Energy- Aware Task Scheduling

by

Waleed M. S. Alsalih

A thesis submitted to the School of Computing

in conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

March, 2005

Copyright © Waleed M. S. Alsalih, 2005
NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.
“In the name of ALLAH, The Most Gracious, The Most Merciful”

To My Parents, My Wife

and My Son
Abstract

Since the emergence of mobile computing, reducing energy consumption of battery-operated computing devices has become a very active research area. The widespread popularity of mobile computing devices, such as laptops, handheld devices and cell phones, motivates this research area. Several hardware based techniques have been proposed; this has led to more energy-efficient systems. Nevertheless, it is presumed in the literature of energy-aware design that software based techniques have the potential to reduce energy demand and contribute to solve the problem.

This research introduces the generic *wireless cooperative system*, which involves a set of heterogeneous computing devices connected via a wireless medium; in this cooperative system, the computational tasks of a resource-limited device are offloaded to one or more nearby devices in a way that conserves energy and improves performance. In such a distributed computing environment, the assignment of computational tasks to different devices and the order of their execution play the vital role in energy conservation and performance improvement, and this is the focus of this research.
In this research, we define new energy-aware task allocation and task scheduling problems that have never yet been addressed, formulate proper task and processor models that fit the new problems and propose heuristic-based algorithms for both problems. To the best of our knowledge, this research is a pioneering work in defining and treating new generation of scheduling problems.

The proposed novel algorithms schedule a set of computational tasks, which may have dependencies and communication, into a set of heterogeneous processors in a way that minimizes both the total consumed energy and the makespan (i.e., the time required to have all tasks completed). Experiments show that significant improvements can be achieved by using the proposed algorithms.

Keywords: Energy-aware design, task scheduling, computation offloading, mobile computing, ubiquitous computing, power management.
Acknowledgements

First of all, I thank ALLAH (The God) for his mercy, help and guidance, without which this document would not be made possible.

It was a great honor for me to be supervised by Dr. Selim G. Akl and Dr. Hossam S. Hassanein. Besides their significant contribution to this document, they influenced my development as a member of the research community of the field. They were absolutely successful in making me enjoy every minute I spend in research activities.

I owe my parents any success I make in my life. Their encouragement and support is a key factor in any achievement I have ever made. I am also indebted to my lovely wife, Nisreen, for her continuous encouragement and patience during the course of my M.Sc. program. I would also like to thank my little son, Khaled, for his patience with me and for the happiness and motivation I always find from his smile.

I would like to thank all members of the parallel computation group, the telecommunication research lab and the School of Computing at Queen’s for their support and friendship.

Special thanks to the Ministry of Higher Education in Saudi Arabia for their financial support.
Content

1 Introduction ........................................... 1
   1.1 Energy management approaches ....................... 2
   1.2 Wireless cooperative system ............................ 4
   1.3 Contributions of this thesis ........................... 5
   1.4 Road map of the document ............................. 7

2 Background ............................................. 8
   2.1 Classical scheduling algorithms ....................... 8
      2.1.1 General scheduling problem ....................... 9
      2.1.2 NP-Completeness ................................. 12
      2.1.3 Optimal scheduling algorithms .................... 13
      2.1.4 NP-Completeness of the scheduling problem .... 15
      2.1.5 List scheduling .................................. 16
      2.1.6 Task allocation ................................ 19
   2.2 Energy-aware scheduling algorithm ................... 21
   2.3 Remote execution platforms .......................... 22
   2.4 Summary ........................................... 23

3 Energy-aware task allocation .......................... 25
   3.1 Problem formulation .................................. 26
   3.2 Classical allocation algorithms ........................ 27
   3.3 Energy-aware Task Allocation algorithm (E.T.A.) ....... 29
List of figures

1.1 The general architecture of wireless cooperative system ........... 5
2.1 Task graph ............................................................................. 11
2.2 In-forest tree structured task graph ...................................... 14
3.1 An example of a processor model and a task model .............. 31
3.2 The task allocation steps of Fig. 3.1 models ......................... 32
3.3 Comparison between E.T.A. results and the optimal ones generated using Stone’s algorithm in category I .................. 35
3.4 Comparison between E.T.A. results and those of Abraham and Davidson approximation algorithm (2-Approx.) in category II ... 35
3.5 Comparison between E.T.A. results and the optimal ones in category III ................................................................. 36
4.1 An example of a processor model, a task model and the corresponding estimated levels ............................................. 45
4.2 Illustration example for the task assignment process of figure 5.2 example with $\alpha = 1$ token/$\mu s$ and $\beta = 1$ token/nJ ............... 46
4.3 Exact levels ........................................................................... 51
4.4 Execution ordering .................................................................. 52
4.5 Makespan comparison ............................................................... 59
4.6 Energy consumption comparison ............................................ 59
4.7 Total cost comparison .............................................................. 59
4.8 An example of a processor model and a task model ............. 60
5.1 MANET routes ...................................................................... 67
List of tables

2.1 Complexity comparison of different scheduling problems ........ 17
3.1 Comparison between the E.T.A. results and those of three other algorithms .................................................. 37
4.1 Comparison of different methods ........................................ 58
List of acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>E.T.A.</td>
<td>Energy-Aware Task Allocation</td>
</tr>
<tr>
<td>E.T.S.</td>
<td>Energy-Aware Task Scheduling</td>
</tr>
<tr>
<td>LS</td>
<td>List Scheduling</td>
</tr>
<tr>
<td>MANET</td>
<td>Mobile wireless Ad hoc NETwork</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>RCP</td>
<td>Ready task Critical Path</td>
</tr>
<tr>
<td>SP</td>
<td>Single Processor scheduling</td>
</tr>
<tr>
<td>TGFF</td>
<td>Task Graphs For Free</td>
</tr>
<tr>
<td>WLAN</td>
<td>Wireless Local Area Network</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Mobility of computing devices requires small, light components. This requirement constrains the size and, hence, energy capacity of batteries used in mobile computing devices. As a result, battery energy has become a critical resource for such devices. Having batteries of high energy density would alleviate the problem but, unfortunately, battery technologies advance at a too modest speed. The density of the best battery technologies, such as Lithium-Ion, is still too low to meet the needs of energy-hungry modern applications [1]. Therefore, there has been a growing realization that attacking the problem by decreasing the energy demand of different components is more promising [2,3].

Several energy management techniques at different levels of application, operating system and hardware have been proposed. But even though these techniques helped alleviating the problem to some extent, the problem still remains and there is a consensus that more work is needed.
Computation offloading is a new technique by which a resource-limited device can defer the execution of a computational task, either partially or totally, to a nearby, more capable server. Most of the research in this area assumes a restricted case of one mobile client and one stationary server. Moreover, the literature lacks a precise optimization method that makes the most of all available resources by choosing the best execution plan that conserves energy and improves performance.

This thesis sets forth the generic idea of a wireless cooperative system, which involves two or more heterogeneous computing devices. Recent advances in wireless communication architectures and computation offloading platforms have made it possible to build such a cooperative environment on the fly and to let resource-limited devices make use of nearby energy and processing resources. However, to make the most of available resources, we believe that a new generation of scheduling methods is needed, ones that consider energy as a valuable resource to be conserved. The development of such scheduling methods is the primary objective of this research.

This chapter begins in Section 1.1 with an overview of previous energy management approaches. It then introduces the vision of wireless cooperative systems and energy-aware task scheduling in Section 1.2. The contributions of this thesis are summarized in Section 1.3. Finally, Section 1.4 provides a road map for the rest of the document.

1.1 Energy management approaches

In a mobile computing device, there are several components that consume battery energy. The display, hard disk, network card and CPU are the main energy consumers in such a device. One way to tackle the problem is to design hardware components that consume less energy. Turning off any idle component is one of the most common
approaches [4]. Some processors support several voltage levels which allows higher levels of the system to scale the voltage in such a way that offers a compromise between energy conservation and processing performance [5,6]. These hardware based approaches have led to more energy-efficient hardware components. However, that has not been enough to meet the energy-hungry modern applications and, therefore, more approaches at higher levels of the system are needed. Hence, special techniques aimed at the application, compiler and operating system levels have been proposed.

Energy-aware adaptation is an application- and system-level approach that maintains a dynamic balancing of quality and energy conservation [2]. This method, however, causes a significant degradation in the mobile application’s fidelity [7]. Energy-aware code generation is a compiler-level approach that exploits the voltage scaling technique to generate energy conscious code [8].

Computation offloading, sometimes called remote execution, is a cooperative approach by which a resource-limited computing device defers computation to a nearby, wall-powered, stationary server [3,9]. Not only does computation offloading help conserve energy of mobile computing devices, but it also has a potential for performance improvement, especially if the processing capability of the server is much greater than that of the client. Previous work in this direction focused on monitoring different resources, predicting the cost of local execution and that of a remote one and deciding between local and remote execution.

The computation offloading approach makes it feasible to have a set of heterogeneous computing devices working in a cooperative manner to tackle the resource limitation problem of mobile computing devices. However, this general cooperative approach has
not yet been properly addressed. Furthermore, there has not been a precise optimization method that efficiently utilizes the energy and processing resources of both client and server. The purpose of this thesis is to fill this gap by introducing the generic wireless cooperative system and the optimization schemes that make the most out of it.

1.2 Wireless cooperative system

Besides the fact that most mobile computing devices are equipped with wireless connection facilities, several computing platforms have been designed specially for mobile computing devices to support process migration and remote execution [3,9,10]. This makes it feasible for heterogeneous mobile and stationary computing devices to work in a cooperative manner. Furthermore, recent advances in Mobile Ad hoc wireless NETworks (MANETs) facilitate constructing such a system on the fly [11].

A wireless cooperative system consists of a set of heterogeneous computing devices that may range from small, resource-limited, mobile devices to stationary, wall-powered, powerful servers. Whenever a resource-limited computing device in such a system has a set of tasks (or subtasks\textsuperscript{1}) to be executed (which may have dependencies and communication requirements among themselves), it uses all available resources in nearby computing devices. A mobile computing device may use resources of one or more nearby stationary servers and/or even the resources of one or more other nearby mobile devices. A personal digital assistant (PDA), for example, may use the resources of a nearby laptop. Figure 1.1 shows the general architecture of the wireless cooperative system.

A wireless cooperative system involves a set of consumers, i.e., mobile applications (tasks), and a set of resources, i.e., energy and processing power. Therefore, finding a

---

\textsuperscript{1} A task may be partitioned to a set of subtasks. However, the task partitioning problem is out of the scope of this work. Rather, it is assumed that the task is already efficiently partitioned.
proper schedule plays the main role for energy conservation and performance improvement. A schedule includes a task assignment that shows for each task which processor is to execute it and an execution order that shows the starting time of each task.

Besides introducing the generic wireless cooperative systems, this thesis formulates new energy-aware allocation and scheduling problems and proposes novel heuristic-based algorithms to solve them.

1.3 Contributions of this thesis

This document proposes the following thesis:
Due to recent advances in wireless communication technologies and remote execution platforms, it is now feasible to construct a wireless cooperative system of heterogeneous computing devices. Energy-aware task scheduling can bring about an efficient use of such a cooperative system to conserve energy and improve performance of mobile computing devices. However, energy-aware task scheduling has not been studied before.

The main contributions of this work are:

- The formal definition of novel energy-aware task allocation problem. The primary objective in this problem is to find a proper assignment of tasks to heterogeneous computing devices, such that the total amount of energy consumed by task execution plus the total amount of energy consumed by task interaction is minimized.

- The formal definition of novel energy-aware task scheduling problem. The primary objective in this problem is to find a proper assignment of tasks to heterogeneous computing devices, and a proper task execution order, such that a cost function of the total amount of consumed energy and the makespan is minimized.

- Designing novel heuristic-based energy-aware algorithms for the defined problems. These algorithms show, by experiments, the ability to find near-optimal solutions.

We believe that this work pioneers new research directions in the field of energy-aware task scheduling towards energy conservative, high performance wireless cooperative systems.
1.4 Road map of the document

This document is organized as follows. Chapter 2 offers a review of classical scheduling algorithms and remote execution platforms. Chapter 3 formulates the allocation problem (of finding a proper assignment of tasks that minimizes energy consumption), and presents a heuristic-based greedy algorithm for obtaining a near-optimal solution to this problem. Chapter 4 states the more general problem of finding a schedule that minimizes a cost function of energy consumption and makespan (i.e., the time required to have all tasks completed), and presents a heuristic-based algorithm to solve it approximately. Chapter 5 concludes this work by summarizing the main contribution, discussing the importance and viability of our approach and suggesting some future research directions.
Chapter 2

Background

The main contributions of this thesis are the novel energy-aware scheduling algorithms for wireless cooperative systems. This chapter presents the required background for this work, namely, classical scheduling algorithms and remote execution platforms. Section 2.1 introduces a general overview of the classical scheduling problems and algorithms, as well as the NP-Completeness of the scheduling problem. Section 2.2 discusses a recently proposed energy-aware scheduling algorithm. Remote execution platforms are explored in Section 2.3. A brief summary concludes this chapter in Section 2.4.

2.1 Classical scheduling algorithms

Scheduling is a classical, common problem in many fields including computer science. It has been defined in different ways and in different contexts. Early work was influenced by the job sequencing problem in industry. Manufacturing processes in industry usually involve a number of steps that transform raw materials to the final product. The job
sequencing problem is to find a proper sequence of these steps that satisfies certain optimization criteria [12].

A task scheduling problem emerges whenever there are valuable resources and consumers which are serviced by these resources under some constraints. A scheduling algorithm is concerned with finding a proper consumption plan, i.e., the assignment of consumers to resources and the consumption order of each resource, which is preferred according to some measures.

In a computing environment, there are several valuable resources: processors, memory space, communication links, battery energy, and so on. There are consumers serviced by these resources, namely, computational tasks to be executed. Finally, there are several performance measures: throughput of the system, utilization of the system, makespan, consumed energy, and so on.

Most of the research in multi-processor scheduling targets time-based performance measures (e.g., makespan, total execution time, communication time, and so on). However, some scheduling schemes can be generalized to target other measures. Even though the main concern of this work is energy measures, it is worth to study time-based scheduling algorithms in order to derive generic techniques that can be used for energy-aware scheduling. Furthermore, this is a preliminary step towards an exhaustive scheme that targets both energy-based measures and time-based ones like the one presented in Chapter 4.

2.1.1 General scheduling problem

In order to define a scheduling problem, one should show:
1. The task model which is a graphical and/or mathematical representation that shows all characteristics of the tasks to be executed,

2. The processor model which is a graphical and/or mathematical representation that shows all characteristics of all processors available on the system, and

3. The cost function to be minimized by the scheduling algorithm.

Using this methodology which was introduced in [12], a general scheduling problem can be defined as follows.

- **Task model** \( T \):
  1. Tasks:
     \[ T = \{ t_1, t_2, \ldots, t_n \} \] is a set of tasks to be executed where \( n \) is the number of tasks.
  2. Interaction among tasks:
     \( Data \) is an \( n \times n \) matrix, where \( Data(i,j) \) is the amount of data units (e.g., bytes) sent from \( t_i \) to \( t_j \).
  3. Precedence relations among tasks:
     \( < \) is a partial order defined on \( T \), where \( t_i < t_j \) means that \( t_j \) can not start execution before \( t_i \) has completed its execution and \( Data(i,j) \) data units have been received by the processor on which \( t_j \) is to be executed.
  4. Execution time of each task on each processor:
     \( ExecT \) is an \( n \times m \) matrix, where \( m \) is the number of processors and \( ExecT(i,j) \) is the execution time of \( t_i \) when executed on \( p_j \).

**Task graphs** are usually used to represent precedence relations and interaction among tasks. A task graph is a Directed Acyclic Graph (DAG) in which there is a node for every task and a directed edge for every precedence relation. An edge between two
nodes carries a weight equal to the amount of interaction between the corresponding tasks. Figure 2.1 shows an example of a task graph.

- **Processor model $\mathcal{P}$:**

  1. Processors:

    $\mathcal{P} = \{p_1, p_2, ..., p_m\}$ is a set of processors available on the system.

  2. Estimated communication delay among processors:

    $CommT$ is an $m \times m$ matrix, where $CommT(i,j)$ is the amount of time required for one unit of data to travel from $p_i$ to $p_j$.

- **Cost function $C$:**

  $C$ is a cost function to be used for measuring the goodness of a schedule. A schedule with minimal cost is preferred. Two common cost functions are:

  $C = makespan$ (this is the time by which all tasks have completed execution) and

  $C = communication\_time + execution\_time$. ($communication\_time$ is the total amount of time spent for interaction among tasks assigned to different processors, and $execution\_time$ is the total amount of time spent for the execution of all tasks).

We can now formulate the general scheduling problem as follows.
Given a task model $T$ and a processor model $P$, find a schedule $S$ that maps each task to a processor and determines the starting time of each task in such a way that minimizes a given cost function $C$, such that:

- There is no execution overlap among tasks which are assigned to the same processor.
- All precedence relations, if there are any, are satisfied.

### 2.1.2 NP-Completeness

To classify computational problems in terms of hardness and time complexity, three interesting classes have been defined: P, NP and NP-Complete. These three classes applies directly to decision problems, of which the answer is “yes” or “no”. However, any problem can be casted as a related decision problem. The class P includes all decision problems which can be solved by a polynomial-time algorithm (i.e., there exists such an algorithm that solves the problem and runs in time $O(n^k)$ for some constant $k$, where $n$ is the size of the input). The class NP includes those decision problems which are “verifiable” in a polynomial time (i.e., if some solution is given with a certificate, it is possible to verify whether it is correct or not in a polynomial time). It is clear that if a problem $Q$ is in P, it is also in NP, since it is possible to verify any solution for $Q$ by solving $Q$ in a polynomial time. However, one of the most controversial questions in computer science is whether or not $P=NP$.

To clearly define the set NP-Complete, we need to define the concept of “Reduction”. We say a decision problem $Q_I$ is reducible to another decision problem $Q_2$ if and only if any instance $A$ of $Q_I$ can be transformed in a polynomial time to an instance $B$ of $Q_2$, and
that $A$ and $B$ have the same answer (i.e., “yes” or “no”). If a problem $Q_1$ is reducible to another problem $Q_2$, we can say that $Q_1$ is not more than a polynomial factor harder than $Q_2$.

A problem $Q$ is in NP-Complete if and only if:

1. $Q$ is in NP, and
2. $L$ is reducible to $Q$, for all problems $L$ that are in NP.

If the second condition applies but the first one does not necessarily do, $Q$ is called NP-hard. NP-Complete problems have the following properties:

1. Because they are all in NP, all NP-complete problems are equivalent to one another (by definition they are all reducible to one another),
2. No polynomial-time algorithm is known for an NP-complete problem (all known algorithms for them run in time exponential in the size of the problems), and
3. Finding a polynomial time algorithm for any one of the NP-complete problems would settle the P=NP question (as all NP problems would be solvable in polynomial time).


2.1.3 Optimal scheduling algorithms

The task scheduling problem in its general form and several restricted forms of it are NP-Complete [12]. Only few scheduling problems have polynomial optimal algorithms. Moreover, these optimal algorithms work only under unrealistic assumptions. Hu in [14] presented an optimal scheduling algorithm which works only when the task graph is a tree. A task graph is a tree if it is either an in-forest, i.e., each node has at most one
successor, or an out-forest, i.e., each node has at most one predecessor. Figure 2.2 shows an in-forest tree-structured task graph.

This algorithm assumes that the task and processor models described in Section 2.1.1 with the following assumptions:

- \( \text{Exec}T(i,j) = \text{constant}, \ 1 \leq i \leq n, \ 1 \leq j \leq m \) (identical processors and identical task execution times).

- \( \text{Data}(i,j) = 0, \ 1 \leq i, j \leq n \) (no communication among tasks).

The cost function to be minimized is:

\[ C = \text{makespan}. \]

We define the following terms:

- A \textit{terminal task} is a task with no successor. In in-forest trees, there is only one terminal task. Task 5 in Fig. 2.2 is a terminal task.

- The \textit{length of a path} in a task graph is the sum of the execution times of all tasks along this path. Note that the communication is not included in the length of a path.
• The level of a task in a task graph is the length of the longest path from the corresponding node to any exit node. Level of task 4 in Fig. 2.2 is 2 (if all tasks have a unit execution time).

• A ready task is a task with no predecessor or a task of which all predecessors have completed execution.

The following simple algorithm is used for the in-forest case; however, it can be used for the case of out-forest with minor modifications.

1. Calculate the level of each task.
2. Whenever a processor is available, assign it the ready task with the largest level.

This is a linear time algorithm that has been proven to generate the minimal makespan schedule. For the same problem but with arbitrary task execution times and identical processors (i.e., different tasks may have different execution times but any one task has the same execution time on different processors), Kaufman presented an optimal algorithm that is similar to the one by Hu [15].

For the case of identical processors, identical task execution times and interval ordered task graph (i.e., task graph nodes can be mapped into intervals on the real line such that two nodes are connected by an edge if and only if the corresponding intervals do not overlap), an optimal algorithm was reported in [16]. Several optimal algorithms were proposed for the case of identical processors, identical task execution times, arbitrary task graph, and two-processor system [12,17].

2.1.4 NP-Completeness of the scheduling problem
Even without considering communication delay, the general scheduling problem and most of its restricted versions have been proven to be NP-Complete [12]. The definition of NP-Completeness applies only to decision problems (i.e., yes/no questions). In order to prove that a problem \( Q_2 \) is NP-complete, one needs to cast \( Q_2 \) as a decision problem \( Q_1 \) and prove that \( Q_1 \) is NP-Complete.

The general scheduling problem is defined as follows. Using the task and processor models described in Section 2.1.1, and under the following assumptions:

- \( \text{Exec}T(i,j) = \text{constant}, 1 \leq i \leq n, 1 \leq j \leq m \) (identical processors and identical task execution times).
- \( \text{Data}(i,j) = 0, 1 \leq i, j \leq n \) (no communication among tasks).

We ask the following question: For some time limit \( k \), is there a schedule \( S \) of length less than or equal to \( k \)?

This problem has been proven to be NP-Complete in [18]. Table 2.1 summarizes the results in [12] and [19], and shows a complexity comparison of some scheduling problems under the following two assumptions:

- For any task \( t_i \), \( \text{Exec}T(i,j) = \text{constant}, 1 \leq j \leq m \) (identical processors).
- \( \text{Comm}T(i,j) = \text{constant}, 1 \leq i, j \leq m \) (identical transmission rate over all links).

### 2.1.5 List scheduling

Optimal scheduling algorithms work only under impractical assumptions, such as tree-structured task graphs, interval-ordered task graphs, identical execution times and two-processor systems. Moreover, communication delays are neglected by almost all optimal scheduling algorithms. However, communication delay is a main factor that strongly
<table>
<thead>
<tr>
<th>Task graph</th>
<th>Task execution time</th>
<th>Number of processors</th>
<th>Communication time</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>identical</td>
<td>arbitrary</td>
<td>0</td>
<td>$O(n)$ ($n$ is the number of tasks)</td>
</tr>
<tr>
<td>interval order</td>
<td>identical</td>
<td>arbitrary</td>
<td>0</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>arbitrary</td>
<td>identical</td>
<td>2</td>
<td>0</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>interval order</td>
<td>identical = $c$</td>
<td>arbitrary</td>
<td>identical = $c$</td>
<td>$O(mn^2)$ ($m$ is the number of processors)</td>
</tr>
<tr>
<td>arbitrary</td>
<td>identical</td>
<td>arbitrary</td>
<td>0</td>
<td>NP-Complete</td>
</tr>
<tr>
<td>arbitrary</td>
<td>1 or 2 time units</td>
<td>2</td>
<td>0</td>
<td>NP-Complete</td>
</tr>
<tr>
<td>interval order</td>
<td>arbitrary</td>
<td>2</td>
<td>0</td>
<td>NP-Complete</td>
</tr>
<tr>
<td>arbitrary</td>
<td>arbitrary</td>
<td>arbitrary</td>
<td>0</td>
<td>NP-Complete</td>
</tr>
</tbody>
</table>

Table 2.1: Complexity comparison of different scheduling problems.

Influences the performance of recent computing systems. In the absence of efficient exact algorithms, researchers in the field have turned to heuristic-based approaches. A heuristic is an intuitive rule that has the potential to lead to near-optimal solutions. List scheduling is a priority-based heuristic in which each task is assigned a priority and whenever more than one task contend for execution, the one with higher priority is selected. Hu’s algorithm, described in Section 2.1.3, was the first list scheduling algorithm. The general list scheduling algorithm can be expressed as follows.

1. Each task is assigned a priority.
2. A ready queue is initialized to have those tasks with no predecessor (the ready queue is a priority queue).
3. While the ready queue is not empty:
   3.1 Get a task $t$ from the front of the queue (the one with the highest priority).
   3.2 Select a processor $p$ to run $t$.
   3.3 Find the starting time of $t$ on $p$.  

17
3.4 Insert in the ready queue all tasks of which all predecessors have been already executed.

Two important decisions to be made here are:

1. In step 1, what criteria to be used for priority?
2. In step 3.2, how to select a processor to run a task?

Both decisions should be made carefully in such a way that "logically" and "intuitively" will improve the quality of the generated schedule (i.e., how close it is to the optimal solution).

In the case of identical processors, using the level of a task as its priority is presumed in the literature to be an effective technique to minimize the makespan, especially when communication is neglected. It is also obvious that a processor that becomes available first should be selected to run a particular ready task.

In [14], Hu presents a level-based list scheduling algorithm that generates optimal schedules when the task graph is a tree, processors are identical and all tasks have the same execution time. Coffman and Graham in [17] show that list scheduling is optimal for scheduling tasks on an arbitrary task graph with identical execution times on only two processors. With an arbitrary number of processors, an arbitrary task graph, and arbitrary task execution times, Graham in [20] derives a precise bound on the makespan of level-based list scheduling. He proves that the makespan of a level-based schedule would never be more than 2 - (1/m) times the makespan of the optimal schedule where m is the number of processors. In the same paper, Graham also shows that this bound can be improved to (4 / 3) – (1 / 3m) if there is no precedence relation. However, when the communication delay is significant, the quality of pure level-based list scheduling is lowered because the communication delay is excluded while computing the level of each
task. It is even quite hard to include the communication delay in the level computation because that depends heavily on the task assignment. This problem is called the *level number* problem for list scheduling [12]. Nevertheless, some approaches have been proposed to adapt level-based list scheduling so that it takes into consideration the communication delay. Task duplication [12], for example, duplicates the execution of the same task on different processors to eliminate the delay caused by communication originated from that task. It is, however, not clear yet how to find a low complexity algorithm with duplication [21]. Another approach is to divide the scheduling process into two stages: task assignment and execution ordering. In the first stage, where communication is excluded, a pure level-based list scheduling is used to assign tasks to different processors. Once the task assignment is decided, the communication delay becomes explicit and can be included in the level computation. The next stage uses a more accurate level computation (that includes communication delays) to reorder the task execution locally on each processor. Even though no quality bounds have been derived for the general problem, extensive experimental studies and quality analyses have shown that two-stage list scheduling is an effective heuristic when communication delay is significant [21].

2.1.6 Task allocation

It is common in the literature to use the term task allocation and task scheduling interchangeably. However, task allocation is a special case of the general task scheduling problem, which is described in Section 2.1.1, in which there is no precedence relation and the cost function to be minimized is:

\[ C = \text{communication\_time} + \text{execution\_time}, \]  

where:
• *communication_time* is the total amount of time spent for interaction among tasks assigned to different processors.

• *execution_time* is the total amount of time spent for the execution of all tasks.

Therefore, there is no need to specify the execution order.

This problem has been proven to be NP-Complete in its general form [12]. However, some optimal algorithms have been proposed for some restricted versions of the problem and some heuristic-based algorithms have been proposed for the more general versions of the problem. The graph theoretic approach has been used intensively to solve this problem. In this approach, the allocation problem is reduced to the problem of finding a *minimum cut in a flow network* [22]. This approach was first introduced by Stone in [23] where he proposed an algorithm that finds the optimal task allocation for two-processor systems. Stone showed that a flow network can be constructed in such a way that there is a one-to-one correspondence between all *cuts* in that network and all possible task allocations, and the weight of a cut equals exactly the cost of the corresponding task allocation. Therefore, finding the minimum weight cut is equivalent to finding the minimum cost task allocation. Stone’s algorithm can be described as follows.

1. Construct a graph $G$ that includes the following nodes and edges:
   1.1. Two distinguished nodes representing the processors $p_1$ and $p_2$.
   1.2. A node for every task in $T$.
   1.3. An edge between every two interacting tasks $t_i$ and $t_j$ with a weight equal to $Data(i,j)$.
   1.4. An edge between every task $t_i$ to $p_1$ with a weight equal to $ExecT(i,2)$.
   1.5. An edge between every task $t_i$ to $p_2$ with a weight equal to $ExecT(i,1)$.
2. Assuming that $p_1$ is the sink and $p_2$ is the source, find a minimum cut on $G$ using the well known max-flow min-cut algorithm [22].

In a more general variation of the problem with an arbitrary number of processors and an identical transmission rate over all communication links, Lo in [24] proposed a heuristic-based algorithm which extends Stone’s approach. An approximation algorithm for the same problem was proposed by Abraham and Davidson; their algorithm is a modified version of Lo’s algorithm and is guaranteed to generate results that are no worse than twice the optimal [12].

2.2 Energy-aware scheduling algorithm

Even though scheduling is a classical research area, most of the work in this area focuses on minimizing time-based cost functions. Therefore, traditional scheduling algorithms are not suitable for mobile computing systems in which a valuable resource is the battery energy. Some uni-processor energy-aware scheduling algorithms have been developed to make an efficient use of hardware based energy management techniques. An energy-aware scheduling algorithm, proposed in [25], orders task execution such that different components of a mobile computing device can have longer idle periods to be shut down. An algorithm, developed in [6], adjusts the CPU speed and voltage level in such a way that increases the number of instructions executed per energy unit. However, multi-processor energy-aware scheduling has not yet received enough attention. In [26] the authors formulate a multi-processor energy-aware scheduling problem for certain architectures of embedded systems and propose a heuristic to solve it. Their algorithm is a level-based list scheduling algorithm that schedules time constrained computational tasks and communication transactions on a Network-on-Chip architecture and aims at
meeting task deadlines with the minimum energy consumption. This may be the only multi-processor energy-aware scheduling algorithm available in the literature at the time of this writing. However, the scheduling problem in [26] is different from ours in terms of the underlying architecture, characteristics of the tasks to be scheduled and cost function to be minimized.

2.3 Remote execution platforms

Computation offloading has been introduced as an energy conservation approach for mobile computing devices. A mobile computing device can defer the execution of an energy consuming computational task to a nearby, more capable server. Recent advances in wireless communication technologies have made such an approach feasible. However, that brought about the need for remote execution platforms that enable the process of migrating computational tasks to a remote server and predict the resource usage of each task on both the local device and the remote one.

A significant amount of research effort has been devoted to the development of remote execution platforms and most of the previous work focused on monitoring different resources of both client and server, predicting the resource usage of local execution and that of remote one, and deciding between local and remote execution.

Remote Processing Framework (RPF) [10] is a client-server-based remote execution platform. On the client’s side, RPF maintains a database storing resource usage statistics of different tasks and decides whether to use local or remote execution. On the server’s side, when remote execution is preferable, RPF performs the migrated task and returns the results back to the client. The decision making process in RPF is simple and achieved by comparing the local and remote executions and choosing the best of the two. Spectra
[9] is a remote execution component of an operating system designed for mobile and pervasive computing devices. Spectra monitors different resources and predicts the tasks usage of these resources. To monitor batteries and predict energy usage of tasks, it exploits the advent of smart batteries that report information about energy levels and power drain. To make an execution decision, Spectra evaluates the local execution and a set of remote execution alternatives and chooses the best of these. Remote Processing Platform (RPP) [3] is another platform that mainly targets handheld devices. It makes use of smart batteries, profiling and algorithmic complexities of computational tasks to predict their energy consumption and maintains a log database that stores information about different computational tasks. The decision making process in RPP is similar to that of Spectra and RPF.

2.4 Summary

The focus of this thesis is energy-aware task scheduling over computation offloading (remote execution) platforms. This chapter presents the preliminary background of both task scheduling and remote execution platforms.

Task scheduling is a traditional problem to which a significant amount of research effort has been devoted. However, most of the proposed scheduling algorithms target time-based performance metrics which makes them not suitable for mobile computing systems where the battery energy is perhaps the most valuable resource. Nevertheless, some elegant scheduling and allocation techniques are worth to be looked into as a first step towards obtaining generic techniques for energy-aware scheduling.

Some energy-aware scheduling algorithms have been proposed for uni-processor systems; they make use of hardware based energy management techniques. However, the
problem of multi-processor energy-aware scheduling has not yet received enough attention.

This chapter also discusses some remote execution platforms that enable cooperative work among heterogeneous computing device and, therefore, brings the need for multi-processor energy-aware scheduling.
Chapter 3

Energy-aware task allocation

The main deterrent from enabling ubiquitous mobile computing is the battery energy limitation. A wireless cooperative system reduces the effect of such a limitation by distributing the computational tasks of a mobile computing device amongst all other devices in an energy-efficient way. This chapter looks into the problem of finding the proper task allocation that minimizes the total consumed energy. Energy is consumed by the execution of a task and by the interaction between two tasks assigned to different computing devices (processors).

Section 3.1 gives a precise formulation for this problem. Section 3.2 presents some classical task allocation algorithms that can be used to solve the problem when special conditions apply. In Section 3.3, we introduce our greedy algorithm that can be used in the general case. Experimental results that show the effectiveness of the proposed algorithm are given in Section 3.4.
3.1 Problem formulation

This section formulates an energy-aware task allocation problem. We define this allocation problem using the following task model, processor model and cost function.

- **Task model \( T \):**
  1. Tasks:
     \[ T = \{ t_1, t_2, ..., t_n \} \] is a set of tasks to be executed where \( n \) is the number of tasks.
  2. Interaction among tasks:
     \( Data \) is an \( n \times n \) matrix, where \( Data(i,j) \) is the amount of data units (e.g., bytes) sent from \( t_i \) to \( t_j \).
  3. Execution energy consumption of each task on each processor:
     \( ExecE \) is an \( n \times m \) matrix, where \( ExecE(i,j) \) is the amount of energy consumed by \( t_i \) when executed on \( p_j \).

- **Processor model \( P \):**
  1. Processors:
     \[ P = \{ p_1, p_2, ..., p_m \} \] is a set of processors available in the system.
  2. Estimated communication energy consumption among processors:
     \( CommE \) is an \( m \times m \) matrix, where \( CommE(i,j) \) is the amount of energy consumed by one unit of data to travel from \( p_i \) to \( p_j \). This includes the energy consumed by the source device, the destination device and any other device involved along the path between the source and destination.

   It is assumed that \( CommE(i,i) = 0, \ 1 \leq i \leq m \), i.e., the cost of communication from a processor to itself is zero. It is also assumed that the task model and the processor model stay unchanged until all tasks complete execution (this is known as static allocation).
Cost function:

\[ C = \text{communication\_energy} + \text{execution\_energy}, \text{ where:} \]

- \text{communication\_energy} is the total amount of energy consumed by interaction among tasks assigned to different processors.
- \text{execution\_energy} is the total amount of energy consumed by the execution of all tasks.

The problem can be formulated now as follows:

Given a task model \( T \) and a processor model \( P \), find a task assignment \( S \) that maps each task to a processor in such a way that minimizes the following cost function:

\[ C = \text{communication\_energy} + \text{execution\_energy} \]

### 3.2 Classical allocation algorithms

The classical task allocation problem, which is described in Section 2.1.6, is the problem of finding the proper task assignment that minimizes the sum of the total execution time and the total communication time. However, in this energy-aware allocation problem, the objective is to minimize the sum of the total execution energy and the total communication energy. Since both time and energy can be seen as a cost metric to be minimized, the two problems are equivalent, i.e., a time-based classical allocation algorithm can be used as an energy-aware allocation algorithm. However, classical allocation algorithms assume that the communication links are identical, i.e., the communication cost between any two processors is the same. This is not a valid assumption in a wireless mobile computing environment, which imposes a high degree of heterogeneity among different communication links. In a wireless communication environment, the communication links have different costs and may be asymmetric, i.e.,
the cost of a link from a processor \( p_i \) to another processor \( p_j \) may be different from that of
the link from \( p_j \) to \( p_i \).

Stone’s allocation algorithm, which is described in Section 2.1.6, was proven to be optimal in terms of minimizing the sum of the total execution time and the total communication time when there are exactly two processors and the communication links in both directions are identical (i.e., they have the same transmission rate). Therefore, if there are only two processors and the communication links between them are identical in terms of energy consumption, Stone’s algorithm can be used to find the minimum-energy task assignment. An energy-aware version of Stone’s algorithm can be described as follows.

1. Construct a graph \( G \) that includes the following nodes and edges:
   1.1 Two distinguished nodes representing the processors \( p_1 \) and \( p_2 \).
   1.2 A node for every task in \( T \).
   1.3 An edge between every two interacting tasks \( t_i \) and \( t_j \) with a weight equal to \( Data(i,j) \cdot CommE(1,2) \).
   1.4 An edge between every task \( t_i \) to \( p_1 \) with a weight equals \( ExecE(i,2) \).
   1.5 An edge between every task \( t_i \) to \( p_2 \) with a weight equals \( ExecE(i,1) \).

2. Assuming that \( p_1 \) is the sink and \( p_2 \) is the source, find a minimum cut on \( G \) using the well known \textit{max-flow min-cut} algorithm [23].

In a more general case with an arbitrary number of processors and identical communication links, the algorithm of Abraham and Davidson, which is introduced in Section 2.1.6, was proven to give a solution no worse than twice the optimal. Therefore,
when this assumption applies in terms of energy cost, the same algorithm can be used as an energy-aware approximation algorithm.

However, there is no classical task allocation algorithm that considers an arbitrary number of processors and arbitrary communication costs, which is the normal situation in a mobile wireless communication environment.

3.3 Energy-aware Task Allocation algorithm (E.T.A.)

This section proposes a heuristic-based, centralized\(^2\), greedy algorithm for the problem defined in Section 3.1. This algorithm tries to minimize the total consumed energy by assigning each task to the processor that executes it with the minimal energy. When an assignment decision is to be made for a task \(t\), this algorithm considers the energy consumption of \(t\)'s execution on different processors and the energy consumption of \(t\)'s interaction with tasks that have been already assigned. However, \(t\)'s interaction with tasks that have not yet been assigned is not considered because the cost of such interaction depends heavily on the assignment of those tasks and is impossible to predict.

In the following algorithm, the overall complexity of each step is shown in curly brackets. In our complexity analysis, we assume that DAG is used to represent the interaction between tasks and is represented using a data structure that has the following property: If a task \(t\) has \(j\) predecessors and \(k\) successors, you can find all predecessors of \(t\) in time \(O(j)\) and all successors of \(t\) in time \(O(k)\). A representation which has two adjacency matrices, one for incoming edges and one for outgoing edges, has this

\(^2\) What is meant by centralized here is to have the algorithm being executed in one device (most likely in the device where tasks are initiated). However, there is no central scheduler.
property. It is also assumed that there are \( m \) processors, \( n \) tasks and \( e \) task interaction links (\( e \) is the number of edges in the DAG). The E.T.A. algorithm follows.

1. While there are still unallocated tasks
   1.1 Pick any unallocated task \( t_i \) {Since we have \( n \) tasks, the overall complexity of this step is \( O(n) \)}
   1.2 Assign \( t_i \) to a processor \( p_k \) such that \( \text{ExecE}(i,j) \leq \text{ExecE}(i,k), 1 \leq k \leq m. \) {Each task will be checked against all processors to find the best one and, therefore, the overall complexity of this step is \( O(nm) \)}
   1.3 For any unallocated task \( t_c \) that receives data from \( t_i \) {Since we have \( e \) interaction links, the overall complexity of this step is \( O(e) \)}
      1.3.1 For any processor \( p_k \) {Since we have \( e \) interaction links and \( m \) processors, this step will be done \( em \) times in time \( O(em) \)}
         1.3.1.1 \( \text{ExecE}(c,k) = \text{ExecE}(c,k) + \text{Data} (i,c) \ast \text{CommE}(j,k). \)
          {This step is similar to step 1.3.1 and has a complexity of \( O(em) \)}
   1.4 For any unallocated task \( t_c \) that sends data to \( t_i \) {Since we have \( e \) interaction links, the overall complexity of this step is \( O(e) \)}
      1.4.1 For any processor \( p_k \) {Since we have \( e \) interaction links and \( m \) processors, this step will be done \( em \) times in time \( O(em) \)}
         1.4.1.1 \( \text{ExecE}(c,k) = \text{ExecE}(c,k) + \text{Data} (c,i) \ast \text{CommE}(k,j). \)
          {This step is similar to step 1.4.1 and has a complexity of \( O(em) \)}

Therefore, by summing up the complexity of all steps, the total complexity of this algorithm is \( O(nm+em). \) However, if \( m \) is constant, the complexity can be abbreviated to \( O(e+n). \)

The idea of adding the communication cost to the execution cost was used first by Lo in her min-cut-based allocation algorithm [24]. However, her algorithm assumes identical
communication costs between different processors and minimizes the sum of the total
execution time and the total communication time. Figure 3.1 shows an example of simple
task and processor models consisting of 2 processors and 3 tasks. Figure 3.2 shows the
task allocation steps carried by E.T.A. if tasks are assigned in the following order: \( t_3, t_1 \)
and then \( t_2 \). In step 1, since \( t_3 \) consumes 2 energy units on \( p_1 \) and 6 energy units on \( p_2 \), \( t_3 \) is
assigned to \( p_1 \). Once \( t_3 \) is assigned to \( p_1 \), if \( t_1 \) is assigned to \( p_2 \), it will consume one more

![Diagram](image)

\[ \begin{array}{c|cc}
   & p_1 & p_2 \\
\hline
   t_1 & 8 & 2 \\
   t_2 & 3 & 1 \\
   t_3 & 2 & 6 \\
\end{array} \]

\[ \begin{array}{c|cc}
   & p_1 & p_2 \\
\hline
   p_1 & 0 & 2 \\
   p_2 & 1 & 0 \\
\end{array} \]

Figure 3.1: An example of a processor model and a task model.

energy unit due to the interaction with \( t_3 \). Therefore, \( \text{ExecE}(1,2) \) is incremented. In the
same way, \( \text{ExecE}(2,2) \) is increased by 16. In step 2, since \( t_1 \) consumes 8 energy units on
\( p_1 \) and 3 energy units on \( p_2 \), \( t_1 \) is assigned to \( p_2 \). Once \( t_1 \) is assigned to \( p_2 \), if \( t_2 \) is assigned
to \( p_1 \), it will consume 3 more energy units due to the interaction with \( t_1 \). Therefore,
\( \text{ExecE}(2,1) \) is increased by 3. Finally, in step 3, \( t_2 \) is assigned to \( p_1 \) because it consumes 6
energy units in \( p_1 \) and 17 energy units in \( p_2 \). Each step in Fig. 3.2 shows the assignment
decision and the corresponding modified \( \text{ExecE} \) table (the most recent changes on \( \text{ExecE} \)
are shown in bold). The optimal allocation, however, is to assign all tasks to \( p_2 \) and that

31
has a total consumed energy of 9 energy units only. The total consumed energy of the generated allocation is 11 energy units.

<table>
<thead>
<tr>
<th>Step 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
</tr>
<tr>
<td>t3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>8</td>
</tr>
<tr>
<td>t2</td>
<td>3</td>
</tr>
<tr>
<td>t3</td>
<td>2</td>
</tr>
</tbody>
</table>

Processor allocation

$\text{ExecE}$

<table>
<thead>
<tr>
<th>Step 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
</tr>
<tr>
<td>t3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>8</td>
</tr>
<tr>
<td>t2</td>
<td>6</td>
</tr>
<tr>
<td>t3</td>
<td>2</td>
</tr>
</tbody>
</table>

Processor allocation

$\text{ExecE}$

<table>
<thead>
<tr>
<th>Step 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
</tr>
<tr>
<td>t2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>8</td>
</tr>
<tr>
<td>t2</td>
<td>6</td>
</tr>
<tr>
<td>t3</td>
<td>2</td>
</tr>
</tbody>
</table>

Processor allocation

$\text{ExecE}$

Figure 3.2: The task allocation steps of Fig. 3.1 models.
3.4 Experimental results

We conducted several experiments to investigate the performance of the E.T.A. algorithm. We used Task Graphs For Free (TGFF) [27] to generate random task and processor models (Benchmarks). However, since there is no allocation algorithm which considers an arbitrary number of processors and heterogeneous communication links, we were forced to generate different benchmark categories, each of which has some restrictions so that we can compare E.T.A. with another allocation algorithm that works only under these restrictions. In addition to the restricted categories, we generated a general category of unrestricted benchmarks, which have an arbitrary number of processors and heterogeneous communication links, and compared the results of E.T.A. against the optimal allocation using a non-polynomial algorithm. However, benchmarks of this category have such a small number of tasks and processors that it is possible to get the optimal solution in a reasonable time using a non-polynomial algorithm.

Category I has 200 benchmarks, each of which has about 150 tasks and about 300 inter-task communication links. The average execution energy consumption is 1000 nJ. The average number of data units on each communication link is 100 bytes. We also assumed a two-processor system with identical communication links ($CommE(1,2) = CommE(2,1) = 1 \text{nJ/byte}$). Since there are only two processors and the communication links between them are identical, Stone’s allocation algorithm can be used to generate the optimal allocation in polynomial time, and we can compare the results of E.T.A. against the optimal ones. Even though it seems curious to compare an optimal algorithm with a non-optimal one, we believe that the performance of E.T.A. in this restricted case can be considered as an indication to its performance in the general case where the optimal
algorithm is not applicable. It turns out that E.T. A. results were within 2.46 \% of the optimal solution in all generated benchmarks. E.T.A. results were also optimal in 4\% of the generated benchmarks. Figure 3.3 shows the results of both algorithms in 20 randomly selected benchmarks.

Category II has 200 benchmarks, each of which has about 150 tasks and about 300 inter-task communication links. The average execution energy consumption is 1000 \( nJ \). The average number of data units on each communication link is 100 bytes. We also assumed a ten-processor system with identical communication links (\( \text{CommE}(i,j) = 1 \) \( nJ/\text{byte} \), \( i \neq j \)). Since the communication links are identical, the algorithm of Abraham and Davidson (Min-Cut) can be applied to generate results, which are no worse than twice the optimal (i.e., 2-Approximation), and we can compare the results of E.T.A. against those of the approximation algorithm. It turns out that the results of E.T.A. consume, on average, 12.75\% less energy compared to those generated by the approximation algorithm. E.T.A. results are worse than those of the approximation algorithm in less than 6\% of the tested benchmarks, and the difference in those benchmarks was never more than 5.9\% of the approximation algorithm results. In about 25\% of the tested benchmarks, E.T.A. results are better than those of the approximation algorithm by less than 10\%. In about 69\% of the benchmarks, E.T.A. results are better than those of the approximation algorithm by more than 10\% but less than 31\%. Figure 3.4 shows the results of both algorithms in 20 randomly selected benchmarks.

Category III has 200 benchmarks, each of which has a maximum of 15 tasks and about 30 inter-task communication links. The average execution energy consumption is 1000 \( nJ \). The average number of data units on each communication link is 100 bytes. With
each benchmark, we generated a random processor model of 4 processors with heterogeneous communication links. The average communication energy consumption between any two processors is \(5 \, \text{nJ/byte}\). We used a non-polynomial algorithm to find the optimal allocation and compare it with the one generated by E.T.A. algorithm. It turns out that E.T.A. results consume, on average, 5.21% more energy compared to the optimal one. In 0.5% of the generated benchmarks, E.T.A. allocations are worse than twice the

![Figure 3.3: Comparison between E.T.A. results and the optimal ones generated using Stone’s algorithm in category I.](image)

![Figure 3.4: Comparison between E.T.A. results and those of Abraham and Davidson approximation algorithm (2-Approx.) in category II.](image)
optimal (in only one of the tested benchmarks, the E.T.A. result consumes 106% more energy compared to the optimal one). In 3.5% of the tested benchmarks, E.T.A. results consume more than 25% (but less than 100%) more energy compared to the optimal one. In 26.5% of the tested benchmarks, E.T.A. results consume more than 5% (but less than 25%) more energy compared to the optimal one. And in 69.5% of the tested benchmarks, E.T.A. results are within 5% of the optimal one. E.T.A. results are optimal in 52.5% of the tested benchmarks. Figure 3.5 shows a comparison between E.T.A. results and the optimal ones in 20 randomly selected benchmarks. Table 3.1 shows a statistical comparison between E.T.A. results and those of the other algorithms.

3.5 Summary

This chapter introduced an energy-aware task allocation algorithm that assigns a set of tasks, which have interactions among themselves, to a set of heterogeneous processors connected through heterogeneous communication links, such that the total consumed
<table>
<thead>
<tr>
<th>Comparison with Stone's optimal algorithm</th>
<th>Comparison with Abraham and Davidson approximation algorithm</th>
<th>Comparison with the exponential optimal algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.T.A. results are within 2.46% of the optimal solution in all tested benchmarks.</td>
<td>The results of E.T.A. consume, on average, 12.75% less energy compared to those generated by the approximation algorithm.</td>
<td>E.T.A. results consume, on average, 5.21% more energy compared to the optimal one.</td>
</tr>
<tr>
<td>E.T.A. results are optimal in 4% of the tested benchmarks.</td>
<td>E.T.A. results are worse than those of the approximation algorithm in less than 6% of the tested benchmarks.</td>
<td>In 0.5% of the tested benchmarks, E.T.A. results are worse than twice the optimal.</td>
</tr>
<tr>
<td>In about 25% of the tested benchmarks, E.T.A. results are better than those of the approximation algorithm by less than 10%.</td>
<td>In 26.5% of the tested benchmarks, E.T.A. results consume more than 5% (but less than 25%) more energy compared to the optimal one.</td>
<td>In 69.5% of the tested benchmarks, E.T.A. results are within 5% of the optimal one.</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison between the E.T.A. results and those of three other algorithms.

energy is minimized. This problem and, hence, the proposed algorithm are novel not only because of considering energy which has recently become a critical system resource, but also because of considering an arbitrary number of processors and heterogeneous communication links. We have experimentally compared the proposed algorithm with other classical algorithms and with the optimal solutions whenever obtainable. These experiments show that the results of the proposed algorithm are near-optimal in most of the tested benchmarks.
The proposed algorithm is suitable for many mobile computing environments where battery energy is the most critical resource of the system. However, the processing performance and timing requirements are not considered in this algorithm, and this is the subject of the next chapter.
Chapter 4

Energy-aware task scheduling

Computation offloading introduces an opportunity for energy conservation. However, that may come at the cost of performance degradation due to the overhead of data transfer among different devices. From another point of view, however, computation offloading has the potential to improve performance; especially when the processing power of nearby devices is much more than that of the client. Moreover, the availability of more than one processor gives the opportunity for parallel processing, which can improve the performance significantly.

The focus of this chapter is finding a proper execution plan that reconciles the two possibly conflicting objectives of energy conservation and performance improvement. The first section introduces the new scheduling problem as a generalized version of the allocation problem discussed in the previous chapter. In Section 4.2, a precise formulation of the problem is given. Section 4.3 proposes a scheduling algorithm to solve the problem. Experimental results are shown in Section 4.4. Section 4.5 discusses the
worst case performance of the proposed algorithm. Finally, a brief summary concludes this chapter in section 4.6.

4.1 Introduction

The algorithm described in Chapter 3 finds a task allocation such that the total consumed energy is minimized regardless of the time needed to have all tasks completed. Therefore, it may lead to an energy conservative, low performance system. In a mobile computing device, both energy conservation and performance may be of significant importance. However, their relative importance is not static and depends on some factors, such as current energy level, type of tasks being executed, and so on.

This chapter formulates a more general scheduling problem that aims at both conserving energy and improving performance. The cost function to be minimized here consists of an energy metric, a performance metric and dynamic system-defined importance weights for each metric. The allocation problem defined in Chapter 3 can be seen as a special case of this scheduling problem.

The novelty of this scheduling problem stems from having two different cost metrics to be optimized, namely, time (makespan) and energy. These cost metrics are of different natures; while parallelism helps in minimizing the makespan, it does not have a clear effect on energy consumption and while the execution order affects the makespan, it has no effect on energy consumption.

4.2 Problem formulation
This section formulates a novel, non-preemptive, centralized scheduling problem. We define this scheduling problem using the following task model, processor model and cost function.

- **Task model $T$:**
  1. Tasks:
     
     $T = \{t_1, t_2, \ldots, t_n\}$ is a set of tasks to be executed where $n$ is the number of tasks.
  2. Interaction among tasks:
     
     *Data* is an $n \times n$ matrix, where $Data(i,j)$ is the amount of data units (e.g., bytes) sent from $t_i$ to $t_j$.
  3. Precedence relation among tasks:
     
     $<$ is a partial order defined on $T$, where $t_i < t_j$ means that $t_j$ can not start execution before $t_i$ has completed its execution and $Data(i,j)$ data units have been received by the processor on which $t_j$ is to be executed.
  4. Execution time of each task on each processor:

     *ExecT* is an $n \times m$ matrix, where $m$ is the number of processors and *ExecT*(i,j) is the execution time of $t_i$ when executed on $p_j$.
  5. Execution energy consumption of each task on each processor:

     *ExecE* is an $n \times m$ matrix, where *ExecE*(i,j) is the amount of energy consumed by $t_i$ when executed on $p_j$.

- **Processor model $P$:**
  1. Processors:

     $P = \{p_1, p_2, \ldots, p_m\}$ is a set of processors available on the system.
  2. Estimated communication delay among processors:
CommT is an m x m matrix, where CommT(i,j) is the amount of time required for one unit of data to travel from pi to pj.

3. Estimated communication energy consumption among processors:

CommE is an m x m matrix, where CommE(i,j) is the amount of energy consumed by one unit of data to travel from pi to pj. This includes the energy consumed by source device, destination device and any other device involved along the path between them.

We assume that CommT(i,i) = CommE(i,i) = 0, 1 ≤ i ≤ m, i.e., the cost of communication from a processor to itself is zero. We also assume that the task model and processor model stay unchanged until all tasks complete execution.

- **Cost function:**

\[ C = \alpha \text{makespan} + \beta \text{total_consumed_energy}, \]  
where:

- makespan is the time by which all tasks are completed.

- total_consumed_energy is the total amount of energy consumed by task execution plus the total amount of energy consumed by task interaction among tasks.

- α and β are system attributes representing relative importance weights between processing performance (time) and energy conservation.

This cost function is composed of two cost metrics of different units. While makespan is a time measured in microseconds, milliseconds, seconds, and so on, total_consumed_energy is energy measured in joules. Adding two cost metrics of different units is confusing, if not meaningless. However, it is possible to use a new cost unit, cost tokens for example, and to map both time and energy to cost tokens.
by using suitable, meaningful values for $\alpha$ and $\beta$. Thus, $\alpha$ (token/time unit) represents how many tokens are equivalent to one time unit and $\beta$ (token/energy unit) represents how many tokens are equivalent to one energy unit. However, other methods may be used to unify the units of both cost metrics and that has no effect on the proposed algorithm.

The problem can be formulated now as follows.

Given a task model $\mathcal{T}$ and a processor model $\mathcal{P}$, find a schedule $\mathcal{S}$ that maps each task to a processor and determines the starting time of each task in such a way that minimizes the following cost function:

$$C = \alpha \text{ makespan} + \beta \text{ total\_consumed\_energy},$$

such that:

- There is no execution overlapping among tasks that are assigned to the same processor.
- All precedence relations are satisfied.
- The schedule is non-preemptive, i.e., each task is assigned to only one processor and once a task starts execution, it can not be interrupted until it finishes.

### 4.3 Energy-aware Task Scheduling algorithm (E.T.S.)

The proposed scheduling algorithm is divided into two stages: task assignment and then execution ordering. In the first stage, each task is assigned to one processor to execute it. In the second stage, the execution order in each processor is decided.

#### 4.3.1 Task assignment

The task assignment algorithm uses a modified version of the level-based list scheduling heuristic (see Section 2.1.5) that is adapted to consider heterogeneous processors and energy consumption. In this scheduling problem, the execution time of a
task varies from one processor to another and, therefore, it is hard to calculate the level of a task because that depends heavily on the task assignment. Nevertheless, it is possible to use the minimum execution time of each task as its predicted execution time and then to use this predicted execution time in computing the level of this task, i.e., it is in fact an estimated level rather than an actual one (as defined in Section 2.1.3). In this algorithm, the priority of a task is its estimated level. Figure 4.1 (a-e) shows an example of task and processor models, and Fig. 4.1 (f) shows the corresponding tasks estimated levels which are computed based on the minimum execution time of each task.

On the other hand, this algorithm uses a greedy processor selection policy based on the finishing time and energy consumption on each processor. The cost of assigning task $t$ to processor $p$ is:

$$AssignmentCost (t,p) = \alpha FT + \beta consumed\_energy,$$ where:

1. $AssignmentCost (t,p)$ is the cost of assigning task $t$ to processor $p$.

2. $FT$ is the finishing time of $t$ on $p$ which is affected by the communication delay, execution delay and current load on $p$.

3. $consumed\_energy$: is the total energy consumed by executing $t$ on $p$ which includes the energy consumed by execution and energy consumed by communication.

4. $\alpha$ and $\beta$ are the same system attributes of the cost function defined in Section 4.2.

What is meant by communication here is the communication from $t$'s predecessors to $p$. The algorithm simply selects the processor that can run $t$ with minimal cost.
The task assignment algorithm can be described as follows.

- First, assign each task a priority equal to its estimated level which includes the minimum execution time of each task along the longest path.

- Maintain a *ready queue* which includes those tasks of which all predecessors have been scheduled. This ready queue is initialized to have those tasks which have no predecessors and whenever a task is scheduled, the ready queue is updated if some tasks become ready.

- As long as the ready queue is not empty, the ready task with the highest priority is removed from the front of the ready queue and assigned to the processor that can execute it with the minimum cost, i.e., using the *AssignmentCost* function. It is also scheduled to start as early as possible on the selected processor.

Figure 4.2 shows the task assignment process of the example in Fig. 4.1 when $\alpha = 1$ token/µs and $\beta = 1$ token/nJ. In step 1, the ready queue is initialized to have those tasks with no predecessors, namely $t_1$, $t_4$ and $t_3$. In step 1, $t_1$ is assigned first because it has the
Figure 4.2: Illustration example for the task assignment process of the example in Fig. 4.1, with $\alpha = 1$ token/\mu s and $\beta = 1$ token/\text{n J}.
highest priority, and it is assigned to \( p_1 \) because \( AssignmentCost(t_1, p_1) = 4 < AssignmentCost(t_1, p_2) = 6 \). Once \( t_1 \) has been assigned, \( t_2 \) becomes ready and is inserted to the ready queue as shown in step 2. Each step in Fig. 4.2 shows the ready queue after the previous assignment step and the current assignment decision. This process continues until \( t_5 \) is assigned in step 6.

To give a formal description of the task assignment algorithm, the following variables are defined.

- \( RQ \): a ready queue that includes all tasks of which all predecessors have been scheduled.
- \( WaitingFor \): a one-dimensional array, where \( WaitingFor[c] \) is the number of immediate predecessors of \( c \) that have not yet been scheduled. Whenever one of \( c \)'s predecessors is scheduled, \( WaitingFor[c] \) is decremented, and when it reaches zero, \( c \) is inserted into \( RQ \).

In the following algorithm, the overall complexity of each step is shown in curly brackets. In our complexity analysis, we assume that DAG is used to represent the interaction between tasks and is represented using a data structure that has the following property: If a task \( t \) has \( j \) predecessors and \( k \) successors, you can find all predecessors of \( t \) in time \( O(j) \) and all successors of \( t \) in time \( O(k) \). A representation which has two adjacency matrices, one for incoming edges and one for outgoing edges, has this property. It is also assumed that there are \( m \) processors, \( n \) tasks and \( e \) task interaction links (\( e \) is the number of edges in the DAG). The task assignment algorithm follows.

1. **Compute the estimated level of each task.**
   
   (Finding the minimum execution time of each task is done in time \( O(mn) \) and finding the level of each task is known to be equivalent to the problem of single-source longest path)
which can be solved using Bellman-Ford shortest path algorithm in time $O(ne)$ [13,14]. Therefore, the total complexity of this step is $O(nm+ne)$.

2. Initialize $RQ$ to have those tasks with no predecessors. (This step is done by one pass through all tasks in time $O(n)$.)

3. Initialize $WaitingFor$. (This step is done by one pass through all tasks along with all interaction links they have in time $O(n+c)$)

4. While $RQ$ is not empty (This step is done in time $O(n)$ because there are $n$ tasks.)
   4.1 Remove from $RQ$ the task ($t$) with the largest estimated level. If two tasks have the same largest estimated level, task id is used to break the tie.
      (This step is done $n$ times and its overall complexity is $O(n^2)$ (it is equivalent to finding the minimum value of an array of size $n$, $n$ times).)
   4.2 $MinCost = +∞$. (This step is done in time $O(n)$ because it will be repeated $n$ times)

4.3 For each processor $p$
   4.3.1 Find the ready time ($rt$) of $t$ on $p$ (i.e., the time by which all predecessors of $t$ have completed execution and all data sent from $t$'s predecessor has arrived to $p$).
      (This step will be done for each task $t$ and on each processor $p$.
      However, if $t$ has $j$ predecessors, this step will be done $jm$ times for $t$ and, therefore, the total complexity of this step over all tasks and all processors is $O(em)$.)

4.3.2 Find the energy consumption ($ec$) of $t$ if being executed on $p$ (this includes the energy consumed by $t$'s execution and by $t$'s interaction with its predecessors).
      (This step is similar to step 4.3.1 and has a complexity of $O(em)$.)

4.3.3 Find the starting time ($st$) and finishing time ($ft$) of task $t$ if being assigned to $p$ (this includes finding the earliest available time slice in $p$ that begins on or after $rt$ and is enough for $t$).
{Initially, there are $m$ time slices (one for each processor). Whenever a task is assigned to a processor, a maximum of one time slice may be added. Therefore, in the worst case, $m$ time slices will be checked for the first scheduled task, $m+1$ time slices will be checked for the second scheduled task, $m+2$ time slices will be checked for the third scheduled task, ... and $m+n-1$ time slices will be checked for the last scheduled task. Therefore, the total complexity of this step is $O(nm + n^2).}$

4.3.4 $\text{AssignmentCost} = \alpha ft + \beta ec$. \{This step will be done for $n$ tasks over $m$ processors in time $O(nm)$\}

4.3.5 If $\text{AssignmentCost} < \text{MinCost}$ \{This step is similar to step 4.3.4 and has a complexity of $O(nm)$\}

4.3.5.1 $\text{MinCost} = \text{AssignmentCost}$ \{This step is similar to step 4.3.4 and has a complexity of $O(nm)$\}

4.3.5.2 $\text{SelectedProcessor} = p$ \{This step is similar to step 4.3.4 and has a complexity of $O(nm)$\}

4.3.5.3 $\text{StartingTime} = st$ \{This step is similar to step 4.3.4 and has a complexity of $O(nm)$\}

4.4 Assign $t$ to $\text{SelectedProcessor}$ and schedule it to start on time $\text{StartingTime}$. \{This step will be done once for each task and, therefore, has a complexity of $O(n)$\}

4.5 For each task $c$ which is an immediate successor of $t$

4.5.1 $\text{WaitingFor}[c] = \text{WaitingFor}[c]-1$.

4.5.2 If $\text{WaitingFor}[c] = 0$, insert $c$ into RQ.

\{Steps 4.5.1 and 4.5.2 will be done for each scheduled task $t$. However, if $t$ has $k$ successors, this step will be done in time $O(k)$ for $t$. Therefore, the total complexity of these steps for all tasks is $O(e)$.\}

Therefore, by summing up the complexity of all steps, the overall complexity of this algorithm is $O(ne+n^2+em+nm)$. However, $m$ is usually much smaller than $n$ and $e$ and, therefore, the complexity can be abbreviated to $O(ne+n^2)$.

49
4.3.2 Execution ordering

There are two factors that may affect the execution ordering quality of the task assignment algorithm described in the previous section:

1. The communication delay was excluded in computing the estimated level.
2. The minimum execution time was considered as an estimate of the execution time of each task.

This may affect the makespan quality of the same assignment. However, even though finding the optimal order for a given assignment is NP-hard [21], several heuristics have been proven by experiment to be effective in minimizing the makespan [21]. One of the heuristics proposed in [21] is used for our execution ordering algorithm and it can be described as follows. Once the task assignment is decided, the exact communication delay and the exact execution time of each task can be determined and, therefore, a more efficient execution order can be made based on the exact level. The execution order is based on computing the exact level of each task which includes the exact execution times and the exact communication delays, and then ordering the task execution on each processor based on the exact levels of those tasks assigned to it. It is obvious that ordering the execution of tasks on the same processor has no effect on energy consumption. Figure 4.3 shows the exact levels of the tasks shown in Fig. 4.1 after being assigned as shown in Fig. 4.2.

Another refinement that can be achieved is to consider a task ready only when all of its immediate predecessors have completed their execution and all of the data it should receive from them have arrived to the processor to which it was assigned. This can help avoid unnecessary idle gaps resulting from scheduling tasks prematurely. The execution
ordering algorithm is based on the Ready Task Critical Path (RCP) heuristic proposed in [21] that can be described as follows.

- First, assign each task a priority equal to its exact level which includes the actual execution times and the actual communication delays along the longest path.

- At each processor $p_i$, maintain a local ready queue $RQ_i$ which includes those tasks assigned to $p_i$ of which all immediate predecessors have been executed and all of the data they are supposed to receive from their immediate predecessors have been received by $p_i$. $RQ_i$ is initialized to have those tasks which have no predecessors.

- At each processor $p_i$, maintain a local clock $LC_i$ which is always set to the maximum of the earliest time at which $p_i$ is available for execution and the earliest time at which $Q_i$ is not empty.

- Maintain a global clock $GL$ which is always set to the minimum local clock among all processors.

- For each scheduling step at time $GL$, go to the available processor $p_i$ (i.e., $LC_i = GL$), get the highest priority task from the front of its $RQ_i$ and schedule it to start as
early as possible on \( p_1 \). If two processors are available at the same time \( GL \), the one with the smallest processor index is chosen.

After applying this execution ordering algorithm to the example in Fig. 4.1 and Fig. 4.2, the execution order of tasks assigned to \( p_1 \) will be the same because \( t_1, t_2 \) and \( t_5 \) all have precedence relations. However, in \( p_2 \), \( t_3 \) will be scheduled to start first before tasks \( t_4 \) and \( t_6 \) because \( t_3 \) has the highest exact level and, therefore, \( t_5 \) will be able to start execution on \( p_1 \) at time 6 and finish at time 8 leading to a shorter, closer to optimal make span. The result of the execution ordering process is shown in Fig. 4.4. The cost of this schedule is \( 1 \times 8 + 1 \times 18 = 26 \) Cost Tokens.

![Gantt Chart](image)

**Figure 4.4 Execution ordering.**

In [21] the authors provide neither a formal statement nor a complexity analysis for their execution ordering algorithm. Based on a careful reading of their paper, we develop a formal description and complexity analysis to their execution ordering algorithm as follows. We begin by defining the following variables and structures.

- **Event**: an event is defined by the following attributes:

  - **Type**: there are two types of event: \( \text{ReadyTask} \) and \( \text{ReadyProcessor} \).
  
  - **EventTime**: the time at which the event occurs.
Processor: if Type is ReadyProcessor, Processor determines which processor becomes ready.

Task: if Type is ReadyTask, Task determines which task becomes ready.

- **EventList**: a list of all events.
- **IdleProcessor**: a Boolean one-dimensional array, where IdleProcessor[p] tells whether processor p is idle or not.
- **TaskAssignment**: a one-dimensional array, where TaskAssignment [t] determines the processor to which task t was assigned by the task assignment algorithm.
- **RQp**: a ready queue including all tasks, which are assigned to the processor p, of which all data sent from their immediate predecessors has arrived to p. There is a queue for each processor.
- **WaitingFor**: a one-dimensional array, where WaitingFor[c] is the number of immediate predecessors of c that have not yet been scheduled. Whenever one of c’s immediate predecessors is scheduled, WaitingFor[c] is decremented, and when WaitingFor[c] reaches zero, a ReadyTask event is added for c.
- **ReadyTime**: a one-dimensional array, where ReadyTime[c] is the time by which all data sent from c’s immediate predecessors has arrived to the processor to which c is assigned (i.e., TaskAssignment [c]). ReadyTime[c] is set to 0 initially and whenever a task t, which is a predecessor of c, is scheduled, if the data sent from t will arrive to TaskAssignment[c] at a time later than ReadyTime[c], ReadyTime[c] is updated properly.

In the following algorithm, the overall complexity of each step is shown in curly brackets. In our complexity analysis, we assume that DAG is used to represent the...
interaction between tasks and is represented using a data structure that has the following
property: If a task $t$ has $j$ predecessors and $k$ successors, you can find all predecessors of $t$
in time $O(j)$ and all successors of $t$ in time $O(k)$. A representation which has two
adjacency matrices, one for incoming edges and one for outgoing edges, has this
property. It is also assumed that there are $m$ processors, $n$ tasks and $e$ task interaction
links ($e$ is the number of edges in the DAG). The execution ordering algorithm follows.

1. Compute the exact level of each task. {This step is similar to computing the
   estimated level in the task assignment algorithm and has a complexity of $O(nm+ne)$}
2. For each processor $p$, initialize $RQ_p$ to have all tasks which are assigned to $p$
   and have no predecessor. {This step can be done by one pass through all tasks in
time $O(n)$}
3. For each task $t$, $ReadyTime[t]=0$. {Since we have $n$ tasks, the complexity of this
   step is $O(n)$}
4. For each processor $p$, $IdleProcessor[p]=TRUE$. {Since we have $m$ processors, the
   complexity of this step is $O(m)$}
5. Initialize $WaitingFor$. { This step is done by one pass through all tasks along with all
   interaction links they have in time $O(n+e)$}
6. Initialize $EventList$ to have $m$ $ReadyProcessor$ events; one for each processor
   (all processors are ready initially). {Since we have $m$ processors, the complexity of
   this step is $O(m)$}
7. While ($EventList$ is not empty)
   7.1 Remove the event $ev$ which has the earliest $EventTime$.
      { We have $O(m+n)$ events and this step is equivalent to finding the minimum
      value of an array of size $m+n$, $m+n$ times. Therefore, the overall complexity of
      this step is $O(m^2+n^2)$}
6. If $ev.Type$ is $ReadyProcessor$ {Since we have $m+n$ events, the complexity
   of this step is $O(m+n)$}
7.2.1 If $RQ_{ev.Processor}$ is not empty {Since we have $m+n$ events, the complexity of this step is $O(n+m)$}

7.2.1.1 Remove from $RQ_{ev.Processor}$ the task $(t)$ with the largest exact level. If two tasks have the same largest level, task id is used to break the tie. {The complexity of this step is $O(n^2)$ (it is equivalent to finding the maximum value of an array of size $n$, $n$ times).}

7.2.1.2 Schedule $t$ to start on time $ev.EventTime$. {Since we have $n$ tasks, the complexity of this step is $O(n)$}

7.2.1.3 Find the finishing time of $t$ ($ft$). {This step is similar to step 7.2.1.2 and has a complexity of $O(n)$}

7.2.1.4 Add the following $ReadyProcessor$ event

[ $ReadyProcessor$, $ft$, $ev.Processor$, 0 ]. {This step is similar to step 7.2.1.2 and has a complexity of $O(n)$}

7.2.1.5 For each task $c$ which is a successor of $t$

7.2.1.5.1 $WaitingFor [c]= WaitingFor [c]-1$. {Since we have $e$ interaction links, the complexity of this step is $O(e)$}

7.2.1.5.2 Compute the arrival time (at) of data sent from $t$ to $c$. {This step is similar to step 7.2.1.5.1 and has a complexity of $O(e)$}

7.2.1.5.3 If $at > ReadyTime[c]$, $ReadyTime[c]=at$. {This step is similar to step 7.2.1.5.1 and has a complexity of $O(e)$}

7.2.1.5.4 If $WaitingFor [c]=0$, add the following $ReadyTask$ event [ $ReadyTask$, $ReadyTime[c]$, 0, c ]. {This step is similar to step 7.2.1.5.1 and has a complexity of $O(e)$}

7.2.1.6 $IdleProcessor [ev.Processor]= FALSE$. {Since we have $n$ tasks, the complexity of this step is $O(n)$}
7.2.2 Else // $RQ_{ev.Processor}$ is empty {Since we have $m+n$ events, the complexity of this step is $O(n+m)$}

7.2.2.1 IdleProcessor [ev.Processor]= TRUE. { Since we have $n$ tasks, the complexity of this step is $O(n)$}

7.3 Else // ReadyTask event. {Since we have $n$ ReadyTask events, the complexity of this step is $O(n)$}

7.3.1 Insert ev.Task into $RQ_{TaskAssignment[ev.Task]}$. {This step is similar to step 7.3 and has a complexity of $O(n)$}

7.3.2 If IdleProcessor[TaskAssignment[ev.Task]] [This step is similar to step 7.3 and has a complexity of $O(n)$]

7.3.2.1 Remove ev.Task from $RQ_{TaskAssignment[ev.Task]}$. {This step is similar to step 7.3 and has a complexity of $O(n)$}

7.3.2.2 Schedule ev.Task to start on time ev.EventTime. {This step is similar to step 7.3 and has a complexity of $O(n)$}

7.3.2.3 Find the finishing time of ev.Task (ft). {This step is similar to step 7.3 and has a complexity of $O(n)$}

7.3.2.4 Add the following ReadyProcessor event

[ ReadyProcessor, ft, TaskAssignment[ev.Task], 0 ]. {This step is similar to step 7.3 and has a complexity of $O(n)$}

7.3.2.5 For each task $c$ which is a successor of ev.Task

7.3.2.5.1 WaitingFor [c]= WaitingFor [c]-1. {Since we have $e$ interaction links, the complexity of this step is $O(e)$}

7.3.2.5.2 Compute the arrival time (at) of data sent from ev.Task to $c$. {This step is similar to step 7.3.2.5.1 and has a complexity of $O(e)$}

7.3.2.5.3 If at > ReadyTime[c], ReadyTime[c]=at. {This step is similar to step 7.3.2.5.1 and has a complexity of $O(e)$}

7.3.2.5.4 If WaitingFor [c]=0, add the following ReadyTask event [ ReadyTask, ReadyTime[c],
0, \ c]$. {This step is similar to step 7.3.2.5.1 and has a complexity of $O(e)$}

7.3.2.6 \textit{IdleProcessor [TaskAssignment[ev.Task]]} = \text{FALSE}. \ {\text{Since we have } n \text{ tasks, the complexity of this step is } O(n)}$

Therefore, by summing up the complexity of all steps, the total complexity of this algorithm is $O(ne+m^2+n^2)$, and since $m$ is usually much less than $n$, the complexity of this algorithm can be abbreviated to $O(ne+n^2)$.

4.4 Experimental results

In order to show the effectiveness of the E.T.S. algorithm, we conducted several experiments on random task and processor models and compared the results of E.T.S. algorithm with those of two other classical methods. Specifically, we compared E.T.S. with List Scheduling (L.S.), which is equivalent to E.T.S. when $\beta$ is set to zero and $\alpha$ is set to 1 (i.e., minimizing the makespan only) and with Single Processor (S.P.) scheduling, which assigns all tasks to one processor that can execute them all with minimal total cost. In fact there is not any other energy-aware scheduling algorithm to compare with the E. T. S., which imposed a restriction on our experiment. However, we chose L.S. to show that a time-based scheduling algorithm is not capable of conserving energy. And we chose S.P. to show that what all processors can do cooperatively is much better than what the best of them can do independently. We used TGFF to generate random benchmarks. We used 200 benchmarks each of which has about 200 tasks and about 400 inter-task communication dependency links. The average execution time is 1000 $\mu$s and the average execution energy consumption is 1000 $nJ$. The average data units on each communication link is 100 bytes. We also generated a random processor model of 5 processors for each
benchmark. The communication delay between any two processors has an average of 5 \( \mu s/byte \) (this includes the queuing, medium access and transmission delays). The average communication energy consumption between any two processors is 5 \( nJ/byte \). In this experiment, we set \( \alpha = 1 \) token/\( \mu s \) and \( \beta = 1 \) token/\( nJ \) to reflect equal importance weights of performance and energy conservation. Table 4.1 shows a comparison between E.T.S., L.S. and S.P. in terms of the average makespan, the average consumed energy and the average total cost over all benchmarks. It turns out that while the makespan of the results generated by E.T.S. is, on average, more than that of the results generated by L.S. by about 12%, the results generated by E.T.S. consume, on average, 22% less energy compared to those generated by L.S., and while the results generated by E.T.S. consume, on average, 15% more energy compared to those generated by S.P., the makespan of E.T.S. results is, on average, only 20% of that of S.P. results. However, in terms of total cost, which is the most important factor, E.T.S. excels. Figures 4.5 – 4.7 show detailed results of 20 randomly selected benchmarks.

<table>
<thead>
<tr>
<th></th>
<th>Avg. Makespan (( \mu s ))</th>
<th>Avg. Consumed Energy (( nJ ))</th>
<th>Avg. Total Cost (cost tokens)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EA</td>
<td>45627</td>
<td>270631</td>
<td>316259</td>
</tr>
<tr>
<td>LS</td>
<td>41347</td>
<td>350827</td>
<td>392174</td>
</tr>
<tr>
<td>SP</td>
<td>236816</td>
<td>229367</td>
<td>466184</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of different methods.

4.5 Worst case analysis

As is the case with many greedy heuristics, the worst performance of the proposed scheduling algorithm occurs when a local optimality is misleading. Assigning a task \( t_1 \) to the processor that can execute it with the minimum cost may significantly increase the minimum cost of another unassigned task \( t_2 \). For example, according to the task and
Figure 4.5: Makespan comparison.

Figure 4.6: Energy consumption comparison.

Figure 4.7: Total cost comparison.
processor models shown in Fig. 4.8, \( t_1 \) will be assigned to \( p_3 \) as it is the processor that executes \( t_1 \) with the minimum cost. And, therefore, \( t_2 \) will suffer either a communication cost that is too high (if being assigned to \( p_1 \) or \( p_2 \)) or an execution cost that is too high (if being assigned to \( p_3 \)). However, if the scheduler were wiser (less greedy), it would have assigned \( t_1 \) to \( p_2 \) and, therefore, it would be possible to assign \( t_2 \) to \( p_2 \) with a much lower cost. If this simple scenario occurs many times during the scheduling process, the result will be far away from the optimal schedule. However, a backtracking approach may alleviate the effect of such a case at the cost of higher algorithmic complexity.

Theoretically, the proposed scheduler does not guarantee any bound on the distance to the optimal schedule. Thus in Fig. 4.8, increasing all values which are in bold arbitrarily does not change the optimal schedule (which assigns both tasks to \( p_2 \)) but pushes the result of the proposed scheduler arbitrarily far from the optimal one. Since there is no bound on how large the bold values are, there is no bound on how far from optimal the result of our E.T.S. algorithm could get. However, the occurrence of such an extreme case requires two conditions to apply:

![Figure 4.8: An example of a processor model and a task model.](image-url)
1. The communication cost is extremely high.

2. The execution costs of some related tasks (i.e., those which have precedence relations among themselves) vary significantly among different processors (see the execution costs of $t_1$ and $t_2$ in Fig. 4.8).

While L.S. has the same problem because it uses a similar greedy heuristic, S.P. does not suffer this problem, yet, it lacks the parallelism which is necessary for minimizing the makespan and does not guarantee a quality bound.

Based on the experiments we have conducted, there was a very small number of benchmarks where L.S. results were better than E.T.S. ones, and the differences in those cases were very slight. Such cases occur when the energy consumption and the communication delay are strongly related, and when energy consumption is so low that the makespan dominates the total cost. It is not a surprise to have E.T.S. results better than L.S. ones in most of the benchmarks, simply because L.S. does not consider energy at all. There are also some benchmarks where S.P. results are better than E.T.S. ones. This happens when the communication cost is high as compared to the execution cost because S.P. has the advantage of not paying any communication cost.

However, when the communication cost is low and energy consumption is significant, which are the real assumptions of remote execution platforms, E.T.S. excels.

4.6 Summary

While the algorithm proposed in the previous chapter targets minimizing the total consumed energy regardless of the time required to have all tasks completed, this chapter introduces a scheduling algorithm that reconciles energy conservation with processing
performance. It maintains a dynamic cost function of energy and time, and gives the system the flexibility to set different importance weights for each of them.

The proposed algorithm is suitable for mobile computing environments where processing performance and energy conservation may have different priorities in different situations.
Chapter 5

Conclusion

Towards enabling ubiquitous, high performance mobile computing, this thesis introduced the wireless cooperative system which consists of heterogeneous mobile and stationary computing devices, which are connected via a wireless medium. Through the use of a remote execution platform, the involved devices can help each other in running different mobile applications (tasks) in order to conserve energy and improve performance. However, this environment can be abstracted to a set of consumers, i.e., mobile applications, and a set of resources, i.e., energy and processing power in each device. Therefore, an efficient scheduler, which finds proper execution plans, is needed; and this was the core of this thesis.

This chapter begins by stating the key contributions in the next section, Section 5.2 discusses the importance of wireless cooperative systems, Section 5.3 shows that this approach is practical and can be applied without significant changes to the existing systems and Section 5.4 presents Mobile wireless Ad Hoc NETwork (MANET) as an
example of a framework for wireless cooperative system, and suggesting some future research directions in Section 5.5.

5.1 Main contribution

The original contribution of this thesis can be summarized as follows.

- It formulates novel energy-aware allocation and scheduling problems that are suitable for recent and future computing environments, such as MANETs and ubiquitous computing environments, in which energy is one of the most valuable resources.

- It proposes a heuristic-based allocation algorithm that shows empirically the ability to generate task assignments which are near-optimal in terms of minimizing the total consumed energy.

- It proposes a heuristic-based scheduler that reconciles energy conservation with performance improvement, and provides the system with the ability to set different importance weights for each of them.

5.2 Importance of wireless cooperative systems

What a mobile computing device can do is limited by how much energy is available in its battery. Having better utilization of device batteries, by designing more energy-efficient hardware and software components, is presumed to be a promising direction. Even though several methods follow this direction, they are still limited by the amount of energy available in the battery. Computation offloading [3,9] was the first, and the only, method that overcomes this limitation. However, computation offloading assumes the availability of a wall-powered stationary server or at least a more capable server (even if it is not stationary), and most of the work in this direction is limited to the case where
there is one resource-limited device and one relatively more capable one as discussed in the previous chapters.

The wireless cooperative system paradigm gives mobile computing devices the possibility to use all resources in nearby computing devices and provides the optimization methods that make efficient use of such a resource-restricted environment. Using a wireless cooperative system, a mobile computing device can perform a computational task that was impossible to be performed by the same device independently. Furthermore, a mobile computing device can have such a high performance that would not be achievable without the cooperation of other computing devices. Moreover, wireless cooperative systems do not require the existence of a wall-powered server and do not limit the number of involved computing devices. Experiments, discussed in Section 4.4, show that the performance of a wireless cooperative system (i.e., all involved devices working cooperatively) is much higher than that of the best involved computing device working independently.

5.3 Viability of wireless cooperative systems

The main keys to make wireless cooperative systems practical are:

1. The ability for mobile computing devices to communicate with each other:

   Most of the computing devices are equipped with wireless communication facilities that allow them to communicate and exchange data. Moreover, recent advances in wireless networks, such as Wireless Local Area Networks WLAN [11], make it possible for these devices to communicate via a wireless medium whenever the necessary infrastructure is available. Furthermore, advances in Mobile wireless Ad Hoc NETworks (MANETs) make it possible for mobile
devices to construct a wireless network on the fly even in non-infrastructured areas (see Section 5.4). Therefore, communication will never be a deterrent from constructing a wireless cooperative system.

2. The ability to predict resource usage:

In the problems defined in Section 3.1 and Section 4.2, we assume that the task model and the processor model are known prior to the execution of any task. This implicitly assumes that the energy consumption and execution time of each task on each device is known in advance. Even though this may seem impractical at first sight, recent advances in remote execution platforms have made it feasible to predict such information accurately [3,9]. Remote execution platforms can predict the execution time and energy consumption of each task on each device by monitoring different resources and maintaining a database of resource usage statistics. Smart batteries, profiling and algorithmic complexity are also used to predict resource usage of different tasks. Remote execution platforms have also enabled the process of migrating computational tasks to a remote server via a wireless medium.

On the other hand, the communication cost can be obtained from the underlying network layer protocol. Several energy-aware routing protocols have been designed to provide energy-efficient routes for wireless networks [28]. Therefore, both the processor model and the task model can be made available.

5.4 MANET as a framework for wireless cooperative systems
A MANET is a wireless, self-organized network, which involves a set of heterogeneous mobile computing devices equipped with wireless communication and networking facilities. This kind of network is constructed on-the-fly, i.e., there is no system administration, no base stations and no routers. To join such a network, a mobile computing device should be able to detect the presence of other such devices and perform the necessary handshaking. Each mobile computing device in such a network has a limited transmitting/receiving range. Nevertheless, each device can communicate directly with any device inside its range and can also communicate indirectly with those devices outside its range. With indirect communication, other devices are used to relay (forward) data from source to destination. Due to the fact that wireless channels are broadcasting ones, some devices may be forced to receive messages that do not belong to them and, therefore, they spend energy on receiving messages of other devices; this is called overhearing. Figure 5.1 shows an example of a MANET of 6 nodes. In Fig. 5.1, since node B is within the transmission range of node A (i.e., the dotted circle), node A can

![Figure 5.1: MANET routes.](image-url)
communicate directly with node B. However, since node F is not within the transmission range of node A, node A can communicate with node F only through other relaying nodes (node C and node D) as shown in the dotted arrows. However, node B will overhear all messages sent from node A to node F.

MANETs have the advantage of being deployable in non-infrastructured environments, such as disaster areas and battlefields. On the other hand, a significant amount of research has been devoted to design energy-efficient protocols for this type of network, which make it suitable for energy-limited mobile computing devices [11,28].

Since the MANET network layer is able to report the energy consumption and the delay of different communication links, deploying the wireless cooperative system on top of MANET is absolutely feasible. The only issue that needs to be considered is to consider the energy consumption of all overhearing nodes as part of the energy cost of different links (i.e., CommE), and this is also available from the underlying network layer which is supposed to have information about the topology of the network.

MANET is an example of a cooperative environment in which different devices are motivated to help each other. For example, a rescue team exploring a disaster area with their mobile computing devices can form a MANET and work cooperatively. However, while MANETs brought about a promising opportunity for an anywhere- and anytime-communication structure, the deployment of a wireless cooperative system on top of a MANET sets forth a novel concept of Ad Hoc Computing, and that is why we choose MANET as an example of an underlying framework for wireless cooperative systems.

5.5 Future work
New research directions are suggested by our work thus far. In this section we define two energy-aware scheduling problems that can be directly derived from this thesis.

1. Scheduling with high degree of mobility:

   The problem defined in Chapter 4 assumes that the processor model stays unchanged until all tasks have completed execution. This assumption involves an implicit assumption of low degree of mobility (i.e., mobile computing devices do not make such a significant movement that alters the communication cost between two devices). It also implicitly assumes that no device will be shut down during the execution of the tasks or during the scheduling process. An interesting problem can be formulated when this assumption is relaxed. Specifically, the problem can be defined as follows.

   With the task model, the processor model and the cost function of Section 4.2, assume that an initial schedule $S_1$ was obtained (using the E.T.S. algorithm or any other energy-aware algorithm) and tasks are being executed. If the processor model changes, find a schedule $S_2$ which is obtained by modifying $S_1$ properly such that the cost of $S_2$ is minimized.

2. Scheduling to keep all devices alive:

   Another task allocation problem can be defined when the main objective is to keep all computing devices alive (i.e., have enough energy to continue functioning). In such a problem, the energy level at each device is an important factor. This problem can be described generally as follows. Find a task assignment that maximizes the following cost function.
\[ C = \min_i (\text{EnergyLevel}(i)), \ 1 < i < m. \text{ where } m \text{ is the number of processors and} \]

\textbf{EnergyLevel}(i) is the amount of energy available at processor \( p_i \).

However, while the task model of Section 3.1 is suitable for this problem, a new processor model needs to be carefully designed.
References


