Algorithms and Models for Optimal Power Management on Smartphones

Dobson, Richard Mark

Awarding institution:
King's College London

Algorithms and Models for Optimal Power Management on Smartphones
PhD Thesis

Richard Dobson

May 11, 2014

Abstract

Smartphones are potent mobile devices which are required to operate for extended periods of time on battery power. In this thesis smartphone power management issues are addressed using algorithmic techniques.

Firstly, we consider power efficient scheduling for heterogeneous multi processor systems that allow dynamic speed scaling. We propose the Virtual Single Processor (VSP) approach which involves computing and utilising optimal system configurations. The VSP is used in combination with an efficient single processor dynamic speed scaling scheduling algorithm to compute highly power efficient schedules. We find that there is an average power saving of between 4.4% (2 processor system) and 8.175% (16 processor system) when compared to an alternative algorithm. Simulations also showed that the VSP approach reduced the objective function of $\sum$Weighted Flow + Energy by 2.31% more than the best known alternative. This work was published as a full paper at MISTA 2011.

Secondly, we discuss low energy Field Programmable Gate Array (FPGA) function mapping. A substantial FPGA power drain is caused by dynamic switching of the routing edges; this can be vastly reduced by mapping the input boolean function such that switching activity is minimised. We formulate the combinatorial optimisation problem, develop a complete neighbourhood function and apply simulated annealing to minimise cumulative switching. We find that our algorithm reduces the cumulative switching activity by an average of 27.44% compared to a genetic algorithm. This work appeared at GreenGEC 2013.

Finally, we examine the sleep state management problem in terms of advice complexity. We begin by showing the advice complexity of the problem is $r \log s$ where r is the number of idle periods and s is the number of sleep states. We design an algorithm which uses a single bit of advice to solve the single sleep state problem and show it to be 1.8-competitve. This is 20% better than the best possible deterministic algorithm. We also show that our algorithm can be improved by adding more advice but only until we have $\lceil \log b \rceil$ advice bits. Finally, in the case with more than 2 states our algorithm uses 1 bit of advice to improve on the deterministic algorithm.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis and the work contained within is the result of several years of work; there have been many people and organisations who have helped me reach this point who I would like to thank here. Firstly I would like to thank my supervisor Dr Kathleen Steinh´ofel for making this opportunity available to me and for the support I have received along the way. I have learnt so much over the past few years and this would not have been possible with out your help and expertise. I would also like to thank the EPSRC and Nokia UK for funding, King's College London for the learning environment and Hong Kong travel grant and the Department of Informatics for the exciting research environment and wealth of knowledge contained within.

I am grateful to Professor Andreas Albrecht for sharing his experience surrounding low powered boolean circuit design amongst other topics. I must also thank Dr Hans-Joachim B´ockenhauer for his collaboration on the advice complexity research; it was fantastic to work with such an expert in the area. Thanks are extended to all who I have worked with throughout my study who have helped to broaden my research horizons.

I would like to express my gratitude to my examiners Professor Iain Stewart and Dr Prudence Wong for the time and effort it took to read and examine my thesis. I thoroughly enjoyed discussing my work during the examination process and truly appreciate Dr Wong's extra effort to attend the oral examination despite difficult circumstances. Along with many useful comments about style and presentation they also highlighted ways to improve the work for which I am very grateful.

Heartfelt appreciation goes to my family who have supported me throughout my life and have offered me many opportunities and chances to achieve my goals. Finally I thank my wife and best friend Amy: for encouraging me to make the jump into research, proof reading countless copies of this thesis and her unending support along the way. I truly could not have done this without you.

Thank you all.

Richard Dobson

Chapter 1

Introduction

Since their inception mobile phones have been transformed. Once large, bulky devices with very limited functionality, they have become sleek, powerful tools with a plethora of additional utilities; from cameras to web browsers. In 1977 Ken Olsen (co-founder of Digital Equipment Corporation) stated that "there is no reason for any individual to have a computer in his home". In contrast to this statement almost all modern households today contain at least one computer. Access to computers has become a crucial part of everyday life and a large proportion of individuals now carry a portable computer (in the form of a smartphone) on their person at all times.

Such is the popularity and prevalence of mobile devices that the number of mobile phone subscriptions has climbed to an estimated 6.8 billion; almost equal in number to the global population [142]. Every device both consumes energy and requires energy to manufacture, cumulatively resulting in a significant level of expenditure worldwide. Energy is expended by mobile devices in two ways; operational energy: required for all components from CPUs to screens and embodied energy: required to research, design and manufacture the device itself. Reducing operational and embodied energy consumption is an ongoing challenge and can be affected by multiple factors such as product lifespan, number of devices manufactured and materials used in the device.

In this thesis we consider the problem of energy efficiency within computational systems, with a focus on mobile devices. Energy efficiency problems are receiving a significant amount of research interest as they are very complex problems and of great importance from a number of perspectives. We can consider the problems' importance on a number of levels.

From a smartphone user's perspective we know that poor battery life is a very common complaint. Whilst the technology inside smart phones has advanced rapidly, battery development has been significantly slower. This has resulted in a power gap which means that if a user makes the most of their smart phone's advanced features such as large touch screens, mobile internet and processor intensive applications then battery life can be just a few hours. Research which improves the use of the available power would enable the battery life to be extended; it would also have the added benefit of reducing the wear on hardware components. This is because a significant amount of power is converted into heat which can have a damaging effect on components if it cannot be dissipated effectively.

Energy efficiency problems clearly have a global environmental importance. If power consumption can be reduced then less power has to be produced, therefore less fossil fuels are used which in turn means less carbon enters the atmosphere. According to most research this is a key cause of global warming and climate change.

In addition to the environment impact or pollution related to energy production we also need to consider the energy supply problem. The number and density of devices requiring energy is increasing at an alarming rate; this is forcing the total energy consumption of the earth upwards at an unsustainable rate. If we can ensure that the devices which already exist are as energy efficient as possible then we will not stifle innovation of or access to computational devices due to the lack of or cost of energy.

For these reasons and many others we consider energy efficiency problems to be of the utmost importance. If they are not addressed through research and innovation it is highly likely that society will face serious energy related issues in the near future. We hope that the work in this thesis will help to address a subset of energy efficiency problems.

In this thesis we primarily focus on minimising the ongoing energy consumption but have also considered the impact on embodied energy. Three different energy efficiency problems are discussed: multi processor scheduling, FPGA mapping and sleep state management. In the following chapters we consider their very different characteristics and suggest novel solutions to each.

Structure of the Thesis

This thesis is structured as follows. First we give a comprehensive overview of literature related to harnessing or reducing the energy which computer systems (with a focus on mobile devices) consume. We begin by discussing how energy is consumed within a range of computational devices such as mobile phones and Field Programmable Gate Arrays (FPGAs). We then consider the work conducted in two of the most promising areas of research. Firstly, algorithms and approaches which reduce the energy consumption of processing units with a focus on sleep states and dynamic speed scaling for multi processor systems; secondly we move onto mapping algorithms which reduce the energy consumed by the dynamic routing of FPGAs.

In Chapter 3 we consider low energy scheduling for heterogeneous multiprocessor systems which allow dynamic speed scaling. We develop the 'Virtual Single Processor' approach to multiprocessor scheduling which combines a set of disparate processors in a pareto optimal way according to energy consumption and overall system processing power. This enables us to produce the same overall processing power as an alternative algorithm [72] whilst consuming between 4.4% and 8.2% less energy with no reduction in total speed. When a VSP is combined with a single processor $\sum$Weighted Flow + Energy scheduling algorithm we find that it can bind more tightly to the objective function than the best alternative [72].

Chapter 4 considers low energy FPGA circuit design. We study the problem of mapping an input boolean function onto a look-up table based Field Programmable Gate Array such that the overall energy consumed is minimised. This is an NP-hard problem and has been subject to much previous study. Our approach applies local search techniques as these have had success in similar areas. We find that when compared to a genetic algorithm approach [122] we can reduce the average switching activity (which is analogous to energy consumption) by an average of 27.44%.

In Chapter 5 we are concerned with the sleep state problem. We consider the problem in terms of advice complexity, where we have an all knowing adviser who can deliver information to the algorithm as needed. In this type of problem there are two goals: firstly to calculate the smallest amount of energy which is required to reach an optimal solution; and then to know how competitive an algorithm can be with a limited amount of energy. We solve the first of these two problems and present an algorithm which uses a small amount of

advice to improve on the best known deterministic algorithm. We find that with just one bit we can improve the competitive ratio to 1.8; more advice bits can be used to reduce the competitive ratio until we have $\lceil \log b \rceil$ advice bits when the competitive ratio converges onto a sub-optimal solution related to the characteristics of the power functions of each state. Finally we show that using a single bit of advice can improve the competitive ratio of the multi-state algorithm.

In the final chapter we summarise the contributions, outline the possible directions for future work and conclude the thesis.

Within the three contribution chapters there are large portions of novel work. The virtual single processor is the first algorithm to consider a multiprocessor system as a harmonised unit rather than disjoint processors. This enables our approach to find solutions that are significantly more efficient than other approaches. In our FPGA chapter we define the combinatorial optimisation problem, present a complete local search neighbourhood function and tailor the simulated annealing algorithm to find very high quality solutions to a NP-hard problem. In our final chapter we find the advice complexity of the sleep state management problem, then present a novel algorithm which uses advice and analyse it to discover it's time complexity. This work is all new and has been conducted during the course of the PhD study.

The VSP work has appeared at 2 international conferences initially as an extended abstract [54] and then as a full paper [55]. We have since been extending the work to submit to an international journal. The FPGA work has appeared at an international conference [56] and we are currently working on extending this approach to other similar problems. Finally, our advice complexity research is being presented here for the first time but will be submitted later this year.

Chapter 2

Related Work

In this chapter we give a comprehensive overview of existing research which has motivated and inspired the work of this thesis. We outline the fundamental concepts which underpin the thesis and discuss seminal works and their place within the academic landscape of low power algorithmic techniques.

In the first section we begin by discussing the motivating factors and breadth of approaches which can be employed to reduce the power consumption of computer systems. We move on to discuss techniques employed to reduce the energy consumption of processing units, the penultimate section discusses power reduction techniques for Field Programmable Gate Arrays and finally we consider the a number of global optimisation techniques with a focus on Simulated Annealing.

2.1 Energy and Power in Computing

In this section we discuss the breath of research which has the aim of reducing the power requirement or energy consumption of computational devices and the seminal papers which inspired large and dynamic areas of research; we begin by defining some fundamental terms.

The Oxford English Dictionary [147] defines energy and power as:

Definition 1 *Energy: "The power of doing work possessed at any instant by a body or system of bodies"*

Definition 2 *Power: "Any form or source of energy or force available for application to work"*

Energy is measured in a variety of units but the SI unit is Joules (J). In computer science (and throughout this thesis) we are most often referring to energy in the context of electrical energy. The energy consumption of a computer system (or constituent component) can be used to refer to the energy which the system (or component) has used over a period of time. Power is measured using the SI derived unit watts (W). A watt is defined by the following equation $W = \frac{J}{s}$ where W is watts, J is joules and s is seconds; therefore power is energy over time. If we reduce the (average) power consumption of a device we are in turn reducing the energy consumption of that device. When we have a fixed energy source such as a battery we wish to reduce the average power consumption as this increases the length of time the device will be operational.

The above definitions show that energy and power are intrinsically linked; therefore if research states that the goal is to reduce the energy consumption of a device then this is much the same as reducing the (average) power consumption. Therefore we discuss research papers which use either of these terms as the problems which they each address are strongly linked.

2.1.1 Energy Consumption in Computational Devices

Computational devices require energy to operate: electrical impulses are used to transfer signals, energy is required to power the CPU, memory and storage devices in addition, auxiliary input and output devices all demand energy [124].

Mahesri and Vardhan [110] analysed the power consumption of a laptop computer (an IBM ThinkPad R40) and its associated components. They managed to directly measure the power consumption of a number of components using an Agilent Oscilloscope and indirectly measure the power consumption of others. Figure 2.1 is a graph which shows the results for a number of their experiments. The authors were able to draw a number of interesting conclusions from their work: firstly the CPU consumes a large proportion of the total energy ($> 50\%$) especially when a CPU heavy task is being performed; the power consumed from the display back light is relatively high; and finally the power requirement of the memory system is relatively low even when it is in high demand.

The CPU has been subject to further research and power modelling as it has the potential to consume such large amounts of energy, especially when it operates at higher speeds. Brooks et al. [30] stated the common cube-root rule which asserts that the power consumption of a processor is equivalent to $P = S^{\alpha}$ where P is the power, S is the speed the processor is operating at and

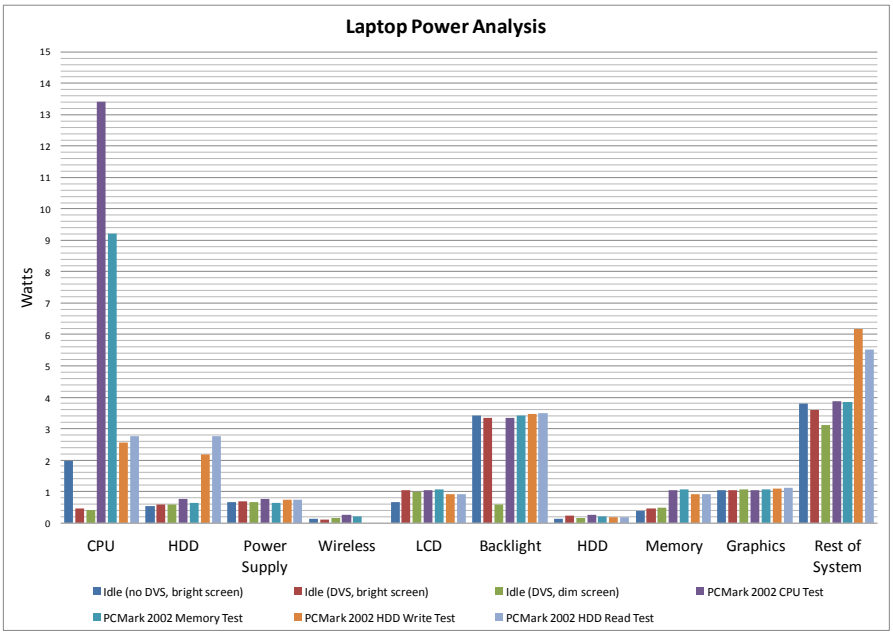

Figure 2.1: A sample of results from Mahesri and Vardhan

α is a constant which the authors suggest to be 3 but is defined by the specific processor and often falls in the range [2, 3].

Carroll and Heiser [35] analysed the power consumption of a typical smart phone (circa 2009) in terms of components. The researchers physically attached voltage and current measuring equipment to the individual components of the Openmoko Neo Freerunner but only to the battery terminals of the HTC Dream and Google Nexus One; this is because the electrical schematics are available for the Freerunner and not for either the HTC or Nexus. They found that the Freerunner consumed 68.8 mW whilst in suspended state, with the GSM modem requiring 31mW which is by far the largest proportion of the power consumption. When the device was placed in idle mode it demanded 268.8 mW; the graphics used the most power 82mW but the GSM (59mw), LCD (48mW), CPU (37mW) and audio (28mW) all consumed significant amounts of power. The authors performed several actions to find out which components consumed the most and least power when different tasks were being performed. When writing to the internal NAND flash memory they found the CPU used 99mW, when using the WiFi and GSM modules the energy consumption was 720mW and 630mW respectively and when making a phone call the GSM module required

820mW. The authors compare the 3 smartphones using a set of macro and micro benchmarks; the results are shown in Table (2.1). Carroll and Heiser's analysis states that the components which use the most power are the GSM module and the display (including the LCD and graphics processing) with the processor using a smaller but still significant proportion of the energy.

Benchmark	**Average System Power (mW)**		
	Freerunner	**G1**	**N1**
Suspend	103.2	26.6	24.9
Idle	333.7	161.2	333.9
Phone Call	1135.4	822.4	746.8
Email (cell)	690.7	599.4	-
Email (WiFi)	505.6	349.2	-
Web (cell)	500.0	430.4	538.0
Web (WiFi)	430.4	270.6	412.2
Network (cell)	929.7	1016.4	825.9
Network (WiFi)	1053.7	1355.8	884.1
Video	558.8	568.3	526.3
Audio	419.0	459.7	322.4

Table 2.1: Results from Carroll and Heiser [35]

In addition to 'standard' computer systems we also consider integrated logic circuits. These are widely used in mobile devices as they can be very small but incredibly powerful for certain tasks such as signal processing. Tuan, Kao, Rahman, Das and Trimberger [140] analysed a Field Programmable Gate Array in terms of power consumption. The authors divide the power consumption into 2 parts: static power (38%); and dynamic power (62%). Static power is consumed any time the FPGA is connected to a power source; this is mostly related to the hardware, therefore other approaches only have a very limited role in reducing this. Dynamic power is consumed when the FPGA is active; planning, placement and software can be employed to reduce this dramatically. Figure 2.2 outlines the power consumption of the static and dynamic power consumption of a Xilinx Spartan-3 FPGA.

We can see that static power is split into config, routing and logic; dynamic power is split into routing, clock and logic. Config refers to the power required to save, load and store the system config, clock power is consumed by signal timing when there is activity in the circuit, routing refers to the power consumed by the routing edges which connect the logic blocks and logic power is consumed by the Look Up Table (LUT) logic blocks. By far the largest proportion of power

consumption is due to the dynamic routing; this is almost entirely caused by the switching activity on the routing edges [140].

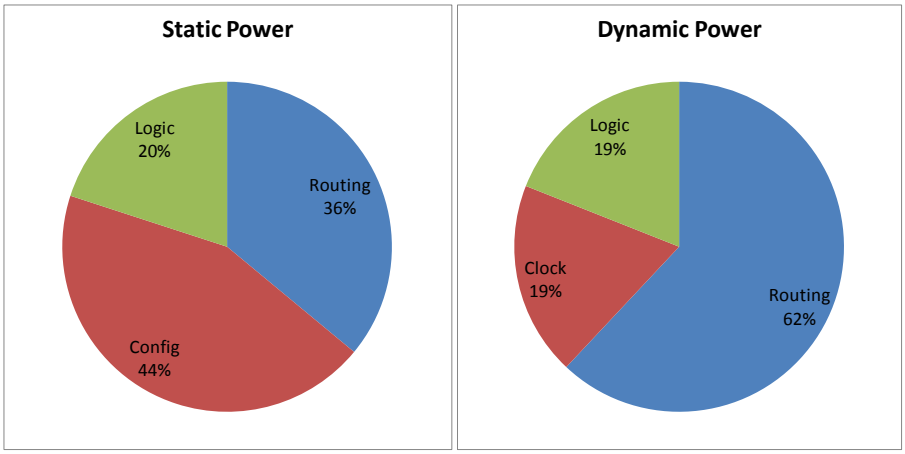

Figure 2.2: Power consumption of an FPGA circuit [140, 141]

Mobile devices depend on mobile energy sources which are often high powered battery technology. Since mobile devices were first invented there has been the ever-present challenge to provide enough power to sustain the device for a reasonable length of time; although battery technology has come a long way in the past decade, the gap between energy demands of the mobile devices and battery capacity has become even greater. Lahiri et al. [90] have illustrated the widening gap by plotting the improved power density of battery technology vs. the growing power demands of processors alone. Even with the latest developments in 3d battery technology introduced by Pikul et al. [126] we still expect there to be a sizable battery gap caused by the industry's insatiable need for increased processor speed and improved graphics performance. When this is combined with the growing environmental concerns about the energy consumed by computing equipment (which in 2008 was estimated to be 168 kW, 2.6% of the global energy consumption [125]) we see a clear motivation to consider the use of algorithms and optimisation techniques to reduce the energy consumption of mobile devices.

2.1.2 Power Reduction

Ellis [57], Brooks et al. [30], Mudge [118], Kant [80] and many others have argued the case for greater power management in computer systems. They

make the point that for many years increased computational power has been the driver of technology developments with energy efficiency and reduced power consumption being very much secondary objectives. As more computational devices are developed which rely on batteries or some other restricted power source the need for efficient power management is becoming more and more crucial. Mahesri & Vardhan [110] and Carroll & Heiser [35] show that different computer components consume differing amounts of energy. If we can learn how to optimise the use of each component and in turn the whole system in terms of power consumption then the impact will be incredible.

There are countless ways in which a computer system can be managed in order to reduce power consumption and extend battery life. Each and every component can be optimised and utilised in ways which can either waste or save energy; to discuss all types of computational systems and components would be unwieldy so in this thesis we concentrate on two of the areas we feel have great potential. First we consider the power consumed by the processing unit of a computer system and second the power consumed by the dynamic routing of FPGAs. We shall therefore discuss these components in detail in the following sections perhaps touching upon other related concepts and components as they are relevant.

2.2 Power Consumption in Processing Units

Over the last few years energy efficiency has become a design constraint for all computer systems ranging from mobile phones to server farms (the vast majority of these systems make use of some kind of processing unit) leading to a wealth of research which aims to reduce the energy consumption of processors. Brooks et al. [30] initiated the research in this area, motivated by the 'cube root rule' which states that the speed of the processor is equal to the cube root of the energy input, $P = S^3$. This is usually presented in the more generalised form $P = S^\alpha + c$ where P is the power consumption, S is the speed (or frequency) of the processor and α & c are constants with α usually being between 2 and 3.

Throughout this section and the remainder of the thesis we discussing many algorithms. Offline algorithms solve problems where all of the relevant information is available at the start of the computation. An online algorithm on the other hand is used to solve a problem where information becomes available over time.

In 1985 Sleator et al. [133] introduced the idea of assessing the worst case performance of an online algorithm by comparing it to the optimal offline solution. This is commonly known as competitive analysis. For an online problem P we have: $O(I)$ which is the optimal solution for input I; $A(I)$ is the online algorithm's solution for the same input. Each algorithm's solution has a cost $C(O(I))$ and $C(A(I))$. We state that A is c-competitive if there exists some constant $\phi \geq 0$ such that for any I the following holds:

$$C(A(I)) \leq c \cdot C(O(I)) + \phi \tag{2.1}$$

If $\phi = 0$ then we can state that A is strictly c-competitive. Furthermore if $c = 1$ and $\phi = 0$ then A is optimal.

In the remainder of this section we focus on 2 common methods for reducing the energy consumption of processors: Sleep States; and Dynamic Speed Scaling. Within each of these methods we look at the various algorithms and techniques which have been suggested to reduce energy consumption and weigh up the merits and downfalls of each method.

2.2.1 Sleep States

A very popular technique for reducing the energy consumption of a computer system is to power down various hardware components. In their study of the power consumption of a smart phone Carroll and Heiser [35] note that powering down the screen during a phone call is a very effective way to reduce the overall energy consumption with very little inconvenience to the user.

The processor is one of the most energy demanding parts of a computer system, with some estimates suggesting that between 50% and 60% of the overall energy consumption is due to the processing unit [8, 110]. This suggests processor sleep states have the potential to facilitate large energy savings. In this section we look at the technique of reducing the energy consumption of the processing units in idle periods by putting the processor into one of a number of low power modes.

We define a set of states $\{s_0, s_1 ... s_n\}$ each of which has an ongoing energy consumption of $p(s_i)$ and a wake energy of $w(s_i)$.

- $w(s_0) = 0$
- $p(s_n) = 0$

- $\forall_{i<n}\ p(s_i) > p(s_{i+1})$
- $\forall_{i>0}\ w(s_i) < w(s_{i+1})$
- s_0 is the active, working state
- $\{s_1 ... s_n\}$ are sleep states

The task is to minimise the total amount of energy consumed by utilising sleep states during idle periods. The problem can be considered either in the offline situation where the lengths and locations of idle periods are known beforehand or in the online situation where we only know the length of an idle period after it has occurred. We can further split this problem into two natural sub-problems, the first being where we have just 1 sleep state and the second being where we have many sleep states.

If there exists just 1 sleep state then by definition the available states must be on and off. This means that the problem is reduced to choosing between an ongoing cost and or a fixed cost. Irani et al. [77] remarked that this is simply an instance of the ski rental problem: a skier must decide whether to rent skis at a daily cost or buy skis outright at a higher fixed cost where the skier does not know the length of their ski trip. In this case the offline problem is simple and can be solved optimally using the following rule: if $p(s_1) \cdot t > w(s_0)$ then sleep, otherwise wait. In the online case the best possible deterministic online algorithm is 2-competitive [62, 77, 78].

Algorithm 1 Irani et al.

If $p(s_1) \cdot t > w(s_0)$ then sleep.
Otherwise idle.

If idle periods are generated by a known probabilistic distribution then Karlin [81] has shown that a randomised online algorithm (ALG-P) can be $\frac{e}{e-1}$-competitive (where e is the base of the natural logarithm) for the average case and that this is optimally competitive.

The Ski rental problem is a perfect analogy if a system has just 1 sleep state but most modern processors have many different levels of sleep states and trying to solve this problem is more complex. Figure 2.3 shows a system with 4 sleep states and their energy efficiency over an idle period.

Irani et al. [77] defined the optimal offline algorithm OPT which calculates

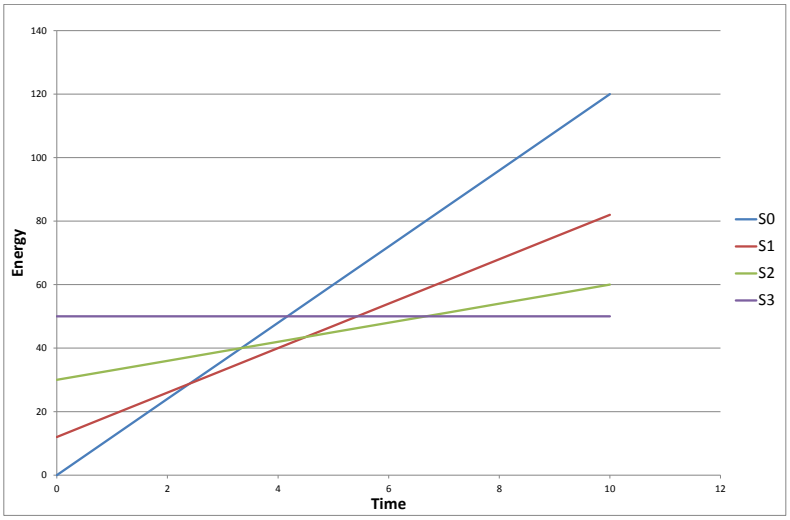

Figure 2.3: Multiple sleep states energy consumption

which is the best sleep state to use for any idle period using equation (2.2).

$$OPT(t) = \min_{1 \leq i \leq n} \{p(s_i) \cdot t + w(s_i)\} \tag{2.2}$$

Equation (2.2) is a logical extension of the offline solution for the 2 state problem and can be visualised using Figure 2.3, if we draw a vertical line up through the graph at the time period we are considering then the first power function line that we intersect corresponds to the optimal sleep state. Irani et al. used the logic which underpins equation (2.2) to inform the development of Lower-Envelope, their online algorithm. The algorithm states that as the length of the idle period increases the lower envelope of the graph should be followed, i.e. when the line corresponding to the current state intersects another line then the system should transition into that state. This algorithm has been shown to be 2-competitive and this was shown to be the best solution any deterministic algorithm can achieve. In the same paper Irani et al. also studied the case where the idle periods are probabilistically distributed. They developed the algorithm ALG-P(l) which is an extension of the solution for probabilistic 2 state systems ALG-P, first presented in [81]. Augustine et al. [16] considered the

more generalised case where transition energies can take arbitrary values. The authors extended Lower-Envelope and showed that it is $(3 + 2\sqrt{2})$-competitive and that this is true for any state based system.

Naturally this problem has been extended to the multiprocessor case where each processor can be in a sleep state at any one time. Demaine et al. [52] consider the situation where each processor has just 2 states. The authors attempt to minimise the total power consumption and develop a $(1 + \frac{2}{3}w(s))$-competitive algorithm and show that the dependence on $w(s)$ (the cost of sleeping) is essential. Sze-Hang Chan et al. [41] consider the sleep state problem for data centre provisioning to minimise the combination of response time and energy and find $O(1)$-competitive algorithms in the offline case. Sze-Hang Chan et al. have since extended this research to the non clairvoyant dynamic data centre provisioning problem [42]. They show that for any $\epsilon > 0$ their SATA algorithm is $(1 + \epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive for the objective of minimising $\sum$Flow and Energy.

Sleep state management has been very impressive but can only save energy when a system is not in use or the jobs it is required to process are not time critical; this has its limitations in many situations. For example some systems have very few idle periods in which to enter sleep states or high priority systems may be configured to never sleep to increase response time and reliability. In the next subsection we will discuss another energy saving system which can be developed (in some cases in collaboration with sleep states) to reduce the energy consumed while the system is active.

2.2.2 Dynamic Speed Scaling

Dynamic Speed Scaling (or Dynamic Voltage Scaling) is a technique which allows the speed of a processor to be modified at runtime. Reducing the operating speed (or frequency) also reduces the power consumption as Power (P) and Speed (S) are intrinsically linked through the equation $P = S^{\alpha} + c$. The well known cube root rule [30] asserts that $\alpha = 3$, suggesting that even small changes in the operating speed can have large changes in the power consumption. The main challenge of systems implementing Dynamic Speed Scaling (DSS) is to manage the conflicting goals of minimising energy consumption whilst maximising quality of service; this is an area where much research has been focused and there are many different approaches and results.

Many modern processors are equipped with Dynamic Speed Scaling (DSS)

through systems such as the Intel SpeedStep and AMD's Cool'n'Quiet or PowerNow!. Mahesri et al. [110] noted that the use of Dynamic Speed Scaling can significantly reduce the power consumption when the system is idle. Carroll et al. [35] stated that DSS has the ability to severely reduce the power consumption of the CPU but found that it was limited in reducing the power consumption of the whole device for their particular smart phone. This is likely to have been because the smartphone under test has a significantly less powerful processor than modern day devices; in cases with higher power processors with more cores we would expect to see DSS having a greater overall effect on the device.

In general, real world implementations of DSS do not allow infinite control over the precise speed of the processor but allow the speed to be set to one of a number of available speeds which the processor has been designed and tested to run at. Bansal [18] has argued that a model which considers a continuous speed function can still be of practical use in this situation as any speed can be simulated by flipping between 2 different speeds in the correct ratio. We shall discuss the work of Bansal et al. in more detail later, but this shows that the following theoretical models are applicable to real world situations.

When DSS is implemented it is crucial to define an objective function which balances power savings against the quality of service. In research, there are 3 main groups of objective functions: jobs have hard deadlines, bounded energy budget or response time and balance energy consumption and response time. We shall consider each of these approaches in the following sub-sections.

Deadline Scheduling

Deadline Scheduling is the task of managing the speed of the processor through DSS whilst scheduling the available jobs such that all jobs meet their deadlines but the overall energy consumption is minimised. The scheduler must define the speed of the processor(s) and schedule each job onto a processor such that all jobs meet their deadlines. This is a non-trivial problem which has been studied for many years. Yao et al. [151] presented their seminal paper in 1995 which initiated research into this fascinating area. The paper starts by defining the model which has been extensively used in subsequent research. For a fixed time interval $[t_0, t_1]$ the task is to schedule a set of jobs J which need to be processed within that time period. Each job $j \in J$ has:

- an arrival time a_j
- a deadline b_j where $b_j > a_j$

- a required number of CPU cycles R_j
- and an interval $[a_j, b_j]$

Furthermore each interval $\Delta = [z, z']$ has an intensity $g(\Delta)$ which is calculated using the following equation:

$$g(\Delta) = \frac{\sum R_j}{z' - z} \tag{2.3}$$

where the sum is taken over all jobs j with $[a_j, b_j] \subseteq [z, z']$.

A schedule is a pair of $H = (S, job)$ defined over $[t_0, t_1]$ where:

- $S(t) \geq 0$ is the speed of the processor at time t
- $job(t)$ defines the job being processed at time t or idle if $S(t) = 0$

$s(t)$ and $job(t)$ must be piecewise constant with finitely many discontinuities. A feasible schedule for an instance J is a schedule H that satisfies:

$$\forall_{j \in J} \int_{a_j}^{b_j} s(t)\delta(job(t), j)dt = R_j \tag{2.4}$$

where $\delta(x, y)$ is 1 if $x = y$ and 0 otherwise. The energy consumption per unit time P is assumed to be a convex function of the processor speed. The goal of any deadline scheduling algorithm is to minimise the following:

$$E(S) = \int_{t_0}^{t_1} P(s(t))dt \tag{2.5}$$

After defining the model the authors move on to describe the optimal offline algorithm (YDS).

Algorithm 2 YDS

For a list of jobs J
While $J \neq \emptyset\{$
Calculate interval I which has the maximum intensity
J_I is the set of jobs in I
Process jobs in J_I according to EDF at speed $S = \frac{1}{|I|} \sum_{J_i \in J_I} R_i$
$J = J \backslash J_I$ Remove I from timeline and update release times and deadlines of unscheduled jobs
$\}$

where EDF is earliest deadline first.

Yao et al. [151] show that YDS produces an optimal schedule for any job set, minimising the overall energy consumption. Yao et al. also calculates the worst case running time of YDS to be $O(j^3)$ but Li et al. [106] reduced this to $O(j^2 \log j)$ where j is the number of jobs. Yao et al. also develop two natural online heuristic algorithms: Average Rate Heuristic (AVR); and Optimal Available (OA). We outline both below.

Algorithm 3 AVR

Each job has a density: $d_j = \frac{R_j}{b_j - a_j}$
The processor speed should equal: $s(t) = \sum_j d_j(t)$

AVR is analysed in [151] and if the power function holds for $P = S^\alpha$ where $\alpha \geq 2$ the authors prove the algorithm to be $(2^{\alpha-1}\alpha^\alpha)$-competitive and they show the lower bound of competitiveness to be α^α.

Algorithm 4 OA

Compute the optimal schedule each time some new jobs arrive using the new jobs and the remaining portion of the existing jobs as if it were a offline problem

The Optimal Available algorithm was not analysed in [151] but was later proven to be (α^α)-competitive by Bansal et al. [20].

It is also clear that Yao, Demers and Shenkers' contribution has been incredible and sparked a wealth of DSS deadline scheduling papers including [6, 9, 17, 38, 37, 40, 74, 92, 152]. These papers and many others consider a plethora of different models of computer systems and all manner of extensions on the original problem proposed by Yao et al.

Deadline scheduling has significantly influenced research into DSS but it has some fundamental drawbacks. Firstly it is not always possible to calculate a feasible schedule for any set of jobs on a standard processor with a maximum speed. Theoretically we can either consider a system with no maximum speed or we can restrict the input set such that we can always guarantee that all deadlines can be met given the maximum speed of the processor(s). Clearly if we are using a real processor then removing the maximum speed is not possible. This means that we must artificially modify job deadlines such that a feasible schedule is possible which undermines the whole system.

Perhaps the most significant issue is that it is not always natural to assign each and every job a deadline. For example many maintenance tasks need to

be performed at some point but it is not vital they are processed when the computer is in high demand. If we were to assign an arbitrary deadline to a task we could end up forcing the computer to perform this work during a very high demand period in order to hit its deadline. Many jobs share this property of needing to be processed at some point but not by a specific deadline, for this reason deadline scheduling is not prominent in common operating systems although it does have its place in real time systems where energy considerations are very much a secondary objective.

Bounded Energy or Bounded Performance

Perhaps as a reaction to the issues of deadline scheduling, many researchers have considered other ways of managing the balance between quality of service and energy efficiency. Bunde [33] was one of the first to consider this; he stated that reducing energy consumption whilst achieving a certain level of service was a bi-criteria problem. A common approach to bi-criteria problems is to bound one factor and achieve the best value for the other. He went on to outline two different types of problem: the 'laptop problem' and the 'server problem'. In the laptop problem we have a fixed energy budget and we wish to provide the best quality of service. In the server problem we have a fixed level of performance and we wish to use the least amount of energy to achieve this.

Pruhs et al. [127] first tackled the problem of minimising the average response time of a set of jobs given a fixed energy budget in 2004; this paper was later updated in 2008 [128]. Pruhs et al. consider the problem of scheduling a set of foreknown, equi-work jobs onto a single processor capable of DSS. The authors develop an algorithm which calculates the optimal schedule for a huge energy budget such that all jobs are completed before the next arrives. They then lower the energy budget and make changes to the schedule such that the makespan increase is minimised but the budget is not breached. The authors show that this algorithm is O(1)-competitive for equi-work jobs providing that the energy budget is relaxed to $(1 + \varepsilon)$.

Bunde [33, 34] has published 2 versions of his paper 3 years apart: an extended abstract in 2006 and a journal article in 2009. He considers the offline problem where the release time and quantity of work is known for all jobs at the start. The paper first tackles the problem of energy efficient makespan scheduling on a uniprocessor. This is a version of the laptop problem where the energy is bounded and the makespan is the measure of the quality of service. Bunde

starts by formalising the problem and then moves on to develop an optimal offline algorithm IncMerge which runs in linear time.

In many bi-criteria situations bounding one of the factors and optimising the other factor within this is a good option. It allows the problem to be simplified such that an effective solution can be found. On the surface both the laptop problem and the server problem seem logical but they are both inherently flawed in an online situation.

Firstly we discuss the laptop problem where we bound the energy and wish to maximise the performance. Consider a situation where we have some fixed energy budget and a processor with unbounded speeds. Upon the arrival of the first job we have to make a choice of how much energy we use. We could use all of the energy budget and process the job as fast as possible which would be optimal for just one job but would fail to process the remaining jobs if any more arrive. Alternatively we can choose to use a portion of our energy and save the rest for jobs which may never arrive. In this situation we would be far from optimal if no more jobs arrive, but better if more do. This scenario shows that for the general laptop problem it is not possible to bound the competitive ratio for the online situation. The server problem provides the same conundrum but in reverse.

The final issue with this approach is that using makespan to measure the quality of service is not ideal. The concept of makespan is that we wish to minimise the finish time of the final job. This means that the release time of the final job has far more power to influence the energy consumption and performance of the overall system. Minimising makespan is good for situations with batch work which needs to be processed in a reasonable time but in real life situations it is unlikely that we want to minimise the makespan of the whole input rather than bounding the performance of certain important jobs.

$\sum$Flow + Energy

In 2007 Albers and Fujiwara [8] developed an objective function which is flexible enough to mean a feasible solution is always possible but still keeps a large emphasis on energy consumption. In their paper they describe an objective function which combines two conflicting measures of $\sum$Flow + Energy and attempts to minimise the total. The $\sum$ Flow is the sum of the difference between a job's release and completion time; Energy refers to the energy consumption of the processing unit.

The flow, or more generally weighted flow [21], has been considered as a good measure of quality of service for some time with many researchers studying minimisation of flow in cases where the speed of the processor is fixed. Kellerer [82, 83] showed that minimising the unweighted flow non-preemptively on a single processor is $n^{\frac{1}{2}-\epsilon}$ hard and many researchers have developed solutions to solve this and many other related problems [31, 63, 123, 102, 43, 23].

Albers and Fujiwara [7, 8] pioneered the development of algorithms that minimise $\sum$Flow + Energy. The authors first show that if the jobs are allowed to have arbitrary sizes then there can be no algorithm which achieves a constant competitive ratio; they therefore presented 2 main algorithms which focus on job scheduling problems with fixed job size. The online algorithm, Phaseball, is a batch processing algorithm which links the processor speed to the number of jobs waiting within the current batch and the value of α within the equation $P = S^{\alpha}$. The speed of the processor is specified by the equation $\sqrt[\alpha]{q/c}$ where α is from $P = S^{\alpha}$, q is the number of jobs waiting within the current batch and c is a value which depends on the value of α. If $\alpha < (19 + \sqrt{161})/10)$ then $c = \alpha - 1$ else $c = 1$.

In [8] Albers and Fujiwara proved this algorithm to have a constant competitive ratio for all values of α.

$$(1 + \Phi)(1 + \Phi \frac{\alpha}{(2\alpha-1)}^{(\alpha-1)} \frac{\alpha^{\alpha}}{(\alpha-1)^{\alpha-1}} \min\{\frac{5\alpha-2}{2\alpha-1}, \frac{4}{2\alpha-1} + \frac{4}{\alpha-1}\} \\ \text{where } \Phi = (1 + \sqrt{5})/2 \approx 1.618 \text{ (the golden ratio)} \tag{2.6}$$

Bansal et al. [21] show that when the cube root rule holds (i.e. $\alpha = 3$) this equates to a bit over 400-competitive; in the same paper Bansal et al. improve the competitive ratio of Phaseball to 4-competitive.

In 2007 Bansal, Pruhs and Stein [21] presented an alternative version of $\sum$Flow + Energy which incorporates job weights. The weight of a job is similar to the priority; a job with higher weight should be processed more quickly than another job with the same amount of work. This is a more general term than basic flow as it allows each job to have a weight which refers to its relative importance; this means that we prioritise jobs accordingly. This allows more control over which job will be processed next and which jobs are allowed to wait for long periods of time. If we set the priority of all jobs to the same value it is the same as standard Flow as suggested by Albers and Fujiwara. The authors present an algorithm Highest Density First (HDF) which is $O(\frac{\alpha^2}{\log^2 \alpha})$-competitive for the minimisation of $\sum$Weighted flow + Energy.

Lam et al. [93] presented 2 algorithms to address the $\sum$Flow + Energy minimisation problem in both the clairvoyant (where job sizes are known) and non-clairvoyant (where job sizes are known only after they have been processed) cases. For the clairvoyant case they develop AJC: the processor is set to speed $n^{\frac{1}{\alpha}}$ where n is the number of active jobs and α is from the power function. The jobs are processed according to shortest remaining processor time first (SRPT). This algorithm is shown to be more effective than Bansal's existing algorithms for both the bounded and infinite speed models: for the bounded speed model the algorithm is

$$\left(\frac{2(\alpha+1)}{\alpha - \left(\frac{\alpha-1}{\alpha+1^{\left(\frac{1}{\alpha-1}\right)}} \right)} \right) \text{-competitive} \tag{2.7}$$

For the non-clairvoyant case the authors minimise $\sum$Flow + Energy in the situation where all jobs are released at time 0. The algorithm AJC* uses processor speed $(\frac{n}{\alpha-1})^{\frac{1}{\alpha}}$ and round robin to schedule the jobs. They show this to be 2-competitive for the model where the maximum speed of the processor is bounded. Lam et al. [96] later produced an improved algorithm 'Shortest Remaining Processor Time' (SRPT) which reduced the competitive ratio to $\frac{\alpha}{\log \alpha}$ which equates to 3.25-competitive when $\alpha = 3$.

Bansal et al.[18] describe solutions where an arbitrary power function dictates the relationship between processor speed and power consumption instead of the usual $P = S^{\alpha} + c$ which other papers have considered. The authors present 2 algorithms: the first uses shortest remaining processor time; and the second uses highest density first. They show the former to be $(3+\epsilon)$-competitive for minimising $\sum$ flow + energy and the latter to be $(2+\epsilon)$-competitive for minimizing fractional weighted flow + energy. The fractional weighted flow of a job is the sum of the fraction of the total work remaining multiplied by the job weight for each time step.

Andrew, Weirman and Tang [14] further improved the best known solution to $\sum$Weighted Flow + Energy minimisation under arbitrary power functions. The authors develop an algorithm which uses SRPT instead of HDF and sets the speed of the processor to P^{-1}(Length of Queue) where P^{-1} is the inverse of the power function. They show this to be 2-competitive for a wide range of power functions. Furthermore they prove that no online algorithm can obtain a worst case competitive ratio of less than 2.

Andrew, Weirman and Lin [15] consider the problem of minimising

$\sum$Response time + Energy. They provide an algorithm which they prove to be 2-competitive and show that no natural speed scaling algorithm can do better. They also demonstrate that dynamic speed scaling allows systems to be robust against uncertain workloads. Finally they show that speed scaling increases unfairness when shortest remaining processor time is used to schedule jobs but that processor sharing remains fair. The authors assert that it is not possible (according to their results and existing systems) for speed scaling algorithms to be optimal, robust and fair but they can be any 2 of these objectives simultaneously.

Recently Bansal et al. [19] discussed speed scaling systems which minimise $\sum$Flow + Energy where only certain speeds are allowed. Li and Yao [107] first considered a dynamic speed scaling model where only certain speeds were allowed but this was for deadline scheduling. Bansal et al. showed that when the power is set to $j'+1$ (where j' is the number of unfinished jobs) and shortest remaining processor time is used for scheduling, the competitive ratio is 3 for minimising total flow + energy. They also show that using highest density first and setting the power to fractional weight of unfinished jobs is 2-competitive for minimising fractional weighted flow + energy.

There are countless other papers which address variations of the single processor dynamic speed scaling problem which focus on minimising $\sum$Flow + Energy. There have also been many reviews of dynamic speed scaling algorithms [76, 78, 3, 4, 5] and an analysis of algorithms and techniques [48].

Sleep States and Dynamic Speed Scaling

Algorithms which use Sleep States and DSS have existed for some time now but they are rarely considered as a pair of techniques which can be used together for even better results. Irani et al. [77, 78] were the first to look at this problem which they call DSS-S. The authors define an offline algorithm in [78] which they show to be 2-competitive compared to the optimal solution. It has yet to be proven whether the problem is NP-hard. They also define an online algorithm which is based upon a standard DSS algorithm.

Lam et al. [91] considered the use of DSS with deadline scheduling and sleep states. The authors define an algorithm IdleLonger which is based upon AJC [95] if it is clairvoyant and LAPS [40] if it is non-clairvoyant. They show that this algorithm is O(1)-competitive if it is clairvoyant regardless of maximum speed and O(1)-competitive if it is non-clairvoyant providing the maximum speed is

unbounded.

2.2.3 Multiprocessor Systems

Over the past few years the prevalence of multiprocessor computer systems has increased at an astounding rate. Multiprocessor systems have the potential to be much more computationally powerful compared to a single processor system as the maximum speed of a single processor is limited by the cube root rule which makes further speed increases unfeasible. Single processors also struggle to disperse heat as they operate at higher rates. These are both problems which can be overcome by using a multiprocessor system; this has led to a widespread increase in the use of multiprocessor systems.

Multiprocessor energy efficient scheduling was first considered by Bunde [34] who proved that the offline problem of power aware scheduling with multiprocessor systems to minimise the makespan is NP-Hard if all jobs require different amounts of work even if all jobs arrive at time 0.

Lam et al. [93, 94, 96] were the first to consider the online problem where the number of processors or cores is not bounded. They design a Classified Round Robin (CRR) algorithm which distributes jobs to processors based on their size in an attempt to balance the load to each processor. They then use the BPS algorithm [21] to define which job should be processed first on each processor and what speed that processor should operate at. This algorithm works best with homogeneous multiprocessor systems, where all processors /cores are equal.

Other researchers have considered variations on the multiprocessor low energy scheduling problem including [39, 136].

Heterogeneous Multiprocessor systems

Morad et al. [116] and Kumar et al. [87] have both argued for the development of heterogeneous multiprocessor systems in order to reduce energy consumption. Morad et al. suggest that heterogeneous multiprocessor chips could be the way in which we can get the most performance for a given power budget. They suggest that the chip should be configured to have a large proportion of low energy low performance cores, a smaller number of medium speed and medium performance cores and few high energy and high performance cores. The system would use the low energy cores for less urgent or demanding jobs and the medium and high speed processors for more demanding or urgent jobs. This would allow the system to combine performance and energy efficiency.

Bower et al. [28] outlined the need to consider the power efficient heterogeneous multiprocessor system, where processors can differ in available speed and power function amongst other things. In the position paper the authors outline the importance of considering this issue and go on to break this down into three main challenges:

1. 'The OS must discover the status of each processor',
2. 'The OS must discover the resource demand of each job',
3. 'Given this information about processors and jobs, the OS must match jobs to processors as well as possible'.

Gupta et al. [72] were the first to take up the problem outlined in [28]. In their paper the authors look at the third challenge: distributing jobs and calculating the processor speeds. They do this in a similar way to Lam et al. [96] by maintaining processor independence. The jobs are distributed to the processor which will result in the least increase in the projected flow assuming that no more jobs are to arrive. Then each processor calculates its own operational speed using $P^{-1}(\sum \text{Fractional Density}) = S$ as in [18] and finally the job with the highest density is run on the processor at any given time. The authors show that this algorithm is 'scalable for scheduling jobs on a heterogeneous multiprocessor with arbitrary power functions to minimize the objective function of weighted flow plus energy'. Gupta et al. [71] later extended their work to non-clairvoyant cases and showed that their solution is bounded speed, bounded competitive against the optimal solution.

Gupta, Im, Krishnaswamy and Moseley [70] have also published a paper which discusses the issues with low energy scheduling over heterogeneous multiprocessor systems. They find that many of the common scheduling techniques used for single processor algorithms are not bounded by a constant to the optimal solution for minimisation of weighted flow even when the special case of fixed speed processors are considered. They also suggest the first scalable non-clairvoyant algorithm for heterogeneous multiprocessor systems which uses late arrival processor sharing.

2.3 Boolean Algebra, Circuits and FPGAs

In this section we describe how energy is consumed within boolean circuits, integrated circuits and FPGAs. We also survey the techniques used to reduce

the power consumption of FPGAs.

Boolean algebra was first proposed by George Boole in 1854 [27] in his seminal work in 'Investigation of the Laws of Thought: On which are Founded the Mathematical Theories of Logic and Probabilities'. Wegener's [145] famous Blue Book defines a boolean function as $f : \{0,1\}^n \rightarrow \{0,1\}^m$. A boolean function can be expressed in a number of ways but the most fundamental is through the use of logical gates:

- Conjunction (and): iff x = y = 1 then 1 else 0
- Disjunction (or): iff x = y= 0 then 0 else 1
- Inversion (not): iff 1 then 0 else 1

Through the use of these few logical gates we can express all boolean functions [145]. There are combination gates (NAND, NOR, XOR etc.) but as we can express all functions (including the function of each combination gate) using the three gates listed above we shall not discuss the others here. Boolean algebra has formed the foundation for all of modern computing.

In 1937 Shannon [131] extended the logic of boolean algebra to form a new model called a boolean circuit. A boolean circuit is a model which is used to implement a boolean function. Boolean circuits use physical implementations of logical gates to realise boolean functions in circuit form. In electronic circuits the value 1 is usually represented as a higher voltage than 0. The gates can be implemented in a variety of different technologies which differ based on the desired use.

From very early on in the development of boolean circuits the energy consumption has been a significant consideration. Initially this may have been motivated by other related issues such as reducing the peak power consumption or improving reliability by reducing heat but this has since become a priority in its own right. Boolean circuits consume energy whilst charging and discharging the connections between the gates [109]. If the value of an edge has to switch from 1 to 0 then it must disperse the energy. Equally if the value has to switch from 0 to 1 then additional energy must be added.

In the 1950s many researchers began to consider the problem of reducing the energy consumption of boolean circuits. Much of this material was only ever published in German or Russian so we are unable to provide a comprehensive survey of the seminal work here. More recently research has been reinitialised

into reducing the energy consumption of VLSI circuits; for overviews and survey articles see [86, 2, 120, 53].

2.3.1 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are an extension of integrated logic circuits and were introduced in 1988 when Freeman [61] filed a US patent on behalf of Xilinx Inc.. The patent describes a configurable logic circuit similar to modern day FPGA circuits. An FPGA is a logic circuit which can be programmed (and depending on the technology used to implement the logic blocks normally reprogrammed) after it has been printed. This was revolutionary at the time and provided a circuit which could implement a wide variety of functions which saved time, money and energy in producing different boards for different uses and also enabled users to fix errors in the circuits logic without requiring new circuit boards.

There are a number of different technologies which can be used to implement a boolean circuit and a number of ways to implement different functionality. For example the logic sections of an FPGA were originally simple logic gates but have since been developed into configurable logic blocks which can represent small logic functions; and then into Look Up Table (LUT) logic blocks which can implement any boolean function with up to l inputs. In this thesis we shall concentrate on LUT based FPGAs as they are some of the most common at this time.

The problem of minimising the energy consumption of an FPGA has been considered for some time. Much research has been dedicated to developing accurate power estimation models for various components in an FPGA circuit [12, 13, 24, 101, 148]. Researchers have considered many different techniques to reduce the power consumption of an FPGA including utilisation of power down techniques [115], low power logic synthesis techniques [139] and glitch reduction techniques [47, 51]. There are many other low power techniques including [45, 50, 65, 99, 111, 119]. For a review of many different low power FPGA techniques see [97].

In an earlier section we saw that many components of an FPGA consume significant amounts of energy but by far the largest single consumer is dynamic routing [140]. Routing refers to the connections between the input, output and logic blocks (Look up Tables / LUTs) and the infrastructure used to implement the re-routability of the circuit. The majority of energy is consumed on these

edges when the circuit is active and they switch from 1 to 0 or the reverse. If the connection has a certain voltage then it is implied that the value is 1; if we wish to remove the value 1 then we need to disperse some energy. If the connection has a low voltage then the value 0 is implied and to switch the value to 1 we must apply more power. It is clear to see that if we increase the frequency of switching (known as switching activity) then the power consumption will increase; hence much of the research aimed at reducing the energy consumed by dynamic routing is aimed at reducing the switching of the edges.

One method suggested to reduce the average power consumption caused by dynamic routing involves altering the function of each LUT such that the overall circuit functionality is the same but the switching activity of the LUTs (and hence the power consumption) is reduced. Chen, Hwang and Liu [44] considered this problem in 1997. They utilise Roth Karp decomposition and local search techniques (Simulated Annealing and Kernighan-Lin) to modify the individual functions that each LUT implements whilst maintaining the same circuit function. The authors found that after applying their algorithms they achieved a greater than 9% average power reduction in comparison to the standard SIS mapping [130]. There are several other papers which present alternative solutions to the same problem [75, 88, 89].

Mashayekhi, Jeddi and Amini [111] introduced methods which reduce switching within each LUT block by inserting fake registers and then using a re-timing method. The authors implemented their methods for two ISCAS89 benchmark circuits and achieved a 25% power reduction over similar re-timing methods without power reduction considerations. Finally, Tinmaung, Howland and Tessier [139] developed logic synthesis methods to reduce power consumption. They achieved an average power reduction rate of 13% for Altura Cyclone II devices compared to the standard SIS logic synthesis methods.

These techniques have been shown to reduce the power consumption of FPGAs but in the following section we shall consider another approach which creates an initial mapping that considers the power consumption from the start rather than as a post layout optimisation.

Low Power LUT based FPGA Mapping

Since the advent of LUT based FPGAs researchers have been considering the best way to map an input function or circuit onto the available LUT based circuit. Algorithms have been developed which aim to minimise the area (physical

size), depth (speed of signal propagation), switching activity (power consumption) or a combination of any of the above [60, 49, 121, 45, 138].

Farrahi and Sarrafzadeh [59, 58] showed that the decision version of the problem is NP-complete even for simple classes of circuits (e.g. 3 level circuits). They then extended this to show that even restricted cases of LUT minimization for FPGA technology mapping are NP-complete [58]. Farrahi and Sarrafzadeh [59] considered mapping boolean circuits onto LUT based FPGAs in 1994. The authors developed a heuristic algorithm (Power Min) which maps the nodes onto k feasible cones (which are analogous to k feasible LUTs) whilst attempting to minimise average power consumption. It is shown that the heuristic described can reduce the power consumption by an average of 14.8% whilst using only 7.1% more LUTs compared to an algorithm designed to minimise area.

Wang and Kwan [143] suggested a heuristic mapping algorithm with the aim of reducing the power consumption whilst maintaining optimal area. The authors' algorithm first generates the LUT mapping which results in the least number of LUTs possible. The algorithm then adjusts the solution to hide the high transition paths inside LUTs which results in reduced power consumption whilst maintaining the number of LUTs. The algorithm reduces the power consumption by 10.38% compared to an alternative bin packing algorithm which guarantees minimum number of LUTs but does not attempt to reduce power.

Wang, Liu, Lai and Wang [144] proposed Power-Map: a heuristic algorithm which relies on a restricted cut enumeration technique to generate many possible solutions and select the best. Once the initial solution is built there is a brief search for a better solution before the final mapping is returned. The authors compare their Power-Map algorithm to the Power Min algorithm from [59]: Power-Map reduced the power consumption by between 14.03% - 14.18% and the number of LUTs by between 6.31% - 6.99% depending on the number of cuts the algorithm is allowed to consider.

Li, Mak and Katkoori [104] developed a multi objective technology mapping algorithm which aims to reduce the power consumption whilst ensuring that the circuit depth is kept optimally small. The authors exploit the fact that LUTs on the non-critical path (the longest path from input to output) can be modified without affecting the depth of the circuit. PowerMap first generates a minimum depth mapping solution and then computes min-height k-feasible nodes which are not on the critical path. The authors compare their algorithm (which is implemented in conjunction with SIS) to a minimum depth mapping algorithm, FlowMap. They find that PowerMap reduces the power consumption by 17.8%

and the number of LUTs by 9.4% with no depth penalty.

Anderson and Najm [11] developed a mapping algorithm which draws on a number of techniques and observations to reduce the power consumption of FPGAs. The authors attempt to map the boolean circuit such that high transition nets (areas with high switching activity) are removed from the routing infrastructure. They also consider logic duplication which has been previously shown to be essential for minimum depth LUT circuits. The algorithm is compared to FlowMap (which minimises depth) and FlowMap-r (which minimises depth whilst trying to reduce the total number of LUTs), both of which are combined with either FlowPack or MP-Pack. The researchers find that their algorithm uses less power, area and connections than any of the alternatives. The power reduction is less than other approaches (8% average) although the experimental analysis includes additional optimization on top of FlowMap which will affect the results.

Li, Mak and Katkoori [105] [103] develop a heuristic algorithm which attempts to minimise the power consumption of the mapping solution. The algorithms (Power Min Map and Power Min Map -d) first generate possible cut based solutions but take a global view when deciding which cut to accept at any point, opting for the cut which is more likely to reduce the power consumption of the overall circuit rather than just the best local cut. A network flow min-cut method is used to compute the initial solution which is then adjusted to further reduce the power consumption using the author's 'cut frontier refinement' method. Power Min Map is compared to Power-Map [144]: Li et al. find that on average Power Min Map reduces the energy by 12.2% and the number of LUTs by 10.6%.

Pandey and Chattopadhyay [122] present the first stochastic algorithm to address the problem of FPGA LUT circuit mapping. The authors begin by reducing the problem to a binate covering problem and then use a genetic algorithm to search for a good solution. The authors compare their algorithm to a basic SIS map and find that they reduce the power consumption by 25.51%. The authors claim that the algorithm in [144] only reduces switching activity by 10% in comparison to the SIS solution which suggests this is a good solution.

Mashayekhi [111] consider a solution which inserts fake registers (which cannot be within the LUTs) into the circuit to force the mapping solution to contain certain low transition edges with the hope that this will hide the high transition edges from the routing edges. Finally the solution is optimised using re-timing methods to further increase the quality of the solution. The authors analyse

their solution using randomised input variables instead of the probabilistic approach used in many other papers. The experiments are limited in their scope as they only consider 2 benchmark circuits and do not compare to any viable alternative solution; instead the authors first apply their algorithm without power optimisation and then again with power optimisation. The experiments show the authors' algorithm reduces the power consumption by 25% for one circuit and 11% for the second circuit.

Bucur, Stefanescu, Supateanu and Cupcea [32] design a mapping tool which builds on the SIS circuit tool. This approach differs from others in that it uses a Monte Carlo simulation to estimate the energy consumption rather than a probability-based approach which most publications use. The tool attempts to minimise the power consumption whilst also considering depth and area. The authors present 3 different solutions and compare them to one another; this makes it hard to compare this approach to others listed here.

Chen, Wei, Zhou and Cai [46] have developed a heuristic algorithm, PowerMap_er which considers both power consumption and edge count simultaneously. The algorithm first generates all cuts for all nodes and maps a solution. It then applies an area-edge recovery method 'depth slack distribution' and finally it recomputes the edge cost. The authors compare the algorithm to Power Min Map -d [103] (Power = -8.5%, Area = -8.4%) and MacroMap (an area optimal algorithm) [146] (Power = -18%, Area = -7%). We must bear in mind that the figures quoted in this paper are maximum improvement rather than average improvement as quoted in many other papers.

2.4 Optimisation Techniques

Minimising the power consumption of a computer system is a hard problem. There are many different components to optimise individually and collectively. There are numerous optimisation techniques which have been applied to various parts of the overall power optimisation process. In this section we give an overview of some global optimisation techniques with a particular focus on Simulated Annealing as this is the method which we employ later in the thesis.

The Oxford English Dictionary [147] defines optimisation as:

Definition 3 *The action or process of making the best of something; (also) the action or process of rendering optimal; the state or condition of being optimal.*

When we talk about optimising the power (or energy) consumption of a

computational system we wish to make it such that the average power (or total energy) consumption is minimal in comparison to the level of performance being achieved such that we either use the least possible energy for a given level of performance or maximise the performance for any amount of energy.

Some optimisation problems can be relatively simple either because the input size is small or due to the configuration of the problem domain. For these problems there have been many simple heuristic algorithms which have solved the problem to optimality in a reasonable time. In general, global combinatorial optimisation problems are much more complex. Many have been proved to be NP-hard or NP-complete and require much more sophisticated algorithms to reach optimal or near optimal solutions in reasonable time. There have been many optimisation algorithms suggested and below we discuss a small number of those which we feel have been important or influential.

Branch and bound algorithms were first proposed as a method for solving discrete and combinatorial optimisation problems by Land et al. in 1960 [98]. Branch and bound begins by assuming that any solution could be the optimal solution. It then divides the solution space into 2 or more separate 'branches' according to some criteria and finally it computes the upper and lower bounds for each branch: if the lower bound of any branch is greater than the upper bound of any other branch then that branch is rejected as it cannot contain the optimal solution. This process is repeated until only the set of optimal solutions remains. Branch and bound is a particularly powerful deterministic algorithm and has been used to solve many hard problems to optimality [100].

Evolutionary algorithms are a set of algorithms which take their inspiration from the way in which biological evolution occurs i.e. natural selection. An evolutionary algorithm first generates a number of solutions; this is the first generation. From this point each solution is assessed using a fitness function. Some solutions (usually the weaker solutions) are rejected and the others (usually the strongest) remain. The remaining solutions are then combined with each other using inheritance, crossover and/or mutation to produce the next generation. This process is repeated until some threshold has been reached which could be quality of the solution, time elapsed or number of generations. At this point the best solution which has been found will be returned. The term Evolutionary Algorithm (EA) is now also used to refer to a category of algorithms which are inspired by natural selection. Genetic algorithms fall within evolutionary algorithms and have been applied to many hard optimisation problems and found many optimal or good results [122, 22, 132].

Particle Swarm optimisation [84] is another school of algorithms which takes their inspiration from nature, in this case the movement of large groups of animals. Again a number of initial solutions (particles) are generated which are called the swarm. Each particle then moves through the search space looking for better solutions based on the best solution it has found and the best solution the whole swarm has found. This enables the swarm as a whole to find very good solutions for hard problems with very little knowledge of the problem domain [150] (this is a trait common to many meta-heuristics). Particle swarm optimisation relies on the particles moving around the solution space to discover good solutions. Many other algorithms have used a similar idea to find good solutions; these algorithms are often referred to as local search algorithms.

2.4.1 Local Search

Local search algorithms explore the solution space using local moves. A local move is defined as a small change in an existing solution such that the overall solution is changed; for example in scheduling this may be swapping the order in which 2 jobs are scheduled. A neighbourhood function is the combination of a number of local moves which is used to explore the solution space of a problem. A neighbourhood function is of most use when it is complete: where the local move set can be used to traverse the entire solution space from any solution to any other solution. This ensures that the local search algorithm which utilises the neighbourhood function has a chance to reach each and every solution and therefore to find a globally optimal solution. Defining a neighbourhood function and proving its completeness can be a hard problem in itself [1].

There are a number of local search algorithms which have been deployed to find optimal solutions to hard problems. The most simple is the hill climb (or decent) algorithm which operates as follows. From the current state we make a local move; if the new solution improves the objective function then the move is accepted, otherwise look for an alternative move. This is repeated until we reach a state where no local move can improve the function. This is said to be a locally optimal solution or if the problem is convex (or concave) the globally optimal solution. Hill climb algorithms have been adapted to find better solutions by running the algorithm many times (possibly in parallel) using multiple (possibly random) start points. Hill climb based algorithms have been shown to find good solutions for a number of different problems [69, 137, 149].

Tabu search [66, 67] is another example of an optimisation algorithm which

uses the local search method. Tabu search explores the solution space using local search but uses a short term memory to avoid settling in a local minima or visiting the same solution many times in quick succession. This has the effect of helping the algorithm to overcome local optima and give the algorithm a better chance of finding the global optimum solution. Tabu search has been used to find good solutions for many different problems and has had numerous positive results [64, 117, 68].

2.4.2 Simulated Annealing

Simulated Annealing is a local search based global optimisation technique which was first proposed by Kirkpatrick et al. [85] in 1983, later independently by Ĉerny [36] in 1985 and is based on the METROPOLIS' method [113]. The general idea behind the algorithm is to utilise the method in which metal is cooled to aid the optimisation of hard problems. In order to ensure the metal is strong it is important that the crystalline structure is free from defects. This is achieved by heating the metal until it is liquid and then allowing it to cool slowly such that the molecules can move around to find their optimal position before they become fixed. When the temperature is high the metal molecules move around freely into configurations which are less optimal but as the metal gets cooler and the metal begins to set the molecules become less likely to move to a position which would result in more defects.

In hard optimisation problems it is common to have many locally optimal solutions (where any local move would result in an increase to the objective function) but a very small number of globally optimal solutions. In these situations a simple greedy heuristic algorithm which only accepts moves which improve the solution would have a high chance of terminating in a non optimal solution. Kirkpatrick et al. use the analogy with metal annealing to inform their choice to allow the algorithm to accept transitions from one state to another which result in a less optimal solution depending on a certain algorithmic parameter called the temperature. This allows the algorithm to find very good solutions and the optimal solution in cases where the algorithm converges.

A simulated annealing algorithm consists of the following components:

- Initial Solution
- Neighbourhood Function
- Acceptance criteria

- Cooling schedule
- Stopping criteria

The initial solution is usually randomly generated but can be chosen based on some criteria to improve the quality or speed of solution. The neighbourhood function should be complete and allow the algorithm to navigate through all possible solutions. The acceptance criteria is the probability that any generated solution will be accepted, this usually takes the following form.

$$a = \min(e^{\frac{(f(x)-f(x'))}{c(k)}}, 1) \tag{2.8}$$

where a is the probability of acceptance, e is the base of a natural logarithm, f is the objective function, x is the current solution, x' is the new solution, c is the cooling schedule and k is the current step. This equation always accepts a solution which improves the objective function and accepts worse solutions with a probability linked to the current temperature and change in objective function.

The cooling schedule should be slow, ideally such that the algorithm has enough time to converge onto the optimal solution. Simulated annealing algorithms can be categorised by their cooling schedule and if one or many steps are made for each temperature. Those algorithms which make many steps at each temperature are modelled by a homogeneous Markov chain which under some natural assumptions tends to the Boltzmann distribution [10, 79]. If the temperature is changed after each step the system is modelled by an inhomogeneous Markov chain as the probabilities change for each step.

There are many general cooling schedules, such as equation (2.9) below, which have been tweaked and applied to a number of unconnected problems and have found many good and optimal results. There are also many cooling schedules which have been developed with particular problems in mind; these have also found good and optimal solutions for hard problems.

$$\begin{aligned} c(0) &= \text{Starting Temperature} \\ c(k) &= c(k-1) \cdot \tfrac{1}{\beta} \end{aligned} \tag{2.9}$$

where $c(0)$ is the manually set starting temperature, $c(k)$ is the temperature at step k and β is a user set parameter which controls the speed of the cooling.

Simulated annealing has been applied to many combinatorial optimisation problems and has been responsible for improving the upper bound and finding

optimal / near optimal results for many hard problems where other algorithms have failed. For example Steinhöfel, Albrecht and Wong [134] applied heuristic simulated annealing to the job shop scheduling problem. In the paper they showed that the algorithm could find optimal solutions for a number of benchmark problems and improved the best known solutions for several more.

Simulated annealing algorithms fall into the category of meta-heuristic algorithms which find very good solutions for hard problems but do not by nature guarantee that the optimal solution will be found. If the convergence of an algorithm can be proved then we can be sure that the algorithm will find the optimal solution before termination.

Convergence

Lundy and Mees [108] studied the convergence of homogeneous simulated annealing algorithms. They developed a formal model and then showed that the algorithm must converge on the optimal solution with probability arbitrarily close to 1. We outline the general model and theory below.

We begin by defining F to be the set of feasible solutions and $F_{min} \subseteq F$ to be the set of optimal feasible solutions.

Consider a pair of feasible solutions $[s, s']$ where $s' \in \eta(s)$. $G[s, s']$ denotes the probability of making a transition from s to s' and η is the neighbourhood function. This is calculated according to equation (2.10):

$$G[s, s'] = \left\{ \begin{array}{ll} \frac{1}{|\eta(s)|} & \text{if } s' \in \eta(s) \\ 0 & \text{otherwise} \end{array} \right\} \tag{2.10}$$

where $|\eta(s)|$ is the size of the neighbourhood i.e. number of feasible local moves from s.

$A[s, s']$ denotes the probability that the transition from s to s' will be accepted; this is calculated according to equation (2.11):

$$A[s, s'] = \left\{ \begin{array}{ll} 1 & \text{if } f(s') < f(s) \\ a = e^{\frac{f(x)-f(x')}{c(k)}} & \text{otherwise} \end{array} \right\} \tag{2.11}$$

where f is the objective function and $c(k)$ is the temperature at step k as defined above.

Therefore the probability of picking and accepting a solution s' when in state

s is given by $Pr\{s \to s'\}$ which is defined in equation (2.12) below:

$$Pr\{S \to S'\} = \left\{ \begin{array}{ll} G[s,s'] \cdot A[s,s'] & \text{if } s \neq s' \\ 1 - \sum_{s \neq Q} G[s,Q] \cdot A[s,Q] & \text{otherwise} \end{array} \right\} \tag{2.12}$$

We also define $a_s(k)$ which denotes the probability of being in solution s after k moves. This is set according to equation (2.13) below:

$$a_s(k) = \sum_{Q} a_Q(k-1) \cdot Pr\{Q \to s\} \tag{2.13}$$

This recursive definition allows us to incorporate all routes from any starting solution to s.

One can consider equation (2.13) as a Markov chain of probabilities for any route from any starting solution to the final solution s. If the optimal solution is reachable from any starting solution with a non-zero probability then the following convergence probability can be shown for an infinite Markov chain:

$$\lim_{c \to 0} \left(\lim_{k \to \infty} \sum_{s \in F \neq F_{min}} a_s(k) \right) \to 0 \tag{2.14}$$

$$\lim_{c \to 0} \left(\lim_{k \to \infty} \sum_{s \in F_{min}} a_s(k) \right) \to 1 \tag{2.15}$$

Theorem 1 *The Markov chain defined by equations 2.11, 2.10 and 2.13 has a probability of 1 to be in an optimal feasible solution $s_{min} \in F_{min}$ after ∞ steps and for a decreasing temperature $c \to 0$*

Lundy and Mees [108] show that although a simulated annealing algorithm will converge they allow an infinite number of steps at each temperature and therefore the convergence time is not bounded.

Mitra et al. [114] suggested the assumption that infinite time can be spent at each temperature was unrealistic. Mitra et al. [114] presented their own proof which considers the time-inhomogeneous model and shows that the algorithm will converge on the optimal solution without the need for time freezing used in [108].

Albrecht showed that a logarithmic cooling schedule $c(k) = \gamma / \ln(k+2)$ converges onto the optimal solution with a probability $1 - \sigma$ after $k > (n/\sigma)^{O(\gamma)}$ steps [10].

There are many textbooks and reviews of simulated annealing algorithms which have been used extensively throughout this thesis [73, 129, 135].

Chapter 3

Low Power Scheduling for Power Heterogeneous Multiprocessor Systems

Mobile computing has advanced considerably over the past decade. Hardware development and minimisation of smart-phones has been broadly consistent with the well known Moores Law which states that the price will halve or performance (number of transistors on a chip) will double every 18 months. The major exception to this has been the development of the battery technology, which has sorely fallen behind advances in other technologies. This forms a design challenge which must be addressed through research into both battery technology and power reduction techniques. In this chapter we consider the problem of reducing the power required.

One of the largest drains of energy in a computer system is the processing unit. In most modern processors, energy consumption and processing speed are intrinsically linked; this is normally through the relationship

$$P = S^{\alpha} + c \tag{3.1}$$

where α is a constant which is typically between 2 and 3, c is some constant, S is speed and P is power. A very effective way of reducing the amount of energy a processor uses is by lowering the operational speed. For this we can use Dynamic Speed Scaling.

This chapter discusses Dynamic Speed Scaling with power heterogeneous multiprocessor systems. Power heterogeneous multiprocessor systems have a collection of processing units all of which have one or more cores. Each processor or core has a power function and set of valid operating speeds. The variety of possible operational speeds and power functions makes the systems significantly more complex than homogeneous multiprocessor systems which have been the focus of the majority of existing work.

The remainder of this chapter is organised as follows: in Section 3.1 we present an overview of the problem being discussed; in Section 3.2 we explain the motivation of the VSP approach; in Section 3.3 we present the DynaVSP algorithm; in Section 3.4 we explain how a VSP can be used in conjunction with other algorithms; the results of our experiments are presented in Section 3.5; and Section 3.6 concludes the chapter.

3.1 Background

Dynamic Speed Scaling (DSS) allows the operating speed of a processor to be modified at runtime. Due to the polynomial relationship between speed and power a sustained small reduction in processor speed can result in a large reduction in total energy consumption. Using DSS to lower the operational speed of a processor or processors is simple but deciding by how much the speed should be reduced is complex. An objective function is used to manage the relationship between energy consumption and performance.

We recall from 2.2.2 that Albers and Fujiwara [8] presented $\sum$ Flow + Energy: an objective function which balances the quality of service against energy consumption. Bansal, Pruhs and Stein [21] extended Albers and Fujiwara's objective function to include job weights $\sum$ Weighted Flow + Energy; this allows for jobs to be differentiated by importance.

Andrew, Wierman and Tang [14] presented an algorithm which has obtained the best competitive ratio to date. The algorithm considers a wide range of power functions and has a competitive ratio of $(2 + \varepsilon)$. The authors also show that there exist some trade-off functions for which no algorithm can be better than (2)-competitive.

3.1.1 Power Heterogeneous Multiprocessor Systems

Power heterogeneous multiprocessor systems is a term used to refer to any multi processor or multi-core computer system which contains processors or cores which are not identical with regards to their power function and possibly their set of available speeds. The beauty of this type of system is that it can contain a complementary set of processors which can be utilised in a very efficient way. Heterogeneous multiprocessor systems have the potential to be very adaptable allowing the computer to be both energy efficient and computationally powerful. Power heterogeneous multiprocessor systems are not currently the most common type of multiprocessor systems but they are more common than one might think. Many multi-processor systems may be heterogeneous due to manufacturing discrepancies or system setup.

We recall from 2.2.3 that Bower, Sorin and Cox [28] identified 3 main hurdles for heterogeneous multiprocessor scheduling:

1. the OS must find the status of the processor,
2. the OS must find the demands of each job and
3. the OS must match jobs to processors as well as possible using the available information.

Gupta, Krishnaswamy and Pruhs [72] were first to suggest a solution for scheduling weighted jobs onto speed scaling processors. The authors focus on the 3rd problem identified in [28] of organising which jobs should be processed on which processor and when. They suggest a simple algorithm which they describe in three parts:

1. Job Selection (which job should run on each processor): Highest Density First
2. Speed Scaling (what speed should each processor run at): The speed is set so the power is the fractional weight of the unfinished jobs
3. Assignment (which processor should each job be assigned to): A new job is assigned to the processor that results in the least increase in the projected future weighted flow, assuming the adopted speed scaling and job selection policies and ignoring the possibility of jobs arriving in the future

The approach which Gupta et al. [72] suggest involves distributing jobs to various processors based on which one would provide the smallest increase in

the projected flow providing no more jobs arrive and the speed of the processors does not change. The processors are then allowed to manage their own speed based on the volume of work they are carrying and the speed scaling policy defined by the algorithm. In the paper the authors prove this algorithm to be 'scalable for scheduling jobs on a heterogeneous multiprocessor with arbitrary power functions to minimize the objective function of weighted flow plus energy'.

Gupta, Krishnaswamy and Pruhs [71] subsequently published a paper which considers the problem of scheduling unweighted jobs non-clairvoyantly over power heterogeneous processors. The authors show the natural non-clairvoyant algorithm they present is bounded-speed and bounded-energy competitive.

3.1.2 Low Energy Scheduling

In [72] the authors present a solution for power heterogeneous multiprocessor systems using $\sum$ Weighted Flow + Energy as their objective function. This solution has been shown to be theoretically sound with the authors proving that the approach is scalable. When we consider a real multiprocessor computer system we often find that there are constraints which complicate the problem domain. For example many multi-core processors require the cores to always run at the same speed. This could be a particular problem as we could include multi-core processors within a heterogeneous multiprocessor system. The approach suggested is not currently compatible with this kind of architecture as the speed of each processor is linked to the number of jobs and not to any other processor.

Another potential issue with the [72] approach is that it requires a significant amount of runtime computation. Each time a job needs to be assigned to a processor the algorithm states that we must calculate which processor will provide the smallest increase in the projected total weighted flow. The authors do not describe an exact algorithm for calculating which processor which will result in the least increase in the total weighted flow; we therefore cannot state an exact amount of run time computation but we can outline a lower bound.

In order to calculate the total weighted flow we must know at what time each active job will finish processing, which has a worst case running time of at least $O(j')$ where j' is the total number of active jobs. The algorithm would then have to compare the increase in projected weighted flow for each processor which would take $O(p)$ time where p is the number of processors. Any algorithm must therefore have a worst case running time of at least $O(j + p)$ and must run every time a new job is being assigned to a processor.

We present a solution that has the ability to overcome both of these issues: the Virtual Single Processor (VSP) approach. A VSP is essentially a collection of processors which have been combined together in a pareto optimal way with regards to overall system speed and power. The VSP is presented (as a single processor) to a DSS algorithm which controls the speed of the overall VSP and specifies which job should be processed first. The VSP in turn translates the VSP speed into speeds for each processor such that the sum of all processor speeds is equal to the VSP speed.

3.2 The Virtual Single Processor

We can think of a multiprocessor system as being a tree graph. The root of the tree is the system level: this is where jobs arrive to be passed down to a processor or core to be processed. The leaves of the tree are the processor level; each leaf represents a processor or core which can be used to process work. Finally the internal nodes represent connections between processors; for example a multi-core processor would have a number of cores connected by an internal node which is then connected to the root (e.g. MC0 in the Figure 3.1).

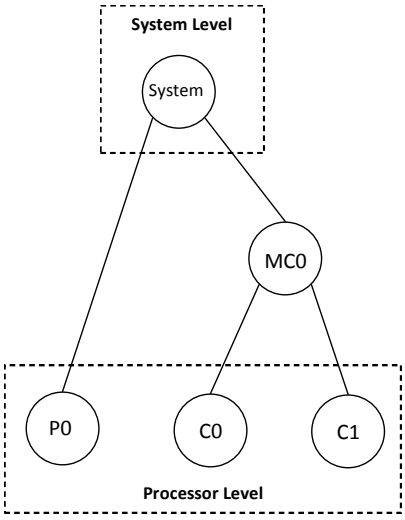

Figure 3.1: A system tree showing the system and processor levels

When using a heterogeneous multiprocessor system the existing solutions

suggest that we distribute the jobs at system level and control the processor speeds at individual processor level as in [96] and [72]. We present an alternative solution where we consider controlling processor speeds and assigning jobs to processors from a system level according to processor speed and job priority.

Consider an example of a 4 processor system (P0, P1, P2, P3). Each processor has a set of speeds and a simple power function in the form of $P = S^{\alpha}$, the attributes of which are outlined below.

P0 (0, 200,300,400, 500) α=2.3

P1 (0, 600, 700,800,900) α=2.35

P2 (0, 100, 300, 500, 700, 900) α=2.5

P3 (0, 1200) α=2.2

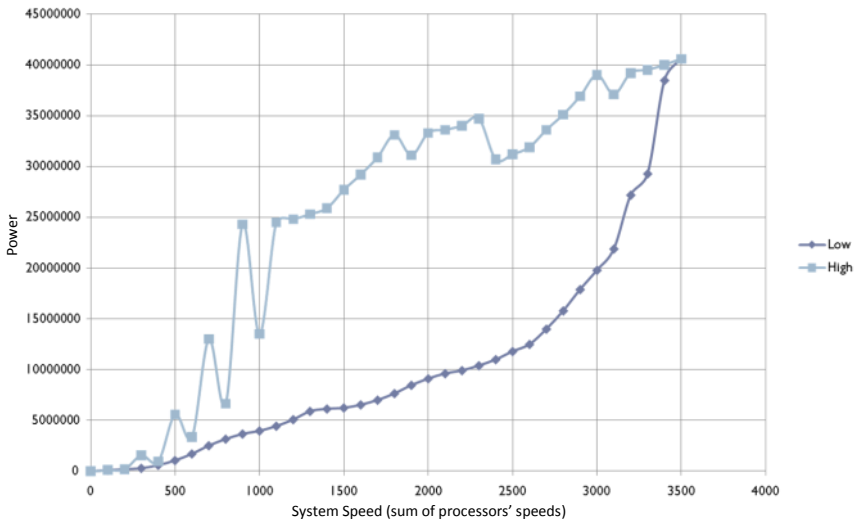

Figure 3.2: System speed vs. power consumption for the best case and worst case processor combinations.

In this simple example there are many ways to combine these processors with 300 unique combinations of processors and speeds. Each combination can be represented by total power requirement and system speed (the total of all individual processors' speeds). There are 14 different combinations which make up the overall system speed of 1400 alone. If we consider the best and worst ways of achieving the system speed of 1400 (with regards to power) we find

that the worst case uses 487% of the power consumed by the most efficient combination. If we look at Figure 3.2 we can see the difference between the most and least power is largest in mid range speeds and the graph converges at either end of the system speed range. All processors must be at speed 0 for the system speed to be 0 and all at maximum speed for the system speed to be maximised.

This simplified case highlights how crucial it is to use the best processor combinations. If we pre-compute the optimal processor combinations before attempting to use a multiprocessor system we can ensure that we always use the most efficient processor combinations. To simplify the search we describe 2 observations which help to find the optimal processor combinations.

Observation 1 *A processor combination can only exist in the optimal VSP if there is no other processor combination which requires less power for an equal or greater system speed.*

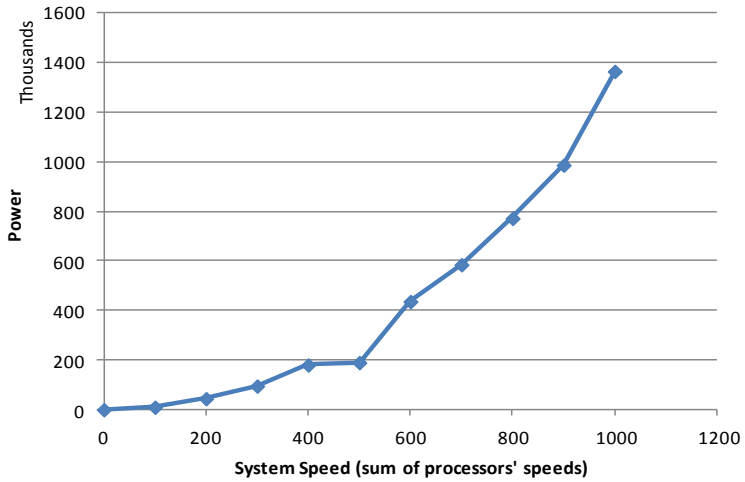

Figure 3.3: The VSP solution after applying Observation 1

Observation 1 is intuitively correct. If a solution V contains a processor combination V_x which can be replaced with another $V_{x'}$ which uses less power and provides an identical or higher system speed then we have found a new solution V' which is more efficient than V, hence V cannot be optimal.

Observation 1 severely reduces the amount of combinations but does not always result in the set of optimal combinations. Some system speeds can

be more efficiently implemented by alternating between two different systems speeds rather than using a combination which is allowable if we only apply Observation 1.

The final method which we use to further reduce the lower envelope of the power function is speed simulation which is described in [18]. Bansal et al. first defined this method in order to show that a processor with a restricted set of speeds could simulate any speed in the range of 0 - max by alternating between 2 different speeds. For example if we have a processor with 2 available speeds (0 and 10) then we can simulate the speed 5 by alternating between the two available speeds in equal amounts. If we wished to simulate the speed 7.5 then we would use speed 10 for $\frac{3}{4}$ of the time and speed 0 for $\frac{1}{4}$ of the overall time. We use this method in order to potentially lower the power required by the VSP.

Observation 2 *If a VSP speed s can be simulated by alternating between two different speeds and the simulated speed requires less power then s is not part of the optimal VSP.*

Figure 3.4 shows an instance in which Observation 2 is used to lower the overall power function of the resulting VSP; the lighter section of the line shows the improvement over applying Observation 1 alone. After applying both Observation 1 and Observation 2 we have computed the optimal VSP as it is not possible to achieve a higher system speed for any power. The remaining set of processor combinations are pareto optimal with regards to power and system speed.

A major advantage of the VSP approach is that it provides a level of abstraction between the single processor algorithm and the multiprocessor system. This abstraction allows us to hide the complexity of the multiprocessor system behind the VSP front. We can hide a multitude of requirements such as processors or cores which always need to operate at the same speed by first producing a small VSP which encapsulates these parameters and then nesting this inside the overall VSP as if it was a single processor.

By pre-computing the VSP we can also remove the burden of calculating which processor is best for each job. This is made possible as the VSP hides the fact that more processors exist and only assigns jobs when a processor has a speed greater than 0 and no job. This means that there is no need for the algorithm to perform costly calculations at run time, which is clearly a great advantage.

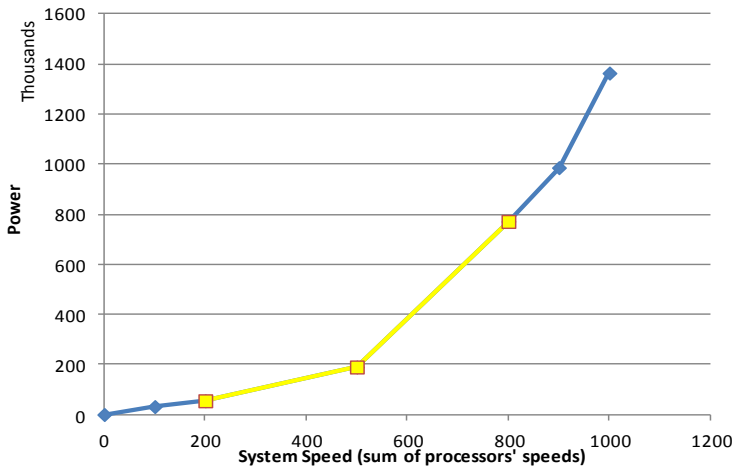

Figure 3.4: The VSP solution after applying Observation 1 & 2

3.2.1 Power Function

When a VSP is computed we obtain a pareto optimal set of processor combinations that can be queried in two ways:

1. Given a system speed the VSP will respond with the processor combination which uses the least amount of power.

2. Given a power level the VSP will deliver the highest possible system speed and the processor combination to achieve it.

We can see 1 as the system power function and 2 as the inverse of the system power function. This is crucial as it allows us to apply a variety of single processor energy reduction algorithms to a power heterogeneous multiprocessor system. For example the single processor $\sum$Weighted Flow + Energy algorithm suggested by Bansal et al. [18] states that the speed of the processor is set to $P^{-1}(n_A^t + 1)$ where P^{-1} is the inverse of the power function and n_A^t is the number of unfinished jobs at time t when applying algorithm A. This algorithm can now be applied to a multiprocessor system if used in conjunction with the VSP approach. Instead of consulting the inverse of the power function we query the VSP to find the maximum speed which can be achieved using $n_A^t + 1$ power.

3.3 Our VSP Algorithm

In this section we define our DynaVSP algorithm which calculates an optimal VSP given a set of processors as an input.

Algorithm 5 DynaVSP

DynaVSP(List[ ] P)

Input: A set of processors or VSPs P of length n

```
if n == 1 then
  return P[0]
end if
if n == 2 then
  VSP v = new VSP
  for each speed x in P[0] do
    for each speed y in P[1] do
      v.addSystemSpeed(x, y)
    end for
  end for
  for each SystemSpeed x in v do
    Check if x is optimal using Observation 1 and 2
  end for
  Return v
end if
if n > 2 then
  Return DynaVSP( [DynaVSP(P[0...n/2]),DynaVSP(P[((n/2) + 1)...n])] )
end if
if n < 1 then
  return null
end if
```

Output: A single VSP

A simple implementation of the DynaVSP algorithm has a worst case running time of $O(s^{2p})$ based on p processors, with each processor having s speeds and an unrestricted power function; in the worst case this is no better than a naive algorithm. The difference is that the naive algorithm performs at this level consistently whereas the DynaVSP algorithm only performs at this level in very few specific situations. The only time the DynaVSP algorithm takes $O(s^{2p})$ time is when every possible processor combination can be part of the optimal VSP. This is because every possible s^p processor combination must be assessed which takes $O(n^2)$ time.

This rare case only occurs when all of the possible combinations are arranged on one strictly non-decreasing line. In the vast majority of situations we find that the points are fairly evenly distributed within an elliptical shape which joins the maximum and minimum speeds, i.e. between the low and high lines in Figure 3.2. In the majority of cases we find that the DynaVSP algorithm calculates the optimal VSP in a reasonable amount of time.

3.4 Using the Virtual Single Processor

It is important that the VSP is not seen as a complete scheduling algorithm. It is a platform which allows scheduling algorithms to be applied to or developed for heterogeneous multiprocessor systems more easily. Once we have constructed our VSP we need three things to utilize it:

1. A job selection policy.
2. A speed scaling policy.
3. To know whether the computer system will allow migration between processors or not (this point is crucial to job selection and processor speed changes).

In the two following subsections we outline how the VSP can be used. This is split into two parts based on whether the multiprocessor system allows migration or not.

3.4.1 Migratory

Incoming jobs are sorted according to their ranking as judged by the job selection policy. The job with the highest rank is always assigned to the fastest processor, the second highest rank with the second fastest processor and so on. This is maintained even when more jobs arrive or processor speeds change. This ensures that the job with the highest priority always finishes quickly. The speed scaling policy is used in conjunction with the system power function to determine what speed our system should operate at. This is then translated into individual processor speeds by the VSP. If a processor is directed to use speed 0 then the job it is currently processing is interrupted and returned to the list of incoming jobs.

3.4.2 Non-Migratory

Once again the incoming jobs are sorted according to their rank as judged by the job selection policy but we also keep a note of 2 things for each processor:

1. The time required to finish the current job being processed if the processor speed stays constant.

2. The current speed of the processor.

We then calculate which processor will allow the highest priority job to finish first, providing processor speeds stay constant, and the second highest to finish second and so on. Jobs are then assigned to the 'correct' processors when they become available. When the speed scaling algorithm decides that the system speed should change, the VSP converts this into individual processor speed changes: if the speed of a processor should rise then this happens straight away; if the speed of a processor should drop then this action is taken after the processor has finished processing the current job. This ensures that no job is trapped on a processor which has speed 0 as this could result in the job never being finished.

3.5 Experimental Analysis

In this section we describe our experimental methods and then state and interpret our results. We test the VSP approach in 2 ways. First we assume the VSP can know and match the speed of an alternative algorithm [72] at any time; we record the total energy consumption and compare the 2 approaches. The second test is a straight simulation test: we implement the VSP and alternative algorithm, simulate a range of processor architectures and input sets and record the total energy consumption and total weighted flow.

We begin by discussing the speed matching results and then move onto the simulation results.

3.5.1 Speed Matching Results

We have shown in Section 3.2 that there exist optimal processor combinations that allow us to get the maximum possible system speed for our energy outlay. By using optimal processor combinations we can either gain a free system speed upgrade with no additional power cost or reduce the power with no degradation

in speed. Conversely if we do not use optimal processor combinations then we are needlessly wasting energy or reducing quality of service. It therefore follows that if we can show that [72] does not consistently use optimal system speeds then the VSP can provide a better solution by simply mirroring the overall system speed decided by [72] but using optimal system speeds. This would reduce the energy consumption whilst maintaining the system speed. We shall assess the efficacy of the VSP approach in comparison to that of the [72] approach by showing that the above is true.

We checked to see if the [72] approach consistently used optimal system speeds through simulation based testing. First we generated and stored a number of heterogeneous multiprocessor systems. The systems were generated at random within the given constraints of maximum speed, number of processors and number of speeds. Table 3.1 shows a breakdown of the types of multiprocessor systems. We generated 3 systems of each type to further broaden the diversity of our test set and bring the total number of test systems to 36.

Processors	**Number of Available Speeds**
2	3 4 5
4	3 4 5
8	3 4 5
16	3 4 5

Table 3.1: Randomly Generated Heterogeneous Multiprocessor Systems

The set of randomly generated heterogeneous multiprocessor systems used in the experiments can be downloaded here:

`http://www.dcs.kcl.ac.uk/pg/dobsonr/HeteroMultiProcessors.zip`

For each test we took one of the heterogeneous multiprocessor systems and created increasing numbers of identical unit size jobs. The [72] algorithm assigned jobs to the processor which would result in the least increase in projected future weighted flow. Each processor then specified its own speed based on the

number of jobs currently assigned to it. At regular intervals we noted the total number of jobs, the overall system speed and the overall energy consumption of the system; we then calculated the energy consumption that would be required to achieve the same system speed if we were using the VSP approach and stored the data.

The test was terminated once the number of jobs had increased to such a volume that the [72] approach had forced all of the processors to reach their maximum speed i.e. maximum system speed.

Number of processors	**Average amount of non-optimal processor combinations**	**Average energy reduction across range**
2	51.07%	4.427%
4	81.708%	5.935%
8	87.527%	7.114%
16	92.371%	8.175%
Average	78.169%	6.413%

Table 3.2: Results for the randomly generated processors experiment

Raw data can be accessed here:

`http://www.dcs.kcl.ac.uk/pg/dobsonr/SpeedMatchingData.zip`

Table 3.2 and Figure 3.5 and Figure 3.6 present the results from the experiments using the randomly generated heterogeneous multiprocessor systems. We can see from Figure 3.5 that the [72] approach does not consistently use optimal system speeds: on average they were not used for 78.169% of the system speed range. In addition to this we can also see the detrimental effect this has on the energy consumption with the VSP approach reducing the energy consumption by an average of 6.413%. Interestingly, as the number of processors increases, the [72] approach uses a decreasing amount of optimal system speeds. This is most likely due to the growing number of possible processor combinations which makes it less likely that the optimal system speed and processor combinations will be used.

Figure 3.6 shows that as the number of processors increases, the energy saving gained through using the VSP approach also increases. This is of great interest as future projections are that the number of processors could reach as high as 16 within mobile devices over the next few years. Figure 3.7 is a graph showing an example of a typical test. We can see that the [72] line converges to

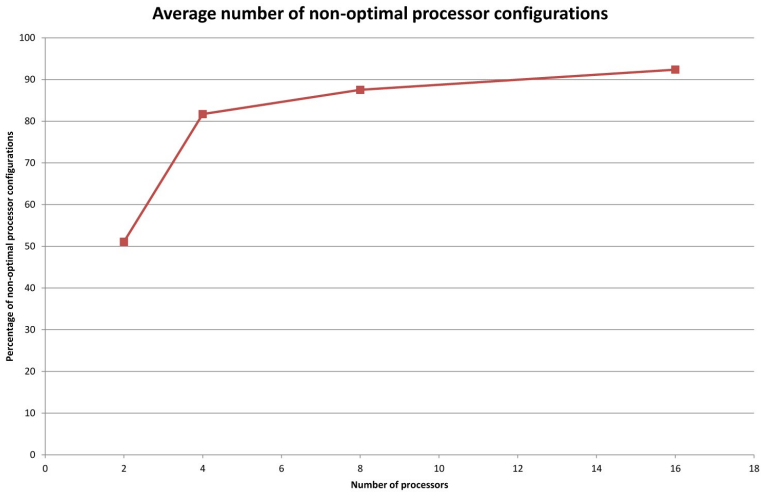

Figure 3.5: Non-optimal processor configurations used by [72] vs. number of processors

the VSP line at either end of the system speed range and the largest difference appears in the mid-range. We would expect a multiprocessor system to operate in the mid-range for the majority of time so the potential energy saving is maximised.

Our results show that the VSP approach can consistently reduce the energy used by a heterogeneous multiprocessor system in comparison to the existing approach in [72]. This means that we can bind more tightly to the objective function (Quality of Service + Energy Consumption) as we can match the system performance whilst reducing the energy consumption.

3.5.2 Simulation Results

In this section we combine the VSP system with the speed scaling policy and job selection policy from the [72] algorithm. We then compare the two approaches using simulations. If the results from the two differing approaches are similar then this will show that the VSP has the potential to be a favourable alternative.

We define 2 multiprocessor systems, which have identical processor configurations and neither allow migration. The first system is used in conjunction with the [72] approach and speed scaling algorithm A. The second system is used as a VSP and also uses speed scaling algorithm A. The VSP approach takes the

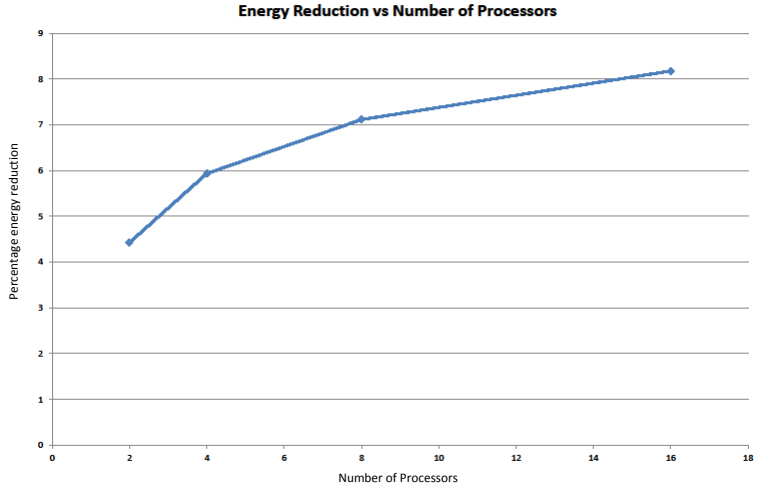

Figure 3.6: The percentage energy saving compared to [72] vs. number of processors

system and uses the VSP algorithm to convert it into an optimal VSP.

We consider a batch of t tasks arriving over time we can compare how each approach will deal with these. The [72] approach will sort the jobs by their density and then calculate which processor will provide the least increase in projected flow for each job. The job is then assigned to this processor. Each processor will calculate the speed it should be running at based on the fractional weighted flow of its work.

The VSP approach will sort the jobs by their density and then calculate what the speed of the VSP should be. The VSP will then instruct the processors as to what speed they should be running at. Jobs are assigned to a processor if its speed is greater than 0 and it does not already have a job. Jobs with higher priorities will be assigned to faster processors. Note that the VSP approach has allowed us to remove the majority of the computation from run time due to the pre-computation of the optimal processor configurations.

Simulations of both approaches were developed in Java and a number of tests were run with a variety of processor configurations and job sets. Java source code can be accessed at the location below:

`http://www.dcs.kcl.ac.uk/pg/dobsonr/VSP_JavaSim.zip`

In this section we will highlight the tests regarding a processor configuration

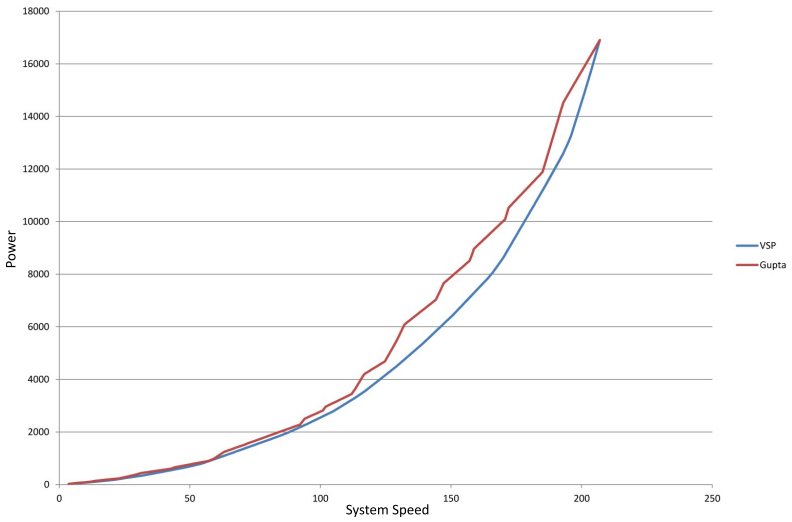

Figure 3.7: Results of a test for a 16 processor system

outlined in [71].

We use the processor configuration from [71] as the authors suggest this is an architecture which allows the system to be both energy efficient and powerful. We have y high powered processors, $2y$ medium powered processors and $4y$ high powered processors (and $y = 1$). Therefore our system setup is:

- 1 High speed processor: $S = \{0, 1000, 2000\}$ $\alpha = 2.8$
- 2 Medium speed processors: $S = \{0, 250, 500, 750, 1000\}$ $\alpha = 2.55$
- 4 Low speed processor: $S = \{0, 50, 100, 150, 200, 250\}$ $\alpha = 2.25$

We split the test data into 3 different categories all of which contain jobs with random weights and sizes:

- Immediate: all jobs are all released at time 0.
- Uniformly random: jobs are released randomly over time.
- Peaks and troughs: jobs are released in surges which are similar to the action of a computer system.

Covering these categories allows us to see how the system performs under a variety of conditions this gives the tests significant experimental validity. The

set of jobs used in these experiments can be downloaded here:

`http://www.dcs.kcl.ac.uk/pg/dobsonr/VSP_Test_Jobs.zip`

Our results are reported in terms of VSP performance in comparison to the [72] algorithm. We found that there were some jobs which were severely throwing the average finish time for both the VSP and [72] algorithms. We therefore have included additional results where a few outlying jobs have been disregarded; these jobs have both a very low weight and a very large size. After removing the outlying jobs we recalculate the averages and have reported this in Figure 3.8.

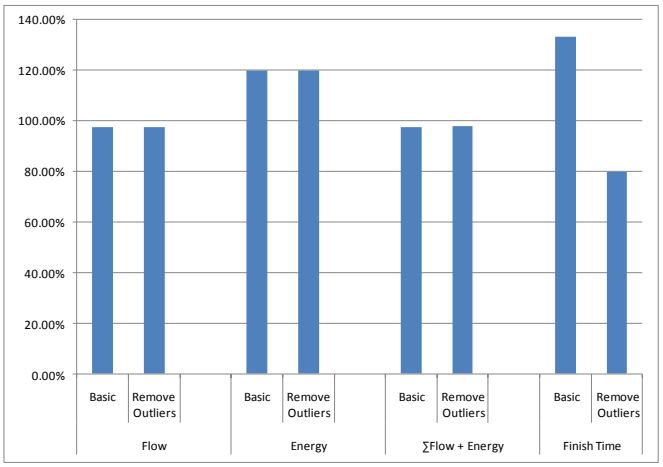

Figure 3.8: Average results from the simulations with and without outliers

The results graph shows that the VSP approach has bounded tighter to the objective function of $\sum$ Weighted Flow + Energy minimisation as it has reduced this by a small margin. We find that in this situtation the VSP algorithm has used more energy but reduced the average weighted flow; this is due to the algorithm which the VSP is teamed with rather than the VSP system itself.

The results show us that the VSP platform is very promising as it has bounded 2.31% tighter to the objective function than a competitive alternative. Our test results are based on using the same speed scaling and job selection algorithm as the [72] approach to allow for a fair comparison but if used in conjunction with the Andrew et al. [15] algorithm we may well have found even better results.

3.6 Conclusions

In this chapter we presented the Virtual Single Processor (VSP) approach to low energy scheduling for power heterogeneous multiprocessor computer systems. We showed that this approach has many advantages over the existing solution:

- A significant reduction in runtime computation.
- The ability to cater for multiprocessor systems with complex requirements.
- Significant reduction in energy consumption.

We defined a recursive algorithm (DynaVSP) to calculate the optimal VSP, showed that the VSP approach is theoretically sound and that it can address many drawbacks of [72]. Our experimental results show that [72] does not consistently use optimal processor configurations and the VSP approach reduced the average energy consumption by between 4.4% (2 processor system) and 8.175% (16 processor system) compared to [72]. In addition we found that the VSP approach can reduce the objective function of $\sum$Weighted Flow + Energy by 2.31% compared to the best alternative [72]. This shows that the VSP approach is a favourable option when looking to use a heterogeneous multiprocessor system in an energy efficient way.

Chapter 4

SA based Power Efficient FPGA LUT Mapping

Logic circuits are an integral part of computer science and ubiquitous in everyday life. Many logic circuits take the form of Application Specific Integrated Circuits (ASICs) which are developed for a specific purpose and once manufactured their function is fixed. Field Programmable Gate Arrays (FPGAs) are logic circuits with additional technology which allows them to be programmed (and reprogrammed, depending on the technology) after being manufactured.

FPGAs have a number of significant advantages over ASICs: they can be reprogrammed such that one physical board can be used for many different applications, reducing the number of boards which need to be manufactured. Also, if a defect in a logic circuit is found it can be corrected without the cost and energy required to print another physical circuit. This makes FPGAs ideal for a multitude of applications including mobile and distributed computing.

The technology which allows FPGAs to be programmed after manufacturing means that they are more complex, can require more physical chip space, cost more to produce and consume more power than an equivalent ASIC. As a result they have to be programmed very carefully. As mobile or distributed devices usually rely on a limited power supply unit, such as a battery or renewable power source, it has become increasingly important to develop FPGAs which are exceptionally power efficient.

The power consumption of an FPGA can be broken down into static power and dynamic power. Static power is consumed whenever there is power running

through the circuit regardless of activity. Dynamic power is consumed when the circuit is active and accounts for 62% of total power consumption for a Xilinx Spartan-3 device [141]. Figure 4.1 outlines the power consumption of various components.

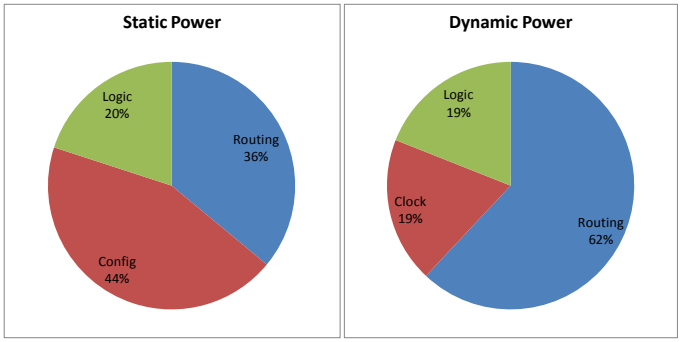

Figure 4.1: Power consumption of a Xilinx Spartan-3 device [140, 141]

By far the largest proportion of the power consumption is the cost of dynamic routing which accounts for 38.44% of the total power consumption of a Xilinx Spartan-3 device. Power consumed by dynamic routing is dependent on the switching activity of the routing edges which is dictated by the switching activity of each logic block and the length of each connecting path. In the majority of modern FPGAs logic blocks are implemented using Look Up Tables (LUTs) which take k_{in} inputs and return a single result.

There have been many algorithms developed which aim to reduce the amount of power consumed by dynamic routing. One of the most fruitful areas under consideration is the computationally hard problem of mapping an input boolean function (usually in the form of a boolean circuit) onto an LUT-based FPGA such that the power consumption is minimised.

The vast majority of existing solutions (which are outlined in the literature review) make use of greedy heuristic algorithms which have the disadvantage of often finding only locally optimal solutions rather than the global optimum. Simulated annealing-based algorithms have had significant success with other hard problems with similar properties and have found globally optimal or near optimal solutions in an acceptable time period. We anticipated that they can provide equally strong results for our problem.

In this chapter we develop a complete local search move set and present a simulated annealing-based algorithm which maps a boolean function onto an

LUT-based FPGA such that the power consumed through dynamic routing is minimised.

We begin by formally defining the problem considered and outlining the power estimation technique used to evaluate solutions. We introduce our local moves and neighbourhood function, prove they are complete and present the Simulated Annealing algorithm which is tailored to the problem at hand. Finally we compare our results to both SIS mapping [130] and a genetic algorithm [122] and show that our algorithm finds better solutions than both of these approaches.

4.1 Problem Definition

A boolean circuit is defined as a directed, acyclic graph $G(N, E)$ where N is a set of nodes and E is a set of edges. Each node $n_i \in N$ is either a primary input, an output or a logical gate (although a logical gate can also be an output node). Node types are distinguished by their in-degree and out-degree (see Table 4.1). Each node also has a transition density $td(n_i)$ which is the number of times the signal changes in unit time. Each directed edge $e_i \in E$ connects one node to another. Figure 4.2(A) is an example of a boolean circuit. The total estimated average power consumption of a boolean circuit is shown in equation (4.1).

$$\begin{aligned} P_{avg}(B) = \quad & \textstyle\sum_{i=0}^{n_{in}} (\frac{1}{2} C_{in} V_{dd}^2 f_i td(n_i)) + \\ & \textstyle\sum_{i=n_{in}+1}^{n} (\frac{1}{2} (C_{in} + C_{out}) V_{dd}^2 td(n_i)) \end{aligned} \tag{4.1}$$

where $P_{avg}(B)$ is the average power consumption of boolean circuit B, n_{in} is the number of input nodes, C_{in} and C_{out} are the circuit capacitances, V_{dd} is the circuit voltage, n is the total number of nodes, f_i is the fan out of input i and $td(n_i)$ is the transition density associated with node n_i. The transition density of a node is defined to be the number of times that it's logical value switches in unit time.

	In degree	Out degree
Primary input	0	≥ 1
Output	≤ 2	0
Logical Gate	2^*	1

Table 4.1: In degree and out degree of node types

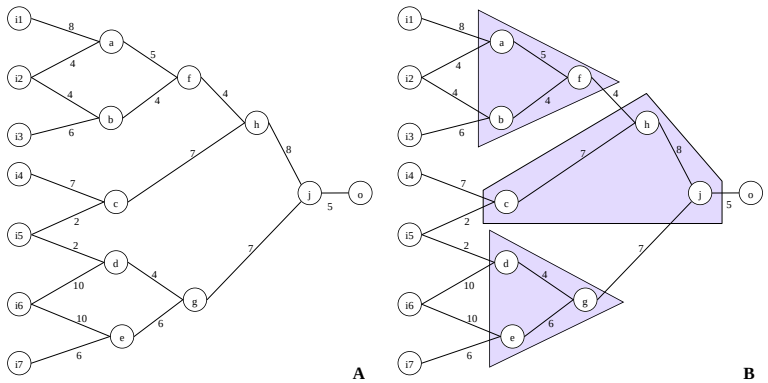

Figure 4.2: A: boolean circuit. B: LUT mapping of A.

A Look Up Table (LUT) circuit can be defined as a directed, acyclic graph $C(N', L, k_{in}, E')$ where N' is a set of nodes, L is a set of LUTs and E' is a set of edges. Each node $n'_i \in N'$ is either a primary input node or an output node and has a transition density $td(n'_i)$. Each $l_i \in L$ is an LUT. An LUT is defined as a special node that can represent any boolean function with up to k_{in} distinct inputs (where $k_{in} \geq 2$) and a single output i.e. an LUT node can replace a boolean circuit (or sub-circuit) with no more than k_{in} input nodes and one output node. Every LUT has a transition density $td(l_i)$ which is analogous to the switching activity of the LUT. Each directed edge $e'_i \in E'$ connects one node or LUT to another node or LUT. The total estimated average power consumption of an LUT circuit is described in equation (4.2).

$$\begin{aligned} P_{avg}(L) = \quad & \textstyle\sum_{i=0}^{n}(\frac{1}{2}C_{in}V_{dd}^2 f_i td(n'_i))+ \\ & \textstyle\sum_{i=0}^{l}(\frac{1}{2}(C_{in}+C_{out})V_{dd}^2 td(l_i)) \end{aligned} \tag{4.2}$$

where all terms are the same as in equation (4.1) with the following additions: L is an LUT-based circuit and l is the total number of LUTs.

A boolean circuit can be implemented by an LUT circuit. A single LUT can represent any boolean circuit (or sub-circuit) with k_{in} or less input nodes and one output node; an LUT circuit can therefore implement the same function as a boolean circuit by substituting sub-circuits for LUTs. The average power consumption of the equivalent LUT circuit is equal to the boolean circuit that

*We state that the in degree of a logical gate is 2 and the out degree is 1. This does not restrict the power of the circuit as any circuit with unbounded gates can be decomposed into an equivalent boolean circuit with bounded gates [145]

contains only the input nodes and output node of each sub-circuit. Figure 4.2(B) shows an example of how boolean circuit 4.2(A) can be mapped onto an LUT circuit where $k_{in} = 4$ and the grey areas represent LUTs. An LUT circuit is considered to have mapped a boolean circuit if the LUT circuit implements the same function as the boolean circuit.

We wish to find a k_{in} feasible LUT mapping of any boolean circuit such that the average expected power consumption is minimised.

4.2 Simulated Annealing

In this section we describe the simulated annealing algorithm which is applied to the problem described above. We begin by outlining a general homogeneous simulated annealing algorithm and a basic implementation of a simulated annealing algorithm (Algorithm 6) which incorporates all of the features discussed in this section.

A simulated annealing algorithm attempts to find an optimal (or at least good approximate) solution to a hard problem by searching the solution space using local moves. Each time a local move is made the new solution is evaluated according to the objective function. If the new solution is better than the current solution it is accepted; if not the new solution is accepted according to the equation (4.3).

$$a = e^{\frac{f(x)-f(x')}{c(k)}} \tag{4.3}$$

where a is the probability the solution will be accepted, f is the objective function, x is the current solution, x' is the new solution and $c(k)$ is the current temperature.

Equation (4.3) relies on $c(k)$ which is defined as the current temperature but more accurately it is the temperature at step k (which in equation (4.3) is the current step). The temperature is defined in 2 parts: $c(0)$ which is the starting temperature; and $c(k)$ which recursively defines the temperature at any step $k > 0$. $c(k)$ also describes the cooling schedule.

We have chosen to define $c(k)$ (equation (4.4)) using a cooling schedule which has been shown to have be very successful for similar problems. Steinhöfel et al. [134] suggest that $c(0)$ should be set such that even the worst possible move has a high probability of being accepted; this depends on the problem being considered.

$$c(k) = c(k-1) \cdot \frac{1}{\beta} \tag{4.4}$$

where β is an algorithm parameter and k is the step of the algorithm.

The temperature is lowered at each iteration of the algorithm according to equation (4.4) until $c(k) < c(\tau)$ where $c(\tau)$ is the threshold. The threshold should be set such that there is a very low possibility that even the least bad move will be accepted; this again depends on the problem being considered.

The homogeneous simulated annealing model states that a number of local moves should be performed at each temperature rather than adjusting the temperature after each move. The length of the Markov chain U (number of moves made and accepted) at each step is determined by the following equation.

$$U = h\eta \tag{4.5}$$

where h is an algorithm parameter and η is the size of the neighborhood.

Finally we require a complete neighborhood function which is defined in Section 4.3 below.

Algorithm 6 Simulated Annealing

```
bestSol = currentSol = a random solution
k=0
while c(k) > c(τ) do
  moveCount = 0
  while moveCount < U do
    newSol = LocalMove(currentSol)
    if newSol is accepted by equation (4.3) then
      moveCount++
      currentSol = newSol
      if f(currentSol) < f(bestSol) then
        bestSol = currentSol
      end if
    end if
  end while
  k++
end while
Return bestSolution
```

4.3 Move Set and Neighborhood Function

In this section we define a complete move set which allows us to transform one k_{in} feasible LUT mapping to any other k_{in} feasible LUT mapping through a series of local moves and describe a method to evaluate the energy consumption of the new covering.

We considered a number of possible local move solutions: node collapsing (and dividing); graph cutting (as used in existing heuristic algorithms); and finally node flag flipping which is the solution we settled on for a number of reasons. Firstly node collapsing (where adjacent logical nodes are combined) requires a significant amount of internal representation, manipulation and validation to execute each move which hampers the performance. Graph cutting (where the tree of logical nodes is iteratively split into disjoint sections) is powerful but requires the set of possible moves (which for larger values of k_{in} can be cumbersome) to either be generated at run time or precomputed and stored, both have major disadvantages including time and space complexity. We decided to use node flag flipping (which is defined in the remainder of this section) as it can be implemented very efficiently in terms of space and time complexity which allows our algorithm to run at the speed required.

In the remainder of this section we describe our local moves and neighbourhood function. We also show that they are complete for traversing the set of all possible solutions. We begin with some definitions.

Definition 4 *Let a each node have a flag which denotes if the outgoing edge is cut or uncut, where all input nodes and the output node must be labeled cut.*

Definition 5 *Let the state of a boolean circuit be an array of all node flags.*

Definition 6 *Let a partition be a boolean circuit which is divided into distinct sections by cut edges. A partition can be represented by a state.*

Definition 7 *Let a k_{in} Feasible Partition be a partition where each section can be implemented by a single k_{in} input LUT.*

A boolean circuit with l logical gates has 2^l different states but not all of these correspond to a k_{in} feasible partition.

4.3.1 Slightly Restricted Boolean Circuits

We begin by considering a slightly restricted version of a boolean circuit where each input node may have only one outgoing edge; see Table 4.2.

	In-degree	Out-degree
Primary input	0	1
Output	1	0
Logical Gate	2	1

Table 4.2: Restricted in-degree and out-degree of node types

Local Move 1 *Given a boolean circuit and a state which represents a k_{in} feasible partition: pick a single Logical Gate and invert the flag. This gives 2 distinct moves:*

- *Cut node: flip the flag on a node from uncut to cut*
- *Uncut node: flip the flag on a node from cut to uncut*

If the resulting state(s) correspond to k_{in} feasible partition(s) then the move is valid.

To make a valid move we can compute all possible moves which result in a k_{in} feasible partition. This can be achieved in $O(l)$ time using a simple BFS based algorithm (a stripped down version of Algorithm 7). Each time a move is made the initial list of all possible moves can updated to reflect the new circuit; this requires worst case time $O(l)$. The worst case only occurs when the local move affects every LUT within the circuit which can only happen when the circuit is trivially small. With sufficiently large circuits the average case the performance in the region of $O(k_{in})$.

Completeness

For a neighbourhood under Local Move 1 to be complete over the set of restricted boolean circuits we need to show that any k_{in} feasible partition can be transformed into any other k_{in} feasible partition through the repeated application of local moves within Local Move 1. To simplify this problem we can consider the following sub-problems:

1. Is the move set directly reversible?
2. Can we use the local move to transition from an initial k_{in} feasible partition to any other k_{in} feasible partition.

If we can prove that the both of the above are true then we have shown that we can move from any k_{in} feasible partition to any other k_{in} feasible partition.

Showing that Local Move 1 is reversible is simple. We begin in k_{in} feasible partition p_x, make a cut (or uncut) node move by flipping the flag on node n_i and transition to another k_{in} feasible partition $p_{x'}$; if the new partition is not k_{in} feasible then the move is not valid. We know that by making the opposite uncut (or cut) node move by flipping the flag on node n_i we must transition back to the original k_{in} feasible partition p_x; hence Local Move 1 is directly reversible.

Definition 8 *Let p_0 be the state where all nodes in a boolean circuit are cut. This corresponds to a k_{in} feasible partition where each node is implemented by a separate LUT.*

Definition 9 *Let p_1 be the state where all logical gate nodes in a boolean circuit are uncut. If there are k_{in} or less inputs this corresponds to a k_{in} feasible partition where all nodes are implemented in a single LUT.*

In order to show that we can use Local Move 1 to reach any k_{in} feasible partition we first simplify the initial problem. We can consider any k_{in} feasible partition as a combination boolean circuits with k_{in} or fewer input nodes and where all logical gates are uncut, i.e. the circuit contains a single k_{in} feasible LUT. See Figure 4.3 for an example of how a Boolean circuit in a complex k_{in} feasible partition can be considered as separate circuits each in state p_1.

By using this method we can, without loss of generality, just consider the situation where we take a boolean circuit with up to k_{in} inputs in state p_0 and use Move Set 1 to transform it into state p_1. This proves that the move set is complete.

Lemma 1 *There exists a chain of uncut moves (swapping a node from cut to uncut) which can transform a boolean circuit with k_{in} or less inputs from state p_0 into state p_1.*

Lemma 2 *A boolean circuit as defined in Section 4.3.1 which has $n-1$ logical gates must have n input nodes and 1 output node.*

The boolean circuit that we define in Section 4.3.1 adheres to a tree structure and a tree with v leaves will have $v-1$ non-leaf nodes where one is the root, therefore Lemma 2 must be true. Lemma 2 shows that the number of input nodes is directly proportional to the number of logical gates. If the maximum number of input nodes allowed is less than or equal to k_{in} then we can have at

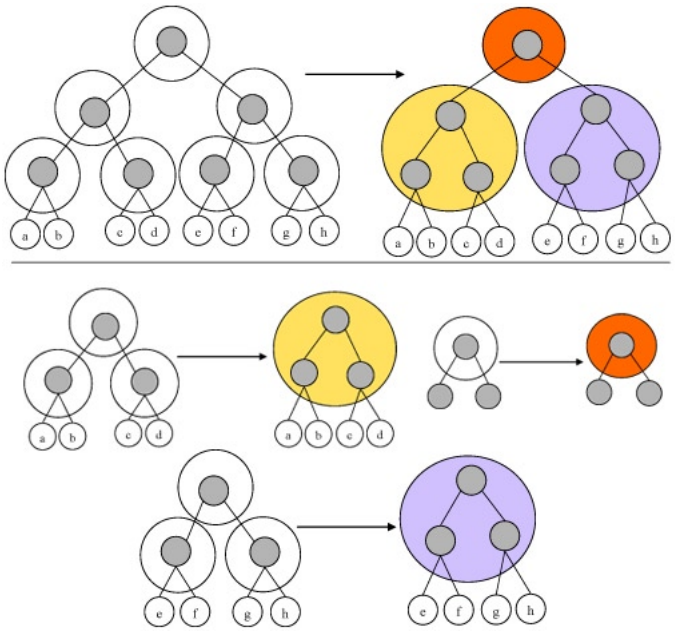

Figure 4.3: A complex partition as a combination of p_1 partitions

most $k_{in}-1$ logical gates. If the maximum number of logical gates is $k_{in}-1$ then any intermediate states must have less then $k_{in}-1$ logical gates and therefore less than k_{in} inputs, i.e. a sub-tree cannot have more leaves than the original tree. We can therefore use the uncut move to transition a boolean circuit from state p_0 to state p_1.

We have proved that we can transition a boolean circuit from state p_0 to state p_1 using Local Move 1 and that Local Move 1 is reversible. By combining these two elements we know that any k_{in} feasible partition can be reached from any other k_{in} feasible partition and hence we have proved that the Local Move 1 is complete.

4.3.2 Unrestricted Boolean Circuit

In the previous sub-section we showed that Local Move 1 is complete over our initial restricted definition of a boolean circuit. We now consider the original boolean circuit definition where each input node has unrestricted out degree.

The neighbourhood over Local Move 1 is not complete over the unrestricted definition of a boolean circuit. This is because Lemma 2 cannot hold as nodes

can now share input nodes; therefore a partition may have more than $k-1$ nodes. To account for this we include additional moves which are defined in Local Move 2.

Local Move 2 *Given a boolean circuit and a state which represents a k_{in} feasible partitioning: pick both children of a node; if the children are not marked that they should always be cut and are either both cut or both uncut then we can invert both flags. This gives 2 distinct moves:*

- *Cut Children: flip the flag on the nodes from uncut to cut*
- *Uncut Children: flip the flag on the nodes from cut to uncut*

If the resulting state(s) are k_{in} feasible partition(s) then the move is valid.

We can use an extended BFS based algorithm to produce a list of all valid moves in $O(l)$ time (Algorithm 7). For brevity we assume that when we refer to the neighbourhood over local move 2 we are implicitly referring to any move which is valid in Local Move 1 or 2.

Algorithm 7 FindValidMoves

Input: Boolean circuit B, State S, Set of valid moves M, Current node c

```
Find all LUTs (output, inners and inputs)
for each LUT (can be multi threaded) do
  for each inner node do
    if |LUT_i.inputs \ node.inputTree.inputs| ≤ k_in then
      M = M∪ node
    end if
  end for
  for each input node do
    if |LUT_i.inputs ∪ LUT_node.inputTree.inputs| ≤ k_in then
      M = M∪ node
    end if
  end for
end for
Return current
```

Completeness

In order to prove the completeness of Move Set 2 we need to show that:

- the move set is reversible

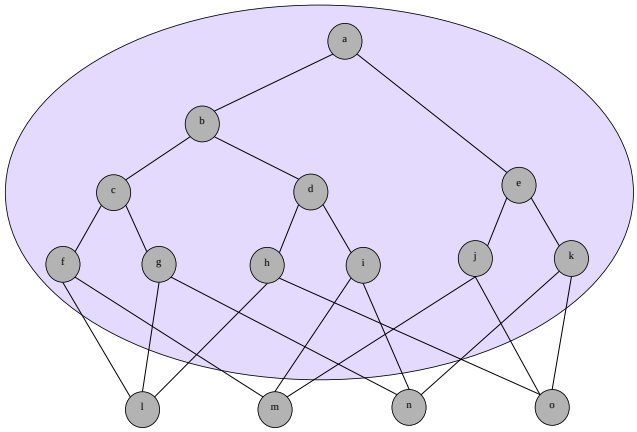

Figure 4.4: A circuit adhering to Lemma 3

- the move set works in situations where Lemma 2 does not hold

In section 4.3.1 we showed that the cut and uncut node moves were reversible very simply. This case is very similar as we are just applying 2 cut node moves at the same time and hence the logic remains the same. We can only use a cut (or uncut) children move if the resulting partition is k_{in} feasible and by applying the opposite move we transition back into the original k_{in} feasible partition. Hence Local Move 2 is directly reversible.

For Lemma 2 to not hold the number of nodes must be greater than or equal to the number of inputs. This can only happen when at least one input node has a fan out greater than 1 and more than one of the fan out edges are inputs of the same LUT.

Definition 10 *O is the set of circuits which have a k_{in} feasible partition where at least one section (LUT) has k_{in} input nodes and at least k_{in} logical gates, i.e. the set of circuits with k_{in} feasible partitions where Lemma 2 does not hold.*

Lemma 3 *There exists a boolean circuit $B \in O$ in p_1 with the set of input nodes I, the set of logical gates W and a single output node o where $|I| \geq |W|$.*

Figure 4.4 is an example of a circuit in partition p_1 where the number of logical gates $\{a, b, c, d, e, f, g, h, i, j, k\}$ is greater than the number of input nodes $\{l, m, n, o\}$. We have therefore found an a circuit where the conditions of Lemma 3 hold and hence Lemma 3 is true.

Lemma 4 *There exists a boolean circuit b_1 (from Lemma 3) in a k_{in} feasible partition p_1. We label the logical gate which is closest to the output node g. If both child nodes of g are cut then the resulting partition must be k_{in} feasible.*

The children of g (which we label g_l and g_r) may be input nodes or logical gates. This gives 3 distinct possibilities:

1. g_l is an input node and g_r is a logical gate,
2. g_l is a logical gate and g_r is an input node,
3. Both g_l and g_r are logical gates.

Note that we have omitted the case where both g_l and g_r are input nodes as the input nodes by definition must be cut so there is no local move is possible.

Cases 1 and 2 are equivalent as they both contain one input node and one logical gate. The description of a boolean circuit states that an input node must be cut so only the logical gate needs to be cut and therefore we can apply Local Move 1. By cutting the logical gate node we are left with a new partition with 2 sections; the first contains only one node g and the second contains all other logical gates. The first section containing g only can have only 2 inputs and hence must be k_{in} feasible. The second section (containing all logical gates apart from g) has an input set $I' \subseteq I$ therefore the number of inputs can be at most k_{in}; the partition is therefore k_{in} feasible.

In case 3 both children are logical gates and hence they both need to be cut and we must apply the cut children move (Local Move 2). Once both cuts are applied we reach a new partition with 3 sections; the first contains only g, the second contains the sub-circuit attached to g_l and the third contains the sub-circuit attached to g_r. The first section containing g only can have only 2 inputs and hence must be k_{in} feasible. The second and third may have between 1 and $|L| - 2$ logical gates in each sub-circuit. Each section has an input set which we will label I_l and I_r where $I_l \subseteq I$ and $I_r \subseteq I$ and $I_l \cup I_r = I$. As each section may have at most $|I|$ inputs the partition must be k_{in} feasible.

We have shown that all cases lead to a new k_{in} feasible partition and hence Lemma 4 must be true. We can now apply Lemma 4 recursively to decompose a circuit from partition p_1 to partition p_0 as is shown in Figure 4.5 ($1 \rightarrow 4$: uncut children and $4 \rightarrow 1$: cut children). We have shown that the move set is reversible and any k_{in} feasible partition can be reached from any other k_{in} feasible partition; therefore the Move Set 2 is complete.

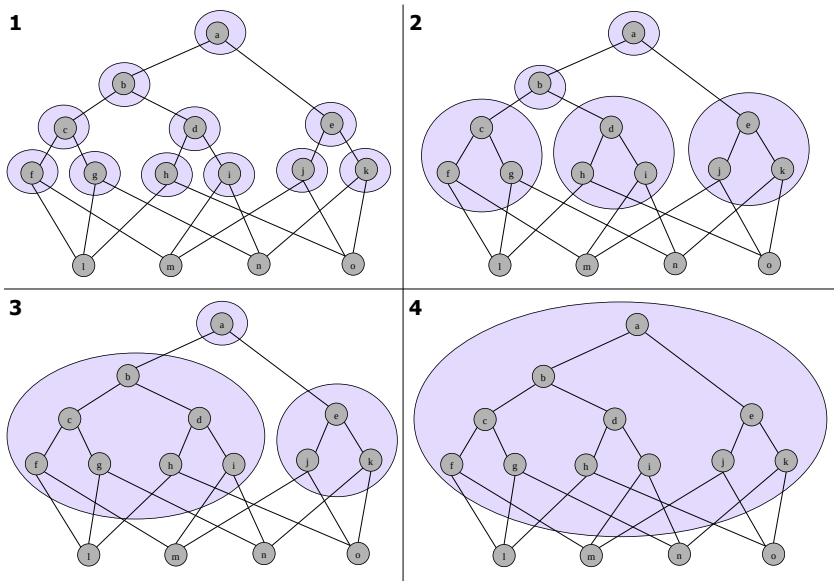

Figure 4.5: Repeated application of Local Move 2

4.4 Simulated Annealing Parameters

In this section we formally define the parameters used in the simulated annealing algorithm.

The starting temperature is set such that any move should be accepted with a high probability. We calculate this by rearranging the acceptance probability to make the $c(0)$ the object:

$$\begin{aligned} a &= e^{\frac{f(c)-f(n)}{c(k)}} \\ c(0) &= \frac{\Delta^w}{\ln pr(\Delta^w)} \end{aligned} \tag{4.6}$$

where Δ^w is the increase in the objective function when the worst possible move is made (which can be simply computed in $O(l)$ time before the simulated annealing algorithm begins) and $pr(\Delta^w)$ is the probability that the move is accepted. When $pr(\Delta^w)$ is set to 0.999 (i.e. 0.01% chance that the move would not be accepted) the equation becomes:

$$c(0) = 1000\Delta^w \tag{4.7}$$

The algorithm stops looping once the temperature drops below a threshold

at which point it is unlikely (< 0.05) that even the least bad move would be accepted. The definition of the problem is such that the least bad move is never very bad so requiring that this has less than a 5% chance of being accepted gives the algorithm ample time to ensure the algorithm has not halted during a gradient descent. This again is defined by rearranging the acceptance criteria equation:

$$\begin{aligned} a &= e^{\frac{f(c)-f(n)}{c(k)}} \\ c(\tau) &= \frac{\Delta^l}{\ln pr(\Delta^l)} \end{aligned} \tag{4.8}$$

where $c(\tau)$ is the threshold, $pr(\Delta^l)$ is the probability that the current solution will be accepted and Δ^l is the increase in cost caused by the least bad move possible. When we set $pr(\Delta^l)$ to 0.05 we get the following equation:

$$c(\tau) = 0.33381\Delta^l \tag{4.9}$$

In the general simulated annealing definition above we define the $c(k)$ (equation (4.4)) and L (equation (4.5)) both of which take additional parameters. The parameter β from equation (4.4) is set to be a small number such that the cooling is slow. For our tests we experimentally set $\beta = 2$. The parameter h from equation (4.5) is set such that the number of moves at a given temperature is sufficiently large. For our tests we experimentally set $h = 20$.

Implementation and Analysis

The simulated annealing algorithm can be implemented very efficiently which is essential due to the number iterations the algorithm requires. We outline the running times of various parts of the simulated annealing algorithm in Algorithm 8. The only possible point of contention is that it takes up to $O(l)$ time to update the list of possible moves. Our analysis shows that in the majority of cases this can be achieved in a small amount of time but for small circuits it is possible that this takes $O(l)$ time as one local move can have ramifications for the entire circuit. When this is the case it is highly likely that the value of l is sufficiently small that the algorithm can still compute all possible moves in a very short time.

Algorithm 8 Simulated Annealing Analysis

```
Generate random solution - O(l)
Evaluate initial solution - O(l)
Calculate all possible moves - O(l)
while temperature > threshold do
  while movesMade < L do
    Pick random move - O(1)
    Generate new solution - O(1)
    Evaluate new solution and possibly accept - O(1)
    Update list of possible moves - O(l)
  end while
end while
Return Best Solution
```

4.5 Results

For our experiments we implemented the Simulated Annealing algorithm using Python, although there are many very good implementations of Simulated Annealing algorithms which are freely available. We tested the algorithm with a combination of randomly generated and MCNC benchmark boolean circuits (which are commonly used in related research). We utilized the MVSIS [29] *strash* command to convert the MCNC circuits (in blif format) into 2 bounded AND2 & INVERTER boolean circuits which were then saved as 'bench' files.

In order to compare our results with those from Pandey et al. [122] and SIS [130] we present our findings in terms of cumulative switching which is the total edge switching activity of the whole circuit. Cumulative switching is calculated using equation (4.10) which is quoted from [122].

$$2 \cdot pr(s) \cdot (1 - pr(s)) \tag{4.10}$$

where $pr(s)$ is the probability of signal s being 1.

We initialized the simulated annealing algorithm with $\beta = 2$ and $h = 20$ and mapped the input boolean circuits onto LUTs with $k_{in} = 5$. Each test is run once until completion and the best solution found is reported. The genetic algorithm from [122] took an average of 3.2 seconds to produce the results in Table 4.3. The simulated annealing algorithm used considerably more time to produce our results ($2\frac{1}{4}$ hours for 5xp1) but they provide such a great reduction in power consumption that we consider this to be a reasonable time cost.

In Table 4.3 we report the best results our Simulated Annealing algorithm

Circuit	Cumulative switching			Comparison	
	SIS	GA	SA	$\frac{SA}{SIS}$	$\frac{SA}{GA}$
Misex2	3.59	2.85	2.37	66.01%	83.16%
Sao2	8.36	7.24	5.22	62.44%	72.10%
Con1	1.53	1.19	0.49	32.02%	41.17%
5xp1	2.24	1.71	1.24	55.36%	72.51%
Rd53	4.32	3.05	2.50	57.87%	81.97%
Z4ml	6.98	5.56	4.72	67.62%	84.89%
			Average	57.04%	72.56%

Table 4.3: Results Comparison [130, 122]

found for each circuit alongside the results for SIS and the Genetic algorithm quoted in [122]. We can see that as expected our SA based algorithm has the ability to severely reduce the power consumption of SIS: in the case of circuit Con1 by 67.98% and by an average of 42.96%. Furthermore we see that the SA algorithm has reduced the power consumption of the genetic algorithm by an average of 27.44% and in the case of Con1 by 58.83%.

4.6 Conclusions

In this chapter we have formally defined the problem of low power LUT-based FPGA mapping as a combinatorial optimisation problem. We introduced local moves and showed the resulting neighbourhood function to be complete for traversing all possible solutions in our problem domain. Experimental results for our proposed simulated annealing procedure have been compared to two alternative approaches and we have demonstrated that the SA algorithm finds significantly better results than both. Most notably our results decrease the cumulative switching (which is analogous to power consumption) by up to 27.44% when compared to an alternative genetic algorithm [122]. These results motivate further investigations of simulated annealing (and other local search algorithms) possibly in combination with more tailored / dedicated cooling schedules.

Chapter 5

Advice Complexity for Sleep State Management

When a computer is turned on, it consumes energy. Even if the system is idle, it will continue to consume energy despite not being actively in use. Many modern devices are equipped with various low-power sleep states, which can reduce the amount of energy consumed when the system is idle. One of the most common low-power states is to dim or turn off the screen of the device. This has been shown to significantly reduce power consumption in smart phones [35].

Sleep state management is a fundamental energy efficiency problem. It is considered to be of great importance as it has the potential to significantly reduce the energy consumption of a computer system without reducing the performance of the system when it is in use. Any offline sleep state problem can be solved optimally using a simple algorithm, but the online version of the problem is much harder.

In this chapter, we design novel algorithms with advice to find optimal and near optimal solutions to sleep state problems. We consider algorithms which use advice from an oracle to find optimal and near optimal solutions to online problems. We devise an algorithm with advice which can optimally solve any sleep state problem using $r \log s$ advice bits where r is the length of the interval and s is the number of states. We also present algorithms which can use small amounts of advice to solve sleep state problems with a better competitive ratio than the best possible deterministic algorithms.

We begin by outlining the preliminaries of the problem and discussing pre-

vious research in the areas of sleep state problems and advice complexity. We then present an algorithm which uses advice to find the optimal solution to the problem. Finally, we discuss algorithms which use small amounts of advice bits to find solutions which are better than those found by the best possible deterministic algorithm.

5.1 Online Algorithms with Advice

We recall from Section 2.2. An online algorithm is used to solve a problem where information becomes available over time. In 1985 Sleator et al. [133] introduced the idea of assessing the worst case performance of the online algorithm by comparing it to the optimal offline solution. This is commonly known as competitive analysis. For an online problem we have: $O(I)$ which is the optimal solution for input I; $A(I)$ is the online algorithm's solution for the same input. Each algorithm's solution has a cost $C(O(I))$ and $C(A(I))$. We state that A is c-competitive if there exists some constant $\phi \geq 0$ such that for any I the following holds:

$$C(A(I)) \leq c \cdot C(O(I)) + \phi \tag{5.1}$$

If $\phi = 0$ then we can state that A is strictly c-competitive. Furthermore if $c = 1$ and $\phi = 0$ then A is optimal.

There are many problems where it has been proven that the competitive ratio of a deterministic online algorithm cannot be lower than a certain bound, but where the offline algorithm is optimal. In these cases, it is interesting to study the advice complexity of the problem.

Advice Complexity was first introduced by Böckenhauer et al. in [26]. Roughly speaking, the advice complexity of a problem is the number of bits of information required to allow an online algorithm with advice to find the optimal offline solution. An online algorithm with advice is an online algorithm which can request information from an all-knowing oracle. The advice is delivered by means of an infinite tape and the aim is to use the fewest advice bits to ensure the optimal solution is found. An algorithm with advice is formally defined in [25] as:

Definition 11 *An online algorithm with advice computes the output sequence* $\mathcal{A}^{\phi} = A^{\phi}(I) = (y_1, ..., y_{n-1})$ *such that* y_i *is computed from* $\phi, x_1, ..., x_n$*, where* ϕ *is the content of the advice tape, i.e., an infinite binary sequence and* $I = (x_1, \ldots, x_n)$*.*

The advice complexity of a problem is the upper bound on the number of bits of information an algorithm with advice A' requires to be 1-competitive. Advice complexity is formally defined in [25] as:

Definition 12 *Algorithm A is c-competitive with advice complexity $s(m)$ if there exists some constant a such that, for every m and for each input sequence I of length at most m, there exists a ϕ such that $C(A^{\phi}(I)) \leq c \cdot C(Opt(I)) + \phi'$ and at most $s(m)$ bits of ϕ have been accessed during the computation of $A^{\phi}(I)$. If $\phi' = 0$, then A is strictly c-competitive with advice complexity $s(m)$.*

In some cases, an algorithm with advice may need to know the entire input in advance to find the optimal solution. In other cases, very few bits can be sufficient to find the optimal solution.

It is also interesting to consider the improvement which can be gained by the addition of a small amount of advice. Böckenhauer et al. [25] studied the classic knapsack problem, for which no deterministic online algorithm can have a bounded competitive ratio. They designed an algorithm that uses a single advice bit to be 2-competitive compaired to the optimal offline algorithm.

5.2 Sleep States Problem Definition

We consider a computer system which, whilst in the wake state, consumes energy at a linear rate. There also exist a number of sleep states which consume energy more slowly, but require additional energy to move back into the wake state.

The computer system has a set of states $S = \{s_0, \ldots, s_k\}$ where s_0 (the wake state) is the only state where work can be processed and all other states are sleep states. Each state has an ongoing power function $p(s_i)$ and a wake power function $w(s_i)$.

The function $p(s_i)$ describes how much energy is used per unit time and $w(s_i)$ describes how much energy is required to return the computer system to s_0.

The state power functions are are defined as follows:

$$\begin{aligned} p(s_i) &> p(s_i + 1) \\ p(s_k) &= 0 \end{aligned} \tag{5.2}$$

$$\begin{aligned} w(s_0) &= 0 \\ w(s_i) &< w(s_i + 1) \end{aligned} \tag{5.3}$$

We assume that, for each time step, a unit-size job may arrive on a random basis; therefore the idle periods are also randomly distributed and of random length. We assume that at the end of an idle period the processor will be automatically awoken such that it cannot sleep whilst a job is available. We wish to minimise the energy consumption during the idle periods by selecting the best possible state.

We recall from 2.2.1 that Irani et al. [77, 78] presented the optimal offline algorithm and deterministic algorithm for the 2 and multi-state models. They showed that by using the following equation their algorithms for the 2-state model and multi-state model are both 2-competitive and that this is the best possible competitive ratio for any deterministic algorithm.

$$OPT(t) = \min_{1 \leq i \leq n} \{p(s_i) \cdot t + w(s_i)\} \tag{5.4}$$

5.3 Online Algorithms with Advice for Sleep State Management

In this section, we consider online algorithms with advice which solve the sleep state management problem. We shall begin by outlining the amount of advice which is needed to be optimal and then consider how good an algorithm can be with less advice.

5.3.1 Optimal Advice Complexity

We begin by considering a system with s states where for each idle period it is possible that any one of the s states may be optimal. Therefore, for a single idle period, we need $\log s$ bits of advice to know which is the optimal state [‡]. In the case where we have r idle periods, we require at most $r \log s$ bits of advice to find the optimal solution. This strategy is described by Algorithm 9.

Algorithm 9 Optimal online algorithm with advice

For each idle period use at most $\log n$ advice bits to pick the optimal state.

Therefore, this problem has an advice complexity of $s(r) = r \log n$.

[‡]Within this chapter please assume that log implies $\log_2$.

5.4 A Single Bit of Advice

In this subsection, we wish to establish how competitive an algorithm can be when using just a single bit of advice where there are only two states (active and sleep). We begin by defining the break-even point;

$$b = \left\lceil \frac{w(s_1)}{p(s_0)} \right\rceil \tag{5.5}$$

The break-even point is the length of idle period where the cost of sleeping immediately and idling are equal and therefore both optimal. The well known Irani et al. [77] algorithm Lower Envelope can be expressed in terms of b.

Algorithm 10 Lower Envelope [77]

Sleep after b time steps

In order to find an algorithm which performs better than Lower Envelope, we propose the creation of a second algorithm which can be used to complement Lower Envelope. We define an alternative algorithm Sleep Sooner in terms of b and an input variable δ.

Algorithm 11 Sleep Sooner

Sleep after δb time steps, where $0 \leq \delta \leq 1$

Algorithm 10 sleeps when the idle period is longer than b; this algorithm has a well-known competitive ratio of 2 [77]. The worst-case for this algorithm occurs when the idle period stops immediately after the algorithm has transitioned into the sleep state s_1.

Algorithm 11 is a slightly modified algorithm which enters the sleep state earlier than Algorithm 10 where the performance is linked to the input parameter δ. The worst-case competitive ratio $\frac{1+\delta}{\delta}$ occurs when the idle period ends immediately after transferring to the sleep state, e.g., if $\delta = 0.5$, then the algorithm has a competitive ratio of 3.

We propose the following algorithm which combines Algorithm 10 and Algorithm 11 to improve on the competitive ratio of either of them in isolation.

Algorithm 12 Combined

Use a single advice bit to inform whether Algorithm 10 or Algorithm 11 will be more efficient for the upcoming set of idle periods.

To find the competitive ratio of Algorithm 12, we first need to understand the behavior and nuances of Algorithms 10 and 11.

Algorithm 10 has been analysed may times before and its behaviour is well known. If $\lambda \leq b$ (where λ is the length of the idle period), then the algorithm emulates the optimal offline algorithm and is therefore 1-competitive. When $\lambda = b$, the algorithm has allowed the system to idle for b time steps and has therefore used $b \cdot p(s_0)$ energy. Once $\lambda > b$, it transfers to the sleep state, which costs $w(s_1)$ energy. This means the algorithm has spent $b \cdot p(s_0) + w(s_1)$. We recall the definition of $b = \frac{w(s_1)}{p(s_0)}$. If we substitute this into our equation, we find that Algorithm 10 has consumed $2 \cdot w(s_1)$ energy, which is twice as much as the optimal algorithm; Algorithm 11 is therefore 2-competitive when $\lambda > b$.

$$CR(\text{Algorithm 10}) = \left\{ \begin{array}{ll} 1 & \text{if } \lambda \leq b \\ 2 & otherwise \end{array} \right\} \tag{5.6}$$

where $CR(A)$ is the competitive ratio of an algorithm A.

We know that Algorithm 11 must be optimal when $\lambda \leq b\delta$, as it has remained in the idle state in the same way that the optimal offline solution would. Algorithm 11 enters the sleep state when $\lambda > b\delta$; therefore, when $b\delta < \lambda \leq b$ holds, we know that the algorithm has committed to using $w(s_1) + (b\delta \cdot p(s_0))$ energy at that point. We rearrange the equation by substituting $w(s_1) = b \cdot p(s_0)$: the energy consumed by this algorithm is

$$\begin{gathered} w(s_1) + (b\delta \cdot p(s_0)) \\ b \cdot p(s_0) + (b\delta \cdot p(s_0)) \\ (\delta + 1) \cdot (bp(s_0)) \end{gathered} \tag{5.7}$$

where $\delta b < \lambda$. When the system stays in the wake state the optimal energy consumption is equal to $\lambda \cdot p(s_0)$ (where λ is the length of the idle period) and therefore the competitive ratio is:

$$\begin{aligned} CR(\text{Algorithm 11}) &= \tfrac{(\delta+1) \cdot bp(s_0)}{\lambda \cdot p(s_0)} \\ &= \tfrac{(\delta+1) \cdot b}{\lambda} \\ &= (\delta + 1) \cdot \tfrac{b}{\lambda}, \quad \text{where } \delta b < \lambda \leq b \end{aligned} \tag{5.8}$$

Finally, when $\lambda > b$, the competitive ratio is formed as follows.

$$\begin{aligned} CR(\text{Algorithm 11}) &= \tfrac{(\delta+1)\cdot bp(s_0)}{w(s_1)} \\ &= \tfrac{(\delta+1)\cdot bp(s_0)}{bp(s_0)} \\ &= (\delta+1), \qquad \text{where } \lambda > b \end{aligned} \tag{5.9}$$

Therefore,

$$CR(\text{Algorithm 11}) = \left\{ \begin{array}{ll} 1 & \text{if } \lambda \leq \delta b \\ (1+\delta)\cdot \frac{b}{\lambda} & \text{if } \delta b < \lambda \leq b \\ (1+\delta) & \text{otherwise} \end{array} \right\} \tag{5.10}$$

As we can see from Equations (5.6) and (5.10), both algorithms perform optimally when $\lambda < \delta b$. Therefore, we do not consider this situation in our analysis. Algorithm 10 performs optimally up until $\lambda > b$, where its competitive ratio goes up to 2. Algorithm 11 performs worse for the critical interval $[\delta b, b]$, but better when $\lambda > b$. We therefore wish to use Algorithm 11 in situations where there are many long idle periods and Algorithm 10 where many idle periods have a length in the critical interval.

A certain fraction $\gamma = [0, 1]$ of the idle periods which occur will have a length in the critical interval. The competitive ratio of Lower Envelope and Sleep Sooner can be calculated in terms of γ. We begin by setting $w(s_1) = 1$ and calculating the energy consumption of each algorithm for the 2 cases where $\lambda = \delta b$ (the start of the critical interval where Sleep Sooner has the worst competitive ratio) and $\lambda \geq b$ (the end of the critical interval where Lower Envelope has the worst competitive ratio). See Figure 5.1 for the energy consumption for each combination of case and algorithm.

Case	**Optimal**	**Lower Envelope**	**Sleep Sooner**
$\lambda = \delta b$	δ	δ	$1+\delta$
$\lambda \geq b$	1	2	$1+\delta$

Figure 5.1: Energy consumption of Optimal Algorithm, Lower Envelope and Sleep Sooner

Therefore the competitive ratio of Lower Envelope and Sleep Sooner are calculated using Equations (5.11) and (5.12) respectively.

$$\text{CR(Algorithm 10)} \quad = \frac{\gamma\delta+2(1-\gamma)}{\gamma\delta+(1-\gamma)} \tag{5.11}$$

$$\text{CR(Algorithm 11)} = \frac{1+\delta}{\gamma\delta+(1-\gamma)} \tag{5.12}$$

We can therefore use the following equation to express the competitive ratio of Algorithm 12.

$$\text{CR(Algorithm 12)} = \min\left(\frac{\gamma\delta+2(1-\gamma)}{\gamma\delta+(1-\gamma)}, \frac{1+\delta}{\gamma\delta+(1-\gamma)}\right) \tag{5.13}$$

The fraction γ can be used to decide which algorithm is best to use in any given situation. To know where the trade-off point is (i.e., the point where it is more cost-effective to use Algorithm 11 than Algorithm 10) we calculate the optimal value of γ. We can use Equation (5.14) to calculate the optimal value of γ.

$$\begin{aligned}
\tfrac{1+\delta}{\gamma\delta+(1-\gamma)} &= \tfrac{\gamma\delta+2(1-\gamma)}{\gamma\delta+(1-\gamma)} \\
1+\delta &= \gamma\delta+2(1-\gamma) \\
1+\delta &= \gamma\delta+2-2\gamma \\
\delta-1 &= \gamma\delta-2\gamma \\
\delta-1 &= (\delta-2)\gamma \\
\gamma &= \tfrac{\delta-1}{\delta-2}
\end{aligned} \tag{5.14}$$

At this point, we need to find the optimal value for δ such that we can implement the algorithm and find the competitive ratio. We now substitute our value for γ back into our original equation and simplify.

$$
\begin{aligned}
\text{CR(Algorithm 10)} &= \tfrac{\gamma\delta+2(1-\gamma)}{\gamma\delta+(1-\gamma)} \\
&= \tfrac{(\delta-2)\gamma+2}{(\delta-1)\gamma+1} \\
&= \tfrac{(\delta-2)\cdot\left(\frac{\delta-1}{\delta-2}\right)+2}{(\delta-1)\cdot\left(\frac{\delta-1}{\delta-2}\right)+1} \\
&= \tfrac{\delta+1}{(\delta-1)\cdot\left(\frac{\delta-1}{\delta-2}\right)+1} \qquad (5.15) \\
&= \tfrac{\delta+1}{\left(\frac{(\delta-1)^2+(\delta-2)}{\delta-2}\right)} \\
&= \tfrac{(\delta+1)\cdot(\delta-2)}{(\delta-1)^2+(\delta-2)} \\
&= \tfrac{\delta^2-\delta-2}{\delta^2-\delta-1}
\end{aligned}
$$

We differentiate Equation (5.15) (using the quotient rule) to find the turning point and hence, the optimal value of δ.

$$
\begin{aligned}
f'(\delta) &= \tfrac{(\delta^2-\delta-1)\cdot(2\delta-1)-(2\delta-1)\cdot(\delta^2-\delta-2)}{[\delta^2-\delta-1]^2} \\
f'(\delta) &= \tfrac{(2\delta^3-2\delta^2-2\delta-\delta^2+\delta+1)-(2\delta^3-2\delta^2-4\delta-\delta^2+\delta+2)}{[\delta^2-\delta-1]^2} \\
f'(\delta) &= \tfrac{(2\delta^3-3\delta^2-\delta+1)-(2\delta^3-3\delta^2-3\delta+2)}{[\delta^2-\delta-1]^2} \qquad (5.16) \\
f'(\delta) &= \tfrac{2\delta-1}{[\delta^2-\delta-1]^2}
\end{aligned}
$$

By setting $f'(\delta) = 0$ we find the turning point.

$$
\begin{aligned}
f'(\delta) &= \tfrac{2\delta-1}{[\delta^2-\delta-1]^2} \\
0 &= \tfrac{2\delta-1}{[\delta^2-\delta-1]^2} \\
0 &= 2\delta - 1 \qquad (5.17) \\
2\delta &= 1 \\
\delta &= 0.5
\end{aligned}
$$

Our calculations show that the optimal value of δ is 0.5. We then substitute

this back into Equation (5.14) to find γ.

$$\begin{aligned} \gamma &= \tfrac{\delta-1}{\delta-2} \\ \gamma &= \tfrac{0.5-1}{0.5-2} \\ \gamma &= \tfrac{-0.5}{-1.5} \\ \gamma &= \tfrac{1}{3} \end{aligned} \tag{5.18}$$

therefore $\gamma = \frac{1}{3}$ which can now be used to inform the tuning of Algorithm 12 as follows.

Algorithm 13 Combined: $\delta = 0.5$, $\gamma = \frac{1}{3}$

If $\frac{1}{3}$ or more of the idle periods fall within the critical interval, use Algorithm 10 else use Algorithm 11

The competitive ratio of Algorithm 13 (and therefore Algorithm 12) is described in the equation below.

$$\begin{aligned} \text{CR(Algorithm 13)} &= \tfrac{1+\delta}{\gamma\delta+(1-\gamma)} \\ &= \tfrac{1+0.5}{\frac{1}{3}\cdot 0.5+(1-\frac{1}{3})} \\ &= \tfrac{1.5}{\frac{5}{6}} \\ &= \tfrac{9}{5} \\ &= 1.8 \end{aligned} \tag{5.19}$$

Algorithm 12 is 1.8-competitive with just 1 bit of advice, an improvement of 20% over the algorithm with no advice.

5.5 Slightly More Advice

Now we have addressed the case where we have a single bit of advice. We now consider the case where we allow our algorithm to use a bits of advice where $1 < a < m \log n$. This allows us to design an algorithm which can select the best from 2^a deterministic algorithms. We have shown that the special case where $a = 1$ has a competitive ratio of 1.8, but we wish to find out if we can receive further benefit from more advice.

Algorithm 14 Sleep Sooner i

If $\lambda \leq b\delta_i$ then idle otherwise sleep
where $\delta_i = \frac{i+1}{2^a}$

We begin by defining 2^a algorithms:

We know from our previous calculations (Equations (5.7), (5.8), (5.9) and (5.10)) that the competitive ratio of Algorithm 14 for an idle period of length λ is:

$$\text{CR(Algorithm 14}(\lambda)) = \left\{ \begin{array}{ll} 1 & \text{if } \lambda \leq b\delta_i \\ (1+\delta_i) \cdot \frac{b}{\lambda} & \text{if } b\delta_i < \lambda \leq b \\ 1+\delta_i & \text{otherwise} \end{array} \right\} \tag{5.20}$$

The competitive ratio of an algorithm for a sequence of idle periods is:

$$\text{CR(Algorithm 14)} = \frac{1+\delta_i}{\gamma'\delta_i + (1-\gamma')} \tag{5.21}$$

where γ' is the proportion of idle periods which fall in the interval $[\delta_i b, b]$.

We devise an algorithm (similar to Algorithm 12) which efficiently combines 2^a instances of Algorithm 14.

Algorithm 15 Sleep Sooner Multi

Use a advice bits to indicate which of the 2^a instances of Algorithm 14 is the most efficient.

As we have multiple algorithms, we also introduce the vector of fractions $\Gamma = (\gamma_0, ...\gamma_a)$ where $\sum_{i=0}^{a} \gamma_i = 1$, γ_i is the fraction of idle periods with a length λ such that $\delta_i \leq \lambda < \delta_{i+1}$ and γ_a is the fraction of idle periods with length $\lambda > b$.

When we introduce Γ, the competitive ratio of any instance of Algorithm 14 can be calculated by the following equation:

$$\text{CR(Algorithm 14}(\Gamma)) = \frac{\sum_{j=0}^{i-1}(\gamma_j\delta_j) + \sum_{j=i}^{a}\gamma_j \cdot (1+\delta_i)}{\sum_{j=0}^{a}(\gamma_j\delta_j)} \tag{5.22}$$

Therefore, Algorithm 15 has the following competitive ratio.

$$\text{CR(Algorithm 15)} = \max_{\Gamma}(\min(A_0(\Gamma), ...A_n(\Gamma))) \tag{5.23}$$

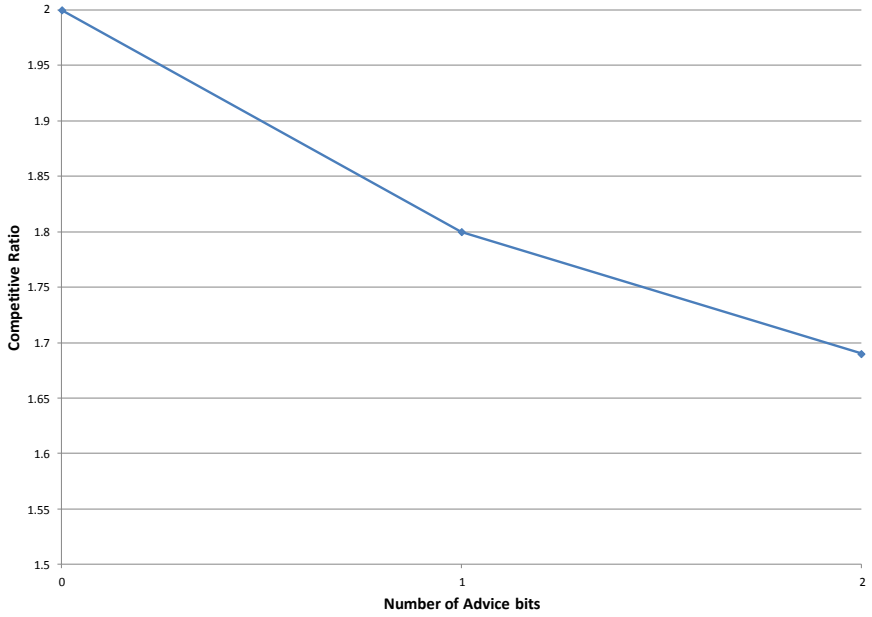

Figure 5.2: Reduction in competitive ratio for increasing amounts of advice.

We know the competitive ratio for this algorithm when $a = 0$ and have proven the competitive ratio for the case where $a = 1$. In addition, we have used a simple deterministic algorithm to estimate (with a high degree of accuracy) the competitive ratio when $a = 3$. These points are plotted in Figure 5.2. Note that the downwards curve has a slowing gradient which suggests there is a limit on the improvement that can be achieved.

As the value of a increases so does the number of algorithms and the interval $[1, b]$ becomes divided into increasingly smaller sections. Once $2^a \geq b$, we have divided the interval into the smallest pieces possible and any additional advice would be wasted. This imposes a natural upper bound on the number of advice bits which are useful to Algorithm 15.

Theorem 2 *The upper bound on the advice bits Algorithm 15 can make use of is* $\lceil \log b \rceil$.

When Algorithm 15 is used, it groups idle periods into bins according to their length. The more advice bits we use, the narrower the range of each bin is. If we consider a discrete time model and we have $\lceil \log b \rceil$ advice bits, then we can express the optimal value for δ which minimises the following equation.

$$\mathrm{CR}(A) = \frac{\sum_{j=0}^{i-1}(\gamma_j \delta_j) + \sum_{j=i}^{a} \gamma_j \cdot (1+\delta_i)}{\sum_{j=0}^{a}(\gamma_j \delta_j)} \tag{5.24}$$

Therefore, the number of advice bits which are of use when using this algorithm which is upper bounded by $\lceil \log b \rceil$.

5.6 More Sleep States

So far we have considered the case where there is just one sleep state. In this section, we consider a system with multiple sleep states.

In Equations (5.2) and (5.3) we defined a series of states in terms of their relationship to one another; as the sleep state becomes deeper, the gradient of each line becomes less steep but the starting point rises. We begin by defining a series of break-even points $(b_0, ...b_n)$ by.

$$b_i = \left\{ \begin{array}{cc} 0 & \text{if } i = 0 \\ \frac{w(s_i) - w(s_{i+1})}{p(s_{i+1}) - p(s_i)} & \text{otherwise} \end{array} \right\} \tag{5.25}$$

We outline the multi-sleep-state version of Lower Envelope in terms of b_i.

Algorithm 16 Lower Enveloper (multi-state)

Transition to state s_i when $\lambda > b_i$

We now describe a new algorithm based on Algorithm 11 which allows for multiple sleep states.

Algorithm 17 Sleep Sooner δ Multi-state

Transition to state s_i when $\lambda > \delta \cdot b_i$

Algorithm 16 has an established general competitive ratio of 2; here we precisely outline the algorithm's competitive ratio for idle periods of variable lengths. We begin by stating that Algorithm 16 is optimal for any $\lambda \leq b_1$ as it will remain in s_0 as will the optimal offline algorithm. The following equation describes the energy which is consumed if $b_i < \lambda \leq b_{i+1}$.

$$\lambda \cdot p(s_i) + 2w(s_i) \tag{5.26}$$

The optimal energy consumption is described by.

$$\lambda \cdot p(s_i) + w(s_i) \tag{5.27}$$

As with Algorithms 10 and 11, the worst cases occur immediately after the algorithm has transitioned to a deeper sleep state, therefore the worst cases for $b_i < \lambda \leq b_n$ are:

$$\frac{b_i \cdot p(s_i) + 2w(s_i)}{b_i \cdot p(s_i) + w(s_i)} \in (1, 2] \tag{5.28}$$

Finally, when $\lambda > b_n$, we find the special case where.

$$b_n \cdot p(s_{n-1}) + w(s_{n-1}) = w(s_n) \tag{5.29}$$

Hence.

$$\begin{gathered} \frac{b_i \cdot p(s_{n-1}) + w(s_{n-1}) + w(s_n)}{w(s_{n-1})} \\ \frac{2w(s_n)}{w(s_{n-1})} = 2 \end{gathered} \tag{5.30}$$

$$cr(\text{Algorithm 16}) = \left\{ \begin{array}{cc} 1 & \text{if } \lambda \leq b_1 \\ \frac{(b_i \cdot p(s_i)) + 2w(s_i)}{(b_i \cdot p(s_i)) + w(s_i)} & \text{if } b_1 < \lambda \leq b_k \\ 2 & otherwise \end{array} \right\} \tag{5.31}$$

We now move on to analyse Algorithm 17; for any $\lambda \leq \delta_1 \cdot b_1$, Algorithm 17 will remain in s_0 as will the optimal offline algorithm and hence it is 1-competitive. When $b_i < \lambda \leq b_{i+1}$, the following amount of energy is consumed.

$$\lambda \cdot p(s_i) + w(s_i) + w(s_j), \quad \text{where } j \geq i \tag{5.32}$$

The optimal energy consumption is described by Equation (5.27). As with Algorithms 10 and 11, the worst cases occur immediately after the algorithm has transitioned to a deeper sleep state, therefore the worst cases for $b_i < \lambda \leq b_n$ are bounded from above by:

$$\frac{\delta_i b_i \cdot p(s_i) + w(s_i) + w(s_j)}{\delta_i b_i \cdot p(s_i) + w(s_i)} = [1, \frac{1 + \delta_i}{\delta_i}] \tag{5.33}$$

Finally, when $\lambda > b_n$, we know that $b_n \cdot p(s_{n-1}) + w(s_{n-1}) = w(s_n)$ (Equation (5.29)) and Algorithm 17 therefore always transfers to the deeper sleep state at

$\delta_i b_i$.

$$\begin{aligned} & \frac{\delta_i(b_i \cdot p(s_{n-1}) + w(s_{n-1})) + w(s_n)}{w(s_{n-1})} \\ &= \frac{(1+\delta_i) \cdot w(s_n)}{w(s_{n-1})} \\ &= (1 + \delta_i) \end{aligned} \tag{5.34}$$

$$cr(\text{Algorithm 17}) = \left\{ \begin{array}{ll} 1 & \text{if } \lambda \leq \delta_1 b_1 \\ \Omega(\frac{(\delta_i b_i \cdot p(s_i)) + w(s_i) + w(s_j)}{(\delta_i b_i \cdot p(s_i)) + w(s_i)}) & \text{if } b_1 < \lambda \leq b_k \\ (1 + \delta_i) & otherwise \end{array} \right\} \tag{5.35}$$

The exact competitive ratio of both algorithms depends on the characteristics of the sleep states.

We now outline an algorithm which combines Algorithm 16 and 17 to perform better than either in isolation.

Algorithm 18 Combined δ Multi-state

Use a single advice bit to inform whether Algorithm 16 or Algorithm 17 will be more efficient for the upcoming set of idle periods.

Algorithm 18 is particularly difficult to analyse as it is so highly dependent on the characteristics of the states that there are so many possibilities to consider. We can perform a initial analysis by considering the simplified Algorithm 19.

Algorithm 19 Simplified Combined δ Multi-state

use Algorithm 16	if $\lambda < b_k$
use Algorithm 17	otherwise

Algorithm 19 allows us to eliminate the worst cases for each algorithm individually and hence reduce the overall competitive ratio. By combining the competitive ratios of Algorithms 16 and 17 we can find the following competitive ratio for Algorithm 19.

$$cr(\text{Algorithm 19}) = \left\{ \begin{array}{ll} 1 & \text{if } \lambda \leq \delta_1 b_1 \\ \frac{(b_i \cdot p(s_i)) + 2w(s_i)}{(b_i \cdot p(s_i)) + w(s_i)} & \text{if } b_1 < \lambda \leq b_k \\ (1 + \delta_i) & otherwise \end{array} \right\} \tag{5.36}$$

We know $\frac{(b_i \cdot p(s_i))+2w(s_i)}{(b_i \cdot p(s_i))+w(s_i)} < 2$ therefore;

$$\text{CR(Algorithm 18)} < 2. \tag{5.37}$$

5.7 Conclusions

In summary, we have considered the sleep state management problem in terms of advice complexity. We have shown the level of advice necessary to produce an optimal solution is $\lceil r \log s \rceil$. We show that a single bit of advice is sufficient to improve the competitive ratio for the single sleep state problem by 20% and that adding more advice can improve the performance, but only until we have $\lceil \log b \rceil$ advice bits. Finally, we show that, when there is more than one sleep state, we can improve the performance of the algorithm with a single bit of advice.

Chapter 6

Conclusions

In this thesis we considered optimal power management within smartphones. We began by reviewing a large amount of relevant literature to understand the nuances of the problem domain and we then identified problems which have not been previously considered or where existing solutions leave room for improvement. Subsequently we developed novel solutions which exploit insight to find superior results.

Our first contribution chapter focused on low energy scheduling for heterogeneous multiprocessor systems with DSS. In contrast to existing solutions we consciously chose to control the speed of processors and schedule jobs whilst considering the energy consumption and performance of the entire system. Existing solutions first divide the jobs between processors and then control the speed of each processor individually. This disjointed approach often results in the system as a whole operating sub-optimally with regards to energy consumption. The Virtual Single Processor (VSP) approach ensures we are efficiently utilising the system as a whole which can save time and energy.

We found that the VSP approach consumed between 4.4% (in a 2 processor system) and 8.2% (in a 16 processor system) less energy than the best alternative algorithm [72] with no reduction in speed. When combined with an existing single processor DSS scheduling algorithm simulations showed that the VSP approach reduced the objective function of $\sum$Weighted Flow + Energy by an average of 2.31% compared to [72]. This shows that the VSP approach can save energy and bound tighter to the objective function than [72] which suggests it is a viable alternative with real potential to reduce energy consumption.

There are a number of interesting open problems related to this area. One of

the most interesting is the combination of sleep states and DSS for heterogeneous multiprocessor systems. The current VSP model does not consider the costs associated with sleeping and waking. If these costs were introduced then we may find that the system operates quite differently. Processors which are awake may be utilised more readily and those which are sleeping would only be roused if there is sufficient demand to warrant the energy cost. This problem needs to be formalised and solved if we are to develop the most efficient multiprocessor computer systems.

There are also many interesting open problems surrounding the energy cost of computing solutions for energy efficiency problems. This is somewhat of a paradox and is similar to the amount of time used to compute a schedule to minimise makespan [112]. In the case of energy efficiency it may make more sense to design scheduling algorithms which find reasonable solutions with a low energy overhead rather than near optimal solutions with a large time and energy cost. This is particularly of interest with mobile devices or any device with a restricted power resource. There is also a great amount of insight to be gained by analysing existing algorithms in terms of their energy cost.

The second problem we considered was mapping an input boolean function onto an LUT based FPGA, such that the energy consumed by the dynamic switching is minimised: this is an NP hard problem. We formulated a combinatorial optimisation problem, developed a complete neighbourhood function and applied a simulated annealing algorithm. We found that our approach solved the benchmark problems to a higher standard than an alternative genetic algorithm [122]. Our solution reduced the average switching activity (which is analogous to energy consumption) by an average of 27.44%. These findings motivate further research into this area with regards to developing a more tailored simulated annealing algorithm and considering alternative local search algorithms which could produce better results or converge onto the solution more quickly.

FPGAs are used for much more than implementing logic circuits; many system on a chip devices integrate FPGAs or are based on FPGAs. It would be of great interest to consider these from an energy efficient perspective, as many system on a chip devices are deployed as mobile or embedded systems which have restricted access to power.

Finally, we studied the advice complexity of the sleep state management problem. The advice complexity of a problem is the amount of information required to enable an online algorithm with advice to produce solutions which are 1-competitive with regard to the optimal offline solution. We found that

the sleep state management problem has an advice complexity of $r \log_2 s$ where r is the number of idle periods and s is the number of states. We went on to design an algorithm that needs just one bit of advice to be 1.8-competitive for the 2 state problem; this is a reduction of 20% compared to the best possible deterministic algorithm. More advice can help to reduce the competitive ratio further, but only until we have $\lceil \log b \rceil$ advice bits, at which point it converges onto a sub-optimal solution. We also considered the case where there is more than one sleep state and found that a single bit of advice could improve the best known algorithm, but only marginally. This is the first time an energy efficiency problem has been considered in terms of advice complexity.

It would be of great interest to harness the insight gained from our advice complexity work to develop online algorithms. We could develop an algorithm which uses the distribution and length of past idle periods to out perform a deterministic algorithm. It would be especially interesting to see if this algorithm could be designed to efficiently gather data and control the sleep patterns within a real smart phone and find out if it could produce a significant reduction in energy consumption.

Future Work

This thesis has touched on a number of distinct areas; power heterogeneous multiprocessor scheduling with DSS, simulated annealing to solve the LUT based FPGA mapping and advice complexity of the sleep state management problem. In each of these areas we strove to make advances towards the ultimate goal of improving the energy efficiency of mobile devices. There are many related problems which we would have loved to work on as we could apply our domain knowledge and techniques to find good solutions; we outline a few of the related problems below.

Development of energy efficient algorithms. Many algorithms are designed with speed in mind but there could be great potential in analysing and designing key algorithms according to energy efficiency. An energy efficient sorting algorithm could help to save huge amounts of energy, especially as we are seeing vast increases in the amount of data being stored and manipulated. There have been some tentative steps into this direction but this is something which I believe has a very strong potential for energy saving.

Mobile devices use a significant amount of energy whilst transferring data, especially when the signal is weak [35]. This is because the device has to use

more energy to generate a more powerful transmission or boost the incoming signal; also more transmissions get lost or distorted due to interference which means duplicate messages have to be sent. If we could design an algorithm to only perform essential or time critical tasks when the signal is low then we could potentially save vast amounts of energy, not only for the device but also for the cell towers.

Summary

This thesis is the culmination of almost four years of work; in that time we have extensively studied how energy is consumed in computational devices, identified areas where improvements can be made and studied these problems further. We have performed detailed research in three areas, produced novel solutions and used experiments, simulations and theoretical work to show that our solutions are better alternative algorithms. I hope that our solutions and alternative approaches can help to forward the research in this growing area.

Chapter 7

Bibliography

[1] Emile Aarts and Jan Korst. *Simulated annealing and Boltzmann machines - a stochastic approach to combinatorial optimization and neural computing*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1990.

[2] Alok Aggarwal, Ashok Chandra, and Prabhakar Raghavan. Energy consumption in VLSI circuits. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 205–216, 1988.

[3] Susanne Albers. Algorithms for energy saving. In *Efficient Algorithms: Lecture Notes in Computer Science*, pages 173–186. Springer Berlin / Heidelberg, 2009.

[4] Susanne Albers. Energy-efficient algorithms. *Communications of ACM*, 53:86–96, 2010.

[5] Susanne Albers. Algorithms for dynamic speed scaling. In *28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011)*, volume 9, pages 1–11. Schloss Dagstuhl, 2011.

[6] Susanne Albers, Antonios Antoniadis, and Gero Greiner. On multi-processor speed scaling with migration. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, pages 279–288. ACM, 2011.

[7] Susanne Albers and Hiroshi Fujiwara. Energy-efficient algorithms for flow time minimization. In *23rd International Symposium on Theoretical Aspects of Computer Science (STACS 2006)*, pages 621–633. Springer, 2006.

[8] Susanne Albers and Hiroshi Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Transactions on Algorithms*, 3:49–66, 2007.

[9] Susanne Albers, Fabian Müller, and Swen Schmelzer. Speed scaling on parallel processors. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 289–298. ACM, 2007.

[10] Andreas Albrecht. A problem-specific convergence bound for simulated annealing-based local search. In *Computational Science and Its Applications (ICCSA) 3*, pages 405–414, 2004.

[11] Jason Anderson and Farid Najm. Power-aware technology mapping for LUT-based FPGAs. In *In Proceedings of 2002 IEEE International Conference on Field-Programmable Technology*, pages 211–218, 2002.

[12] Jason Anderson and Farid Najm. Switching activity analysis and prelayout activity prediction for FPGAs. In *Proceedings of the 2003 international workshop on System-level interconnect prediction*, pages 15–21, 2003.

[13] Jason Anderson and Farid Najm. Power estimation techniques for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(10):1015–1027, 2004.

[14] Lachlan Andrew, Adam Wierman, and Ao Tang. Optimal speed scaling under arbitrary power functions. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 39–41, 2009.

[15] Lachlan LH Andrew, Minghong Lin, and Adam Wierman. Optimality, fairness, and robustness in speed scaling designs. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 37–48, 2010.

[16] John Augustine, Sandy Irani, and Chaitanya Swamy. Optimal power-down strategies. *SIAM Journal on Computing*, 37(5):1499–1516, 2008.

[17] Nikhil Bansal, David Bunde, Ho-Leung Chan, and Kirk Pruhs. Average rate speed scaling. In *Proceedings of the 8th Latin American conference*

on Theoretical informatics (LATIN'08), pages 240–251. Springer-Verlag, 2008.

[18] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with an arbitrary power function. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09)*, pages 693–701. Society for Industrial and Applied Mathematics, 2009.

[19] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with an arbitrary power function. *ACM Transactions Algorithms*, 9(2), 2013.

[20] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Dynamic speed scaling to manage energy and temperature. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, pages 520–529. IEEE, 2004.

[21] Nikhil Bansal, Kirk Pruhs, and Cliff Stein. Speed scaling for weighted flow time. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '07)*, pages 805–813. Society for Industrial and Applied Mathematics, 2007.

[22] John Beasley and Paul Chu. A genetic algorithm for the set covering problem. *European Journal of Operational Research*, 94(2):392–404, 1996.

[23] Luca Becchetti, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Online weighted flow time and deadline scheduling. *Journal of Discrete Algorithms*, 4(3):339–352, 2006.

[24] Shilpa Bhoj and Dinesh Bhatia. Pre-route interconnect capacitance and power estimation in FPGAs. In *International Conference on Field Programmable Logic and Applications, 2007*, pages 159–164. IEEE, 2007.

[25] Hans-Joachim Böckenhauer, Dennis Komm, Richard Královič, and Peter Rossmanith. *On the Advice Complexity of the Knapsack Problem.* Springer, 2012.

[26] Hans-Joachim Böckenhauer, Dennis Komm, Rastislav Krlovi, Richard Krlovi, and Tobias Mmke. On the advice complexity of online problems. In *Algorithms and Computation*, volume 5878 of *Lecture Notes in Computer Science*, pages 331–340. Springer Berlin Heidelberg, 2009.

[27] George Boole. *An Investigation of the Laws of Thought: On which are Founded the Mathematical Theories of Logic and Probabilities.* Walton and Maberly, 1854.

[28] Fred Bower, Daniel Sorin, and Landon Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE Micro*, 28:17–25, May 2008.

[29] Robert Brayton and Sunil Khatri. Multi-valued logic synthesis. In *VLSI Design, 1999. Proceedings. Twelfth International Conference On*, pages 196–205. IEEE, 1999.

[30] David Brooks, Pradip Bose, Stanley Schuster, Hans Jacobson, Prabhaka Kudva, Alper Buyuktosunoglu, J Wellman, Victor Zyuban, Manish Gupta, and Peter Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20:26–44, 2000.

[31] James Bruno, Edward Coffman Jr, and Ravi Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications ACM*, 17(7):382–387, 1974.

[32] Ion Bucur, Nicolae Cupcea, Adrian Surpateanu, Costin Stefanescu, and Florin Radulescu. Power-aware, depth-optimum and area minimization mapping of K-LUT based FPGA circuits. *WSEAS Transactions on Computers*, 8(11):1812–1824, 2009.

[33] David Bunde. Power-aware scheduling for makespan and flow. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '06)*, pages 190–196. ACM, 2006.

[34] David Bunde. Power-aware scheduling for makespan and flow. *Journal of Scheduling*, 12:489–500, 2009.

[35] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 21–21, 2010.

[36] Vladimír Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[37] Ho-Leung Chan, Joseph Wun-Tat Chan, Tak-Wah Lam, Lap-Kei Lee, Kin-Sum Mak, and Prudence Wong. Optimizing throughput and energy in online deadline scheduling. *ACM Transaction Algorithms*, 6(1):10:1–10:22, 2009.

[38] Ho-Leung Chan, Wun-Tat Chan, Tak-Wah Lam, Lap-Kei Lee, Kin-Sum Mak, and Prudence Wong. Energy efficient online deadline scheduling. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '07)*, pages 795–804. Society for Industrial and Applied Mathematics, 2007.

[39] Ho Leung Chan, Jeff Edmonds, and Kirk Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 1–10. ACM, 2009.

[40] Ho-Leung Chan, Tak-Wah Lam, and Rongbin Li. Tradeoff between energy and throughput for online deadline scheduling. In *Proceedings of the 8th international conference on Approximation and online algorithms (WAOA'10)*, pages 59–70. Springer-Verlag, 2011.

[41] Sze-Hang Chan, Tak-Wah Lam, Lap-Kei Lee, Chi-Man Liu, and Hing-Fung Ting. Sleep management on multiple machines for energy and flow time. In *Proceedings of the 38th international colloquim conference on Automata, languages and programming - Volume Part I (ICALP'11)*, pages 219–231. Springer-Verlag, 2011.

[42] Sze-Hang Chan, Tak-Wah Lam, Lap-Kei Lee, and Jianqiao Zhu. Nonclairvoyant sleep management and flow-time scheduling on multiple processors. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, pages 261–270. ACM, 2013.

[43] Chandra Chekuri, Sanjeev Khanna, and An Zhu. Algorithms for minimizing weighted flow time. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 84–93. ACM, 2001.

[44] Chau-Shen Chen, TingTing Hwang, and Chang Liu. Low power FPGA design - a re-engineering approach. In *Proceedings of the 34th annual Design Automation Conference (DAC '97)*, pages 656–661, New York, NY, USA, 1997. ACM.

[45] Deming Chen and Jason Cong. Daomap: a depth-optimal area optimization mapping algorithm for FPGA designs. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 752–759. IEEE Computer Society, 2004.

[46] Juanjuan Chen, Xing Wei, Qiang Zhou, and Yici Cai. Power optimization through edge reduction in lut-based fpga technology mapping. In *Proceedings of the International Communications of Circuits and Systems, 2009. (ICCCAS 2009)*, pages 1087–1091. IEEE, 2009.

[47] Lei Cheng, Deming Chen, and Martin Wong. Glitchmap: an FPGA technology mapper for low power considering glitches. In *Proceedings of the 44th annual Design Automation Conference*, pages 318–323. ACM, 2007.

[48] Daniel Cole, Dimitrios Letsios, Michael Nugent, and Kirk Pruhs. Optimal energy trade-off schedules. In *2012 International Green Computing Conference (IGCC)*, pages 1–10. IEEE, 2012.

[49] Jason Cong and Yuzheng Ding. Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(1):1–12, 1994.

[50] Jason Cong, Chang Wu, and Yuzheng Ding. Cut ranking and pruning: enabling a general and efficient FPGA mapping solution. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 29–35. ACM, 1999.

[51] Tomasz Czajkowski and Stephen Brown. Using negative edge triggered ffs to reduce glitching power in FPGA circuits. In *Proceedings of the 44th annual Design Automation Conference*, pages 324–329. ACM, 2007.

[52] Erik Demaine, Mohammad Ghodsi, Mohammad Taghi Hajiaghayi, Amin Sayedi-Roshkhar, and Morteza Zadimoghaddam. Scheduling to minimize gaps and power consumption. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 46–54. ACM, 2007.

[53] Srinivas Devadas and Sharad Malik. A survey of optimization techniques targeting low power VLSI circuits. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 242–247. ACM, 1995.

[54] Richard Dobson and Kathleen Steinhöfel. Low energy scheduling with power heterogeneous multiprocessor systems. In *VII ALIO/EURO*, VII ALIO/EURO, 2011.

[55] Richard Dobson and Kathleen Steinhöfel. Low energy scheduling with power heterogeneous multiprocessor systems. In *In proceedings of the 5th Multidisciplinary International Conference on Scheduling : Theory and Applications (MISTA 2011)*, pages 297–307, 2011.

[56] Richard Dobson and Kathleen Steinhöfel. Sa based power efficient FPGA LUT mapping. In *Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*, GECCO '13 Companion, pages 1545–1552. ACM, 2013.

[57] Carla Schlatter Ellis. The case for higher-level power management. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, pages 162–167. IEEE, 1999.

[58] Amir Farrahi and Majid Sarrafzadeh. Complexity of the lookup-table minimization problem for FPGA technology mapping. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(11):1319–1332, 1994.

[59] Amir Farrahi and Majid Sarrafzadeh. FPGA technology mapping for power minimization. In *Field-Programmable Logic Architectures, Synthesis and Applications*, volume 849 of *Lecture Notes in Computer Science*, pages 66–77. Springer Berlin / Heidelberg, 1994.

[60] Robert J Francis, Jonathan Rose, and Kevin Chung. Chortle: a technology mapping program for lookup table-based field programmable gate arrays. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 613–619. ACM, 1991.

[61] Ross Freeman. Configurable electrical circuit having configurable logic elements and configurable interconnects, 1989.

[62] Hiroshi Fujiwara and Kazuo Iwama. Average-case competitive analyses for ski-rental problems. In *Algorithms and Computation*, pages 476–488. Springer, 2002.

[63] Michael Garey, David Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[64] Michel Gendreau, Alain Hertz, and Gilbert Laporte. A tabu search heuristic for the vehicle routing problem. *Management science*, 40(10):1276–1290, 1994.

[65] Anandaroop Ghosh, Somnath Paul, and Swarup Bhunia. Energy-efficient application mapping in FPGA through computation in embedded memory blocks. In *25th International Conference on VLSI Design 2012*, pages 424–429. IEEE, 2012.

[66] Fred Glover. Tabu search - part I. *ORSA Journal on computing*, 1(3):190–206, 1989.

[67] Fred Glover. Tabu search - part II. *ORSA Journal on computing*, 2(1):4–32, 1990.

[68] Fred Glover and Manuel Laguna. *Tabu search*, volume 22. Springer, 1997.

[69] Stephen Goldfeld, Richard Quandt, and Hale Trotter. Maximization by quadratic hill-climbing. *Econometrica: Journal of the Econometric Society*, pages 541–551, 1966.

[70] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling heterogeneous processors isn't as easy as you think. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '12)*, pages 1242–1253. SIAM, 2012.

[71] Anupam Gupta, Ravishankar Krishnaswamy, and Kirk Pruhs. Nonclairvoyantly scheduling power-heterogeneous processors. In *Proceedings of the International Conference on Green Computing (GREENCOMP '10)*, pages 165–173. IEEE Computer Society, 2010.

[72] Anupam Gupta, Ravishankar Krishnaswamy, and Kirk Pruhs. Scalably scheduling power-heterogeneous processors. In *Proceedings of the 37th international colloquium conference on Automata, languages and programming (ICALP'10)*, pages 312–323. Springer-Verlag, 2010.

[73] Bruce Hajek. A tutorial survey of theory and applications of simulated annealing. In *Decision and Control, 1985 24th IEEE Conference on*, volume 24, pages 755–760. IEEE, 1985.

[74] Xin Han, Tak-Wah Lam, Lap-Kei Lee, Isaac To, and Prudence Wong. Deadline scheduling and power management for speed bounded processors. *Theoretical Computer Science*, 411(40):3587–3600, 2010.

[75] Jan-Min Hwang, Feng-Yi Chiang, and TingTing Hwang. A re-engineering approach to low power FPGA design using spfd. In *Proceedings of the 35th annual Design Automation Conference (DAC '98)*, pages 722–725, New York, NY, USA, 1998. ACM.

[76] Sandy Irani and Kirk R Pruhs. Algorithmic problems in power management. *ACM SIGACT News*, 36(2):63–76, 2005.

[77] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '03)*, pages 37–46. Society for Industrial and Applied Mathematics, 2003.

[78] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. *ACM Trans. Algorithms*, 3, 2007.

[79] Mark Jerrum and Alistair Sinclair. Polynomial-time approximation algorithms for the ising model. *SIAM Journal on computing*, 22(5):1087–1116, 1993.

[80] Krishna Kant. Toward a science of power management. *Computer*, 42(9):99–101, 2009.

[81] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch, and Susan Owicki. Competitive randomized algorithms for non-uniform problems. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms (SODA '90)*, pages 301–309. Society for Industrial and Applied Mathematics, 1990.

[82] Hans Kellerer, Thomas Tautenhahn, and Gerhard Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. In *In Preceedings of the 38th Annual Symposium on Theory of Computing*, pages 418–426, May 1996.

[83] Hans Kellerer, Thomas Tautenhahn, and Gerhard Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. *SIAM Journal on Computing*, 28(4):1155–1166, 1999.

[84] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948. IEEE, 1995.

[85] Scott Kirkpatrick, Daniel Gelatt, and Mario Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[86] Gloria Kissin. Measuring energy consumption in VLSI circuits: a foundation. In *STOC*, pages 99–104, 1982.

[87] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81–92. IEEE, 2003.

[88] Balakrishna Kumthekar, Luca Benini, Enrico Macii, and Fabio Somenzi. In-place power optimization for LUT-based FPGAs. In *Proceedings of the 35th annual Design Automation Conference (DAC '98)*, pages 718–721. ACM, 1998.

[89] Balakrishna Kumthekar, Luca Benini, Enrico Macii, and Fabio Somenzi. Power optimisation of FPGA-based designs without rewiring. In *Computers and Digital Techniques, IEE Proceedings-*, volume 147, pages 167–174. IET, 2000.

[90] Kanishka Lahiri, Sujit Dey, Debashis Panigrahi, and Anand Raghunathan. Battery-driven system design: A new frontier in low power design. In *Proceedings of the 2002 Asia and South Pacific Design Automation Conference (ASP-DAC '02)*, pages 261–. IEEE Computer Society, 2002.

[91] Tak-Wah Lam, Lap-Kei Lee, Hing-Fung Ting, Isaac K. To, and Prudence Wong. Sleep with guilt and work faster to minimize flow plus energy. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming: Part I (ICALP '09)*, pages 665–676. Springer-Verlag, 2009.

[92] Tak-Wah Lam, Lap-Kei Lee, Isaac To, and Prudence Wong. Energy efficient deadline scheduling in two processor systems. In Takeshi Tokuyama, editor, *Algorithms and Computation*, volume 4835 of *Lecture Notes in Computer Science*, pages 476–487. Springer Berlin Heidelberg, 2007.

[93] Tak-Wah Lam, Lap-Kei Lee, Isaac To, and Prudence Wong. Competitive non-migratory scheduling for flow time and energy. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures (SPAA '08)*, pages 256–264. ACM, 2008.

[94] Tak-Wah Lam, Lap-Kei Lee, Isaac To, and Prudence Wong. Nonmigratory multiprocessor scheduling for response time and energy. *Parallel and Distributed Systems, IEEE Transactions on*, 19(11):1527–1539, 2008.

[95] Tak-Wah Lam, Lap-Kei Lee, Isaac To, and Prudence Wong. Speed scaling functions for flow time scheduling based on active job count. In *Proceedings of the 16th annual European symposium on Algorithms (ESA '08)*, pages 647–659. Springer-Verlag, 2008.

[96] Tak-Wah Lam, Lap-Kei Lee, Isaac To, and Prudence Wong. Improved multi-processor scheduling for flow time and energy. *Journal of Scheduling*, pages 1–12, 2009. 10.1007/s10951-009-0145-5.

[97] Julien Lamoureux and Wayne Luk. An overview of low-power techniques for field-programmable gate arrays. In *Adaptive Hardware and Systems, 2008. AHS'08. NASA/ESA Conference on*, pages 338–345. IEEE, 2008.

[98] Ailsa Land and Alison Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[99] Siobhán Launders, Colin Doyle, and Wesley Cooper. Switching-activity directed clustering algorithm for low net-power implementation of FPGAs. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pages 415–424. Springer, 2005.

[100] Eugene Lawler and Davi Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

[101] Hyung Gyu Lee, Sungyuep Nam, and Naehyuck Chang. Cycle-accurate energy measurement and high-level energy characterization of FPGAs. In

Quality Electronic Design, 2003. Proceedings. Fourth International Symposium on, pages 267–272. IEEE, 2003.

[102] Stefano Leonardi and Danny Raz. Approximating total flow time on parallel machines. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 110–119. ACM, 1997.

[103] Hao Li, Srinivas Katkoori, and Wai-Kei Mak. Power minimization algorithms for LUT-based FPGA technology mapping. *ACM Trans. Des. Autom. Electron. Syst.*, 9(1):33–51, 2004.

[104] Hao Li, Wai-Kei Mak, and Srinivas Katkoori. Lut-based FPGA technology mapping for power minimization with optimal depth. In *VLSI, 2001. Proceedings. IEEE Computer Society Workshop on*, pages 123 –128, 2001.

[105] Hao Li, Wai-Kei Mak, and Srinivas Katkoori. Efficient LUT-based FPGA technology mapping for power minimization. In *Proceedings of the Asia and South Pacific Design Automation Conference, 2003 (ASP-DAC 2003)*, pages 353–358, 2003.

[106] Minming Li, Andrew Yao, and Frances Yao. Discrete and continuous min-energy schedules for variable voltage processors. *Proceedings of the National Academy of Sciences of the United States of America*, 103(11):3983–3987, 2006.

[107] Minming Li and Frances Yao. An efficient algorithm for computing optimal discrete voltage schedules. *SIAM Journal on Computing*, 35(3):658–671, 2005.

[108] Michel Lundy and Alistair Mees. Convergence of an annealing algorithm. *Mathematical programming*, 34(1):111–124, 1986.

[109] Walter H Macwilliams. Gating circuits, 1953. US Patent 2,627,039.

[110] Aqeel Mahesri and Vibhore Vardhan. Power consumption breakdown on a modern laptop. In *Power-aware computer systems*, pages 165–180. Springer, 2005.

[111] Morteza Mashayekhi, Zahra Jeddi, and Esmail Amini. Power optimization of LUT based FPGA circuits. In *Optimization of Electrical and Electronic Equipment, 2008. OPTIM 2008. 11th International Conference on*, pages 37–40, 2008.

[112] Andrew McGregor. A problem in scheduling: Your time starts now... In *Proceedings of Fun with Algorithms*, Fun with Algorithms 2004, pages 34–40, 2004.

[113] Nicholas Metropolis, Arianna Rosenbluth, Marshall Rosenbluth, Augusta Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21:1087, 1953.

[114] Debasis Mitra, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. Convergence and finite-time behavior of simulated annealing. In *Decision and Control, 1985 24th IEEE Conference on*, volume 24, pages 761–767, 1985.

[115] José Monteiro, Srinivas Devadas, Pranav Ashar, and Ashutosh Mauskar. Scheduling techniques to enable power management. In *Design Automation Conference Proceedings 1996, 33rd*, pages 349–352. IEEE, 1996.

[116] Tomer Morad, Uri Weiser, Avinoam Kolodny, Mateo Valero, and Eduard Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *Computer Architecture Letters*, 5(1):14–17, 2006.

[117] Hiroyuki Mori and Yuichiro Goto. A parallel tabu search based method for determining optimal allocation of facts in power systems. In *Power System Technology, 2000. Proceedings. PowerCon 2000. International Conference on*, volume 2, pages 1077–1082. IEEE, 2000.

[118] Trevor Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52 –58, April 2001.

[119] Rajarshi Mukherjee and Seda Ogrenci Memik. Power-driven design partitioning. In *Field Programmable Logic and Application*, pages 740–750. Springer, 2004.

[120] Farid Najm. A survey of power estimation techniques in VLSI circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):446–455, 1994.

[121] Kuo-Rueih Ricky Pan and Massoud Pedram. FPGA synthesis for minimum area, delay and power. In *European Design and Test Conference, 1996. ED TC 96. Proceedings*, page 603, 1996.

[122] Rohit Pandey and Santanu Chattopadhyay. Low power technology mapping for LUT based FPGA - a genetic algorithm approach. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 79–84, 2003.

[123] SS Panwalkar and Wafik Iskander. A survey of scheduling rules. *Operations research*, 25(1):45–61, 1977.

[124] David Patterson and John Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2005.

[125] Mario Pickavet, Willem Vereecken, Sofie Demeyer, Pieter Audenaert, Brecht Vermeulen, Chris Develder, Didier Colle, Bart Dhoedt, and Piet Demeester. Worldwide energy needs for ict: The rise of power-aware networking. In *Advanced Networks and Telecommunication Systems, 2008. ANTS '08. 2nd International Symposium on*, pages 1–3, 2008.

[126] James Pikul, Hui Gang Zhang, Jiung Cho, Paul Braun, and William King. High-power lithium ion microbatteries from interdigitated three-dimensional bicontinuous nanoporous electrodes. *Nature communications*, 4:1732, 2013.

[127] Kirk Pruhs, Patchrawat Uthaisombut, and Gerhard Woeginger. Getting the best response for your erg. In *In Scandanavian Workshop on Algorithms and Theory*, pages 14–25, 2004.

[128] Kirk Pruhs, Patchrawat Uthaisombut, and Gerhard Woeginger. Getting the best response for your erg. *ACM Trans. Algorithms*, 4:38:1–38:17, 2008.

[129] Rob A Rutenbar. Simulated annealing algorithms: An overview. *Circuits and Devices Magazine, IEEE*, 5(1):19–26, 1989.

[130] Ellen Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul Stephan, Robert Brayton, and Alberto Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. 1992.

[131] Claude Shannon. A symbolic analysis of relay and switching circuits. *American Institute of Electrical Engineers, Transactions of the*, 57(12):713–723, 1937.

[132] SN Sivanandam and SN Deepa. *Introduction to genetic algorithms*. Springer Publishing Company, Incorporated, 2007.

[133] Daniel Sleator and Robert Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985.

[134] Kathleen Steinhöfel, Andreas Albrecht, and Chak-Kuen Wong. Two simulated annealing-based heuristics for the job shop scheduling problem. *European Journal of Operational Research*, 118(3):524–548, 1999.

[135] Balram Suman and Prabhat Kumar. A survey of simulated annealing as a tool for single and multiobjective optimization. *Journal of the operational research society*, 57(10):1143–1160, 2005.

[136] Hongyang Sun, Yuxiong He, and Wen-Jing Hsu. Energy-efficient multiprocessor scheduling for flow time and makespan. *CoRR abs/1010.4110*, 2010.

[137] T Tanaka, T Toumiya, and T Suzuki. Output control by hill-climbing method for a small scale wind power generating system. *Renewable Energy*, 12(4):387–400, 1997.

[138] Maxim Teslenko and Elena Dubrova. Hermes: LUT FPGA technology mapping algorithm for area minimization with optimum depth. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 748–751. IEEE Computer Society, 2004.

[139] Kevin Oo Tinmaung, David Howland, and Russell Tessier. Power-aware FPGA logic synthesis using binary decision diagrams. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays (FPGA '07)*, pages 148–155. ACM, 2007.

[140] Tim Tuan, Sean Kao, Arif Rahman, Satyaki Das, and Steve Trimberger. A 90nm low-power FPGA for battery-powered applications. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 3–11. ACM, 2006.

[141] Tim Tuan, Arifur Rahman, Satyaki Das, Stephen Trimberger, and Sean Kao. A 90-nm low-power FPGA for battery-powered applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):296–300, 2007.

[142] International Telecommunications Union. The world in 2013: Ict facts and figures. *Telecommunication Development Bureau*, 4, 2013.

[143] Chua-Chin Wang and Cheng-Pin Kwan. Low power technology mapping by hiding high-transition paths in invisible edges for LUT-based FPGAs. In *Proceedings of 1997 IEEE International Symposium on Circuits and Systems, 1997 (ISCAS '97)*, volume 3, pages 1536 –1539, 1997.

[144] Zhi-Hong Wang, En-Cheng Liu, Jianbang Lai, and Ting-Chi Wang. Power minimization in LUT-based FPGA technology mapping. In *Proceedings of the Asia and South Pacific Design Automation Conference, 2001 (ASP-DAC 2001)*, pages 635 –640, 2001.

[145] Ingo Wegener. *The complexity of Boolean functions*. Eiley-Teubner, 1987.

[146] Xing Wei, Juanjuan Chen, Qiang Zhou, Yici Cai, Jinian Bian, and Xianlong Hong. Macromap: A technology mapping algorithm for heterogeneous FPGAs with effective area estimation. In *International Conference on Field Programmable Logic and Applications, 2008. (FPL '08)*, pages 559–562. IEEE, 2008.

[147] Edmund Weiner and John Simpson, editors. *Oxford English Dictionary*. Oxford University Press, Oxford, oed online edition, 2013.

[148] Francis Wolff, Michael Knieser, Dan Weyer, and Chris Papachristou. High-level low power FPGA design methodology. In *National Aerospace and Electronics Conference, 2000. NAECON 2000. Proceedings of the IEEE 2000*, pages 554–559. IEEE, 2000.

[149] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H Xia, and Li Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th international conference on World Wide Web*, pages 287–296. ACM, 2004.

[150] Xiao-feng Xie, Wen-jun Zhang, and Zhi-lian Yang. Overview of particle swarm optimization. *Control and Decision*, 18(2):129–134, 2003.

[151] Frances Yao, Alan Demers, and Scott Shenker. A scheduling model for reduced cpu energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS '95)*, pages 374–382. IEEE, IEEE Computer Society, 1995.

[152] Sushu Zhang, Karam Chatha, and Goran Konjevod. Approximation algorithms for power minimization of earliest deadline first and rate monotonic schedules. In *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07)*, pages 225–230. ACM, 2007.