Approximation Algorithms for Task Allocation with QoS and Energy Considerations

by

Bader N. Alahmad

B.S. Computer Engineering, Jordan University of Science and Technology, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE STUDIES

(Electrical and Computer Engineering)

The University Of British Columbia

(Vancouver)

May 2011

© Bader N. Alahmad, 2011

Abstract

We consider general problems of allocating tasks to processors where each task is associated with a set of service classes. A service class is a tuple: the first element represents the resource utilization and the second element the quality of service associated with that resource utilization. Before allocating a task to a processor, we need to assign it to a service class. We consider some elementary problems that arise from this setting. What is the minimum number of processors needed if we also need to attain a minimum aggregate QoS level? Given a fixed set of processors, what is the maximum attainable aggregate QoS? Such questions apply to the partitioned scheduling of real-time tasks on multiprocessors and to other resource allocation problems. The basic questions are NP-Hard, and we present polynomial time approximation algorithms for certain special, but practical, cases. We make interesting use of maximum flows in a bipartite graph while developing the polynomial time approximation schemes. We then integrate energy expenditure to the model above and address the problem of partitioning a set of independent, periodic, real-time tasks over a fixed set of heterogeneous processors while minimizing the energy consumption of the computing platform subject to a guaranteed quality of service requirement. This problem is NP-Hard and we present a fully polynomial time approximation scheme for this problem. The main contribution of our work is in tackling the problem in a completely discrete, and possibly arbitrarily structured, setting. In other words, each processor has a discrete set of speed choices. Each task has a computation time that is dependent on the processor that is chosen to execute the task and on the speed at which that processor is operated.

Further, the energy consumption of the system is dependent on the decisions regarding task allocation and speed settings.

Preface

Chapter 3 is based on a draft initiated by Sathish Gopalakrishnan[1], in collaboration with Nathan Fisher [2]. My contribution was writing parts of the aforementioned chapter, and deriving some of the mathematical proofs, in addition to verifying previously written proofs and correcting some mistakes. In general, I contributed to about one third of the content of chapter 3. I would like to indicate that the work in chapter 3 has not been published in any scientific conference or journal to the date of submission of the thesis.

[1] Supervisor, Assistant Professor, Department of Electrical and Computer Engineering, University of British Columbia

[2] Assistant Professor, Department of Computer Science, Wayne State University

Table of Contents

Abstract . . . . **ii**

Preface . . . . **iv**

Table of Contents . . . . **v**

List of Tables . . . . **vi**

List of Figures . . . . **vii**

Glossary . . . . **viii**

Acknowledgments . . . . **x**

1 Introduction **1**

2 Background and Related Work **6**

2.1 Complexity Theory, Combinatorial Optimization and Approximation Algorithms: Preliminaries 6

2.1.1 Complexity Theory 6

2.1.2 Combinatorial Optimization 12

2.1.3 Approximation Algorithms 14

2.2 Related Work 18

3 On Task Allocation Problems with Service Classes **23**

3.1 Motivation: Virtual Machine Allocation 23

3.2 Definitions and Formal Problem Formulation 25

3.2.1 Definitions . . . 25
3.2.2 Problem Definition . . . 27
3.2.3 Problem Hardness . . . 28
3.2.4 Restricted Problem Formulation . . . 29
3.3 Solving Restrictions to TASC . . . 29
3.3.1 Fixed Service Classes . . . 30
3.3.2 Minimal Service Class . . . 32
3.3.3 TASC . . . 34
3.3.4 Maximizing QoS with m Processors . . . 35
3.4 Experimental Evaluation and Further Extensions . . . 35

4 Energy Efficient Task Partitioning and Real-Time Scheduling on Heterogeneous Multiprocessor Platforms with QoS Requirements . . . 37
4.1 Motivation . . . 37
4.2 System Model and Notation . . . 39
4.2.1 Platform . . . 39
4.2.2 Task and Scheduling Model . . . 40
4.2.3 Energy Model . . . 41
4.2.4 Quality-of-Service Model . . . 42
4.2.5 Task Set Encoding . . . 42
4.2.6 Notes Regarding Assumptions . . . 42
4.3 Problem Formulation . . . 45
4.4 Assignment of Tasks to Processors . . . 49
4.4.1 Exact and Optimal Dynamic Program Formulation . . . 49
4.4.2 The Fully Polynomial-Time Approximation Scheme . . . 53

5 Conclusions . . . 65

Bibliography . . . 68

Appendix A A Modified Pruning Technique: Energy Rounding . . . 75
A.1 Running-Time Analysis . . . 80

List of Tables

Table 3.1 Amazon EC2 Instance Pricing and Configuration. (1 EC2 Compute Unit Provides CPU Speeds of About 1 GHz.) (Amazon.com [5]) 23

Table 4.1 Basic Math Benchmark (Single Iteration) 38

Table 4.2 Fast Fourier Transform (FFT) Benchmark (1000 Iterations) 39

List of Figures

Figure 3.1 Hierarchical Scheduling Using a VMM/Hypervisor 26
Figure 3.2 Network Flow and Feasible Assignments 31

Glossary

EDF Earliest Deadline First

PTAS Polynomial Time Approximation Scheme

FPTAS Fully Polynomial Time Approximation Scheme

FFT Fast Fourier Transform

TM Turing Machine

DVS Dynamic Voltage Scaling

P Polynomial-time, the complexity class containing languages decidable by deterministic TMs in polynomial-time

NP Non-Deterministic Polynomial-time, the complexity class containing languages decidable in polynomial-time by non-deterministic TMs

CONP Complement Non-Deterministic Polynomial-time, a language is in the complexity class coNP if and only if its complement is in NP

L Logarithmic-space, the complexity class containing languages decidable in polynomial-time by deterministic TMs that use space that is logarithmic in the size of the input

SAT Satisfiability Problem

RM Rate Monotonic

VMM Virtual Machine Monitor

WCEC Wosrt Case Execution Cycles

WCET Wosrt Case Execution Time

Acknowledgments

I offer my enduring gratitude to the faculty, staff and my fellow students at UBC, who have inspired me to continue my work in this field. I owe particular thanks to my adviser, Dr. S. Gopalakrishnan, for his continuing support, and for giving me the liberty to work on what I love, and whose penetrating questions and provocative discussions taught me to question more deeply.

Special thanks are owed to my parents, who have supported me throughout my years of education, both morally and financially.

I owe particular thanks to my friends for their support, and especially Yazan Boshmaf, who dedicated considerable time listening to me talking about my research problems and engaging in fruitful discussions, and whose ideas and thoughts certainly contributed to the work.

Chapter 1

Introduction

Several fundamental questions in resource allocation for multiprocessor computing systems relate to task allocation schemes that satisfy a set of constraints. For a real-time system, timeliness, or the ability of the system to meet deadlines, is a vital property. Implicit-deadline periodic tasks are well understood from a scheduling perspective Buttazzo, Liu [12, 42]. These are tasks that release jobs periodically, and each job needs to be completed before the next job of the same task is released. A task τ is characterized by a period P and execution time e; the tasks utilization is $u = e/P$. For a set $\Gamma = \{\tau_1, \ldots, \tau_n\}$ of tasks, the total utilization is $U = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} e_i/P_i$.

In one of the simplest models for real-time multiprocessor systems, a set of implicit-deadline periodic tasks needs to be allocated to processors such that no task will miss its deadline Carpenter, Funk, Holman, Srinivasan, Anderson, and Baruah [13]. When tasks are scheduled using the earliest deadline first policy, the problem of partitioning tasks among a set of processors is akin to the bin packing problem. Using EDF, a set of implicit-deadline tasks running on the same processor is schedulable if the total processor utilization is no greater than 1. Partitioning a task set over m processors is feasible if the corresponding bin packing problem Coffman, Garey, and Johnson [18], i.e., how many unit-capacity bins are needed to pack items of sizes $u_1, u_2, \ldots, u_n$? has a solution that is no greater than m. The bin packing problem is computationally hard (NP-hard), and so is the

partitioning problem (Garey and Johnson [25]).

Extensive work has concentrated on understanding the schedulability of real-time task sets on uniprocessor and multiprocessor platforms only in the homogeneous or "single-mode" task model (Buttazzo, Liu [12, 42]). We, however, view the system from a quality-of-service perspective. In general, quality of service is a measure of the maximum error a task might tolerate, or the minimum perceived precision in a dual sense. In our model, quality of service is fully specified by *service classes*, each service class being a tuple (u_j, q_j). Each tuple represents a utilization for the task and the associated quality index, and task τ_i is associated with a set of service classes. We restrict our attention to monotonically-increasing tuples, i.e., if (u_j, q_j) and (u_k, q_k) are two service classes of τ_i and $u_j \geqslant u_k$ then $q_j \geqslant q_k$.

We consider the problem of selecting the optimal choice of service classes in a multiprocessor setting. Although we describe the problem of interest in the specific setting of real-time scheduling for multiprocessors, this problem is a generic resource allocation problem and our interest in the problem stemmed from recent developments in server consolidation and virtualization, which establishes some more context for the applicability of this model and the results (Section 3.1).

We extend the setting above to incorporate energy expenditure of the computing system as the objective value to be minimzed, while maintaining a desired quality of service score. We consider a resource allocation problem where we are given a set of heterogeneous processors with discrete speed settings and a set of periodic, independent, real-time tasks. How do we partition the tasks across the available set of processors and choose appropriate speed settings for those processors such that we achieve minimum energy consumption while satisfying some specified quality of service requirement? In this setting, the energy consumed by the complete system depends on the task allocation and on the speeds of the processors. This problem is motivated by two important issues to consider in the design of current and future embedded systems: energy consumption and processor heterogeneity. Furthermore, the emphasis on a system model with completely discrete set of choices is based on architectural considerations as well as empirical evi-

dence that suggests that such a model is indeed the appropriate model to apply.

Energy in embedded systems and especially battery-powered devices is a valuable resource whose expenditure needs to be kept at minimum in order to increase the lifetime of such systems. At the same time, tasks running on those systems should be appropriately serviced according to their computational and timeliness requirements. There are two major contributors to the overall energy consumption in a processor: (i) the *dynamic* power consumption due to switching activities and (ii) the *leakage* power consumption due to leakage current draw in CMOS circuits (Jejurikar, Pereira, and Gupta [32].) The former depends on the speed at which the processor is operating, while the latter is present whenever the processor is *on*, and is a constant. In addition to the energy consumed by the processor, the total energy consumed by the computing systems depends on the energy consumed by other devices and peripherals in the system (e.g., memory, I/O). This energy consumption is not dependent on the processor speed. This aspect of energy modeling is important because it allows us to capture energy consumption beyond what is specific to the processing units.

Most embedded systems now constitute multiple processors in order to increase the processing throughput. In addition, each processor can be configured to run at a speed (frequency) from a limited set of allowable speeds. Moreover, the processor's operating speed can be varied while the system is running without interrupting task execution. Heterogeneous computing systems consist of multiple processing components having different architectures and computing capabilities, interconnected using different connectivity paradigms. For example, the Nomadik™ platform (Wolf [58]) includes an ARM processor, a video accelerator and an audio accelerator, each of which is itself a heterogeneous multiprocessor. Such systems better meet the demand of applications with diverse requirements, because the computational requirements of a task might differ significantly on different processing elements, and heterogeneous systems allow tasks to be matched to computing elements that better serve their requirements.

In a heterogeneous platform the assignment of tasks to processors may

impact the end-user quality of service because certain processor types may be better suited to certain tasks. For example, executing a graphics task on a specialized graphics processor yields better results than performing the same operation on a general-purpose processor. In a system with heterogeneous compute units and multiple speed settings for each processor, a system architect may explore the trade-off between quality of service and energy efficiency. This is the stage of the design process that we target in this thesis.

To simplify the discussion, we can think of a platform with a fixed number of processors. We need to decide the processor type for each processor from a set of available processor types (permitting heterogeneity in the platform), then selecting a particular speed for a processor and finally assigning tasks to that processor. A task would have a certain worst-case execution time at the selected speed and each instance of a periodic task will consume a certain amount of energy (*active-time energy*) that depends on the type of processor selected and the associated speed of the processor. Additionally when a processor is idle it will consume some energy (*idle-time energy*).

Our energy expenditure model is central to our contribution. We relax many of the impractical assumptions underlying state-of-the-art solutions, including the work of Yang, Chen, Kuo, and Thiele [60], which is the closest to our efforts. In a sense, prior models assume 1) continuity of speed levels on machines 2) linearity of WCET requirements of tasks with respect to processor speed 3) constant WCEC of tasks with respect to speed levels on processors 4) linear interpolation of the energy expenditure when calculations result in speed levels that are not available on the processor, in case of a discrete processor speed model, and 5) that energy expenditure on a processor depends solely on the total utilization of the processor. These assumptions are agnostic to the fact that different tasks exercise differential, and somewhat arbitrary quality of execution as speed varies on a processor. A consequence of such assumptions is that previous models cannot capture the energy as consumed by devices other than the processor, which is due to nonlinearities of task execution requirements and the overall energy consumption when tasks spend considerable fraction of their time interacting

with I/O devices, such as memory and disk (I/O bound tasks). Therefore, those models are very difficult to scale to account for energy expenditure of the compute system as a whole (see chapter 4, section 4.1 for an experimental illustration of some of the intricacies mentioned above, and section 4.2.6 for a more elaborate discussion). Due to the practical needs of discrete and arbitrarily structured settings, our solutions are combinatorial, and our algorithms employ a blend of matching and enumeration techniques, with dynamic programming being the general algorithmic tool. In summary, our contributions are

1. New model for energy expenditure that alleviates previous impractical assumptions;

2. PTASs for varieties of (strongly NP-Hard) task allocation problems, where tasks execute on identical machines, and the goal is to minimize the number of machines on which the task set is to be deployed, while maintaining a certain quality of service score;

3. An FPTAS for the (NP-Hard in the ordinary sense) problem of allocating tasks on unrelated parallel machines (heterogeneous platform), where the number of machines is *fixed*, and the goal is to find an assignment of tasks to service classes and simultaneously build a platform from an assortment of given machine types, and then assign tasks (as they are equipped with service classes) to the machines comprising the platform such that the platform expends as minimum energy as possible while the aggregate quality of service score of the solution assignment is above that of a given quality of service threshold, and

4. Provable worst case bounds on the quality of the solutions returned by our approximation schemes relative to the optimal solution (produced by the exact, intractable algorithm), expressed in terms of a user-supplied error factor that determines the quality vs. efficiency trade-off.

Chapter 2

Background and Related Work

2.1 Complexity Theory, Combinatorial Optimization and Approximation Algorithms: Preliminaries

This section highlights important definitions and sets the theoretical ground upon which our work is based. The presentation in this section is highly influenced by the beautiful books of Papadimitriou and Steiglitz [45], Vazirani [54] and Arora and Barak [7]. Readers familiar with the notions introduced here can skip directly to the core problems in chapters 3 and 4.

2.1.1 Complexity Theory

A decision problem is one which demands either a "yes" or a "no" answer. Decision problems are best described in terms of languages. Consider an alphabet Σ. A language $L \subseteq \Sigma^*$ is a set of strings over Σ, where L encodes instances whose answer to the decision problem is "yes". We say that a language L is *recognizable* if there exists a Turing Machine (TM) [1] that, when run on an input $x \in \Sigma^*$, produces "yes" and halts and if $x \in L$. This

[1] The terms "Turing Machine" and "Algorithm" will be used interchangeably.

TM, however, is not required to halt and say "no" if $x \notin L$, but might loop forever without producing an answer; it is only capable of determining membership in the language L if x is a "yes" instance. A stronger notion of computability is that of *decidability*; that is, if membership in the language for any input can be determined by some TM that always halts: if $x \in L$, then the TM produces "yes" and halts, and if $x \notin L$, then the TM halts and produces "no". Therefore, a language $L \subseteq \{0,1\}^*$ is *decidable* if both L and its complement $\overline{L}$ are recognizable. An example of an undecidable language is the famous *Halting Problem*, which can be stated informally as: given a computer program description and input, is the program ever going to halt when run on the input? This problem is reminiscent of the *program verification* problem, in which we need to check that the program adheres to its specifications over all possible inputs. The formal language associated with the halting problem is

$$L_{\texttt{HALT}} = \{\langle M, w\rangle \mid M \text{ is a TM (or program) that halts on input } w\},$$

where $L_{\texttt{HALT}}$ contains all valid program (TM) encoding and input pairs for which program M halts when presented with input w. In his seminal paper, Turing [52] showed that regardless of the computing power, the halting problem cannot be solved by any algorithm (computer), thus drawing the limits on solvability by machines in general.

Intuitively, a decision problem is a language that has context specific semantics attached to it. For example, consider the following decision problem. `CLIQUE`: Given a graph $G = (V, E)$ and an integer k, does G contain a clique of size at least k ? The required answer to `CLIQUE` as posed above is in the form "yes"/"no". We can write the language corresponding to `CLIQUE` as

$$L_{\texttt{CLIQUE}} := \{G \mid G \text{ contains a clique of size} \geq k\}. \tag{2.1}$$

Accordingly, $L_{\texttt{CLIQUE}}$ is the set of graph encodings, assuming any reasonable encoding such as the adjacency matrix representation, which contain cliques having at least k vertices. Therefore, the language $L_{\texttt{CLIQUE}}$ is decidable if there exists a TM that, given x, can decide in a finite number of steps

whether or not x is a valid graph encoding and that it contains a clique of size at least k, which is exactly deciding whether or not $x \in L_{\texttt{CLIQUE}}$.

The definitions above concerning computation do not consider the efficiency of the algorithm deciding a language: They only require the computation to halt in a finite number of steps, with no regard to the number of steps the machine takes until it finishes its computation. In the following, we shall classify languages, or problems, into sets, according to their *hardness*, which is quantified in terms of the efficiency of the algorithms that so far exist to solve them. The most accepted dichotomy of problems according to their hardness of computation is that of polynomial time solvability. A language is said to be polynomial-time decidable if there exists an algorithm that can decide the language in time polynomial in the size of the input, assuming a reasonable input encoding.

For input x, we shall denote the size of x as $\texttt{SIZE}(x)$, which we define as the number of bits required to encode x input in binary. In fact, we will always assume a binary encoding and thus require our alphabet to be $\Sigma = \{0, 1\}$, so that $x \in \{0, 1\}^*$. For example, consider the problem `PRIMES`: Given an integer N, is N prime?. According to the definition of the size of an instance above, $\texttt{SIZE}(N) = \log_2 N$. As another example, an instance of `CLIQUE` is of the form $I = (G, k)$; the size of the input is comprised of the *dimension* of I, namely $n := |V|$, and the magnitudes of numbers involved in the input, namely k whose size is $\log_2 k$, so $\texttt{SIZE}(I) = n + \log_2 k$.

Later when we discuss our algorithms, the input encoding will be crucial to the analysis of the running time of the algorithm and thus some notions need to be made precise at this point. For a rational number of the form $x = p/q$, where $p, q \in \mathbb{Z}$ and $q \neq 0$, we will assume that x can be represented as a pair (p, q) and thus will require $\mathcal{O}(\log p + \log q)$ bits to be encoded in binary. If we talk of real numbers, then we will assume that we have at our disposal a function $\texttt{APPROX}(n, t)$, where n is the real number and t is the desired number of digits after the decimal point. The function $\texttt{APPROX}(n, t)$ thus returns t decimal places of the real number n. Accordingly, the number of bits required to represent any real number n restricted to the first t decimal digits is equal to the number of bits required to store the procedure

$\texttt{APPROX}(n, t)$. Note that, since the number of valid procedures (or, in general, TM encodings) is countably infinite, and the real numbers are uncountable, it follows that not all real numbers can be represented this way. More specifically, we cannot establish a one-to-one correspondence between the integers (procedures, TM encodings) and real numbers, for otherwise we can fall into a contradiction, which can be shown by a simple diagonalization argument.

The class P contains all languages that are polynomial-time decidable (or computable if we talk of problems or functions). For example, if PRIMES is to be in P, then there should exist an algorithm that decides whether or not a given integer N is prime in $O((\log N)^c)$ for some constant $c > 0$. In fact, PRIMES was shown to belong to the class P by Agrawal, Kayal, and Saxena [2]. Another example of a problem in P is the undirected graph reachability, or the undirected s-t connectivity problem USTCON: Given an undirected graph $G = (V, E)$, and two nodes $s, t \in V$, is there a path in G from s to t ?. As a matter of fact, Reingold [48] proved, in a breakthrough, the stronger result that USTCON is in log-space (the class), thus settling a long lasting open question. The class contains those languages for which there exist TMs that can decide them using $O(\log n)$ space, where n is the size of the input. On the other hand, a language $L \subseteq \{0, 1\}^*$ is said to be in the class NP if there is a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time TM M such that, on input $x \in \{0, 1\}^*$:

- if $x \in L$, then there exists a *certificate* of the solution, a string z whose size is polynomially bounded by the size of x, i.e., $\texttt{SIZE}(z) \leqslant p(\texttt{SIZE}(x))$, or $z \in \{0, 1\}^{p(\texttt{SIZE}(x))}$, such that $M(x, z)$ accepts, and
- if $x \notin L$, then for any string $z \in \{0, 1\}^{p(\texttt{SIZE}(x))}$, $M(x, y)$ rejects;

M is called a *verifier* TM. Note that the definition above requires only "yes" certificates to be polynomially bounded in the size of the input, but not necessarily the "no" certificates. Some authors define a language as being in NP if there exists a non-deterministic polynomial-time TM that decides the language. The name NP stands for Non-deterministic Polynomial-time,

which is derived from the latter definition. An example of a problem in NP is the above-mentioned CLIQUE; a verifier takes as input a graph G and a subgraph H and checks whether or not H is a complete subgraph of G with $V(H) \subseteq V(G)$ of size at least k by checking, for every pair of vertices $i, j \in V(H)$, whether there exists an edge $e = \{i, j\}$ in H that is also an edge in G. This checking requires examining $\binom{|V(H)|}{2} = O(|V(H)|^2)$ vertices, which is certainly a polynomial in the size of G, namely $|V|$. In a sense, the class NP captures those problem that have "succinct" and "efficiently" verifiable "yes" certificates.

Note that our definition of the class NP requires only "yes" certificates to be polynomially bounded in the size of the input, but not necessarily the "no" certificates. For example, consider the Boolean Satisfiability Problem SAT: Let φ be a Boolean formula (a propositional logic formula) with n variables $x_1, \ldots, x_n$, and m clauses $c_1, \ldots, c_m$ in Conjunctive Normal Form (CNF), i.e., $\varphi = c_1 \wedge \ldots \wedge c_m$, where each clause is a disjunction ($\vee$) of literals (variables either negated or not). The question is: given such a formula, is there an assignment of the variables $x_1, \ldots, x_n$ such that φ is satisfied? A "yes" certificate is any assignment for which φ evaluates to `true` (for a suitably defined truth system); such a certificate has length n and is therefore linear in the size of φ, thus $\mathtt{SAT} \in \mathsf{NP}$. On the other hand, a "no" certificate (i.e., for deciding whether φ is NOT satisfiable) is an enumeration of all possible truth assignments of the variables, which is 2^n, an exponential in the size of the formula.

A language $L \subseteq \{0,1\}^*$ in the class coNP if its complement $\overline{L} = \{x \in \{0,1\}^* \mid x \notin L\}$ is in NP. The languages in coNP have succinct and quickly verifiable "no" certificates (counterexamples). For example, the complement of SAT, namely Boolean formulas that are NOT satisfiable for any truth assignment of their variables, is in coNP; we define the language of $\overline{\mathtt{SAT}}$ as

$$L_{\overline{\mathtt{SAT}}} = \{\varphi \mid \varphi \notin \mathtt{SAT}\}.$$

A "yes" certificate to an instance of $\overline{\mathtt{SAT}}$ is the set of all truth assignments of variables (2^n), while a "no" certificate is a single truth assignment of the

variables such that the input formula evaluates to `true`.

If a language L contains some instance that have short "yes" certificates and others that have short "no" certificates, then $L \in \mathsf{NP} \cap \mathsf{coNP}$. An example of a problem in $\mathsf{NP} \cap \mathsf{coNP}$ is `FACTORING`: Given three numbers N, L, U decide if N has a factor M in the interval $[L, U]$. The certificate for membership in NP is the factor M. The certificate for membership in coNP is a certificate of membership in `PRIMES`. To verify membership in `PRIMES`, one should exhibit an integer x that is coprime to N and satisfies the following

- $x^{N-1} \equiv 1 \pmod N$,
- for every prime factor p of $N-1$, $x^{(p-1)/N} \not\equiv 1 \pmod N$.

In fact, one should be able present the prime factorization of $N-1$ for each N recursively as required by the second condition (but prime factors get mercifully smaller at each recursive step). This primality test was conceived by Pratt [47], and he showed that the test requires time that is polynomial in $\log N$, thus showing that `PRIMES` $\in \mathsf{NP}$.

Now we overview the concept of NP-Completeness. Given two languages $L_1 \subseteq \{0,1\}^*$ and $L_2 \subseteq \{0,1\}^*$, we say that L_1 is polynomial time reducible to L_2, denoted as $L_1 \leq_p L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $x \in L_1$ iff $f(x) \in L_2$. Language L_2 is NP-Hard if $L_1 \leq_p L_2$ for every $L_1 \in \mathsf{NP}$. If, in addition, $L_2 \in \mathsf{NP}$, then L_2 is NP-Complete. We point out that the aforementioned `SAT` is the first problem that was shown to be NP-Complete, independently by Cook [19] and Levin [39] (The Cook-Levin Theorem). There are, however, problems that in NP but are not known to be NP-Complete, for they lack a polynomial-time reduction from a known NP-Complete problem. An example of such a problem is `GRAPH_ISOMORPHISM`: Given two $n \times n$ adjacency matrices M_1, M_2, decide if M_1 and M_2 define the same graph, up to renaming of vertices. Assume that the graph $G = (V, E)$ represented by matrix M has its vertices labeled by integers in $[n] = \{0, \ldots, n-1\}$ (the multiplicative group of integers modulo n), where $n := |V|$. Then the cer-

tificate is the permutation $\pi : [n] \to [n]$ such that M_2 is equal to M_1 after reordering M_1's indices according to π.

2.1.2 Combinatorial Optimization

Optimization problems concern themselves with finding the "best" configuration or a set of parameters to achieve some goal. If the variables to be determined are discrete, then optimization problems are called *combinatorial*. On the other hand, in continuous optimization problems one seeks an assignment of variables from the real line, or sometimes one needs to search for a function. We will be concerned only with combinatorial optimization problems. The variables of an optimization problem obey a set *constraints*, usually specified by a set of equalities or inequalities. Configurations whose variables are set such that all constraints are satisfied are called *feasible*. The goal that an optimization problem must achieve is either the minimization or the maximization of the value of an *objective* function, by making a choice of a feasible configuration.

Formally, an optimization problem Π consists of set of instances. Each instance $I \in \Pi$ is a pair (F, c), where F is a set; the domain of feasible points, and c is the cost function

$$c : F \to \mathbb{R}.$$

The problem is to find $f \in F$ such that

$$c(f) \leqslant c(y) \quad \text{for all } y \in F.$$

The point f is called (globally) optimal with respect to instance I, and we shall denote its cost as $\textsc{Opt}_\Pi(I)$. We will often drop the subscript Π when the problem is well understood from the context, and we shall denote the optimal value of the problem over all instances (or if we are considering a generic instance) simply as $\textsc{Opt}$. Intuitively, an instance of a problem is a fixed input that encodes enough information to produce a solution. For instance, consider the Traveling Salesperson Problem `TSP`: Given a set

of n nodes, $\binom{n}{2}$ numbers $d_{i,j}$ denoting the distances between all pairs of nodes, find a closed circuit (i.e., a "salesperson tour") that visits every node exactly once and has minimum total length. If we denote as π a cyclic permutation on n objects, then if we interpret $\pi(j)$ as the city visited after city j, $j \in \{1, \ldots, n\}$, then the goal is to minimize the length of the tour having a cost function defined as the map

$$c \,:\, \pi \rightarrow \sum_{j=1}^{n} d_{j,\pi(j)}.$$

The set F corresponding to an instance of TSP can thus be defined as

$$F = \{\text{all cyclic permutations } \pi \text{ on } n \text{ objects}\}.$$

As another example, consider the Linear Programming problem LP: Let m, n be positive integers, $\mathbf{b} \in \mathbb{Z}^m$, $\mathbf{c} \in \mathbb{Z}^n$, and A be an $m \times n$ matrix matrix such that $a_{i,j} \in \mathbb{Z}$. Then an instance of LP is defined as

$$F = \{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} \leqslant \mathbf{b}, \quad \mathbf{x} \geqslant \mathbf{0}\},$$

and the goal is to maximize the cost c, which is defined as the map

$$c \,:\, \mathbf{x} \rightarrow \mathbf{c}^T\mathbf{x}.$$

Consider the following legitimate question: what if the decision version corresponding to a combinatorial optimization problem is NP-Complete? Does Hardness carry over from decision problems to their optimization versions? The answer to this question is YES. For any minimization (maximization) optimization problem, one can formulate its decision version by upper (resp. lower) bounding the value of its objective (cost) function by some rational number B and ask: "Is there $f \in F$ such that $c(f) \leqslant B$ (resp. $c(f) \geqslant B$ for maximization)?". If we have an "oracle" that returns the optimal solution of an optimization problem in time polynomial in the input size, then an algorithm for the decision version can query the oracle for the

optimal solution, compute its cost c (assuming, of course, the c is easy to compute) and then output the answer by comparing the value of the cost to the constant (bound) associated with it. But this means that the decision problem is decidable is polynomial-time and, if the decision problem is NP-Complete, then this entails that $\mathsf{P} \neq \mathsf{NP}$. Therefore, we conclude that the optimization problem is at least as hard as the decision version.

For example, let us consider the optimization problem associated with the decision clique, which we call `MAX_CLIQUE`: Given a graph $G = (V, E)$, find a clique (complete subgraph) in G of maximum size. If there is a polynomial-time algorithm for solving the optimization `MAX_CLIQUE`, then an algorithm for solving the recognition `CLIQUE` would simply query the optimization algorithm for the max. clique, compute its cost, and say "yes" if cost $\geqslant B$, and otherwise reject and say "no".

Conversely, if an algorithm exists for deciding the recognition version of an optimization problem, then an algorithm for solving the optimization problem can use the decision version decider as a subroutine and compute the optimal solution by dynamic programming techniques. We will abuse notation and denote NP-*optimization* problems whose decision versions are NP-Complete as NP-Hard.

2.1.3 Approximation Algorithms

It is highly believed that, given the conjecture that $\mathsf{P} \neq \mathsf{NP}$, no efficient, polynomial-time algorithms for solving NP-Hard optimization problems will ever exist. Therefore, one sacrifices the exactness of the returned solution and settles for algorithms that produce feasible sub-optimal, but provably near-optimal solutions, which run in time polynomial in the size of the input.

We make the notion of approximation algorithms for NP-*optimization* problems precise by introducing some definitions (see Vazirani [54]). Let Π be a minimization problem, and let $\delta : \mathbb{N} \to \mathbb{Q}^+$, be a function, where $\delta \geqslant 1$. We say that algorithm A is a δ-*factor approximation algorithm* for problem Π if, on each instance $I \in \Pi$, A produces a feasible solution solution f for I

such that

$$c(f) \leqslant \delta(\texttt{SIZE}(I))\textsc{Opt}(I) \qquad [\delta \geqslant 1, \text{ minimization}],$$

and the running time of A is bounded by a polynomial in $\texttt{SIZE}(I)$. The definition for maximization problems is the same as that of minimization problems, except that we require that $\delta \leqslant 1$ and that

$$c(f) \geqslant \delta(\texttt{SIZE}(I))\textsc{Opt}(I) \qquad [\delta \leqslant 1, \text{ maximization}].$$

We see that, for both minimization and maximization problems, the closer δ is to 1, the better the quality of the solution produced by A.

Approximation algorithm design techniques vary greatly among problems, and problems that are seemingly related might have inherently different approximability characteristics. For example, one might ask the question: since NP-Complete problems are polynomial-time reducible to each other, can a δ-factor approximation algorithm for one NP-Complete problem be used to approximate other NP-Complete problems with the same approximation factor? The answer to this question is NO, because not all polynomial-time reductions preserve the approximation factor. Consider, for example, the following NP-Hard optimization problems

- Maximum Independent Set `MAX_IND_SET`: Given a graph $G = (V, E)$, find an independent set of vertices of maximum size. An independent set in a graph is a set of vertices, no two of which are adjacent.

- Minimum Cardinality Vertex Cover `MIN_VERTEX_COVER`: Given a graph $G = (V, E)$, find a subset of vertices $V' \subseteq V$ that covers all edges in G, i.e., every $v' \in V'$ is incident on at least one $e \in E$, and such that V' has minimum size.

There exists a polynomial-time reduction `MAX_IND_SET` $\leq_p$ `MIN_VERTEX_COVER` with regard to exact solution: since no edge is common between any nodes in an independent set, the remaining vertices not in the independent set must touch all edges at least once in G and thus form a vertex cover; therefore, the

minimum vertex cover is the complement of the maximum independent set. Denote as Vc the maximum independent set in G and let Is denote the minimum vertex cover in G. Thus $\text{Vc} = V \setminus \text{Is}$ and therefore $|\text{Vc}| = |V| - |\text{Is}|$. As an illustration of the phenomenon above, consider an input graph $G = (V, E)$ with $|V| = 1000$ vertices. If $|\text{Is}| = 510$, then $|\text{Vc}| = 490$. Suppose that we have a 2-approximation algorithm for `MIN_VERTEX_COVER`. Let Vc', Is' denote the sets containing the approximate vertex cover and independent set in G, respectively. According to the values above, an application of the approximation algorithm for `MIN_VERTEX_COVER` might result in $|\text{Vc}'|$ getting as large as 980 in the worst case. If we apply the reduction described above, then we get $|\text{Is}'| = |V| - |\text{Vc}'| = 20$, which is indeed a sever deterioration compared to the optimal value of 510. As a matter of fact, $|\text{Is}'|$ approaches 0 in the worst as $|\text{Vc}|$ approaches $|V|/2$. Therefore we see that having an approximation algorithm for either `MAX_IND_SET` or `MIN_VERTEX_COVER` cannot be used directly to solve the other problem by the simple reduction, and thus different techniques might be needed for different problems.

Some problems cannot be approximable to within any δ. One example is the `TSP` in its full generality: A δ-factor approximation algorithm for `TSP`, for any δ, can be used to decide the Hamiltonian Circuit problem, which is NP-Complete, thus showing that P = NP ! Once the distance matrix $[d_{i,j}]$ is equipped with a metric space, then the triangle inequality automatically holds and we get the metric TSP (Δ-`TSP`) problem, for which Christofides[17] showed a 3/2-approximation algorithm. On the other hand, some problems allow approximability to any required degree. Let Π be an NP-Hard optimization problem. Let $\epsilon > 0$ be an error parameter that specifies the desired trade-off between the accuracy (quality) of the solution and the running time of the algorithm. We say that an algorithm A_ϵ is an *approximation scheme* for Π if, given as input (I, ϵ), where $I \in \Pi$ is an instance of Π, then A_ϵ produces a solution f such that

- $c(f) \leqslant (1 + \epsilon)\text{Opt}$, if Π is minimization;
- $c(f) \geqslant (1 - \epsilon)\text{Opt}$, if Π is maximization.

The bounds above say that the relative error of the value of solution

produced by A_ϵ with respect to the value of the optimal solution should be bounded above by at most ϵ

$$\frac{|c(f) - \textsc{Opt}|}{\max(\textsc{Opt}, c(f))} \leqslant \epsilon.$$

Since in minimization problems $c(f) \geqslant \textsc{Opt}$, we require that

$$\frac{c(f) - \textsc{Opt}}{c(f)} \leqslant \epsilon,$$

from which we get

$$c(f) \leqslant \frac{1}{1-\epsilon}\textsc{Opt} \leqslant (1+\epsilon)\textsc{Opt}.$$

Similarly for maximization problems, $c(f) \leqslant \textsc{Opt}$, so we require that

$$\frac{\textsc{Opt} - c(f)}{\textsc{Opt}} \leqslant \epsilon,$$

from which we get

$$c(f) \geqslant (1-\epsilon)\textsc{Opt}.$$

If A_ϵ runs in time polynomial in $\texttt{SIZE}(I)$, then A_ϵ will be called a *Polynomial-Time Approximation Scheme* (PTAS). A PTAS is not restricted to run in time polynomial in $1/\epsilon$. For example, the functions $(1/\epsilon)^3 n^4$ and $n^{3^{1/\epsilon^2}}$ are allowable running times of PTASs. Such running times make PTASs of theoretical significance only, but they might be doomed to perform poorly in practice (e.g., setting $\epsilon = 0.1$ in $n^{3^{1/\epsilon^2}}$). If A_ϵ is required to run in time that is polynomial in both $\texttt{SIZE}(I)$ and $1/\epsilon$, then A_ϵ becomes a *Fully Polynomial-Time Approximation Scheme*. Examples of running time functions for FPTAS include $(1/\epsilon)^3 n^4$ and $n^{5/2}(\log(1/\epsilon))^7$.

An FPTAS is the best one can hope (both practically and theoretically) for an NP-Hard optimization problem modulo the conjecture $\mathsf{P} \neq \mathsf{NP}$. One such problem that admits an FPTAS is the knapsack problem. Consider the optimization integer knapsack problem INT_KNAPSACK: Given an integer K and n objects having values $v_1, \ldots, v_n$ and weights $w_1, \ldots, w_n$, find integers

$x_1, \ldots, x_n$ such that $\sum_{i=1}^{n} v_i x_i$ is maximized and such that the constraint $\sum_{i=1}^{n} w_i x_i \leqslant K$ is satisfied. `INT_KNAPSACK` is NP-Hard, since it decision version is NP-Complete, and it was shown to be NP-Hard by a polynomial-time reduction from the partition problem (see Papadimitriou and Steiglitz [45] chap. 15), which we define as follows. `PARTITION`: given integers $c_1, \ldots, c_n$, is there a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{j \in S} c_j = \sum_{j \notin S} c_j$?

Luckily enough, there is a dynamic programming-based, pseudo-polynomial time algorithm that solves `INT_KNAPSACK` exactly in $\mathcal{O}(n^2 c)$, where c is the value of the optimal cost. The value c that appears in the running time need not be polynomially bounded in the dimension of the problem, namely n, and hence the "pseudo"-polynomiality. However, the mere existence of a pseudo-polynomial time algorithm for `INT_KNAPSACK` is the key for obtaining an FPTAS for it. In general, dynamic programing techniques are usually employed to derive approximation schemes for optimization problems, although the running time of the resulting algorithms can be prohibitively large to be used in practice. We will rely on a special dynamic programming formulation later to derive an FPTAS for the problem concerning energy minimization on multiprocessor platforms (chapter 4).

2.2 Related Work

The problem of task allocation with service classes and QoS constraints is closely connected to several well-studied problems, particularly bin packing. de la Vega and Lueker [21] and Karmarkar and Karp [33] have developed approximation schemes for bin packing. Many variants of the bin packing problem have been studied, including bin packing with variable bin sizes (Friesen and Langston [23]). The class constrained bin packing problem deals with bins with C compartments Shachnai and Tamir [51]. The goal is to find a bin packing such that there are no more than C different classes of items in a bin.

The close connection between bin packing and partitioned scheduling for real-time tasks on multiprocessors has been explored by Lopez, Diaz, Garcia, and Garcia [43], who identified utilization bounds for partitioned

multiprocessor scheduling with EDF.

The closest effort related to our work is that of Lee, Lehoezky, Rajkumar, and Siewiorek [36] on QoS optimization with discrete QoS options. The resource allocation methods described by Lee, Lehoezky, Rajkumar, and Siewiorek is applicable to the problem described in this article but the methods with good approximation ratios have pseudo-polynomial runtime complexity while our algorithms are polynomial-time approximations. Lee, Lehoezky, Rajkumar, and Siewiorek [36] developed a framework for optimizing QoS with discrete settings in real-time environments from the *end users'* perspective. The authors considered QoS quantitatively and presented resource planning algorithms for multiple applications, multiple resources (disk, network, processor, etc.) and multiple QoS dimensions, so as to maximize the *utility* of end users. Their best algorithm in terms of performance is a PTAS for solving the *single resource multiple QoS dimensions* problem (we look at the case of multiple processors or resources and one QoS dimension). Their algorithm is based on dynamic programming, and produces a solution that is at most $(1 - \epsilon)$ far from optimal. Further, the authors developed a user friendly interface through which end users can specify their QoS requirements by choosing a parameterized utility curve that best matches their needs.

The *imprecise computation* model (Liu, Lin, Shih, Yu, Chung, and Zhao [41]) divides the execution requirement of a job into *mandatory* and *optional* cycles. A task should be fully granted the execution of its mandatory requirements before it reaches its deadline, whereas the optional execution is a reward that the task receives depending on the state of the system. Accordingly, the reward is an increase in the precision of computation (e.g., increased frame rate in video streaming applications), and the goal is to maximize the reward (the amount of optional part executed), or minimize the error (the amount of unfinished optional part) in a dual sense. Khemka, Shyamsundar, and Subrahmanyam [34] developed an optimization scheme for reducing error when imprecise tasks are scheduled on a multiprocessor system. In their model they assumed continuous job sizes and convexity conditions on the error functions, and they permitted jobs to start on one

processor and then migrate to other processors. In the problems described in this article, utilization is varied in discrete steps, and each step is associated with a quality index. This approach offers some new ideas for task allocation.

The *Increasing Reward with Increasing Service* (IRIS) model (Dey, Kurose, and Towsley [22]) does not require the execution times of tasks to be known apriori. A task receives as much execution time (value) as possible before it deadline expires, with no upper limits on the obtainable reward. Both the imprecise computation and IRIS models assume that the reward function is a non-decreasing concave function.

We have presented a simple model for a service class. Lee, Shih, and Sha [37] used the term service class to refer to a pre-determined resource allocation in a problem related to radar systems. A service class in their work refers to some allocation of resources that pertain to certain situations (heavy load, light load, etc.), and precomputation allows them to perform online optimization depending on the system load.

As energy minimization is concerned, a vast body of research is dedicated for investigating possible techniques for saving energy on computing systems. Approaches range from designing low power hardware, or that with power management capabilities, to designing energy-aware scheduling algorithms.

On one hand, energy minimization can be achieved by means of *Dynamic Frequency/Voltage Scaling*, which is the most common technique used to minimize the energy consumption of processors in an online fashion. On the other hand, *Power Management* schemes aim at finding a suitable assignment of speeds to processors given prior knowledge of task parameters at design time, i.e. before tasks start execution.

In their work, Irani, Shukla, and Gupta [30] considered uniprocessor systems, and assumed that the power function $P(s)$ is continuous and convex with respect to the speed s. Further they assume that speed is a continuous function of time $s(t)$ with no upper limit. The authors defined the *critical speed*, which is the speed at which the energy function $P(s)/s$ is at minimum. They developed an offline approximation algorithm that runs the processor at the critical speed and provided a lower bound of 3 from

optimal. Moreover, the authors studied the gain of turning the processor to the dormant mode if it becomes idle, by incorporating the wakeup switching overhead to the analysis. They concluded that it is rewarding to do so if the processor's idle period is no less than the *break-even time*, which is equal to the ratio between the energy consumed in waking up from the sleep mode to the power consumption at the minimum frequency. Otherwise tasks can be run at a higher speed to create longer idle periods, where turning the processor to the dormant mode becomes beneficial. The authors also designed an online algorithm with an optimized competitive ratio of 193 for a cubic power function. The impracticality of their solution lies in their assumptions about the continuity of speed and power.

One of the significant results is that by Ishihara and Yasuura [31], where the authors considered processors with discretely variable voltages. They showed that a single voltage alternation can bring the energy expenditure to a minimum. Specifically, for an ideal system with continuous voltage levels, they found out that the voltage that brings the execution time of a task to exactly the deadline is the voltage that minimizes the energy expenditure. If that voltage is not available on the system, then its two immediate neighbors can minimize the energy consumption.

Chen [16] proposed DVS-based algorithms for minimizing the *expected* energy expenditure of frame-based tasks on uniprocessors with discrete frequencies. Their approach is probabilistic and required a discrete probability distribution of task execution cycles. The authors further considered tasks with different power characteristics. They defined an *operating point* to be the normalized energy at a certain frequency. For a set of operating points per task, the authors applied a convex hull algorithm to eliminate energy-inefficient operating points and produce a set of *usable points* that forms a convex set. Afterward, they greedily changed the operating frequency of the processor at certain points starting from the highest frequency in the ordered set of usable points. This intuitively means that the processor slows down as a task executes more.

Based on the work above by Ishihara and Yasuura [31], Bini, Buttazzo, and Lipari [11] also incorporated the findings of Seth, Anantaraman,

Mueller, and Rotenberg [50] (about how tasks' WCET scale with speed) and considered task execution cycles that do not scale with speed (spent waiting for device bus). They devised algorithms for scheduling on a single processor with discrete frequency steps to minimize energy.

Most recently, Yang, Chen, Kuo, and Thiele [60] proposed an FPTAS for scheduling real time tasks on multiprocessor with discrete speeds. They used the trimming-the-state-space technique that we used in our work. The authors assumed that the WCET of tasks are given at the highest speed $s_j^{\max}$ and scaled the WCET linearly as speed changes. According to their energy model, the minimum energy is achieved by operating each processor at speed equal to $U_j s_j^{\max}$. If this speed is not available on the processor, then they chose its immediate neighbors $s_{j,k}$ and $s_{j,k+1}$ such that $s_{j,k} < U_j s_j^{\max} < s_{j,k+1}$. The energy in their case is the value of the function resulting from the linear interpolation of the points $(s_{j,k}, tP_j(s_{j,k})), (s_{j,k+1}, tP_j(s_{j,k+1}))$ at $U_j s_j^{\max}$, where t is the interval of energy measurement. This assumes that energy is a linear function of the speed when the workload is fixed, falling in the subtlety that the WCEC of tasks remains constant with speed. Moreover, the authors do not consider leakage power consumption in their model.

Rusu, Melhem, and Mossé [49] considered a restricted version of the problem we study: their goal was to maximize the reward of real-time tasks executing on a uniprocessor system, subject to an energy budget that the system should never exceed. The authors, however, assume that the processor speed and the power function associated with the speed are continuous.

Chapter 3

On Task Allocation Problems with Service Classes

3.1 Motivation: Virtual Machine Allocation

Standard On-Demand Instances	Linux/UNIX Usage	Configuration
Small	0.10 per hour	1.7 GB memory, 1 EC2 compute unit
Large	0.40 per hour	7.5 GB memory, 4 EC2 compute units
Extra Large	0.80 per hour	15 GB memory, 8 EC2 compute units
High CPU On-Demand Instances	**Linux/UNIX Usage**	**Configuration**
Medium	0.20 per hour	1.7 GB memory, 5 EC2 compute units
Extra Large	0.80 per hour	7 GB memory, 20 EC2 compute units

Table 3.1: Amazon EC2 Instance Pricing and Configuration. (1 EC2 Compute Unit Provides CPU Speeds of About 1 GHz.) (Amazon.com [5])

The initial motivation for this study is resource allocation and control in virtualized environments. Many computing applications are moving to consolidated hosting setups. These services are hosted by the providers using virtual machine technology (an example is VMWare's ESX Server [55] or the Xen hypervisor [9, 46]). There are several reasons for choosing virtualization as an enabler for such computational clouds: performance isolation, security, management ease and flexibility for users. Of particular interest is performance isolation. End-users can instantiate virtual machines

for their applications and provision these virtual machines with the resources needed to achieve performance goals.

The ability of an application/service to respond to requests or perform its tasks within some time constraints depends on the amount of resources allocated to the application. The resources could be the amount of processing time, the amount of memory or the fraction of the network data rate. Often the primary resource being controlled is the fraction of processing time allocated to an application. In a virtualized setting, several virtual machines share the same physical hardware and access to the hardware resources is controlled by some time sharing mechanism (Figure 3.1). The constant bandwidth server mechanism [1], for example, can be used to guarantee a slice of a shared processor to a virtual machine.

Controlling performance in a virtualized hosting environment requires detailed performance monitoring. Such monitoring will determine the relationship between resource allotment and quality of service(Urgaonkar, Shenoy, and Roscoe [53]). This information will then be useful in improving resource allocation and virtual machine migration in systems such as Sandpiper (Wood, Shenoy, Venkataramani, and Yousif [59]). The Sandpiper system does in fact find heuristic solutions to multi-capacity bin packing problems for deciding upon resource management, and we believe our methods can be integrated into such a system easily. Moreover, the set of service classes can evolve over time to reflect the most recent observations correlating resource allocation and performance goals.

This approach to resource allocation is also applicable to cloud computing, which also uses virtualization as a substrate for offering infrastructure services. Many computing systems users have shifted hardware maintenance and management to third-parties such as Yahoo!, Google and Amazon [5]. This model of computational service hosting permits organizations and individuals to deploy large-scale services and pay only for the marginal cost of resource usage. The cloud service operator can allocate multiple virtual machines to the same hardware unit as long as resources are not overloaded. Application developers who choose to utilize the computational clouds can provision their virtual machines for optimal performance. Cloud service

providers offer multiple tiers of service with graduated pricing. Amazon Web Services, for example, offers several choices for virtual machine sizing and pricing in their Elastic Compute Cloud service (Table 3.1 [5]). Service classes can be associated with pricing details and applications can be deployed across multiple virtual machine instances to meet performance goals while minimizing the cost of launching compute instances. This example further emphasizes the use of the service class model and the potential applicability of our main results.

Quality-of-service is almost always associated with certain cost/precision trade-offs. For example, consider a search engine where the back-end, having received a user query, searches its index database for matching documents, ranks those documents (using some ranking algorithm) and returns the top N documents in ranked order (Baek and Chilimbi[8]). The service provider might consider, for example, executing less cycles running the ranking algorithm for the sake of faster response, in addition to saving energy on the search engine servers. The consequences of such *approximation* of the output of computation become relevant when the QoS metric is clearly defined. We can define the QoS loss metric as the percentage of user requests, compared to the full fledged-service case, that either return the same top N documents but in a different rank order or return different top N documents. Towards the goal of practically enabling such approximations by systematic means, Baek and Chilimbi[8] developed a programming framework called "Green", that allows programmers to approximate loops and expensive functions, and provides statistical QoS guarantees.

3.2 Definitions and Formal Problem Formulation

3.2.1 Definitions

Given some computational tasks, and a set of service classes for each task, and an unlimited pool of processors, we would like to find a) an assignment of tasks to service classes, and b) an assignment of tasks to processors such that

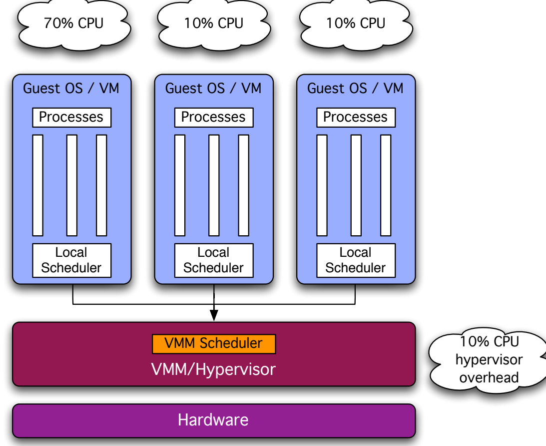

Figure 3.1: Hierarchical Scheduling Using a VMM/Hypervisor

- no unit is over-utilized,
- the aggregate quality of service requirements are met, and
- we use as few a number of processors as possible to achieve the goals.

Definition 1 (Resource Set). *The set of resource types, denoted* $R := \{x_1, x_2, \ldots, x_r\}$*, is called the resource, where* $r := |R|$*.*

CPU, memory, storage space and network bandwidth are typical examples of resource types. A *processor* is a realization of a certain resource type. In our setting, a processor π_k is a resource supply that provides fractions of certain resource types; $\pi_k = \langle y_1, \ldots, y_r \rangle$, $0 \leq y_j \leq 1$ is the amount of resource x_j that can be supplied by the processor.

Definition 2 (Service Class). *A service class is a tuple*

$$sc_k = (\pi_k, \langle (u_{k,1}, q_{k,1}), \ldots, (u_{k,r}, q_{k,r}) \rangle),$$

where $u_{k,j}, q_{k,j} \in (\epsilon, 1]$.

A service class represents some fraction of a resource supplied by a processor and the associated quality (as the minimum fraction of the maximum

quality of service) when these resources are allocated to the relevant task. Our definitions provide a very general description of processors and service classes. In this article we shall primarily discuss the case of identical processors with one resource (CPU). Therefore, we drop the processor index π_k in the service class description, and thus we are left with a single pair (u, q) specifying a service class.

There is no loss of generality when we assume $q \in (\epsilon, 1]$; we can scale all quality metrics to this set. If (u, q) is a service class for some task then q is a scalar that represents that perceived quality of service when the task is guaranteed a resource utilization u.

The reason that both the utilization and the quality index are bounded from below is two-fold. No application has arbitrarily low utilization; there is usually a lower bound to account for context switch and other overheads. Similarly, any task, when it runs, improves the QoS by some amount that is not arbitrarily small. Later we will discuss the implications of this problem setting and propose methods to remove these restrictions for further generalization.

Tasks: A task is an entity that requires computational resources and $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denotes the set of tasks. The number of tasks is $|\Gamma| = n$. Each task τ_i is associated with a service class set $SCS_i = \{sc_{i,j} = (u_j, q_j)\}$ that consists of service classes. The service class set is essentially a discrete function that maps utilizations of a task quality of service values. We do not make assumptions on how exactly the quality of service index changes with respect to changes in the utilization, but rather we assume an arbitrary utilization-to-quality of service function, with some structure (below).

Increased service, increased quality: We will make a natural restriction that for all service classes associated with a particular task, an increase in any resource dimension implies an increase in the quality index.

3.2.2 Problem Definition

Our definitions provide a very general description of processors and service classes. In this article we shall primarily discuss the case of identical pro-

cessors with one resource (CPU). Later we shall briefly discuss the multiple resource case. We leave problems related to variable capacity processors to a future article.

Task allocation with service classes (TASC): Given a set of tasks Γ and a service class set for each task, we seek an algorithm that selects, for all i, a service class for τ_i from SCS_i, and then maps the tasks to processors without over-utilizing any of the processor's resources.

Further, we would like to minimize the number of processors used, m, subject to $\sum_{i=1}^{n} q_i \geqslant Q$, where q_i is the quality index assigned to τ_i and Q is the aggregate QoS goal.

3.2.3 Problem Hardness

The problem of interest, TASC, is NP-Hard in the strong sense (Papadimitriou [44]); the simplest case of exactly one resource attached to a processor, equally priced and equal capacity processors, and a default service class for each task with no penalty, reduces the problem to a bin packing problem, which is NP-hard in the strong sense.

For problems that are NP-Hard in the strong sense, we cannot find a polynomial time algorithm that approximates the optimal solution unless P = NP (Garey and Johnson [25]). The usual measure for understanding the performance of algorithms for NP-hard problems is the (asymptotic) approximation ratio. If algorithm A solves, approximately, an NP-hard problem, then its approximation ratio is defined as

$$R_A := \limsup_{n \to \infty} sup_I \left\{ \frac{A(I)}{\textsc{Opt}(I)} : \textsc{Opt}(I) = n \right\},$$

where I is the input to the algorithm, i.e., I is an instance of the NP-hard problem.

Despite the above hardness results, there are in fact practical restrictions of the the TASC problem that permit polynomial-time approximations. In this article we describe an algorithm for solving a restricted, but practical, case of TASC. For the related bin packing problem, de la Vega and Lueker [21] showed that there exists a polynomial time algorithm such that

$A_{dL}(I) \leq (1+\epsilon)\text{OPT}(I) + 1$ for all instances I of the bin packing problem. Karmarkar and Karp [33] showed the existence of a polynomial time algorithm such that $A_{KK}(I) \leq \text{OPT}(I) + O(\log^2(\text{OPT}(I)))$ for all bin packing problem instances I.

3.2.4 Restricted Problem Formulation

The hardness of TASC leads us to first consider some restricted cases and obtain some insight. For the general case we develop heuristics based on our understanding of the restricted cases.

1. **Minimal service class** (Problem MSC): In this variation, there is a minimal service class (u_{min}, q_{min}). There may be many service classes but they are all associated with resource utilization greater than u_{min} and with a quality score greater than q_{min}.

2. **Fixed service classes** (Problem FSC): This is the simplest variation where there is a finite number of priced service classes K (that the service provider offers), and each task is assigned to one of these K service classes.

We start with the simplest case, FSC, and proceed to identify polynomial time algorithms for MSC. We shall then provide some comments about the unrestricted TASC problem.

3.3 Solving Restrictions to TASC

We use the following intuition in designing algorithms for solving the restricted versions of TASC described in Section 3.2.4. An approximate solution to the bin packing problem can be found in polynomial time. The key step towards solving TASC then is to enumerate assignments of tasks to service classes. If each task has one of K choices for a service class, we may have to examine K^n possible service class assignments before we find an optimal solution. We need to avoid the exponential space search and examine only $p(\texttt{SIZE}(I))$ assignments of tasks to service classes where p is

a polynomial function of the size of the input instance. We can then solve the bin packing problem for each of these assignments and identify the least cost assignment of service classes and mapping of tasks to processors.

To solve the bin packing problem, we do not apply the methods developed specifically for the bin packing problem. Instead we rely on an approximation algorithm for the minimum makespan problem, where a problem instance consists of m parallel machines and a set of independent jobs that needs to execute on these machines and the goal is to minimize the time taken to complete all jobs (the makespan). This problem can be solved to within $(1+\epsilon)$OPT in polynomial time (Hochbaum and Shmoys [27]). In our case, we will treat a bin packing problem as a minimum makespan problem, vary the number of processors from 1 to n (number of tasks), and use the polynomial time algorithm to determine if there is some schedule with makespan $1+\epsilon$. From that solution we can retrieve the solution for the bin packing instance trivially. The minimum number of processors needed to obtain a solution with makespan of $1+\epsilon$ or less is the optimal solution for the bin packing (task allocation) problem; since the optimal makespan could be $1+\epsilon$ we may have to use processors with capacity $1+\epsilon$.

3.3.1 Fixed Service Classes

Let us consider the case when the number of service classes is fixed as K. We first need to find an assignment of tasks to service classes. Let us suppose that n_j denotes the number of tasks assigned to sc_j. We need to find a vector $\langle n_1, n_2, \ldots, n_K \rangle$ such that

$$n_1 + n_2 + \cdots + n_K = n,$$

and a feasible allocation of tasks to processors such that no unit is overloaded, the resulting total quality index is above the required threshold and the number of processors used is minimized.

We know, using an elementary combinatorial argument, that there are $\binom{n+K-1}{n}$ integral vectors that satisfy the equality $\sum_{j=1}^{K} n_j = n$ (this bound can be interpreted as the number of ways n indistinguishable balls can be

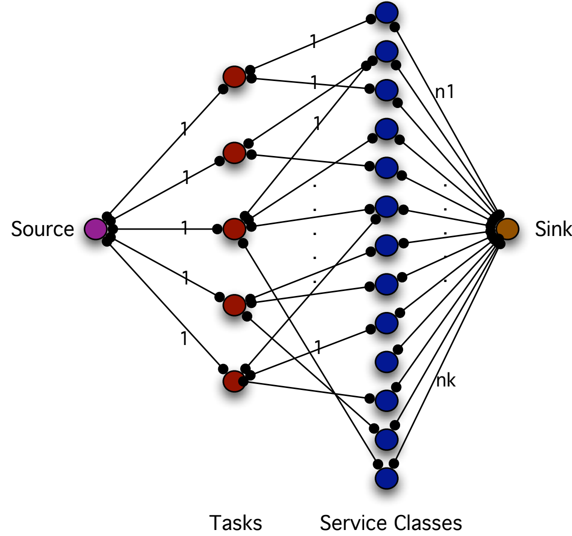

Figure 3.2: Network Flow and Feasible Assignments

packed into K bins). For fixed K, the number of assignments of tasks to service classes is a polynomial in n. For each of these assignments, we can solve a bin packing problem, in polynomial time, to find a good allocation of tasks to processors. There is, however, one intermediate step to consider.

We need to verify that a vector $\langle n_1, n_2, \ldots, n_K \rangle$ is indeed a feasible assignment of tasks to service classes. There may not exist a set $\{\tau_i\}$ with cardinality n_j (for some value of n_j) such that $sc_j \in SCS_i$. For this verification, we represent tasks and service classes as vertices in a bipartite graph and, interestingly, perform network flow computations to solve the subproblem. Consider a bipartite graph with a vertex for each task and a vertex for each service class. An edge with capacity 1 exists between a task and the service classes that are appropriate for that task. Add a source, and connect the source to each task with an edge of capacity 1. Add a sink and connect service class sc_j to the sink with an edge of capacity n_j. A maximum flow of n exists iff $\langle n_1, n_2, \ldots, n_K \rangle$ represents a feasible assignment. The maximum flow in a network can be computed in polynomial time

Algorithm 1: Complete algorithm for FSC

for all integral vectors $\langle n_1, n_2, \ldots, n_K \rangle$ such that $\sum_{j=1}^{K} n_j = n$ **do**
  verify that $\langle n_1, n_2, \ldots, n_K \rangle$ is a feasible assignment of tasks to service classes using a polynomial time max flow algorithm.
  if $\langle n_1, n_2, \ldots, n_K \rangle$ is feasible **then**
    solve the task allocation problem using the Hochbaum-Shmoys minimum makespan algorithm (Hochbaum and Shmoys [27]) at most n times
  end if
end for
return the solution that uses the smallest number of bins and satisfies $\sum_{i=1}^{n} q_i \geq Q$.

(Ahuja, Magnanti, and Orlin [3]) to verify feasibility and obtain the service class assignments. (Figure 3.2 illustrates the bipartite graph representation of this subproblem.)

When the number of service classes is fixed for all TASC problem instances, and given an assignment of tasks to service classes, the subproblem of allocation tasks to processors using the minimum number of units is no longer NP-Hard, and the optimal solution may be found in polynomial time using various techniques including dynamic programming.

Each subproblem of the FSC can be solved in polynomial time; we therefore have a polynomial time algorithm (Algorithm 1) to find the optimal solution to this problem.

The above discussion helps us state the following lemma. (The proof naturally follows from the discussion.)

Lemma 1. *An optimal solution to FSC can be found in polynomial time.*

3.3.2 Minimal Service Class

We shall now leverage the algorithm just described to solve the problem when service classes are part of the input instance but there is a particular minimum service class (u_{min}, q_{min}). Hochbaum-Shmoys' PTAS for the minimum makespan problem requires that item sizes be chosen from a fixed

set. In our setting, both u and q are rational numbers lower bounded by ϵ and upper bounded by 1, thus they may assume an infinite number of values. In order to bound the number of possible utilization/reward values, we use a chosen parameter ϵ, $0 < \epsilon < \min\{u_{min}, q_{min}\}$, and then quantize the resource utilization and the service quality. For service class $sc_k = (u_k, q_k)$, we round up the resource utilization and service quality to the nearest pair of the form $(\epsilon + \alpha\epsilon^2, \epsilon + \beta\epsilon^2)$ for some non-negative integers α, β. Setting $\alpha = \lceil \frac{u_k - \epsilon}{\epsilon^2} \rceil$ and $\beta = \lceil \frac{q_k - \epsilon}{\epsilon^2} \rceil$ satisfies the quantization requirements.

Let S_α, S_β denote the sets containing values that α, β may assume, respectively. Since $u_k, q_k \leqslant 1$, it follows that both α and β can get as large as $\lceil \frac{1-\epsilon}{\epsilon^2} \rceil$. Accordingly, $S_\alpha = S_\beta = \{0, \ldots, \lceil \frac{1-\epsilon}{\epsilon^2} \rceil\}$. Therefore, following the quantization, there are at most $|S_\alpha|.|S_\beta| \leqslant \lceil \frac{1}{\epsilon^4} \rceil$ service classes.

We can find optimal solutions to instances of the quantized problem (qMSC) using processors with capacity $1+\epsilon$; the solution can be determined, in polynomial time, using the approach for FSC (Algorithm 1).

We now have the following lemma relating MSC and qMSC.

Lemma 2. *The optimal solution to an instance of qMSC uses at most the same number of processors as the optimal solution to the corresponding MSC instance, except that processors may need to be of capacity $1 + \epsilon$ for qMSC. The aggregate quality of the solution to qMSC is at most $(1 - \epsilon)Q$ if the corresponding instance of MSC has a feasible solution with quality of service score at least Q.*

Proof. Suppose that the optimal solution to an instance of MSC assigns some task τ to service class $(\epsilon + \alpha'\epsilon^2, \epsilon + \beta'\epsilon^2)$ and this task includes a service class $(\epsilon + (\alpha' - 1)\epsilon^2, \epsilon + \beta'\epsilon^2)$. The optimal solution to qMSC for this problem instance will assign τ to service class $(\epsilon + (\alpha' - 1)\epsilon^2, \epsilon + \beta'\epsilon^2)$, which leads to a decrease in total utilization. If the optimal solution to the MSC instance used m processors then the optimal solution to the qMSC instance will use $m' \leq m$ processors.

When we replace the original service classes by the quantized service classes, we see that on any processor used there may be up to $1/\epsilon$ tasks, and the resource utilization of each of these tasks can increase by ϵ^2 at

most. Therefore the maximum possible increase in resource utilization of an individual processor is ϵ; this increase can be satisfied by using a processor of capacity $(1+\epsilon)$.

The quality factor achieved by the optimal solution to an instance of qMSC is either Q or higher. If Q' is the quality factor achieved by qMSC, then the maximum increase in the quality score for each task is ϵ^2, which occurs when there are Q'/ϵ tasks, which is, of course, the maximum number of tasks. If we undid the quantization then the maximum drop is at most $Q'\epsilon$. The aggregate quality score will always be within a $1-\epsilon$ factor of the target quality score. □

By the analysis above, the algorithm might require the capacity of some processors to be augmented by at most ϵ times their maximum capacity for task allocation to be possible. For a processor running at a certain speed, say s, this processor might need to operate at a speed of $(1+\epsilon)s$ in the worst case. Practically, if s is the maximum speed at which the processor can run, then the operating speed of the processor cannot be increased and hence our task allocation scheme might not be possible to implement. One way to overcome this impracticality is to start operating the processor at speed $(1-\epsilon)s$ and then increasing its speed to its maximum s if required. Algorithmically, this amounts to necessity starting the algorithm with bin capacities of $1-\epsilon$ instead of 1, so that the capacity of any processor does not exceed unity as a result of our algorithm. This, however, might increase the number of bins required to pack all tasks compared to the $(1+\epsilon)$OPT upper bound guaranteed by the Hochbaum-Shmoys algorithm.

3.3.3 TASC

We now return to the general TASC problem. In the case when service classes can be arbitrarily small, it is possible to obtain service class assignments and processor allocations by separating tasks into large and small sizes (as is typically done to solve bin packing problems (de la Vega and Lueker [21])) using a parameter ϵ as the threshold. In the case of TASC, both the utilization and quality index can be large or small. Further, the

QoS constraint is disconnected from the optimality criterion that refers to the number of processors used. This is problematic because we are unable to bound the QoS loss when quality indices are small. If there was no lower bound on the quality index then the upper limit on the number of tasks with a low quality index is at least Q/ϵ, which is hard to bound.

For most practical settings there is a lower bound on the utilization that can be assigned to a task and on the quality of service attained when a task is assigned the smallest utilization quantum. For that setting, MSC, we can find a solution in polynomial time for the smallest number processors using a small speedup of $1 + \epsilon$ and within a $1 - \epsilon$ factor of the desired QoS.

We shall now consider one other variant of MSC that is of practical interest.

3.3.4 Maximizing QoS with m Processors

In this variant of MSC, we consider a set of m processors and the service classes are defined as expected. The goal is to maximize the QoS that can be achieved with a limited number of processors. We can approach this problem exactly as we did the MSC. The only modification needed is to use the minimum makespan routine only once and with m processors (in Algorithm 1). We search for the assignment and allocation with the highest QoS. Since we have a minimal service class, we can also prove (as we did in Lemma 2) that the obtained QoS is within $1 - \epsilon$ of the optimal QoS.

3.4 Experimental Evaluation and Further Extensions

Although we have analyzed and shown that the algorithms for TASC and related problems run in polynomial time, they are tedious and do not lend themselves to efficient implementation. In this section we focus on a simple implementation choice that results in acceptable performance and makes the algorithms usable. The bin packing routine and the maximum flow routine are central elements that influence the performance of the algorithms for TASC variations.

We replaced the minimum makespan routine for finding a bin packing with an algorithm due to Korf [35] that performs optimal bin packing. This algorithm is not polynomial time but performs remarkably well in practice. For the maximum flow problem, we used an implementation of the highest label preflow-push algorithm (Ahuja, Kodialam, Mishra, and Orlin [4]).

Our experimental platform used an Intel Core Duo processor running at 2.8GHz with 4GB memory. We generated, at random, 100,000 test instances for each problem size. For large problem sizes (> 60 tasks, 12 service classes per task), our implementation found an optimal solution in less than 9 seconds for all problem instances. For small problem sizes (< 25 tasks, 10 service classes per task), our implementation found an optimal solution in less than 1 seconds.

Although we have not analyzed algorithms for task allocation on processors with multiple resources, the basic algorithms themselves are easy to extend. Obtaining approximation ratios, however, is challenging; Garey and Graham showed that with d-capacitated bins the approximation ratio for any reasonable algorithm (First Fit, Best Fit are examples) is at most $d+1$ (Garey and Graham [24]). Woeginger [57] has shown that there cannot be an asymptotic PTAS for the multi-capacity bin packing problem.

We implemented heuristics (Leinberger, Karypis, and Kumar [38]) for the multi-capacity bin packing problem. We were able to solve problems with 40 tasks and 3 resources in less than 3 seconds. (We did not run any larger size experiments.)

We have not tabulated all the results here but we believe that this quantitative evaluation demonstrates the feasibility of using service-class driven task allocation in many performance management scenarios.

Chapter 4

Energy Efficient Task Partitioning and Real-Time Scheduling on Heterogeneous Multiprocessor Platforms with QoS Requirements

4.1 Motivation

Prior attempts at solving variants of this problem (see Section 2.2 for a discussion of related work) assume that the worst-case number of execution cycles (WCEC) of a task is speed-independent. This assumption then implies that the worst-case execution time (WCET) of a task scales linearly with processor speed. The original assumption is inaccurate if a task accesses peripheral devices that run at speeds different from that of the processor. To illustrate this drawback in prior work and to motivate our work with discrete settings, we performed empirical studies using two applications from the MiBench embedded applications benchmark suite Guthaus, Ringenberg, Ernst, Austin, Mudge, and Brown [26]: *Basic Math* and *Fast Fourier Trans-*

form (FFT) (with 64 random sinusoids and 65536 samples). We executed the former once and the latter 1000 times on an AMD® Phenom™II X4 925 Quad Core Processor 2.8GHz, which has discretely variable speeds in the set $\{800\text{MHz}, 1.6\text{GHz}, 2.1\text{GHz}, 2.8\text{GHz}\}$ per core, and measured the execution time as well as the energy consumption for each application at each speed step. The Basic Math benchmark includes some I/O operations and we found that speed scaling does not result in corresponding changes in execution time (Table 4.1). For this benchmark, we also found that energy consumption increases with increases in speed. For the FFT benchmark, which does not include the same amount of I/O as the Basic Math benchmark, execution time decreased as we would expect with an increase in processor speed and it is energy-efficient to run this application at a higher speed (Table 4.2) . Our observations highlight the fact that variations in execution time and energy consumption are different for different applications. These variations are not easily captured by closed-form functions and require discrete modeling.

We present a system and task model (Section 4.2) that captures system implementations better than earlier models. We focus on determining static speed settings for processors and identifying a mapping between tasks and processors such that the the energy expenditure of the system is minimized and the timeliness requirements of tasks respected. In addition, the task allocation scheme should guarantee that the overall QoS satisfies a specified requirement (Section 4.3). Our contribution is a fully polynomial-time approximation scheme (Section 4.4) for a problem that can be shown to be NP-Hard.

Frequency (GHz)	Average Power (Watts)	Execution Time (sec)
0.8	86.5	32.34
1.6	89.9	31.7
2.1	94	31.61
2.8	100.2	31.75

Table 4.1: Basic Math Benchmark (Single Iteration)

Frequency (GHz)	Average Power (Watts)	Execution Time (sec)
0.8	84.3	1080.4
1.6	88.96	538.2
2.1	94.78	410.23
2.8	104.11	307.74

Table 4.2: Fast Fourier Transform (FFT) Benchmark (1000 Iterations)

4.2 System Model and Notation

4.2.1 Platform

We have K distinct physical processor types at our disposal, with an unlimited pool of processors of each type. We will use the notation π_k to refer to a processor type; therefore, the set $\Pi = \{\pi_1, \ldots, \pi_K\}$ includes all distinct processor types. Each processor type may allow for a discrete number of speed settings. Let $S_k = \{s_1, \ldots, s_{\lambda_k}\}$ be the set of distinct speed levels associated with processor type π_k, where $\lambda_k := |S_k|$. We use $\lambda_{\max}$ to denote the maximum number of speed settings that any processor type may permit, i.e., $\lambda_{\max} = \max_k\{\lambda_k\}$. We introduce the notion of a *logical processor type*, corresponding to a *unique* physical processor and speed pair; a physical processor running at a certain speed. by creating λ_k logical processor types for processor type π_k. Therefore a logical processor type is defined as $\pi'_k := (\pi_k, s_\ell) \in \{\pi_k\} \times S_k$. There are at most $\lambda = \lambda_{\max} \times K$ logical processor types. The set $\Pi' = \{\pi'_1, \ldots, \pi'_\lambda\}$ includes all logical processor types.

Our objective is to construct a platform composed of M heterogeneous physical processors, possibly with identical types, each operating at a certain speed (thus the platform consists of M logical processors). We denote such a platform composition as a *platform configuration*

$$C = \langle \pi'_1, \ldots, \pi'_M \rangle,$$

where $\pi'_j \in \Pi'$ and π'_js in C are not necessarily distinct. We can think

of a platform as having a fixed number, specifically M, processor "slots", disregarding the order in which processors are arranged. Thus we seek to fill these slots with "suitable" logical processors. Since we allow identical logical processor types in platform configurations, it follows that the number of possible assignments of logical processors to M slots is at most the number of solutions to the equation

$$m_1 + m_2 + \cdots + m_\lambda = M,$$

where m_j is a non-negative integer that represents the number of instances of logical processor type π'_j that a platform configuration includes. Therefore, we have at most $\Lambda = \binom{\lambda+M-1}{M}$ possible platform configurations. The latter bound can be interpreted combinatorially as the number of ways M indistinguishable balls can be packed into λ bins, since the order of processors in a platform configuration does not matter.

λ, M and K, and as a consequence Λ, are considered part of the platform model and are treated as constants.

4.2.2 Task and Scheduling Model

We consider a task set, Γ, with N periodic, implicit-deadline, real-time tasks such that $\Gamma = \{T_1, \ldots, T_N\}$. This task set needs to be executed on a suitable heterogeneous computing platform. Task T_i recurs every P_i time units and, when executing on logical processor type π'_k has a worst-case execution time of $e_{i,k}$. Since $e_{i,k}$ is specified per logical processor type, it is dependent upon the physical processor type and the frequency at which it is operating. For convenience we will treat P_j to be a positive integer. Correspondingly, the utilization of T_i when assigned to logical processor type π'_k is $u_{i,k} = e_{i,k}/P_i$.

In this discussion, for simplicity, we restrict our attention to the partitioned scheduling of real-time tasks on the heterogeneous multiprocessor platform with earliest deadline first scheduling at each processor. For a certain platform configuration, say C, $z_{i,j}$ is an indicator variable that is 1 if T_i is assigned to π'_j in C and 0 otherwise. All tasks satisfy their timing requirements if $U_j = \sum_{i=1}^{N} z_{i,j} u_{i,j} \leqslant 1$, for every π'_j in C, $j \in \{1, \ldots, M\}$.

We also note that this work easily applies to the situation when tasks are simply given a fraction of a processor's bandwidth, as may be the case when virtual machines are scheduled in a cloud computing environment.

4.2.3 Energy Model

To account for the energy consumed by the processor and by other associated units (memory, I/O, etc.), our energy model has an active-time energy component and an idle-time energy component. For each physical processor type π_k we have an idle time power consumption, $p_{idle,k}$ (which, of course, all of its λ_k logical processor types inherit) that needs to be accounted for every time unit that a processor is idle. For each logical processor type and task pair, (π'_k, T_i), we denote the energy consumed by executing an instance of T_i on π'_k as $W_{i,k}$. In its simplest form, $W_{i,k}$ can be defined as $W_{i,k} = u_{i,k}\tau p_k$, where τ is the interval of energy measurement and p_k is the dynamic power consumption of logical processor type π'_k. Given its dependence on the dynamic power consumption of a processor operating at a certain speed, $W_{i,k}$ can be interpreted as a penalty that task T_i incurs as it executes on logical processor type π'_k. We, however, do not restrict $W_{i,k}$ to be of the latter form. Since $W_{i,k}$ can include additional parameters that capture energy consumption at the system level, we treat $W_{i,k}$ as a single quantity.

The basic task set consists of periodic tasks and therefore it is sufficient to study and minimize the energy consumed in an interval of length $L = \text{LCM}(P_1, \ldots, P_N)$ because all activity repeats in an identical manner every L time units. This time interval consists of active intervals and idle intervals on the different processors. Depending on the platform configuration, C, and the mapping of tasks to processors, the energy consumed by the system in an interval of length L can be computed as

$$E_C = \sum_{i=1}^{N} \sum_{j=1}^{M} \frac{L}{P_i} z_{i,j} W_{i,j} + L \sum_{j=1}^{M} (1 - U_j) p_{idle,j}, \tag{4.1}$$

where the first term represents the active-time energy consumption and the

second term represents the idle-time energy consumption.

4.2.4 Quality-of-Service Model

We denote the quality of service derived (or delivered) by a task T_i when it is assigned to logical processor type π'_k using a real number $r_{i,k}$. We can also interpret $r_{i,k}$ as a reward obtained by the particular task-to-processor mapping. Our model allows for further generality by allowing the reward to depend on the logical processor type. The aggregate quality of service delivered by the system, or equivalently the reward obtained by the system, is a function of the different $r_{i,k}$ values. *We use a simple sum to describe the aggregate reward although other functions can be accommodated by our framework.*

4.2.5 Task Set Encoding

According to the system model above, we encode the requirements of each task T_i in a set of triples (options) O_i, where vectors in O_i are defined as $O_{i,k} = \langle \pi'_k, u_{i,k}, r_{i,k}, W_{i,k} \rangle$. Task T_i is said to be *defined on logical processor type* π'_k if there exists an integer k such that $\pi'_k \in O_{i,k}$. In other words, a task might not be allowed to execute on some logical processor types, for which its execution requirements will not be given. For an instance I of our problem, if we assume that rational numbers of the form $x = p/q$, $p, q \in \mathbb{Z}$, $q \neq 0$ are represented as a pair (p, q), then we will need $\mathcal{O}(\log p + \log q)$ space to encode x in binary. Denote as $\texttt{SIZE}(I)$ the number of bits required to represent instance I in binary (or more generally any combinatorial object). Then $\texttt{SIZE}(I) \leqslant N + \sum_{i=1}^{N} \log_2 P_i + \log_2 Q + \sum_{i=1}^{N} \sum_{k=1}^{\lambda} (\log_2 \pi'_k + \texttt{SIZE}(u_{i,k}) + \texttt{SIZE}(r_{i,k}) + \texttt{SIZE}(W_{i,k}))$, assuming $P_1, \ldots, P_N$ and Q are integers.

4.2.6 Notes Regarding Assumptions

Assumption 1. *Power ratings include leakage (static) power consumption.*

We assume that the leakage power consumption p_{leak} is implicit in both $W_{i,k}$ and p_{idle}. For instance, when a is processor idle, its power consumption can be modeled in its simplest form as $p_{idle} = \min_k\{p_k\} + p_{leak}$, where p_k

is the dynamic power rating at some speed, say s_k. Nevertheless, more sophisticated idle power-saving models are employed in current technologies (Intel® Core^{TM} i7 Series incorporates advanced idle modes [29]). This assumption, however, makes the model clearer and simplifies the analysis, without degrading the accuracy of the model.

This entails that no assumptions exist on how task utilizations scale with different speeds. Consider the relationship between the speed of the processor and the execution requirements of tasks. Suppose that some task is known to require a worst case utilization of u_k when running on a processor operating at speed s_k. If the task is to run on a different speed s_ℓ, where $s_k \neq s_\ell$, on the same processor, then a seemingly intuitive way to adjust the execution requirement of this task is to scale u_k linearly as $\hat{u}_k = \frac{u_k s_k}{s_\ell}$. The latter scaling is naïve for many reasons. First, assuming that the WCEC of a task is constant and does not change with processor speed, executing a task at a higher speed will shorten its utilization. It is evident from equation (4.1) that decreasing the utilization increases both the idle and the leakage power consumption. Second, the naïve approach assumes that the WCEC is constant with respect to speed. Indeed, this is inaccurate because a task might access other devices throughout the course of its execution, such as memory and disk. Such devices typically run at bus frequencies that might be divergent from the processor's frequency (usually constant or have limited configurable speeds compared to those of the processor). Therefore, if the device bus has constant latency, then increasing the processor speed will increase the discrepancy between the speed of the processor and that of the device bus. Accordingly, the task will spend more cycles waiting for the device (doing no useful work). Therefore, increasing the processor speed might in fact increase the WCEC required by a task and therefore increase the WCET overall (which is not what one would expect from increasing the speed of the processor). The latter situation becomes more noticeable if the task is I/O-bound. Since energy depends on the WCET, the energy expenditure might be significantly larger than that captured by the naïve method.

Finally, suppose that devices can operate in three states, namely the

active, *standby* and the *off* state. Further, assume that all devices that a task requires are turned on in the standby state at the instance the task starts execution. Slowing down the processor will cause devices to wait longer in the standby state to be accessed by tasks requiring them, thus increasing the device standby energy. If the task is I/O-bound, then the device idle energy might dominate the energy consumption of the task and the overall energy expenditure might increase.

Assumption 2. *Each processor adopts a uniprocessor scheduling policy that is capable of utilizing the processor up to* 100%*.*

An example of such a policy is *Earliest Deadline First* (EDF) (Liu and Layland [40]). This assumption, alongside the previous assumption, guarantee that each processor is a bin of unit capacity. To see this, let D_i, $D_i \leqslant P_i$, denote the *relative deadline* of task T_i. Let $\texttt{demand}(t) > 0$ be the worst case execution requirements of a set of N periodic tasks that arrive and must complete within a contiguous interval of length t. Baruah, Howell, and Rosier [10] derived the following expression for the execution-time demand of the task set on one processor

$$\texttt{demand}(t) = \sum_{i=1}^{N} \left\lfloor \frac{t + P_i - D_i}{P_i} \right\rfloor e_i. \tag{4.2}$$

Further, they derived sufficient and necessary conditions for the schedulability of N periodic tasks on one processor that employs EDF. Specifically, the task set is schedulable iff $\texttt{demand}(t) \leqslant t$. In our work we assume that tasks have *implicit deadlines*, i.e., $D_i := P_i$, and since our energy measurement interval is L, the schedulability condition on one processor becomes

$$\texttt{demand}(L) = \sum_{i=1}^{N} \left\lfloor \frac{L}{P_i} \right\rfloor e_i \leqslant L. \tag{4.3}$$

Moreover we have

$$\sum_{i=1}^{N} \left\lfloor \frac{L}{P_i} \right\rfloor e_i \leqslant \sum_{i=1}^{N} \frac{L}{P_i} e_i = L \sum_{i=1}^{N} u_i \leqslant L,$$

from which we get that $\sum_{i=1}^{N} u_i \leqslant 1$, which is our unit-capacity bin condition. Further, we assume that the uniprocessor scheduling policy is *work-conserving*, that is, the scheduler always dispatches a task that is ready for execution as soon as the processor becomes idle. The extension to other similar scheduling policies that provide utilization bounds is straightforward.

Assumption 3. *Problem settings are discrete.*

The set of processor speeds is arbitrarily structured, as well as the set of penalties incurred by different processor speeds. Further, we make no assumptions regarding the structure of the set of rewards obtained by tasks as they execute on different processors. More specifically, we neither make assumptions about the nature of the reward functions – whether they are uniform, concave, etc. – nor about the relationship of rewards to processors and speeds. In other words, tasks are heterogeneous with respect to their power characteristics; different tasks may be associated with different power functions.

4.3 Problem Formulation

For the system model above, we wish to establish appropriate pairings between (i) processors and speeds and (ii) tasks and processors, so as to satisfy the following requirements:

1. Each processor is assigned exactly one power rating (speed) for the whole duration of its operation;
2. The total energy expenditure of all processors is at a minimum;
3. A processor should never exceed its capacity;

4. Once assigned to a processor, a task should execute on that processor in its entirety;

5. For a certain aggregate quality of service score Q, a minimum guarantee at the system level should be achieved.

For a certain platform configuration $C = \langle \pi'_1, \ldots, \pi'_M \rangle$, we formulate the task assignment problem as a mathematical program as follows: Given a task set encoding $O = \{O_i\}_{i=1}^{N}$, we represent the task settings per C as follows:

Let $z_{i,j}$ be the following indicator variable:

$$z_{i,j} = \begin{cases} 1 & \text{if } T_i \text{ has been assigned to processor } \pi'_j, \\ 0 & \text{otherwise.} \end{cases}$$

Its associated binary $N \times M$ matrix, call the *assignment matrix*, is:

$$Z_C = \begin{pmatrix} z_{1,1} & \cdots & z_{1,M} \\ \vdots & \ddots & \vdots \\ z_{N,1} & \cdots & z_{N,M} \end{pmatrix}$$

The utilization matrix is given by:

$$U_C = \begin{pmatrix} u_{1,1} & \cdots & u_{1,M} \\ \vdots & \ddots & \vdots \\ u_{N,1} & \cdots & u_{N,M} \end{pmatrix}$$

where

$$u_{i,j} = \begin{cases} u_{i,k} & \text{if } \exists k \text{ s.t. } \pi'_j = \pi'_k \in O_{i,k}, \\ 0 & \text{otherwise.} \end{cases}$$

The reward matrix is:

$$R_C = \begin{pmatrix} r_{1,1} & \cdots & r_{1,M} \\ \vdots & \ddots & \vdots \\ r_{N,1} & \cdots & r_{N,M} \end{pmatrix}$$

where

$$r_{i,j} = \begin{cases} r_{i,k} & \text{if } \exists k \text{ s.t. } \pi'_j = \pi'_k \in O_{i,k}, \\ 0 & \text{otherwise.} \end{cases}$$

The energy matrix is given by:

$$\mathcal{W}_C = \begin{pmatrix} W_{1,1} & \cdots & W_{1,M} \\ \vdots & \ddots & \vdots \\ W_{N,1} & \cdots & W_{N,M} \end{pmatrix}$$

where

$$W_{i,j} = \begin{cases} W_{i,k} & \text{if } \exists k \text{ s.t. } \pi'_j = \pi'_k \in O_{i,k}, \\ 0 & \text{otherwise.} \end{cases}$$

Finally, define the idle power vector as:

$$P_{idle,C} = \begin{bmatrix} p_{idle,1} & \cdots & p_{idle,M} \end{bmatrix}^T$$

where $p_{idle,j} = p_{idle,k}$ if $\pi'_j = \pi'_k$ for some k. The mathematical program can be written as:

$$
\begin{aligned}
\text{minimize} \quad & E = \sum_{i=1}^{N}\sum_{j=1}^{M}\frac{L}{P_i} z_{i,j} W_{i,j} + L\sum_{j=1}^{M}(1-U_j)p_{idle,j} && (4.4)\\
\text{subject to} \quad & \sum_{j=1}^{M} z_{i,j} = 1 \qquad \forall i \in \{1,\cdots,N\} && (4.5)\\
& \sum_{i=1}^{N} u_{i,j} z_{i,j} \leqslant 1 \qquad \forall j \in \{1,\cdots,M\} && (4.6)\\
& \sum_{j=1}^{M}\sum_{i=1}^{N} r_{i,j} z_{i,j} \geqslant Q \\
& z_{i,j} \in \{0,1\} \qquad \forall i \in \{1,\cdots,N\}, \forall j \in \{1,\cdots,M\}
\end{aligned}
$$

where Q is the minimum aggregate QoS score that the system should achieve. Constraint (4.5) says that a task should be assigned to a single processor, and constraint (4.6) guarantees that a processor's capacity is not exceeded. Therefore, the goal is to find the matrix Z that represents an assignment of tasks to processors that minimizes the energy expenditure of configuration C.

Intractability: The solution points represented by matrix Z_C are required to be binary, for we do not split a task across processors, so fractional assignments are not of our interest. This renders our mathematical program computationally intractable. In fact, it is not difficult to show that this problem is NP-Hard by reduction from the PARTITION problem, which was shown by Garey and Johnson [25] to be NP-Complete. Therefore a polynomial-time approximation algorithm for solving this problem within a factor of the optimal solution might be our only resort unless P = NP.

4.4 Assignment of Tasks to Processors

We resort to a dynamic programming (DP) approach to solve this problem. We show that an exact DP formulation has complexity that is exponential in time and space requirements. We then use the exact DP to obtain a modified DP that is computationally tractable and is a fully polynomial-time approximation scheme (FPTAS).

4.4.1 Exact and Optimal Dynamic Program Formulation

For a certain platform configuration of the from $C = \langle \pi'_1, \ldots, \pi'_M \rangle$, we equip each task T_i with a set of states $\mathcal{S}_i = \{\varphi_1, \ldots, \varphi_{|\mathcal{S}_i|}\}$. Each state $\varphi \in \mathcal{S}_i$ is an M-dimensional vector that encodes a partial feasible schedule from T_1 up to T_i on every π'_j in C. More precisely, a state has the form

$$\varphi \doteq \langle (Y^{\varphi}_{i,1}, E^{\varphi}_{i,1}), \ldots, (Y^{\varphi}_{i,M}, E^{\varphi}_{i,M}) \rangle,$$

where, for every pair $(Y^{\varphi}_{i,j}, E^{\varphi}_{i,j})$, $j \in \{1, \ldots, M\}$, $Y^{\varphi}_{i,j}$ is a set that contains information about tasks assigned to processor π'_j, i.e., $Y^{\varphi}_{i,j} \doteq \{\langle T_\ell, u_{\ell,j}, r_{\ell,j} \rangle\}_\ell$ for every task assigned to π'_j, where $1 \leqslant \ell \leqslant i$. $E^{\varphi}_{i,j}$ is the energy expenditure of π'_j with respect to the schedule $Y^{\varphi}_{i,j}$. Let $U^{\varphi}_{i,j}$ be the total utilization (workload) on π'_j

$$U^{\varphi}_{i,j} = \sum_{\langle T_\ell, u_{\ell,j}, r_{\ell,j} \rangle \in Y^{\varphi}_{i,j}} u_{\ell,j},$$

then $E^{\varphi}_{i,j}$ is computed as

$$E^{\varphi}_{i,j} = \sum_{\langle T_\ell, u_{\ell,j}, r_{\ell,j} \rangle \in Y^{\varphi}_{i,j}} \frac{L}{P_\ell} W_{\ell,j} + L(1 - U^{\varphi}_{i,j}) p_{idle,j},$$

in accordance with equation (4.4).

The input is a set of task option encodings O_i and the desired quality score Q. In order to reflect T_i's parameters with respect to the configuration in consideration C, each O_i is processed in a way similar to the definition of U_C and R_C matrices in the mathematical program above to produce X_i. It is easy to see that this input processing is polynomial in the input size.

We utilize the framework developed by Woeginger [56] for formulating dynamic programs as follows. Let the initial state be $\mathbf{0} = \langle(\emptyset, 0), \ldots, (\emptyset, 0)\rangle$, where $|\mathbf{0}| = M$. We start with the initial state space $\mathcal{S}_0 = \{\mathbf{0}\}$, and generate new states in the state space $\mathcal{S}_i$ from $\mathcal{S}_{i-1}$ by means of a finite set of mapping functions $\mathcal{F}_C = \{f_1, \ldots, f_M\}$. Each $f_j \in \mathcal{F}_C$ specifies how a task is packed into the j-th processor π'_j in configuration C and generates a new state from it. For a certain $\varphi \in \mathcal{S}_{i-1}$ and task T_i, $\varphi' = f_j(C, X_i, \varphi)$ is the state that results from φ by updating the j-th pair $(Y^{\varphi}_{i-1,j}, E^{\varphi}_{i-1,j})$ in φ as follows: the triplet $\langle T_i, u_{i,j}, r_{i,j}\rangle$ is added to the schedule $Y^{\varphi}_{i-1,j}$ to produce $Y^{\varphi'}_{i,j}$ and the value of the new energy expenditure of the updated schedule is computed according to $u_{i,j}$ as follows

$$\begin{aligned} E^{\varphi'}_{i,j} &= \sum_{\langle T_\ell, u_{\ell,j}, r_{\ell,j}\rangle \in Y^{\varphi}_{i-1,j}} \frac{L}{P_\ell} W_{\ell,j} + \frac{L}{P_i} W_{i,j} + L(1 - (U^{\varphi}_{i-1,j} + u_{i,j}) p_{idle,j} \\ &= E^{\varphi}_{i-1,j} + \frac{L}{P_i} W_{i,j} - L u_{i,j} p_{idle,j}, \end{aligned} \tag{4.7}$$

where $E^{\mathbf{0}}_{0,j} = p_{idle,j}$ is the initial energy that processor π'_j in configuration C consumes when it is not assigned any tasks.

For each f_j there exists a mapping $h_{f_j} \in \mathcal{H}_C$, that serves as a tool to rule out infeasible assignments. We define h_{f_j} as

$$h_{f_j}(C, X_i, \varphi) = \begin{cases} U^{\varphi}_{i-1,j} + u_{i,j} - 1 & \text{if } u_{i,j} > 0, \\ \infty & \text{otherwise.} \end{cases} \tag{4.8}$$

A new state $\varphi' \in \mathcal{S}_i$ can be produced from $\varphi \in \mathcal{S}_{i-1}$ if, and only if $h_{f_j}(C, X_i, \varphi) \leqslant 0$. This definition of h_{f_j} captures both the requirements that a processor's capacity should never be exceeded as stated in inequalities (4.6), and that a task must be assigned to at least one processor over which it is defined for a schedule to be feasible. A platform configuration will therefore be invalid if there is at least one task that cannot be assigned to any processor in this configuration, over all states in the state

space of the preceding task. This can occur for a task T_i and configuration C either because T_i is not defined on any π'_j in C, i.e., $u_{i,j} = 0$, and for which $h_{f_j}(C, X_i, \varphi) = \infty$ for all $\varphi \in \mathcal{S}_{i-1}$, or because assigning T_i to π'_j will violate its capacity constraint, i.e., $U^{\varphi}_{i-1,j} + u_{i,j} > 1$, and for which $h_{f_j}(C, X_i, \varphi) = U^{\varphi}_{i-1,j} + u_{i,j} - 1 > 0$, for all $\varphi \in \mathcal{S}_{i-1}$.

Since $\mathcal{S}_N$ contains all feasible schedules of all N tasks across the processors in a platform configuration, it follows that there should exist an optimal schedule $\varphi^* \in \mathcal{S}_N$ for which $E(\varphi^*)$ is minimum, i.e., $\textsc{Opt} = E(\varphi^*) = \min\{E(\varphi) \mid \varphi \in \mathcal{S}_N\}$, where $E(\varphi) = \sum_{j=1}^{M} E^{\varphi}_{N,j}$.

Algorithm ScheduleExact considers all configurations and, for each task T_i, generates all distributions of utilizations to processors φ from those of T_{i-1} and eliminates infeasible ones according to h_{f_j}. For each configuration, it finds the state in the state space $\mathcal{S}_N$ of the N-th task with the minimum energy expenditure and adds both its energy and reward to the solution space. Finally, the solution space is ordered by non-decreasing energy and the first solution point to achieve the desired QoS score is chosen.

It should be noted that the solution points in the resulting solution space $\mathcal{A}$ form a partially ordered set, where some elements are incomparable. A point (E_q, R_q) is *energy inefficient with respect to reward* if there exists at least one point (E_ℓ, R_ℓ) such that $E_q \geqslant E_\ell$ and $R_q \leqslant R_\ell$. This means that the first solution consumes more energy than the second while attaining less reward. Those points are incomparable, so prior to ordering $\mathcal{A}$, the algorithm eliminates solution points that are energy inefficient with respect to reward to produce a total ordering on the solution space $\mathcal{A}$.

Proposition 1. ScheduleExact *is exponential in both time and space.*

Proof. The algorithm starts with the initial state $\mathcal{S}_0 = \{\langle(\emptyset, 0), \ldots, (\emptyset, 0)\rangle\}$, so $|\mathcal{S}_0| = 1$. Consider some task assignment to a particular configuration C. At each iteration i of the algorithm, $1 \leqslant i \leqslant N$, each option $x_i \in X_i$ can generate one or more states from each $\varphi_{i-1} \in \mathcal{S}_{i-1}$, because C might aggregate multiple identical processors that match x_i. Whenever T_i is assigned to a processor, that processor is 'stripped out' because T_i cannot be assigned to the same processor more than once. Therefore the number of

Algorithm 2: SCHEDULEEXACT($\langle O_1, \ldots, O_N, \rangle, Q$)

1 $\mathcal{A} \leftarrow \emptyset$
2 $\mathcal{S}_0 \leftarrow \{\langle(\emptyset, 0), \ldots, (\emptyset, 0)\rangle\}$
3 **foreach** C *of the* $\binom{\lambda+M-1}{M}$ *configurations* **do**
4   invalidConfiguration $\leftarrow$ `true`
5   **for** $i \leftarrow 1$ **to** N **do**
6         $\mathcal{S}_i \leftarrow \emptyset$
7   **foreach** $\varphi \in \mathcal{S}_{i-1}$ **do**
8   Process O_i according to C to generate X_i
9   **for** $j \leftarrow 1$ *to* M **do**
10   **if** $h_{f_j}(C, X_i, \varphi) \leqslant 0$ **then**
11                     $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup \{f_j(C, X_i, \varphi)\}$
12   invalidConfiguration $\leftarrow$ `false`
13   **end if**
14   **end for**
15   **end foreach**
16   **if** invalidConfiguration *is* `true` **then**
17   Move to the next configuration
18   **end if**
19   **end for**
20   Examine every pair of states φ and $\varphi' \in \mathcal{S}_N$. If $E(\varphi) \leqslant E(\varphi')$ and $R(\varphi) \geqslant R(\varphi')$ then remove φ' from $\mathcal{S}_N$
21   Sort $\mathcal{S}_N$ by non-decreasing $E(\varphi)$; sort entries with equal energy value by non-increasing reward $R(\varphi)$
22     $\hat{\varphi} \leftarrow$ the first φ in the sorted $\mathcal{S}_N$ such that $R(\varphi) \geqslant Q$
23   **if** *no* φ *in* $\mathcal{S}_N$ *achieves* $R(\varphi) \geqslant Q$ **then**
24   Move to the next configuration
25   **end if**
26     $\mathcal{A} \leftarrow \mathcal{A} \cup \{(C, \hat{\varphi})\}$
27 **end foreach**
28 **if** *all platform configurations are invalid* **then**
29   Report "No feasible schedule exists" and **terminate**
30 **end if**
31 Examine every pair (C, φ) and $(C', \varphi') \in \mathcal{A}$. If $E(\varphi) \leqslant E(\varphi')$ and $R(\varphi) \geqslant R(\varphi')$ then remove (C', φ') from $\mathcal{A}$
32 Sort $\mathcal{A}$ by non-decreasing $E(\varphi)$; sort entries with equal energy value by non-increasing reward $R(\varphi)$
33 **return** the first (C, φ) in the sorted $\mathcal{A}$

possible assignments of a task to a configuration is at most M. This means that the state space expands by a factor of at most M for each task. Thus the total number of states in the i-th iteration is bounded from above by

$$|\mathcal{S}_i| \leqslant M|\mathcal{S}_{i-1}|.$$

The total number of states is

$$\prod_{i=1}^{N} \frac{|\mathcal{S}_i|}{|\mathcal{S}_{i-1}|} = \frac{|\mathcal{S}_N|}{|\mathcal{S}_0|} \leqslant M^N$$

so

$$|\mathcal{S}_N| \leqslant M^N.$$

Considering all configurations, ScheduleExact will require at least $\mathcal{O}\left(\binom{\lambda+M-1}{M} M^N\right)$ time and space, which proves the claim. □

4.4.2 The Fully Polynomial-Time Approximation Scheme

The exponential state space generated above can be significantly reduced by observing that some states are 'close' to each other, and close states can be pruned without severely sacrificing optimality.

Define the error factor ϵ to be an input that specifies the desired trade off between the accuracy of the solution and the running time of the algorithm. It can be in the range

$$0 < \epsilon \leqslant 1. \tag{4.9}$$

We use the technique of Trimming-The-State-Space, introduced by Ibarra and Kim [28] and treated rigorously by Woeginger [56], in order to thin out the state space by merging 'close' states and bring down the size of the state space to a polynomial in N and $\frac{1}{\epsilon}$. Doing so allows us to derive an FPTAS that produces a solution having a value $\widetilde{E}$ whose relative error with respect to the value of the optimal solution E^* is bounded above by

the error factor ϵ, i.e., $\frac{\widetilde{E}-E^*}{E^*} \leqslant \epsilon$, and therefore the approximate solution is at most $(1+\epsilon)$ far from the optimal solution: $\widetilde{E} \leqslant (1+\epsilon)E^*$.

In order to do so, we need a measure of closeness between states to quantitatively decide on how to prune states so that the error resulting from such pruning is bounded and controlled, which we state in the following definitions.

Definition 3. *Let x, y be any real numbers. We say that x and y are Δ-close, where $\Delta \geqslant 1$ is the trimming factor, if $\frac{1}{\Delta}x \leqslant y \leqslant \Delta x$.*

The following proposition lists some useful and easily provable properties of Δ-closeness.

Proposition 2. *Properties of Δ-closeness*

1. *Δ-closeness is both reflexive and symmetric.*
2. *If x is Δ_1-close to y and y is Δ_2-close to z, then x is $\Delta_1\Delta_2$-close to z.*

We extend the definition above naturally to M-dimensional vectors over $\mathbb{R}^M$.

Definition 4. *Let $\mathbf{x}, \mathbf{y}$ be any vectors in $\mathbb{R}^M$. We say that $\mathbf{x}$ and $\mathbf{y}$ are Δ-close, where $\Delta \geqslant 1$ is the trimming factor, if $\frac{1}{\Delta}x_j \leqslant y_j \leqslant \Delta x_j$ for every $j \in \{1, \ldots, M\}$.*

Now we make **Definition 4** specific to the vectors in our application.

Definition 5. *Let $\mathcal{S}_i$ be the state space of task T_i. Two states $\varphi, \varphi' \in \mathcal{S}_i$ are Δ-close with respect to energy expenditure, where $\Delta \geqslant 1$ is the trimming factor, if $\frac{1}{\Delta}E^{\varphi}_{i,j} \leqslant E^{\varphi'}_{i,j} \leqslant \Delta E^{\varphi}_{i,j}$ for all $j \in \{1, \ldots, M\}$.*

We denote two states that are Δ-close with respect to energy as $[E, \Delta]$-close. This means that $E^{\varphi}_{i,j}$ and $E^{\varphi'}_{i,j}$ are 'good' representatives of each other, for all of the M processors in the platform, and either φ or φ' can be kept

and the other discarded; the decision of which to keep depends on the application [1]. By pruning states according to the measure above, we can obtain a solution that is 'close-enough' to the optimal while being able to control the error propagation resulting from such pruning.

State Space Pruning: From now on, we shall denote the unpruned state space as $\mathcal{U}$ and the pruned state space (resulting from refining $\mathcal{U}$) as $\mathcal{P}$. The idea of state space pruning is to partition the state space into groups containing $[E, \Delta]$-close states, and then selecting one state from each non-empty group to enter the state space. We denote such groups as Δ-boxes $\mathcal{B}_k$, where $\mathcal{B}_k \subseteq \mathcal{U}$ for all k (Woeginger [56]).

Consider the state space $\mathcal{U}_i$ generated by T_i. Let $\mathcal{E}_{i,j}$ be a multiset containing the energy values of the j-th coordinate of every $\varphi \in \mathcal{U}_i$

$$\mathcal{E}_{i,j} = \{E^{\varphi}_{i,j} \mid \varphi \in \mathcal{U}_i\}.$$

In other words, $\mathcal{E}_{i,j}$ is the set of energy values associated with schedules on processor π'_j among all partial feasible schedules up to the i-th task T_i in some platform configuration. Let $E_{i,\max} = \max_j \max\{\, E^{\varphi}_{i,j} \mid \varphi \in \mathcal{U}_i \,\}$. Then $E_{i,\max}$ is the maximum energy value across all processors and states in the state space $\mathcal{U}_i$. Now let $L_i = \lceil \log_\Delta \max(1, E_{i,\max}) \rceil$. Every vector in $\mathcal{U}_i$ is in $[0, \Delta^{L_i}]^M$ and thus $\mathcal{U}_i \subseteq [0, \Delta^{L_i}]^M$. We create $L_i + 1$ intervals, where $\mathcal{I}_0 = [0, 1)$, $\mathcal{I}_k = [\Delta^{k-1}, \Delta^k)$ for all $k \in \{1, \ldots, L_i - 1\}$, and $\mathcal{I}_{L_i} = [\Delta^{L_i - 1}, \Delta^{L_i}]$. Each $E^{\varphi}_{i,j} \in \mathcal{E}_{i,j}$ resides in exactly one interval, and cannot exceed Δ^{L_i}. We partition the energy values in each $\mathcal{E}_{i,j}$ to intervals as defined above, mapping each $E^{\varphi}_{i,j} \in \mathcal{E}_{i,j}$ to the interval to which it belongs. Assuming that $\texttt{SIZE}(E_{i,\max})$ is polynomial in the size of the instance $\texttt{SIZE}(I, \epsilon)$, the number of the thereby constructed intervals will be polynomially bounded in $\texttt{SIZE}(I, \epsilon)$. Further, we claim that for any two states, if for every coordinate j the energy values of both states are in the same interval, the those states are $[E, \Delta]$-close.

Claim 1. *Let $\mathcal{U}_i$ be the state space of task T_i. Let φ, φ' be two states in $\mathcal{U}_i$.*

[1] The reader can refer to Cormen, Leiserson, Rivest, and Stein [20] chap. 35 for an example of pruning as applied to the `SUBSET_SUM` problem.

If $E^{\varphi}_{i,j}$ *and* $E^{\varphi'}_{i,j}$ *are in the same interval for every* $j \in \{1, \dots, M\}$*, then* φ *and* φ' *are* $[E, \Delta]$*-close.*

Proof. Let $E^{\varphi}_{i,j}$, $E^{\varphi'}_{i,j} \in \mathcal{I}_k = [\Delta^{k-1}, \Delta^k]$ for some $k \in \{1, \dots, L_i\}$, for all coordinates j, $j \in \{1, \dots, M\}$. Then

$$\Delta^{k-1} \leqslant E^{\varphi'}_{i,j} \leqslant \Delta^k \quad \forall j \in \{1, \dots, M\} \tag{4.10}$$

and

$$\Delta^{k-1} \leqslant E^{\varphi}_{i,j} \leqslant \Delta^k \quad \forall j \in \{1, \dots, M\}. \tag{4.11}$$

Write the RHS of (4.11) as $\frac{1}{\Delta}E^{\varphi}_{i,j} \leqslant \Delta^{k-1}$ and the LHS of (4.11) as $\Delta^k \leqslant \Delta E^{\varphi}_{i,j}$. Substituting the rewritten inequalities into (4.10) yields

$$\frac{1}{\Delta}E^{\varphi}_{i,j} \leqslant \Delta^{k-1} \leqslant E^{\varphi'}_{i,j} \leqslant \Delta^k \leqslant \Delta E^{\varphi}_{i,j}$$

for every $j \in \{1, \dots, M\}$. Thus φ and φ' are $[E, \Delta]$-close by our definition of $[E, \Delta]$-closeness (definition 5). □

Algorithm SchedulePruned is a realization of the idea above. Procedure Prune performs state pruning. For every coordinate of every state φ, in addition to the partial schedule $Y^{\varphi}_{i,j}$ and the energy value $E^{\varphi}_{i,j}$ of this schedule, the algorithm maintains the number of the interval to which $E^{\varphi}_{i,j}$ belongs, which we will denote as $\ell^{\varphi}_{i,j}$, where $\ell^{\varphi}_{i,j} \in \{0, \dots, L_i\}$.

The pruning procedure will admit a single state from every $\mathcal{B}_k$ where $\mathcal{B}_k \cap \mathcal{U} \neq \emptyset$ into the pruned state space $\mathcal{P}$. Specifically, we choose the state with maximum reward over all states in the Δ-box in consideration to enter the pruned state space. Therefore, the number of states in the pruned state space is exactly the number of non-empty Δ-boxes. As a matter of fact, procedure Prune relies on the contra-positive of claim 1: if φ, $\varphi' \in \mathcal{U}_i$ are $[E, \Delta]$-far, i.e., there exists j such that either $E^{\varphi'}_{i,j} > \Delta E^{\varphi}_{i,j}$ or $E^{\varphi'}_{i,j} < \Delta^{-1} E^{\varphi}_{i,j}$, then $E^{\varphi'}_{i,j}$ and $E^{\varphi}_{i,j}$ must be in different Δ-boxes. Accordingly, the number of Δ-boxes of the i-th task will be bounded above by the maximum number of intervals over M coordinates (which we will show is polynomial in the input size when M is constant).

We use the observations above to bound the cardinality of the pruned state space as produced by our pruning procedure. First, we need to impose a technical condition on the packing functions f_j with respect to the trimming factor Δ, so that the error propagation resulting from pruning is controlled.

Proposition 3. *If state φ is $[E, \Delta]$-close to state φ', then for any $f_j, f_k \in F_C$, $f_j(C, X, \varphi)$ is $[E, \Delta]$-close to $f_k(C, X, \varphi')$.*

Functions that satisfy the condition above will be said to have the property of being $[E, \Delta]$-*closeness preserving*. This condition will be used later in establishing the correspondence between the state space that results from ScheduleExact and those produced by SchedulePruned.

The pruning factor Δ: We choose the pruning factor Δ to be equal to $1 + \frac{\epsilon}{2N}$. Therefore Δ is an increasing function with respect to user specified error factor ϵ.

Lemma 3. *The number of states in each iteration after pruning, $|\mathcal{P}_i|$, is polynomial in N and $1/\epsilon$.*

Proof. The cardinality of $\mathcal{P}_i$ is bounded above by the number of Δ-boxes $\mathcal{B}_{k,i}$ induced by the $[E, \Delta]$-closeness relation on $\mathcal{U}_i$, which in turn is bounded above by the number of intervals L_i in the i-th iteration. Let b_i be the number of Δ-boxes $\mathcal{B}_{i,k}$, $k \in \{1, \ldots, b_i\}$. Further let

$$B_i = |\{\, k \mid \mathcal{B}_{i,k} \cap \mathcal{U}_i \neq \emptyset, \quad k \in \{1, \ldots, b_i\}\,\}|.$$

The number of Δ-boxes is at most $(1 + L_i)^M$ and therefore

$$|\mathcal{P}_i| = B_i \leqslant b_i \leqslant (1 + L_i)^M.$$

We know that

$$L_i = \lceil \log_\Delta \max(1, E_{i,\max}) \rceil = \left\lceil \frac{\ln \max(1, E_{i,\max})}{\ln\left(1 + \frac{\epsilon}{2N}\right)} \right\rceil, \qquad (4.12)$$

Algorithm 3: SchedulePruned($\langle O_1, \ldots, O_N \rangle, \epsilon, Q$)

1 $\mathcal{A} \leftarrow \emptyset$
2 $\mathcal{P}_0 \leftarrow \{\langle (\emptyset, 0), \ldots, (\emptyset, 0) \rangle\}$
3 **foreach** C *of the* $\binom{\lambda+M-1}{M}$ *configurations* **do**
4   invalidConfiguration $\leftarrow$ `true`
5   **for** $i \leftarrow 1$ **to** N **do**
6   Process O_i according to C to generate X_i
7         $\mathcal{U}_i \leftarrow \emptyset$
8   **foreach** $\varphi \in \mathcal{P}_{i-1}$ **do**
9   **for** $j \leftarrow 1$ *to* M **do**
10   **if** $h_{f_j}(C, X_i, \varphi) \leqslant 0$ **then**
11                     $\mathcal{U}_i \leftarrow \mathcal{U}_i \cup \{f_j(C, X_i, \varphi)\}$
12   invalidConfiguration $\leftarrow$ `false`
13   **end if**
14   **end for**
15   **end foreach**
16   **if** invalidConfiguration *is* `true` **then**
17   Move to the next configuration
18   **end if**
19         $\mathcal{P}_i \leftarrow$ Prune($\mathcal{U}_i, N, \epsilon$)
20   **end for**
21   Examine every pair of states φ and $\varphi' \in \mathcal{P}_N$. If $E(\varphi) \leqslant E(\varphi')$ and $R(\varphi) \geqslant R(\varphi')$ then remove φ' from $\mathcal{P}_N$
22   Sort $\mathcal{P}_N$ by non-decreasing $E(\varphi)$; sort entries with equal energy value by non-increasing reward $R(\varphi)$
23     $\hat{\varphi} \leftarrow$ the first φ in the sorted $\mathcal{P}_N$ such that $R(\varphi) \geqslant Q$
24   **if** *no* φ *in* $\mathcal{P}_N$ *achieves* $R(\varphi) \geqslant Q$ **then**
25   Move to the next configuration
26   **end if**
27     $\mathcal{A} \leftarrow \mathcal{A} \cup \{(C, \hat{\varphi})\}$
28 **end foreach**
29 **if** *all platform configurations are invalid* **then**
30   Report "No feasible schedule exists" and **terminate**
31 **end if**
32 Examine every pair (C, φ) and $(C', \varphi') \in \mathcal{A}$. If $E(\varphi) \leqslant E(\varphi')$ and $R(\varphi) \geqslant R(\varphi')$ then remove (C', φ') from $\mathcal{A}$
33 Sort $\mathcal{A}$ by non-decreasing $E(\varphi)$, sort entries with equal energy value by non-decreasing reward $R(\varphi)$
34 **return** the first (C, φ) in the sorted $\mathcal{A}$

```
Procedure PRUNE(𝒰, N, ϵ)
1  Δ ← (1 + ϵ/(2N))
2  𝒫 ← ∅
3  Let E_max ← max_j max{E_j^φ | φ ∈ 𝒰}
4  Let L = ⌈log_Δ max(1, E_max)⌉
5  Compute Δ^2, ... Δ^L
6  foreach φ ∈ 𝒰 do
7      for j = 1 to M do
8          Consider (Y_j^φ, E_j^φ, ℓ_j^φ)
9          if E_j^φ < 1 then
10             Set ℓ_j^φ ← 0
11         else
               /* Find the interval in which E_j^φ resides */
12             for ℓ' ← 1 to L do
13                 if Δ^(ℓ'-1) ⩽ E_j^φ ⩽ Δ^ℓ' then
14                     Set ℓ_j^φ ← ℓ' and move to j + 1
15                 end if
16             end for
17         end if
18     end for
19 end foreach
20 Group states in 𝒰 for which ℓ_j^φ = ℓ_j^φ' for every j ∈ {1, ..., M}, for
   every φ, φ' ∈ 𝒰, in Δ-boxes ℬ_1, ..., ℬ_b
21 for  k ← 1 to b do
22     if ℬ_k ∩ 𝒰 ≠ ∅ then
23         Add to 𝒫 the state φ ∈ ℬ_k for which R(φ) is maximum
24     end if
25 end for
26 return 𝒫
```

but

$$\frac{\ln \max(1, E_{i,\max})}{\ln\left(1+\frac{\epsilon}{2N}\right)} \leqslant \frac{2N\left(1+\frac{\epsilon}{2N}\right)\ln \max(1, E_{i,\max})}{\epsilon} \tag{4.13}$$

$$\leqslant \frac{3N}{\epsilon}\ln \max(1, E_{i,\max}), \qquad \text{(by inequality (4.9))}$$

(Inequality (A.3) is obtained using $\ln(x+1) \geqslant \frac{x}{x+1}$ when $x > -1$). Thus

$$L_i \leqslant \lceil 1 + 3N/\epsilon \ln \max(1, E_{i,\max})\rceil.$$

The cardinality of $\mathcal{P}_i$ is therefore

$$|\mathcal{P}_i| \leqslant (1+L_i)^M \leqslant (1 + \lceil 1 + 3N/\epsilon \ln \max(1, E_{i,\max})\rceil)^M. \tag{4.14}$$

The latter bound is polynomial in N, $1/\epsilon$, and the number of bits required to encode the input in binary, where the number of processors M is constant. This concludes the proof. □

In order to obtain the approximation ratio of our FPTAS, we need a lemma relating the state spaces $\mathcal{U}$ and $\mathcal{P}$ of SchedulePruned to the state space $\mathcal{S}$ maintained by ScheduleExact. In fact, we need to show the effect of pruning on the value of the optimal solution produced by our algorithm. The following lemma shows how severely the optimal solution can deteriorate as a result of pruning.

Lemma 4. *If $\varphi^* \in \mathcal{S}_N$ is the state that yields the optimal solution in the exact DP formulation (Algorithm* ScheduleExact*), and $\tilde{\varphi} \in \mathcal{P}_N$ is the state returned by* SchedulePruned*, then $\tilde{\varphi}$ is at most $[E, \Delta^N]$-close to φ^*.*

Proof. State φ^* is produced by a chain of N-applications of functions $f_j \in \mathcal{F}_C$. Denote as $\varphi_i^* \in \mathcal{S}_i$ the state produced as a result of applying some $f_j \in \mathcal{F}_C$ to $\varphi_{i-1}^* \in \mathcal{S}_{i-1}$, i.e., $\varphi_i^* = f_j(C, X_i, \varphi_{i-1}^*)$ and thus $\varphi^* = \varphi_N^* = f_j(C, X_i, \varphi_{N-1}^*)$.

Consider the decomposition of φ^* into its N partial schedules $\varphi_1^*, \ldots, \varphi_N^*$. We show, by induction on i, that for $\varphi_i^* \in \mathcal{S}_i$, there exists a state $\tilde{\varphi}_i \in \mathcal{P}_i$ that is $[E, \Delta^i]$-close to φ_i^*. There is nothing to show for $i = 0$, since $\mathcal{S}_0 = \mathcal{P}_0$. Consider $\varphi_{i-1}^* \in \mathcal{S}_{i-1}$. By the induction hypothesis, there exists $\tilde{\varphi}_{i-1} \in \mathcal{P}_{i-1}$ that is $[E, \Delta^{i-1}]$-close to φ_{i-1}^*. By construction of $\mathcal{U}_i$, the set $\mathcal{U}_i$ contains the state $\hat{\varphi}_i = f_j(C, X_i, \tilde{\varphi}_{i-1})$. By construction of $\mathcal{P}_i$, there exists a state $\tilde{\varphi}_i \in \mathcal{P}_i$ that is $[E, \Delta]$-close to $\hat{\varphi}_i$. Since φ_{i-1}^* is $[E, \Delta^{i-1}]$-close to $\tilde{\varphi}_{i-1}$, the condition in proposition 3 guarantees that $\tilde{\varphi}_i$ is $[E, \Delta^{i-1}]$-close to φ_i^*. Since φ_i^* is $[E, \Delta^{i-1}]$-close to $\tilde{\varphi}_i$ and $\tilde{\varphi}_i$ is $[E, \Delta]$-close to $\hat{\varphi}_i$, it follows by proposition 2.2 that φ_i^* is $[E, \Delta^i]$-close to $\hat{\varphi}_i$.

Applying this result for $i = N$ proves the lemma. □

Now we are ready to obtain the desired approximation factor of our FPTAS.

Lemma 5. *The energy expenditure $\widetilde{E}$ of the solution produced by* SCHEDULEPRUNED *is $(1 + \epsilon)$ from the optimal solution E^*, i.e., $\widetilde{E} \leqslant (1 + \epsilon)E^*$*

Proof. From Lemma 4 we have

$$\begin{aligned} E_{N,j}^{\tilde{\varphi}} &\leqslant \left(1 + \frac{\epsilon}{2N}\right)^N E_{N,j}^{\varphi^*} \\ &\leqslant (1 + \epsilon) E_{N,j}^{\varphi^*} \end{aligned}$$

for every $j \in \{1, \ldots, M\}$, where we use the inequality $(1 + \frac{x}{n})^n \leqslant 1 + 2x$, $0 \leqslant x \leqslant 1$, $n \geqslant 1$, with $n = N$ and $x = \epsilon/2$. This is a reasonable approximation that can be verified by noticing that the left hand side is convex over $[0, 1]$ and the right hand side is linear. To formally prove its validity in our setting, we know that:

$$\begin{aligned} \left(1 + \frac{\epsilon}{2N}\right)^N &= e^{N \ln(1 + \epsilon/2N)} && (4.15) \\ &\leqslant e^{\epsilon/2} && (4.16) \\ &\leqslant 1 + \epsilon/2 + (\epsilon/2)^2 && (4.17) \\ &\leqslant 1 + \epsilon, && (4.18) \end{aligned}$$

where (4.16) is obtained by $\ln(1+x) \leqslant x$, when $|x| \leqslant 1$, and inequality (4.18) follows from inequality (4.9) by the following

$$\begin{gathered} 0 < \epsilon \leqslant 1 \\ 0 < (\epsilon/2)^2 \leqslant \epsilon/2 \\ \epsilon/2 < \epsilon/2 + (\epsilon/2)^2 \leqslant \epsilon \\ 1 + \epsilon/2 < 1 + \epsilon/2 + (\epsilon/2)^2 \leqslant 1 + \epsilon. \end{gathered}$$

Accordingly, the energy expenditure of our solution is

$$\begin{aligned} \widetilde{E} = E(\tilde{\varphi}) &= \sum_{j=1}^{M} E_{N,j}^{\tilde{\varphi}} \\ &\leqslant (1+\epsilon) \sum_{j=1}^{M} E_{N,j}^{\varphi^*} \\ &= (1+\epsilon)E(\varphi^*) = (1+\epsilon)\textsc{Opt}, \end{aligned}$$

which proves the lemma.

□

Theorem 1. *Algorithm* SchedulePruned *is a fully polynomial time approximation scheme that runs in time polynomial in N, $1/\epsilon$, and the number of bits required to encode the input in binary, where the number of processors M is constant.*

Proof. Let us start by bounding the number of steps required by procedure Prune, which is the bottleneck of SchedulePruned. Setting the interval numbers for every energy value across all states and processors (line 6 to 19) requires at most a total of $M(L_i+1)|\mathcal{U}_i|$ comparisons with the different $\Delta^{\ell'}$ values, where L_i is as defined in line 4. Grouping states into Δ-boxes (line 20) can be done as follows: Start with any vector in $\mathcal{U}_i$, create a Δ-box for it and insert it in this Δ-box. Then pick another vector and compare

with the first vector: if both have exactly the same interval indexes for all M processors, then insert the vector being examined into the Δ -box where the first vector resides, otherwise create a new Δ-box for it and insert it there. Therefore, for the k-th vector in $\mathcal{U}_i$, say φ_k, we will need to compare the interval indexes of φ_k to only one vector in the thereby created Δ-boxes, and insert to the Δ-box to which it belongs, or otherwise create a new Δ-box for it. If, at the worst, when examining the k-th vector, every vector so far examined resides in its own Δ-box (i.e., all are Δ-far), then PRUNE will need to perform $k-1$ vector comparisons. Therefore, the maximum number of interval index comparisons performed by the grouping procedure is bounded above by

$$\sum_{k=2}^{|\mathcal{U}_i|} M(k-1) = \mathcal{O}(|\mathcal{U}_i|^2),$$

which is the worst case asymptotic time complexity of PRUNE.

Processing the input as in line 6 can be done in time linear in the size of the specification of each task input per platform configuration.

For the elimination of incomparable solution points (line 21), we will need to examine at most $\binom{|\mathcal{P}_N|}{2} = \mathcal{O}(|\mathcal{P}_N|^2)$ vectors. The solution points per $\mathcal{P}_N$ can be sorted (line 22) by an optimal sorting algorithm using $\Theta(|\mathcal{P}_N| \log |\mathcal{P}_N|)$ comparisons.

Similarly for the set $\mathcal{A}$, we note that the cardinality of $\mathcal{A}$ cannot exceed the number of configurations Λ; therefore, elimination of incomparable solution points (line 32) will need to examine at most $\binom{|\Lambda|}{2} = \mathcal{O}(|\Lambda|^2)$ vectors, which is constant with respect to the size of the input (i.e., does not grow as the input size changes). Sorting $\mathcal{A}$ (line 33) requires $\Theta(|\Lambda| \log |\Lambda|)$, which, again, is a constant in the size of the input.

Since $|\mathcal{U}_i| \leqslant M|\mathcal{P}_{i-1}|$, we can write all of the bounds above in terms of $\mathcal{P}_{i-1}$ instead of $\mathcal{U}_i$, where $|\mathcal{P}_{i-1}|$ is as defined in (4.14). Let $\gamma_i = (1 + \lceil 1 + 3N/\epsilon \ln \max(1, E_{i,\max}) \rceil)$. Accordingly, the number of comparisons required for setting the index intervals in PRUNE as disscued above can be written as

$$M(L_i + 1)|\mathcal{U}_i| \leqslant M^2 \gamma_i \gamma_{i-1}^M.$$

Further, let $\widehat{E} = \max_i E_{i,\max}$. Let $\gamma = (1 + \lceil 1 + 3N/\epsilon \ln \max(1, \widehat{E}) \rceil)$. If we write the bounds above in terms of $\mathcal{P}_{i-1}$ and sum them over N, then the asymptotic time complexity of Algorithm SchedulePruned is

$$T(N, \epsilon) = \mathcal{O}(N\gamma^{2M}).$$

This concludes the proof. □

Practical Considerations

The running time of algorithm SchedulePruned is a polynomial with degree equal to twice the number of machines comprising the platform, $2M$. In addition, the hidden constants (at least $M^3\binom{\lambda+M-1}{M}$) can be considerably large for large enough platforms. As a consequence, SchedulePruned might not exhibit a fast running time for a considerably large number of processors. This makes our algorithm most suitable for environments that exhibit a static behavior in terms of task requirements and system conditions, so that the algorithm is executed in an offline fashion.

The running time of SchedulePruned, however, can be significantly enhanced by using the efficient implementation of partitioned scheduling using a look-up table approach, which was devised by Chattopadhyay and Baruah [15]. In brief, their algorithm computes all possible configurations for each bin independently of the task set requirements *only once per platform* (this is the computationally intensive part, but the number of configurations is polynomially bounded in the size of the problem instance), and then uses the stored tables to assign tasks to processors in an efficient manner (this is the efficient implementation part). Doing so will allow SchedulePruned to respond faster to changes in the platform, such as overload conditions, processor failures, etc., without the need to re-assemble the platform for every run of the algorithm.

Chapter 5

Conclusions

The service class model for task assignment is a simple method to express resource requirements and the associated reward/quality of service/performance/cost. This model can be used for several resource management problems, especially when the relationship between resource utilization and quality is highly non-linear. We used a basic service class model and provided some preliminary answers to immediate questions raised by the model. For restricted cases of QoS optimization with task allocation, we described polynomial time approximation schemes. Polynomial-time exact algorithms cannot exist unless $\mathsf{P} = \mathsf{NP}$; thus, approximations are the best we can hope for in polynomial-time. Further, the only measure of aggregate quality that we used was summations. There may be other better metrics for capturing the aggregate quality of service, and optimization using those metrics requires more mathematical insight. An interesting question to ponder is whether there are certain classes of functions that are amenable to joint treatment with task allocation (and bin packing).

Designing real-time computing systems to satisfy quality of service requirements while simultaneously reducing energy consumption is a computationally hard problem. While it may not be possible to obtain an optimal solution in polynomial time, we have been able to establish that near-optimal solutions can be obtained in polynomial time.

The strengths of our work are in this joint treatment of QoS and en-

ergy issues and in the accurate modeling of the execution time variations of tasks with processor speeds and in being able to capture capturing energy consumption in resources other than the processor. The holistic modeling of execution times and energy consumption improves the state of the art in this design problem.

We chose to restrict our attention to implicit-deadline real-time tasks and employed utilization bounds as a test for schedulability. We would like to remark that this is not a significant restriction in itself. The framework can support exact schedulability tests and arbitrary deadlines as long as one employs a polynomial time schedulability test. While exact schedulability tests do not run in polynomial time, we can instead use approximate tests (such as the one proposed by Chakraborty, Künzli, and Thiele [14]) to obtain polynomial time approximation schemes.

We have considered the situation when we have a fixed number of processor types and processors belong to one of these processor types. This opens up the way for further work on heuristics that have good performance and are faster than the dynamic programming approach that we have presented. Limited heterogeneity in compute units is a dominant choice in the design of embedded systems and our scheme is well suited to addressing this common case. Recently, Andersson, Raravi, and Bletsas [6] have examined the problem of task allocation without energy or QoS considerations on a platform with two processor types. Their heuristic methods may be an initial direction to consider for the more general problem we have defined.

Whereas we have considered the quality of service as being related only to the processor type that a task is assigned to, we believe that it is possible to extend our work to cover the case when a task has multiple service classes. Consider, as an example, a video application that can operate a high, medium or low quality. This adds another choice dimension where one needs to select the appropriate service class for a task in conjunction with the choices that we have discussed in this article. Task allocation and scheduling with multiple classes of service per task is a problem that has theoretical and practical implications for the design of real-time systems.

We also note that we have dealt with the case where energy consumed by

executing a job is independent of other jobs that are executed on the same processor. This may not always be the case. For example, jobs that have mutual cache interference may lead to higher energy consumption because of an increased number of cache misses that need to be satisfied by main memory. Considering such interference between tasks is a direction for future work. It first needs extensive measurements and methods for estimating the interference between tasks.

Bibliography

[1] L. Abeni and G. Buttazzo. Integrating multimedia application in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 4–13, Dec. 1998.

[2] M. Agrawal, N. Kayal, and N. Saxena. Primes is in p. *Ann. of Math*, 2:781–793, 2002.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, 1993.

[4] R. K. Ahuja, M. Kodialam, A. K. Mishra, and J. B. Orlin. Computational investigations of maximum flow algorithms. In *European Journal of Operations Research*, volume 97, pages 509–542, 1997.

[5] Amazon.com. EC2: Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[6] B. Andersson, G. Raravi, and K. Bletsas. Assigning Real-Time Tasks on Heterogeneous Multiprocessors with Two Unrelated Types of Processor. To appear at The 31st IEEE Real-Time Systems Symposium November 30 - December 3, 2010, San Diego, CA, USA.

[7] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press; first edition, April 20, 2009.

[8] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi:http://doi.acm.org/10.1145/1806596.1806620. URL http://doi.acm.org/10.1145/1806596.1806620.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[10] S. Baruah, R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, November 1990.

[11] E. Bini, G. Buttazzo, and G. Lipari. Minimizing cpu energy in real-time systems with discrete speed management. *ACM Transactions on Embedded Computing Systems*, 8(4):1–23, 2009. ISSN 1539-9087. doi:http://doi.acm.org/10.1145/1550987.1550994.

[12] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, volume 23 of *Real-Time Systems Series*. Springer, 2 edition, 2005.

[13] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In J. Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman Hall/CRC Press, 2004.

[14] S. Chakraborty, S. Künzli, and L. Thiele. Approximate schedulability analysis. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium*, page 159, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1851-6.

[15] B. Chattopadhyay and S. Baruah. A lookup-table driven approach to partitioned scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 257–265, april 2011. doi:10.1109/RTAS.2011.32.

[16] J.-J. Chen. Expected energy consumption minimization in dvs systems with discrete frequencies. In *SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 1720–1725, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-753-7. doi:http://doi.acm.org/10.1145/1363686.1364095.

[17] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. *Technical report no. 388, Graduate School of Industrial Administration, Carnegie-Mellon University*, 1976.

[18] E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In D. Hochbaum, editor, *Approximation Algorithms for* NP*-Hard Problems*, pages 46–93. PWS Publishing, Boston, 1996.

[19] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. doi:http://doi.acm.org/10.1145/800157.805047. URL http://doi.acm.org/10.1145/800157.805047.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. McGraw-Hill Science/Engineering/Math, 2009. ISBN 0262033844.

[21] W. F. de la Vega and G. S. Lueker. Bin packing can be solved within $(1 + \epsilon)$ in linear time. *Combinatorica*, 1(4):349–355, 1981.

[22] J. Dey, J. Kurose, and D. Towsley. On-line scheduling policies for a class of iris (increasing reward with increasing service) real-time tasks. *Computers, IEEE Transactions on*, 45(7):802 –813, jul. 1996. ISSN 0018-9340. doi:10.1109/12.508319.

[23] D. K. Friesen and M. A. Langston. Variable sized bin packing. *SIAM Journal of Computing*, 15(222–230), 1986.

[24] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource contraints. *SIAM Journal of Computing*, 4(2):187–201, June 1975.

[25] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of* NP*-completeness*. W. H. Freeman, 1979.

[26] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. pages 3 – 14, dec. 2001. doi:10.1109/WWC.2001.990739.

[27] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the ACM*, 34(1):144–162, January 1987.

[28] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975. ISSN 0004-5411. doi:http://doi.acm.org/10.1145/321906.321909.

[29] Intel Corporation. Intel® Core™ i-7 900 Mobile Processor Extreme Edition Series, Intel Core i7-800 and i7-700 Mobile Processor Series Datasheet. http://download.intel.com/design/processor/datashts/320765.pdf, 2009.

[30] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. *ACM Transactions on Algorithms*, 3(4):41, 2007. ISSN 1549-6325. doi:http://doi.acm.org/10.1145/1290672.1290678.

[31] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED '98: Proceedings of the 1998 International Symposium on Low power electronics and design*, pages 197–202, New York, NY, USA, 1998. ACM. ISBN 1-58113-059-7. doi:http://doi.acm.org/10.1145/280756.280894.

[32] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. pages 275 – 280, 2004.

[33] N. Karmarkar and R. M. Karp. An efficient approximation scheme for the one-dimensional bin packing problem. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science*, pages 312–320, November 1982.

[34] A. Khemka, R. K. Shyamsundar, and K. V. Subrahmanyam. Multiprocessors scheduling for iprecise computations in a hard real-time environment. In *Proceedings of the International Parallel Processing Symposium*, pages 374–378, April 1993.

[35] R. E. Korf. An improved algorithm for optimal bin packing. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1252–1258, 2003.

[36] C. Lee, J. Lehoezky, R. Rajkumar, and D. Siewiorek. On quality of service optimization with discrete qos options. In *Real-Time Technology and Applications Symposium, 1999. Proceedings of the Fifth IEEE*, pages 276–286, 1999. doi:10.1109/RTTAS.1999.777680.

[37] C.-G. Lee, C.-S. Shih, and L. Sha. Online QoS optimization using service classes in surveillance radar systems. *Real-Time Systems*, 28 (1):5–37, October 2004.

[38] W. Leinberger, G. Karypis, and V. Kumar. Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints. In *Proceedings of the International Conference on Parallel Processing*, pages 404–413, November 1999.

[39] L. Levin. Universal search problems. *Problems of Information Transmission, translated into English by Trakhtenbrot, B. A. (1984). "A survey of Russian approaches to perebor (brute-force searches) algorithms". Annals of the History of Computing, 6 (4): 384-400*, 9 (3), 1973.

[40] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[41] J. Liu, K.-J. Lin, W.-K. Shih, A.-s. Yu, J.-Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *Computer*, 24(5): 58 –68, may. 1991. ISSN 0018-9162. doi:10.1109/2.76287.

[42] J. W.-S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[43] J. M. Lopez, J. L. Diaz, M. Garcia, and D. F. Garcia. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 25–33, June 2000.

[44] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[45] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, Mineola, NY, 2 edition, 1998.

[46] I. Pratt, K. Fraser, S. Hand, C. Limpach, and A. Warfield. Xen 3.0 and the art of virtualization. In *Linux Symposium*, volume 2, pages 65–78, July 2005.

[47] V. R. Pratt. Every prime has a succinct certificate. *SIAM J. Comput.*, 4(3):214–220, 1975.

[48] O. Reingold. Undirected st-connectivity in log-space. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, STOC '05, pages 376–385, New York, NY, USA, 2005. ACM. ISBN

1-58113-960-8. doi:http://doi.acm.org/10.1145/1060590.1060647. URL http://doi.acm.org/10.1145/1060590.1060647.

[49] C. Rusu, R. Melhem, and D. Mossé. Maximizing rewards for real-time applications with energy constraints. *IN ACM Transactions on Embedded Computing Systems*, 2(4):537–559, 2003. ISSN 1539-9087. doi:http://doi.acm.org/10.1145/950162.950166.

[50] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. Fast: Frequency-aware static timing analysis. *ACM Transactions on Embedded Computing Systems*, 5(1):200–224, 2006.

[51] H. Shachnai and T. Tamir. Polynomial time approximation schemes for class-constrained packing problems. In *Proceedings of the Workshop on Approximation Algorithms*, pages 238–249, 1999.

[52] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of The London Mathematical Society*, s2-42:230–265, 1937. doi:10.1112/plms/s2-42.1.230.

[53] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, pages 239–254, December 2002.

[54] V. V. Vazirani. *Approximation Algorithms*. Springer, March 22, 2004.

[55] VMWare, Inc. VMWare ESX Hypervisor. http://www.vmware.com/products/vi/esx/.

[56] G. J. Woeginger. When does a dynamic programming formulation guarantee the existence of an fptas? *Electronic Colloquium on Computational Complexity (ECCC)*, (084), 2001.

[57] G. J. Woeginger. There is no asymptotic PTAS for two-dimensional vector packing. *Information Processing Letters*, 64(6):293–297, December 1997.

[58] W. Wolf. The future of multiprocessor systems-on-chips. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 681–685, New York, NY, USA, 2004. ACM. ISBN 1-58113-828-8. doi:http://doi.acm.org/10.1145/996566.996753.

[59] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 229–242, April 2007.

[60] C.-Y. Yang, J.-J. Chen, T.-W. Kuo, and L. Thiele. An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems. In *DATE*, pages 694–699, 2009.

Appendix A

A Modified Pruning Technique: Energy Rounding

We present a slightly different technique for state space pruning, which might result in a slightly improved running time; one that depends on the *ratio* of energy values instead of maximum energy values, which was the case in the original pruning technique. In addition, we present a generalization to binary search, one that locates an element in a list based on Δ-closeness instead of arithmetic equality.

We modify procedure PRUNE as following (the modified version is reported in PRUNE2). In addition to the input state space $\mathcal{U}$, PRUNE2 maintains an intermediate state space, $\widetilde{\mathcal{U}}$, which contains *energy-rounded* versions of the states in $\mathcal{U}$. For each state in the untrimmed input state space $\mathcal{U}$, and for each processor π'_j in C, PRUNE2 sorts the states in $\widetilde{\mathcal{U}}$ in an increasing order of the energy value of π'_j, then for the state being examined, say φ, it determines whether the energy value E_j^{φ} corresponding to π'_j is $[E, \Delta]$-close to any energy value of π'_j in the sorted list of rounded states. If so, it *rounds* the energy value E_j^{φ} to the energy value of the corresponding processor in the state to which it was found to be $[E, \Delta]$-close in $\widetilde{\mathcal{U}}$. Say that E_j^{φ} is Δ-close to $E_j^{\tilde{\varphi}}$, where $\tilde{\varphi} \in \widetilde{\mathcal{U}}$, such that $\Delta^{-1} E_j^{\tilde{\varphi}} \leqslant E_j^{\varphi} \leqslant \Delta E_j^{\tilde{\varphi}}$. Then rounding is performed simply by replacing φ's j-th coordinate with that of $\tilde{\varphi}$'s, i.e., setting $(Y_j^{\varphi}, E_j^{\varphi}) := (Y_j^{\tilde{\varphi}}, E_j^{\tilde{\varphi}})$. After rounding is done, PRUNE2 parti-

tions the rounded states into Δ-boxes, where states in the same Δ-box have equal energy components for all processors. Finally, the pruning procedure admits a single state from every non-empty Δ-box into the pruned state space. Therefore, the number of states in the pruned state space is exactly the number of non-empty Δ-boxes.

Procedure Prune2 relies on *Generalized Binary Search* (procedure GBS) to locate the position of the energy value of the unrounded state in the ordered list of (rounded) energy values, for the same processor, based on $[E, \Delta]$-closeness instead of exact equality. Procedure GBS is a natural generalization of binary search in our context, and its correctness follows from the invariant that distinct energy values for the same processor in the pruned state space are always Δ-far. Further, note that GBS becomes the well-known equality-based binary search algorithm if we set $\Delta = 1$.

The resulting pruned state space possesses the following property: *any two energy values for the same processor across all states in the pruned state space are either equal or Δ-far, and equal energy values correspond to the same schedule.*

Note that any two vectors in different Δ-boxes are $[E, \Delta]$-far, i.e., for $\varphi \in \mathcal{B}_{i,k}$ and $\varphi' \in \mathcal{B}_{i,l}$, $k \neq l$, there exists j such that either $E_{i,j}^{\varphi'} > \Delta E_{i,j}^{\varphi}$ or $E_{i,j}^{\varphi'} < \frac{1}{\Delta} E_{i,j}^{\varphi}$. Thus the maximum number of $\mathcal{B}_k$s is upper bounded by a polynomial in the number of distinct energy values of each processor.

In light of the new pruning procedure, we recast lemma 3 to obtain a bound on the cardinality of the pruned state space.

Lemma 6. *The number of states in each iteration after pruning as produced by* Prune2*, $|\mathcal{P}_i|$, is polynomial in N and $1/\epsilon$.*

Proof. Let $\mathcal{E}_{i,j}$ be the set containing all distinct energy values of the j-th coordinate of every $\varphi \in \mathcal{P}_i$

$$\mathcal{E}_{i,j} = \{E_{i,j}^{\varphi} \mid \varphi \in \mathcal{P}_i\}.$$

The cardinality of $\mathcal{P}_i$ is bounded above by the number of non-empty Δ-boxes $\mathcal{B}_{k,i}$ induced by the $[E, \Delta]$-closeness relation on $\mathcal{U}_i$, which in turn

Procedure PRUNE2($\mathcal{U}, N, \epsilon$)

```
1  Δ ← (1 + ε/2N)
2  𝒫 ← ∅
3  Ũ ← ∅
4  foreach φ ∈ 𝒰 do
5      for j = 1 to M do
6          Let ℒ ← {(Y_j^φ̃, E_j^φ̃) | φ̃ ∈ Ũ}
7          Sort ℒ in ascending order of E_j^φ̃
8          Let ℒ′ ← {E_j^φ̃ | (Y_j^φ̃, E_j^φ̃) ∈ ℒ}
9          [found, k] ← GBS(ℒ′, E_j^φ, Δ, 1, |ℒ′|)
10         if found is true then
               /* k is the index of the entry (state) in the
               sorted ℒ */
11             (Y_j^φ, E_j^φ) ← ℒ[k]
12         end if
13     end for
14     Add the rounded φ to Ũ
15 end foreach
16 Group states in Ũ for which E_j^φ = E_j^φ′ for every φ, φ′ ∈ Ũ in Δ-boxes
   ℬ_1, ..., ℬ_b
17 For every ℬ_k, Add to 𝒫 any φ ∈ ℬ_k if ℬ_k ∩ Ũ ≠ ∅
18 return 𝒫
```

is bounded above by product of the number of distinct, Δ-far energy values across all M processor in the platform configuration. Let b_i be the number of Δ-boxes $\mathcal{B}_{i,k}$, $k \in \{1, \dots, b_i\}$. Further let $B_i = |\{k \mid \mathcal{B}_{i,k} \cap \mathcal{U}_i \neq \emptyset \quad k \in \{1, \dots, b_i\}\}|$. Accordingly

$$|\mathcal{P}_i| = B_i \leqslant b_i \leqslant \prod_{j=1}^{M} |\mathcal{E}_{i,j}|.$$

Assume that energy values in each $\mathcal{E}_{i,j}$ are sorted in (strictly) increasing order such that

$$E_j^{\varphi_q} < E_j^{\varphi_\ell} \qquad \text{when } q < \ell,\ \forall j \in \{1, \dots, M\}.$$

Procedure GBS(list, needle, Δ, min, max)

output: A pair [found, index]: found is `true` if needle is successfully located in list; `false` otherwise. index is the location of of needle in list if it is found

```
1  while min < max do
2      mid = min + (max − min)/2
3      if (1/Δ) list[mid] ⩽ needle ⩽ Δ list[mid] then
4          return [true, mid]
5      end if
6      if needle > Δ list[mid] then
7          min ← mid + 1
8      else
9          max ← mid − 1
10     end if
11 end while
12 return [false, ]
```

Any two consecutive non-zero energy values in the total orderings above are related by

$$\frac{E_j^{\varphi_{\upsilon+1}}}{E_j^{\varphi_\upsilon}} > \left(1 + \frac{\epsilon}{2N}\right) \qquad \forall \upsilon \in \{1, \ldots, |\mathcal{E}_{i,j}| - 1\}.$$

Therefore,

$$\begin{aligned} E_{i,j}^{\varphi_2} &> \Delta E_{i,j}^{\varphi_1} \\ E_{i,j}^{\varphi_3} &> \Delta E_{i,j}^{\varphi_2} > \Delta^2 E_{i,j}^{\varphi_1} \\ &\vdots \\ E_{i,j}^{\varphi_{|\mathcal{E}_{i,j}|}} &> \Delta E_{i,j}^{\varphi_{|\mathcal{E}_{i,j}|-1}} > \cdots > \Delta^{(|\mathcal{E}_{i,j}|-1)} E_{i,j}^{\varphi_1}. \end{aligned} \tag{A.1}$$

From (A.1) we get

$$\beta_{i,j} = \frac{E_{i,j}^{\varphi_{|\mathcal{E}_{i,j}|}}}{E_{i,j}^{\varphi_1}} > \Delta^{(|\mathcal{E}_{i,j}|-1)}. \tag{A.2}$$

The constant $\beta_{i,j}$ is not defined if $\mathcal{E}_{i,j}$ does not contain two non-zero energy values, which happens in the following cases

1. The initial state, i.e. $\beta_{0,j}$,
2. If $|\mathcal{E}_{i,j}| = 1$,
3. If $|\mathcal{E}_{i,j}| = 2$ and $E_{i,j}^{\varphi_1} = 0$.

If $\beta_{i,j}$ is defined, then, from (A.2), we get

$$\begin{aligned}
\log_\Delta \Delta^{(|\mathcal{E}_{i,j}|-1)} &< \log_\Delta \beta_{i,j} \\
\Rightarrow |\mathcal{E}_{i,j}| &< 1 + \frac{\ln \beta_{i,j}}{\ln\left(1 + \frac{\epsilon}{2N}\right)} \\
&\leqslant 1 + \frac{2N\left(1 + \frac{\epsilon}{2N}\right)\ln \beta_{i,j}}{\epsilon} && \text{(A.3)} \\
&\leqslant 1 + \frac{3N \ln \beta_{i,j}}{\epsilon} && \text{(by inequality (4.9))} \\
&= \mathcal{O}\left(N/\epsilon \log \beta_{i,j}\right) \quad \forall i \in \{1, \ldots, N\} && \text{(A.4)}
\end{aligned}$$

(Inequality (A.3) is obtained using $\ln(x+1) \geqslant \frac{x}{x+1}$ when $x > -1$).

It might be the case that $|\mathcal{E}_{i,j}| > 2$ and $E_{i,j}^{\varphi_1} = 0$. Noticing that there cannot possibly be more than one zero element in $\mathcal{E}_{i,j}$ — since it contains distinct energy values — the situation can be handled easily by modifying equation (A.1) to use $E_{i,j}^{\varphi_2}$ as the minimum energy instead of $E_{i,j}^{\varphi_1}$. Nevertheless, this will yield the same asymptotic upper bound as that in equation (A.4).

The cardinality of $\mathcal{P}_i$ is therefore

$$\begin{aligned}
|\mathcal{P}_i| &\leqslant \prod_{j=1}^{M} |\mathcal{E}_{i,j}| && \text{(A.5)} \\
&\leqslant (N/\epsilon)^M \prod_{j=1}^{M} \log \beta_{i,j} \qquad [\text{if } \beta_{i,j} \text{ is defined}] && \text{(A.6)}
\end{aligned}$$

$\square$

We will also assume that $\texttt{SIZE}(\beta_{i,j})$ is polynomially bounded in $\texttt{SIZE}(I,\epsilon)$ in order for the bound in (A.6) to be polynomial in $\texttt{SIZE}(I,\epsilon)$. Further, under the assumption that the packing functions f_j are $[E,\Delta]$-closeness preserving (satisfy the condition in proposition 3), lemma 4 says that every energy value of the optimal solution can be rounded down at most N times as a result of pruning, so every energy value of the solution returned by SCHEDULEPRUNED is at most $\left(1+\frac{\epsilon}{2N}\right)^N$ times the corresponding energy value in the optimal solution, and thus the $1+\epsilon$ approximation factor is directly obtained as we did earlier.

A.1 Running-Time Analysis

Procedure GBS is just an extension of binary search and therefore requires $\mathcal{O}(\log n)$ time to locate an element in a list of size n. Therefore, the number of energy comparisons carried out for the state space $\mathcal{U}_i$ is bounded above by

$$\sum_{k=2}^{|\mathcal{U}_i|} 2M\log k - 1 = \mathcal{O}(|\mathcal{U}_i|\log|\mathcal{U}_i|).$$

Grouping states into Δ-boxes can be done as follows: Start with any vector in $\widetilde{\mathcal{U}}_i$, create a Δ-box for it and then insert it in this Δ-box. Then pick another vector and compare with the first vector: if both have exactly the same energy values for all M processors, then insert the vector being examined in the Δ -box where the first vector resides, otherwise create a new Δ-box for it and insert it there. Therefore, for the k-th vector in $\widetilde{\mathcal{U}}_i$, say φ_k, we will need to energy-wise compare φ_k to only one vector in the thus created Δ-boxes, and insert to the Δ-box to which it belongs, or otherwise create a new Δ-box for it. If, at the worst, when examining the k-th vector, every vector so far examined resides in its own Δ-box, then the algorithm will need to perform k vector comparisons. Therefore, the maximum number of

vector comparisons of the grouping procedure is bounded above by

$$\sum_{k=2}^{|\tilde{\mathcal{U}}_i|} k = \mathcal{O}(|\mathcal{U}_i|^2).$$

Therefore the time complexity of PRUNE2 is $\mathcal{O}(|\mathcal{U}_i|^2)$.

If we let $\widehat{\beta} := \max_i \max_j \beta_{i,j}$, then $|\mathcal{P}_i| = \mathcal{O}\left(\left(N/\epsilon \log \widehat{\beta}\right)^M\right)$. If we write the bounds above in terms of $\mathcal{P}_{i-1}$ and sum them over N, then the the time complexity of Algorithm 3 with the new pruning procedure is

$$T(N, \epsilon) = \mathcal{O}\left(N \left(N/\epsilon \log \widehat{\beta}\right)^{2M}\right).$$

This bound is polynomial in N, $1/\epsilon$, and the number of bits required to encode the input in binary, where the number of processors M is constant.