is towards developing more system-level tools and methodologies. The importance of such development was pointed out by Alberto Sangiovanni-Vincentelli in his 40th Design Automation Conference Key Note Address [117]:

"*High- or system-level design is a bridge to the future. We all agree that raising the level of abstraction is essential to increasing design productivity by orders of magnitude. I am indeed very passionate about this field, and I believe our future rides on the success* *of design methodologies and tools in this area. This work started almost in parallel with logic synthesis, and researchers developed several commercial tools. Despite these facts, the design community has not widely accepted this approach; much work remains to be done."*

The goal of this thesis is to provide such automated tools that facilitate the synthesis process at the system level.

1.3 Contributions of this Thesis

In this research, we address the key trends in the synthesis of implementations for embedded multiprocessors and present algorithms for system-level synthesis of embedded systems. The system-level synthesis problem is constituted of selection of the target architecture (resource allocation), mapping of the algorithm onto the selected architecture (resource binding) and scheduling. Considering the importance of mapping and scheduling in the quality of the final solution(s) we address these problems first separately and then in the context of system-level synthesis. In the context of mapping and scheduling, one of the important issues that we address is the increasing importance of managing inter-processor communication in an efficient manner. This importance is due to the increasing interest among embedded system architects in innovative communication architectures, such as those involving optical interconnection technologies, and hybrid electro-optical structures [131]. Effective experimentation with unconventional architectures requires adequate design tools that can exploit such architectures. We also address the increased compile time tolerance in embedded system design. This increased-time results because embedded multiprocessor systems are typically designed as final implementations for *dedicated* functions; modifications to embedded system implementations are rare, and this allows embedded system design tools to employ more thorough, time-intensive optimization techniques [97]. We show that our proposed algorithms, provide solutions that match or outperform existing techniques.

Existing techniques for the embedded system-level synthesis make many simplifying assumptions; for example they only consider a subset of embedded systems classes (single mode, single graph, non-iterative, etc.) or optimize a small set of objectives (e.g. power and time), however our work provides a modular multi-objective multi-constraint framework that synthesizes for a superset of different embedded system classes. More specific contributions are outlined below:

1.3.1 Two-step Embedded Multiprocessor Scheduling

In Chapter 3 we illustrate the effectiveness of the two-phase decomposition of multiprocessor scheduling into clustering and cluster-scheduling or merging and mapping task graphs onto embedded multiprocessor systems. Clustering is a pre-processing step that is applied to constrain the remaining steps of synthesis, especially scheduling, so that they can focus on strategic processor assignments. We provide a novel solution to the clustering step and a framework for comparing our proposed solution against the other leading techniques.

1.3.2 Clustering-based Heterogeneous Multiprocessor Scheduling

The concept of clustering has been widely applied to various applications and research problems such as parallel processing, load balancing and partitioning. Clustering is also often used as a front-end to multiprocessor system synthesis tools to constrain the remaining steps of synthesis, especially scheduling, so that they can focus on strategic processor assignments. However, application of clustering to heterogeneous systems has not been studied well and the existing studies are applicable to limited scenarios. In Chapter 4 we propose the first comprehensive studies for application of clustering for heterogeneous multiprocessor systems and provide a strong case for application of our clustering technique as a pre-processing step for any system-level synthesis application.

1.3.3 Multi-mode multi-task Embedded Systems Synthesis

The final goal of system-level synthesis is to find an implementation of the system that satisfies a number of constraints. Due to the complex nature of this problem, presence of multiple constraints and optimization of several objectives, probabilistic search techniques (as will be discussed in the next chapter) seem to be most efficient in searching the vast solution space and finding the Pareto-optimal solutions. In Chapter 5, we propose a modular co-synthesis framework called CHARMED that synthesizes multi-mode, multi-task embedded systems under a number of hard constraints; optimizes a comprehensive set of objectives; and provides a set of alternative trade-off points, generally known as Pareto-optimal solutions. Our framework allows the designer to independently configure each dimension of the design evaluation space as an optimization objective (to be minimized or maximized) or as a constraint (to be satisfied). Additionally to the best of our knowledge CHARMED is the first multi-objective EA to handle multiple constraints as well. We consider two approaches to system-level synthesis that can effectively implement multiple system-level optimizations such as dynamic voltage scaling, dynamic reconfiguration of FPGAs, etc. Furthermore, we propose a pre-processing step (clustering) to the synthesis to modify the to-be-implemented embedded system to provide a better input to the synthesis algorithm.

1.3.4 Combined Assignment, Scheduling and Power Management Techniques

For multiprocessor embedded systems, the technique of dynamic voltage scaling (DVS) can be applied to scheduled applications (task graphs) for energy reduction. DVS utilizes slack in the schedule to slow down processes and save energy. Therefore, it is generally believed that the maximal energy saving is achieved on a schedule with the minimum parallel-time (completion time), or equivalently the maximal slack. Most current approaches treat task assignment, scheduling, and DVS separately. In Chapter 6, we present a framework called CASPER (Combined Assignment, Scheduling, and PowER-management) that challenges this common belief by integrating task scheduling and DVS under a single iterative optimization loop via a genetic algorithm. Through extensive experiments we validate the energy efficiency of our proposed integrated framework or CASPER. The framework targets both homogeneous and heterogeneous multiprocessor embedded systems. Furthermore, most of the optimization algorithms for similar problems in the literature always start from an arbitrary point (solution) in the solution space and take constructive or iterative steps to refine that solution. Hence, most of these algorithms end up getting trapped in local minima. In chapter 6 we use a re-calibration or refinement step to combine the power of genetic algorithms and local search heuristics to find better solutions to the energy efficient scheduling problem.

1.4 Outline of Thesis

The remainder of this thesis is organized as follows: Chapter 2 presents an overview of the system level synthesis problem and identifies difficulties associated with current techniques for design space exploration. Chapter 3 looks at the problem of two-step multiprocessor scheduling for a fully-connected network of homogeneous processors and presents techniques and results that show the effectiveness of our proposed techniques. In Chapter 4 we present a clustering-oriented solution for the heterogeneous multiprocessor system scheduling problem. Chapter 5 looks at a multi-objective evolutionary technique for synthesis of architectures with different costs and constraints. Chapter 6, presents an alternate solution to the system-level synthesis (SLS) problem where the emphasis is on power optimization. Finally, in Chapter 7, we conclude this thesis with a summary of the work and discuss directions for related future work.

Chapter 2

System Synthesis: Definitions and Assumptions

In the context of EDA, we refer to *Synthesis* as the process of converting a behavioral representation of a design into a structural one [46] [40]. The synthesis process consists of several stages that are preceded by one another and are performed at different levels of abstractions. Different stages of the synthesis steps can be summarized as follows:

- *System level* — Accepts an input specification in the form of communicating concurrent processes. The synthesis task is to generate the general system structure defined by processors, ASICs, buses, etc. System-level synthesis operates at the highest level of abstraction where fundamental decisions are taken which have great influence on the structure, cost and performance of the final product.

- *High level* — Accepts an input description in the form of behavioral description which captures the functionality of the designed system and produces an RT-level implementation.

- *Logic level* — Accepts Boolean equations or some kind of finite state machines and produces gate-level netlists.

- *Physical level* — Accepts a gate-level netlist and produces the final implementation of the design in a given technology.

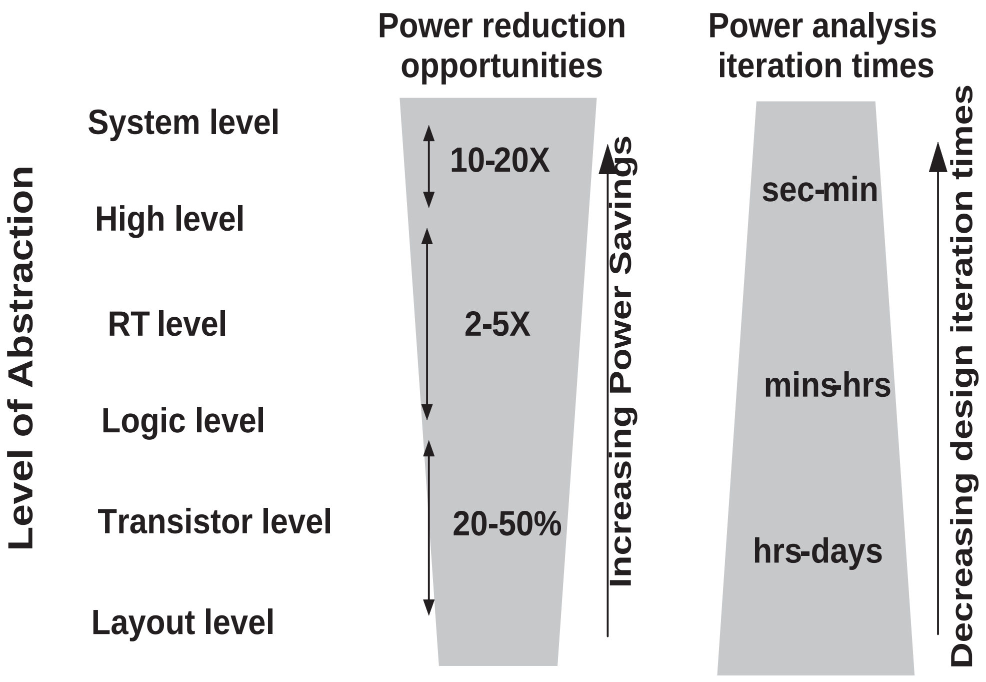


Figure 2.1: Benefits of high-level power analysis and optimization [115][36].

During the system-level synthesis most decisions regarding the system architecture are made that directly effect the system performance, cost and area and hence there is a great potential for a more efficient design at that level. Additionally by employing system-level scheduling and optimization techniques one has better opportunities to optimize the power and to offer a better solution under given constraints (see Figure 2.1). Consequently, in this thesis we are most interested in the design flow at the system level and will concentrate on the system-level synthesis.

2.1 System-Level Synthesis

System synthesis starts with the system-level synthesis. A system-level synthesis tool takes the initial system specification represented as a set of interacting processes and a set of design constraints as an input and generates the behavioral modules of the

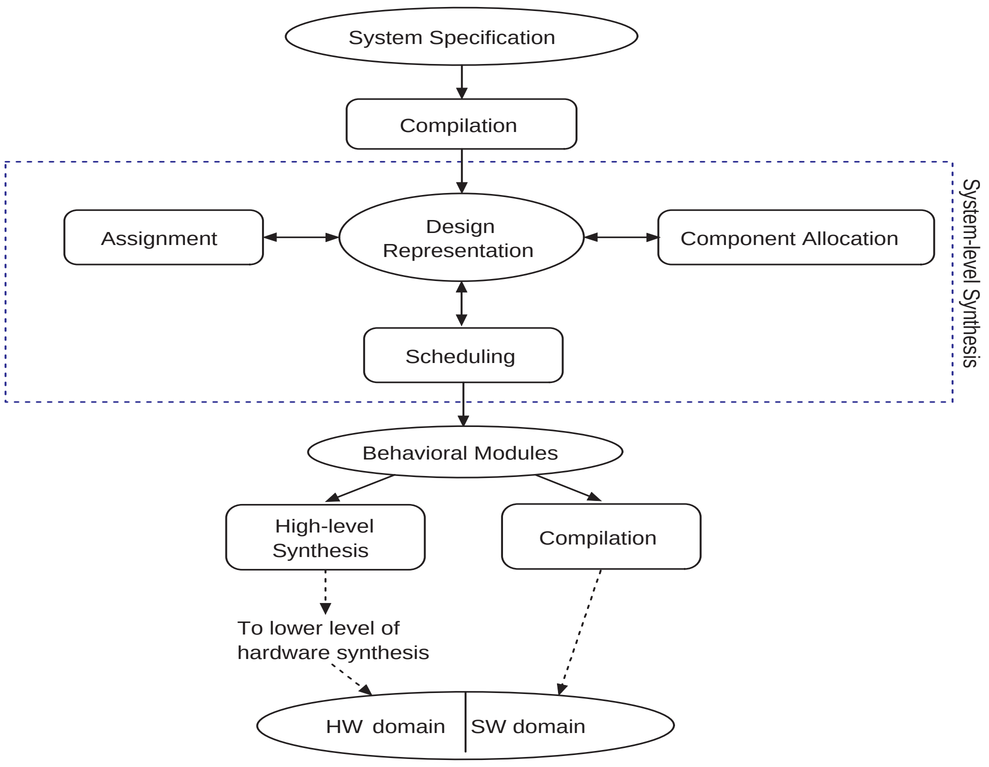


Figure 2.2: Design Flow with system-level synthesis [40].

systems and their assignment onto system components. An overview of the design flow with system-level synthesis is given in Figure 2.2.

As it can be seen in Figure 2.2 the first step of the design flow is the system specification. The selection of a suitable representation for system specification is a very important aspect of the design methodology and is still an active area of research. The representations can be i) software-oriented, e.g. ANSI-C, C++, system C [136], Java, etc., ii) hardware-oriented, e.g. Verilog [58], VHDL [57], ESTEREL [15], or iii) graphical based, such as in state machines (Statecharts [52]), Petri nets [103], dataflow graphs [29], synchronous dataflow graphs [85], etc. Depending on the application characteristics (e.g. control oriented, data driven , etc) one could decide on the appropriate representation that best represents the employed class of applications. In this work, we focus on embedded signal processing applications and *dataflow models* (specially synchronous dataflow graphs) have proven to be very useful for specifying such applications. Dataflow graphs are useful models of computations (MoCs) for signal processing systems since they capture the intuitive expressivity of block diagrams, flow charts, and signal flow graphs, while providing the formal semantics needed for system design and analysis tools [132]. For the rest of this thesis, we will assume that the (embedded system) application is provided as a dataflow graph.

Once the system to be implemented is specified using one of the mentioned representations, the system architecture and various computation/communication resources have to be *allocated*, the system has to be *assigned* to these resources and the execution of the system on these resources has to be *scheduled*. These three steps are the main steps of the system-level synthesis and can be described as follows:

1. Allocation: Determines the quantity of each type of resource, i.e. processing elements (PEs), communication resources (CRs), etc. to implement the system. The PEs can be microprocessors, micro-controllers, DSPs, ASIPs, ASICs, and FPGAs.

2. Assignment: Selects a suitable resource (PE or CR) to execute a computational task or a communication event.

3. Scheduling: Determines the ordering of the tasks (communication events) that are assigned to each PE (CR), and determines precisely when each task/communication event should commence execution.

These steps are interdependent and they should ideally be performed simultaneously and not separately in a pre-specified order. However, allocation, assignment and scheduling are each known to be NP-complete for distributed systems [47] and it is not computationally conceivable to solve these problems optimally at the same time. Hence, to keep the complexity of the problems manageable, most synthesis techniques address these steps separately but iterate several times to improve the quality of the solutions and allow some indirect interactions between the three stages.

In this thesis, we consider the NP-complete problem of scheduling separately as well as in the context of system-level synthesis. Studying the literature one could find several types of scheduling problems based on the following characteristics [37]:

- Hard deadline/Soft deadline — Tasks to be scheduled may each have hard or soft deadlines. A task with a hard-deadline must finish by the given time (its deadline) or the schedule is invalid. A task with a soft-deadline can finish after the given time (its deadline) without making the schedule invalid.

- Unconstrained resources/Constrained resources — A scheduler with no constraints on the number of resources can utilize as many resource as will benefit the schedule while a scheduler with constraint on the number of resources can only use a limited set of available resources.

- Multi-processors/Single processor — In the multi-processor case, tasks are distributed among several resources (processors) to run on while in the single processor case, all tasks run on a single resource. It is obvious that in the multi-processor scenario the scheduler has to perform the assignment as well as ordering and timing which makes the problem more complex.

- Heterogeneous processors/Homogeneous processors — Heterogeneous processors have different types and properties, i.e. tasks can have different execution times on different processors. Homogeneous processors have similar execution times.

- Presence of inter-processor communication (IPC)/Absence of inter-processor communication — A scheduler in the presence of IPC has to take the data transmission time into account. Such consideration increases the problem complexity as the scheduler has to generate schedules for communication resources as well as computational resources. In the absence of IPC, the scheduler assumes zero time for data transfer between two processors.

- Dependent tasks/Independent tasks — With independent tasks no previous execution sequence is imposed while for the dependent tasks there exist precedence constraints i.e. tasks have to follow some (weak) execution ordering.

- Single iteration/Iterative — In the single iteration, it is assumed that the task set is executed only once, while in the iterative execution the task set repeats more than once.

- Periodic/Aperiodic — In periodic execution, the application executes at a given period while in an aperiodic execution, the task set repeats irregularly.

- Non-preemptive/Preemptive – In a non-preemptive scheduling, once a task starts execution, it runs to its completion. In a preemptive scheduling, a running task can be interrupted at any time instance by the scheduler and be replaced by another task and resume execution from the step at which it was interrupted.

In this thesis we address several instances of the scheduling problem. In all the instances we consider resource-constrained, multiprocessor, IPC conscious , precedence constraints, and non-preemptive scheduling. While addressing the scheduling problem independently (as a stand-alone problem), we assume single iteration and soft-deadlines. We consider both instances of homogeneous and heterogeneous processors. In the context of system-level synthesis, we additionally assume the presence of both hard and soft deadline tasks, heterogeneous processors and periodic execution. More specifically, in the earlier chapters of this thesis we present some efficient solutions for the scheduling problem based on the clustering techniques and use the presented techniques and results in the later chapters when we present our solution for the system-level synthesis.

After the three steps of allocation, assignment and scheduling are carried out, the system performance and costs (e.g. area, power consumption, price, etc.) are evaluated and depending on the constraints and optimization criteria either the synthesis terminates with an acceptable system implementation or iterates and revise some of the decision to either meet the constraints or improve the system performance or costs.

Some closely related terms with system-level synthesis are *hardware-software co-design*, *hardware-software co-synthesis* and *hardware-software partitioning*. There are many definitions of these terms; some researchers distinguish between them and some use them interchangeably. We distinguish between these terms and use the following definitions for these terms throughout these thesis when applicable: Hardware-software co-design is the concurrent design of the hardware and software portions of a computer system. Hardware-software co-synthesis is the automated design of a hardware-software computer system. And the hardware-software partitioning problem allows only two different processors, of different types, in the allocation.

2.2 Complexity of the Synthesis Problem

System-level synthesis and embedded multiprocessor synthesis, is comprised of three interdependent main stages of allocation, assignment and scheduling. These problems are each known to be NP-complete for distributed systems [47].

A tractable or easy problem can be solved by algorithms that run in polynomial time; i.e., for a problem of size n, the time or number of steps needed to find the solution is a polynomial function of n. The problem becomes intractable if there is no known polynomial time solution for it. Furthermore, a problem is called NP (nondeterministic polynomial) if its solutions can be checked/verified by an algorithm whose run time is polynomial in the size of the input. A problem is considered NP-complete if its solutions can be verified quickly, and a quick algorithm to solve this problem can be used to solve all other NP problems quickly. Thus, finding an algorithm to solve any NP-complete problem implies that an algorithm and hence a solution can be found for all such problems, since any problem belonging to this class can be transformed into any other member of the class. It is not known whether any polynomial-time algorithms will ever be found for NP-complete problems, and determining whether these problems are tractable or intractable remains one of the most important questions in theoretical computer science. To find a solution to an NP-complete problem, there are a number of approaches that are outlined in the following sections.

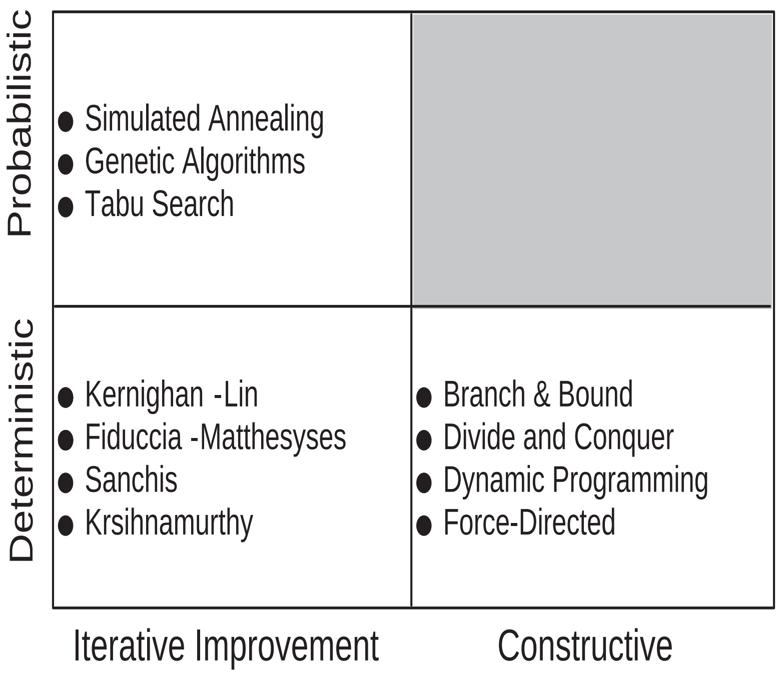


Figure 2.3: Classification of optimization algorithms [142].

2.2.1 Optimization Algorithms

The existing optimization algorithms to solve NP-complete problems in general and system-level synthesis and closely related problems i.e. hardware/software co-design and co-synthesis in particular are of very diverse nature and hence it is hard to present a comprehensive taxonomy for such algorithms. Wong et al. present a classification of optimization algorithms in [142] that can very well be applied to our problem. Their classification is based on two main criteria: i) the way in which the solution is built e.g. constructive or iterative improvement and ii) the presence or absence of randomness during optimization i.e. deterministic or probabilistic (see Figure 2.3.) Constructive algorithms as their name suggests, build the solution step by step and once the solution is constructed, it is not changed (e.g improved, etc). Iterative improvement algorithms on the other hand start from a complete solution (a trivial or poor solution) and iteratively make changes to the solution to improve it.

As it can be seen from the Figure, all algorithms can be classified as constructive or iterative improvement in one dimension and probabilistic or deterministic in another one. A survey of the literature shows that most algorithms belong to the constructive/deterministic set while only a very small set can be found that belong to the constructive/probabilistic. Examples of constructive deterministic algorithms are greedy algorithms, branch and bound and dynamic programming. Randomized version of deterministic algorithms are examples of constructive/probabilistic algorithms. The most commonly used iterative improvement probabilistic algorithms are genetic algorithms [50], simulated annealing [73][63] and Tabu search [19]. Probabilistic algorithms are inherently flexible and provide a tradeoff between the time and solution quality. The later characteristic is specially of our interest. This is due to the fact that embedded multiprocessor systems are typically designed as final implementations for dedicated functions and modifications to embedded system implementations are rare or simply do not occur. Hence embedded system designers can accept significantly longer compilation time. This increased compile time tolerance allows embedded system design tools to employ more thorough, time-consuming optimization techniques. Among the existing probabilistic techniques genetic algorithms (GAs) or evolutionary algorithms (EAs) are the most powerful and flexible and time tolerant algorithms. Moreover, they are very well suited to the multi-dimensional optimization nature of the problems we are addressing in this work. Most optimization algorithms in this thesis are based on genetic or evolutionary algorithms. The multi-objective optimization is described in the following Section.

2.2.2 Multi-objective Optimization

As mentioned in Section 2.1, the system-level synthesis tries to optimize several objectives i.e. system costs simultaneously. Examples of system costs are: parallel time (makespan), area, price, power consumption, etc. Optimization of these objectives/costs is often conflicting i.e. one can not be optimized without increasing other costs, for example one cannot optimize a system for speed without increasing the power consumption. Problems like this are known as either a multi-objective, multi-criteria, or a vector optimization problem.

In general, multi-objective optimization can be defined [109] as the problem of finding a vector of decision variables that i) satisfies constraints and ii) optimizes a vector function whose elements represent the objective functions. These functions form a mathematical description of performance criteria which are usually in conflict with each other. Hence, the term "optimize" means finding such a solution would give the values of all the objective functions acceptable to the designer. Formally, we can state it as follows: Find the vector $\vec{x} = [x_1, x_2, ..., x_n]^T$ such that it

- satisfies the m inequality constraint:

$$g_i(\vec{x}) \geq 0, i = 1, 2, ..., m; \tag{2.1}$$

  where g is a vector representing the constraints.

- satisfies the p equality constraints

$$h_i(\vec{x}) = 0, i = 1, 2, ..., p; \tag{2.2}$$

where h is a vector representing the constraints.

- optimizes the vector function

$$\vec{f}(\vec{x}) = [f_1(\vec{x}), f_2(\vec{x}), ..., f_k(\vec{x})]^T. \tag{2.3}$$

where f is a vector representing the objectives.

Most multi-objective problems do not have a unique and single solution but a set of solutions that satisfy the constraints while are better than one another in at least one optimization cost. Such solutions are called *Pareto points* in the design space. The concept of *Pareto optimum* was formulated by Vilfredo Pareto [110][109], and constitutes by itself the origin of research in multi-objective optimization. By definition $\vec{x}^\star$ is Pareto optimal if there exist no feasible vector $\vec{x}$ which would decrease some criterion without causing a simultaneous increase in at least one other criterion. The Pareto optimum gives a set of solution called *non-inferior* or *non-dominated* solutions. A point $\vec{x}^\star$ is a *non-dominated solution* if there is no $\vec{x}$ such that $f_i(\vec{x}) \leq f_i(\vec{x}^\star)$, for $i = 1, ..., n$ and for at least one value of i, $f(\vec{x}) < f(\vec{x}^\star)$. Figure 2.4 shows an example of a design space and a set of Pareto points for an optimization problem with two costs of area and parallel-time.

Some of the traditional methods for generating the Pareto-optimal set are as follows (more details can be found in [43][7]):

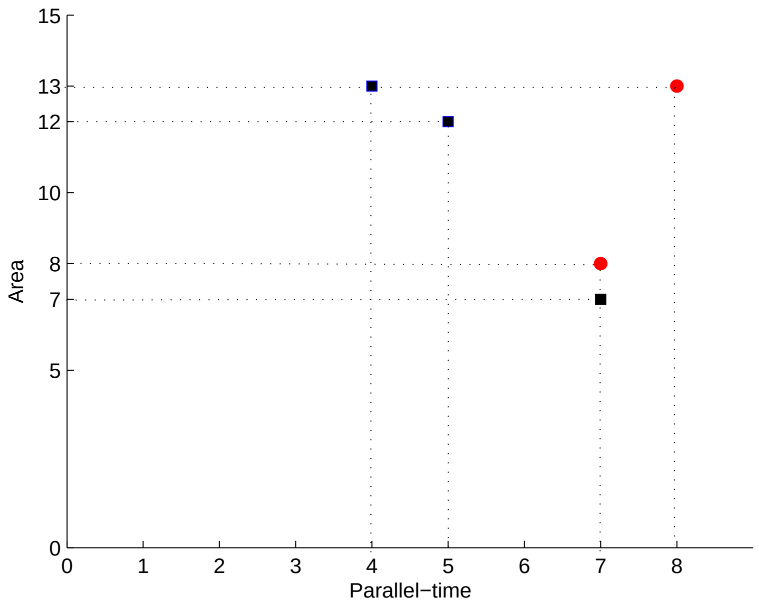


Figure 2.4: Example of a design space with Pareto points [30].

- Weighting Objectives — the multi-objective optimization problem is reduced to a single-objective optimization problem by forming a linear weighted combination of the objectives. In this method, the weighs are selected based on the importance of each objective function. The weights are then varied to yield the Pareto Optimal set.

- Hierarchical Optimization Method — in this method, different objectives are assigned a priority, and the optimization is done in a descending order of cost priorities. Each objective function is then minimized individually subject to a constraint that does not allow the minimum for the new function to exceed a prescribed fraction of a minimum of the previous function.

- Constraint Method — the multi-objective optimization problem is reduced to a single-objective optimization problem by transforming $k-1$ of the k objectives into constraints i.e. optimizing only a single cost. For example, in a synthesis problem with area and time as optimization objectives, one can treat time as a constraint and minimize the area under time constraint.

The main disadvantage of the weighting objective technique is that it cannot generate all Pareto-optimal solutions with non-convex trade-off surfaces which typically underlie system synthesis scenarios. Furthermore, forming these linear combinations can be awkward because they involve computing weighted sums of values associated with heterogeneous metrics. There is no clear methodology for formulating the linear combinations, and their physical interpretation is ambiguous. The main disadvantage with other technique is that they require some problem-specific knowledge to decide upon the priorities which may not be available. Another potential problem with these methods is their restricted application areas [26]. Moreover, as it is pointed out by Zitzler [151] these traditional methods all require multiple optimization runs to obtain an approximation of the Pareto-optimal set. This is due to the fact that these runs are performed independently and hence synergies can not be exploited. This may lead to high computation overhead. As an efficient alternative, evolutionary algorithms (EAs) have attracted a great deal of attention to replace the traditional methods. EA techniques work on a populations of solutions and hence are capable of searching a larger portion of the solution space and generating an entire set of Pareto-optimal solutions in a single optimization run.

In this thesis we employ multi-objective evolutionary algorithms. One place that we use this technique is in conjunction with evolutionary algorithms in the context of *Multi-Objective Genetic Local Search Algorithm*. Multi-objective EAs are described in the following section.

2.2.3 Multi-Objective Evolutionary Algorithms (MOEA) Optimization

The complex, combinatorial nature of the system synthesis problem and the need for simultaneous optimization of several incommensurable and often conflicting objectives has led many researchers to experiment with evolutionary algorithms (EAs) as a solution method. EAs seem to be especially suited to multi-objective optimization as due to their inherent parallelism, they have the potential to capture multiple Pareto-optimal solutions in a single simulation run and may exploit similarities of solutions by recombination. A brief introduction of EAs follows.

In general, genetic or evolutionary algorithms, inspired by observation of the natural process of evolution, are frequently touted to perform well on nonlinear and combinatorial problems [50]. They operate on a population of solutions rather than a single solution in the search space, and due to the domain independent nature of EAs, they guide the search solely based on the fitness evaluation of candidate solutions to the problem, whereas heuristics often rely on very problem-specific knowledge and insights to get good results.

In a typical evolutionary algorithm, the search space of all possible solutions of the problem is being mapped onto a set of finite strings (chromosomes) over a finite alphabet and hence each of the individual solutions is represented by an array or strings of values. The basic operators of such an evolutionary algorithm that are applied to candidate solutions to form new and improved solutions are *crossover, mutation and selection* [50]. The crossover operator is defined as a mechanism for incorporating the attributes of two parents into a new individual. The mutation operator is a mechanism for introducing necessary attributes into an individual when those attributes do not already exist within the current population of solutions. Selection is basically the same as Darwinian survival of the fittest creature and is the process in which individual strings are copied according to their fitness value and being passed to the next generation [50]. An outline of a general evolutionary algorithm is given in Figure 2.5.

```
INPUT: $N$ (population size), $p_{cross}$ (crossover probability), $p_{mut}$
(mutation probability), termination condition (time, number of
generations, etc.)
OUTPUT: $O$ (A non-dominated set of solutions).
Step 1  Initialization: Set $t = 0$, Generate an initial population
$P(t = 0)$ (of size $N$) randomly.
Step 2  Fitness Assignment: For each individual $i \in P(t)$ compute
different objective values and calculate the scalar fitness value $F(i)$.
Step 3  Selection: Select $N$ individuals according to the selection
algorithm and copy them to form the temporary population (mating
pool)$P'$.
Step 4  Crossover: Apply the crossover operator $\frac{N}{2}$ times
to individuals $\in P'$ to generate $N$ new children. Copy each set
of children (or their parents) according to the crossover probability
($p_{cross}$) to the temporary population $P''$.
Step 5  Mutation: Apply the mutation operator to each individual
$i \in P''$ according to the mutation probability ($p_{mut}$).
Step 6  Termination: Set $P(t + 1) = P'' and t = t + 1$,
If the termination criterion is met then copy the best solution(s) to $O$
and stop, otherwise go to step 2.
```

Figure 2.5: Outline of a typical Evolutionary Algorithm.

In a single-objective optimization, objective function and fitness function are often the same, however in multi-objective problems they are different and fitness assignment and selection have to address these difference and take multiple objectives into account. Some of the existing multi-objective EAs consider objectives separately, some use aggregation techniques, and some employ methods that applies the concept of Pareto dominance directly, to guide the solution space search towards the Pareto-optimal. Pareto-based techniques seem to be most popular in the field of evolutionary multi-objective optimization (more details can be found in [151]).

In order to find the Pareto-optimal set in a single optimization run, an evolutionary algorithm have to perform a multi-modal search to find multiple solutions that vary from each other to a great extent. Hence, maintaining a diverse population is essential for the efficiency of multi-objective EAs to prevent premature convergence and achieve a well distributed and well spread non-dominated set. Again, existing multi-objective EAs employ different techniques to overcome the problem of premature convergence, some of the most frequently use methods are *Fitness Sharing*, *Restricted Mating*, *Isolation by Distance*, etc. Details on these techniques can be found in [151].

In this work we employ the strength Pareto evolutionary algorithm (SPEA) [154] that uses a mixture of established and new techniques in order to approximate the Pareto-optimal set and it is shown to have superiority over other existing multi-objective EAs [154]. SPEA uses a mixture of established and new techniques in order to find multiple Pareto-optimal solutions in parallel. It's main characteristics are as follows:

(a) Stores non-dominated solutions externally in a second, continuously updated population;

(b) Evaluates an individual's fitness dependent on the number of external non-dominated points that dominate it, and uses the concept of Pareto dominance in order to assign scalar fitness values to individuals;

(c) Preserves population diversity using the Pareto dominance relationship, and

(d) Incorporates a grouping procedure in order to reduce the non-dominated set without destroying its characteristics.

The dominance relation used in MOEA optimization (also used in the SPEA algorithm) is defined as follows:

Definition 1: *Given two solutions a and b and a minimization problem, then a is said to* dominate *b iff*

$$\forall i \in \{1, 2, ..., n\} : f_i(a) \leq f_i(b) \wedge$$
$$\exists j \in \{1, 2, ..., n\} : f_j(a) < f_j(b). \tag{2.4}$$

All solutions that are not dominated by another solution are called *non-dominated*. The solutions that are non-dominated within the entire search space are denoted as *Pareto optimal* and constitute the *Pareto-optimal set*. An outline of the SPEA is given in Figure 2.6.

```
INPUT: $N$ (population size), $XN$ (external set/archive size), $p_{cross}$
(crossover probability), $p_{mut}$ (mutation probability), $G$ (number of
generations)
OUTPUT: $O$ (A non-dominated set of solutions).
Step 1   Initialization: Set $t = 0$, Generate an initial population $P(t)$
randomly. Initialize the external set $XP(t)$ to null ($= \emptyset$).
Step 2   Fitness Assignment: For each individual $i \in P(t)$ compute
different objective values and calculate fitness values of individuals in
$P(t)$ and $XP(t)$.
Step 3   Environmental Selection: Copy all non-dominated individuals
in $P(t)$ and $XP(t)$ to $XP(t + 1)$.
Step 4   Termination: If $t > G$ or other stopping criterion is met then
set $O = XP(t + 1)$ and stop.
Step 5   Mating Selection: Perform binary tournament selection on
$XP(t + 1)$ to fill the mating pool.
Step 6   Variation: Apply crossover and mutation operators to the mating
pool, set $P(t + 1)$ to the resulting population and $t = t + 1$. Go to Step 2.
```

Figure 2.6: Outline of the Strength Pareto Evolutionary Algorithm (SPEA) [154]

As mentioned earlier, one of our solutions for the multi-objective problem of system synthesis is based on SPEA. We have adapted and modified this algorithm to fit our problem. These modifications and changes are addressed in the relevant chapters.

2.3 System Specification

As mentioned earlier, we represent the embedded system applications that are to be mapped into the parallel architecture in forms of the widely-used *task graph model.* A task graph is a directed acyclic graph (DAG) $G = (V, E)$ that is constituted of $|V|$ tasks $\{v_1, v_2, ..., v_{|V|}\}$ in which there is a partial order : $v_i < v_j$ that implies task v_i has higher scheduling priority than v_j and v_j can not start execution until v_i finishes. This restriction is due to the data dependency between the two task nodes. Task nodes are in one-to-one correspondence with the computational tasks in the application. E represents the set of communication edges where each member is an ordered pair of tasks. We also define the following function:

- $wcet : V \times PE \rightarrow \Re^+$ denotes a function that assigns the worst case execution time $(wcet(v_i, pe_j))$ to task v_i of set V running on processing element pe_j. In homogeneous multiprocessor systems this function in reduce to a one-dimensional function $wcet : V \rightarrow \Re^+$ (i.e. $wcet(v_i)$). The execution of tasks are assumed to be non-preemptive.

- $C : V \times V \times CR \rightarrow \Re^+$ denotes a function that gives the cost (latency) occurred on each communication edge on a given communication resource (CR). That is $C(v_i, v_j, cr_k)$ is the cost of transferring data between v_i and v_2 on communication resource cr_k if they are assigned to different processing element. This value is zero if both tasks are running on the same processing element. $C(v_i, v_j, cr_k)$ is reduced to $C(v_i, v_j)$ when we consider a homogeneous communication network.

When addressing the system synthesis we assume each embedded system is characterized by multiple modes of functionality. A multi-mode embedded system supports multiple functions or operational modes, of which only one is running at any instant. We assume there are M different modes of operation and each mode is comprised of G_m task graphs, where m varies between 0 and $M-1$. $G_{m,i}(V,E)$ represents the ith task graph of mode m.

There is also a period $\pi(m,i)$ associated with each task graph. For each mode we form a hyper task graph $G_{Hm}(V,E)$ that consists of copies of high rate task graphs that are active for the hyper-period given in the following equation:

$$\pi_H(m) = \underset{i=0}{\overset{|G_m(V,E)|-1}{LCM}} \{\pi(m,i)\}. \tag{2.5}$$

LCM or the least common multiple of a set of numbers is the smallest multiple that is exactly divisible by every member of the set.

Each task is also characterized by a set of attributes given in the following equation:

$$V_{attr} = [\textit{type}, \mu_i, WCET, p_{avg}]^T, \tag{2.6}$$

where *type* denotes the type of the task or its functionality, and μ_i denotes the amount of instruction memory required to store the task. $WCET$ and p_{avg} denote the worst-case execution time and average power consumption, respectively. These values depend on the PE the task is running on. Each edge is also characterized by a single attribute given in the following equation:

$$E_{attr} = [\delta], \tag{2.7}$$

where δ denotes the size of data communicated in terms of the data unit. Once the edge is assigned to a communication resource (CR), the worst case communication time and average power consumption can be computed using the corresponding CRs attributes.

The target architecture we consider, consists of different processing elements (PE) and communication resources (CRs). PEs and CRs can be of various types. A final solution may be constituted of multiple instances of each PE or CR type. We represent the target architecture in the form of a directed graph $G_T(V_T, E_T)$ where V_T and E_T represent the processing elements and communication links respectively. More details on PEs and CRs follows.

Processing Elements (PEs)

A processing element (PE) is a hardware unit for executing tasks. We model several types of PEs: general-purpose processors (GPPs), digital signal processors (DSPs), application-specific integrated circuits (ASICs), and FPGAs. Tasks mapped to processors are implemented in software and run sequentially, while tasks mapped onto an ASIC or FPGA are implemented in hardware and can be performed in parallel if the designated unit is not engaged. Each PE can be characterized by the following attribute vector:

$$PE_{attr} = [\alpha, \kappa, \mu_d, \mu_i, p_{idle}]^T, \tag{2.8}$$

where α denotes the area of the processor, κ denotes the price of the processor, μ_d denotes the size of data memory, μ_i denotes the instruction memory size ($\mu_i = 0$ for ASICs) and p_{idle} denotes the idle power consumption of the processor.

Throughout this thesis we would use the term homogeneous multiprocessor systems to refer to a system of multiple identical GPPs or DSPs. These PEs do not share memory and communication relies solely on message-passing. In the context of homogeneous multiprocessor systems we use the terms PE and processor interchangeably.

Communication Resources (CRs)

A communication resource (CR) is a hardware resource for communication messages. Each CR also has an attribute vector:

$$CR_{attr} = [p, p_{idle}, \vartheta]^T, \tag{2.9}$$

where p denotes the average power consumption per each unit of data to be transferred, p_{idle} denotes idle power consumption and ϑ denotes the worst case transmission rate or speed per each unit of data.

In homogeneous multiprocessor systems we assume identical communication resources (links) for the system.

Chapter 3

Efficient Techniques for Clustering-Oriented Scheduling onto Homogeneous Embedded Multiprocessors

In this chapter we illustrate the effectiveness of the two-phase decomposition of multiprocessor scheduling into clustering and cluster-scheduling or merging and mapping task graphs onto embedded multiprocessor systems. We describe efficient and novel partitioning (clustering) and scheduling techniques that aggressively streamline inter-processor communication and can be tuned to exploit the significantly longer compilation time that is available to embedded system designers.

We take a new look at the two-phase method of scheduling that was introduced by Sarkar [119], and explored subsequently by other researchers such as Yang and Gerasoulis [149] and Kwok and Ahmad [79]. In this decomposition task clustering is performed as a compile-time pre-processing step and in advance of the actual task to processor mapping and scheduling process. This method, while simple, is a remarkably capable strategy for mapping task graphs onto embedded multiprocessor architectures that aggressively streamlines inter-processor communication and altogether has made it worthwhile for researches to incorporate this approach. In addition to the mentioned attractive qualities our work exposes and exploits that this decomposition scheme offers a number of other unique properties as follows: It introduces more modularity and hence more flexibility in allocating compile-time resources throughout the optimization process. This increased compile time tolerance allows us to employ a more thorough, time-intensive optimization technique [97]. Moreover, in most of the follow-up work, the focus has been on developing simple and fast algorithms (e.g., mostly constructive algorithms that choose a lower complexity approach over a potentially more thorough one with a higher complexity, and that do not revisit their choices) for each step [79][90][113][149] and relatively little work has been done on developing algorithms at the other end of the complexity/solution-quality trade-off (i.e., algorithms such as genetic algorithms that are more time consuming but have the potential to compute higher quality solutions). To our best knowledge, there has been also little work on evaluating the idea of decomposition or comparing scheduling algorithms that are composed of clustering and cluster-scheduling (or merging) (i.e. two-step scheduling algorithms) against each other or against one-step scheduling algorithms.

Our contribution in this chapter is as follows: We first introduce an evolutionary algorithm based clusterization function algorithm (CFA) and present the solution formulation. Next we evaluate CFA's performance versus two other leading clustering algorithms such as Sarkar's Internalization Algorithm (SIA) [119] and Yang and Gerasoulis's Dominant Sequence Clustering (DSC) [149]. To make a fair comparison, we introduce the randomized version of the two clustering algorithm RDSC (randomized version of DSC) and RSIA (randomized version of SIA). We use the mentioned five algorithms in conjunction with a cluster-scheduling (or merging) algorithm called Clustered Ready List Scheduling Algorithm(CRLA) and show that the choice of clustering algorithm can significantly change the overall performance of the scheduling. We address the potential inefficiency implied in using the two phases of clustering and merging with no interaction between the phases and introduce a solution that while taking advantage of this decomposition increases the overall performance of the resulting mappings. We present a general framework for performance comparison of guided random-search algorithms against deterministic algorithms and an experimental setup for comparison of one-step against two-step scheduling algorithms. This framework helps to determine the importance of different steps in the scheduling problem and effect of different approaches in the overall performance of the scheduling. We show that decomposition of the scheduling process improves the overall performance and that the quality of the solutions depends on the quality of the clusters generated in the clustering step. We also discuss why the parallel execution time metric is not a sufficient measure for performance comparison of clustering algorithms.

This chapter is organized as follows. In section 3.1 we present the background, notation and definitions used in this chapter. In section 3.2 we state the problem and our proposed framework. In section 3.3, we present the input graphs we have used in our experiments. Experimental results are given in section 3.4 and we conclude the chapter in section 3.5 with a summary of the chapter.

3.1 Background

3.1.1 Clustering and Scheduling

The concept of *clustering* has been broadly applied to numerous applications and research problems such as parallel processing, load balancing and partitioning [119][89][102]. Clustering is also often used as a front-end to multiprocessor system synthesis tools and as a compile-time pre-processing step in mapping parallel programs onto multiprocessor architectures. In this research we are only interested in the latter context, where given a task graph and an infinite number of fully-connected processors, the objective of clustering is to assign tasks to processors. In this context, clustering is used as the first step to scheduling parallel architectures and is used to group basic tasks into subsets that are to be executed sequentially on the same processor. Once the clusters of tasks are formed, the task execution ordering of each processor will be determined and tasks will run sequentially on each processor with zero intra-cluster overhead. The inter-cluster communication overhead however is contingent upon the underlying intercommunication network and *Send* and *Receive* primitives issued by parallel tasks [20][23]. If we assume zero delays for loading (unloading) data to (from) buffer and switching then it can be shown that the lower bound for the communication overhead between every two cluster is equal to the maximum cost of communications edges crossing those clusters and the upper bound equals to the sum of the communication cost of all the tasks belong to those clusters.

The target architecture for the clustering step is a clique of an infinite number of processors. The justification for clustering is that if two tasks are clustered together and are assigned to the same processor while an unbounded number of processors are available then they should be assigned to the same processor when the number of processors is finite [119].

In general, regardless of the employed communication network model, in the presence of heavy inter-processor communication, clustering tends to adjust the communication and computational time by changing the granularity of the program and forming coarser grain graphs. A perfect clustering algorithm is considered to have a decoupling effect on the graph, i.e. it should cluster tasks that are heavily dependent (data dependency is relative to the amount of data that tasks exchange or the communication cost) together and form composite nodes that can be treated as nodes in another task graph. After performing clustering and forming the new graph with composite task nodes, there has to be a scheduling algorithm to map the new and simpler graph to the final target architecture. To satisfy this, clustering and list scheduling (and a variety of other scheduling techniques) are used in a complimentary fashion in general. Consequently, clustering typically is first applied to constrain the remaining steps of synthesis, especially scheduling, so that they can focus on strategic processor assignments.

The clustering goal (as well as the overall goal for this decomposition scheme) is to minimize the parallel execution time while mapping the application to a given target architecture. The **parallel execution time** (or simply parallel time) is defined by the following expression:

$$\tau_{par} = \max(tlevel(v_x) + blevel(v_x) | v_x \in V), \tag{3.1}$$

where $tlevel(v_x)$ ($tlevel(v_x)$) is the length of the longest path between node v_x and the source (sink) node in the *scheduled graph*, including all of the communication and computation costs in that path, but excluding $t(v_x)$ from $tlevel(v_x)$. Here, by the scheduled graph, we mean the task graph with all known information about clustering and task execution ordering modeled using additional zero-cost edges. In particular, if v_1 and v_2 are clustered together, and v_2 is scheduled to execute immediately after v_1, then the edge (v_1, v_2) is inserted in the scheduled graph.

Although a number of innovative clustering and scheduling algorithms exist to date, none of these provide a definitive solution to the clustering problem. Some prominent examples of existing clustering algorithms are:

- Dominant sequence clustering (DSC) by Yang and Gerasoulis [147],
- Linear clustering by Kim and Browne [72], and
- Sarkar's internalization algorithm (SIA) [119].

In the context of embedded system implementation, one limitation shared by many existing clustering and scheduling algorithms is that they have been designed for general purpose computation. In the general-purpose domain, there are many categories of applications for which short compile time is of major concern. In such scenarios, it is highly desirable to ensure that an application can be mapped to an architecture within a matter of seconds. Thus, the clustering techniques of Sarkar, Kim, and specially, Yang have been designed with low computational complexity as a major goal. However, in embedded application domains, such as signal/image/video processing, the quality of the synthesized solution is by far the most dominant consideration, and designers of such systems can often tolerate compile times on the order of hours or even days if the synthesis results are markedly better than those offered by low complexity techniques. We have explored a number of approaches for exploiting this increased run-time-tolerance, that will be presented in sections 3.2.1 and 3.2.2. The first employed approach is based on the genetic algorithms that is briefly introduced in 3.1.2 and explored in section 3.2.1. In this chapter we assume a clique topology for the interconnection network where any number of processors can perform inter-processor communication simultaneously. We also assume dedicated communication hardware that allows communication and computation to be performed concurrently and we also allow communication overlap for tasks residing in one cluster.

3.1.2 Genetic Algorithms

Given the intractable nature of clustering and scheduling problems and the promising performance of Genetic Algorithms (GA) on similar problems, it is natural to consider a solution based on GAs, which may offer some advantages over traditional search techniques. GAs, inspired by observation of the natural process of evolution, are frequently touted to perform well on nonlinear and combinatorial problems [50]. A survey of the literature, reveals a large number of papers devoted to the scheduling problem while there are no GA approaches to task graph clustering. As discussed earlier in this chapter, the 2-phase decomposition of scheduling problem offers unique advantages that is worth being investigated and experimented thoroughly. Consequently, in this work we develop efficient GA approaches to clustering and mapping/merging task graphs which is discussed in the following section. More details about our solution representation and operator (crossover, mutation, etc.) implementation are given in 3.2.1.

3.1.3 Existing Approaches

IPC-conscious scheduling algorithm have received high attention in the literature and a great number of them are based on the framework of clustering algorithms [119][149][88][90]. This group of algorithms, which are the main interest of this work, have been considered as scheduling heuristics that directly emphasize reducing the effect of IPC to minimize the parallel execution time. Among existing clustering approaches are Sarkar's Internalization Algorithm (SIA) [119] and the Dominant Sequence Clustering (DSC) algorithm of Yang and Gerasoulis [149]. As introduced in section 3.1.1, Sarkar's clustering algorithm has a relatively low complexity. This algorithm is an edge-zeroing refinement algorithm that builds the clustering, step by step by examining each edge and clustering it only if the parallel time is not increased. Due to its local and greedy choices this algorithm is prone to becoming trapped in poor search space. DSC, builds the solution incrementally as well. It makes changes with regard to the global impact on the parallel execution time, but only accounts for the local effects of these changes. This can lead to the accumulation of suboptimal decisions, especially for large task graphs with high communication costs, and graphs with multiple critical paths. Nevertheless, this algorithm has been shown to be capable of producing very good solutions, and it is especially impressive given its low complexity.

In comparison to the high volume of research work on the clustering phase, there has been little research on the cluster-scheduling or merging phase [82]. Among a few merging algorithms are Sarkar's task assignment algorithm [119] and Yang's Ready Critical Path (RCP) algorithm [148]. Sarkar's merging algorithm is a modified version of list scheduling with tasks being prioritized based on their ranks in a topological sort ordering. This algorithm has a relatively high time complexity. Yang's merging algorithm is part of the scheduling tool PYRROS [146], and is a low complexity algorithm based on the load-balancing concept. Since merging is the process of scheduling and mapping the clustered graph onto the target embedded multiprocessor systems, it is expected to be as efficient as a scheduling algorithm that works on a non-clustered graph. Both of these algorithms lack this motive by oversimplifying assumptions such as assigning an ordering-based priority and not utilizing the (timing) information provided in the clustering step. A recent work on physical mapping of task graphs into parallel architectures with arbitrary interconnection topology can be found in [75]. A technique similar to Sarkar's has been used by Lewis, et al. as well in [87]. GLB and LLB [113] are two cluster-scheduling algorithms that are based on the load-balancing idea. Although both algorithms utilize timing information, they are inefficient in the presence of heavy communication costs in the task graph. GLB also makes local decisions with respect to cluster assignments which results in poor overall performance.

Due to deterministic nature of SIA and DSC, neither can exploit the increased compile time tolerance in embedded system implementation. There has been some research on scheduling heuristics in the context of compile-time efficiency [79][88]; however, none studies the implications from the compile time tolerance point of view. Additionally, since they concentrate on deterministic algorithms, they do not exploit compile time budgets that are larger than the amounts of time required by their respective approaches.

There has been some probabilistic search implementation of scheduling heuristics in the literature, mainly in the forms of genetic algorithms (GA). The genetic algorithms attempt to avoid getting trapped in local minima. Hou et al. [53] , Wang and Korfhage [139], Kwok and Ahmad [80], Zomaya et al. [155], and Correa et al. [22] have proposed different genetic algorithms in the scheduling context. Hou and Correa use similar integer string representations of solutions. Wang and Korfhage use a two-dimensional matrix scheme to encode the solution. Kwok and Ahmad also use integer string representations, and Zomaya et al. use a matrix of integer substrings. An aspect that all of these algorithms have in common is a relatively complex solution representation in the underlying GA formulation. Each of these algorithm must at each step check for the validity of the associated candidate solution and any time basic genetic operators (crossover and mutation) are applied, a correction function needs to be invoked to eliminate illegal solutions. This overhead also occurs while initializing the first population of solutions. These algorithms also need to significantly modify the basic crossover and mutation procedures to be adapted to their proposed encoding scheme. We show that in the context of the clustering/merging decomposition, these complications can be avoided in the clustering phase, and more streamlined solution encodings can be used for clustering.

Correa et al. address compile-time consumption in the context of their GA approach. In particular, they run the lower-complexity search algorithms as many times as the number of generations of the more complex GA, and compare the resulting compile-times and parallel execution times (schedule makespans). However, this measurement provides only a rough approximation of compile time efficiency. More accurate measurement can be developed in terms of fixed compile-time budgets (instead of fixed numbers of generations). This will be discussed further in 3.2.2.

As for the complete two-phase implementation there is also a limited body of research work providing a framework for comparing the existing approaches. Liou, et. al address this issue in their paper [90]. They first apply three average-performing merging algorithms to their clustering algorithm and next run the three merging algorithms without applying the clustering algorithm and conclude that clustering is an essential step. They build their conclusion based on problem- and algorithmic-specific assumptions. We believe that reaching such a conclusion may need a more thorough approach and a specialized framework and set of experiments. Hence, their comparison and conclusions cannot be generalized to our context in this work. Dikaiakos et al. also propose a framework in [38] that compares various combinations of clustering and merging. In [113], Radulescu et al., to evaluate the performance of their merging algorithm (LLB), use DSC as the base for clustering algorithms and compare the performance of DSC and four merging algorithms (Sarkar's, Yang's, GLB and LLB) against the one-step MCP algorithm [143]. They show that their algorithm outperforms other merging algorithms used with DSC while it is not always as efficient as MCP. In their comparison they do not take the effect of clustering algorithms into account and only emphasize merging algorithms.

Some researchers [81][100] have presented comparison results for different clustering (without merging) algorithms (classified as Unbounded Number of Clusters (UNC) scheduling algorithms) and have left the cluster-merging step unexplored. In section 3.4, we show that the clustering performance does not necessarily provide an accurate answer to the overall performance of the two-step scheduling and hence cluster comparison does not provide important information with respect to the scheduling performance. Hence, a more accurate comparison approach should compare the two-step against the one-step scheduling algorithms. In this research we will give a framework for such comparisons that take the compile-time budget into account as well.

3.2 The Proposed Mapping Algorithm and Solution Description

3.2.1 CFA:Clusterization Function Algorithm

In this section, we present a new framework for applying GAs to multiprocessor scheduling problems. For such problems any valid and legal solution should satisfy the precedence constraints among the tasks and every task should be present and appear only once in the schedule. Hence the representation of a schedule for GAs must accommodate these conditions. Most of the proposed GA methods satisfy these conditions by representing the schedule as several lists of ordered task nodes where each list corresponds to the task nodes run on a processor. These representations are typically *sequence* based [44]. Observing the fact that conventional operators that perform well on bit-string encoded solutions, used in many GAs, do not work on solutions represented in the forms of sequences, opens up the possibility of gaining a high quality solution by designing a well-defined representation. Hence, our solution representation only encodes the mapping-related information and represents it as a single subset of graph edges β, with no notion of an ordering among the elements of β. This representation can be used with a wide variety of scheduling and clustering problems. Our technique is also the *first* clustering algorithm that is based on the framework of genetic algorithms.

Our representation of clustering exploits the view of a clustering as a subset of edges in the task graph. Gerasoulis and Yang have suggested this view of clustering in their characterization of certain clustering algorithms as being *edge-zeroing* algorithms [49]. One of our contributions in this research is to apply this subset-based view of clustering to develop a natural, efficient genetic algorithm formulation. For the purpose of a genetic algorithm, the representation of graph clusterings as subsets of edges is attractive since *subsets have natural and efficient mappings into the framework of genetic algorithms.*

Derived from the *schema* theory (a schema denotes a similarity template that represents a subset of $\{0, 1\}^l$), canonical GAs provide near-optimal sampling strategies over subsequent generations [8]. Canonical GAs use binary representations of each solution as fixed-length strings over the set $\{0, 1\}$) and efficiently handle the optimization problems of the form $f : \{0, 1\} \rightarrow \Re$. Furthermore, binary encodings in which the semantic interpretations of different bit positions exhibit high symmetry allow us to leverage extensive prior research on genetic operators for symmetric encodings rather than forcing us to develop specialized, less-thoroughly-tested operators to handle the underlying non-symmetric, non-traditional and sequence-based representation. For example, in our case, each bit corresponds to the existence or absence of an edge within a cluster. Consequently, our binary encoding scheme is favored both by schema theory, and significant prior work on genetic operators. Furthermore, by providing no constraints on genetic operators, our encoding scheme preserves the natural behavior of GAs. Finally, conventional GAs assume that symbols or bits within an individual representation can be independently modified and rearranged, however the solution that represents a schedule must contain exactly one instance of each task and the sequence of tasks should not violate the precedence constraint. Thus, any deletion, duplication or moving of tasks constitutes an error. The traditional crossover and mutation operators are generally capable of producing infeasible or illegal solutions. Under such a scenario, the GA must either discard or repair (to make it feasible) the non-viable solution. Repair mechanisms transform infeasible individuals into feasible ones. However, the repair process may not always be successful. Our proposed approach never generates an illegal or invalid solution, and thus saves repair-related synthesis time that would otherwise have been wasted in locating, removing or correcting invalid solutions.

Our approach to encoding clustering solution is based on the following definition.

Definition 1: Suppose that β_i is a subset of task graph edges. Then $f_{\beta_i} : E \rightarrow \{0, 1\}$ denotes the **clusterization function** associated with β_i. This function is defined by equation (3.2):

$$f_{\beta_i}(e) = \begin{cases} 0 & if(e \in \beta_i), \\ 1 & otherwise. \end{cases} \tag{3.2}$$

where E is the set of communication edges and e denotes an arbitrary edge of this set. When using a clusterization function to represent a clustering solution, the edge subset β_i is taken to be the set of edges that are contained in one cluster. To form the clusters we use the information given in β (zero and one edges) and put every pair of task nodes that are joined with zero edges together. The set β is defined as in (3.3),

$$\beta = \bigcup_{i=1}^{n} \beta_i \tag{3.3}$$

An illustration is shown in Figure 3.1. It can be seen in Figure 3.1.a that all the edges of the graph are mapped to 1, which implies that the β_i subsets are empty or $\beta = \emptyset$. In Figure 3.1.b edges are mapped to both 0s and 1s and four clusters have been formed. The associated β_i subsets of zero edges are given in Figure 3.1.c. Figure 3.2 shows another clusterization function and the associated clustered graph. It can be seen in Figure 3.2.a

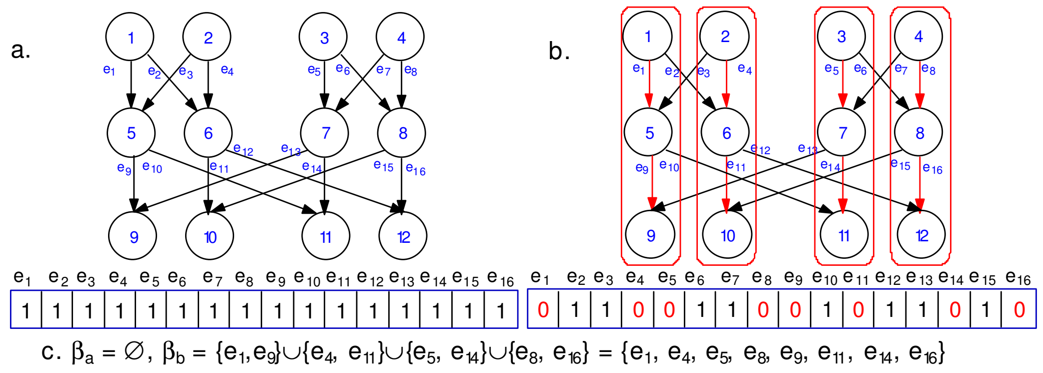

a.
b.
c. $\beta_a = \varnothing$, $\beta_b = \{e_1, e_9\} \cup \{e_4, e_{11}\} \cup \{e_5, e_{14}\} \cup \{e_8, e_{16}\} = \{e_1, e_4, e_5, e_8, e_9, e_{11}, e_{14}, e_{16}\}$

Figure 3.1: (a) An application graph representation of an FFT and the associated clusterization function f_{β_a}; (b) a clustering of the FFT application graph, and f_{β_b}(c) the resulting subset β_b of clustered edges, along with the (empty) subset β_a of clustered edges in the original (unclustered) graph.

that tasks t_2, t_3, t_9, t_{10} and t_{12} do not have any incoming or outgoing edges that are mapped to 0 and hence do not share any clusters with other tasks. These tasks form clusters with single tasks and also are the only tasks running on the processors they are assigned to. Hence, when using the clusterization function definition to map zero edges onto clusters, tasks that are joined with zero edges are mapped onto the same clusters and tasks with no zero edges connected to them form single-task clusters. This is shown in Figure 3.2.b. Given the clustered graph and clusterization function we can define a node subset C (similar to the edge subset β) that represents the clustered graph. In this definition, every subset C_i (for an arbitrary i) is the set of heads and tails (the head is the node to which the edge points and the tail is the node from which the edge leaves) of edges that belong to edge subset β_i. Hence every clustering of a graph or a clustered graph can be represented using either the edge subset β or the node subset C representation (an example of the node subset representation of a task graph is given in Figure 3.2.c).

In this work the term *clustering* represents a clustered graph, where every pair of

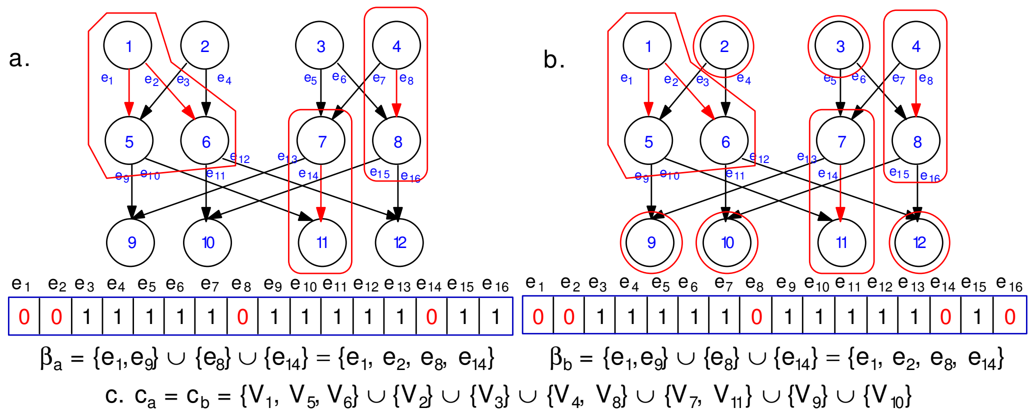

a.
b.
1
2
3
4
5
6
7
8
9
10
11
12
e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11 e12 e13 e14 e15 e16
0 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1
0 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0
βa = {e1,e9} ∪ {e8} ∪ {e14} = {e1, e2, e8, e14}
βb = {e1,e9} ∪ {e8} ∪ {e14} = {e1, e2, e8, e14}
c. ca = cb = {V1, V5, V6} ∪ {V2} ∪ {V3} ∪ {V4, V8} ∪ {V7, V11} ∪ {V9} ∪ {V10}

Figure 3.2: (a) A clustering of the FFT application graph and the associated clusterization function f_{β_a}. (b) The same clustering of the FFT application graph, and f_{β_b} where single-task clusters are shown, (c) Node subset representation of the clustered graph.

nodes in each cluster is connected by a path. A *clustered graph* in general can have tasks with no connections that are clustered together. In this research, however, we do not consider such clusters. We also use the term clustering and clustered graph interchangeably. Because it is based on clusterization functions to represent candidate solutions, we refer to our GA approach as the *clusterization function algorithm* (CFA). The CFA representation offers some useful properties, they are described bellow:

Property 1: Given a clustering β there exists a clusterization function that generates it.

Proof: Our proof is derived from the function definition in(3.2). Given a clustering of a graph, we can construct the clusterization function f_{beta} by examining the edge list. Starting from the head of the list, for each edge (or ordered pair of task nodes) if both head and tail of the edge belong to the same cluster i.e. $\forall e_k | e_k = (v_i, v_j)$ if $(v_i \in c_x) \wedge (v_j \in c_x)$ then the associated edge cost would be zero and according to (3.2), $f(e_k) = 0$ (this edge also belongs to β_x i.e. $e_k \in \beta_x$. If the head and tail of the edge do not belong to the same cluster i.e. $(((v_i \in c_x) \wedge (v_j \notin c_x)) \vee ((v_i \notin c_x) \wedge (v_j \in c_x)))$ then $f(e_k) = 1$. Hence by examining the edge list we will construct the clusterization function and this concludes the proof.

Property 2: Given a clusterization function, there is a unique clustering that is generated by it.

Proof: The given clusterization function f_β can be represented in form of a binary array with the length equal to $|E|$ where the ith element of array is associated with the ith edge e_i and the binary values determine weather the edge belongs to a cluster or not. By constructing the clusters from this array we can prove the uniqueness of the clustering. We examine each element of the binary array and remove the associated edge in the graph if the binary value is 1. Once we have examined all the edges and removed the proper edges the graph is partitioned to connected components where each connected component is a cluster of tasks. Each edge is either removed or exists in the final partitioned graph depending on its associated binary value. Hence anytime we build the clustering or clustered graph using the same clusterization function we will get the same connected components, partitions or clusters, and consequently, the clustering formed by a clusterization function is unique. The time complexity of forming clusters is $O(|E|)$.

There is also an implicit use of knowledge in CFA-based clustering. In most GA-based scheduling algorithms, the initial population is generated by randomly assigning tasks to different processors. The population evolves through the generations by means of genetic operators and the selection mechanism while the only knowledge about the problem that is taken into account in the algorithm is of a structural nature, through the verification of solution feasibility. In such GAs the search is accomplished entirely at random considering only a subset of the search space. However, in CFA the assignment of tasks to clusters or processors is based on the edge zeroing concept. In this context, clustering tasks nodes together is not entirely random. Two task nodes will only be mapped onto one cluster if there is an edge connecting them and they can not be clustered together if there is no edge connecting them, because this clustering can not improve the parallel time. Although GAs do not need any knowledge to guide their search, GAs that do have the advantage of being augmented by some knowledge about the problem they are solving have been shown to produce higher quality solutions and to be capable of searching the design space more thoroughly and efficiently [1][22]. The implementation details of CFA are as follows.

Coding of Solutions The solution to the clustering problem is a clustered graph and each individual in the initial population has to represent a clustering of the graph. As mentioned in the previous section, we defined the clusterization function to efficiently code the solutions. Hence, the coding of an individual is composed of an n-size binary array, where $n = |E|$ and $|E|$ is the total number of edges in the graph. There is a one to one relation between the graph edges and the bits, where each bit represents the presence or absence of the edge in a cluster.

Initial Population The initial population consists of binary arrays that represent different clusterings. Each binary array is generated randomly and every bit has an equal chance for being 1 or 0.

Genetic Operators As mentioned earlier, our binary encodings allow us to leverage extensive prior research on genetic operators rather than forcing us to develop specialized, less-thoroughly-tested operators to handle the non-traditional and sequence-based representation. Hence, the genetic operators for reproduction (*mutation* and *crossover*) that we use are the traditional two-point crossover and the typical mutator for a binary string chromosome where we flip the bits in the string with a given probability [50]. Both approach are very simple, fast and efficient and none of them lead to an illegal solution, which makes the GA a repair-free GA as well.

For the *selection* operator we use binary tournament with replacement [50]. Here, two individuals are selected randomly, and the best of the two individuals (according to their fitness value) is the winner and is used for reproduction. Both winner and loser are returned to the pool for the next selection operation of that generation.

Fitness Evaluation As mentioned in section 3.1.2, a GA is guided in its search solely by its fitness feedback, hence it is important to define the fitness function very carefully. Every individual chromosome represents a clustering of the task graph. The goal of such a mapping is to minimize the parallel time; hence, in CFA, fitness is calculated from the parallel time τ_{par}, (from (3.1)), as follows in 3.4:

$$F(Ind_i, P(t)) = \tau_{par_{WC}}(P(t)) - \tau_{par}(Ind_i, P(t)), \tag{3.4}$$

where $F(Ind_i, P(t))$ is the fitness of an individual Ind_i in the current population $P(t)$; and $\tau_{par_{WC}}(P(t))$ is the maximum or worst case parallel time computed in $P(t)$; and $\tau_{par}(Ind_i, P(t))$ is the parallel time of that individual in $P(t)$. Thus, to evaluate the fitness of each individual in the population, we must first derive the unique clustering that is given by the associated clusterization function, and then schedule the associated clusters. Then from the schedule, we compute the parallel time of each individual in the current population and the fitness for each individual will be its distance from the worst solution. The more the distance the fitter the individual is. To schedule tasks in each cluster, we have applied a modified version of list scheduling that abandons the restrictions imposed by a global scheduling clock, as proposed in the DLS algorithm [129]. Since processor assignment has been taken care of in the clustering phase, the scheduler needs only to order tasks in each cluster and assign start times. The scheduler orders tasks based on the precedence constraints and the priority level [119] (the task with the highest $blevel$ has the highest priority). Additionally, to reduce the processor idle times, an insertion scheme has been applied where a lower priority task can be scheduled ahead of a higher priority task if it fits within the idle time of the processor and also satisfies its precedence constraints when moved to this position. The parallel time of the associated scheduled graph constitutes the fitness of each individual (member of the GA population) as defined in 3.4.

The implemented search method in our research is based on *simple (non-overlapping) genetic algorithms*. Once the initial population is generated and has been evaluated, the algorithm creates an entirely new population of individuals by selecting solution pairs from the old population and then mating them by means of the genetic operators to produce the new individuals for the new population. The simple GA is a desirable scheme in search and optimization, where we are often concerned with convergence or off-line performance [50]. We also allow elitism in CFA. Under this policy the best individual of $P(t)$ or the current population is unconditionally carried over to $P(t + 1)$ or the next generation to prevent losing it due to the sampling effect or genetic operator disruption [151][28]. During our experiments we observed that different clusterings can lead to the same fitness value, and hence in our implementation, we copy the n best solutions to the next generations. In our tests n varied from 1 to 10 percent of the population so in the worst case 90% of the solutions were being updated in each generation.

The process of reproduction and evaluation continues while the termination condition is not satisfied. In this work we ran the CFA for a fixed number of generations regardless of the graph size or applications.

3.2.2 Randomized Clustering : RDSC, RSIA

Two of the well-known clustering algorithms discussed earlier in this chapter, DSC and SIA, are deterministic heuristics, while our GA is a guided random search method where elements in a given set of solutions are probabilistically combined and modified to improve the fitness of populations. To be fair in comparison of these algorithms, we have implemented a randomized version of each deterministic algorithm — each such randomized algorithm, like the GA, can exploit increases in additional computational resources (compile-time tolerance) to explore larger segments of the solution space.

Since the major challenge in clustering algorithms is to find the most strategic edges to "zero" in order to minimize the parallel execution time of the scheduled task graph, we have incorporated randomization into to the edge selection process when deriving randomized versions of DSC (RDSC) and SIA (RSIA).

In the randomized version of SIA, we first sort all the edges based on the sorting criteria of the algorithm i.e. the highest IPC cost edge has the highest priority. The first element of the sorted list — the candidate edge to be zeroed by insertion in a cluster — then is selected with probability pr, where pr is a parameter of the randomized algorithm (we call pr the *randomization parameter*); if this element is not chosen, the second element is selected with probability pr; and so on, until some element is chosen, or no element is returned after considering all the elements in the list. In this last case (no element is chosen); a random number is chosen from a uniform distribution over $\{0, 1, ..., |T| - 1\}$ (where T is the set of edges that have not yet been clustered).

In the randomized version of the DSC algorithm, at each clustering step two node priority lists are maintained: a partial free task (a task node is partially free if it is not scheduled and at least one of its predecessors has been scheduled but not all of its predecessors have been scheduled) list and a free task (a task node is free if all its predecessors have been scheduled) list, both sorted in descending order of their task priorities (the priority for each task in the free list is the sum of the task's $tlevel$ and $blevel$. The priority value of a partial free task is defined based on the $tlevel$, IPC and computational cost — more details can be found in [149]). The criterion for accepting a zeroing is that the value of $tlevel(v_x)$ of the highest priority free list does not increase by such zeroing. Similar to RSIA, we first sort based on the sorting criteria of the algorithm, the first element of each sorted list then is selected with probability pr, and so on. Further details on this general approach to incorporating randomization into greedy, priority-based algorithms can be found in [153], which explores randomization techniques in the context of DSP memory management.

When $pr = 0$, clustering is always randomly performed by sampling a uniform distribution over the current set of edges, and when $pr = 1$, the randomized technique reduces to the corresponding deterministic algorithm. Each randomized algorithm version begins by first applying the underlying (original) deterministic algorithm, and then repeatedly computing additional solutions with a "degree of randomness" determined by pr. The best solution computed within the allotted (pre-specified) compile-time tolerance (e.g., 10 minutes, 1 hour, etc.) is returned. Our randomized algorithms, by way of running the corresponding deterministic algorithms first, maintain the performance bounds of the deterministic algorithms. A careful analysis of the (potentially better) performance bounds of the randomized algorithms is an interesting direction for the future study. Experimentally, we have found the best randomization parameters for RSIA and RDSC to be 0.10 and 0.65, respectively.

Both RDSC and RSIA are capable of generating all the possible clusterings (using our definition of clustering given in 3.2.1). This results because in both algorithms the base for clustering is zeroing (IPC cost of) edges by clustering the edges and all edges are visited at least once (In RSIA edges are visited exactly once) and hence every two task nodes have the opportunity of being mapped onto the same cluster.

3.2.3 Merging

Merging is the final phase of scheduling and is the process of mapping a set of clusters (as opposed to task nodes) to the parallel embedded multiprocessor system where a finite number of processors is available. This process should also maintain the minimum achievable parallel time while satisfying the resource constraints and must be designed to be as efficient as scheduling algorithms. As mentioned earlier for the merging algorithm, we have modified the ready-list scheduling heuristic so it can be applied to a cluster of nodes (CRLA). This algorithm is indeed very similar to the Sarkar's task assignment algorithm except for the priority metric: studying the existing merging techniques, we observed that if the scheduling strategy used in the merging phase is not as efficient as the one used in the clustering phase, the superiority of the clustering algorithm can be negatively effected. To solve this problem we implemented a merging algorithm (clustered ready-list scheduling algorithm or CRLA) such that it can use the timing information produced by the clustering phase. We observed that if we form the priority list in order of increasing $(LST, TOPOLOGICAL\_SORT\_ORDERING)$ of tasks (or $blevel$), tasks preserve their relative ordering that was computed in the clustering step. $LST(v_i)$ or the latest starting time of task v_i is defined as

$$LSTv_i = LCT(v_i) - wcet(v_i), \tag{3.5}$$

where $LCT(v_i)$ or the latest completion time is the latest time at which task v_i can complete execution. Similar to Sarkar's task assignment algorithm, the same ordering is also maintained when tasks are sorted within clusters.

In CRLA (similar to Sarkar's algorithm) initially there are no tasks assigned to the n_p available processors. The algorithm starts with the clustered graph and maps it to the processor through $|V|$ iterations. In each stage, a task at the head of the priority list is selected and along with other tasks in the same cluster is assigned to one of the n_p processors that gives the minimum parallel time increase from the previous iteration. For cluster to processor assignment we always assume all the processors are idle or available. The algorithm finishes when the number of clusters has been reduced to the actual number of physical processors. An outline of this algorithm is presented in Figure 3.3. In the

INPUT: A clustered graph G_c, with execution time $wcet(V)$, inter-cluster communication estimates $C(e)$, n_p processors, n_c clusters with task ordering within clusters.
OUTPUT: An optimized mapping and scheduling of the clustered graph onto n_p processors.

```
1   Initialize list $LIST$ of size $n_p$ s.t. $List(p) \leftarrow \emptyset$, FOR $p = 1 : n_p$;.
2   Initialize $PRIORITYLIST \leftarrow (v_1, v_2, ..., v_|V|)$ where $v_i$s are sorted based on
    their $blevel$ or $(LST, TOPOLOGICALSORTORDERING)$
3   FOR $j \leftarrow 1$ to $|V|$
4     IF $(proc(v_i) \notin \{1, ..., n_p\})$
5       Select a processor $i$, s.t. the merging of $cluster(v_j)$ and $LIST(i)$ gives the
        best parallel time $\tau_{par}$.
6       Merge $cluster(v_j)$ and $LIST(i)$.
7       Assign all the tasks on $cluster(v_j)$ to processor $i$, update$LIST(i)$.
8       For all tasks on $LIST(i)$ set $proc(v_k) \leftarrow i$.
9     ENDIF
10  ENDFOR
```

Figure 3.3: A sketch of the employed cluster-scheduling or merging algorithm (CRLA).

following section we explain the implementation of the overall system.

3.2.4 Two-phase mapping

In order to implement the two-step scheduling techniques described earlier, we used the three addressed clustering algorithms; CFA, RDSC and RSIA in conjunction with the CRLA. Our experiments were setup in two different formats that are explained in sections 3.2.4 and 3.2.4.

First Approach

In the first step, the clustering algorithm, being characterized by their probabilistic search of the solution space, had to run iteratively for a given time budget. Through extensive experimentation with CFA using small and large size graphs we found that running CFA for 3000 iterations (generations) is the best setup for CFA. CFA finds the solution to smaller size graphs in earlier generations (~ 1500) but larger size graphs need more time to perform well and hence we set the number of iteration to be 3000 for all graph sizes. We then ran CFA for this number of iterations and recorded the running time of the algorithm as well as the resulting clustering and performance measures. We used the recorded running time of CFA for each input graph to determine the allotted running time for RDSC or RSIA on the same graph. This technique allows comparison under equal amounts of running time. After we found the results of each algorithm within the specified time budget, we used the clustering information as an input to the merging algorithm described in section 3.2.3 and ran it once to find the final mapping to the actual target architecture. In most cases, the number of clusters in CFA's final result is more than the number in RSIA or RDSC. RSIA tends to find solutions with smaller numbers of clusters than the other two algorithms. To compare the performance of these algorithms we set the number of actual processors to be less than the minimum achieved number of clusters. Throughout the experiments we tested our algorithms for 2, 4, 8 and 16 processor architectures depending on the graph sizes.

Second Approach

Although CRLA employs the timing information provided in the clustering step, the overall performance is still sensitive to the employed scheduling or task ordering scheme in the clustering step. To overcome this deficiency we modified the fitness function of CFA to be the merging algorithm. Hence, instead of evaluating each cluster based on its local effect (which would be the parallel time of the clustered graph mapped to an infinite processor architecture) we evaluate each cluster based on its effect on the final

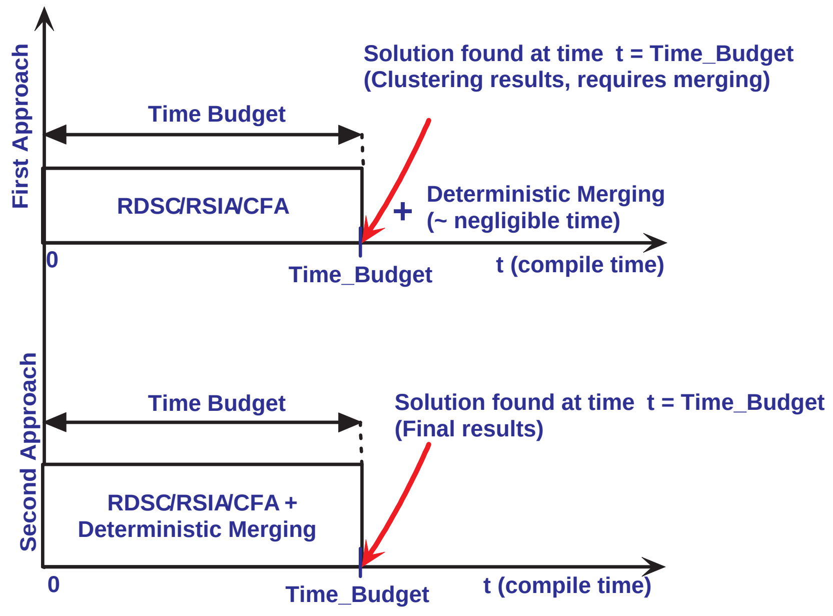


Figure 3.4: The diagrammatic difference between the two different implementations of the two-step clustering and cluster-scheduling or merging techniques. Both find the solution at the given time budget.

mapping. Except for this modification, the rest of the implementation details for CFA remain unchanged. RDSC and RSIA are not modified although the experimental setup is changed for them. Consequently, instead of running these two algorithms for as long as the time budget allows, locating the best clustering, and applying merging in one step, we run the overall two-step algorithm within the time budget. That is we run RDSC (RSIA) once, apply the merging algorithm to the resulting clustering, store the results, and start over. At the end of each iteration we compare the new result with the stored result and update the stored result if the new one shows a better performance.

The difference between these two approaches is shown in Figure 3.4. Experimental results for this approach are given in section 3.4.

For the second proposed approach the fitness evaluation may become time-consuming as the graph size increases. Fortunately, however, there is a large amount of parallelism in the overall fitness evaluation process. Therefore, for better scalability and faster run-time, one could develop a parallel model of the second framework. One such model (micro-grain parallelism [80]) is the asynchronous master-slave parallelization model [50]. This model maintains a single local population while the evaluation of the individuals is performed in parallel. This approach requires only knowledge of the individual being evaluated (not the whole population), so the overheard is greatly reduced. Other parallelization techniques such as course-grained and fine-grained [80] can also be applied for performance improvements to both approaches, while the micro-grain approach would be most beneficial for the second approach, which has a costly fitness function.

3.2.5 Comparison Method

The performance comparison of a two-step scheduling algorithm against a one-step approach is an important comparison that needs to be carefully and efficiently done to avoid any biases towards any specific approaches. The main aim of such a comparison is to help us answer some unanswered questions regarding the performance and effectiveness of multi-step scheduling algorithms such as the following: Is a pre-processing step (clustering here) advantageous to the multiprocessor scheduling? What is the effect of each step on the overall performance? Should both algorithms (for clustering and merging) be complex algorithms or an efficient clustering algorithm only requires a simple merging algorithm? Can a highly efficient merging algorithm make up for a clustering algorithm with poor performance? What are the important performance measures for each step?

The merging-step of a two-step scheduling technique is a modified one-step ready list scheduling heuristic that instead of working on single task nodes, runs on clusters of nodes. Merging algorithms must be designed to be as efficient as scheduling algorithms and to optimize the process of "cluster to physical processor mapping" as opposed to "task node to physical processor mapping". To compare the performance of a two-step decomposition scheme against a one-step approach, since our algorithms are probabilistic (and hence time-tolerant) search algorithms (e.g. CFA, RDSC and RSIA), we need to compare them against a one-step scheduling algorithm with similar characteristics, i.e. capable of exploiting the increased compile time and exploring a larger portion of the solution space. To address this need, first we selected a one-step evolutionary based scheduling algorithm, called *combined genetic-list algorithm* or CGL [22], that was shown to have outperformed the existing one-step evolutionary based scheduling algorithms (for homogeneous multiprocessor architectures.) Next we selected a well-known and efficient list scheduling algorithm (that could also be efficiently modified to be employed as cluster-scheduling algorithm). The algorithm we selected is an important generalization of list-scheduling, which is called ready-list scheduling that has been formalized by Printz [112]. Ready-list scheduling maintains the list-scheduling convention that a schedule is constructed by repeatedly selecting and scheduling ready nodes, but eliminates the notion of a static priority list and a global time clock. In our implementation we used the $blevel(v_x)$ metric to assign node priorities. We also used the insertion technique (to exploit unused time slots) to further improve the scheduling performance. With the same technique described in section 3.2.2, we also applied randomization to the process of constructing the priority list of nodes and implemented a *randomized ready-list* scheduling (RRL) technique that can exploit increases in additional computational resources (compile time tolerance).

We then set up an experimental framework for comparing the performance of the two-step CFA (the best of the three clustering algorithms CFA, RDSC and RSIA [68]) and CRLA against one-step CGL and one-step RRL algorithm. We also compared DSC and CRLA against the RL algorithm (step 3 in Figure 3.5).

In the second part of these experiments, we study the effect of each step in overall scheduling performance. To find out if an efficient merging can make up for an average performing clustering, we applied CRLA to several clustering heuristics: first we compared the performance of the two well-known clustering algorithms (DSC and SIA) against the randomized versions of these algorithms (RDSC and RSIA) with CRLA as the merging algorithm. Next, we compared the performance of CFA and CRLA against RDSC and RSIA. By keeping the merging algorithm unchanged in these sets of experiments we are able to study the effect of a good merging algorithm when employed with clustering techniques that exhibit a range of performance levels.

To find out the effect of a good clustering while combined with an average-performing merging algorithm we modified CRLA to use different metrics such as topological ordering and static level to prioritize the tasks and compared the performance of CFA and CRLA against CFA and the modified-CRLA. We repeated this comparison for RDSC and RSIA. In each set of these experiments we kept the clustering algorithm fixed so we can study the effect of a good clustering when used with different merging algorithms. The outline of this experimental set up is presented in Figure 3.5.

Step 1.
Select a well-known efficient single-phase scheduling algorithm.
(insertion-based Ready-List Scheduling (RL) with $blevel$ metric)
Step 2.
Modify the scheduling algorithm to get
a) An algorithm that accepts clusters of nodes as input (Clustered Ready-List Scheduling (CRLA)),
b) An algorithm that can exploit the increased compile time (Randomized Ready-List Scheduling (RRL))
Step 3.
Compare the performance of a one-phase scheduling algorithm vs a two phase scheduling algorithm.
a) CFA + CRLA vs. RRL
b) CFA + CRLA vs. CGL
c) DSC + CRLA vs. RL
Step 4.
Compare the importance of clustering phase vs. merging phase
a) CFA + CRLA vs.RDSC + CRLA b) CFA + CRLA vs.RSIA + CRLA
c) DSC + CRLA vs.RDSC + CRLA d) SIA + CRLA vs.RSIA + CRLA
e) CFA + CRLA vs.CFA + CRLA (using different metrics)
f) RDSC + CRLA vs.RDSC + CRLA (using different metrics)
g) RSIA + CRLA vs. RSIA + CRLA (using different metrics)

Figure 3.5: Experimental setup for comparing the effectiveness of a one-phase scheduling approach versus the two-phase scheduling method.

3.3 Input Benchmark Graphs

In this study, all the heuristics have been tested with three sets of input graphs. The description of each sets is given in the following sections.

3.3.1 Referenced Graphs

The Reference Graphs (RG) are task graphs that have been previously used by different researchers and addressed in the literature. This set consists of 29 graphs (7 to 41 task nodes). These graphs are relatively small graphs but do not have trivial solutions and expose the complexity of scheduling very adequately. Graphs included in the RG set are given in Table 3.1.

Table 3.1: Referenced Graphs (RG) Set

No.	Source of Task Graphs	No.	Source of Task Graphs
1	Ahmad and Kwok [2](13 nodes)	16	McCreary et al. [100](20 nodes)
2	Al-Maasarani [4](16 nodes)	17	McCreary et al. [100](28 nodes)
3	Al-Mouhamed [5](17 nodes)	18	McCreary et al. [100](28 nodes)
4	Bhattacharyya(12 nodes)	19	McCreary et al. [100](28 nodes)
5	Bhattacharyya(14 nodes)	20	McCreary et al. [100](32 nodes)
6	Chung and Ranka [17](11 nodes)	21	McCreary et al. [100](41 nodes)
7	Colin and Chretienne [20](9 nodes)	22	Mccreary and Gill [99](9 nodes)
8	Gerasoulis and Yang [49](7 nodes)	23	Shirazi et al. [124](11 nodes)
9	Gerasoulis and Yang [49](7 nodes)	24	Teich et al. [135](9 nodes)
10	Karplus and Strong [66](21 nodes)	25	Teich et al. [135](14 nodes)
11	Kruatrachue and Lewis [78](15 nodes)	26	Yang and Gerasoulis [147](7 nodes)
12	Kwok and Ahmad [79](18 nodes)	27	Yang and Gerasoulis [149](7 nodes)
13	Liou and Palis [90](10 nodes)	28	Wu and Gajski [143](16 nodes)
14	McCreary et al. [100](15 nodes)	29	Wu and Gajski [143](18 nodes)
15	McCreary et al. [100](15 nodes)		

3.3.2 Application Graphs

This set (AG) is a large set consists of 300 application graphs involving numerical computations (Cholesky factorization, Laplace Transform, Gaussian Elimination, Mean value analysis, etc., where the number of tasks varies from 10 to 2000 tasks), and digital signal processing (DSP). The DSP-related task graphs include N-point Fast Fourier Transforms (FFTs), where N varies between 2 and 128; a collection of uniform and non-uniform multi-rate filter banks with varying structures and number of channels; and a compact disc to digital audio tape (cd2dat) sample-rate conversion application.

Here, for each application, we have varied the *communication to computation cost ratio* (CCR), which is defined in (3.6):

$$CCR = \frac{\sum C(e)/|E|}{\sum wcet(x)/|V|} \tag{3.6}$$

Specifically, we have varied the CCR between 0.1 to 10 when experimenting with each task graph.

3.3.3 Random Graphs

This set (RANG) was generated using Sih's random benchmark graph generator [128]. Sih's generator attempts to construct synthetic benchmarks that are similar in structure to task graphs of real applications. The RANG set consists of two subsets: the first subset (ssI) contains graphs with 50 to 500 task nodes and CCRs of 0.1, 0.2, 0.5, and 1 to 10. The second subset (ssII) contains graphs with an average of 50 nodes and 100 edges and different CCRs (0.1, 0.5, 1.0, 2.0 and 10).

3.4 Performance Evaluation and Comparison

In this section, first we present the performance results and comparisons of clustering and merging algorithms described in section 3.2. All algorithms were implemented on an Intel Pentium III processor with a 1.1 GHz CPU speed. To make a more accurate comparison we have used the Normalized Parallel Time (NPT) that is defined as:

$$NPT = \frac{\tau_{par}}{\sum_{v_i \in CP} wcet(v_i)}, \tag{3.7}$$

where τ_{par} is the parallel time. The sum of the execution times on the CP (Critical Path) represents a lower bound on the parallel time. In our experiments, running times of the algorithms are not useful measures, because we run all the algorithms under an equal time-budget.

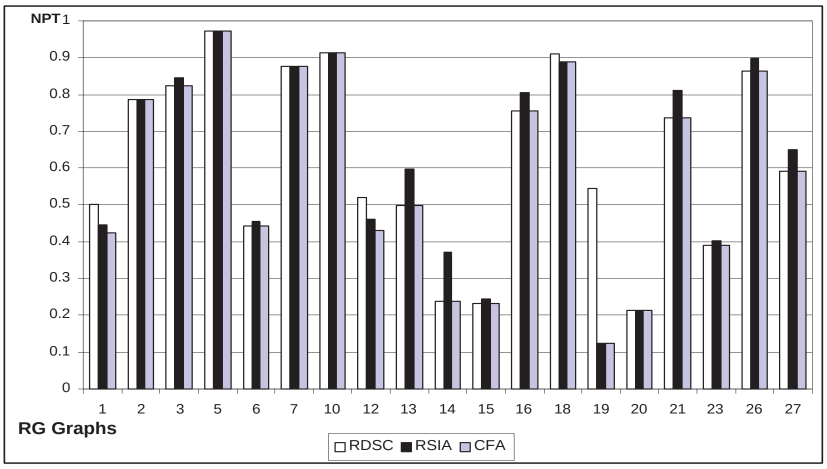


Figure 3.6: Normalized Parallel Time (NPT) generated by RDSC, RSIA and CFA for the RG set.

3.4.1 Results for the Referenced Graphs (RG) Set

The results of applying CFA and randomized clustering algorithms (RDSC and RSIA) to a subset of the RG set is given in Figure 3.6. The x-axis shows the graph number as given in Table 3.1.

It was observed that in the clustering step CFA constantly performed better than or as good as the randomized algorithms. On average CFA outperformed RDSC by 4.25%, and RSIA by 4.3%.

The results of the performance comparisons of one-step scheduling algorithms versus two-step scheduling algorithms for a subset of the RG set are given in Figure 3.7. The first four graphs show the performance of the CFA and CRLA against randomized ready list scheduling and a one step genetic-list scheduling (CGL) algorithm for 2 and 4 processor architectures. A quantitative comparison of these algorithms is given in Tables 3.2 and 3.3. It can be seen that given two equally good one-step and two-step scheduling algorithms, the two-step algorithm can actually gain better performance compared to the single-step algorithms. DSC is a relatively good clustering algorithm but not as efficient as CFA or its randomized version (RDSC). However, it can be observed that when used against a one-step scheduling algorithm, it still can offer better solutions over 70% of the time.

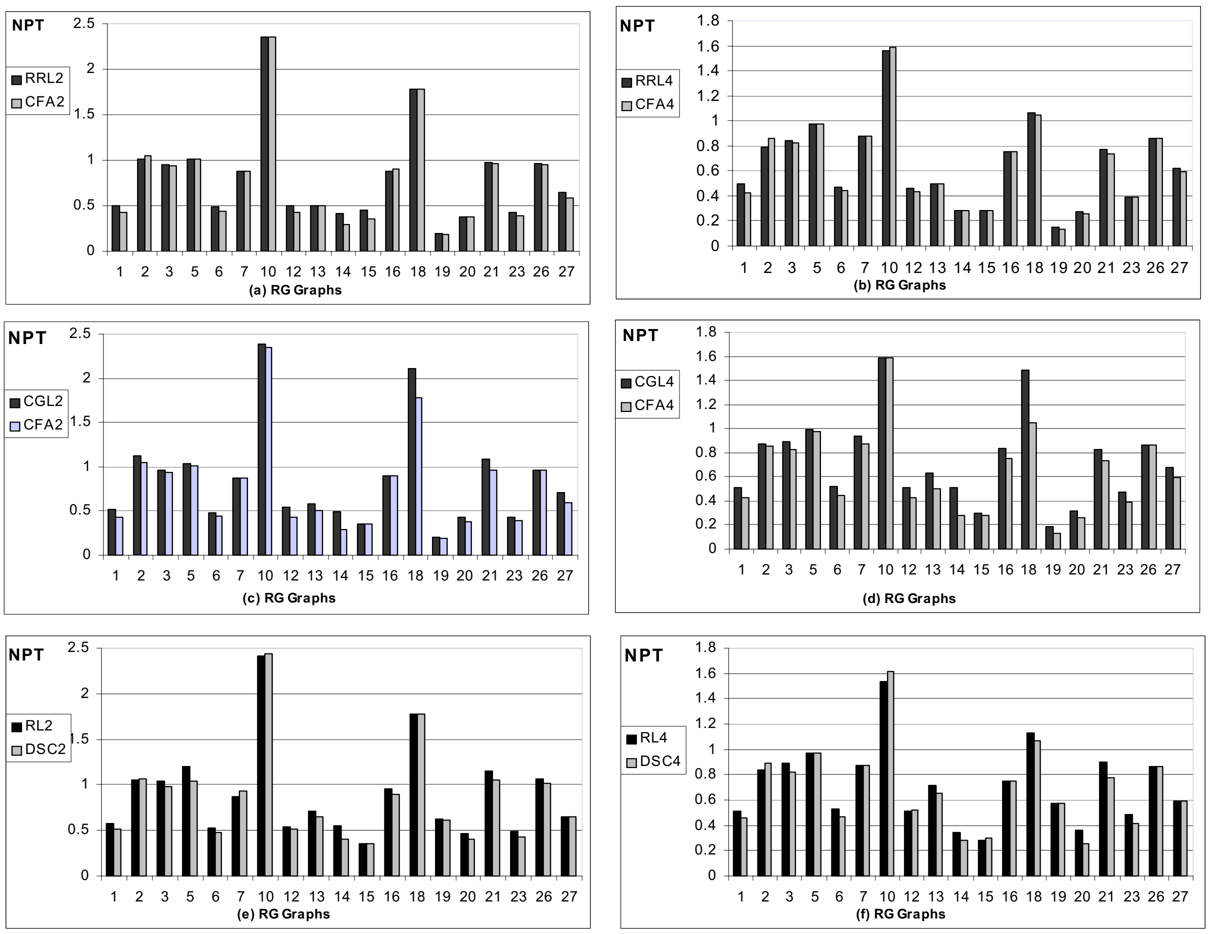


Figure 3.7: Effect of one-phase vs. two phase scheduling. RRL vs. CFA + CRLA on (a) 2 and (b) 4-processor architecture. CGL vs. CFA + CRLA on (c) 2 and (d) 4-processor architecture. RL vs. DSC + CRLA on (e) 2 and (f) 4-processor architecture.

To study the effect of clustering we ran our next set of experiments. The comparison between results of merging (CFA, RDSC and RSIA) and (DSC, RDSC, SIA, RSIA) using CRLA onto 2 and 4-processor architectures are given in Figures 3.8 and 3.9 respectively.

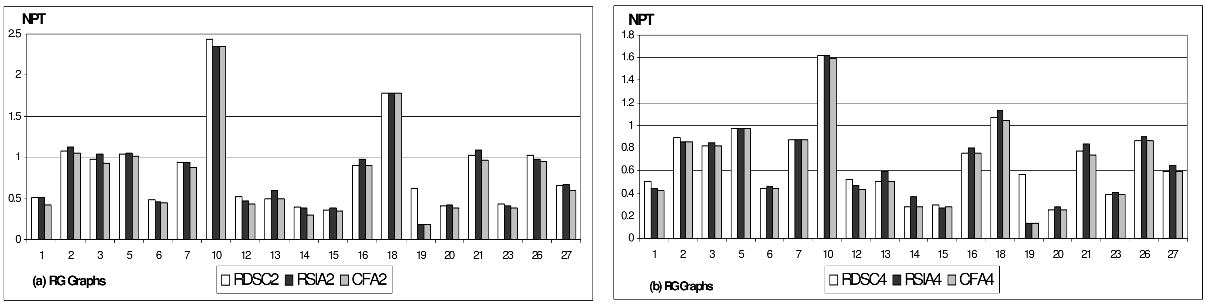


Figure 3.8: Mapping of a subset of RG graphs onto (a) 2-processor, and (b) 4-processor architectures applying CRLA to the clusters produced by the RDSC, RSIA and CFA algorithms.

It can be seen that the better the quality of the clustering algorithms the better the overall performance of the scheduling algorithms. In this case CFA clustering is better than RDSC and RSIA and RDSC are RSIA and better than their deterministic versions.

3.4.2 Results for the Application Graphs (AG) Set

The result of applying the clustering and merging algorithms to a subset of application graphs (AG) representing parallel DSP (FFT set) are given in this section. The number of nodes for the FFT set varies between 100 to 2500 nodes depending on the matrix size N.

The results of the performance comparisons of one-step scheduling algorithms versus two-step scheduling algorithms for a subset of the AG set are given in Figure 3.10, Figure 3.11and Figure 3.12.

A quantitative comparison of these algorithms is given in Tables 3.2 and 3.3.

The experimental results of studying the effect of clustering on the AG set are given in Figure 3.13 and Figure 3.14. We observed that CFA performs its best in the presence

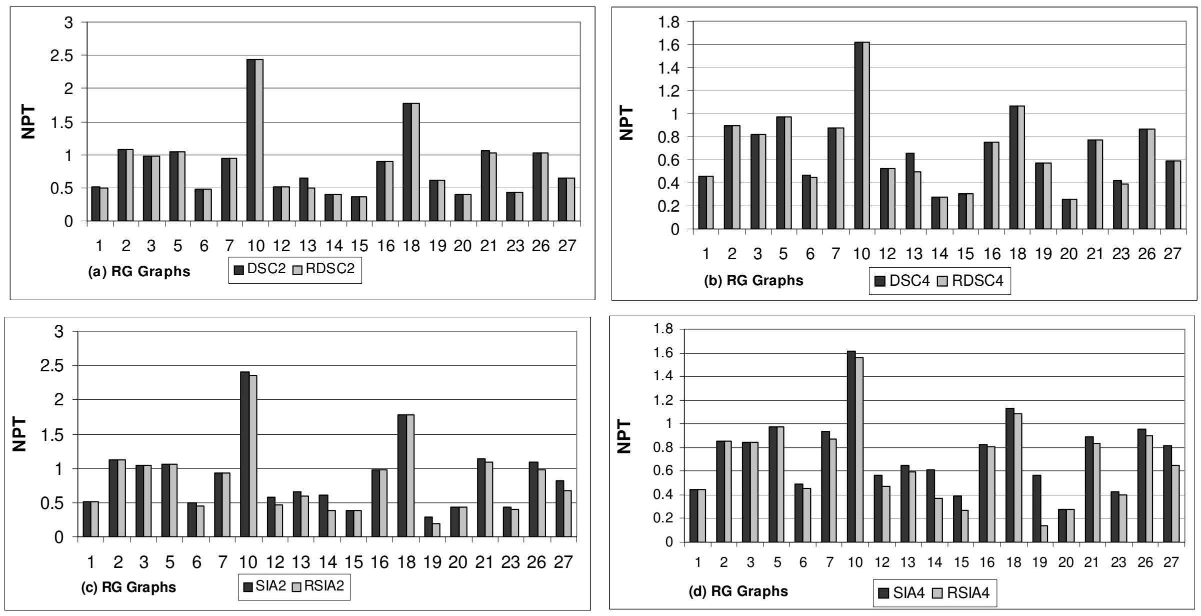


Figure 3.9: Effect of Clustering: Performance comparison of DSC, RDSC, SIA and RSIA on RG graphs mapped to (a,c) 2-processor, (b,d) 4-processor architectures using CRLA algorithm.

of heavy inter-processor communication (e.g. $CCR = 10$). In such situations, exploiting parallelism in the graph is particularly difficult, and most other algorithms perform relatively inefficiently and tend to greedily cluster edges to avoid IPC (over 97% of the time, CFA outperformed other algorithms under high communication costs). The trend in multiprocessor technology is toward increasing costs of inter-processor communication relative to processing costs (task execution times) citeBenini:2001 , and we see that CFA is particularly well suited toward handling this trend (when used prior to the scheduling process).

Figure 3.15 shows the clustering and merging results for an FFT application by CFA, and the two randomized algorithms RDSC and RSIA onto the final 2-processor ar-

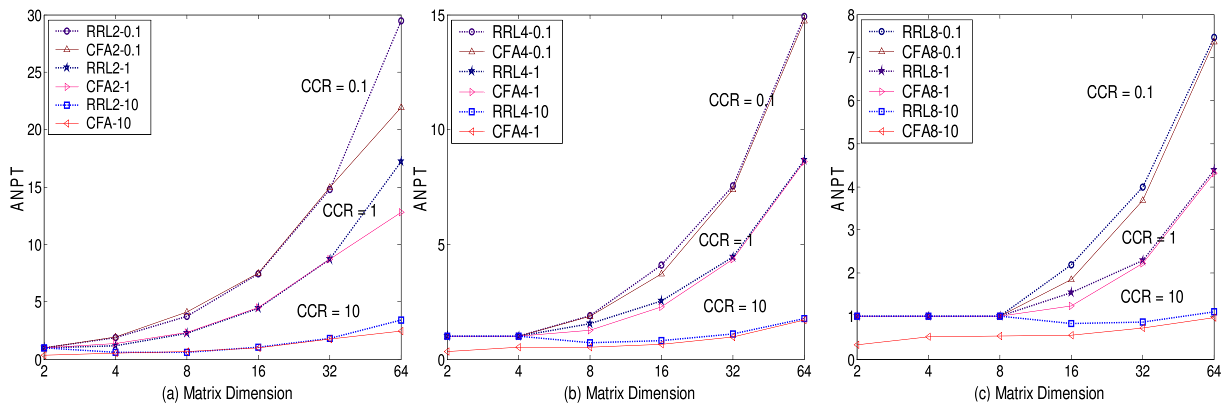


Figure 3.10: One-phase Randomized Ready-List scheduling (RRL) vs. Two Phase CFA + CRLA for a subset of AG set graphs mapped to (a) 2-processor, (b) 4-processor, (c) 8-processor architectures.

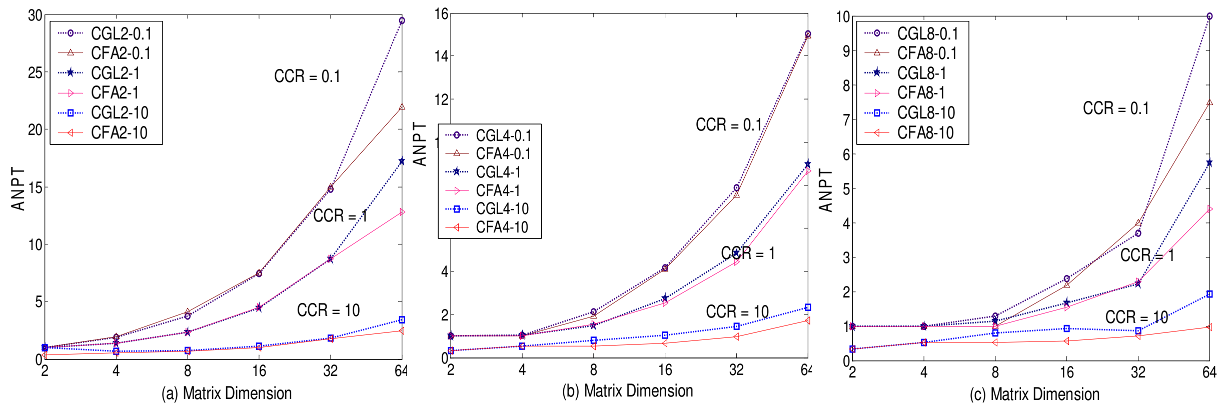


Figure 3.11: One Phase CGL vs. Two Phase CFA + CRLA for a subset of AG graphs mapped to (a) 2-processor, (b) 4- processor, (c) 8-processor architectures.

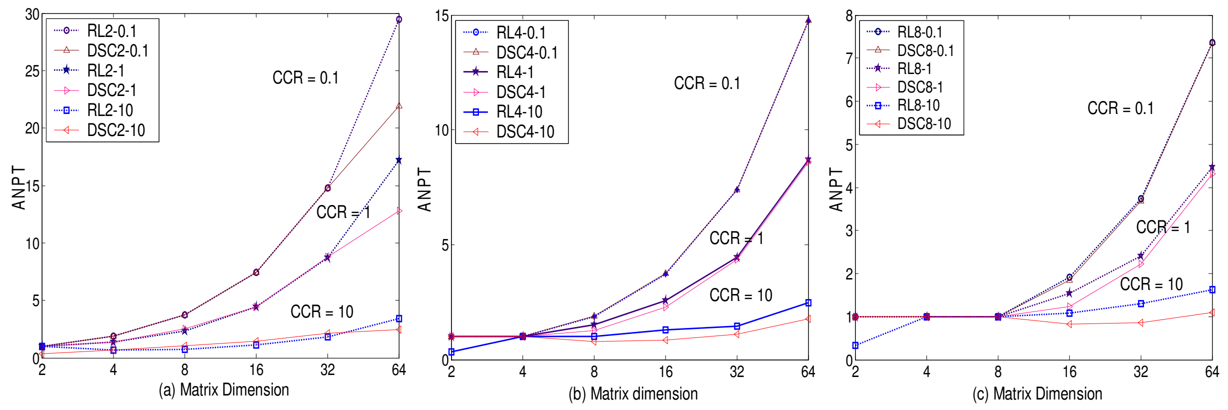


Figure 3.12: One Phase Ready-list Scheduling (RL) vs. Two Phase DSC for a subset of AG set graphs mapped to (a) 2-processor, (b) 4-processor, (c) 8-processor architectures.

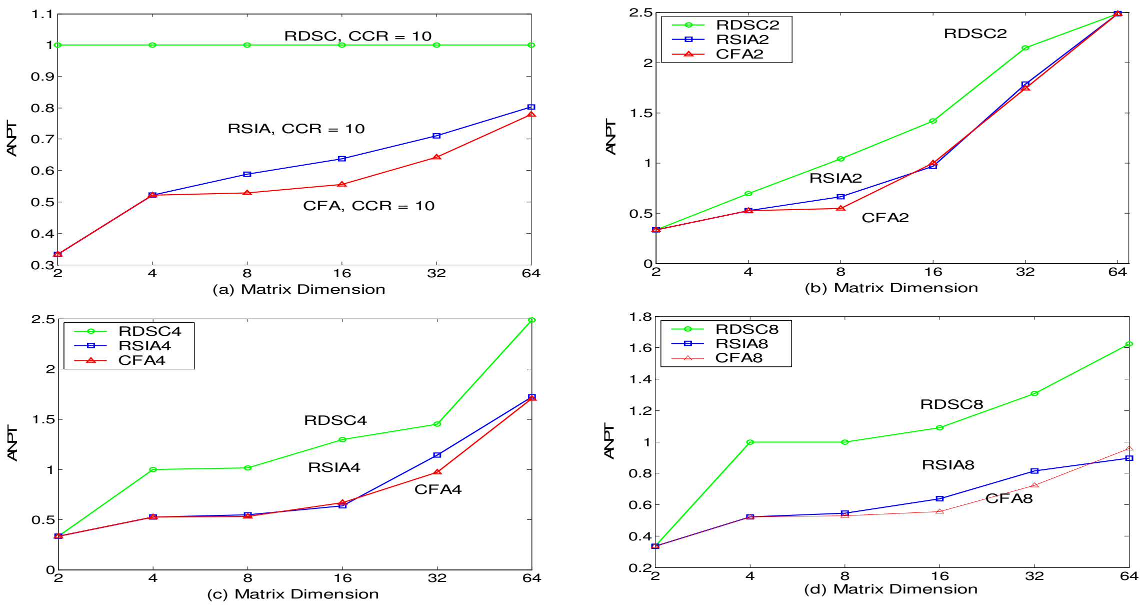


Figure 3.13: Average Normalized Parallel Time from applying RDSC, RSIA and CFA to a subset of AG set (for CCR = 10), (a) results of clustering algorithms, (b) results of mapping the clustered graphs onto a 2-processor architecture, (c) results of mapping the clustered graphs onto a 4-processor architecture, (d) results of mapping the clustered graphs onto an 8-processor architecture.

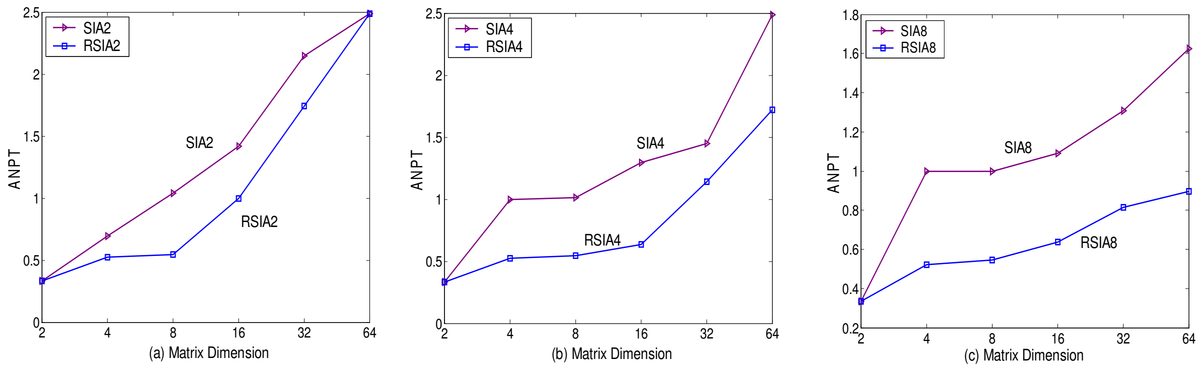


Figure 3.14: Effect of Clustering: Performance comparison of SIA and RSIA on a subset of AG graphs mapped to (a) 2-processor, (b) 4-processor, (c) 8-processor architecture using CRLA algorithm.

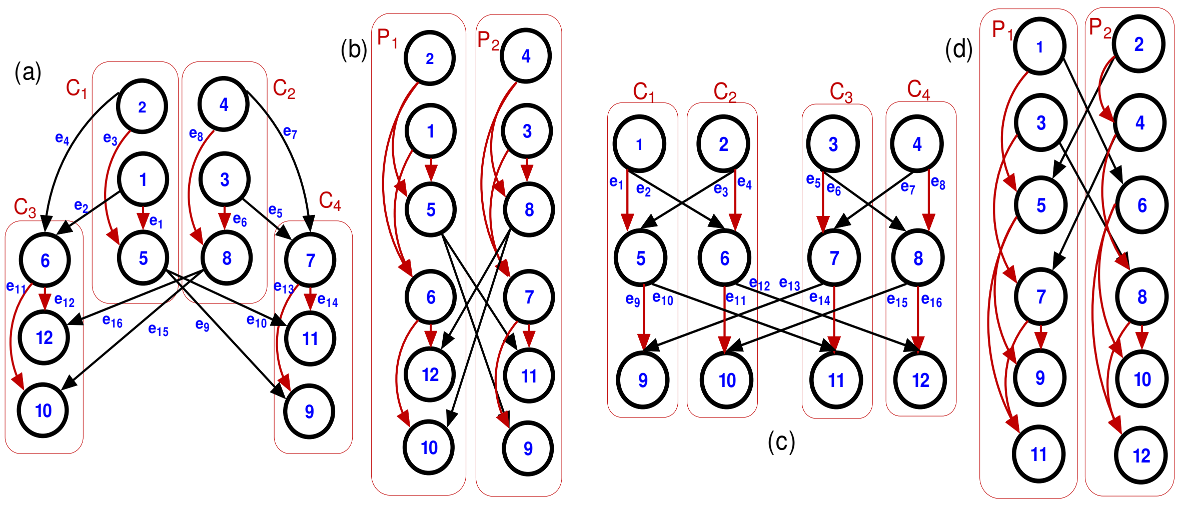


Figure 3.15: Results for FFT application graphs clustered using (a) CFA (PT = 130) and (c) RDSC and RSIA (PT = 150) and final mapping of FFT application graphs onto a two-processor architecture using the clustering results of (b) CFA (PT = 180) and (d) RDSC and RSIA (PT = 205).

chitecture. Our studies on some of the DSP application graphs, including a wide range of filter banks, showed that while the final configurations resulting from different clustering algorithms achieve similar load-balancing and inter-processor communication traffic, the clustering solutions built on CFA results are able to outperform clusterings derived by the other two algorithms.

3.4.3 Results for the Random Graphs (RANG) Set

In this section we have shown the experimental results (in terms of average NPT or ANPT) for setI of the RANG task graphs. Figure 3.16 shows the results of comparing the one-step randomized ready-list scheduling (RRL) against the two step CFA and CRLA. Figure 3.17 shows the results of comparing the one-step probabilistic scheduling algorithm CGL against the two-step guided search scheduling algorithm CFA and

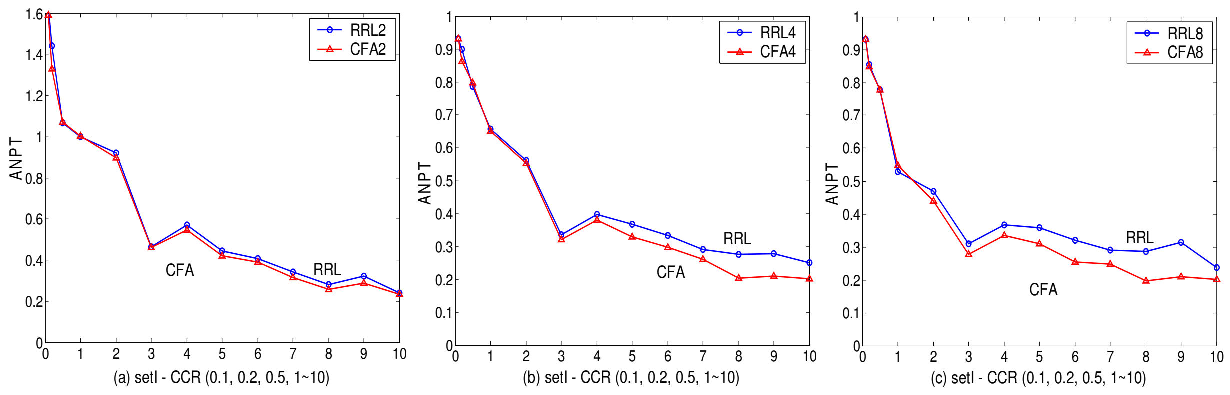


Figure 3.16: One Phase Randomized Ready-List scheduling (RRL) vs. Two Phase CFA + CRLA for RANG setI graphs mapped to (a) 2-processor, (b) 4-processor, (c) 8-processor architectures.

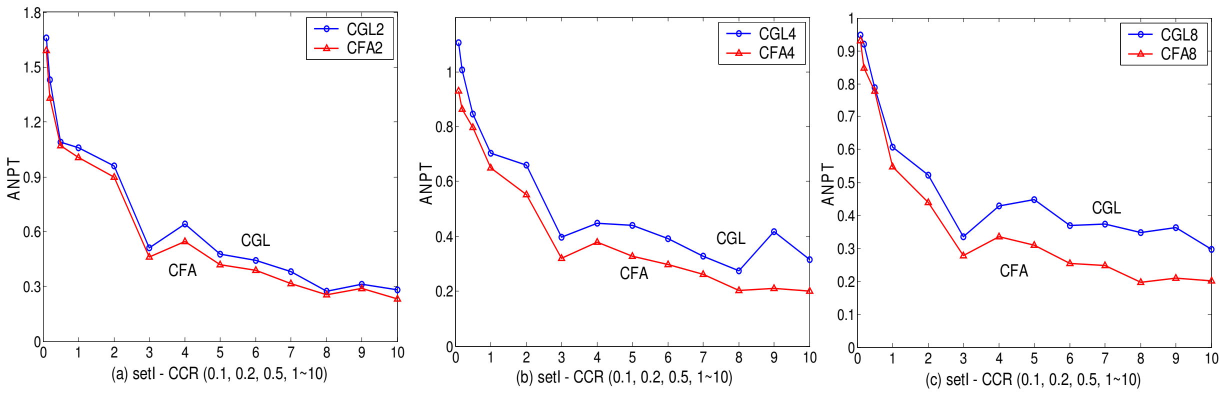


Figure 3.17: One Phase CGL vs. Two Phase CFA + CRLA for RANG setI graphs mapped to (a) 2-processor, (b) 4-processor, (c) 8-processor architectures.

CRLA. Figure 3.18 shows the results of comparing the one-step ready-list (RL) scheduling against the two-step DSC algorithm and CRLA. The experimental results of studying the effect of clustering are given in Figure 3.19 and Figure 3.20. In general, it can be seen that as the number of processors increases the difference between the algorithms performance becomes more apparent. This is because when the number of processors is small the merging algorithm has limited choices for the mapping of clusters and hence most tasks end up running on the same processor regardless of their initial clustering.

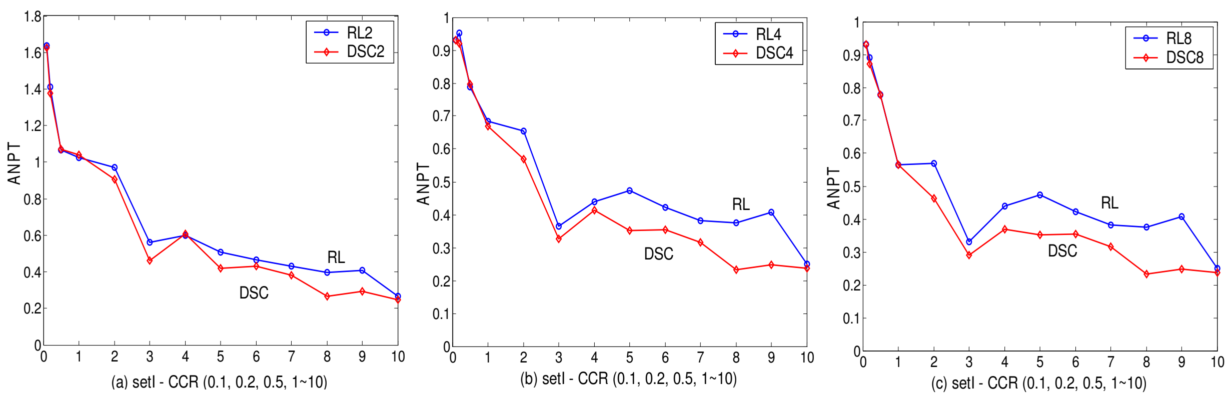


Figure 3.18: One Phase Ready-list Scheduling (RL) vs. Two Phase DSC for RANG setI graphs mapped to (a) 2-processor, (b) 4-processor, (c) 8-processor architectures.

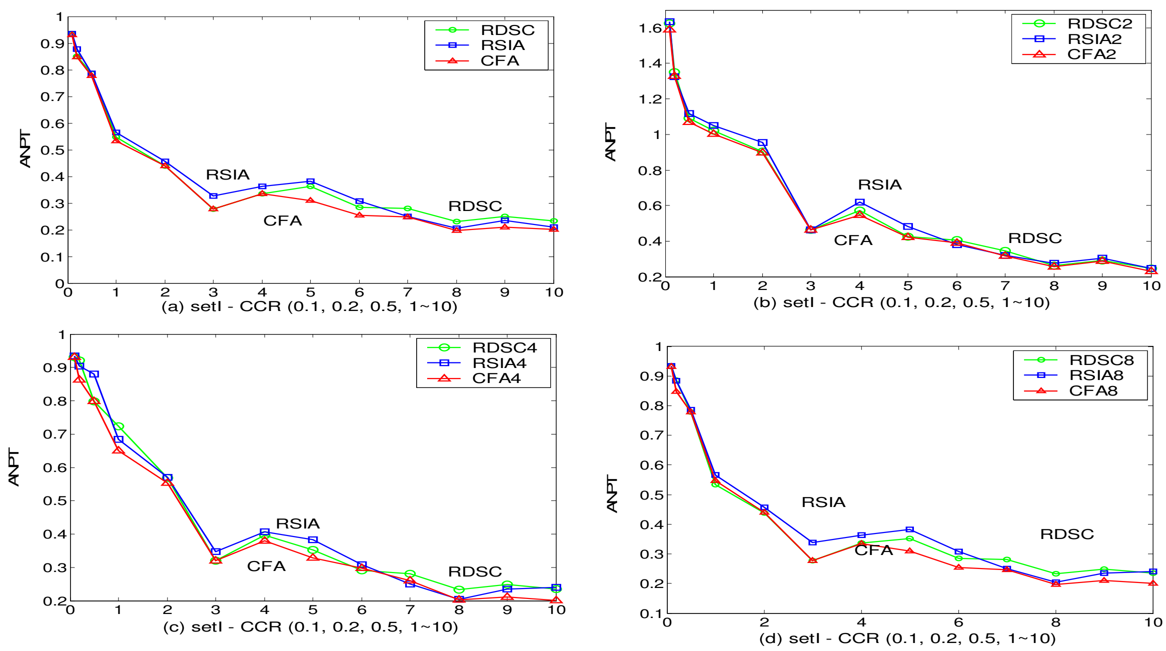


Figure 3.19: Average Normalized Parallel Time from applying RDSC, RSIA and CFA to RANG setI, (a) results of clustering algorithms, (b) results of mapping the clustered graphs onto a 2-processor architecture, (c) results of mapping the clustered graphs onto a 4-processor architecture, (d) results of mapping the clustered graphs onto an 8-processor architecture.

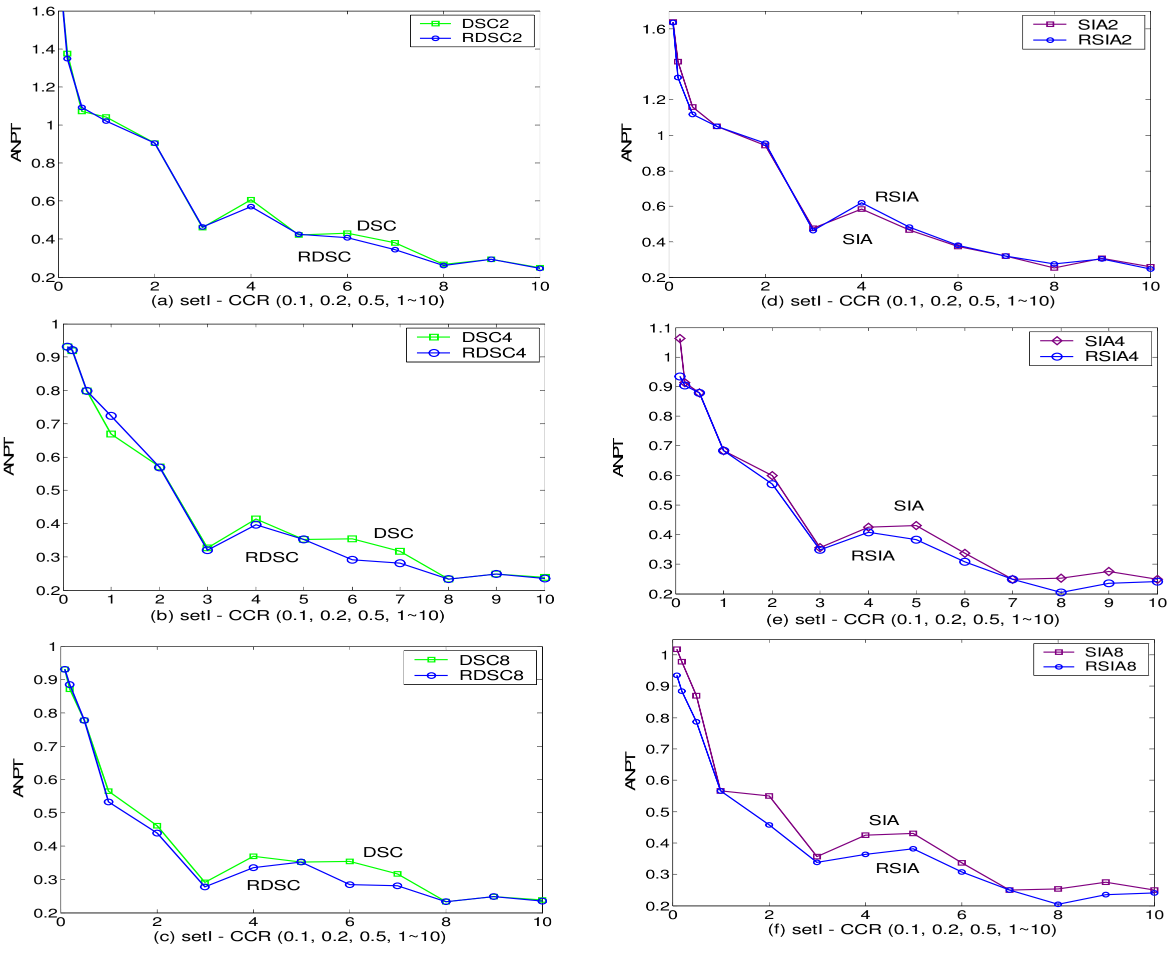


Figure 3.20: Effect of Clustering: Performance comparison of DSC, RDSC, SIA and RSIA on RANG setI graphs mapped to (a,d) 2-processor, (b,e) 4-processor, (c,f) 8-processor architecture using CRLA algorithm.

A quantitative comparison of these scheduling algorithms is also given in Tables 3.2 and 3.3. It can be seen that given two equally good one-step and two-step scheduling algorithms, the two-step algorithm gains better performance compared to the single-step algorithm. DSC is a relatively good clustering algorithm but not as efficient as CFA or RDSC. However, it can be observed that when used against a one-step scheduling algorithm, it still can offer better solutions (up to 14% improvement). It can also be observed that the better the quality of the clustering algorithms the better the overall performance of the scheduling algorithms. In this case CFA clustering is better than RDSC and RSIA and RDSC are RSIA and better than their deterministic versions.

Table 3.2: Performance Comparison of CFA, CGL, RDSC and RSIA

Algo.	RRL(%)				CGL(%)				RDSC(%)				RSIA(%)			
CFA + CRLA	>	=	<	% imp.	>	=	<	% imp.	>	=	<	% imp.	>	=	<	% imp.
RG	78	15	7	6.0	84.5	11.5	4	17.6	84	12	4	5.0	84	16	0	8.0
AG	46	27	27	10.5	64	28	8	11.1	44	56	0	11.4	50	50	0	3.0
RANG	82	10	8	9.0	100	0	0	18.0	82.3	12.7	5	5.0	92.7	7.3	0	6.0
Avg.	*69*	*17*	*14*	*8.5*	*83*	*13*	*4*	*15.6*	*70*	*27*	*3*	*7.1*	*75.6*	*24.4*	*0*	*5.7*

Table 3.3: Performance Comparison of DSC and RL

Algo.	RL(%)-RG set				RL(%)-AG set				RL(%)-RANG set				*Avg.*
-	>	=	<	%.imp	>	=	<	%.imp	>	=	<	%.imp	%imp.
DSC + CRLA	70.9	11	18.1	10.0	47	36	17	14.0	10	80	10	14.0	*9.3*

We have not presented the results of applying different metrics graphically, however, a summary of the results is as follows: for both test graph sets when tested with different merging algorithms (we used CRLA with three different priority metrics: topological sort ordering, static level and a randomly sorted priority list) each clustering algorithm did best with the original CRLA (using $blevel$ metric), moderately worse with static level and worst with random level. As shown in the literature the performance of the list scheduling algorithm highly depends on the priority metrics used and we observed that this was also the case for the original CRLA. Employing the information provided in clustering in original CRLA was also another strength for the algorithm. We also implemented an evolutionary based merging algorithm, however, we did not get significant improvement in the results. We conclude that as long as the merging algorithm utilizes the clustering information and does not restrict the processor selection to the idle processors at the time of assignment (local decision or greedy choice), it can efficiently schedule the clusters without further need for complex assignment schemes or evolutionary algorithms.

We also observed that in several cases where the results of clustering (parallel time) were equal, CFA could outperform RDSC and RSIA after merging (this trend was not observed for RDSC vs. DSC and RSIA vs. SIA). We also noted that there are occasional cases that two clustering results with different parallel times provide similar answers in the final mapping. There are also cases where a worse clustering algorithm (worse parallel time) finds better final results.

To find the reason for the first behavior, we studied the clustering results of each algorithm separately. CFA tends to use the most number of clusters when clustering tasks: there are several cases where two clusters could be merged with no effect on the parallel time. CFA keeps them as separate clusters. However, both RSIA and RDSC accept such clustering, i.e., when the result of clustering does not change the parallel time, and they tend to cluster as much as possible in the clustering step. Providing more clusters and clustering only those tasks with high data dependency gives more flexibility to the merging algorithm for mapping the results of CFA. This characteristic of CFA is the main reason that even in case of similar parallel time for clustering results, CFA is still capable of getting better overall performance.

For the second behavior we believe that the reason is behind the scheduling scheme (or task ordering) used in the clustering step. CFA uses an insertion based task scheduling and ordering, which is not the case for the other clustering algorithms. Hence, there are cases where similar clusterings of tasks end up providing different parallel times. This behavior was only observed for two cases. For a worse algorithm performing better at the end (only observed in the case of RSIA and SIA) the explanation is similar to that for the first behavior. A clustering algorithm should be designed to adjust the communication and computation time by changing the granularity of the program. Hence when a clustering algorithm ignores this fact and groups tasks together as much as possible, many tasks with little data dependencies end up together, and while this approach may give a better parallel time for clustering, it will fail in the merging step due to its decreased flexibility.

Observing these behaviors, we believe that the performance of clustering algorithms should only be evaluated in conjunction with the cluster-scheduling step as the clustering results do not determine the final performance accurately.

3.5 Summary and Conclusions

In this chapter we presented an experimental setup for comparing one-step scheduling algorithms against two-step scheduling (clustering and cluster-scheduling or merging) algorithms. We have taken advantage of the increased compile-time tolerance of embedded systems and have employed more thorough algorithms for this experimental setup. We have developed a novel and natural genetic algorithm formulation, called CFA, for multiprocessor clustering, as well as randomized versions, called RDSC and RSIA, of two well-known deterministic algorithms, DSC [149] and SIA [119], respectively. The experimental results suggest that a pre-processing or clustering step that minimizes communication overhead can be very advantageous to multiprocessor scheduling and two-step algorithms provide better quality schedules. We also studied the effect of each step of the two-step scheduling algorithm in the overall performance and learned that the quality of clusters does have a significant effect on the overall mapping performance. We also showed that the performance of a poor-performing clustering algorithm cannot be improved with an efficient merging algorithm. A clustering is not efficient when it either combines tasks inappropriately or puts tasks that should be clustered together in different clusters. In the former case (combining inappropriately), merging cannot help much because merging does not change the initial clustering. In the latter case, merging can sometimes help by combining the associated clusters on the same processor. However, in this case the results may not be as efficient as when the right tasks are mapped together initially. Hence, we conclude that the overall performance is directly dependent on the clustering step and this step should be as efficient as possible.

The merging step is important as well and should be implemented carefully to utilize information provided in clustering. A modified version of ready-list scheduling was shown to perform very well on the set of input clusters. We observed that in several cases the final performance is different than the performance of the clustering step (e.g., a worse clustering algorithm provided a better merging answer). This suggests that the clustering algorithm should be evaluated in conjunction with a merging algorithm as their performance may not determine the performance of the final answer. One better approach to compare the performance of the clustering algorithms may be to look at the number of clusters produced or cluster utilization in conjunction with parallel time. In most cases the clustering algorithm with a smaller parallel time and more clusters resulted in better results in merging as well. A good clustering algorithm only clusters tasks with heavy data dependencies together and maps many end nodes (sinks) or tasks off the critical paths onto separate clusters giving the merging algorithms more flexibility to place the not-so-critically located tasks onto physical processors.

Chapter 4

CHESS: Clustering-Oriented Heuristics for Heterogeneous Systems Scheduling

In the presence of multiple processors the heterogeneity of the processors has been shown to be an important attribute in improving the system's performance [42][45][95][126] [127][140]. Most of the multiprocessor scheduling approaches assume the target system is homogeneous. Multiprocessor scheduling is an NP-complete problem for homogeneous systems and adding the heterogeneity factor to the problem adds another dimension to the search space and makes the problem more complicated to handle. For example in the case of homogeneous scheduling, a task's finish time is the same on every processor if the start time is the same, however this is not true for heterogeneous multiprocessor systems. Simply because tasks have different execution rate on different processors and hence this is a parameter to take into account when computing the potential finish-time of tasks on different processors. Moreover, in heterogeneous computing environment the scheduling decisions are made not only on the number of processors but also on the capability of the processors. In this study we investigate a class of heterogeneous scheduling algorithm that utilize a pre-processing step of clustering. The clustering technique has proven very efficient for homogeneous multiprocessor scheduling [71] [49] and been widely and successfully applied to other applications such as parallel processing, load balancing and partitioning [89][102]. Clustering is also often used as a front-end to multiprocessor system synthesis tools. In this context, clustering refers to the grouping of actors into subsets that execute on the same processor. The purpose of clustering is thus to constrain the remaining steps of synthesis, especially scheduling, so that they can focus on strategic processor assignments. In this work, we address the challenges in employing clustering in heterogeneous scheduling, efficient approaches to these problems and compare our proposed approach against the state-of-the-art multiprocessors scheduling techniques for heterogeneous computing systems.

The remainder of this chapter is organized as follows: In Section 4.1 we present a short survey of the literature on the related scheduling algorithms. In Section 4.2, we provide a formal statement of the problem. In Section 4.3 we present our proposed solution CHESS (i.e. Clustering-based Heuristics for HEterogeneous Systems Scheduling) and its four different versions. In Section 4.4 we present the algorithms that we use to compare our heuristics against. In section 4.5, we present the input graphs we have used in our experiments. The summary of the experimental results and comparisons are presented in Section 4.6. We conclude the chapter with a summary in section 4.7.

4.1 Related Work

The multiprocessor mapping and scheduling problem in general has been extensively studied and various heuristics were proposed in the literature [1][22][41][53][59] [79][125] [129][130][140][143][149]. Most of these algorithms target homogeneous multiprocessor systems and only a few of the proposed heuristics support heterogenous processors.

One of the very first works in the field of heterogeneous computing was done by Menasce et al. in 1990 [95]. They investigated the problem of scheduling computations to heterogeneous multiprocessing environments. Their model of the heterogeneous system consists of one fast processor and a number of slower processors. They examined both dynamic and static scheduling techniques and used Markov chains to analyze the performance of different scheduling approaches. They assumed no communication delays in the employed DAGs. They investigated several schemes including: the LTF/MFT (Largest Task First/Minimizing finish-time), WL (Weighted Level), CPM (Critical Path Method) and HNF (Heavy Node First). The LTF/MFT algorithm works by picking the largest task from the ready tasks list and schedules it to the processor which allows the minimum finish-time, while the other three strategies select candidate processors based on the execution time of the task. They found that LTF/MFT significantly outperforms all the others including WL, CPM and HNF which means an efficient scheduling algorithm for heterogeneous systems should concentrate on reducing the finish-times of tasks. More thorough investigation is needed in the presence of IPC.

The group of algorithms that take the IPC into account can be classified as deterministic and probabilistic search algorithms. A large subset of deterministic algorithms are based on the classical *list scheduling heuristics*. Examples of these algorithms are El-Rewini and Lewis's *Mapping Heuristic* (MH) algorithm [41], Sih and Lee's well-known *Dynamic Level Scheduling* (DLS) heuristic [129], Iverson et al.'s *Levelized-min Time* (LMT) algorithm [59], Oh and Ha's *Best-Imaginary-Level* (BIL) heuristic [106], Radulescu and van Gemund's modified *Fast Critical Path* (FCP) and *Fast Load Balancing* (FLB) algorithms [114], Topcuoglu et al.'s *Heterogeneous Earliest-Finish-Time*

(HEFT) and *Critical-Path-on-a-Processor* (CPOP) algorithms [138][137] and Dogan et al.'s LDBS task-duplication-based algorithm [39]. Examples of probabilistic search algorithms are Shroff et al.'s genetic simulated annealing algorithm [125], Singh and Youssef's genetic algorithm in [130] and Wang et al's genetic algorithm based approach in [140]. MH algorithm considers the processor heterogeneity, interconnection topology and link contention. However in this work we are only interested in a network of fully-connected heterogeneous processors and hence we will not discuss the network topology related features in this work. In MH, each task is given a priority based on its $blevel$. Each ready task is then executed on a processor that gives the earliest finish time. The time complexity of this algorithm when link contention is not considered, is shown to be $O(v^2p)$ for v tasks and p processors.

The DLS algorithm is a compile-time, static list scheduling heuristic. It selects the ready task and the processor to run the task at each scheduling step. The selection is by finding the ready task and processor pair that have the highest dynamic level. Dynamic level is computed based on the static-level and the earliest start time (that is a function of data arrival and processor availability) metrics. The complexity of the DLS algorithm is shown to be $O(v^3p)$.

The LMT algorithm uses a two-phase approach. The first phase uses a technique called level sorting to order the subtasks based on the precedence constraints. The level sorting technique clusters subtasks that are able to execute in parallel. The second phase of the uses a min time algorithm to assign the subtasks level by level. The min time algorithm is a greedy method that attempts to assign each subtask to the fastest available processor. If the number of subtasks is more than the number of machines, then the smallest subtasks are merged until the number subtasks is equal to the number of machines. Then the subtasks are ordered in descending order by their average computation time. Each subtask is assigned to the machine with the minimum completion time. Sorting the subtasks by the average computation time increases the likelihood of larger subtasks getting faster machines. For a fully-connected graph the time complexity of the LMT algorithm is shown to be $O(v^2p^2)$.

BIL algorithm first computes the best-imaginary-level (BIL) of all tasks. The BIL of task v_i indicates the critical path length including the IPC overhead assuming that the node is scheduled on processor P_j, based on the critical assumption that all descendant nodes can be scheduled at best times. The BIL of a node is then used to computer a priority order over the nodes. Once the BIL is computed for each tasks, the algorithm computes a priority order. To select a task, the level of each task is adjusted to measure the *Best Imaginary Makespan* (BIM). For each task, there exist p different BIM values — one for each processor. Assuming there exist k runnable tasks at a scheduling stage, the priority of a task is defined as the kth smallest BIM value, or the largest finite BIM value if the kth smallest BIM value id undefined. The selected task is the one with the highest priority. Next the algorithm determines the optimal processor for the selected task. If the number of ready tasks is greater than the number of processors, the execution time becomes more important than the communication overhead and hence the BIM value is revised to reflect this. The processor with the highest revised value is selected. The time complexity of the BIL algorithm has shown to be $O(v^2p\log p)$.

The FLB algorithm [114] utilizes a list called the ready-list that contains all ready-nodes to be scheduled at each step. A ready-node is a node that all of its predecessors are already scheduled. At each step, the finish time of each ready-node of the ready-list is computed for all the processors and the task-processor pair that minimizes the finish time is selected. The complexity of the FLB algorithm is $O(vlogv + e)$.

HEFT and CPOP both have low-complexity and have been shown to have good performances. To assign a priority to a task, the HEFT algorithm uses the $blevel$ value of the task. Ready tasks are sorted with respect to decreasing $blevel$ values. The $blevel$s are computed based on mean computation and mean communication costs. Tie breaking is done randomly. The processor is then selected using the EFT values. The algorithm also uses an insertion based policy that considers the possible insertion of a task in an earliest idle slot time between two already-scheduled tasks on a processor. The time complexity of HEFT algorithm is shown to be $O(v^2p)$. The CPOP algorithm uses $blevel + tlevel$ to assign the task priority. Initially, ready tasks that are on the critical path are selected for scheduling. The critical path processor CPP is the one that minimizes the cumulative computation costs of the tasks on the critical path. If the selected task is on the critical path then it is assigned to the CPP, otherwise it is assigned to the processor that minimizes the EFT. The time complexity of CPOP algorithm is shown to be $O(ep)$.

The *Level Duplication Based Scheduling algorithm* (LDBS) is a list scheduling approach which uses replication to schedule a DAG onto a heterogeneous system. LDBS schedules tasks of the same topological level on the processors that minimize their finishing times. Let $List_j$ be the list of tasks with the same topological level j. At each iteration, the task $v \in List_j$ with the highest $blevel$ is selected. The algorithm is an insertion-based technique i.e. utilizes the idle periods between previously scheduled tasks. The immediate predecessors are replicated if finish time of v is reduced.

Another subset of these heuristics are based on the evolutionary algorithms such as Singh and Youssef's work in [130], where they assume infinite numbers of machines and communication links for each type. Other examples are [53] [140]. These proposed algorithms have been shown to be slow and not as well-performing as the list-scheduling based heuristics.

In this study, we are targeting embedded systems for signal and image processing applications. These applications are data-driven i.e. streams of data are coming in that need to be processed immediately to meet the real time constraints. Additionally there is a strict memory requirement for embedded system, that altogether makes the task-duplication based scheduling algorithms not suitable for scheduling such systems. Hence, in this study we base our comparison upon algorithms that do not replicate tasks. Amongst these techniques, HEFT algorithm has been experimentally shown [137][13] to outperform the other techniques in its class and hence we use HEFT to compare against our proposed techniques.

4.2 Problem Statement

The heterogeneous multiprocessor scheduling problem is the problem of mapping an application onto a set of heterogeneous processors. The application to be mapped onto the heterogeneous system is represented as a directed acyclic graph (see section 2.3). Each node of this graph represents a computation and has type. Each type of computation takes different execution time to complete on a given processor in a heterogeneous multiprocessor system. Edges represent the data transfer between the computation nodes and similarly in a heterogeneous network the communication time depends on the link on which the data is being transferred. Some heterogeneous scheduling algorithms only consider the processor heterogeneity, in this work we assume both link and processors are heterogeneous. One difficulty in scheduling of applications to heterogeneous system is that before the actual mapping of a task (communication edge) onto a processor (link) the execution time of the task (edge) is not known (as opposed to mapping a task (edge) onto a homogeneous multiprocessor system where the task execution is the same on every processor (link) in the system) and hence many classic and traditional scheduling techniques are not directly applicable to heterogeneous system scheduling. For example, in the classic list scheduling heuristic to make the priority list one needs to include the task's execution time in the priority metric (e.g. blevel, tlevel or static level) calculation. Additionally some of the priority metrics become obsolete, for example the *Earliest Start Time* or *EST* metric does not provide much useful information in case of heterogeneous multiprocessor because one task with a larger EST on a faster processor can end up finishing faster on that processor than on a processor on which it has a smaller EST. To take advantage of useful metrics such as blevel or tlevel for the heterogeneous systems, one common approach is to substitute the exact execution time with the average execution time. In this work we investigate the use of other values such as the worst case execution time or the median value. Another scheduling technique that becomes hard to employ in the context of the heterogeneous processors is the clustering-based scheduling (or two-step scheduling). In the clustering-based scheduling, tasks are initially clustered to form a more balanced and coarse-grain graph and then are mapped to the target architecture. The evaluation of clustering is based on the assumption of availability of infinite number of homogeneous processors. The justification for the clustering is that if in the presence of infinite number of processors two tasks end up running on the same processor then they should be assigned to the same processor when the number of processors is finite [119]. This justification does not necessarily hold in the case of heterogeneous processors mainly because the effective communication time between two tasks depends on which processors they are running on and that which link is transferring the data. However, clustering has shown promising results when used as a pre-processing step to scheduling or synthesis [24][69][71] and it is an efficient technique in reducing the search space effectively. Hence, in this work we target a class of heterogeneous algorithms that are based on the two-step scheduling approach of clustering and cluster-scheduling (or merging). For the clustering step, we use the CFA algorithm that was introduced in Chapter 3. In CFA (or any other clustering algorithm) the set of clusters need to be scheduled on the virtual processors to evaluate the effectiveness of the clustering and this requites the information about the computation cost of the tasks or communication cost of edges. Since we are targeting a heterogeneous system, the computation cost of each task depends on the processor it is mapping to and that is the information that is not available in the clustering phase, hence we need to use an estimate value for the computation cost of the tasks. We used four different estimates for the computation/ommunication cost in the CFA that are as follows (defined only for computation cost – similar definition applies to communication cost):

- *Average Computation Cost* (ACC): The sum of the tasks computation cost on all processors divided by the number of processors,

- *Median of the Computation Cost* (MCC): The middle value in the set of all computation cost values arranged in increasing order,
- *Random Computation Cost* (RCC): Randomly pick a computation cost for the task,
- *Worst Case computation Cost* (WCC): The worst case value among all processors.

For the cluster-scheduling or merging step we used two different techniques, one deterministic approach and one GA-based technique. We also, proposed a combined clustering/merging solution that uses the merging technique as the fitness evaluation of the CFA. These techniques are further explained in the following section.

4.3 CHESS: Our proposed solution

CHESS is a class of heterogeneous multiprocessing scheduling algorithms that we have proposed and are based on the idea of two-step scheduling. CHESS algorithms consist of two parts, clustering and cluster-scheduling. There are four heuristics in the CHESS class. The first two, employ an adapted version of the CFA [68] for the clustering phase and use a deterministic merging and a GA-based merging for the merging phase. We call these algorithms *Separate-Clustering Deterministic Merging (SCDM)* and *Separate-Clustering GA-based Merging algorithms (SCGM)* respectively. The last two heuristics do not have a separate clustering evaluation phase, in other words the clustering is only evaluated based on how good a merging results it generates. These approaches, use CFA to generate clusters and them apply a merging algorithm to evaluate thee clusters. We call these techniques *Combined Clustering and Merging (CCM)* techniques. Our first CCM algorithm uses a determinist merging algorithm and the second one uses GA-based technique. We call these algorithms *Combined-Clustering Deterministic Merging (CCDM)* and *Combined-Clustering GA-based Merging (SCGM)* algorithms respectively. A brief description of these heuristics are as follows:

- Separate-Clustering Deterministic Merging (SCDM)

  1. Apply CFA to form the clusters, evaluate each cluster using an estimation of computation and communication cost,
  2. Map clusters onto the heterogeneous system using a deterministic algorithm.

- Separate-Clustering GA-based Merging (SCGM)

  1. Apply CFA to form the clusters, evaluate each cluster using an estimation of computation and communication cost,
  2. Map clusters onto the heterogeneous system using a genetic algorithm-based merging.

- Combined-Clustering Deterministic Merging (CCDM)

  1. Modify CFA to take a deterministic merging algorithm as its fitness function.

- Combined-Clustering GA-based Merging (CCGM)

  1. Implement the clustering and merging as a nested GA algorithm, where the outer GA forms the clusters and the inner GA merges them and evaluates them.

```
1   Set the computation costs of tasks and communication costs of edges with estimate values.
2   Compute $blevel$ for all tasks by traversing graph upward, starting from the exit task.
3   Sort the tasks in a scheduling list by non-increasing order of $blevel$ values.
4   WHILE there are unscheduled clusters in the list DO
5       Select the first task, $v_i$, from the list for scheduling.
6       FOR each processor $p_k$ in the processor-set DO
7           Compute $\tau_{par}$ if $v_i$ and its cluster are mapped onto $p_k$.
8       ENDFOR
9       Assign task $v_i$ and its cluster to the processor $p_k$ that minimizes the $\tau_{par}$ of the graph
            (break the ties by assigning $v_i$ to the processor which minimizes the $EFT(v_i, p_k)$)
10      Remove all the tasks within the newly-assigned cluster from the list.
11  ENDWHILE
```

Figure 4.1: An outline of the deterministic Merging algorithm.

4.3.1 CHESS-SCDM: Separate Clustering and Deterministic Merging

CHESS-SCDM performs the clustering and cluster-scheduling in two separate phases. It first uses a slightly modified version of the CFA — the modification is in using the four above mentioned estimates of computation cost i.e. ACC, MCC, WCC and RCC instead of the actual computation cost that is not available — and finds the best clustering i.e. a clustering that minimizes the parallel time. Once a clustering is found a deterministic merging algorithm is applied to map the clusters onto the given limited number of heterogeneous processors. An outline of the deterministic merging is given in Figure 4.1. In the homogeneous version of the cluster-merging algorithm (see Chapter 3) the start time of a task on a processor was only dependent on the existing tasks schedule, while in the heterogeneous case the processor selection and the corresponding task's execution time on that machine can affect the start time. Hence to break the ties we use the *Earliest Finish Time* or *EFT* measure.

4.3.2 CHESS-SCGM: Separate Clustering and GA-based Merging

Similar to the SCDM, SCGM runs the modified version of the CFA algorithm first and once the best clustering is found a genetic algorithm is applied to the clustering to find an optimized mapping for it. Some details of the GA Merging algorithm (GM) is as follows:

Solution Representation: Solutions in GM represent the assignment of clusters to PEs. These assignments are encoded in an integer array of size n_c (where n_c is the number of clusters that the modified CFA has generated). Each element of the array determines the PE number that the associated cluster is mapped to.

Initial Population: The initial population of GM consists of POP_SIZE (to be set experimentally) assignment arrays (for one cluster). For each solution, an integer number between 1 and n_p is randomly assigned to each column of the assignment arrays.

Fitness Evaluation: To evaluate how good a mapping is, a scheduling algorithm is applied to each mapping. Since the task (or cluster) to PE mapping is known, the scheduling algorithm only needs to orders tasks on each processor according to a priority metric ($blevel$) here and compute the longest path.

The SCGM returns a mapping that provides the smallest parallel time for the clustering found by CFA.

4.3.3 CHESS-CCDM: Combined Clustering and Deterministic Merging

CCDM is also based on the modified version of CFA that uses the deterministic merging (DM) algorithm introduced in section 4.3.1 as the fitness value. The calcula-

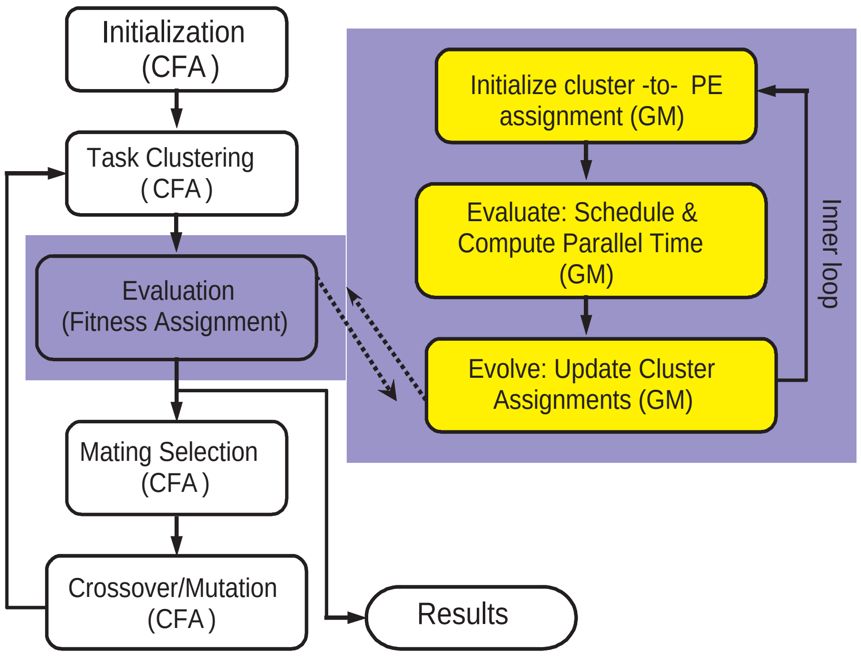

Initialization (CFA)
Task Clustering (CFA)
Evaluation (Fitness Assignment)
Mating Selection (CFA)
Crossover/Mutation (CFA)
Results
Initialize cluster -to- PE assignment (GM)
Evaluate: Schedule & Compute Parallel Time (GM)
Evolve: Update Cluster Assignments (GM)
Inner loop

Figure 4.2: Flow of nested CCGM algorithm.

tion of priorities in merging part of the CCDM algorithm requires the algorithm to use estimated values for the computation costs.

4.3.4 CHESS-CCGM: Combined Clustering and GA-based Merging

CCGM is a nested genetic algorithm where the outer GA employs CFA to form clusters and the inner GA uses the genetic merging (GM) algorithm introduced in section 4.3.1 as the fitness value. In CCGM algorithm no estimated values are needed. An outline of this nested genetic algorithm is given in Figure 4.2.

4.4 The Heterogeneous-Earliest-Finish-Time (HEFT) Algorithm

The HFET algorithm (4.3) is an application scheduling algorithm for a bounded number of heterogeneous processors, which has two major phases: a task prioritizing phase for computing the priorities of all tasks and a processors selection phase for selecting the tasks in the order of their priorities and scheduling each selected tasks on its "best" processor, which minimizes the task's finish time.

Task Prioritizing Phase — This phase requires the priority of each task to be set with the $blevel$ rank value, which is based on mean computation and mean communication costs. The task list is generated by sorting the tasks by decreasing order of $blevel$. Tie-breaking is done randomly. The decreasing order of $blevel$ values provides a topological order of tasks, which is a linear order that preserve the precedence constraints.

Processor Selection Phase — For most of the task scheduling algorithm, the earliest available time of a processor p_j for a task execution is the time when p_j completes the execution of its last assigned task. However, the HFET algorithm has an insertion-based policy which considers the possible insertion of a task in an earlier idle time slot between two already scheduled tasks on a processor. Such insertion is only performed when two conditions are met: first, the length of the idle time slot, i.e. the difference between execution start time and finish time of two tasks that were consecutively scheduled on the same processor, should be at least as large as the computation time of the candidate task to be inserted, and second, Additionally, this insertion should not violate any precedence constraints.

The HFET algorithm has an $O(ep)$ time complexity for e edges and p processors. For a dense graph when the number of edges is proportional to $O(v^2)$ (v is the number of tasks), the time complexity is on the order of $O(v^2p)$.

```
1  Set the computation costs of tasks and communication costs of edges with mean values.
2  Compute $blevel$ for all tasks by traversing graph upward, starting from the exit task.
3  Sort the tasks in a scheduling list by non-increasing order of $blevel$ values.
4  WHILE there are unscheduled tasks in the list DO
5     Select the first task, $v_i$, from the list for scheduling.
6     FOR each processor $p_k$ in the processor-set DO
7        Compute $EFT(v_i, p_k)$ value using the insertion-based scheduling policy.
8     ENDFOR
9     Assign task $v_i$ to the processor $p_k$ that minimized the $EFT$ of task $v_i$
10 ENDWHILE
```

Figure 4.3: An outline of the HEFT algorithm.

4.4.1 The Randomized HEFT (RHEFT) Algorithm

Three of the algorithms that we proposed earlier in this chapter are based on genetic algorithms where elements in a given set of solutions are probabilistically combined and modified to improve the fitness of populations. The algorithm that we have chosen for comparison (HEFT) on the other hand is a fast deterministic algorithm, hence to be fair in comparison of these algorithms, we have implemented a randomized version of the HEFT algorithm employing a similar method as described in section 3.2.2. The resulting randomized algorithm (RHEFT), like the GA, can exploit increases in additional computational resources (compile time tolerance) to explore larger segments of the solution space.

Since the major challenge in scheduling algorithms is the selection of the "best" task and the "best" processor in order to minimize the parallel execution time of the scheduled task graph, we have incorporated randomization into to the i) task selection only, ii) processor selection only, iii) task and processor selection together, when deriving the randomized version of HEFT i.e. RHEFT algorithm.

In the *task-only* randomized version of HEFT, we first sort all the tasks based on their $blevel$ i.e. the sorting criteria of the algorithm. The first element of the sorted list — the candidate tasks to be schedule — then is selected with probability p, where p is a parameter of the randomized algorithm (we call p the *randomization parameter*); if this element is not chosen, the second element is selected with probability p; and so on, until some element is chosen, or no element is returned after considering all the elements in the list. In this last case (no element is chosen), a random number is chosen from a uniform distribution over $\{0, 1, ..., |T| - 1\}$ (where T is the set of ready tasks that have not been scheduled yet).

In the *processor-only* randomized version of HEFT, we first compute the $EFT(v_i, p_j)$ for all the processors and then sort all the processor based on the EFT values that they provide for the tasks in an increasing order. The first element of the sorted list — the processor to be selected to schedule the task on it — then is selected with probability p, and so on.

In the *combined tasks-processor* randomized version of HEFT, we apply the randomization parameter to the selection of both tasks and processors in the algorithm. The employed method is as described above.

4.5 Input Benchmark Graphs

In this study, all the heuristics have been tested with a large set of randomly generated input graphs that were generated using TGFF, a publicly available random graph generator from Princeton university [33]. A set of parameters that we varied to generate a wide-variety of random graphs are as follows:

- $|V|$ or number of nodes. We varied the number of nodes as follows: $|V| = \{20, 40, 60, 80, 100, 200, 400\}$,

- CCR or the communication to computation ratio (see 3.6). CCR is the average communication cost by the average computation cost. A high value of CCR means that there is little parallelism in the graph and that the application is dominated by the communication costs. A small value of the CCR implies high level of parallelism in the graph and a computation-intensive application. The CCR values used are as follows: $CCR = \{0.1, 0.5, 1, 5, 10\}$,

- in-degree or the number of incoming edges. The in-degree values used are as follows: $in - degree = \{1, 2, 3, 4, 5, v\}$,

- out-degree or the number of outgoing edges. The out-degree values used are as follows: $out - degree = \{1, 2, 3, 4, 5, v\}$.

The parameters we have employed and the resulting DAGs are in accordance with the parameters and DAGs used in similar experiments in the literature.

4.6 Experimental Results

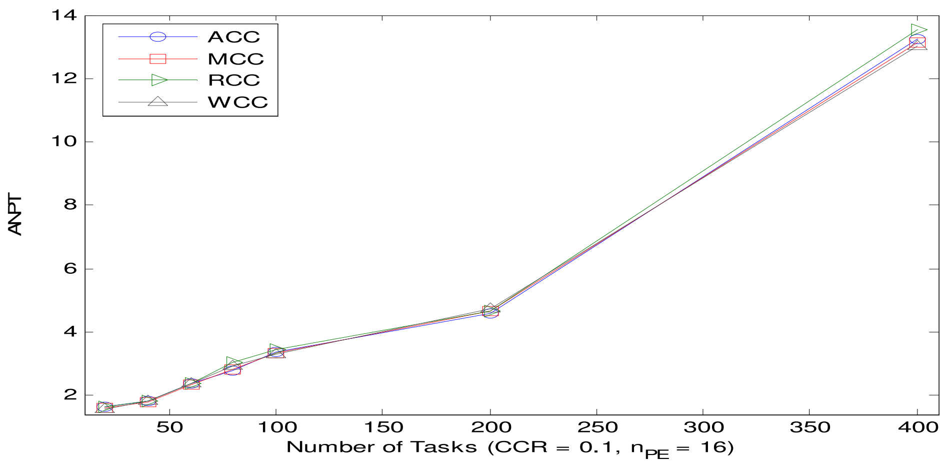
4.6.1 Performance study with respect to computation cost estimates

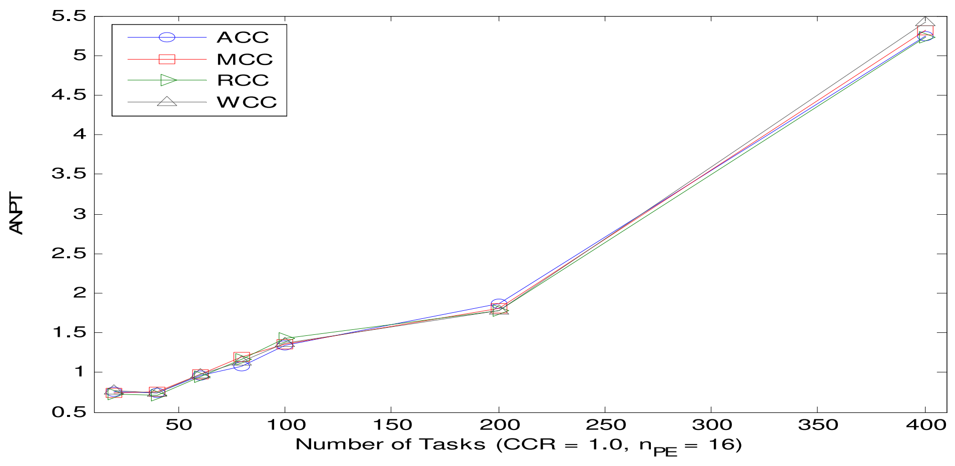
Our first set of experiments are carried out with the purpose of learning more about the effect of different computation cost estimates in the pre-processing step of clustering. More specifically, we are interested to know which computation cost estimate (ACC, MCC, RCC or WCC) when used in the clustering step generates better clustering of the graph. A better clustering is a clustering that when used as an input in the second step of merging provides a better final mapping onto the target architecture with the smallest parallel time. For this study we employed the two separate clustering (SC) algorithms introduced in Section 4.3, i.e. SCGM and SCDM. We first ran the CFA algorithm on our data set 4 times, each time using one of the following 4 values for computation costs, ACC, MCC, RCC and WCC. Once the best clustering in each case was found we then applied the merging algorithms (Deterministic Merging and GA Merging) and found the final mapping's parallel time.

Figure 4.4 shows the parallel time achieved by SCGM algorithm when different cost estimates are used in the clustering step of this algorithm. The x-axis shows the number of tasks and the y-axis shows the average normalized parallel time (ANPT). As it can be seen in the Figure the resulting parallel times are very close to each other and there is no obvious superiority for one cost estimate over the other. We have presented a subset of ANPT obtained from running SCGM on 2, 4, 8 and 16 processors for CCR values of 0.1, 1 and 10 in Table 4.1.

Again as it can be observed from Table 4.1 the ANPT values for different cost estimates are very close. Once we compared all the values we noted that the best NPT for $CCR < 1$ are obtained with ACC estimates while for $CCR \geq 1$ are obtained with ACC estimate. And the Worst values are generated when using random estimates i.e. RCC. On average the best values (using ACC estimate) are up to 3.2% better than the worst PT (using other estimates) computed.

For the SCDM algorithm the cost estimate values also play a role in the merging phase since the DM algorithm needs to use an estimated values for costs to compute





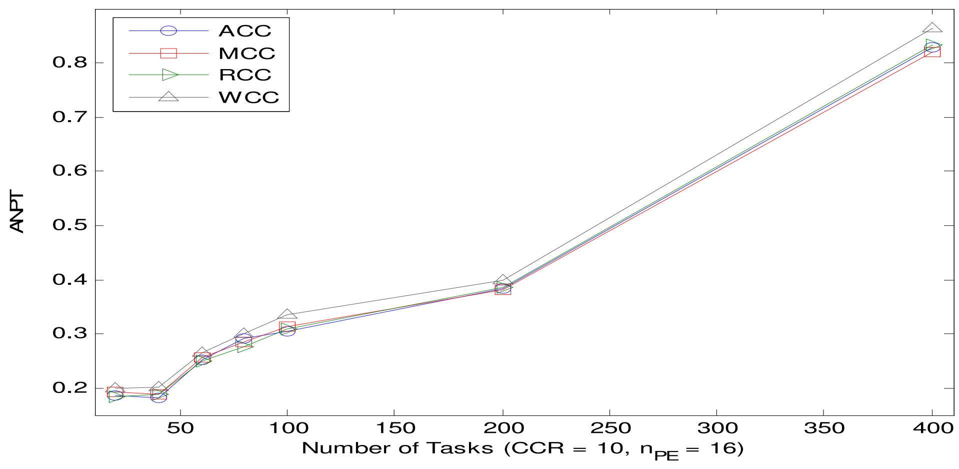


Figure 4.4: Effect of different cost estimates on parallel time using SCGM algorithm for CCR values of 0.1, 1 and 10 and 16 processors.

Table 4.1: ANPT values using different cost estimates with SCGM algorithm.

		0.1				**1.0**				**10.0**			
n_{PE}	$\|V\|$	A	M	R	W	A	M	R	W	A	M	R	W
2	20	2.15	2.15	2.14	2.13	1.04	1.05	1.04	1.05	0.21	0.20	0.20	0.22
	40	2.51	2.53	2.51	2.54	1.08	1.10	1.10	1.10	0.19	0.19	0.19	0.20
	60	3.78	3.81	3.77	3.82	1.65	1.66	1.66	1.66	0.29	0.30	0.30	0.30
	80	4.89	4.85	4.85	4.83	2.17	2.17	2.15	2.16	0.37	0.37	0.37	0.38
	100	6.34	6.36	6.35	6.29	2.83	2.79	2.85	2.82	0.48	0.48	0.47	0.48
	200	8.75	9.06	9.00	9.14	3.84	3.77	3.80	3.88	0.61	0.61	0.62	0.60
	400	26.35	26.02	26.12	26.21	12.14	12.07	11.95	12.09	1.93	1.94	1.93	1.93
4	20	1.71	1.66	1.64	1.70	0.86	0.86	0.85	0.87	0.19	0.20	0.20	0.21
	40	1.99	1.97	1.97	1.94	0.88	0.86	0.89	0.90	0.19	0.19	0.19	0.20
	60	2.78	2.80	2.89	2.79	1.26	1.27	1.23	1.25	0.26	0.26	0.26	0.27
	80	3.44	3.41	3.48	3.53	1.55	1.57	1.54	1.56	0.32	0.31	0.32	0.31
	100	4.36	4.37	4.37	4.32	1.98	2.02	1.99	1.98	0.38	0.38	0.38	0.46
	200	6.34	6.33	6.22	6.16	2.71	2.78	2.71	2.72	0.47	0.47	0.47	0.47
	400	18.63	18.03	18.15	18.44	8.29	8.23	8.38	8.15	1.33	1.37	1.38	1.42
8	20	1.54	1.53	1.51	1.52	0.75	0.74	0.76	0.77	0.19	0.19	0.19	0.20
	40	1.71	1.70	1.72	1.74	0.78	0.76	0.74	0.78	0.19	0.18	0.18	0.20
	60	2.23	2.26	2.25	2.31	0.98	0.99	0.99	1.00	0.25	0.25	0.25	0.26
	80	3.02	2.99	2.88	3.00	1.23	1.25	1.23	1.25	0.28	0.28	0.28	0.29
	100	3.61	3.43	3.62	3.59	1.54	1.54	1.54	1.50	0.32	0.32	0.32	0.34
	200	4.91	4.67	4.80	5.01	1.97	1.99	1.91	1.98	0.40	0.38	0.38	0.39
	400	12.19	12.21	12.75	12.05	5.38	5.42	5.87	5.40	0.91	0.92	0.90	0.90
16	20	1.63	1.59	1.65	1.56	0.75	0.74	0.73	0.77	0.19	0.19	0.18	0.20
	40	1.84	1.79	1.81	1.81	0.74	0.75	0.71	0.74	0.18	0.19	0.19	0.20
	60	2.39	2.35	2.37	2.39	0.95	0.97	0.95	0.97	0.25	0.26	0.25	0.26
	80	2.80	2.84	3.03	2.93	1.08	1.19	1.17	1.14	0.29	0.29	0.28	0.30
	100	3.35	3.32	3.46	3.29	1.34	1.36	1.43	1.37	0.30	0.31	0.31	0.33
	200	4.59	4.65	4.67	4.73	1.86	1.80	1.77	1.79	0.38	0.38	0.39	0.40
	400	13.25	13.14	13.53	13.02	5.25	5.32	5.23	5.42	0.83	0.82	0.83	0.86

the priority metric values. Hence, there are 16 combinations of estimated values used in clustering and merging steps as follows:

$\{ACC, MCC, RCC, WCC\}_{SC} \times \{ACC, MCC, RCC, WCC\}_{DM}$.

Figures 4.5 and 4.6 show the parallel time achieved by SCDM algorithm when different cost estimates are used in the clustering step of this algorithm for $8-$ and $16-$processor architecture. As it can be seen in the Figures there are 16 graphs in each plot. Each graph is associated with two different cost estimates; one for clustering and one for deterministic merging. For example, the graph labeled AM uses ACC estimates and MCC estimates in clustering and merging steps respectively. It can be observed from these Figures that all the estimates, provide nearly similar values which shows that perhaps the final results (NPTs) are not very sensitive to the cost estimates in the clustering

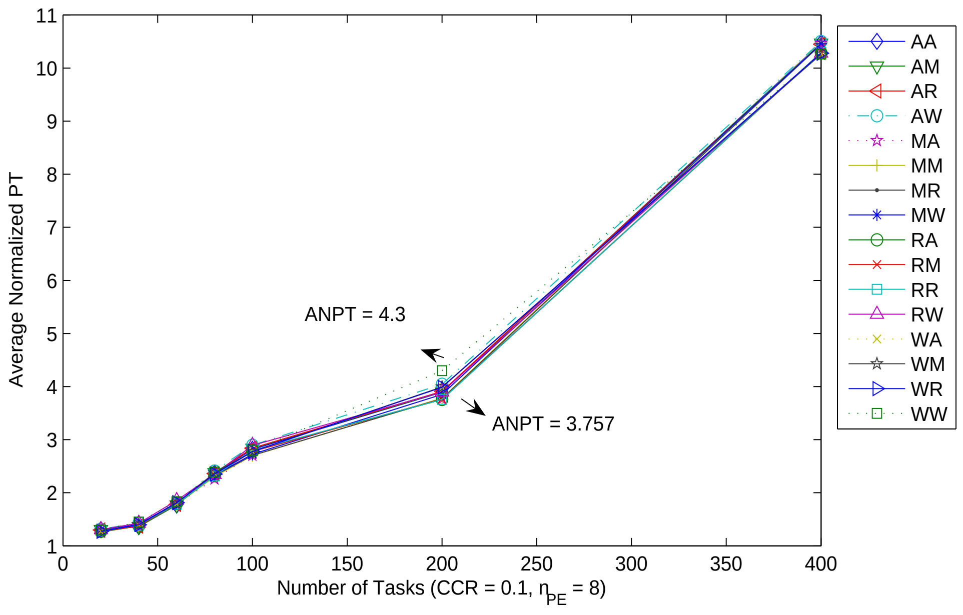


Figure 4.5: Effect of different cost estimates on parallel time using SCDM algorithm for CCR value of 0.1 and 8 processors.

and/or merging step. Upon comparison of all the 16 values obtained for each configuration over all the graph data set, we observed that the best NPT for $CCR < 1$ are obtained for the WA combination i.e. with WCC estimates for clustering and ACC estimates for merging. For $CCR \geq 1$ minimum values are obtained using the AA estimates, i.e. ACC estimates for both clustering and merging. And the Worst values are generated when using RCC and WCC estimates in the merging step. On average the best values (using WA and AA estimates) are up to 1.53% better than the worst PT (using other estimates) computed.

In conclusion, while the difference between the best and worst results for SCGM and SCDM using different estimates is not very large (3.2% at most), both results confirm that the use of WCC estimates for CCR values < 1 and ACC estimates for CCR values ≥ 1 in the clustering step provide the best results. One explanation is that when

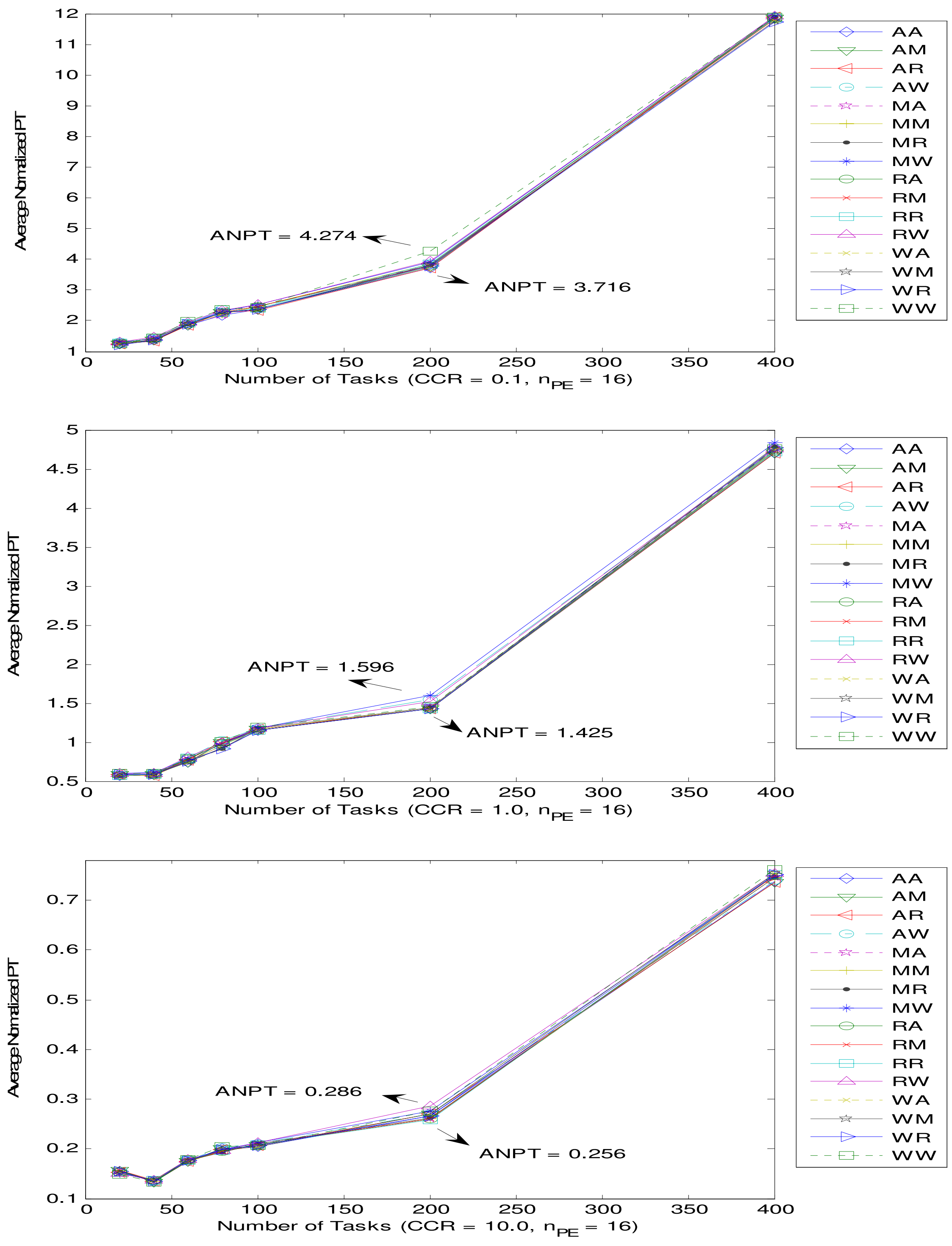


Figure 4.6: Effect of different cost estimates on parallel time using SCDM algorithm for CCR values of 0.1, 1 and 10 and 16 processors.

CCR is smaller than 1 the application is computation-intensive and suitable for parallelism and hence the clustering should internalize only a small number of edges and form many clusters with small number of tasks in them. Consequently using ACC/MCC or RCC values may make the clustering algorithm to group more tasks together thinking the costs are smaller than what they are, not utilizing the available parallelism and resulting in over-clustering or poor clustering. When the CCR is ≥ 1, the application is more or less communication-intensive which means there is little parallelism available and hence clustering algorithm does not over cluster. Hence the ACC values suffice to form a balanced clustering. The only drawback is that generating smaller clusters (large in number, small in size) makes the time to merge the clusters relatively longer.

4.6.2 Performance study of different heterogeneous scheduling algorithms

In this section we present the performance comparison of our proposed clustering based heterogeneous scheduling algorithms against one another and also HEFT algorithm. First, we study the effectiveness of separate clustering technique versus combined clustering technique by comparing SCDM against CCDM and SCGM against CCGM. Basically, we use the two different clustering technique with the same merging algorithm (first with the DM algorithm and next with the GM algorithm). The results for a subset of configurations are given in Figure 4.7. As it can be observed from the Figure that CCDM algorithm outperforms the SCDM algorithm most of the time. A quantitative comparison of these two algorithms for a subset of benchmarks and configurations is given in Table 4.2. A quantitative analysis over all the benchmarks and configurations shows that

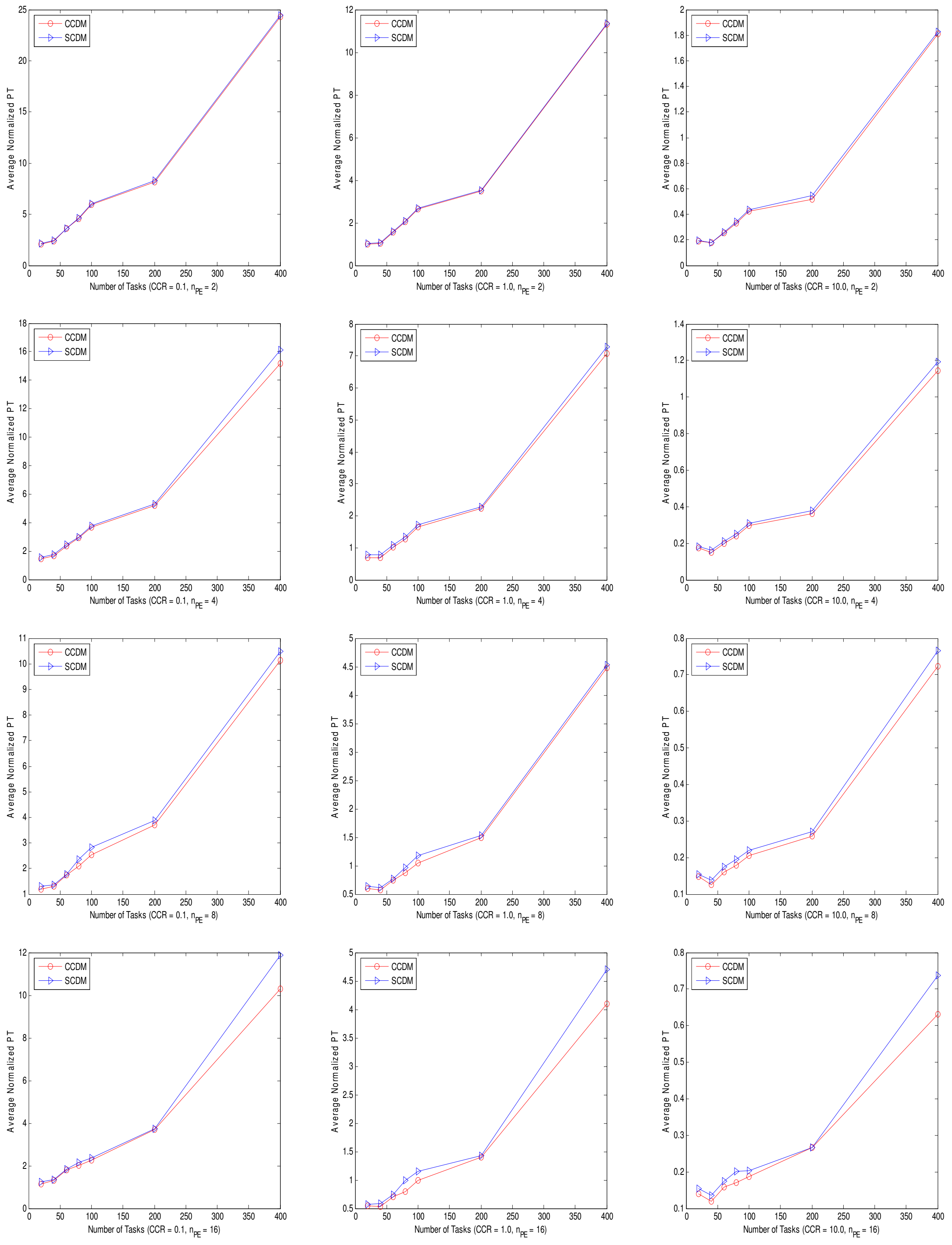


Figure 4.7: Performance comparison of two different clustering approach; separate clustering and deterministic merging vs. combined clustering and deterministic merging (i.e. CCDM vs. SCDM) on 2, 4 and 8 and 16 processors.

Table 4.2: Performance Comparison of CCDM against SCDM algorithm

Algo.	SCDM(%)															
	$n_{PE} = 2$				$n_{PE} = 4$				$n_{PE} = 8$				$n_{PE} = 16$			
CCDM	$>$	$=$	$<$	% imp.	$>$	$=$	$<$	% imp.	$>$	$=$	$<$	% imp.	$>$	$=$	$<$	% imp.
0.1	76.4	19.0	4.6	2.2	78.3	14.8	7.0	4.3	79.5	6.7	13.8	4.1	74.5	2.8	22.7	4.6
1.0	80.4	18.7	0.9	2.2	76.2	17.2	6.6	3.2	74.9	7.9	17.2	5.2	71.3	9.3	19.4	5.9
10.0	61.5	38.1	0.4	2.3	59.8	39.3	0.9	3.9	70.0	26.4	3.5	6.6	71.8	25.9	2.3	8.4
Avg.	72.8	25.3	2.0	2.2	71.4	23.7	4.8	3.8	74.8	13.7	11.5	5.3	72.5	12.7	14.8	6.3

the combined clustering approach using deterministic merging is more efficient that the separate clustering approach using a similar merging technique. Similarly, we compared CCGM and SCGM (the separate clustering and combined clustering techniques that employ a GA-based merging technique). Figure 4.8 and Table 4.3 provide the detailed results for a subset of our benchmarks. These results again confirm that the combined clustering technique is more efficient that the separate clustering technique (here used with a GA-based merging algorithm). More specifically, CC-based algorithms outperform SC-based algorithms on average by 7.0% over 88.7% of the time. Now, to find out which merg-

Table 4.3: Performance Comparison of CCGM against SCGM algorithm

Algo.	SCGM(%)															
	$n_{PE} = 2$				$n_{PE} = 4$				$n_{PE} = 8$				$n_{PE} = 16$			
CCGM	$>$	$=$	$<$	% imp.	$>$	$=$	$<$	% imp.	$>$	$=$	$<$	% imp.	$>$	$=$	$<$	% imp.
0.1	90.2	4.5	5.3	5.7	87.3	0.4	12.3	8.9	81.1	0.8	18.0	10.7	80.3	0.0	19.7	10.0
1.0	87.7	6.6	5.7	4.6	85.2	2.0	12.7	7.4	83.2	0.8	16.0	10.3	79.5	0.4	20.1	9.4
10.0	69.3	25.8	4.9	5.0	70.9	15.6	13.5	6.5	74.2	11.1	14.8	10.1	72.5	9.8	17.6	8.8
Avg.	*82.4*	*12.3*	*5.3*	*5.1*	*81.1*	*6.0*	*12.8*	*7.6*	*79.5*	*4.2*	*16.3*	*10.4*	*77.5*	*3.4*	*19.1*	*9.4*

ing algorithm performs better we compared the two proposed merging techniques once with separate clustering and once combined with the clustering. First we ran SCDM and SCGM algorithm against each other. The results for a subset of configurations are given in Figure 4.9. As it can be observed from the Figure, SCDM algorithm constantly outperforms the SCGM algorithm. A quantitative comparison of these two algorithms for a subset of benchmarks and configurations is given in Table 4.4. The date given in Table 4.4

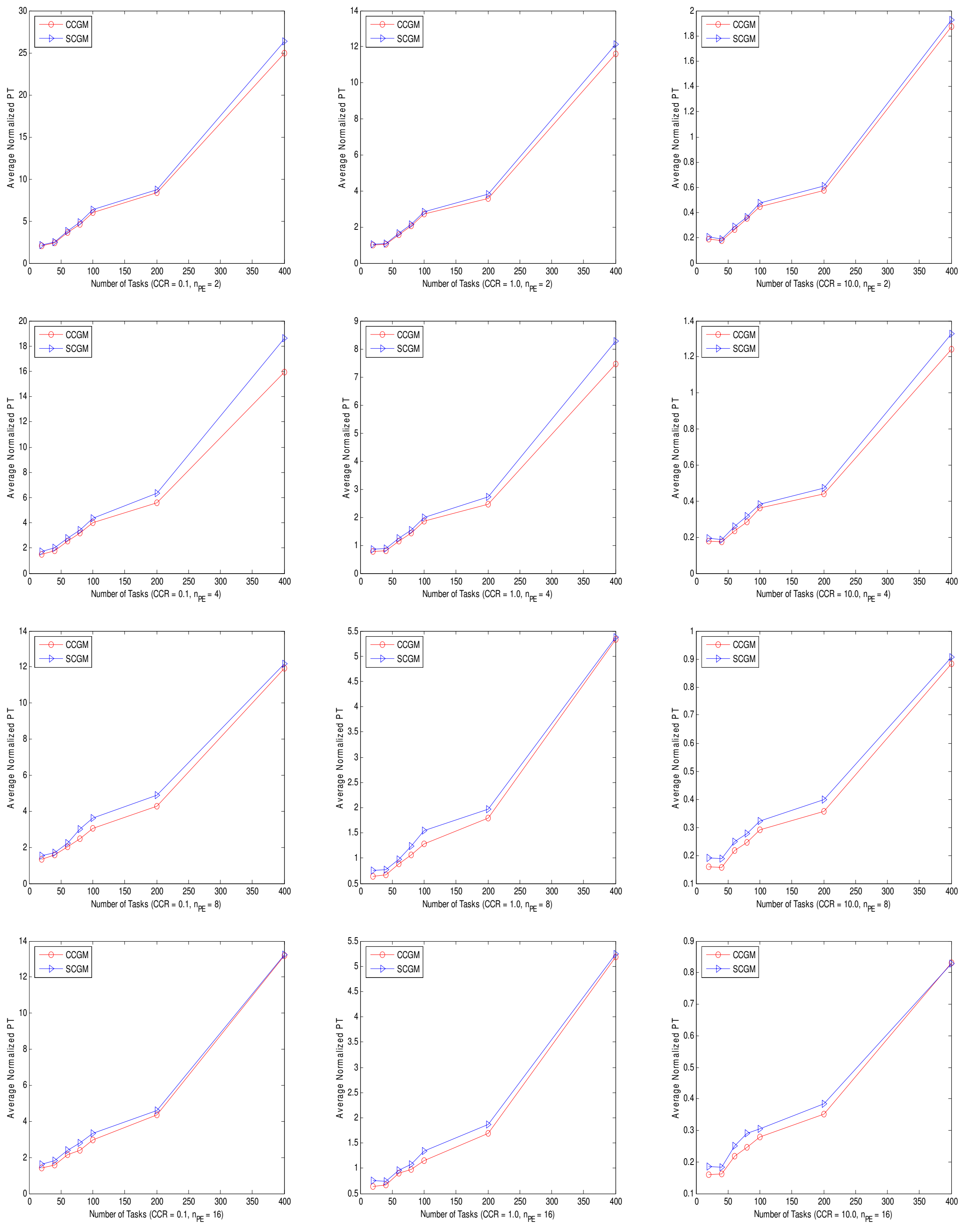


Figure 4.8: Performance comparison of two different clustering approach; separate clustering and GA merging vs. combined clustering and GA merging (i.e. CCGM vs. SCGM) on 2, 4 and 8 and 16 processors.

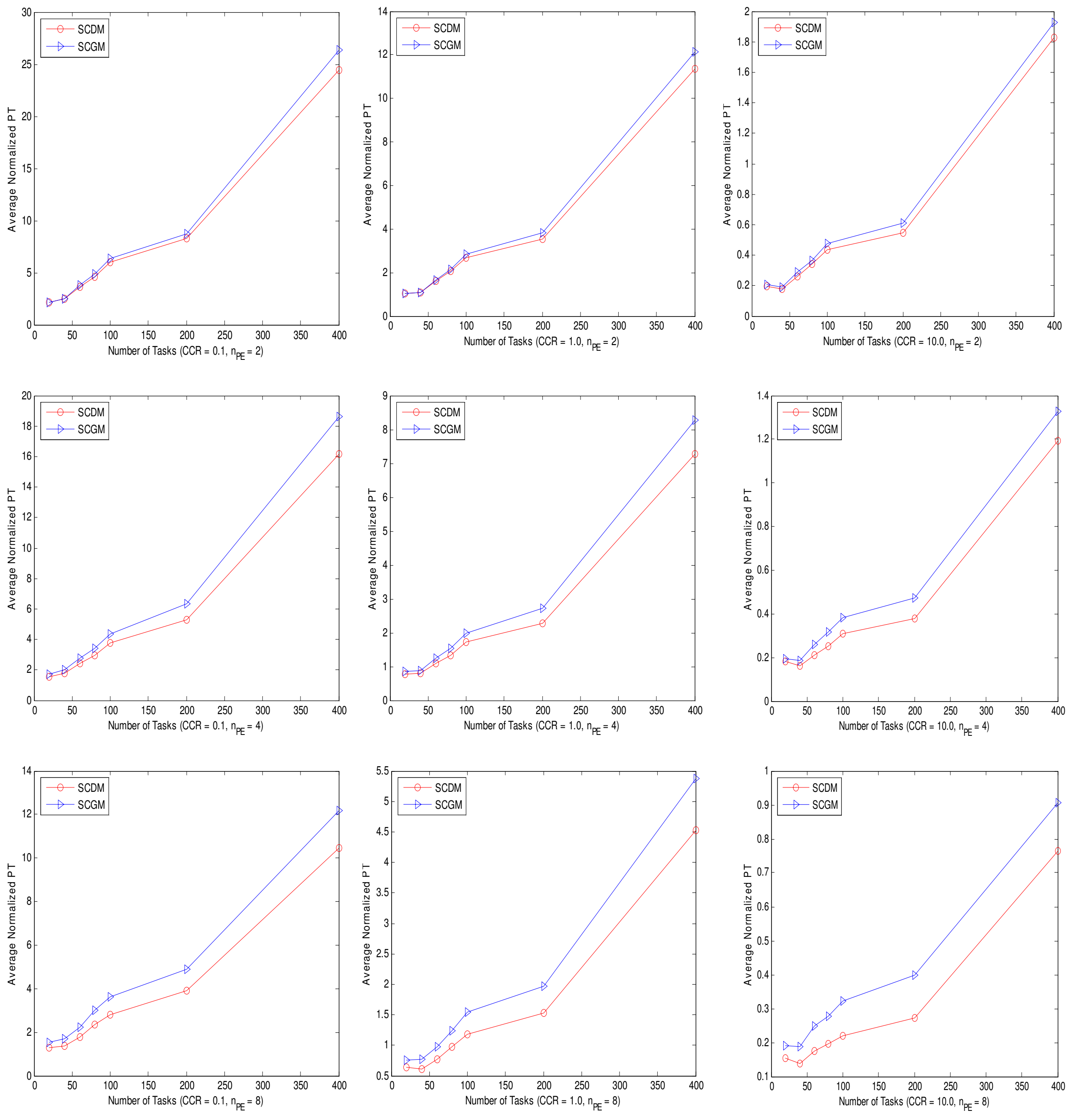


Figure 4.9: Performance comparison of the two GM algorithms (SCGM and CCGM) on 2, 4 and 8 processors.

Table 4.4: Performance Comparison of SCDM against SCGM algorithm

	SCGM(%)															
Algo.	$n_{PE} = 2$				$n_{PE} = 4$				$n_{PE} = 8$				$n_{PE} = 16$			
SCDM	$>$	$=$	$<$	% imp.	$>$	$=$	$<$	% imp.	$>$	$=$	$<$	% imp.	$>$	$=$	$<$	% imp.
0.1	70.5	13.1	16.4	6.6	95.5	2.0	2.5	13.7	95.1	4.5	0.4	20.7	95.5	1.6	2.9	24.5
1.0	64.3	17.2	18.4	6.7	91.8	5.7	2.5	14.1	95.5	2.0	2.5	21.3	94.7	0.8	4.5	22.0
10.0	59.0	30.3	10.7	11.5	74.6	16.4	9.0	17.7	84.8	11.1	4.1	25.4	86.5	9.4	4.1	25.3
Avg.	*64.6*	*20.2*	*15.2*	*8.3*	*87.3*	*8.0*	*4.7*	*15.2*	*91.8*	*5.9*	*2.3*	*22.4*	*92.2*	*3.9*	*3.9*	*23.9*

reveals that for a small percent of the time SCGM (i.e. 6.5%) outperforms the SCDM algorithm. Over all the given benchmarks and configurations SCDM outperforms SCGM over 83.9% by 17.44% on average. Our results show that in case of separate clustering algorithms a deterministic merging can provide significantly better results compared to a genetic algorithm based merging technique.

We also compared the two combined clustering (CC) algorithms against each other. A subset of results are given in Figure 4.10. Similar to the SC algorithms, in the combined clustering-based algorithms, the deterministic merging approach seems to be superior to the genetic algorithm based merging approach. A quantitative comparison of these two algorithms for a subset of benchmarks and configurations is also presented in Table 4.5. The comparison results over all benchmarks and configurations shows that the CCDM

Table 4.5: Performance Comparison of CCDM against CCGM algorithm

	CCGM(%)															
Algo.	$n_{PE} = 2$				$n_{PE} = 4$				$n_{PE} = 8$				$n_{PE} = 16$			
CCDM	$>$	$=$	$<$	% imp.	$>$	$=$	$<$	% imp.	$>$	$=$	$<$	% imp.	$>$	$=$	$<$	% imp.
0.1	60.2	26.9	13.0	3.2	96.7	2.5	0.8	9.3	95.4	1.3	3.3	15.1	95.8	0.0	4.2	20.5
1.0	59.1	30.4	10.4	4.5	92.1	4.8	3.1	10.4	94.1	2.9	2.9	16.7	90.7	1.9	7.4	19.8
10.0	64.3	34.0	1.6	9.0	74.0	23.3	2.7	16.2	82.8	15.9	1.3	22.7	83.3	14.4	2.3	25.8
Avg.	61.2	30.4	8.3	5.6	87.6	10.2	2.2	12.0	90.8	6.7	2.5	18.2	90.0	5.4	4.6	22.0

algorithm outperform CCGM on average by 14.4% and over 82.4% of the time. These result show that the DM merging seems to be a more efficient choice for cluster scheduling compared to a GA-based merging.

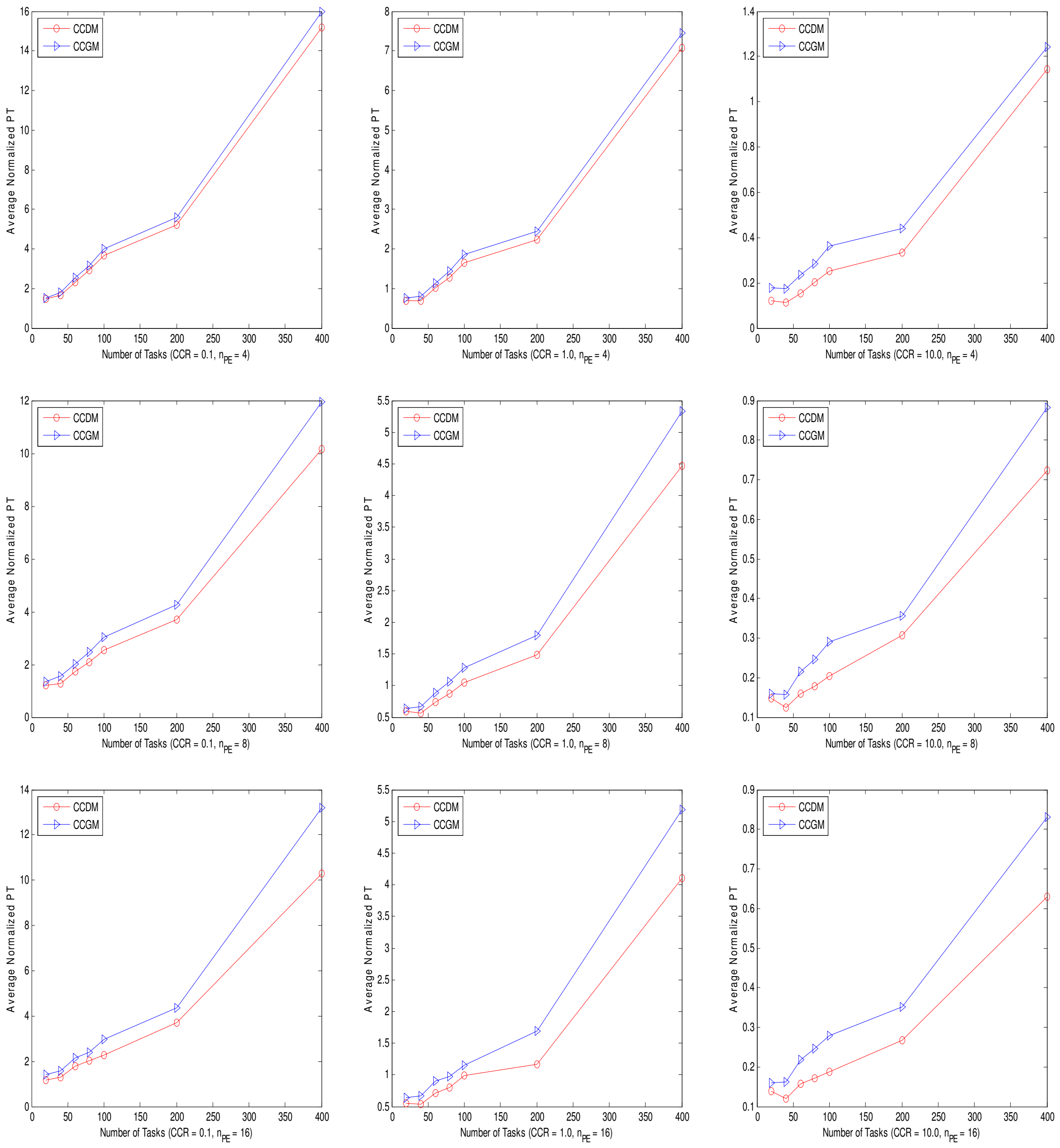


Figure 4.10: Performance comparison of two CC algorithms (CCDM and CCGM) on 4, 8 and 16 processors.

Before comparing our best techniques against the HEFT algorithm, we ran the randomized versions of HEFT to find out what the effect of randomization (section 4.4.1) on HEFT algorithm's performance is. We applied randomization to task and processor selection, once separately and once simultaneously by varying the randomization parameter p from 0 to 1.0 by a step-size of 0.1. The *task-only* randomization version of HEFT outperforms HEFT for $p \geq 0.7$ and the *processor-only* randomized version outperforms HEFT for $p = 0.8$. The performance improvements in both case are not very significant however. Additionally, the performance of the *task-only* randomized version is superior than the *processor-only* randomized version. The best results were obtained for the *combined tasks-processor*version where $p_{taskselection} = 0.9$ and $p_{processorselection} = 0.8$. These results are given in Table 4.6.

Table 4.6: Performance Comparison of Randomized HEFT algorithm

	HEFT(%)									
	$n_{PE} = 2$		$n_{PE} = 4$		$n_{PE} = 8$		$n_{PE} = 12$		$n_{PE} = 16$	
RHEFT	$<$	$\%imp.$	$<$	$\%imp.$	$<$	$\%imp.$	$<$	$\%imp.$	$<$	$\%imp.$
0.1	67.6	0.6	85.2	2.0	84.0	2.6	82.4	2.0	81.6	2.1
1.0	67.2	0.6	83.6	2.6	85.2	3.2	84.4	2.7	82.0	2.9
10.0	48.0	0.6	59.4	2.0	65.2	2.1	64.3	2.4	64.8	2.7
Avg.	60.9	0.6	76.1	2.2	78.1	2.6	77.0	2.4	76.1	2.6

On average the randomized HEFT algorithm outperforms its deterministic version by 2.1% more than 73.0% of the time. We have used the best results of randomized HEFT when comparing our algorithms against the HEFT algorithm in the following experiments.

Now to evaluate our techniques with other leading techniques, we took the best of the two CC and SC algorithms and compared then with the best of results of HEFT algorithm. The quantitative results are given in Tables 4.7 and 4.8.

The comparison results over all benchmarks and configurations shows that the

Table 4.7: Performance Comparison of CCDM against HEFT algorithm

Algo.	HEFT(%)															
	$n_{PE} = 2$				$n_{PE} = 4$				$n_{PE} = 8$				$n_{PE} = 16$			
CCDM	>	=	<	% imp.	>	=	<	% imp.	>	=	<	% imp.	>	=	<	% imp.
0.1	72.2	27.8	0.0	3.0	77.8	19.4	2.8	3.7	86.1	8.3	5.6	4.0	58.3	22.2	19.4	1.0
1.0	66.7	33.3	0.0	2.9	80.6	19.4	0.0	4.3	72.2	13.9	13.9	4.3	72.2	19.4	8.3	4.5
10.0	83.3	13.9	2.8	3.5	66.7	33.3	0.0	4	86.1	2.8	11.1	12.28	63.9	22.2	13.9	4.5
Avg.	74.1	25.0	0.9	3.1	75.0	24.1	0.9	4.0	81.5	8.3	10.2	6.9	64.8	21.3	13.9	3.3

CCDM algorithm outperform HEFT algorithm on average by 4.3% and over 73.0% of the time. A closer look at the results shows that on smaller size graphs (i.e. $|V| < 100$) the difference between CCDM is more than 12%. The reduced performance in case of larger size graphs is mainly due to the fact that we run both algorithms for a given time budget and CCDM is computationally more time consuming than HEFT algorithm and hence it is not able to perform its best for this given budget when it deals with larger size graphs. One would expect much superior performance from CCDM if time is not a tight constraint. We additionally observed that on average when the $CCR \geq 1.0$ the CCDM algorithm has its best performance, which shows the significance and importance of employing the pre-processing step of clustering in the presence of heavy communication costs.

We also compared HEFT against SCDM. The quantitative results are given in Table 4.8. Once more we observed that in the presence of heavy communication cost ($CCR \geq 1.0$) the SCDM algorithm has better performance than the HEFT algorithm.

Table 4.8: Performance Comparison of SCDM against HEFT algorithm

Algo.	HEFT(%)															
	$n_{PE} = 2$				$n_{PE} = 4$				$n_{PE} = 8$				$n_{PE} = 16$			
SCDM	>	=	<	% imp.	>	=	<	% imp.	>	=	<	% imp.	>	=	<	% imp.
0.1	50.0	33.3	16.7	1.4	44.4	25.0	30.6	1.0	55.6	5.6	38.9	0.4	25.0	8.3	66.7	-4.5
1.0	61.1	33.3	5.6	1.9	69.4	19.4	11.1	2.2	63.9	11.1	25.0	3.2	61.1	27.8	11.1	2.5
10.0	66.7	33.3	0.0	2.1	50.0	33.3	16.7	2.2	63.9	0.0	36.1	2.0	41.7	38.9	19.4	1.0
Avg.	59.3	33.3	7.4	1.8	54.6	25.9	19.4	1.8	61.1	5.6	33.3	1.9	42.6	25.0	32.4	-0.3

On average SCDM is better than HEFT by 1.3% more than 50.0% of the time.

4.7 Summary and Conclusions

In this chapter we presented the first comprehensive studies of clustering based scheduling algorithms for heterogeneous multiprocessor systems. We proposed two different algorithms for the scheduling problem; the classical approach where the clustering is evaluated first and then the merging is applied and our approach where the clusterings are evaluated w.r.t. merging results. We called this approached separate clustering (SC) and combined clustering (CC) techniques. Since clustering step is performed prior to the task/edge assignment the actual computation and communication values are not known when computing the performance of the clustering step. We used four different estimated values and experimentally showed that the average value is the best estimate when using clustering.

We also proposed two different merging algorithms; one deterministic heuristic and one genetic algorithm approach. Our experimental results showed that when the clustering is evaluated w.r.t. the final mapping (i.e. the combined clustering and merging approach) the overall performance is much higher. We also experimentally showed that when combined with clustering a deterministic cluster-scheduling or merging technique is more efficient than a GA-based merging. Ideally, a GA-based technique could provide a superior results given an unlimited time and resources, but for a given time budget (which is our case) an efficient deterministic technique outperforms the GA technique. Finally, we experimentally showed that our CCDM algorithm, the combined clustering and deterministic merging technique outperforms HEFT a leading heterogeneous multiprocessor scheduling technique. A general conclusion reached from the experimentation done with clustering techniques is that in the presence of heavy communication cost, a pre-processing step of clustering that internalizes heavy data communication by grouping the sender and receiver tasks together can significantly improve the overall performance of the final scheduling as well as providing a less complex search space.

Chapter 5

CHARMED: A Multi-objective Co-synthesis Framework for Multi-mode Embedded Systems

In this chapter, we present a modular co-synthesis framework called CHARMED (or Co-synthesis of HARdware-software Multi-mode EmbeddeD Systems) that provides a solution for the problem of hardware-software co-synthesis of periodic, multi-mode, distributed, embedded systems — current and emerging embedded systems often involve multiple application subsystems. Such applications may either run concurrently (single-mode) or in a mutually exclusive fashion, depending on operational modes (multi-mode). A high-frequency (HF) radio communications system is one example of such a multi-mode application. It provides a single integrated system solution to current and future HF voice and data communications requirements for military airborne operations. The integrated multi-mode system provides data communications capability over HF with modems, video imaging systems, secure voice devices, and data encryption devices, while continuing to provide voice HF communications capability. Mobile telephony, audio decoding systems and video encoding systems are other examples of multi-mode applications. A key characteristic of multi-mode systems is that the subsystems most often share sub-functions that are executed in each mode. For example consider a laptop PC that is used for watching a movie or transmitting video sequences. MPEG4 video decoder and encoder are being exercised by these video applications in mutually exclusive fashion re-

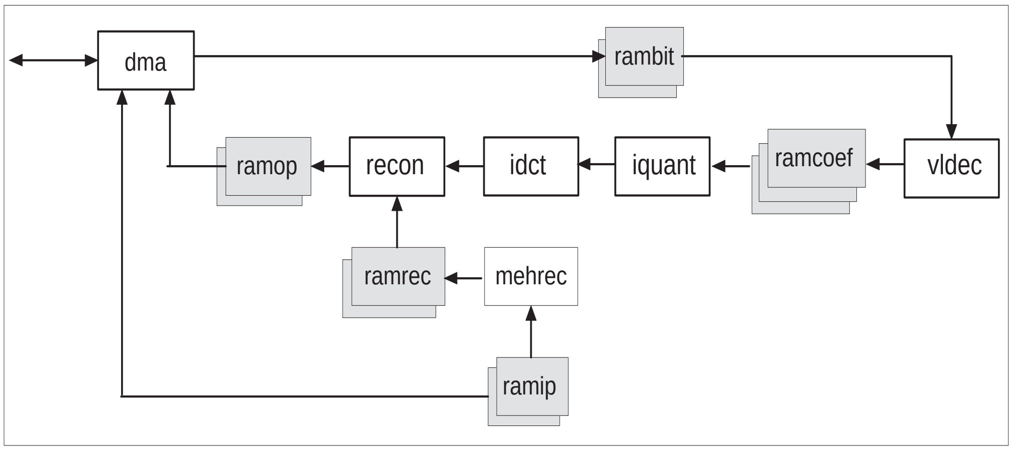


Figure 5.1: MPEG-4 Video Compression Decoder block diagram.

spectively. As it can be seen in Figures 5.1 and 5.2 the encoder and decoder cores share many sub-functions/units such as motion estimator (mefpel and mehpel), discrete cosine transform (fdct and idct) and quantization units (quant and iquant). Such sharing brings up the possibility of more optimal implementation of the overall system when implementation of different modes are optimized simultaneously due to inter-mode resource sharing. Concurrent mode optimization is what we are considering in our synthesis approach.

Additionally, in this framework we perform the synthesis under several constraints while optimizing for a set of objectives. We allow the designer to fully control the performance evaluation process, constraint parameters, and optimization goals. Once the synthesis is performed, we provide the designer a non-dominated set (Pareto front) of implementations on streamlined architectures that are in general heterogeneous and distributed.

We also employ the pre-processing step of clustering when appropriate to provide a more optimized and compact representation of the system and to reduce the complexity of the solution space and expedite the search. The experimental results demonstrate the

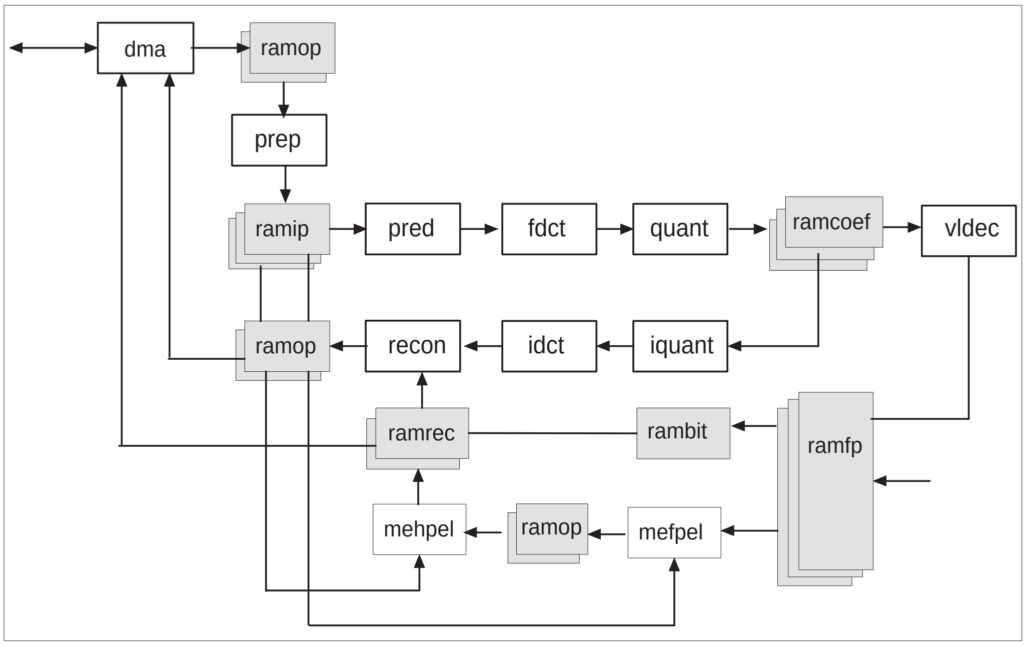


Figure 5.2: MPEG-4 Video Compression Encoder block diagram.

effectiveness of the CHARMED framework in computing efficient co-synthesis solutions within a reasonable amount of time.

Our contribution in this work is as follows: We propose a modular co-synthesis framework that is the first comprehensive algorithms that synthesizes multi-mode, multi-task embedded systems under a number of hard constraints; optimizes a comprehensive set of objectives; and provides a set of alternative trade-off points, generally known as Pareto-optimal solutions. Our framework allows the designer to independently configure each dimension of the design evaluation space as an optimization objective (to be minimized or maximized) or as a constraint (to be satisfied). Furthermore, we employ a hierarchical evolutionary algorithm architecture that utilizes a pre-processing step of clustering (based on the powerful CFA technique [71]) to optimize the application to be synthesized and a parallelization technique to expedite the co-synthesis process.

The remainder of this chapter is organized as follows: In the next Section(5.1) we present a survey of the literature . In Section 5.2 we state the problem we are addressing in the chapter formally. In Section 5.3 we briefly describe the employed multi-objective optimization algorithm and the associated modifications we have made in this work. In Section 5.4 we describe the implementation details of our first co-synthesis algorithm. This implementation does not consider FPGAs as an available resource (PE). In Section 5.5 we describe the implementation details of our second co-synthesis algorithm that provides a mean for better memory and power management and also includes FPGAs as target PEs. In Section 5.6 we introduce the parallel version of our synthesis algorithm. We give the experimental results in Section 5.7 and conclude the chapter with Section 5.8.

5.1 Related Work

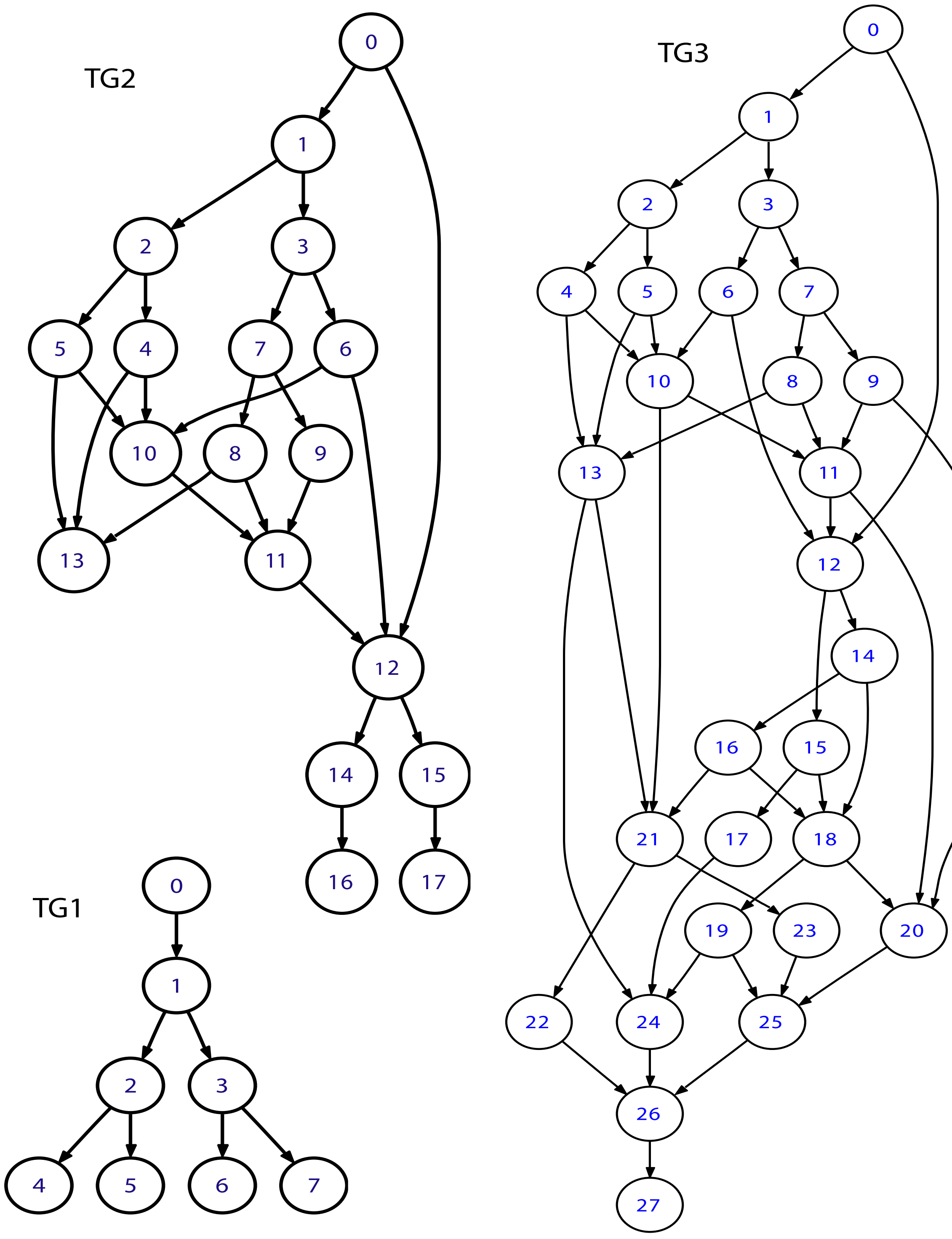
Embedded systems have a variety of constraints and optimization goals such as memory, performance, price, area, power, runtime, number of physical links, etc. to be accommodated. To satisfy such pressing design demands, researchers have shifted from optimal approaches such as MILP that could handle only a subset of such requirements for small task graphs [111] to deterministic heuristic [108][24][65][54] and probabilistic search heuristic [121][33][135] approaches. Many of these approaches only focus on single-mode systems or regard multi-mode systems as multiple single-mode systems and optimize them separately ([24] [107] [33] [135]). However, if a task is commonly used across different modes then these algorithms may not be able to find the most efficient solutions. Additionally, sharing of hardware resources among tasks that are not active simultaneously can greatly reduce system cost. [121], [108] and [65] consider multi-mode applications while optimizing only for a small set of costs concerning embedded systems. All of the deterministic heuristic methods, convert the multi-dimensional optimization problem to a single-dimensional optimization problem by forming a linear combination of the objectives (e.g. power, cost). The main disadvantage of this technique is that it cannot generate all Pareto-optimal solutions with non-convex trade-off surfaces, which typically underlie hardware-software co-synthesis scenarios. Furthermore, forming these linear combinations can be awkward because they involve computing weighted sums of values associated with heterogeneous metrics. There is no clear methodology for formulating the linear combinations, and their physical interpretation is ambiguous. [121], [33] and [135] employ evolutionary algorithms to overcome the two drawbacks mentioned above and target significantly larger problem instances. However, they optimize for a significantly smaller set of system costs compared to what we consider in this chapter.

Additionally most HW-SW co-synthesis algorithms do not tackle FPGAs [111][34] [152] and there are only a small number of co-synthesis algorithms than can handle dynamically reconfigurable hardware (i.e. FPGAs). The use of dynamically reconfigurable hardware adds another dimension to the problem complexity, since the ordering in which tasks are scheduled on the reconfigurable hardware directly effects the amount of reconfiguration data required and hence the performance of the overall system. One of the first studies targeting FPGAs is the CORDS work [35]. In CORDS co-synthesis system, the ordering and scheduling is performed using a greedy deterministic approach that eliminates the reconfiguration delay by scheduling same-type tasks consecutively. In this algorithm, multiple tasks are not allowed to execute concurrently on the same FPGA.

Subsequent works targeting reconfigurable hardware are [25][61][105][123]. [61] further reduces the reconfiguration time overhead by performing incremental reconfiguration of tasks which partially share configuration data with tasks that have already been reconfigured. It (as well as [105]) however make the simplifying assumptions of considering only one processor and one FPGA. In [123] a deterministic algorithm is employed to tackle the FPGA-related allocation and scheduling. In our implementation we handle these tasks by means of evolutionary algorithm techniques and operators.

5.2 Problem statement

The co-synthesis problem considered in this chapter is defined as the problem of optimally mapping the task-level specification of the embedded system onto a heterogeneous multiprocessor architecture. The embedded system applications are represented in terms of the task graph model described in 2.3. Each system is characterized by multiple modes of functionality, where each mode can comprise of several task graphs. An example of a three-mode three-task graph embedded system is given in Figure 5.3. The optimization goal is to find a set of implementations that simultaneously minimize multiple objectives for which the corresponding objective vectors cannot be improved in any dimensions without degradation in another. An implementation is described by selection of a set of processing elements (PE) and communication resources (CR) (allocation), mapping of the application onto the selected architecture (assignment) and scheduling each task and data communication on the system resources. Each implementation, represented by solution vector $\vec{x}$, is evaluated with respect to a set of objectives that are



Modes	Mode 1		Mode 2		Mode 3		
Task Graph	TG2	TG3	TG1	TG2	TG1	TG2	TG3
Period	900	1400	300	450	300	900	700

Figure 5.3: A 3-mode 3-task graph embedded system.

as follows: area ($\alpha(\vec{x})$), price ($\kappa(\vec{x})$), number of links ($\ell_n(\vec{x})$), memory requirement of each PE ($\overrightarrow{\mu}(\vec{x})$), power consumption or (energy consumption) ($p(\vec{x})$) and parallel-time or completion-time ($\tau_{par}(\vec{x})$).

Initially all of these goals are defined explicitly as separate optimization criteria, however, our framework allows the designer to formulate any of these goals as a constraint, e.g., that the size of the system must not exceed given dimensions. The algorithm always takes the upper bound (not to be violated if the optimization goal is formulated as a constraint) for each optimization goal as an input. However, the input vector $\Omega_0 = [\alpha_0, \kappa_0, \ell_{n_0}, \overrightarrow{\mu}_0, p_0, \tau_{par_0}]$ will determine the optimization/constraint setting. An entry of 0 means strictly optimization, an entry of 1 means formulate as a constraint and an entry of 2 means optimize while satisfying the constraint. An entry of -1 means to discard that goal for the current problem instance. For example $\Omega_0 = [\alpha_0, \kappa_0, \ell_{n_0}, \overrightarrow{\mu}_0, p_0, \tau_{par_0}] = [0, 0, 1, -1, 0, 2]$ configures CHARMED to find an implementation such that it minimizes the area, dollar cost, power consumption and parallel-time; meets the deadline; and does not exceed the given number of inter-processor links. An example of the set of solutions found by CHARMED for the embedded system given in Figure 5.3 is given in Tables 5.1 and 5.2. Table 5.1 presents the overall system costs. Table 5.2 presents the per-mode system costs. The corresponding configuration is given as in $\Omega_0 = [0, -1, -1, 0, 0, 0]$.

Table 5.1: Overall system costs found by CHARMED for the system given in Figure 5.3

Costs	Area	Memory
sol1	7599.0	32814
sol2	8514.0	31240
sol3	6845.0	33669

Table 5.2: System costs for individual modes found by CHARMED for the system given in Figure 5.3

System Costs		Area	Memory Size	Energy Consumption	Parallel Time
sol1	m1	7215.0	31209	3214310.6	2769.2
	m2	2783.0	22747	5547252.5	2189.6
	m3	7581.0	32660	8986228.9	3403.2
sol2	m1	7215.0	31083	3209843.6	2745.9
	m2	2783.0	22747	5542785.4	2189.6
	m3	8514.0	30209	8765174.3	3307.5
sol3	m1	6477.0	32704	3311090.5	2928.6
	m2	2783.0	22747	5644032.4	2189.6
	m3	6827.0	33404	9144038.7	3657.3

5.3 Evolutionary Multi-objective Optimization

The complex, combinatorial nature of the co-synthesis problem and the need for simultaneous optimization of several incommensurable and often competing objectives has led many researchers to experiment with evolutionary algorithms (EAs) as a solution method. EAs seem to be especially suited to multi-objective optimization as due to their inherent parallelism, they have the potential to capture multiple Pareto-optimal solutions in a single simulation run and may exploit similarities of solutions by recombination. Hence, we have adapted the Strength Pareto Evolutionary Algorithm (SPEA), an evolutionary algorithm for multi-objective optimization shown to have superiority over other existing multi-objective EAs [154]. SPEA algorithm details are provided in Section 2.2.3. One issue about the SPEA technique is that it does not handle constraints and only concentrates on unconstrained optimization problems. Hence, we have modified this algorithm to solve constrained optimization problems by employing the constraint-dominance relation (in place of dominance relation) defined as follows [27]:

Definition 2: *Given two solutions a and b and a minimization problem, a is said to* constrained-dominate *b if*

1. Solution a is feasible and solution b is not, or

2. Solutions a and b are both infeasible, but solution a has a smaller overall constraint violation, or

3. Solutions a and b are feasible and solution a dominates solution b.

More implementation details on the employed multi-objective EA are given in the next section.

5.4 CHARMED: Our Proposed Algorithm

CHARMED is a multi-objective evolutionary algorithm based on Strength Pareto Evolutionary Algorithm [154](see Section 2.2.3). It is constituted of two main components of task clustering and task mapping. A high-level overview of CHARMED is depicted in Figure 5.4. CHARMED starts by taking input parameters that consist of: a system specification in terms of task graphs, PE and CR libraries, and an optimization configuration vector Ω_0. These inputs are then parsed and appropriate data structures such as attribute vectors and matrices are created. Next, the solution pool of the EA-based clustering algorithm (called multi-mode clusterization algorithm or MCFA) is initialized and task clustering is formed based on each solution. Solutions (clusters) are then evaluated using the coreEA, i.e. for each clustering, coreEA is initialized by a solution pool representing different mappings of the clustering onto different distributed heterogeneous systems. These mappings are evaluated for different system costs such as area, price, power consumption, etc. coreEA fine-tunes the solutions iteratively for a given

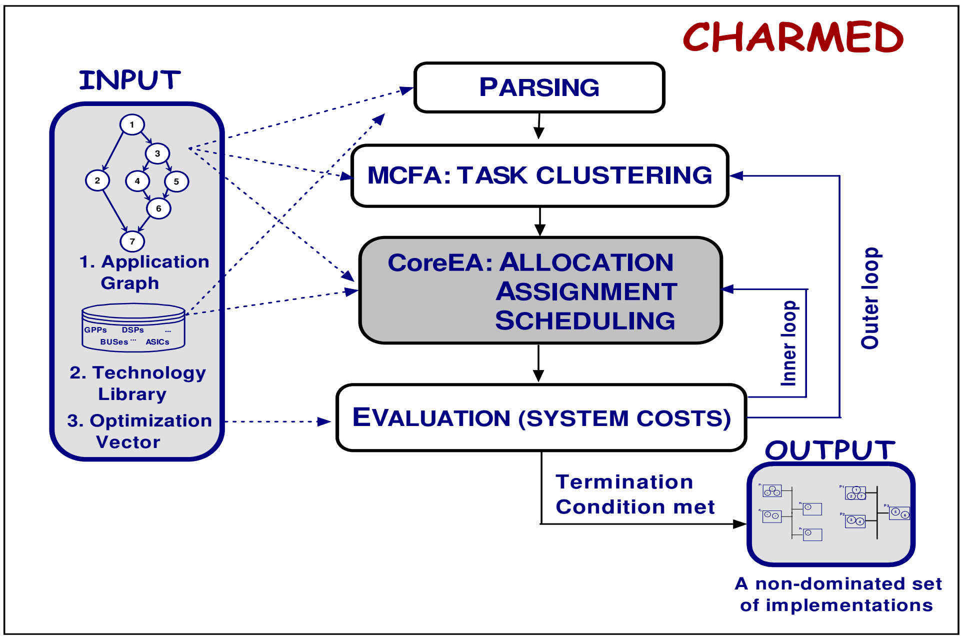


Figure 5.4: CHARMED framework.

number of generations and then returns the fittest solutions and fitness values. Once all the fitness values for all the clusters are determined, the clustering EA (MCFA) proceeds with the evolutionary process and updating the clustering population until the termination condition is met. The outline of this algorithm is presented in Figure 5.5.

In Step 3 of CHARMED the coreEA, another SPEA based evolutionary algorithm is invoked for each individual (i.e. clustering) in the MCFA population. coreEA finds a set of non-dominated solutions (of size XN_{II}) for each cluster which means that N_I clusters will have a total of $XN_{II} \times N_I$ solutions. These solutions are stored in a temporary population $P_{Itemp}(t)$ and we use this temporary population (as well as $XP_I(t)$) to form $XP_I(t+1)$ in Step 4. More details on MCFA nd coreEA algorithms are given in the

PARAMETERS: N_I (MCFA population size), N_{II} (coreEA population size)
XN_I (MCFA archive size), XN_{II} (coreEA archive size)
INPUT: A set of task graphs $G_{m,i}(V, E)$, processing elements, communication resources library and an initial optimization vector Ω_0.
OUTPUT: A non-dominated set (Å) of architectures which are in general heterogeneous and (distributed) together with task mappings onto these architectures.

Step 1 **Initialization (MCFA):** Generate an initial population $P_I(t)$ of binary string of size $\sum_{m=0}^{M-1} \sum_{i=0}^{|G_m(V,E)|-1} |E_{m,i}|$. Randomly initialize with 0 and 1s. Create the empty archive (external set) $XP_I(t) = \emptyset$ and $t = 0$.
Step 2 **Task Clustering:** Decode each binary string and form the associated clusters.
Step 3 **Fitness Assignment (coreEA):** Perform mapping and scheduling for each individual (representing a set of clusters). Compute different system costs as indicated by Ω_0 for each individual. Calculate fitness values of individuals in $P_I(t)$ and $XP_I(t)$.
Step 4 **Environmental Selection:** Copy all non-dominated individuals in ${P_I}^1(t)$ and $XP_I(t)$ to $XP_I(t+1)$. If $|XP_I(t+1)| \neq XN_I$ adjust $XP_I(t+1)$ accordingly.
Step 5 **Termination:** If $t > T$ or other stopping criterion is met then set $A = XP_I(t+1)$ and stop.
Step 6 **Mating Selection:** Perform binary tournament selection on $XP_I(t+1)$ to fill the mating pool.
Step 7 **Variation:** Apply crossover and mutation operators to the mating pool and set $P_I(t+1)$ to the resulting population. Increment generation counter $t = t + 1$ and go to Step 2.

Figure 5.5: Flow of CHARMED

following sections.

5.4.1 MCFA: Multi-Mode Clusterization Function Algorithm

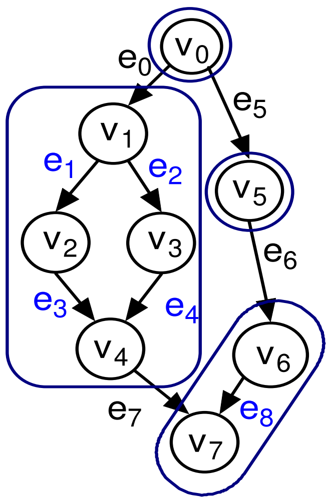
As we previously pointed out in Chapter 3, *Clustering* is often used as a front-end to multiprocessor system synthesis tools [24][54]. In this context, clustering refers to the grouping of tasks into subsets that execute on the same PE. The purpose of clustering is thus to reduce the complexity of the search space and constrain the remaining steps of synthesis, especially assignment and scheduling. The clustering algorithms employed in earlier co-synthesis research have been designed to form task clusters only to favor one of the optimization goals, e.g. to cluster tasks along the critical path or higher energy-level

[1]This is a temporary population that has its member repeated XN_{II} as explained in the text.

path. Such algorithms are relatively simple and fast but suffer from a serious drawback, namely that globally optimal or near-optimal clusterings with respect to all system costs may not be generated. Hence in this work we adapt the clusterization function algorithm (CFA), which we introduced in Chapter 3. The effectiveness of CFA has been demonstrated for the minimum parallel-time scheduling problem, that is, the problem of scheduling a task graph to minimize parallel-time for a given set of allocated processors. However, the solution representation in CFA is not specific to parallel-time minimization, and is designed rather to concisely capture the complete design space of possible graph clusterings. Therefore, it is promising to apply this representation in other synthesis problems that can benefit from efficient clustering. One contribution of this work is to apply CFA in the broader contexts of multi-mode task graphs, co-synthesis, and multi-objective optimization. In doing so, we demonstrate much more fully the power of the clustering representation that underlies CFA. Our multi-mode extension of CFA is called MCFA and its implementation details are as follows:

Solution Representation: Our representation of clustering exploits the view of a clustering as a subset of edges in the task graph. The coding of clusters for a single task graph in MCFA is composed of a n-size binary string, where $n = |E|$ and E is the set of all edges in the graph. There is a one to one relation between the graph edges and the bits, where each bit represents the presence or absence of the edge in a cluster. The details of this encoding and decoding procedure for a simple task graph are given in Figure 5.6.

Assuming M modes and $|G_m(V, E)|$ task graphs for each mode, the total size of the binary string that would capture the clustering for all task graphs across different modes is $n_{all} = \sum_{m=0}^{M-1} \sum_{i=0}^{|G_m(V,E)|-1} |E_{m,i}|$.



1	0	0	0	0	1	1	1	0
e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8

```
Function FormCluster(Input: A binary String, Output: Cluster of Tasks)
 Mark all tasks as UNCLUSTERED
 cluster_count = 0;
 FOR  i ← 0  to  |E|
    IF ($bit_i$ == 0)
       IF (head($e_i$) and tail($e_i$) are UNCLUSTERD)
          CLUST(cluster_count) ∪= {head($e_i$), tail($e_i$)}
          cluster_count++;
          Mark head($e_i$) & tail($e_i$)  as CLUSTERED
       ELSEIF (head($e_i$) is UNCLUSTERED &  tail($e_i$) is CLUSTERED)
          CLUST($CLUST^{-1}$(tail($e_i$)) ∪= {head($e_i$)}
       ELSEIF (head($e_i$) is CLUSTERED & tail($e_i$) is UNCLUSTERED)
          CLUST($CLUST^{-1}$(head($e_i$)) ∪= {tail($e_i$)}
       ELSEIF ($CLUST^{-1}$(head($e_i$)) ≠ $CLUST^{-1}$(tail($e_i$))
          Merge two CLUST sets
          Update other CLUST sets
          cluster_count--;
       ENDIF
    ENDIF
 ENDFOR
```

Figure 5.6: An illustration of binary string representation of clustering in MCFA and the associated procedure for forming the clusters from the binary string.

Initial Population: The initial population of MCFA consists of N_I (to be set experimentally) binary strings that represent different clusterings. Each binary array is initialized randomly with equal probability for a bit of 1 or 0. The external population size (where the non-dominated solutions are stored) is XN_I.

Genetic Operators: We will discuss the crossover and mutation operators in Section 5.4.3. For the selection operator we use binary tournament with replacement [8]. Here, two individuals are selected randomly, and the best of the two individuals (according to their fitness values) is the winner and is used for reproduction. Both winner and loser are returned to the pool for the next selection operation of that generation.

Fitness Evaluation: Clusterings are evaluated using coreEA, which is described in detail in the next section(5.4.2).

The key characteristic of MCFA (or CFA) is the natural, binary representation for clusterings that, unlike previous approaches to clustering in co-synthesis, is not specialized for one specific optimization objective (e.g., critical path minimization), but rather, can be configured for different, possibly multi-dimensional, co-synthesis contexts based on how fitness evaluation is performed.

5.4.2 coreEA: mapping and scheduling

coreEA is the heart of the CHARMED framework and its goal is to find a set of implementations for each member of the MCFA solution pool (or each clustering). It runs once for each member. coreEA starts by creating a PE and a CR allocation string for the given solution (or clustering). The lengths of these string are equal to the number of PE and CR types, respectively. We initialize them such that every cluster and every inter-cluster communication (ICC) edge has at least one instance of PE or CR that it can execute on. Each entry of the string represents the number of available instances of the associated PE type. Based on these allocation strings and the numbers of clusters and ICC edges we then initialize the population of coreEA. Further design details of this EA are as follows:

Solution Representation: Solutions in coreEA represent the assignment of clusters to PEs and ICCs to CRs. These assignments are encoded in two different binary matrices, hence each solution is represented with a pair of matrices. Using the allocation arrays we compute the total number of available PEs ($|PE_{avail}|$) and CRs ($|CR_{avail}|$)(including different instances of a same type). The assignment matrix for clusters is of size $|PE_{avail}| \times |clusters|$ and for ICCs is of size $|CR_{avail}| \times |ICC|$. $|clusters|$ denotes the number of clusters in the solution and $|ICC|$ denotes number of ICC edges. Each column of the cluster (ICC) assignment matrix corresponds to a cluster (ICC) that has to be assigned to a PE (CR). Each row of this matrix corresponds to an available PE (CR). Each column of the PE (CR) assignment matrix possesses exactly one non-zero row that determines the PE (CR) that the cluster (ICC) is assigned to.

Initial Population: The initial population of coreEA consists of N_{II} (to be set experimentally) pair of assignment matrices (one for clusters and one for ICCs). For each solution representing the PE (CR) assignment, exactly one 1 is randomly assigned to each column of each assignment matrix. The external population size (where the non-dominated solutions are stored) is XN_II.

Genetic Operators: The crossover and mutation operators of coreEA are discussed in Section 5.4.3. For the selection operator, we use a technique similar to the one described in Section 5.4.1.

Fitness Evaluation: Each member of the solution pool of coreEA that is a pair of assignment matrices representing cluster-to-PE and ICC-to-CR mapping, is employed to construct a schedule for each clustering. Once the ordering and assignment of each task is known we calculate other objectives and constraints given in Ω_0. Since the modes are mutually exclusive, it is possible to employ scheduling methods that are used for single mode systems. Scheduling a task graph for a given allocation and for a single mode is a well-known problem which has been extensively studied and for which good heuristics are available. Hence, we employ a deterministic method that is based on the classic list scheduling heuristic to find the ordering of tasks on each PE and the associated schedule.

Once clusters are mapped and scheduled to the target architecture, we compute different system costs across different modes and check for constraint violations of individual modes. Next, using the constrained-dominance relation we calculate the fitness value for each individual [154]. In certain problems, the non-dominated set can be extremely large and maintaining the whole set when its size exceeds reasonable bounds is not advantageous. Too many non-dominated individuals might also reduce selection pressure and slow down the search. Thus, pruning the external set while maintaining its characteristics, before proceeding to the next generation is necessary [154]. The pruning process is based on computing the phenotypic distance of the objective values. Since the magnitude of each objective criterion is quite different, we normalize the distance with respect to each objective function. More formally, for a given objective value $f_1(\vec{x})$, the distance between two solutions $\vec{x_i}$ and $\vec{x_j}$ with respect to f_1 is normalized as follows:

$$\frac{(f_1(\vec{x_i}) - f_1(\vec{x_j}))^2}{(\max(f_1(t)) - \min(f_1(t)))^2}. \tag{5.1}$$

For area, price, power consumption and parallel-time, the $\max(f_1(t))$ and $\min(f_1(t))$ denote the worst and best case values of the corresponding system cost among all the members in generation t, respectively. The maximum number of links or $\max(\ell_n(t))$ is computed from the maximum possible number of physical inter-processor links for the given graph set $G_{m,i}(V, E)$ and the processor configuration, i.e.

$$\max(\ell_n(t)) = \max(|PE_{used}| \times (|PE_{used}| - 1), |E_{m,i}|), \tag{5.2}$$

where $|E_{m,i}|$ is the number of edges in the graph. The equation implies that the maximum number of inter-processor links that make sense for a network is not necessarily that corresponding to a fully connected-network; depending on the number of edges in the graph, this number can be smaller. Minimum number of links is equal to $|PE_{used}| - 1)$. The maximum value for the memory requirement is equal to the size of data and instruction memory. The minimum value is computed using the smallest amount of memory used among the solutions in generation t. If optimization goals are formulated as constraints, the maximum values are replaced by the given constraint values.

The flow of coreEA is given in Figure 5.7.

coreEA can be employed without the pre-processing step of clustering by simply substituting groups of tasks (clusters) by individual tasks.

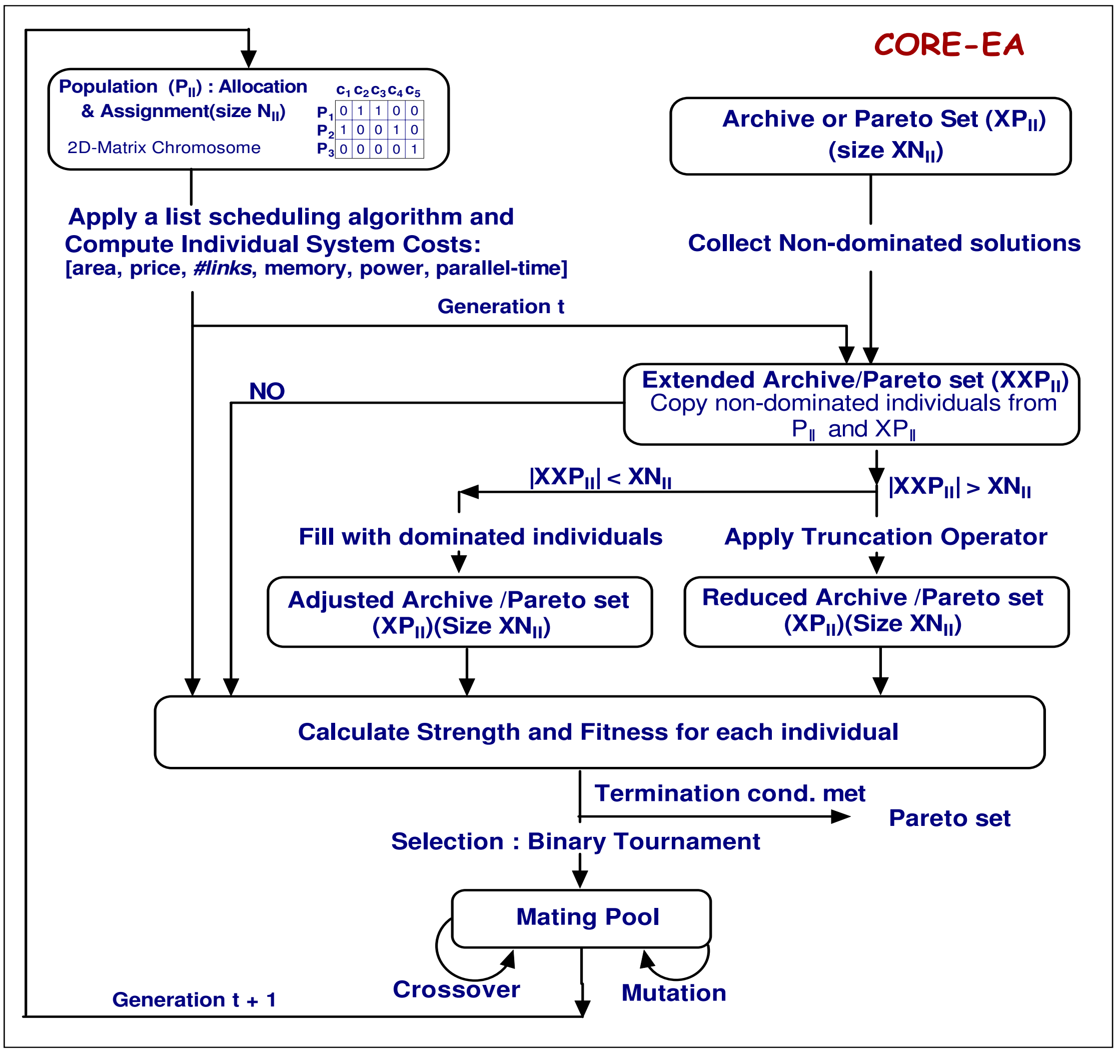

CORE-EA
Population (P_II) : Allocation & Assignment(size N_II)
2D-Matrix Chromosome
c1 c2 c3 c4 c5
P1 0 1 1 0 0
P2 1 0 0 1 0
P3 0 0 0 0 1
Archive or Pareto Set (XP_II) (size XN_II)
Apply a list scheduling algorithm and Compute Individual System Costs:
[area, price, #links, memory, power, parallel-time]
Collect Non-dominated solutions
Generation t
NO
Extended Archive/Pareto set (XXP_II) Copy non-dominated individuals from P_II and XP_II
|XXP_II| < XN_II
|XXP_II| > XN_II
Fill with dominated individuals
Apply Truncation Operator
Adjusted Archive /Pareto set (XP_II)(Size XN_II)
Reduced Archive /Pareto set (XP_II)(Size XN_II)
Calculate Strength and Fitness for each individual
Termination cond. met
Pareto set
Selection : Binary Tournament
Mating Pool
Crossover
Mutation
Generation t + 1

Figure 5.7: Flow of coreEA Algorithm.

5.4.3 Multi-mode genetic operators

The design of genetic operators for manipulating multi-mode systems requires special considerations that take into account both the global aspects of the systems, while respecting the local properties associated with individual modes. This point is elaborated as follows:

- For a given single-mode application, all system costs (area, power, price, parallel time, etc.) are the direct result of the final assignment and scheduling of task graphs of that mode. However, for multi-mode applications, some system costs such as area and price are highly dependent on the influence of individual modes and are a combination of the areas and prices of individual modes considered in isolation. Hence to minimize these costs, each individual mode should be minimized as well. However, there are other system costs such as parallel-time and power that mostly depend only on individual modes. For example, if a set of clusterings of mode i does not meet the required deadline it can have several reasons as follows : i) improper clustering of task graphs of that mode, ii) inefficient cluster (ICCs) to PE (CR) assignment or iii) inefficient selection of PEs and CRs. The first two problems can only be fixed by intra-mode changes i.e. exchanging tasks among clusters or changing the cluster assignments (these can be achieved by applying evolutionary operators i.e. mutation and crossover to the part of the solution string representing this mode). The last problem can be fixed by changing the type or number of given PEs or CRs among different modes i.e. applying evolutionary operators across modes. On the other hand, for the same system, if the price is not minimized effectively, it is largely due to inefficient assignments across

all modes and improvement can only be made by swapping bits across the modes.

- If a candidate solution has a high fitness, then it is reasonable to assume that it is made up of smaller parts that in turn conferred a certain degree of fitness on the overall solution. Therefore, by selecting highly fit solutions for the purpose of crossover to create the next generation, we are giving more chance to those solutions that contain good *building blocks*.

Motivated by the above observations, we apply evolutionary operators once to each mode (intra-mode) to preserve good local building blocks of that mode and once across modes (inter-mode) to preserve global building blocks.

The genetic operators for reproduction (mutation and crossover) that we use are the traditional two-point crossover and the typical mutator for a binary string chromosome where we flip the bits in the string with a given probability. We apply both operators once locally within the modes and again globally across different modes according to some probability i.e. we do not always apply both operators at the same time. So at the start of the crossover or mutation process based on a probability value we decide whether to apply the evolutionary operators only intra-mode, inter-mode or both intra- and inter-mode. Some techniques apply these operators once and only across all modes [121], which to our opinion may not fully take advantage of the evolutionary operator power within modes. An example of applying the crossover operator to a binary string representing the clustering encoding of MCFA is given in Figure 5.8.

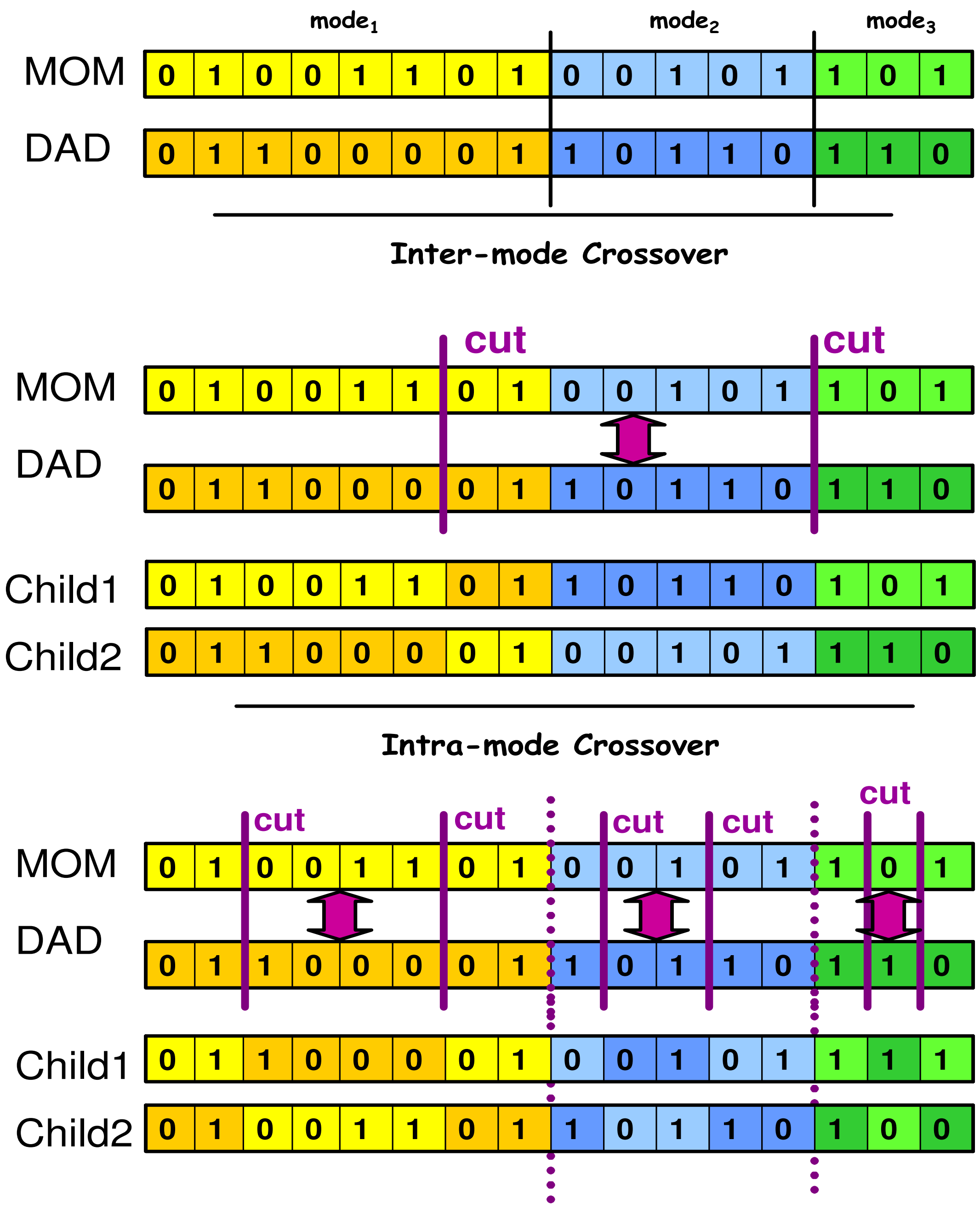


Figure 5.8: An example of inter-mode and intra-mode crossover for MCFA algorithm.

5.5 CHARMED-plus: Our Proposed Algorithm

In this section we describe the CHARMED-plus that is the CHARMED framework extended to handle synthesis of systems containing dynamically reconfigurable hardware. The motivation for introducing CHARMED-plus is the importance of task ordering on the PEs and its effect on i) system costs such as energy consumption, parallel-time and memory management; and ii) scheduling tasks on reconfigurable hardware. The two issues are further explained below:

- System Costs (Parallel-Time, Energy Consumption, Memory Requirement): One key difference among many scheduling algorithms specifically list scheduling algorithms is the task prioritization policy that decides the ordering in which tasks are scheduled on their designated processors. In a general scheduling problem when the only optimization criterion is to meet the dealing or increase the performance the it has been shown that the choice of blevel 3.1 priority metric is very effective. However when there are multiple optimization criteria, an ordering that minimizes the parallel time may lead to an increased energy consumption or memory requirement. Area, price and number of links are not effected by the ordering decisions.

- Reconfigurable Hardware: FPGAs can execute multiple tasks simultaneously and can be reconfigured dynamically to execute a set of new tasks. The reconfiguration process adds delays to the system and increases the power consumption. To reduce the effect of the reconfiguration overhead, one should try to order tasks such that the reconfiguration required for each FPGA is minimized.

Considering the two design issues given above, a scheduling policy should select tasks and order them such that the memory requirement, energy consumption, reconfiguration overhead and parallel time are minimized. Designing a scheduling algorithm capable of addressing all the above criteria is a new multi-objective optimization problem of its own. Like any other multi-objective problem, EA seem to be the best choice for addressing this problem. However, instead of designing a new EA-based scheduling technique we modify CHARMED to handle the ordering of tasks as part of the evolutionary process as well. The modifications are only applied to the coreEA algorithm and are given as follows:

Solution Representation: In this modified version of coreEA or coreEA-plus, each solution in addition to representing the assignment matrices includes M sorted strings representing the execution order of the tasks in the corresponding schedule for each mode. The size of each string is equal to the total number of tasks executing in that mode (from multiple task graphs constituting that mode). Each string is ordered in ascending order of the task heights, which guarantees that the precedence constraints are satisfied. The height $height(v_j)$ of a task v_j is a random integer whose value is such that:

$$\forall v_i \in R_{v_j} \wedge \forall v_k \in U_{v_j} : max(h_{init}(v_i)) + 1 \leq height(v_j) \leq min(h_{init}(v_k)) - 1 \quad (5.3)$$

where R_{v_j} (U_{v_j}) is the set of immediate predecessors (successors) of v_j and h_{init} is defined as,

$$h_{init}(v_i) = \begin{cases} 0, & if R_{v_i} = \emptyset, \\ 1 + \max\limits_{v_j \in R_{v_i}} h_{init}(v_j), & otherwise. \end{cases} \quad (5.4)$$

Initial Population: To initialize each string, we first calculate the height value for each task in the mode associated with that string. Next, for each height $\hbar$, we pick a random task v_r from $V(\hbar)$ ($V(\hbar)$ is defined as the set of tasks in G with height $\hbar$), and assign it to the string. We repeat the random selection until all remaining tasks from $V(\hbar)$ are assigned to the string.

Genetic Operators: The crossover and mutation operators for the core-EA plus are as follows:

- Crossover — Crossover in coreEA-plus is a two-step process. First, one of the two data structures (assignment matrices or sorted string) representing the solutions is randomly selected for manipulation. A call is then made to the crossover operation pertaining to that structure. Both child solutions receive a copy of the newly-generated structure, and the remaining structure is directly copied from one of the two parents. The crossover operator for assignment matrices are described in previous section. The crossover operator for the sorted string is described as follows: We cut each string in 2 parts by randomly choosing a height h and partitioning the tasks with heights larger and smaller than h into right and left sets respectively. We keep the left sets and exchange the right sets to get two new strings.

- Mutation — Mutation, like crossover, is a two-step process. First, one of the two data structures of the solution representation is randomly selected for manipulation. A call is then made to the mutation operation pertaining to that structure. The mutation operator for the sorted string is described as follows: We randomly choose a task v_i, then pick another task v_j among all the tasks with the same height as v_i at random and then exchange the position of the two tasks.

Fitness Evaluation: Fitness evaluation of coreEA-plus is relatively simpler compared to the coreEA because the ordering decision is already made using the sorted string. So as in the initial coreEA, once the ordering and assignment of each task is known we calculate other objectives and constraints given in Ω_0.

5.6 Parallel CHARMED

Fitness evaluation is usually very time-consuming. Fortunately, however, there is a large amount of parallelism in the overall fitness evaluation process. Therefore, we have developed a parallel version of the CHARMED framework that provides an efficient and highly scalable means for leveraging additional computational power to reduce synthesis time.

We employ the asynchronous master-slave parallelization model [50]. This method, also known as distributed fitness evaluation, uses a single population and the evaluation of the individuals and/or the application of evolutionary operators are performed in parallel. The selection and mating is done globally, hence each individual may compete and mate with any other. The evaluation process normally requires only knowledge of the individual being evaluated (not the whole population), so there is no need to communicate during this phase, and this greatly reduces overhead. The asynchronous master-slave algorithm does not stop to wait for any slow processors and/or until all the evaluations are returned and selection waits only until a fraction of the population has been processed. A high level description of parallel CHARMED is given in Figure 5.9.

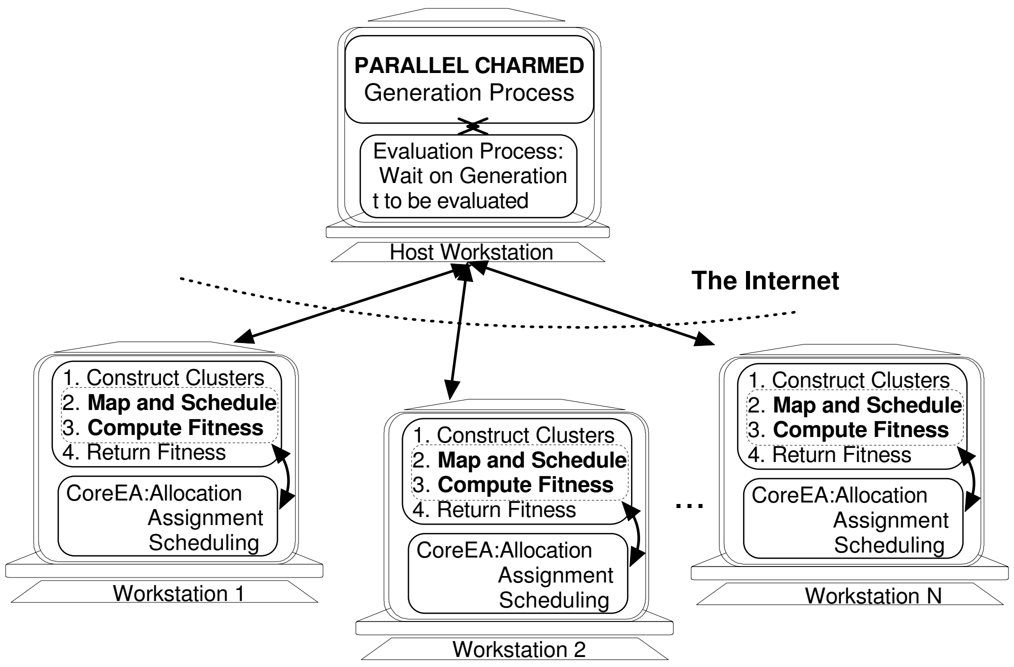


Figure 5.9: Parallel CHARMED framework.

5.7 Experimental results

We evaluated our proposed co-synthesis framework on several benchmarks to demonstrate its capability to produce high quality solutions in terms of different optimization goals. All algorithms were implemented using C++. The benchmarks consist of 12 random task graphs TG1-TG12 ($10 \sim 100$ nodes) and 7 multi-task graphs MTG1-MTG7 that were generated using TGFF [33]. The population size of MCFA and coreEA are 100 and 50. MCFA runs for 1000 generations and coreEA runs for 500 generations. In case of parallel CHARMED, the fraction of the population that the algorithm waits on, is 80% of the original population, in other words once it receives the results from 80 members (running remotely) it proceeds to the next generation.

There are several aspect of CHARMED framework that we would like to evaluate as follows: i) its hierarchical structure and use of the pre-processing step of clustering; ii) multi-mode optimization; iii) multi-objective optimization (CHARMED vs. CHARMED plus) and handling dynamically reconfigurable hardware and vi) parallel CHARMED. Experimental results for each step are as follows:

i) Effect of clustering — To study the effectiveness of our clustering approach we first run CHARMED without the MCFA pre-processing step (coreEA only) and apply coreEA directly to tasks (instead of clusters of tasks). Next, we run CHARMED with the clustering step, i.e. MCFA + coreEA. The goal is to optimize price, power consumption and parallel-time. The results for a subset of graphs are given in Table 5.3. It can be seen from the table that CHARMED finds better quality solutions when it is employed with the pre-processing step of clustering. Additionally, as the size of the task graph increases, clustering step becomes more effective. Solutions found by CHARMED using clustering (MCFA + coreEA) dominate 50% to 100% of the solutions found using CHARMED without clustering (or coreEA).

ii) Multi-mode optimization — In order to study the effect of integrating multiple system modes and optimizing them jointly vs. optimizing modes separately, we created multi-mode applications by combining task graphs from the TG set (interpreted each task graph as a separate mode). We run CHARMED once for each mode and once for the combinations of modes. The optimization goals for these tests are area and parallel-time. Results are given in Table 5.4. The M_is in the table represent individual modes, corresponding to task graphs listed in the first column respectively. ”Total” represents the estimated system area. It can be seen from the table that optimizing all system modes simultaneously can significantly improve the results.

Table 5.3: Effect of clustering: CHARMED without clustering step (coreEA only) vs. CHARMED with clustering step (MCFA + coreEA)

Task Graph	$\|V\|/\|E\|$	coreEA Price	coreEA Power Consumption	coreEA Parallel-Time	MCFA + coreEA Price	MCFA + coreEA Power Consumption	MCFA + coreEA Parallel-Time
TG2	18/25	*122*	*92.03*	*107*	122	87.9	107
		196	*102.6*	*101*	195	133.6	99
		226	121.5	92	196	98.7	100
		330	142.2	91	196	108.2	93
TG3	28/47	*226*	*199.3*	*195*	226	169.2	162
		226	*243.4*	*153*	226	173.1	151
		299	*241*	*153*	226	210.5	150
		330	*282.9*	*136*	330	270.4	135
TG5	48/85	*226*	*413.4*	*321*	122	250.8	274
		330	*374.6*	*241*	226	280.6	262
		330	419.4	230	226	353.1	241
		330	460.1	223	330	411.4	240
TG7	68/119	361	563.2	300	361	606.9	270
		361	*638.1*	*276*	361	631.3	261
		465	*805.5*	*267*	361	641.2	251
		495	*619.6*	*277*	361	691.4	245
TG9	88/161	*496*	*895.4*	*333*	465	927.3	302
		496	*1031.3*	*313*	465	938.2	288
		600	*881.5*	*358*	465	969.2	283
		600	*929.7*	*324*	496	873.4	299
TG10	98/181	630	1049.6	353	496	1054.6	343
		630	*1075.586*	*342*	496	1070.6	332
		630	*1092.6*	*330*	496	1107.9	324
		630	*1140.2*	*328*	496	1168.3	322

iii) CHARMED-plus — To demonstrate the performance of CHARMED-plus i.e. CHARMED equipped with ordering strings, we compared the results of scheduling for both approaches. In CHARMED the tasks' ordering is based on their blevel metric and in CHARMED-plus the priority of each task is given in an ordered string and is determined and modified in the evolutionary process. The results are shown in Table 5.5 and represent the parallel-time and the power consumed during reconfiguration.

vi) Parallel CHARMED — We also compared the performances of parallel CHARMED

Table 5.4: Effect of optimizing modes separately vs. optimizing all modes jointly.

Task Graph	Separate Modes				Integrated Modes				
	Area				Area				%imp.
	M_1	M_2	M_3	Total	M_1	M_2	M_3	Total	
TG1&TG2&TG3	0.509	0.809	0.509	1.118	0.509	0.818	0.509	0.818	26.8
TG3&TG5&TG6	0.509	0.818	0.780	1.39	0.509	0.818	1.09	1.09	21.6
TG2&TG6&TG7	0.809	0.780	1.08	1.39	0.818	1.09	1.09	1.09	21.6

Table 5.5: CHARMED-plus vs. CHARMED scheduling results

Task Graphs	$\|V\|/\|E\|$	CHARMED		CHARMED-plus		%Improvement	
		Recon. Power	Parallel Time	Recon. Power	Parallel Time	Recon. Power	Parallel. Time
TG1	8/7	23.04	151.87	22.86	153.44	0.76	-1.04
TG2	18/25	69.06	575.49	66.54	572.19	3.64	0.57
TG3	28/47	90.82	745.63	90.00	733.12	0.91	1.68
TG4	40/63	731.75	1645.98	145.19	927.84	80.16	43.63
TG5	48/85	189.75	1372.75	168.82	1193.07	11.03	13.09
TG6	58/97	1438.06	3320.72	1263.49	3113.22	12.14	6.25
TG7	68/119	3433.37	4896.27	2675.22	4718.05	22.08	3.64
TG8	78/143	3458.36	5364.63	2816.02	5354.74	18.57	0.18
TG9	88/161	3068.15	5208.73	3060.25	4906.48	0.26	5.80
TG10	98/181	4629.97	6299.23	3404.66	5713.79	26.46	9.29
TG11	31/56	88.96	770.84	88.22	705.96	0.83	8.42
TG12	36/50	111.79	762.82	105.30	762.04	5.81	0.10
Avg. Improvement		-		-		**15.22**	**7.6**

and CHARMED. The results of our comparison clearly show the effectiveness of the parallelization techniques employed in parallel CHARMED. If we run both algorithms for the same amount of time, the solution quality of parallel CHARM-ED is significantly better than CHARMED's solution quality. Results of running CHARMED and parallel CHARMED on a subset of TG set to optimize for price and power, are given in Table 5.6. If we run both algorithms for the same number of generations, parallel CHARMED achieves a speedup between 4 to 8. This number however is a function of number of available remote hosts, network traffic and problem size. For these tests, we used a network of 24 workstations of Sun Ultra 5/10 UPA/PCI (UltraSPARC-IIi 440MHz) systems.

Table 5.6: Performance Comparison of CHARMED vs. parallel CHARMED

Task Graph	$\|V\|/\|E\|$	CHARMED		Parallel CHARMED	
		Price	Power Consumption	Price	Power Consumption
TG1	8/7	*61*	*37.7*	61	36.9
TG2	18/25	*92*	*103.9*	92	87.9
TG5	48/85	*239* *269*	*607.2* *522.4*	239	447.3
TG6	58/97	*269*	*714*	239 269	721 675.9
TG9	88/161	*509*	*1187.7*	496	1041.1
TG10	98/181	*540* *556*	*1421.4* *1274.3*	539	1299.1

5.8 Summary and Conclusions

In this chapter, we have presented a modular framework called CHARMED for hardware-software co-synthesis of multi-mode embedded systems. At the heart of CHARMED is an efficient new evolutionary algorithm, called coreEA, for allocation, assignment, and scheduling of clustered, multi-mode task graphs. CHARMED also includes a novel integration of several synergistic techniques for multi-objective co-synthesis, including a hierarchical evolutionary algorithm architecture; the CFA-based binary representation for clustering [68]; the SPEA method for multi-objective evolutionary algorithms [154]; the constraint dominance concepts of Deb et al. [27]; and optionally, the asynchronous master-slave parallelization model [50]. Our framework allows the designer to flexibly control the performance evaluation process by configuring design evaluation metrics as optimization goals or constraints, as desired. This flexibility enables the designer to narrow down the search to special regions of interest. Our parallelization technique, parallel

CHARMED, is shown to provide an efficient means for applying additional computational resources to the co-synthesis process. This enables application of CHARMED to large instances of co-synthesis problems.

Chapter 6

CASPER: An Integrated Framework for Energy-Driven Scheduling on Embedded Multiprocessor Systems

For multiprocessor embedded systems, task scheduling, which includes assigning tasks to processors and deciding the execution order of tasks on the same processor, has been well-studied as a tool to minimize an application's parallel-time (3.1) (or completion time). In addition to real-time constraint, energy consumption has become a major design issue for modern real-time embedded systems, especially battery-operated portable devices. Such systems also operate under tight and hard deadlines. While the early task completion (before the deadline) may not bring the system extra benefit, one can utilize the extra available time to improve other valuable system performance parameters such as energy consumption. Energy consumption is a quadratic function of the supply voltage and processor speed, and reducing the supply voltage and thus processing speed can save energy, but at the cost of increased execution time. Dynamic voltage scaling (DVS) is a promising method for embedded systems to exploit multiple supply voltage and clock frequency levels and to achieve the highest possible energy efficiency for time-varying computational loads while meeting the deadline constraint.

Currently dynamic power is still the dominant factor in designs for most embedded systems. While we acknowledge that with the continuous feature size reduction (below 0.1 micron) the static power becomes one of the foremost challenges, we believe that the dynamic power stays one of the fundamental challenges facing the designers. This trend is specially evident in large circuits and increased functionality requirements as well as continuing emphasis on rising clock frequencies [133]. Hence, in this work, we will mainly focus on the reduction of dynamic power.

A large number of papers devoted to the energy-aware voltage-scheduling problem only consider single-processor systems or independent tasks (e.g., see [62]). There are also some research works that address dependent tasks on multiprocessors. However, in most of these works the authors propose that their algorithms are to be used in the inner loop of a system-level optimization tool and hence proceed with the assumption that either the whole process of task assignment to the processors and the ordering of tasks or one of these steps (task assignment or task ordering) is determined a priori and do not factor the effect of ordering or assignment in the DVS results [51][91][101]. One serious drawback to this assumption is that globally optimal voltage scheduling may not be generated. We believe that the integration of task assignment and ordering and voltage scheduling is essential since different assignments and orderings provide voltage schedulers with great flexibility and potential energy saving that can be achieved. Additionally, since DVS utilizes slack in the schedule to slow down processes and save energy, therefore, it is generally believed that the maximal energy saving is achieved on a schedule with the minimum parallel-time, or equivalently the maximal slack. This is another reason that most current approaches treat task assignment, scheduling, and DVS separately. In this work, we present a framework called CASPER (Combined Assignment, Scheduling, and PowER-management) that challenges this common belief by integrating task scheduling and DVS under a single iterative optimization loop via generic algorithm.

The idea of integrating scheduling into the power management process has been studied for heterogeneous multiprocessor systems. The work in [122] employs two nested genetic algorithms (GAs) where the outer GA generates the assignments and the inner one explores various orderings. This algorithm is not however efficient in terms of run time. Furthermore, little research has been done for such integration for the homogeneous multiprocessor case. Although the homogeneous scenario can typically be handled as a special case of techniques that address heterogeneous multiprocessor systems, one can expect that when we limit the target architecture to homogeneous processors, the result would better reflect the effect of ordering on DVS as the effect of processor selection and assignment has been toned down. Additionally, all the available compile time is spent on optimizing the task ordering and scheduling that would have otherwise been divided among allocation and assignment.

In our approach, we present a genetic algorithm framework that can be applied to both heterogeneous and homogeneous multiprocessor systems. It thoroughly searches the solution space to find an assignment and ordering of tasks on each processing element (PE) and generates a schedule that meets the deadline and minimizes the power consumption simultaneously. We study the impact of i) integrating the scheduling process into the power optimization framework for DVS-enabled embedded multiprocessor systems and ii) combining task mapping, ordering and scheduling and encoding them in the form of a single chromosome in a GA framework. Additionally, we also present a refinement to be applied to CASPER's solutions (i.e. schedules) to speed up the convergence process of CASPER to better quality solutions.

The remainder of this chapter is organized as follows: The energy-efficient mapping and scheduling problem is defined in Section 6.1. In Section 6.2 we present our solution to this problem. Results and comparisons are given in Section 6.3. Finally, we conclude the chapter in Section 6.4.

6.1 Problem Statement and Assumptions

We consider an embedded application represented in terms of an acyclic task graph $G = (V, E)$ that was introduced in 2.3. The multiple processor system being used to implement the application consists of n_{PE} processor elements (PE) of the following types: general-purpose processors, application-specific integrated circuits, and FPGAs. We also assume that tasks can have hard or soft deadlines. A hard deadline must be met at runtime to ensure the correctness and feasibility of the solution. We represent the set of tasks with hard deadlines as V_d.

We adopt the following models for the DVS-enabled processor's dynamic power consumption p_d and operational frequency f:

$$p_d = C_{ef} \cdot V_{dd}^2 \cdot f, \tag{6.1}$$

$$f = k \cdot (V_{dd} - V_t)^2 / V_{dd}, \tag{6.2}$$

where C_{ef} is the effective switching capacitance, V_{dd} is the supply voltage, k is a circuit dependent constant and V_t is the threshold voltage. The problem formulation remains the same and our approach are still applicable for other models. We use them mainly for the purpose of illustrating the idea and comparison with existing results where these models are used [55][120][122].

Given the application and multiple processor system described briefly as above (introduced in 2.3), we want to find (i) a mapping of tasks to PEs, (ii) an ordering of the tasks and edges, and (iii) the voltage profile for each task such that all the hard deadline constraints are met and the total energy consumption is minimized.

6.2 Proposed Algorithmic Solution

Our proposed solution is an iterative improvement framework that integrates task assignment, ordering and scheduling, and static power management all in one phase. We call this framework the Combined Assignment, Scheduling, and PowER-management algorithm or CASPER. A high-level overview of CASPER is depicted in Figure 6.1.

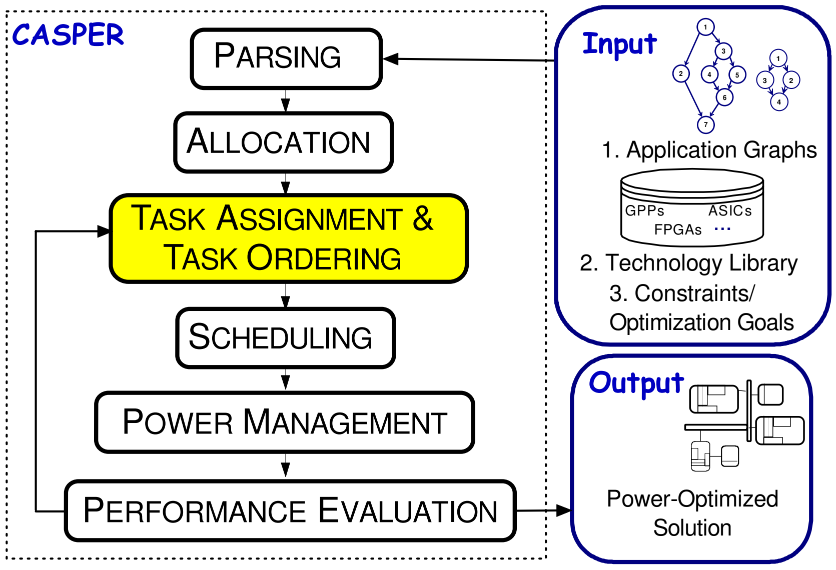


Figure 6.1: CASPER framework.

CASPER takes the application task graphs, the sets of PEs and CRs, and the constraint/optimization requirements as input. It first parses these inputs and creates the appropriate data structures. Next, it uses a simple standard algorithm to allocate PEs and CRs such that every task and every edge has at least one instance of PE or CR that it can execute on. It then uses a genetic algorithm that combines the task assignment, ordering and scheduling, as well as power management by DVS to find the most energy efficient solution (see the loop in Figure 6.1). Details on the genetic algorithm is given below in Section 6.2.1. Section 6.2.2 briefly describes the two power management techniques that we have selected for homogeneous and heterogeneous multiple processor systems. We mention that any slack distribution based power management method can be integrated into the CASPER framework. We selected these two because they provide the best available results. And Section 6.2.3 describes the details of a refinement strategy devised to speedup the convergence of the GA to a better quality solutions.

6.2.1 Combined Assignment and Scheduling

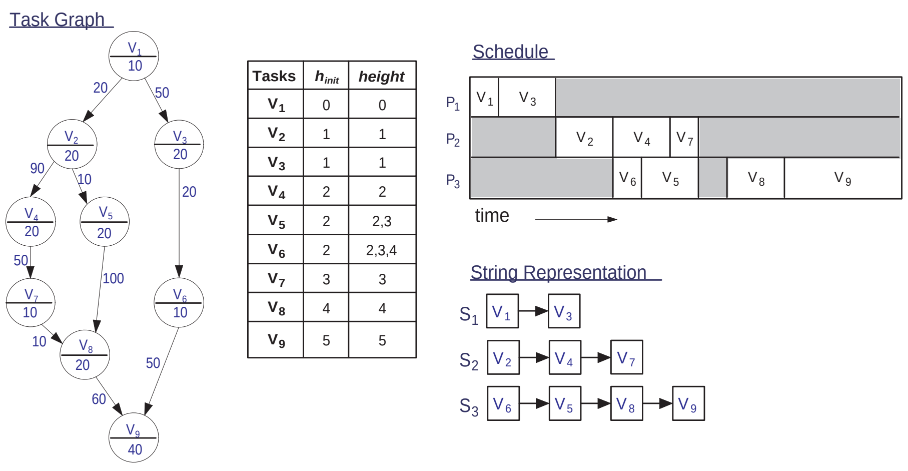
The core part of CASPER is a genetic-list scheduling algorithm that encodes both assignment and ordering in a single chromosome (similar representation has been used for a multiprocessor scheduling algorithm called CGL [22]). In this representation, each individual solution or chromosome is encoded as a list of n_{PE} strings, with each string corresponding to one allocated PE of the target system (n_{PE} represents the number of allocated PEs in the system). The strings maintain both the assignment and execution order of the tasks on each PE. Figure 6.2 illustrates the relationship between an arbitrary schedule for a task graph and its corresponding string representation for a homogeneous multiprocessor system. The list of tasks within each PEs of the schedule is ordered in ascending order of the task heights, which guarantees that the precedence constraints are satisfied. The definition of height $height(v_j)$ of a task v_j is given in section 5.5 and is repeated below for convenience. Height of a task v_j is a random integer whose value is such that:

$$\forall v_i \in R_{v_j} \wedge \forall v_k \in U_{v_j} : max(h_{init}(v_i)) + 1 \leq height(v_j) \leq min(h_{init}(v_k)) - 1 \quad (6.3)$$

where R_{v_j} (U_{v_j}) is the set of immediate predecessors (successors) of v_j and h_{init} is defined as,

$$h_{init}(v_i) = \begin{cases} 0, & if R_{v_i} = \emptyset, \\ 1 + \max\limits_{v_j \in R_{v_i}} h_{init}(v_j), & otherwise. \end{cases} \quad (6.4)$$

A randomized version of list scheduling is used to generate the initial population as



Tasks	h_{init}	height
V_1	0	0
V_2	1	1
V_3	1	1
V_4	2	2
V_5	2	2,3
V_6	2	2,3,4
V_7	3	3
V_8	4	4
V_9	5	5

Figure 6.2: Illustration of the string representation of a schedule.

follows: For each height $\hbar$ perform the following steps: (1) Pick a random task v_r from $V(\hbar)$ ($V(\hbar)$ is defined as the set of tasks in G with height $\hbar$). (2) Pick a processing element pe_r that can execute v_r at random. (3) Assign v_r to pe_r. (4) Repeat Steps (1)-(3) until all remaining tasks from $V(\hbar)$ are scheduled [22].

Once the population is generated, the chromosomes' fitness needs to be evaluated. The chromosome's performance measure or fitness consists of two parts: the first part is a measure of constraint satisfaction (satisfying the deadline) and the second part is based on the schedule performance with respect to energy ($\sum_E$). This is because objective measures are, in practice, meaningless if the schedule is infeasible (i.e., violates the constraints). Hence, the optimization measures should not be considered until the given constraint has been satisfied. The degree to which the constraint is violated determines how feasible the schedule is, and if the schedule is feasible the objective performance is then considered.

It is also important to notice that if a single fitness value can represent both an infeasible solution with good objective performance and a feasible solution with poor objective performance, the GA may be deceived and end up favoring infeasible solutions with better objective fitness values. The fitness value for this problem with (time) constraint performance measure τ_{const} and (energy dissipation) optimization performance measure $\sum_{Eopt}$ for each individual chromosome I_i in population P_t is defined in (6.5):

$$fitness_i(I_i, P_t) = \begin{cases} \frac{\tau_{const}(I_i,P_t)}{2} & if(\Delta_c(I_i, P_t) > 0), \\ \frac{1+\sum_{Eopt}(I_i,P_t)}{2} & if(\Delta_c(I_i, P_t) \leq 0), \end{cases} \tag{6.5}$$

where

- $\tau_{const}(I_i, P_t)$ is the constraint performance measure and is defined as,

$$\tau_{const}(I_i, P_t) = \begin{cases} \frac{1}{1+\Delta_c(I_i,P_t)} & if(\mathit{\Delta}_c(I_i, P_t) > 0), \\ 1 & if(\mathit{\Delta}_c(I_i, P_t) \leq 0). \end{cases} \tag{6.6}$$

  Here, $\Delta_c(I_i, P_t)$ is a measure of time constraint (deadline) violation and is defined as $\Delta_c(I_i, P_t) = \sum\limits_{v \in V_d} (\tau_{end}(v) - \tau_d(v))$, where $\tau_{end}(v)$ is the finish time of task v in the schedule and $\tau_d(v)$ is task v's hard deadline.

- $\sum_{E\,opt}(I_i, P_t)$ represents the fitness of the individual chromosome I_i with respect to the energy p and is defined as

$$\sum_{E\ \ opt}(I_i, P_t) = \frac{\max\limits_i(\sum_E(I_i, P_t)) - \sum_E(I_i, P_t)}{\max\limits_i(\sum_E(I_i, P_t))}. \tag{6.7}$$

Most research works give the most weight to the solutions that have a larger global slack (the difference between the deadline and the parallel-time and do not consider local slack (gaps between the tasks) as important. Such techniques employ scheduling algorithms that find the minimum-length parallel-time and feed their results to the associated power management algorithms. However, we regard both global and local slack as equally important, and consequently use an integrated approach to find a solution that has an overall slack distribution (global and local) that saves the most energy. This can also be seen from Equation (6.5). The second condition of this equation shows that for all the solutions that satisfy the time constraint (or meet the deadline) the effect of constraint satisfaction is a constant number and not a function of the global slack.

It can also be observed from Equation (6.5) that infeasible solutions are allowed in the solution. Considering infeasible solutions in the intermediate states of optimization is to make the solution space as continuous as possible. In complex systems we expect that most of the obtained schedules are not feasible. If such schedules are not accepted as members of the population then we cannot guarantee that starting from any solution the entire solution space can be searched.

The selection process allows the algorithm to take biased decision favoring good solutions. We use the "roulette wheel" principle to randomly select an individual in population P_t. The better the fitness of the individual the better the odds of it being selected. The selected individuals are then crossed to make new solutions (a cutting place is decided based on a randomly chosen height). The mutation randomly transforms a solution to a new solution with a single exchange of two tasks in the scheduled solution. By use of the height values, crossover and mutation always maintain the precedence constrains and hence never generate any invalid solutions(see S6 and S7 in Fig. 6.3). Once the new individuals are generated, the genetic algorithm proceeds by evaluating the new solutions and repeating the same steps of selection, crossover and mutation until the termination condition is met (such as the maximum number of generation is reached or the energy saving in two consecutive generations is less than 1%).

The outline of our algorithm is presented in Figure 6.3. The power management algorithms used in Step 3 are described in the following section.

INPUT: A task graph G, n_{PE} PEs and time-constraint τ_d.
OUTPUT: An energy-optimized mapping of the task graph onto multiple PEs.

Step 1 Generate initial population P_t (of size POP_SIZE) where each individual is a list of strings of size P. Each string represents an ordering of a subset of tasks on a PE.
Step 2 Compute the finish times of tasks for each individual.
Step 3 Apply the power management algorithm to each individual and compute the corresponding energy dissipation $\sum_E$.
Step 4 Calculate the fitness of each individual based on τ_{const} and $\sum_{E_{opt}}$.
Step 5 Select k individuals from P_t according to their fitness values using a roulette wheel, where $k = POP\_SIZE$.
Step 6 Perform the crossover operation $\frac{k}{2}$ times to generate k new "offspring" individuals: cut each string in 2 parts by randomly choosing a height h and partitioning the tasks with heights larger and smaller than h into right and left sets respectively.
Keep the left sets and exchange the right sets to get two new strings.
Step 7 Perform the mutation (with low probability): randomly choose task v_i, then pick another task v_j among all the tasks with the same height as v_i at random and then exchange the position of the two tasks.
Step 8 If the maximum number of generations is reached stop, otherwise go to Step 2.

Figure 6.3: Flow of CASPER

6.2.2 Power Management Techniques

In this part, we briefly introduce the power management algorithms that we use in our experimentation. Specifically, we use the *Static Power Management with Proportional Distribution and Parallelism Algorithm* (PDP-SPM) for homogeneous system and the Power Variation Dynamic Voltage Scheduling (PV-DVS) algorithm for heterogeneous systems. The only reason that we are using them in Step 3 of Figure 6.3 is that they have been reported to outperform other techniques in energy efficiency by a large margin. More details about these two algorithms can be found in [55] and [122] respectively. However, the proposed CASPER framework can adopt any existing power management methods.

Static Power Management with Proportional Distribution and Parallelism Algorithm (PDP-SPM)

PDP-SPM algorithm is a static power management (SPM) technique for homogeneous system to reduce energy consumption by utilizing slack, both global and local, and parallelism among the processors. For a scheduled task graph, it applies the following two phases repetitively: (1) proportionally distribute the slack among the tasks under the deadline constraint; and (2) create new (local) slack based on parallelism and return to the first phase to re-distribute it.

In the first phase, the algorithm distributes slack, both the global and local static slack, to the tasks hierarchically. First, the global slack is distributed to all vertices proportionally to their execution time. Each vertex will have its execution time scaled up by a factor of δ. However, this does not guarantee that the new parallel-time will be increased by the same factor δ because the inter-processor communication cost does not scale. Therefore, this process is applied repetitively until the new parallel-time violates the deadline τ_d. Then the CPU time assigned to all the vertices along critical paths will be scaled down to meet the deadline and marked as final. There may still exist local slack and hence the algorithm continues to scale up the execution time for those vertices that have not been marked as final. At the end of this phase, little or none slack is expected.

In the second phase, PDP-SPM re-allocates the CPU time assigned to each task based on the system's degree of parallelism (that is, the number of PEs running at the same time). The basic idea is to *create new slack* by reducing the CPU time assigned to the tasks with the minimal degree of parallelism. Such new slack will be redistributed using the same procedure as in the first phase. If this results in energy reduction, CPU time will be reduced from this same task again until little or no energy saving can be achieved. Then this process restarts with another task of the minimal degree of parallelism until all the tasks are examined.

Power Variation (PV) DVS Algorithm

For heterogeneous system, we consider PV-DVS algorithm, which reports significantly higher energy reduction than other DVS scheduling approaches [122]. This algorithm is based on a constructive heuristic using the energy difference ($\Delta E(v)$): the energy saving obtained by extending task v's execution time by a time quantum of Δt.

The algorithm first calculates the available slack times of each hard deadline task to identify all extendable tasks. Next, it calculates the slack time of all tasks and inserts all the tasks with a slack time greater than a Δt_{min} into a priority queue. The energy difference $\Delta E(v)$ for all the extendable tasks in the priority queue are then calculated and the queue is sorted in decreasing order of the energy differences (or tasks energy saving potential). The algorithm then iterates until no extendable tasks are left in the priority queue.

In each iteration the algorithm picks the first element of the priority queue and extends it by Δt and updates the energy dissipation value of the selected task. The extension is then propagated through the mapped and scheduled task graph. Next, the inextensible tasks are removed from the extendable task priority queue. Taking into account the tasks in the priority queue the time quantum Δt is recalculated, energy differences are updated and priority queue is reordered. At this point, the algorithm either invokes a new iteration or ends, based on the state of the extendable queue.

6.2.3 Refinement

CASPER, or any genetic algorithm (GA), with appropriately set parameters (e.g. initial population or crossover/mutation rate) should be able to search the entire solution space (in case of CASPER, find all different scheduling of the application) and find the global optimum. However, this may take a very long time. One promising approach for improving the convergence speed to the optimal (sub-optimal) solution is the use of local search in GAs. Such hybridizations of genetic algorithms with local search are inspired by models of adaptation in natural systems that combine the evolutionary adaptation of a population with individual learning within the lifetimes of its members [77]. These methods have been the subject of many studies [98][76] and it has been shown that GAs if combined with the neighborhood search algorithms can improve their search-abilities and perform well (even superior in some instances compared to simple GAs) on complex combinatorial optimization problems. The idea of local search is to refine a given initial solution point in the solution space by searching through the neighborhood of the solution point (see Figure 6.4). In our combined assignment and scheduling (CASPER) algorithm, the right choices for assignment and ordering are what provide the SPM algorithm with more energy saving opportunities. Hence it will be beneficial to employ a local search that improves these aspects of the schedule. While the CASPER's initialization, mutation and crossover techniques are very effective and efficient and capable of generating

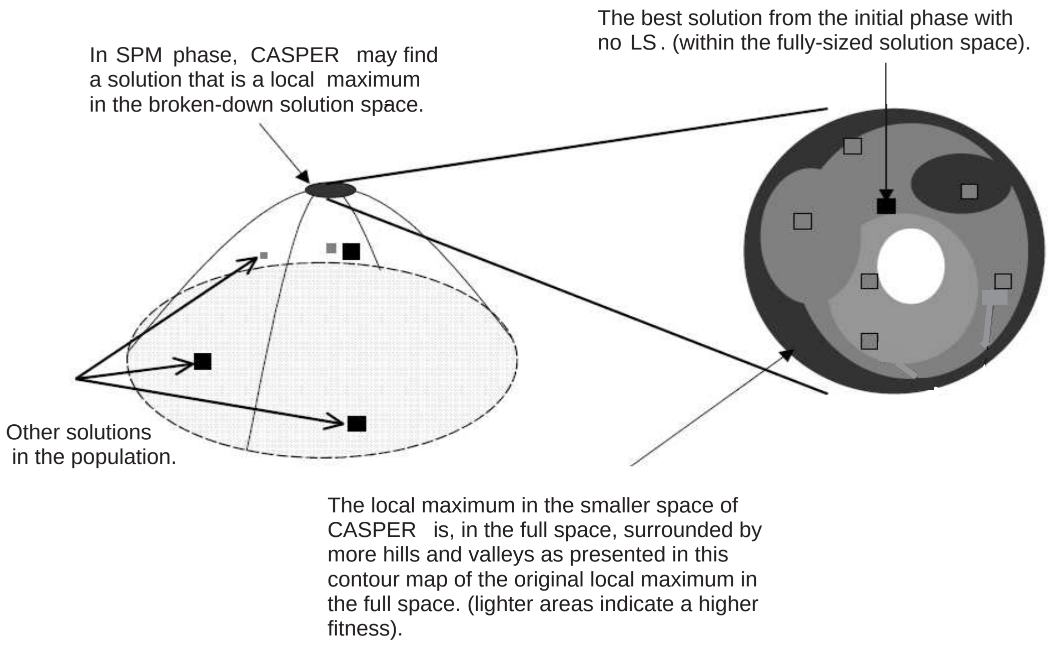

The best solution from the initial phase with no LS. (within the fully-sized solution space).
In SPM phase, CASPER may find a solution that is a local maximum in the broken-down solution space.
Other solutions in the population.
The local maximum in the smaller space of CASPER is, in the full space, surrounded by more hills and valleys as presented in this contour map of the original local maximum in the full space. (lighter areas indicate a higher fitness).

Figure 6.4: Neighborhood search of a Local Maximum.

every possible solution (i.e. schedule) [22] and the rules governing the evolution process (such as survival of the fittest) guide each generation toward better starting points in the solution space; however, the use of knowledge to guide the search can be quite valuable and effective. In CASPER (or almost all other scheduling + SPM techniques), the only information taken from the schedule after application of SPM is the amount of saving, in our refinement phase we take advantage of other information such as the new execution times and voltages for some knowledge-based guidance. The SPM algorithm keeps scaling the voltage (execution times) till no further energy reduction can be achieved. This results in an application with new execution times. Now the question is that if the scheduler had initially started with these new and slower execution times and had tried to optimize this application for performance and energy would we have ended up with the same results? This is certainly a question worth exploring for an answer. This idea is the base for our local search algorithm. In our hybridized implementation of CASPER (HCASPER) or CASPER with local search the employed LS operator is applied to all solutions in the offspring population, before applying the selection operator (after Step 4 and before Step 5 in Figure 6.1). An outline of the employed local search (LS) algorithm is given in Figure 6.5.

Step 1	Start from an initial solution s
Step 2	Find a neighbor solution $s\prime$ of s.
Step 3	If $s\prime$ is better than s, set $s = s\prime$ and return to Step 2.
Step 4	Stop and return.

Figure 6.5: Outline of the local search algorithm

In this algorithm, the initial solution in Step 1 of the algorithm is a schedule with SPM applied to it. In Step 2, a neighbor solution is found by re-scheduling the solution using the new execution times resulted from applying the SPM technique and re-applying the SPM. In Step 3 the new results are evaluated and if there has been further energy saving, Step 2 is repeated, otherwise the local search returns with no change to the initial solution.

We have employed two different re-scheduling techniques (employed in Step 2 of Figure 6.5) to find new solutions as follows:

- **Ordering-Only (OO):** The ordering-only re-scheduling technique is based on CRLA algorithm introduced in 3. Original CRLA takes n clusters (where each cluster includes several tasks) and maps them to m identical processors where $n > m$, orders them on the processors and schedules them. The modified version of CRLA takes the mapping (or assignment) information as an input from the to-be-refined solution as well and hence its function is only to order tasks on their designated processors and schedule them. Since the execution times of tasks have changed, the relative priority of tasks have changed as well and hence CRLA potentially can generate a different schedule. The SPM algorithm is then applied to this newly generated schedule.

- **Assignment and Ordering (AO)** The assignment and ordering re-scheduling strategy, employs a modified version of HEFT algorithm [137] that is a very efficient heterogeneous multiprocessor scheduling technique. This algorithms re-schedules (assignment, ordering and scheduling the application entirely, using the new execution times. An outline of the employed algorithm is given in Figure 6.6.

```
1. Compute blevel for all tasks.
2. Sort all tasks in a ready-list by non-increasing order of blevel values.
3. WHILE there are unscheduled tasks in the list
4.     Select the first task $v_i$ from the ready-list
5.         FOR each PE $p_j$
6.             IF (($t(v_i, p_j) \leq t(v_i)$) AND ($p_j$ is DVS-enabled) AND ($V_{dd}(t_i, p_j) > V_t(t_i, p_j)$))
7.                 Compute $EFT(t_i, p_j)$ value using an insertion-based scheduling policy
8.         Assign task $t_i$ to PE $p_j$ that minimizes $EFT(t_i)$, break ties using $E(t_i, p_j)$
```

Figure 6.6: Outline of the Assignment and Ordering re-scheduler

As it can be seen from the algorithm, when choosing a PE to map the task onto it, we have to make sure that the target PE is capable of slowing down the task to the level of new execution times. The insertion-based scheduling policy employed in line 6 of the algorithm is also a revised algorithm that only considers holes that exist after inextensible tasks.

Both OO and AO scheduling algorithms check for feasibility of the schedule at each step of the algorithm (i.e. satisfying hard deadlines)

The effectiveness of the refinement step are experimentally evaluated and presented in Section 6.3.

6.3 Experimental Results

The goal of our experiments is twofold: (i) to measure the effectiveness of an integrated framework versus the one that separates task assignment, ordering, and power management; (ii) to evaluate our integrated framework CASPER against another synthesis approach [122], which is the current state-of-the-art.

For the first goal, we compare CASPER with the Heterogeneous/Homogeneous Genetic List Scheduling (HGLS or CASPER without power management). HGLS is the same as CASPER except that the power management phase is moved out from the optimization loop. Therefore, the genetic algorithm finds a solution that is optimized for parallel-time, on which the power management technique will be applied.

For the second goal, we mention that synthesis approach proposed in [122] separates task mapping (assignment) and scheduling into two nested optimization loops. The outer loop (GMA) is a genetic algorithm optimizing for mapping, and the inner loop (EE-GLSA) is an energy efficiency Genetic List Scheduling Algorithm. We hereby refer to this approach as GMA+EE-GLSA.

All algorithms were implemented using LEDA, a C++ class library of efficient graph-related data structures and algorithms, on an Ultra SPARC-IIi/440MHz. The GA parameters are set as follows: population size = 70 with 50% generation overlap, mutation rate = 0.2 and crossover rate = 0.7. We used different sets of benchmarks for homogeneous/heterogeneous target architectures as follows:

- The homogeneous multiprocessors set consists of two subset of task graphs:
    - The first set is the Referenced Graph (RG) set that includes task graphs that have been used by different researchers. This set consists of 10 task graphs that are represented as RG1-RG10. RG1 and RG2 are taken from [4] and [5], respectively. RG3 is a quadrature mirror filter bank, RG4 is based on gaussian elimination for solving four equations in four variables [100], RG5 and RG6 are different implementations of the fast Fourier transform (FFT) [100], RG7 is an adaptation of a PDG of a physics algorithm [100], RG8 is an implementation of the Laplace transform [143], RG9 is another implementation of FFT and RG10 is based on mean value analysis [81]. The deadline assigned to each graph in the RG set was computed using a method similar to that used in [33] based on the graph's maximum length path and the average execution times of the tasks.
    - The second set is the TG set and consists of 5 large random task graphs ($50 \sim 100$ nodes) that were generated using TGFF [33].
- The heterogeneous set consists of 25 TGFF generated task graphs (tgff1 - tgff25) used by Schmitz et al. [122]. The specification includes graphs of 8 to 100 task nodes that are mapped to heterogeneous architectures containing *power managed* DVS-PEs and non-DVS enabled PEs. Accordingly, the power dissipation varies among the executed tasks (with maximal variation of 2.6 times on the same PEs).

6.3.1 Homogeneous System

To evaluate the effectiveness of the integration process, we first ran HGLS, for a given number of generations (500 generations here). Once HGLS generates the final solution (a schedule with minimum parallel-time), we apply the PDP-SPM algorithm to this result and measure the energy saving for the schedule. Next we run CASPER for the same number of generations, using the same PDP-SPM algorithm as the power management method in Step 3 (Figure 6.3) and find a schedule that minimizes the energy consumption while meeting the deadline. We then compare the results. It should be noted that both algorithms indeed use the same task assignment and scheduling scheme with the difference that HGLS generates the minimum-parallel-time schedule with no regard to energy saving while CASPER finds a schedule that consumes less energy. Scheduling and power management are performed at compile time and hence the genetic algorithm run-time can be tolerated.

We assume all PEs are homogeneous and tasks have similar worst case execution times on each PE. The PEs supports DVS with four different voltages and their corresponding clock frequencies as below: ((1.75V,1000MHz), (1.40V, 800MHz), (1.20V, 600MHz) and (1.00V, 466MHz)).

The experimental results for RG and TG sets are given in Table 6.1. The last column labeled $\%improv$ shows the percent improvement (in energy reduction) that the integrated CASPER has vs. the non-integrated approach of HGLS + PDP-SPM. RG graphs are mapped to 4- and 6-PE architectures (depending of the graph size) and TG graphs are mapped to a 6-PE system, which is a reasonable scale for a power/energy-sensitive embedded multiprocessor system. As expected, the parallel-time-driven HGLS

Table 6.1: Energy saving by CASPER and HGLS for RG and TG set.

Task Graph	$\lvert V\rvert/\lvert E\rvert$	τ_d	**HGLS + PDP-SPM** τ_{par}	**% saving**	**Proposed (CASPER)** τ_{par}	**% saving**	**%improv.**
RG1	16/24	65	44	57.4	45	60.7	7.8
RG2	17/28	50	37	49.1	38	54.3	10.2
RG3	14/15	130	102	41.0	102	44.0	5.1
RG4	20/39	2120	1596	50.4	1597	52.3	3.7
RG5	28/32	225	150	57.4	151	61.5	9.5
RG6	28/32	460	265	64.1	265	65.5	4.1
RG7	41/69	925	585	58.5	610	62.2	9
RG8	18/29	665	390	62.0	420	65.6	9.3
RG9	95/158	151	118	47.1	122	50.5	6.4
RG10	361/684	17154	11933	58.8	12818	62.2	8.1
TG1	43/74	1400	1014	47.1	1025	50.5	6.5
TG2	68/119	2000	1345	57.1	1353	59.3	5.3
TG3	93/170	3300	2462	49.3	2472	53.5	8.4
TG4	93/170	3300	2132	59.5	2172	67.3	19.3
TG5	113/216	5400	4325	47.8	4422	50.0	4.2
Average Energy Saving				**53.8**	-	**57.3**	**7.8**

(usually) finds better parallel-times than CASPER (the two columns labeled by μ in Table 6.1). HGLS's achieved energy consumption, however, are consistently worse than that of CASPER. Even in those instances where both algorithms find similar parallel-times (e.g. RG3), CASPER is capable of saving more power. This shows that various task assignment and ordering pairs may generate similar parallel-times, and a non-integrated framework (where the schedule is used as an input to the power-management algorithm) has no way of distinguishing among such solutions on the basis of their energy saving efficiency. On average, HGLS saves 53.8% energy and CASPER saves 57.8%, with a 7.8% improvement over HGLS.

To evaluate the performance of CASPER on different configurations, we ran some tests while varying the deadline and number of processors as follows: First, we varied the number of available processors from 2 to 8 while keeping the deadline fixed (τ_d =

200). Next we varied the deadline from the initial deadline ($\tau_{d_{init}} = 120$, computed as stated earlier in this section) to twice its value by a factor of 20% while keeping the number of processors constant ($|P| = 8$). The results for RG9 are given in Tables 6.2 and 6.3 and graphically presented in Figure 6.7. It can be seen from these tables that CASPER shows significant improvements and maintains its superiority over HGLS under all configurations. In these experiments, CASPER outperforms HGLS by more than 11% on average.

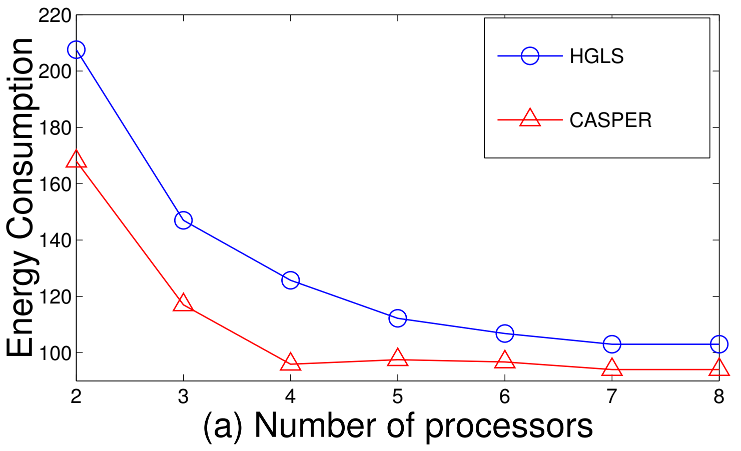
	HGLS + PDP-SPM			**Proposed (CASPER)**			
$\|P\|$	τ_{par}	E_{HGLS}	**% saving**	τ_{par}	E_{CASPER}	**% saving**	**% improv.**
2	150	207.6	28.0	151	168.0	41.6	19.0
3	133	147.0	49.0	134	117.0	56.00	20.4
4	121	125.7	56.3	122	95.9	66.7	23.7
5	114	112.2	61.0	116	97.5	66.1	13.1
6	109	106.8	62.9	110	96.7	66.4	9.5
7	106	103	64.2	108	94.0	67.3	8.7
8	105	103.0	64.2	106	94.0	67.3	8.7
Avg. Saving			**55.1**	-		**62.1**	**14.7**

Table 6.2: Energy Saving by CASPER and HGLS on RG9 task graph with variable number of processors and $\tau_d = 200$.

	HGLS + PDP-SPM			**Proposed (CASPER)**			
τ_d	τ_{par}	E_{HGLS}	**% saving**	τ_{par}	E_{CASPER}	**% saving**	**% improv.**
120	105	158.8	44.8	108	146.7	49.0	7.6
144	105	131.9	54.2	110	117.9	59.0	10.6
168	105	117.4	59.2	105	102.4	64.4	12.8
192	105	106.1	63.1	107	94.7	67.1	10.7
216	105	97.0	66.3	105	94.0	67.3	3.1
240	105	95.0	67.0	105	94.0	67.3	1.0
Avg. Saving			**59.1**	-		**62.3**	**7.6**

Table 6.3: Energy saving by CASPER and HGLS on RG9 task graph with variable deadlines and $|P| = 8$.

In summary, the experimental results presented in this section show that our integrated energy-driven approach achieves 7.0% to 14.7% more energy savings on schedules with longer parallel times.



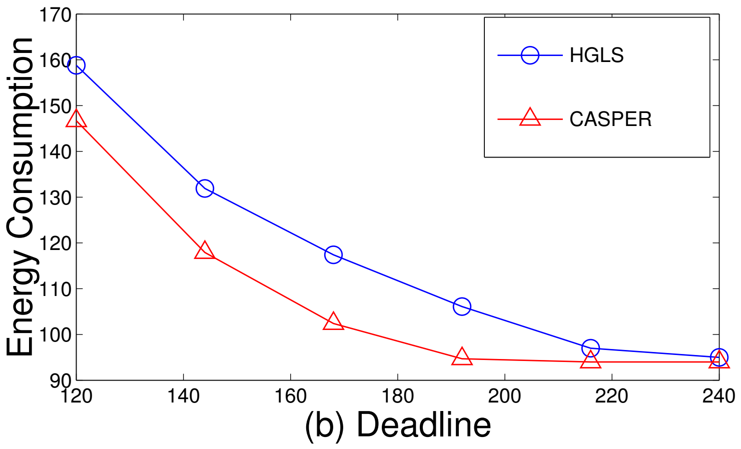


Figure 6.7: Energy consumptions by CASPER and HGLS on RG9 for (a) variable number of processors, (b) variable deadline values.

6.3.2 Heterogeneous System

First, to evaluate the effectiveness of the integration process, we ran HGLS on the tgff task sets and applied the PV-DVS technique to the results for energy optimization. We then ran CASPER on the same task sets with the same amount of run time. Results of these experiments are reported in Table 6.4. Columns 4 and 5 show the energy reduction achieved by HGLS + PV-DVS and CASPER respectively. One can see that CASPER outperforms HGLS + PV-DVS in energy efficiency by 17.13% (the last column). We mention

Table 6.4: Energy saving by CASPER and GMA + EE-GLSA for benchmarks of [122].

Task Graph	$\|V\|/\|E\|$	**GMA + EE-GLSA**	**HGLS+PV-DVS**	**CASPER**	**%improv vs. GMA**	**%improv vs. HGLS**
		%Saving	**%Saving**	**%Saving**		
tgff1	8/9	70.6	78.12	78.44	26.65	1.46
tgff2	26/43	47.08	71.50	76.31	55.24	16.88
tgff3	40/77	66.86	79.57	79.57	38.36	0.00
tgff4	20/33	82.88	20.09	87.62	27.71	84.51
tgff5	40/77	54	76.29	76.47	48.84	0.76
tgff6	20/26	82.14	83.21	85.39	18.19	13.00
tgff7	20/27	28.75	31.68	34.31	7.80	3.84
tgff8	18/26	72.44	15.37	73.23	2.86	68.37
tgff9	16/15	46.28	52.21	61.67	28.64	19.79
tgff10	16/21	23.58	56.37	56.37	42.91	0.00
tgff11	30/29	25.79	20.86	20.86	-6.64	0.00
tgff12	36/50	80.45	13.18	82.73	11.64	80.10
tgff13	37/36	61.22	26.60	51.75	-24.43	34.26
tgff14	24/33	17.09	21.03	23.30	7.48	2.87
tgff15	40/63	22.85	17.30	21.18	-2.17	4.69
tgff16	31/56	28.97	22.27	23.79	-7.29	1.96
tgff17	29/56	45.32	49.66	52.55	13.23	5.75
tgff18	12/15	30.02	28.44	28.44	-2.26	0.00
tgff19	14/19	47.14	36.78	36.78	-19.60	0.00
tgff20	19/25	76.42	77.42	77.42	4.24	0.00
tgff21	70/99	33.41	60.61	72.27	58.35	29.60
tgff22	100/135	47.48	47.81	55.30	14.89	14.36
tgff23	84/151	61.97	83.99	85.76	62.54	11.05
tgff24	80/112	72.08	61.24	69.73	-8.41	21.90
tgff25	49/92	26.44	40.94	48.74	30.31	13.20
Avg. Energy Saving		**50.05**	**46.90**	**58.40**	**17.16**	**17.13**

that we give both algorithms the same run time for a "fair" comparison. During our initial experiments we noticed that CASPER stops before its stopping condition (Step 8 in Figure 3) is reached which meant that it had not converged for most of cases. Since then we improved our implementation of PV-DVS and the CASPER code w.r.t speed and also changed some of the GA parameters such as mutation and crossover. We made the mutation rate variable, to start from a higher value of 0.6 and be reduced to 0.2 after 30% of available time is passed. These changes resulted in significant performance improvement of CASPER.

Next, we compare CASPER framework against GMA + EE-GLSA algorithm using similar configuration (same allocation and constraints). The results are also shown in Table 6.4. Column 3 gives the energy reductions (with respect to a task execution at nominal supply voltage) achieved by mapping and energy efficient scheduling algorithm (GMA + EE-GLSA) presented in [120]. Our results show that the proposed single loop CASPER framework saves 17.16% more energy over GMA+EE-GLSA that uses two nested optimization loops, even when we restrict its run time as explained above. Potentially, the elimination of one loop may also give us large saving in run time.

Despite the changes that we made to the code and the parameter setting, there are still some cases that the CASPER algorithm does not converge within the given time budget. Such cases are shown as negative improvements in column 6 of Table 6.4. As mentioned in section 6.2.3, we equipped CASPER with a local search/quided search technique to improve the convergence of the algorithm. Results for OO and AO re-scheduling techniques are given in Tables 6.5 and 6.6 respectively.

It can be seen that the ordering-only re-scheduling technique does offer only a small improvement while the assignment-ordering based re-scheduling provides significant improvements of the results. The OO technique mostly resulted in little or no improvements after two neighborhood searches while the AO technique kept improving its results for up to several (up to five) neighborhood searches and additional schedules. In applying AO technique nearly 30% of local searches were terminated early due to schedule infeasibility, this trend was not observed in the OO technique. Additionally we monitored the energy saving values before and after applying the refinement process and we observed that in many cases the solution that had a smaller saving provides an overall better saving in the refinement step. This observation again re-confirms the importance of integrating

Table 6.5: Energy saving by HCASPER + OO re-scheduler and GMA + EE-GLSA for benchmarks of [122].

Task Graph	$\|V\|/\|E\|$	**HCASPER + OO %Saving**	**%improv vs. GMA**	**%improv vs. HGLS**
tgff1	8/9	78.82	27.96	3.22
tgff2	26/43	76.31	55.23	16.88
tgff3	40/77	79.57	38.35	0.00
tgff4	20/33	87.73	28.33	84.65
tgff5	40/77	76.49	48.89	0.86
tgff6	20/26	86.97	27.04	22.42
tgff7	20/27	38.28	13.38	9.66
tgff8	18/26	73.23	2.87	68.37
tgff9	16/15	64.69	34.27	26.12
tgff10	16/21	56.37	42.91	0.00
tgff11	30/29	21.19	-6.2	0.42
tgff12	36/50	82.94	12.74	80.35
tgff13	37/36	51.75	-24.42	34.27
tgff14	24/33	24.06	8.41	3.84
tgff15	40/63	21.18	-2.16	4.69
tgff16	31/56	22.73	-8.79	0.6
tgff17	29/56	53.01	14.06	6.66
tgff18	15-Dec	28.44	-2.26	0.00
tgff19	14/19	39.43	-14.59	4.19
tgff20	19/25	78.31	8.02	3.95
tgff21	70/99	72.3	58.4	29.68
tgff22	100/135	55.3	14.89	14.36
tgff23	84/151	86.41	64.27	15.14
tgff24	80/112	69.76	-8.31	21.97
tgff25	49/92	48.82	30.42	13.34
Avg. Energy Saving		**58.96**	**18.55**	**18.62**

the scheduling and SPM under one single framework. In summary, we apply the same power management technique (PV-DVS in this case) in all the three algorithms, their difference in energy efficiency indicates that combining task mapping, ordering, scheduling, and power management in the same loop, rather than separate them, yields better solution.

6.4 Conclusions

In this work we presented an integrated approach for task mapping and scheduling onto homogeneous and heterogeneous embedded multiprocessors using a genetic algorithm. We employed a solution representation (for our GA) that encodes both task

Table 6.6: Energy saving by HCASPER + AO re-scheduler and GMA + EE-GLSA for benchmarks of [122].

Task	$\lvert V\rvert/\lvert E\rvert$	**HCASPER + AO**	**%improv**	**%improv**
Graph		**%Saving**	**vs. GMA**	**vs. HGLS**
tgff1	8/9	80.77	34.60	12.13
tgff2	26/43	76.31	55.23	16.88
tgff3	40/77	79.57	38.35	0.00
tgff4	20/33	87.62	27.69	84.51
tgff5	40/77	94.66	88.40	77.50
tgff6	20/26	85.39	18.20	13.01
tgff7	20/27	46.30	24.63	21.40
tgff8	18/26	96.93	88.87	96.38
tgff9	16/15	61.67	28.65	19.80
tgff10	16/21	56.37	42.91	0.00
tgff11	30/29	35.92	13.65	19.03
tgff12	36/50	82.73	11.66	80.11
tgff13	37/36	61.06	-0.41	46.95
tgff14	24/33	44.61	33.19	29.86
tgff15	40/63	50.96	36.43	40.70
tgff16	31/56	41.14	17.13	24.27
tgff17	29/56	75.77	55.69	51.87
tgff18	12/15	28.44	-2.26	0.00
tgff19	14/19	36.78	-19.60	0.00
tgff20	19/25	77.42	4.24	0.01
tgff21	70/99	91.60	87.38	78.67
tgff22	100/135	78.08	58.27	58.01
tgff23	84/151	85.76	62.56	11.08
tgff24	80/112	69.73	-8.42	21.89
tgff25	49/92	48.74	30.32	13.21
Avg. Energy Saving		**66.97**	**33.1**	**32.69**

assignment and ordering into a single chromosome and hence significantly reduces the search space and problem complexity. We employed two leading power management techniques (for homogeneous and heterogeneous embedded systems) in the fitness function of our genetic algorithm and integrated framework. We experimentally showed that this integrated framework can save on average about 18% more energy compared to a non-integrated technique using the same power management techniques. Our results showed that a scheduling algorithm (HGLS here) if employed in an integrated framework with a power management algorithm, is capable of improving itself with respect to energy efficiency. More broadly, we also showed that a task assignment and scheduling that generate a better parallel-time do not necessarily save more power, and hence, integrating task scheduling and slack distribution based power management methods is crucial for fully exploiting the energy-saving potential of an embedded multiprocessor implementation. We also evaluated our synthesis framework and showed that it produces solutions with higher energy efficiency than GMA + EE-GLSA, one of the best known techniques. Furthermore, we added a refinement phase to CASPER that utilizes the information (e.g. extended tasks' execution costs) obtained from the power-management step to re-schedule the tasks to explore further energy saving opportunities.

Chapter 7

Conclusions and Future Work

In this thesis, we have explored the system-level synthesis problem at various levels, starting from multiprocessor scheduling of fully-connected homogeneous embedded systems, to hardware-software co-synthesis of multi-mode, multi-task embedded systems on heterogeneous, arbitrarily-connected, multiple-PE embedded systems. Our proposed solutions are mainly based on evolutionary algorithm (EA) techniques. EAs, in addition to being flexible and naturally amenable to multiple-objective formulations, are applicable to complex and large search spaces. EAs are also scalable — in particular, EAs can trade off optimization times for solution quality, and one expects the solution quality to improve as EAs run for longer times (a characteristic that is not inherent in deterministic algorithms).

Hence, in our proposed methodology, to maintain a framework for fair comparison and more fully exploit the power of deterministic algorithms, we have applied randomization techniques to deterministic algorithms to make them also capable of exploring larger segments of the solution space. In our framework, all algorithms run for a limited time-budget. The choice of limited time-budget reflects the amount of time designers are willing to wait for a solution. What can be achieved by a given EA or randomized deterministic algorithm under such a time budget is a function of the available computational power relative to the complexity of the input instances. Hence, with increases in computational power some algorithms that prove inferior under a given time budget may emerge as superior techniques, and vice versa. Our experiments in the thesis reflect comparisons between different techniques based on the computational power available in medium-range personal computers and workstations at the present time.

However, our methodology of driving the optimization process based on a designer's time budget (e.g., rather than based on some fixed number of EA generations, which is standard practice with EAs), configuring EAs carefully with respect to the time budget, and considering randomized deterministic algorithms (rather than simply abandoning deterministic techniques when large time budgets are available) is applicable and useful regardless of the amount of available computational power. The in-depth development of this methodology, and the extensive experimentation demonstrating that under present technology, our methodology can be applied to yields significant improvements in synthesis quality are two major contributions of this thesis.

More specific summaries of the work presented in this thesis are as follows.

In Chapter 3 we investigated the problem of two-step multiprocessor scheduling for homogeneous systems. A two-step scheduling starts by clustering (i.e grouping of tasks into subsets that execute on the same processor and hence eliminate the heavy inter-tasks (processor) communication costs) tasks and ends by mapping of the clusters onto the target architecture. In this chapter, motivated by the availability of increased compile-time tolerance for embedded systems we developed a novel and natural genetic algorithm formulation, called CFA, for multiprocessor clustering. We also presented a randomization technique to be applied to leading deterministic state-of-art clustering techniques to make the comparisons (a time-intensive evolutionary algorithm vs. fast deterministic approaches) meaningful. We demonstrated the first comprehensive experimental setup for comparing one-step scheduling algorithms against two-step scheduling (clustering and cluster-scheduling or merging) algorithms. We experimentally showed that a pre-processing or clustering step that minimizes communication overhead can be very advantageous to multiprocessor scheduling and two-step algorithms provide better quality schedules. We also observed that the cluster-scheduling or merging results are very sensitive to the scheduling approach used in the clustering step and if two clustering use different scheduling techniques that result in different evaluation of their performance and later be employed in the same merging step, the results may not be consistent with what clustering evaluation had indicated. Hence, one better approach to compare the performance of the clustering algorithms may be to look at the number of clusters produced or cluster utilization in conjunction with parallel time. This could be a direction for future work.

In Chapter 4 we demonstrated a clustering-based scheduling algorithm for heterogeneous multiprocessor systems. Clustering as a pre-processing step has been shown to be an effective approach to reducing the search space in many multiprocessor system synthesis problems. However, in the context of heterogeneous systems the application of clustering is not straightforward since when the clustering is done, no information on the assignment and scheduling is available. Hence, the evaluation of clustering has to be done based on an estimation of the costs of the final target architecture. In this chapter we investigate various estimated values for evaluating the clustering. We also, investigated the effectiveness of clustering approach for the heterogeneous multiprocessor system. We demonstrated various approaches for mapping the clustering results to the final target architecture and through extensive experiments showed that clustering should always be evaluated w.r.t. the final mapping and not independently. One important conclusion of this work was the effectiveness of clustering and its application as a pre-processing step or technology-independent optimization step to be employed in system-level synthesis tools. Future works for clustering-based scheduling algorithms is extending the work to include interconnection-constrained networks.

In Chapter 5 we explored the problem of hardware-software co-synthesis of multi-mode, multi-task embedded systems. To our knowledge this is one of the first comprehensive works studying the most general formulation of the problem. Our proposed co-synthesis framework CHARMED makes no assumption on the hardware architecture or network topology, it is capable of handling multiple objective and multiple constraints simultaneously and efficiently, and is designed to handle every optimization goal (e.g. memory requirement or energy consumption) and architecture (e.g. dynamically reconfigurable hardware) individually and efficiently. Most optimization problems that arise in hardware-software co-design are highly complex, in this chapter we demonstrated how the design space can be greatly and efficiently reduced by applying a pre-processing (technology-independent) optimization step of clustering. CHARMED is further improved to handle dynamically reconfigurable hardware and provide a better framework for application of power management techniques such as DVS and optimization of systems memory requirements. One direction for future work is to add a refinement step that uses the possibly sub-optimal solutions generated by the allocation/assignment phase as the starting point for its local search. Looking into another method of parallelizing EAs that searches different subspaces of the search space in parallel and is less likely to get trapped in low-quality subspaces, could also be another direction for future work.

In Chapter 6 we presented a framework for static power management of embedded multiprocessor systems. A key distinguishing feature of our technique is that we perform task assignment, task ordering and scheduling and static power management together — existing power management algorithms assume a given application mapping and scheduling exists before applying the power management. One serious drawback to this assumption is that globally optimal voltage scheduling may not be generated. We believe that the integration of task assignment and ordering and voltage scheduling is essential since different assignments and orderings provide voltage schedulers with great flexibility and potential energy saving that can be achieved. Our results showed that a scheduling algorithm if employed in an integrated framework with a power management algorithm, is capable of improving itself with respect to energy efficiency. More broadly, we also showed that a task assignment and scheduling that generate a better parallel-time do not necessarily save more power, and hence, integrating task scheduling and slack distribution based power management methods is crucial for fully exploiting the energy-saving potential of an embedded multiprocessor implementation. We further demonstrated that a hybrid EA/local search algorithm can be very effective for solving complex optimization problems. We presented two hybridized algorithms, HCASPER+OO and HCASPER+AO, for the dynamic voltage scaling problem. OO and AO are both scheduling algorithm that use the newly increased execution costs of the tasks and find a new schedule. OO does not re-assigns tasks and only performs re-ordering based on new priorities arising from new execution costs. AO on the other hand does re-assign tasks and accepts an assignments that reduces the task's finish time. Such an assignment while helps the performance may lead to an increased energy consumption. Hence looking into defining new assignment policies that consider both time and energy is one direction for future work. Nevertheless HCASPER+AO does achieve significant energy saving.

[1] I. Ahmad and M. K. Dhodhi, "Multiprocessor Scheduling in a Genetic Paradigm," Parallel Computing, vol. 22, pp. 395-406, 1996.

[2] I. Ahmad and Y.-K. Kwok, "On Parallelizing the Multiprocessor Scheduling Problem," IEEE Transactions on Parallel and Distributed Systems, vol. 10, no. 4, pp. 414-432, April 1999.

[3] I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and W. Shu,"CASCH: A Tool for Computer Aided Scheduling," IEEE Concurrency, vol. 8, no. 4, pp. 21-33, 2000.

[4] A. Al-Maasarani, Priority-Based Scheduling and Evaluation of Precedence Graphs with Communication Times, M.S. Thesis, King Fahd University of Petroleum and Minerals, Saudi Arabia, 1993.

[5] M.A. Al-Mouhamed, "Lower Bound on the Number of Processors and Time for Scheduling Precedence Graphs with Communication Costs," IEEE Trans. Software Engineering, vol. 16, no. 12, pp. 1390-1401, Dec. 1990.

[6] J. Axelsson, "Architecture synthesis and partitioning of real-time systems: A comparison of three heuristic search strategies," in Proc. of Int. Workshop on Hardware/Software Co-Design, pp. 161-165, Mar. 1997.

[7] S. Azarm, "Multiobjective optimum design: Notes." $http : //www.glue.umd.edu/ azarm/optimum/notes/multi/multi.html$

[8] T. Back, U. Hammel, and H.-P. Schwefel, "Evolutionary computation: comments on the history and current state," IEEE Transactions on Evolutionary Computation, vol. 1, pp. 3-17, 1997.

[9] N. K. Bambha and S. S. Bhattacharyya, "System Synthesis for Optically-Connected, Multiprocessors on Chip," International Workshops on System on Chip for Real Time Processing, July 2002.

[10] N. K. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharyya, "Intermediate representations for design automation of multiprocessor DSP systems," Journal of Design Automation for Embedded Systems 7, no. 4, pp. 307323, 2002.

[11] N. Bambha and S. S. Bhattacharyya, "Joint application mapping/interconnect synthesis techniques for embedded chip-scale multiprocessors," IEEE Transactions on Parallel and Distributed Systems, 16(2):99-112, February 2005.

[12] S. Banerjee, T. Hamada, P.M. Chau, and R.D. Fellman, "Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems," IEEE Transactions on Signal Processing 43:8, pp. 1468-1484, June 1995.

[13] O. Beaumont, V. Boudet, and Y. Robert, "The iso-level scheduling heuristic for heterogeneous processors," In Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, 2002.

[14] Luca Benini, Giovanni De Micheli, "Powering Networks on Chip," International System Synthesis Symposium, Octo-ber 2001.

[15] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," Proceedings of the IEEE, vol. 79, pp. 12701282, Sep 1991.

[16] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," Int. Jour. Computer Simulation, vol. 4, pp. 155182, April 1994.

[17] Y. C. Chung and S. Ranka, "Application and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed-Memory Multiprocessors," In Proc. Supercomputing92, pp. 512-521, Nov. 1992.

[18] B. Cirou and E. Jeannot,"Triplet: a Clustering Scheduling Algorithm for Heterogeneous Systems," In IEEE ICPP International Workshop on Metacomputing Systems and Applications (MSA2001),Valencia, Spain, September 2001.

[19] F. Clover,"Tabu search part I," J. Comput., vol. 1, no. 3, pp. 190206, 1989.

[20] J. Y. Colin and P. Chretienne,"C.P.M. Scheduling with Small Computation Delays and Task Duplication," Operations Research, pp. 680-684, 1991.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms. McGraw-Hill Book Company, NY, 2001.

[22] R. C. Correa, A. Ferreira and P. Rebreyend, "Scheduling Multiprocessor Tasks with Genetic Algorithms," IEEE Tran. on Parallel and Distributed Systems, Vol. 0, 825-837, 1999.

[23] R. Cypher, "Message-Passing models for blocking and nonblocking communication," in DIMACS Workshop on Models, Architectures, and Technologies for Parallel Computation, Technical Report 93-87. September 1993.

[24] B. P. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-software co-synthesis of heterogeneous distributed embedded systems," IEEE Trans. on VLSI Systems, vol. 7, pp. 92104, Mar. 1999.

[25] B. Dave, "CRUSADE: Hardware/software co-synthesis of dynamically reconfigurable heterogeneous real-time distributed embedded systems," in Proc. of Design, Automation and Test in Europe Conf., pp. 97104, Mar. 1999.

[26] K. Deb, "Evolutionary algorithms for multi-criterion optimization in engineering design," In Proceedings of Evolutionary Algorithms in Engineering and Computer Science (EUROGEN99), 1999.

[27] K. Deb, A. Pratap, and T. Meyarivan, "Constrained test problems for multi-objective evolutionary optimization," First International Conference on Evolutionary Multi-Criterion Optimization, pp 284–298. Springer Verlag, 2001.

[28] K. A. De Jong, An analysis of the behavior of a class of genetic adaptive systems. Ph. D. thesis, University of Michigan. 1975.

[29] G. De Micheli, Synthesis and Optimization of Digital Circuits. McGraw-Hill, 1994.

[30] G. De Micheli and R. K. Gupta, "Hardware/software co-design," Proc. of IEEE, vol. 85, pp. 349365, Mar. 1997.

[31] T. L. Dean and M. Boddy. "An analysis of time-dependent planning". In Proceedings of the Seventh National Conference on Artificial Intelligence, pages 4954, 1988.

[32] R. P. Dick and N. K. Jha, "MOGAC: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems," in Proc. of Int. Conf. on Computer-Aided Design, pp. 522529, Nov. 1997.

[33] R. Dick, D. Rhodes, and W. Wolf, "TGFF: Task Graphs for Free," In Proc. Int. Workshop Hardware/Software Codesign, P.97-101, March 1998.

[34] R. P. Dick and N. K. Jha, "MOGAC: A multiobjective genetic algorithm for hardware-software co-synthesis of distributed embedded systems," IEEE Trans. on Computer-Aided Design, vol. 17, pp. 920935, Oct. 1998.

[35] R. P. Dick and N. K. Jha, "CORDS: Hardware-software co-synthesis of reconfigurable real-time distributed embedded systems," in Proc. of Int. Conf. on Computer-Aided Design, pp. 6268, Nov. 1998.

[36] R. P. Dick, PhD Thesis, 2001.

[37] Handouts of the Embedded System Design Automation course (ECE 510-2), Northwestern University, 2004.

[38] M. D. Dikaiakos, A. Rogers and K. Steiglitz, "A Comparison of Techniques used for Mapping Parallel Algorithms to Message-Passing Multiprocessors," Proc. of the Sixth IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas 1994.

[39] A. Dogan and F Ozguner, "LDBS: A duplication based scheduling algorithm for heterogeneous computing systems," In Proceedings of the International Conference on Parallel Processing (ICPP02), pp. 352, Vancouver, B.C., Canada, August 2002.

[40] P. Eles, K. Kuchcinski, Z. Peng, System Synthesis with VHDL, Kluwer Academic Publishers, 1997.

[41] H. El-Rewini and T. G. Lewis, "Scheduing Parallel Program Tasks onto Arbitray Target Machines, " J. Parallel and Distributed Computing, vol. 9, pp. 138-153, 1990.

[42] M. D. Ercegovac, "Heterogeneity in supercomputer architectures," Parallel Comput. 7, 367372, 1988.

[43] H. A. Eschenauer, J. Koski, , and A. Osyczka, Multicriteria Design Optimization : Procedures and Applications, Springer-Verlag, 1986.

[44] B. R. Fox and M. B. McMahon, “Genetic operators for sequencing problems,” in Foundations of Genetic Algorithms, G. Rawlins, Ed.: Morgan Kaufmann Publishers Inc., 1991.

[45] R. F. Freund and H. J. Siegel, “Heterogeneous processing,” IEEE Computer 26, 6 (June), 1317, 1993.

[46] D. D. Gajski, N. D. Dutt, Allen C-H. Wu, Steve Y-L. Lin, High-Level Synthesis: Introduction to Chip and System Design, Kluwer Academic Publishers, 1992.

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

[48] A. Garvey and V. Lesser, “Design-to-time real-time scheduling,” IEEE Transactions on Systems, Man and Cybernetics, 23(6):14911502, 1993.

[49] A. Gerasoulis and T. Yang, “A comparison of clustering heuristics for scheduling directed graphs on multiprocessors.” Journal of Parallel and Distributed Computing, Vol. 16, pp. 276-291, 1992.

[50] D. E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, 1989.

[51] F. Gruian and K. Kuchcinski, “LEneS: Task scheduling for low-energy systems using variable supply voltage processors,” *Proc. of Asia and South Pacific Design Automation Conference*, pp. 449-455, Jan. 2001.

[52] D. Harel,“Statecharts: A visual approach to complex systems,” Science of Computer Programming, vol. 8, pp. 231274, 1987.

[53] E. S. H. Hou, N. Ansari and H. Ren, “A Genetic Algorithm for Multiprocessor Scheduling,” IEEE Tran. on Parallel and Distributed Systems, Vol. 5, pp. 113-120, 1994.

[54] J. Hou and W. Wolf, “Process partitioning for distributed embedded systems,” in Proceedings of Int. Workshop on Hardware/Software Co-Design, pp. 7076, March 1996.

[55] S. Hua and G. Qu, “Power Minimization Techniques on Distributed Real-Time Systems by Global and Local Slack Management,” IEEE/ACM Asia South Pacific Design Automation Conference, January 2005.

[56] T. Ibaraki, “Combination with local search,” in Handbook of Genetic Algorithms, U.K. Ibaraki, T. Back, D. Fogel, and Z. Michalewicz, Eds. London: Inst. Physics, Oxford Univ. Press, pp. D3.2-1D3.2-5, 1997.

[57] Institute of Electrical and Electronic Engineers, Standard VHDL Language Reference Manual, IEEE 1076-1993, 1993.

[58] Institute of Electrical and Electronic Engineers, Standard Description Language Based on the Verilog Hardware Description Language, IEEE 1364-1995, 1995.

[59] M. Iverson, F. Ozguner, and G. Follen, "Paralelizing Existing Applications in a Distributed Heterogeneous Environment," Proc. Heterogeneous Computing Workshop, pp. 93-100, 1995.

[60] A. Jaszkiewicz,"Genetic local search for multi-objective combinatorial optimization," European Journal of Operational Research, vol. 137, no. 1, pp. 50-71, February 2002.

[61] B. Jeong, S. Yoo, S. Lee, and K. Choi, "Hardware-software cosynthesis for runtime incrementally reconfigurable FPGAs," in Proc. of Asia and South Pacific Design Automation Conf., pp. 169174, January 2000.

[62] N. K. Jha, "Low power system scheduling and synthesis," *Proc. of Int. Conf. on Computer Aided Design*, pp. 259-263, 2001.

[63] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation, part II, graph coloring and number partitioning," Operation Research, vol. 39, no. 3, pp. 378406, 1991.

[64] A. Kalavade and E. A. Lee, "A hardware-software codesign methodology for DSP applications," IEEE Design and Test of Computers, vol. 3, pp. 1628, September 1993.

[65] A. Kalavade and P. A. Subrahmanyam, Hardware/Software Partitiong for Multifunction Systems. IEEE Trans. on Computer-Aided Design, 17(9):819836, September 1998.

[66] K. Karplus and A. Strong, Digital synthesis of Plucked-string and drum timbers, Computer Music Journal, 1983.

[67] A. Khan, C. L. McCreary and M. S. Jones, "A comparison of multiprocessor scheduling heuristics," In Proceedings of the 1994 International Conference on Parallel Processing, vol. II, pp. 243-250, 1994.

[68] V. Kianzad and S. S. Bhattacharyya, "Multiprocessor clustering for embedded systems," Proc. of the European Conference on Parallel Computing, pp. 697-701, Manchester, United Kingdom, August 2001.

[69] V. Kianzad and S. S. Bhattacharyya, "CHARMED: A multiobjective cosynthesis framework for multi-mode embedded systems," In Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors, pp. 28-40, September 2004.

[70] V. Kianzad, S.S. Bhattacharyya, and G. Qu, "CASPER: An Integrated Energy-Driven Approach for Task Graph Scheduling on Distributed Embedded Systems," 16th IEEE International Conference on Application-specific Systems, Architectures and Processors, July 2005.

[71] V. Kianzad and S. S. Bhattacharyya, "Efficient techniques for clustering and scheduling onto embedded multiprocessors," IEEE Transactions on Parallel and Distributed Systems, 2006. To appear.

[72] S. J. Kim and J. C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," in Proc. of the Int. Conference on Parallel Processing, pp. 1-8, 1988.

[73] S. Kirkpatrick, C. Gelatt, and M. Vecchi,"Optimization by simulated annealing," Science, vol. 220, pp. 671680, 1983.

[74] J. D. Knowles and D. W. Corne,"M-PAES: A memetic algorithm for multiobjective optimization," in Proc. 2000 Congress on Evolutionary Computation, pp. 325-332, July 2000.

[75] N. Koziris, M. Romesis, P. Tsanakas and G. Papakonstantinou, "An Efficient Algorithm for the Physical Mapping of Clustered Task Graphs onto Multiprocessor Architectures," Proc. of 8th Euromicro Workshop on Parallel and Distributed Processing, (PDP2000), IEEE Press, pp. 406-413, Rhodes, Greece.

[76] N. Krasnogor, and J. Smith, "A Memetic Algorithm With Self-adaptive Local Search: TSP as a case study," in Proceedings of Genetic and Evolutionary Computation Conference, pp. 987-994, July 2000.

[77] N. Krasnogor and J. Smith, "A tutorial for competent memetic algorithms: model, taxonomy, and design issues," IEEE Transactions on Evolutionary Computation, Issue 5, pp. 474 - 488 October 2005.

[78] B. Kruatrachue and T.G. Lewis, "Duplication Scheduling Heuristics (DSH): A New Precedence Task Scheduler for Parallel Processor Systems," Technical Report, Oregon State University, Corvallis, OR 97331, 1987.

[79] Y. Kwok and I. Ahmad, "Dynamic critical path scheduling: an effective technique for allocating task graphs to multiprocessors," IEEE Tran. on Parallel and Distributed Systems, Vol. 7, pp. 506-521, 1996.

[80] Y. Kwok and I. Ahmad, "Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using A Parallel Genetic Algorithm," Journal of Parallel and Distributed Computing, 1997.

[81] Y. Kwok and I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," Journal of Parallel and Distributed Computing, vol. 59, no. 3, pp. 381-422, December 1999.

[82] Y. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," ACM Computing Surveys, vol. 31, no. 4, pp. 406-471, December 1999.

[83] Y. Kwok and I. Ahmad, “ Link Contention-Constrained Scheduling and Mapping of Tasks and Messages to a Network of Heterogeneous Processors ,” Cluster Computing, vol. 3, no. 2, pp. 113-124, 2000.

[84] A. La Rosa, L. Lavagno, C. Passerone, “Hardware/Software Design Space Exploration for a Reconfigurable Processor,” In Proceeding of Design, Automation and Test in Europe Conference and Exhibition (DATE’03), March 2003.

[85] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” Proceedings of the IEEE, vol. 75, pp. 12351245, Sep 1987.

[86] R. Lepre and D. Trystram, “A new clustering algorithm for scheduling task graphs with large communication delays,” International Parallel and Distributed Processing Symposium, 2002.

[87] T. Lewis and H. El-Rewini, “Parallax: A tool for parallel program scheduling,” IEEE Parallel and Distributed Technology, vol. 1, no. 2, pp. 62-72, May 1993.

[88] G. Liao, G. R. Gao, E. R. Altman, and V. K. Agarwal, “A comparative study of DSP multiprocessor list scheduling heuristics,” in Proceedings of the Hawaii Internationl Conference on System Sciences, 1994.

[89] P. Lieverse, E. F. Deprettere, A. C. J. Kienhuis and E. A. De Kock, “A clustering approach to explore grain-sizes in the definition of processing elements in dataflow architectures.” Journal of VLSI Signal Processing, Vol. 22, pp. 9-20, August 1999.

[90] J. Liou and M. A. Palis, “A Comparison of General Approaches to Multiprocessor Scheduling,” 11th International Parallel Processing Symposium (IPPS), Geneva, Switzerland, pp. 152-156, April 1997.

[91] J. Luo and N. K. Jha, “Power-profile driven variable voltage scaling for heterogeneous distributed real-time embedded systems,” *Int. Conf. on VLSI Design*, Jan. 2003.

[92] M. Maheswaran, T. D. Braun, and H. J. Siegel, “Heterogeneous Distributed Computing,”Encyclopedia of Electrical and Electronics Engineering, John Wiley & Sons, New York, NY, Vol. 8, pp. 679-690, 1999.

[93] N. Mehdiratta, and K. Ghose, “A bottom- up approach to task scheduling on distributed memory multiprocessor,” In Proceedings of the International Conference on Parallel Processing, CRC Press, Inc., Boca Raton, FL, pp. 151154, 1994.

[94] B. Mei, P. Schaumont, and S. Vernalde, “A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems,” In proceeding of the 11th ProRISC workshop on Circuits, Systems and Signal Processing, Netherlands, Nov. 2000.

[95] D. Menasc and V. Almeida, "Cost-performance analysis of heterogeneity in supercomputer architectures," In Proceedings on Supercomputing 90, J. L. Martin, Ed. IEEE Computer Society Press, Los Alamitos, CA, pp. 169-177, 1990.

[96] B. Meyer, Object-oriented software construction. 2nd ed., Prentice Hall, 1997.

[97] P. Marwedel and G. Goossens, Code Generation for Embedded Processors. Kluwer Academic Publishers, 1995.

[98] P. Merz and B. Freisleben, "Genetic Local Search for the TSP: New Results", In Proceedings of the 1997 IEEE International Conference on Evolutionary Computation, Piscataway, NJ, pp. 159-164, 1997.

[99] C. McCreary and H. Gill, "Automatic Determination of Grain Size for Efficient Parallel Processing," Comm. ACM, vol. 32, pp. 1073-1078, Sept. 1989.

[100] C. L. McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle, "A comparison of heuristics for scheduling DAGS on multiprocessors," in Proc. of the Int. Parallel Processing Symp., pp. 446-451, 1994.

[101] R. Mishra, N. Rastogi, D. Zhu, D. Mossé, and R. Melhem, "Energy aware scheduling for distributed real-time systems, " *Int. Parallel and Distributed Processing Symp.*, pp. 243-248, April 2003.

[102] J. N. Morse, Reducing the size of the nondominated set: Pruning by clustering. Computers and Operations Research, Vol. 7, No. 1-2, pp. 55-66, 1980.

[103] T. Murata, "Petri nets: Properties, analysis and applications," Proceedings of the IEEE, vol. 77, pp. 541580, April 1989.

[104] A. K. Nand, D. Degroot, D. L. Stenger, "Scheduling directed task graphs on multiprocessor using simulated annealing," in Proc. of the Int. Conference on Distributed Computer Systems, pp. 20-27, 1992.

[105] J. Noguera, and R. M. Badia, "A HW/SW partitioning algorithm for dynamically reconfigurable architectures," in proceedings of Design Autmation and Test In Europe Conference, pp. 729-734, March 2001.

[106] H. Oh and S. Ha, "A Static Scheduling Heuristicfor Heterogeneous Processors," Second International Euro-Par Conference Proceedings, Volume II, Lyon, France, August 1996.

[107] H. Oh and S. Ha, "Hardware-software co-synthesis technique based on heterogeneous multiprocessor scheduling," in Proc. of Int. Workshop on Hardware/ Software Co-Design, pp. 1831878, May 1999.

[108] H. Oh and S. Ha, "Hardware-software co-synthesis of multi-mode multi-task embedded systems with real-time constraints," in Proc. of the Int. symposium on Hardware/software codesign, pp. 133-138, May 2002.

[109] A. Osyczka, Multicriteria optimization for engineering design. In J. S. Gero Ed. Design Optimization, pp. 193-227. Academic Press, 1985.

[110] V. Pareto, Cours D'Economie Politique, Volume I and II. F. Rouge, Lausanne.

[111] S. Prakash and A. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," Journal of Parallel and Distributed Computing, vol. 16, pp. 338351, Dec. 1992.

[112] H. Printz, Automatic Mapping of Large Signal Processing Systems to a Parallel Machine. Ph.D. Thesis, school of computer Science, Carnegie Mellon University, May 1991.

[113] A. Radulescu, A. J. C. van Gemund, and H.-X. Lin. "LLB: A fast and effective scheduling algorithm for distributed memory systems." In Proc. Int. Parallel Processing Symp. and Symp. on Parallel and Distributed Processing, pp. 525-530, 1999.

[114] A. Radulescu and A. J. C. van Gemund, "Fast and effective task scheduling in heterogeneous systems," In Proceeding of Heterogeneous Computing Workshop, 2000.

[115] A. Raghunathan, N. K. Jha, and S. Dey, High-level Power Analysis and Optimization. Kluwer Academic Publishers, 1997.

[116] M. Rinehart, V. Kianzad, and S. S. Bhattacharyya, "A modular genetic algorithm for scheduling task graphs," Technical Report UMIACS-TR-2003-66, Institute for Advanced Computer Studies, University of Maryland at College Park, June 2003. Also Computer Science Technical Report CS-TR-4497.

[117] Alberto Sangiovanni-Vincentelli, "The Tides of EDA," IEEE Design and Test of Computers, vol. 20, no. 6, pp. 59-75, November/December, 2003.

[118] A. Sangiovanni-Vincentelli, "System-level design: a strategic investment for the future of the electronic industry." VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT), pp. 1 - 5, April 2005.

[119] V. Sarkar, Partitioning and Scheduling Parallel Programs for Multiprocessors. MIT Press, 1989.

[120] M. Schmitz, B. Al-Hashimi, and P. Eles, "Energy-efficient mapping and scheduling for DVS enabled distributed embedded systems," *Design, Automation and Test in Europe Conference*, March 2002.

[121] M. Schmitz, B. Al-Hashimi, and P. Eles, "A Co-Design Methodology for Energy-Efficient Multi-Mode Embedded Systems with Consideration of Mode Execution Probabilities," Proc. Design, Automation and Test in Europe, 2003.

[122] M. Schmitz, B. Al-Hashimi, and P. Eles, "Iterative Schedule Optimisation for Voltage scalable Distributed Embedded Systems," *ACM Trans. on Embedded Computing Systems*, vol. 3, pp. 182-217, 2004.

[123] L. Shang and N. K. Jha, "Hardware-software Co-synthesis of Low Power Real-Time Distributed Embedded Systems with dynamically reconfigurable fpgas," in Proc. of Int. Conf. on VLSI Design, pp. 345352, January 2002.

[124] B. Shirazi, H. Chen, and J. Marquis, "Comparative Study of Task Duplication Static Scheduling versus Clustering and Non-clustering Techniques," Concurrency: Practice and Experience, vol. 7, no.5, pp. 371-390, August 1995.

[125] P. Shroff, D. W. Watson, N. S. Flann, and R. Freund, "Genetic Simulated Annealing for Scheduling Date-Dependent Tasks in Heterogeneous Environments, " In Proceedings of Heterogeneous Computing Workshop, pp. 98-104, 1996.

[126] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping Computer-vision-related Tasks onto Reconfigurable Parallel-processing Systems," IEEE Computer 25, 2, pp. 54-64, Feb. 1992.

[127] H. J. Siegel, H. G. Dietz, and J. K. Antonio, "Software support for heterogeneous computing," ACM Comput. Surv. 28, 1, pp. 237-239, 1996.

[128] G. C. Sih, "Multiprocessor Scheduling to Account for Interprocessor Communication", Ph.D. Dissertation, ERL, University of California, Berkeley, CA 94720, April 22, 1991.

[129] G. C. Sih and E. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures." IEEE Tran. on Parallel and Distributed systems, Vol. 4, No. 2, 1993.

[130] H. Singh and A. Youssef, "Mapping and Scheduling Heterogeneous Task Graphs Using Genetic Algorithms," Proc. Heterogeneous Computing Workshop, pp. 86-97, 1996.

[131] D. Spencer, J. Kepner, and D. Martinez, "Evaluation of advanced optoelectronic interconnect technology," MIT Lincoln Laboratory August 1999.

[132] S. Sriram and S. S. Bhattacharyya, Embedded Multiprocessors: scheduling and Synchronization. Inc. Marcel Dekker, 2000.

[133] D. Sylvester and H. Kaul, "Power-driven challenges in nanometer design," *IEEE Design and Test of Computers*, pp. 12-21, Nov. 2001.

[134] R. Szymanek and K. Kuchcinski, "Partial Task Assignment of Task Graphs under Heterogeneous Resource Constraints," In Proceeding of 40th Design Automation Conference (DAC'03) June 2003.

[135] J. Teich, T. Blickle and L. Thiele, "An Evolutionary approach to system-level Synthesis," Workshops on Hardware/Software Codesign, March 1997.

[136] The SystemC community, The Open SystemC initiative. http://www.systemc.org/.

[137] H. Topcuoglu, S. Hariri and M.-Y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," IEEE Transactions on Parallel and Distributed Systems 13(3): 260-274, 2002.

[138] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task scheduling algorithms for heterogeneous processors," In Proceedings of the 8th Heterogeneous Computing Workshop, pp. 3-14, San Juan, Puerto Rico, April 1999. IEEE Computer Society Press.

[139] P. Wang and W. Korfhage, "Process Scheduling Using Genetic Algorithms," IEEE Symposium on Parallel and Distributed Processing, pp. 638-641, 1995.

[140] L. Wang, H. J. Siegel, and V. P. Roychowdhury, "A Genetic-Algorith Based Approach for Task matching and Scheduling in Heterogeneous Computing Environments," In Proceedings of Heterogeneous Computing Workshop, 1996.

[141] W. Wolf, Computers as Components: Principles of Embedded Computing System Design, Morgan Kaufman Publishers, 2001.

[142] J. Wong, F. Koushanfar, S. Megerian, M. Potkonjak, "Probabilistic Constructive Optimization Techniques," IEEE Transactions of CAD, vol. 23, no. 6, pp. 859- 868, June 2004.

[143] M.-Y. Wu and D. D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," IEEE Trans. Parallel and Distributed Systems, vol. 1, no. 3, pp. 330-343, July 1990.

[144] Y. Xie and W. Wolf, "Co-synthesis with custom ASICs," in Proc. of Asia and South Pacific Design Automation Conf., pp. 129133, January 2000.

[145] "Xilinx part information." http://www.xilinx.com/partinfo/.

[146] T. Yang and A. Gerasoulis, "PYRROS: States scheduling and code generation for message passing multiprocessors," In Proceedings of 6th ACM Int. Conference on Supercomputing, 1992.

[147] T. Yang and A. Gerasoulis, "List Scheduling with and without Communication Delays," Parallel Computing, vol. 19, pp. 1321-1344, 1993.

[148] T. Yang. *Scheduling and Code Generation for Parallel Architectures*. Ph.D. thesis, Dept. of CS, Rutgers University, May 1993.

[149] T. Yang and A. Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors," IEEE Tran. on Parallel and Distributed Systems, Vol. 5, pp. 951-967, 1994.

[150] S. Zilberstein and S. Russell, "Optimal Composition of real-time systems," Artificial Intelligence, 82(1-2):181-213, 1996.

[151] E. Zitzler, Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications. Swiss Federal Institute of Technology (ETH) Zurich. TIK-Schriftenreihe Nr. 30, Diss ETH No. 13398, Shaker Verlag, Germany, ISBN 3-8265-6831-1, December 1999.

[152] E. Zitzler and L. Thiele, Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. IEEE Transactions on Evolutionary Computation, 3(4), pp. 257-271, November 1999.

[153] E. Zitzler, J. Teich, and S. S. Bhattacharyya, Optimized software synthesis for DSP using randomization techniques. Technical report, Computer Engineering and Communication Networks Laboratory, Swiss Federal Institute of Technology, Zurich, July 1999.

[154] E. Zitzler, M. Laumanns, L. Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization," Evolutionary Methods for Design, Optimisation, and Control, pp. 95-100, 2002.

[155] A. Y. Zomaya, C. Ward and B. Macey, "Scheduling for parallel processor systems: comparative studies and performance issues," IEEE Tran. on Parallel and Distributed Systems, Vol. 10, pp. 795-812, 1999.